

# **The Little Book of Algorithms**

**Version 0.3.5**

Duc-Tam Nguyen

2025-10-14

# Table of contents

<b>Content</b>	<b>20</b>
<b>The Cheatsheet</b>	<b>26</b>
Page 1. Big Picture and Complexity . . . . .	26
Page 2. Recurrences and Master Theorem . . . . .	29
Page 3. Sorting at a Glance . . . . .	32
Page 4. Searching and Selection . . . . .	36
Page 5. Core Data Structures . . . . .	40
Page 6. Graphs Quick Use . . . . .	44
Page 7. Dynamic Programming Quick Use . . . . .	48
Page 8. Mathematics for Algorithms Quick Use . . . . .	51
Page 9. Strings and Text Algorithms Quick Use . . . . .	55
Page 10. Geometry, Graphics, and Spatial Algorithms Quick Use . . . . .	60
Page 11. Systems, Databases, and Distributed Algorithms Quick Use . . . . .	65
Page 12. Algorithms for AI, ML, and Optimization Quick Use . . . . .	70
<b>The Book</b>	<b>77</b>
Chapter 1. Foundations of algorithms . . . . .	77
1. What Is an Algorithm? . . . . .	77
2. Measuring Time and Space . . . . .	79
3. Big-O, Big-Theta, Big-Omega . . . . .	81
4. Algorithmic Paradigms (Greedy, Divide and Conquer, DP) . . . . .	84
5. Recurrence Relations . . . . .	87
6. Searching Basics . . . . .	90
7. Sorting Basics . . . . .	93
8. Data Structures Overview . . . . .	96
9. Graphs and Trees Overview . . . . .	99
10. Algorithm Design Patterns . . . . .	103
Chapter 2. Sorting and Searching . . . . .	108
11. Elementary Sorting (Bubble, Insertion, Selection) . . . . .	108
12. Divide-and-Conquer Sorting (Merge, Quick, Heap) . . . . .	112
13. Counting and Distribution Sorts (Counting, Radix, Bucket) . . . . .	116
14. Hybrid Sorts (IntroSort, Timsort) . . . . .	120
15. Special Sorts (Cycle, Gnome, Comb, Pancake) . . . . .	124
16. Linear and Binary Search . . . . .	129

17. Interpolation and Exponential Search . . . . .	132
18. Selection Algorithms (Quickselect, Median of Medians) . . . . .	136
19. Range Searching and Nearest Neighbor . . . . .	140
20. Search Optimizations and Variants . . . . .	144
Chapter 3. Data Structures in Actions . . . . .	149
21. Arrays, Linked Lists, Stacks, Queues . . . . .	149
22. Hash Tables and Variants (Cuckoo, Robin Hood, Consistent) . . . . .	153
23. Heaps (Binary, Fibonacci, Pairing) . . . . .	158
24. Balanced Trees (AVL, Red-Black, Splay, Treap) . . . . .	163
25. Segment Trees and Fenwick Trees . . . . .	168
26. Disjoint Set Union (Union-Find) . . . . .	173
27. Probabilistic Data Structures (Bloom, Count-Min, HyperLogLog) . . . . .	178
28. Skip Lists and B-Trees . . . . .	183
29. Persistent and Functional Data Structures . . . . .	188
30. Advanced Trees and Range Queries . . . . .	193
Chapter 4. Graph Algorithms . . . . .	197
31. Traversals (DFS, BFS, Iterative Deepening) . . . . .	197
32. Strongly Connected Components (Tarjan, Kosaraju) . . . . .	203
33. Shortest Paths (Dijkstra, Bellman-Ford, A*, Johnson) . . . . .	209
34. Shortest Path Variants (0-1 BFS, Bidirectional, Heuristic A*) . . . . .	215
35. Minimum Spanning Trees (Kruskal, Prim, Boruvka) . . . . .	220
36. Flows (Ford-Fulkerson, Edmonds-Karp, Dinic) . . . . .	226
37. Cuts (Stoer-Wagner, Karger, Gomory-Hu) . . . . .	232
38. Matchings (Hopcroft-Karp, Hungarian, Blossom) . . . . .	238
39. Tree Algorithms (LCA, HLD, Centroid Decomposition) . . . . .	244
40. Advanced Graph Algorithms and Tricks . . . . .	250
Chapter 5. Dynamic Programming . . . . .	255
41. DP Basics and State Transitions . . . . .	255
42. Classic Problems (Knapsack, Subset Sum, Coin Change) . . . . .	260
43. Sequence Problems (LIS, LCS, Edit Distance) . . . . .	265
44. Matrix and Chain Problems . . . . .	270
45. Bitmask DP and Traveling Salesman . . . . .	275
46. Digit DP and SOS DP . . . . .	279
47. DP Optimizations (Divide & Conquer, Convex Hull Trick, Knuth) . . . . .	284
48. Tree DP and Rerooting . . . . .	289
49. DP Reconstruction and Traceback . . . . .	294
50. Meta-DP and Optimization Templates . . . . .	300
Chapter 6. Strings and Text Algorithms . . . . .	306
51. Number Theory (GCD, Modular Arithmetic, CRT) . . . . .	306
52. Primality and Factorization (Miller-Rabin, Pollard Rho) . . . . .	312
53. Combinatorics (Permutations, Combinations, Subsets) . . . . .	316
54. Probability and Randomized Algorithms . . . . .	320
55. Sieve Methods and Modular Math . . . . .	325

56. Linear Algebra (Gaussian Elimination, LU, SVD) . . . . .	330
57. FFT and NTT (Fast Transforms) . . . . .	335
58. Numerical Methods (Newton, Simpson, Runge-Kutta) . . . . .	340
59. Mathematical Optimization (Simplex, Gradient, Convex) . . . . .	345
60. Algebraic Tricks and Transform Techniques . . . . .	350
Chapter 7. Strings and Text Algorithms . . . . .	354
61. String Matching (KMP, Z, Rabin-Karp, Boyer-Moore) . . . . .	354
62. Multi-Pattern Search (Aho-Corasick) . . . . .	358
63. Suffix Structures (Suffix Array, Suffix Tree, LCP) . . . . .	363
64. Palindromes and Periodicity (Manacher) . . . . .	368
65. Edit Distance and Alignment . . . . .	372
66. Compression (Huffman, Arithmetic, LZ77, BWT) . . . . .	376
67. Cryptographic Hashes and Checksums . . . . .	380
68. Approximate and Streaming Matching . . . . .	384
69. Bioinformatics Alignment (Needleman-Wunsch, Smith-Waterman) . . . . .	387
70. Text Indexing and Search Structures . . . . .	391
Chapter 8. Geometry, Graphics, and Spatial Algorithms . . . . .	395
71. Convex Hull (Graham, Andrew, Chan) . . . . .	395
72. Closest Pair and Segment Intersection . . . . .	399
73. Line Sweep and Plane Sweep Algorithms . . . . .	403
74. Delaunay and Voronoi Diagrams . . . . .	407
75. Point in Polygon and Polygon Triangulation . . . . .	410
76. Spatial Data Structures (KD, R-tree) . . . . .	414
77. Rasterization and Scanline Techniques . . . . .	418
78. Computer Vision (Canny, Hough, SIFT) . . . . .	421
79. Pathfinding in Space (A*, RRT, PRM) . . . . .	426
80. Computational Geometry Variants and Applications . . . . .	430
Chapter 9. Systems, Databases, and Distributed Algorithms . . . . .	434
81. Concurrency Control (2PL, MVCC, OCC) . . . . .	434
82. Logging, Recovery, and Commit Protocols . . . . .	438
83. Scheduling (Round Robin, EDF, Rate-Monotonic) . . . . .	443
84. Caching and Replacement (LRU, LFU, CLOCK) . . . . .	447
85. Networking (Routing, Congestion Control) . . . . .	450
86. Distributed Consensus (Paxos, Raft, PBFT) . . . . .	454
87. Load Balancing and Rate Limiting . . . . .	458
88. Search and Indexing (Inverted, BM25, WAND) . . . . .	462
89. Compression and Encoding in Systems . . . . .	467
90. Fault Tolerance and Replication . . . . .	471
Chapter 10. AI, ML and Optimization . . . . .	475
91. Classical ML (k-means, Naive Bayes, SVM, Decision Trees) . . . . .	475
92. Ensemble Methods (Bagging, Boosting, Random Forests) . . . . .	480
93. Gradient Methods (SGD, Adam, RMSProp) . . . . .	485
94. Deep Learning (Backpropagation, Dropout, Normalization) . . . . .	489

95. Sequence Models (Viterbi, Beam Search, CTC) . . . . .	493
96. Metaheuristics (GA, SA, PSO, ACO) . . . . .	497
97. Reinforcement Learning (Q-learning, Policy Gradients) . . . . .	501
98. Approximation and Online Algorithms . . . . .	504
99. Fairness, Causal Inference, and Robust Optimization . . . . .	507
100. AI Planning, Search, and Learning Systems . . . . .	511
<b>The Plan</b>	<b>515</b>
Chapter 1. Foundations of Algorithms . . . . .	515
Chapter 2. Sorting and Searching . . . . .	519
Chapter 3. Data Structures in Action . . . . .	523
Chapter 4. Graph Algorithms . . . . .	526
Chapter 5. Dynamic Programming . . . . .	530
Chapter 6. Mathematics for Algorithms . . . . .	534
Chapter 7. Strings and Text Algorithms . . . . .	538
Chapter 8. Geometry, Graphics, and Spatial Algorithms . . . . .	542
Chapter 9. Systems, Databases, and Distributed Algorithms . . . . .	546
Chapter 10. AI, ML, and Optimization . . . . .	550
<b>Chapter 1. Foundations of Algorithms</b>	<b>555</b>
Section 1. What is an algorithms? . . . . .	555
1 Euclid's GCD . . . . .	555
2 Sieve of Eratosthenes . . . . .	557
3 Linear Step Trace . . . . .	560
4 Algorithm Flow Diagram Builder . . . . .	562
5 Long Division . . . . .	565
6 Modular Addition . . . . .	568
7 Base Conversion . . . . .	570
8 Factorial Computation . . . . .	573
9 Iterative Process Tracer . . . . .	576
10 Tower of Hanoi . . . . .	579
Section 2. Measuring time and space . . . . .	582
11 Counting Operations . . . . .	582
12 Loop Analysis . . . . .	584
13 Recurrence Expansion . . . . .	587
14 Amortized Analysis . . . . .	590
15 Space Counting . . . . .	593
16 Memory Footprint Estimator . . . . .	596
17 Time Complexity Table . . . . .	598
18 Space–Time Tradeoff Explorer . . . . .	600
19 Profiling Algorithm . . . . .	603
20 Benchmarking Framework . . . . .	606

Section 3. Big-O, Big-Theta, Big-Omega . . . . .	609
21 Growth Rate Comparator . . . . .	609
22 Dominant Term Extractor . . . . .	612
23 Limit-Based Complexity Test . . . . .	615
24 Summation Simplifier . . . . .	617
25 Recurrence Tree Method . . . . .	620
26 Master Theorem Evaluator . . . . .	623
27 Big-Theta Proof Builder . . . . .	627
28 Big-Omega Case Finder . . . . .	629
29 Empirical Complexity Estimator . . . . .	632
30 Complexity Class Identifier . . . . .	635
Section 4. Algorithm Paradigms . . . . .	638
31 Greedy Coin Example . . . . .	638
32 Greedy Template Simulator . . . . .	640
33 Divide & Conquer Skeleton . . . . .	643
34 Backtracking Maze Solver . . . . .	646
35 Karatsuba Multiplication . . . . .	649
36 DP State Diagram Example . . . . .	652
37 DP Table Visualization . . . . .	655
38 Recursive Subproblem Tree Demo . . . . .	658
39 Greedy Choice Visualization . . . . .	661
40 Amortized Merge Demo . . . . .	664
Section 5. Recurrence Relations . . . . .	667
41 Linear Recurrence Solver . . . . .	667
42 Master Theorem . . . . .	670
The Three Cases . . . . .	671
43 Substitution Method . . . . .	674
44 Iteration Method . . . . .	677
45 Generating Function Method . . . . .	681
46 Matrix Exponentiation . . . . .	685
47 Recurrence to DP Table . . . . .	689
48 Divide & Combine Template . . . . .	692
Generic Template (Pseudocode) . . . . .	693
49 Memoized Recursive Solver . . . . .	695
50 Characteristic Polynomial Solver . . . . .	698
Section 6. Searching basics . . . . .	702
51 Search Space Visualizer . . . . .	702
52 Decision Tree Depth Estimator . . . . .	705
53 Comparison Counter . . . . .	708
54 Early Termination Heuristic . . . . .	712
55 Sentinel Technique . . . . .	714
56 Binary Predicate Tester . . . . .	717
57 Range Test Function . . . . .	720

58 Search Invariant Checker . . . . .	723
59 Probe Counter . . . . .	726
60 Cost Curve Plotter . . . . .	728
Section 7. Sorting basics . . . . .	731
61 Swap Counter . . . . .	731
62 Inversion Counter . . . . .	734
63 Stability Checker . . . . .	737
64 Comparison Network Visualizer . . . . .	740
65 Adaptive Sort Detector . . . . .	742
66 Sorting Invariant Checker . . . . .	745
67 Distribution Histogram Sort Demo . . . . .	748
68 Key Extraction Function . . . . .	750
69 Partially Ordered Set Builder . . . . .	753
70 Complexity Comparator . . . . .	756
Section 8. Data Structure Overview . . . . .	759
71 Stack Simulation . . . . .	759
72 Queue Simulation . . . . .	762
73 Linked List Builder . . . . .	765
74 Array Index Visualizer . . . . .	768
75 Hash Function Mapper . . . . .	771
76 Binary Tree Builder . . . . .	774
77 Heap Structure Demo . . . . .	777
78 Union-Find Concept . . . . .	780
79 Graph Representation Demo . . . . .	783
80 Trie Structure Visualizer . . . . .	787
Section 9. Graphs and Trees overview . . . . .	791
81 Graph Model Constructor . . . . .	791
82 Adjacency Matrix Builder . . . . .	794
83 Adjacency List Builder . . . . .	797
84 Degree Counter . . . . .	800
85 Path Existence Tester . . . . .	803
86 Tree Validator . . . . .	806
86 Tree Validator . . . . .	809
87 Rooted Tree Builder . . . . .	813
88 Traversal Order Visualizer . . . . .	816
89 Edge Classifier . . . . .	820
90 Connectivity Checker . . . . .	824
Section 10. Algorithm Design Patterns . . . . .	828
91 Brute Force Pattern . . . . .	828
92 Greedy Pattern . . . . .	830
93 Divide and Conquer Pattern . . . . .	833
94 Dynamic Programming Pattern . . . . .	836
95 Backtracking Pattern . . . . .	839

96 Branch and Bound . . . . .	842
97 Randomized Pattern . . . . .	845
98 Approximation Pattern . . . . .	848
99 Online Algorithm Pattern . . . . .	850
100 Hybrid Strategy Pattern . . . . .	853
<b>Chapter 2. Sorting and searching</b>	<b>857</b>
Section 11. Elementary sorting . . . . .	857
101 Bubble Sort . . . . .	857
102 Improved Bubble Sort . . . . .	860
103 Cocktail Shaker Sort . . . . .	864
104 Selection Sort . . . . .	868
105 Double Selection Sort . . . . .	871
106 Insertion Sort . . . . .	875
107 Binary Insertion Sort . . . . .	879
108 Gnome Sort . . . . .	883
109 Odd-Even Sort . . . . .	887
110 Stooge Sort . . . . .	891
Section 12. Divide and conquer sorting . . . . .	895
111 Merge Sort . . . . .	895
112 Iterative Merge Sort . . . . .	899
113 Quick Sort . . . . .	904
114 Hoare Partition Scheme . . . . .	909
115 Lomuto Partition Scheme . . . . .	913
116 Randomized Quick Sort . . . . .	918
117 Heap Sort . . . . .	923
118 3-Way Quick Sort . . . . .	927
119 External Merge Sort . . . . .	932
120 Parallel Merge Sort . . . . .	936
Section 13. Counting and distribution sorts . . . . .	942
121 Counting Sort . . . . .	942
122 Stable Counting Sort . . . . .	946
123 Radix Sort (LSD) . . . . .	951
124 Radix Sort (MSD) . . . . .	956
125 Bucket Sort . . . . .	961
126 Pigeonhole Sort . . . . .	965
127 Flash Sort . . . . .	970
128 Postman Sort . . . . .	976
129 Address Calculation Sort . . . . .	980
130 Spread Sort . . . . .	985
Section 14. Hybrid sorts . . . . .	990
131 IntroSort . . . . .	990
132 TimSort . . . . .	996

133 Dual-Pivot QuickSort . . . . .	1002
134 SmoothSort . . . . .	1008
135 Block Merge Sort . . . . .	1012
136 Adaptive Merge Sort . . . . .	1017
137 PDQSort (Pattern-Defeating QuickSort) . . . . .	1023
138 WikiSort . . . . .	1028
139 GrailSort . . . . .	1033
140 Adaptive Hybrid Sort . . . . .	1038
Section 15. Special sorts . . . . .	1045
141 Cycle Sort . . . . .	1045
142 Comb Sort . . . . .	1050
143 Gnome Sort . . . . .	1054
144 Cocktail Sort . . . . .	1058
145 Pancake Sort . . . . .	1064
146 Bitonic Sort . . . . .	1068
147 Odd-Even Merge Sort . . . . .	1072
148 Sleep Sort . . . . .	1077
149 Bead Sort . . . . .	1081
150 Bogo Sort . . . . .	1086
Section 16. Linear and Binary Search . . . . .	1091
151 Linear Search . . . . .	1091
152 Linear Search (Sentinel) . . . . .	1095
153 Binary Search (Iterative) . . . . .	1099
154 Binary Search (Recursive) . . . . .	1103
155 Binary Search (Lower Bound) . . . . .	1107
156 Binary Search (Upper Bound) . . . . .	1111
157 Exponential Search . . . . .	1115
158 Jump Search . . . . .	1120
159 Fibonacci Search . . . . .	1124
160 Uniform Binary Search . . . . .	1129
Section 17. Interpolation and exponential search . . . . .	1134
161 Interpolation Search . . . . .	1134
162 Recursive Interpolation Search . . . . .	1139
163 Exponential Search . . . . .	1143
164 Doubling Search . . . . .	1147
165 Galloping Search . . . . .	1152
166 Unbounded Binary Search . . . . .	1157
167 Root-Finding Bisection . . . . .	1161
168 Golden Section Search . . . . .	1166
169 Fibonacci Search (Optimum) . . . . .	1170
170 Jump + Binary Hybrid . . . . .	1175
Section 18. Selection Algorithms . . . . .	1180
171 Quickselect . . . . .	1180

172 Median of Medians . . . . .	1185
173 Randomized Select . . . . .	1190
174 Binary Search on Answer . . . . .	1195
175 Order Statistics Tree . . . . .	1199
176 Tournament Tree Selection . . . . .	1204
177 Heap Select (Min-Heap) . . . . .	1208
178 Partial QuickSort . . . . .	1212
179 BFPRST Algorithm (Median of Medians Selection) . . . . .	1217
180 Kth Largest Stream . . . . .	1221
Section 19. Range Search and Nearest Neighbor . . . . .	1226
181 Binary Search Range . . . . .	1226
182 Segment Tree Query . . . . .	1231
183 Fenwick Tree Query . . . . .	1235
184 Interval Tree Search . . . . .	1239
Key Intuition . . . . .	1244
185 KD-Tree Search . . . . .	1245
186 R-Tree Query . . . . .	1249
187 Range Minimum Query (RMQ), Sparse Table Approach . . . . .	1253
188 Mo's Algorithm . . . . .	1258
189 Sweep Line Range Search . . . . .	1263
190 Ball Tree Nearest Neighbor . . . . .	1268
Section 20. Search Optimizations and variants . . . . .	1272
191 Binary Search with Tolerance . . . . .	1272
Example . . . . .	1273
192 Ternary Search . . . . .	1276
Example (Unimodal Function) . . . . .	1277
193 Hash-Based Search . . . . .	1280
Example (Simple Lookup) . . . . .	1280
194 Bloom Filter Lookup . . . . .	1284
Example (Cache Lookup) . . . . .	1284
195 Cuckoo Hash Search . . . . .	1286
Example (Small Table) . . . . .	1287
196 Robin Hood Hashing . . . . .	1290
Example (Collision Handling) . . . . .	1291
197 Jump Consistent Hashing . . . . .	1295
Example (Adding Buckets) . . . . .	1295
198 Prefix Search in Trie . . . . .	1298
Example (Words: app, apple, bat) . . . . .	1299
199 Pattern Search in Suffix Array . . . . .	1302
Example (Text = "banana") . . . . .	1303
200 Search in Infinite Array . . . . .	1306
Example . . . . .	1307

<b>Chapter 3. Data Structure in Action</b>	<b>1311</b>
Section 21. Arrays, linked lists, stacks, queues . . . . .	1311
201 Dynamic Array Resize . . . . .	1311
202 Circular Array Implementation . . . . .	1316
203 Singly Linked List Insert/Delete . . . . .	1320
204 Doubly Linked List Insert/Delete . . . . .	1325
205 Stack Push/Pop . . . . .	1330
206 Queue Enqueue/Dequeue . . . . .	1334
207 Deque Implementation . . . . .	1338
208 Circular Queue . . . . .	1342
209 Stack via Queue . . . . .	1347
210 Queue via Stack . . . . .	1352
Section 22. Hash Tables and Variants . . . . .	1357
211 Hash Table Insertion . . . . .	1357
212 Linear Probing . . . . .	1362
213 Quadratic Probing . . . . .	1367
214 Double Hashing . . . . .	1372
215 Cuckoo Hashing . . . . .	1377
216 Robin Hood Hashing . . . . .	1382
217 Chained Hash Table . . . . .	1387
218 Perfect Hashing . . . . .	1392
219 Consistent Hashing . . . . .	1397
220 Dynamic Rehashing . . . . .	1400
Section 23. Heaps . . . . .	1405
221 Binary Heap Insert . . . . .	1405
222 Binary Heap Delete . . . . .	1409
223 Build Heap (Heapify) . . . . .	1414
224 Heap Sort . . . . .	1418
225 Min Heap Implementation . . . . .	1422
226 Max Heap Implementation . . . . .	1427
227 Fibonacci Heap Insert/Delete . . . . .	1433
228 Pairing Heap Merge . . . . .	1438
229 Binomial Heap Merge . . . . .	1444
230 Leftist Heap Merge . . . . .	1450
Section 24. Balanced Trees . . . . .	1456
231 AVL Tree Insert . . . . .	1456
232 AVL Tree Delete . . . . .	1462
233 Red-Black Tree Insert . . . . .	1470
234 Red-Black Tree Delete . . . . .	1478
235 Splay Tree Access . . . . .	1483
236 Treap Insert . . . . .	1490
237 Treap Delete . . . . .	1495
238 Weight Balanced Tree . . . . .	1502

239 Scapegoat Tree Rebuild . . . . .	1508
240 AA Tree . . . . .	1515
Section 25. Segment Trees and Fenwick Trees . . . . .	1521
241 Build Segment Tree . . . . .	1521
242 Range Sum Query . . . . .	1525
243 Range Update (Lazy Propagation Technique) . . . . .	1529
244 Point Update . . . . .	1535
245 Fenwick Tree Build . . . . .	1540
246 Fenwick Update . . . . .	1544
247 Fenwick Query . . . . .	1547
248 Segment Tree Merge . . . . .	1551
249 Persistent Segment Tree . . . . .	1555
250 2D Segment Tree . . . . .	1560
Section 26. Disjoint Set Union . . . . .	1565
251 Make-Set . . . . .	1565
252 Find . . . . .	1569
253 Union . . . . .	1573
254 Union by Rank . . . . .	1579
255 Path Compression . . . . .	1584
256 DSU with Rollback . . . . .	1589
257 DSU on Tree . . . . .	1594
258 Kruskal's MST (Using DSU) . . . . .	1600
259 Connected Components (Using DSU) . . . . .	1605
260 Offline Query DSU . . . . .	1610
Section 27. Probabilistic Data Structure . . . . .	1615
261 Bloom Filter Insert . . . . .	1615
262 Bloom Filter Query . . . . .	1619
263 Counting Bloom Filter . . . . .	1624
264 Cuckoo Filter . . . . .	1628
265 Count-Min Sketch . . . . .	1633
266 HyperLogLog . . . . .	1637
267 Flajolet–Martin Algorithm . . . . .	1641
268 MinHash . . . . .	1646
269 Reservoir Sampling . . . . .	1650
Mathematical Intuition . . . . .	1652
270 Skip Bloom Filter . . . . .	1655
Section 28. Skip Lists and B-Trees . . . . .	1660
271 Skip List Insert . . . . .	1660
272 Skip List Delete . . . . .	1664
273 Skip List Search . . . . .	1667
274 B-Tree Insert . . . . .	1671
275 B-Tree Delete . . . . .	1675
276 B+ Tree Search . . . . .	1680

278 B* Tree . . . . .	1684
279 Adaptive Radix Tree . . . . .	1687
280 Trie Compression . . . . .	1691
Section 29. Persistent and Functional Data Structures . . . . .	1694
281 Persistent Stack . . . . .	1694
282 Persistent Array . . . . .	1699
283 Persistent Segment Tree . . . . .	1703
284 Persistent Linked List . . . . .	1707
286 Finger Tree . . . . .	1712
287 Zipper Structure . . . . .	1716
289 Trie with Versioning . . . . .	1720
Test Case . . . . .	1723
290 Persistent Union-Find . . . . .	1724
Section 30. Advanced Trees and Range Queries . . . . .	1727
291 Sparse Table Build . . . . .	1727
292 Cartesian Tree . . . . .	1731
293 Segment Tree Beats . . . . .	1734
294 Merge Sort Tree . . . . .	1738
295 Wavelet Tree . . . . .	1742
Core Operations . . . . .	1744
296 KD-Tree . . . . .	1747
How Nearest Neighbor Works . . . . .	1749
297 Range Tree . . . . .	1751
Step-by-Step Table (2D Query) . . . . .	1753
298 Fenwick 2D Tree . . . . .	1755
Step-by-Step Update Example . . . . .	1756
299 Treap Split/Merge . . . . .	1758
Step-by-Step Split Example . . . . .	1760
300 Mo's Algorithm on Tree . . . . .	1762
Step-by-Step Example . . . . .	1763
<b>Chapter 4. Graph Algorithms</b> . . . . .	<b>1767</b>
Section 31. Traversals . . . . .	1767
301 Depth-First Search (Recursive) . . . . .	1767
302 Depth-First Search (Iterative) . . . . .	1770
303 Breadth-First Search (Queue) . . . . .	1774
304 Iterative Deepening DFS . . . . .	1778
305 Bidirectional BFS . . . . .	1782
306 DFS on Grid . . . . .	1786
307 BFS on Grid . . . . .	1790
308 Multi-Source BFS . . . . .	1795
309 Topological Sort (DFS-based) . . . . .	1799
310 Topological Sort (Kahn's Algorithm) . . . . .	1803

Section 32. Strongly Connected Components . . . . .	1807
311 Kosaraju's Algorithm . . . . .	1807
312 Tarjan's Algorithm . . . . .	1812
313 Gabow's Algorithm . . . . .	1817
314 SCC DAG Construction . . . . .	1822
315 SCC Online Merge . . . . .	1826
316 Component Label Propagation . . . . .	1829
317 Path-Based SCC . . . . .	1833
318 Kosaraju Parallel Version . . . . .	1839
319 Dynamic SCC Maintenance . . . . .	1843
320 SCC for Weighted Graph . . . . .	1847
Section 33. Shortest Paths . . . . .	1851
321 Dijkstra (Binary Heap) . . . . .	1851
322 Dijkstra (Fibonacci Heap) . . . . .	1855
323 Bellman–Ford . . . . .	1858
324 SPFA (Queue Optimization) . . . . .	1862
325 A* Search . . . . .	1867
326 Floyd–Warshall . . . . .	1871
327 Johnson's Algorithm . . . . .	1874
328 0–1 BFS . . . . .	1879
329 Dial's Algorithm . . . . .	1883
330 Multi-Source Dijkstra . . . . .	1886
Section 34. Shortest Path Variants . . . . .	1890
331 0–1 BFS . . . . .	1890
332 Bidirectional Dijkstra . . . . .	1893
333 A* with Euclidean Heuristic . . . . .	1897
334 ALT Algorithm (A* Landmarks + Triangle Inequality) . . . . .	1900
335 Contraction Hierarchies . . . . .	1904
336 CH Query Algorithm (Shortcut-Based Routing) . . . . .	1908
337 Bellman–Ford Queue Variant (Early Termination SPFA) . . . . .	1912
338 Dijkstra with Early Stop . . . . .	1916
339 Goal-Directed Search . . . . .	1920
340 Yen's K Shortest Paths . . . . .	1924
Section 35. Minimum Spanning Trees . . . . .	1928
341 Kruskal's Algorithm . . . . .	1928
342 Prim's Algorithm (Heap) . . . . .	1932
343 Prim's Algorithm (Adjacency Matrix) . . . . .	1935
344 Borůvka's Algorithm . . . . .	1939
345 Reverse-Delete MST . . . . .	1943
346 MST via Dijkstra Trick . . . . .	1946
347 Dynamic MST Maintenance . . . . .	1949
348 Minimum Bottleneck Spanning Tree . . . . .	1953
349 Manhattan MST . . . . .	1956

350 Euclidean MST (Kruskal + Geometry) . . . . .	1959
Section 36. Flows . . . . .	1962
351 Ford–Fulkerson . . . . .	1962
352 Edmonds–Karp . . . . .	1966
353 Dinic’s Algorithm . . . . .	1970
354 Push–Relabel . . . . .	1974
355 Capacity Scaling . . . . .	1979
356 Cost Scaling . . . . .	1983
357 Min-Cost Max-Flow (Bellman-Ford) . . . . .	1986
358 Min-Cost Max-Flow (SPFA) . . . . .	1989
359 Circulation with Demands . . . . .	1993
360 Successive Shortest Path . . . . .	1996
Section 37. Cuts . . . . .	2000
361 Stoer–Wagner Minimum Cut . . . . .	2000
362 Karger’s Randomized Cut . . . . .	2003
363 Karger–Stein Minimum Cut . . . . .	2006
364 Gomory–Hu Tree . . . . .	2010
365 Max-Flow Min-Cut Theorem . . . . .	2013
366 Stoer–Wagner Repeated Phase . . . . .	2017
367 Dynamic Min Cut . . . . .	2020
368 Minimum s–t Cut (Edmonds–Karp) . . . . .	2024
369 Approximate Min Cut . . . . .	2027
370 Min k-Cut . . . . .	2030
Section 38. Matchings . . . . .	2034
371 Bipartite Matching (DFS) . . . . .	2034
372 Hopcroft–Karp . . . . .	2037
373 Hungarian Algorithm . . . . .	2041
374 Kuhn–Munkres (Max-Weight Matching) . . . . .	2044
375 Blossom Algorithm . . . . .	2047
376 Edmonds’ Blossom Shrinking . . . . .	2050
377 Greedy Matching . . . . .	2054
378 Stable Marriage (Gale–Shapley) . . . . .	2056
379 Weighted b-Matching . . . . .	2060
380 Maximal Matching . . . . .	2063
Section 39. Tree Algorithms . . . . .	2066
381 Euler Tour LCA . . . . .	2066
382 Binary Lifting LCA . . . . .	2070
383 Tarjan’s LCA (Offline DSU) . . . . .	2074
384 Heavy-Light Decomposition . . . . .	2077
385 Centroid Decomposition . . . . .	2081
386 Tree Diameter (DFS Twice) . . . . .	2085
387 Tree DP (Subtree-Based Optimization) . . . . .	2087
388 Rerooting DP (Compute All Roots’ Answers) . . . . .	2090

389 Binary Search on Tree (Edge Weight Constraints) . . . . .	2093
390 Virtual Tree (Query Subset Construction) . . . . .	2096
<b>Section 40. Advanced Graph Algorithms and Tricks . . . . .</b>	<b>2099</b>
391 Topological DP (Dynamic Programming on DAG) . . . . .	2099
392 SCC Condensed Graph DP (Dynamic Programming on Meta-Graph) . . . . .	2102
393 Eulerian Path . . . . .	2105
394 Hamiltonian Path . . . . .	2109
395 Chinese Postman Problem (Route Inspection) . . . . .	2112
396 Hierholzer's Algorithm . . . . .	2115
397 Johnson's Cycle Finding Algorithm . . . . .	2119
398 Transitive Closure (Floyd–Warshall) . . . . .	2122
399 Graph Coloring (Backtracking) . . . . .	2125
400 Articulation Points & Bridges . . . . .	2129
<b>Chapter 5. Dynamic Programming . . . . .</b>	<b>2133</b>
<b>Section 41. DP basic and state transitions . . . . .</b>	<b>2134</b>
401 Fibonacci DP . . . . .	2134
402 Climbing Stairs . . . . .	2137
403 Grid Paths . . . . .	2140
404 Min Cost Path . . . . .	2144
405 Coin Change (Count Ways) . . . . .	2148
406 Coin Change (Min Coins) . . . . .	2151
407 Knapsack 0/1 . . . . .	2155
408 Knapsack Unbounded . . . . .	2158
409 Longest Increasing Subsequence (DP) . . . . .	2162
410 Edit Distance (Levenshtein) . . . . .	2166
<b>Section 42. Classic Problems . . . . .</b>	<b>2171</b>
411 0/1 Knapsack . . . . .	2171
412 Subset Sum . . . . .	2175
413 Equal Partition . . . . .	2178
414 Count of Subsets with Sum . . . . .	2182
415 Target Sum . . . . .	2186
416 Unbounded Knapsack . . . . .	2191
417 Fractional Knapsack . . . . .	2195
418 Coin Change (Min Coins) . . . . .	2200
419 Coin Change (Count Ways) . . . . .	2204
420 Multi-Dimensional Knapsack . . . . .	2208
<b>Section 43. Sequence Problems . . . . .</b>	<b>2213</b>
421 Longest Increasing Subsequence ( $O(n^2)$ DP) . . . . .	2213
422 LIS (Patience Sorting) – $O(n \log n)$ Optimized . . . . .	2216

423 Longest Common Subsequence (LCS) . . . . .	2221
424 Edit Distance (Levenshtein) . . . . .	2224
425 Longest Palindromic Subsequence . . . . .	2229
426 Shortest Common Supersequence (SCS) . . . . .	2233
427 Longest Repeated Subsequence . . . . .	2237
428 String Interleaving . . . . .	2241
429 Sequence Alignment (Bioinformatics) . . . . .	2245
430 Diff Algorithm (Myers / DP) . . . . .	2250
<b>Section 44. Matrix and Chain Problems</b>	<b>2256</b>
431 Matrix Chain Multiplication . . . . .	2256
432 Boolean Parenthesization . . . . .	2261
433 Burst Balloons . . . . .	2266
434 Optimal BST . . . . .	2269
435 Polygon Triangulation . . . . .	2275
436 Matrix Path Sum . . . . .	2279
437 Largest Square Submatrix . . . . .	2282
438 Max Rectangle in Binary Matrix . . . . .	2286
439 Submatrix Sum Queries . . . . .	2290
440 Palindrome Partitioning . . . . .	2294
<b>Section 45. Bitmask DP and Traveling Salesman</b>	<b>2299</b>
441 Traveling Salesman Problem (TSP), Bitmask DP (Held–Karp) . . . . .	2299
442 Subset DP (Over Subsets of States) . . . . .	2303
443 Hamiltonian Path DP (State Compression) . . . . .	2307
444 Assignment Problem DP (Mask over Tasks) . . . . .	2311
445 Partition into Two Sets (Balanced Load) . . . . .	2315
446 Count Hamiltonian Cycles (Bitmask Enumeration) . . . . .	2319
447 Steiner Tree DP . . . . .	2323
448 SOS DP (Sum Over Subsets) . . . . .	2328
449 Bitmask Knapsack (State Compression) . . . . .	2331
450 Bitmask Independent Set (Graph Subset Optimization) . . . . .	2335
<b>Section 46. Digit DP and SOS DP</b>	<b>2340</b>
451 Count Numbers with Property (Digit DP) . . . . .	2340
452 Count Without Adjacent Duplicates . . . . .	2344
453 Sum of Digits in Range . . . . .	2349
454 Count with Mod Condition (Digit Sum mod M) . . . . .	2355
455 Count of Increasing Digits . . . . .	2359
456 Count with Forbidden Digits . . . . .	2364
457 SOS DP Subset Sum . . . . .	2369
458 SOS DP Superset Sum . . . . .	2372
459 XOR Basis DP . . . . .	2376

460 Digit DP for Palindromes . . . . .	2379
<b>Section 47. DP Optimizations</b>	<b>2385</b>
461 Divide & Conquer DP (Monotone Optimization) . . . . .	2385
462 Knuth Optimization . . . . .	2389
463 Convex Hull Trick (CHT) . . . . .	2393
464 Li Chao Tree . . . . .	2397
465 Slope Trick . . . . .	2402
466 Monotonic Queue Optimization . . . . .	2406
467 Bitset DP . . . . .	2410
468 Offline DP Queries . . . . .	2414
469 DP + Segment Tree . . . . .	2418
470 Divide & Conquer Knapsack . . . . .	2423
<b>Section 48. Tree DP and Rerooting</b>	<b>2427</b>
471 Subtree Sum DP . . . . .	2427
472 Diameter DP . . . . .	2431
473 Independent Set DP . . . . .	2435
474 Vertex Cover DP . . . . .	2439
475 Path Counting DP . . . . .	2444
476 DP on Rooted Tree . . . . .	2448
477 Rerooting Technique . . . . .	2452
478 Distance Sum Rerooting . . . . .	2458
479 Tree Coloring DP . . . . .	2463
480 Binary Search on Tree DP . . . . .	2467
<b>Section 49. DP Reconstruction and Traceback</b>	<b>2472</b>
481 Reconstruct LCS . . . . .	2472
482 Reconstruct LIS . . . . .	2475
483 Reconstruct Knapsack . . . . .	2479
484 Edit Distance Alignment . . . . .	2483
485 Matrix Chain Parentheses . . . . .	2488
486 Coin Change Reconstruction . . . . .	2492
487 Path Reconstruction DP . . . . .	2496
488 Sequence Reconstruction . . . . .	2501
489 Multi-Choice Reconstruction . . . . .	2505
490 Traceback Visualization . . . . .	2508
<b>Section 50. Meta-DP and Optimization Templates</b>	<b>2512</b>
491 State Compression Template . . . . .	2512
492 Transition Optimization Template . . . . .	2515
493 Space Optimization Template . . . . .	2519
494 Multi-Dimensional DP Template . . . . .	2523

495 Decision Monotonicity . . . . .	2526
496 Monge Array Optimization . . . . .	2530
497 Divide & Conquer Template . . . . .	2533
498 Rerooting Template . . . . .	2537
499 Iterative DP Pattern . . . . .	2542
500 Memoization Template . . . . .	2545

# Content

*A Friendly Guide from Numbers to Neural Networks*

- [Download PDF](#) - print-ready
- [Download EPUB](#) - e-reader friendly
- [View LaTex](#) - `.tex` source
- [Source code \(Github\)](#) - Markdown source
- [Read on GitHub Pages](#) - view online

Licensed under **CC BY-NC-SA 4.0**.

## **Chapter 1. Foundations of Algorithms**

- 1. What Is an Algorithm?
- 2. Measuring Time and Space
- 3. Big-O, Big-Theta, Big-Omega
- 4. Algorithmic Paradigms (Greedy, Divide and Conquer, DP)
- 5. Recurrence Relations
- 6. Searching Basics
- 7. Sorting Basics
- 8. Data Structures Overview
- 9. Graphs and Trees Overview
- 10. Algorithm Design Patterns

## **Chapter 2. Sorting and Searching**

- 11. Elementary Sorting (Bubble, Insertion, Selection)
- 12. Divide-and-Conquer Sorting (Merge, Quick, Heap)
- 13. Counting and Distribution Sorts (Counting, Radix, Bucket)
- 14. Hybrid Sorts (IntroSort, Timsort)
- 15. Special Sorts (Cycle, Gnome, Comb, Pancake)
- 16. Linear and Binary Search
- 17. Interpolation and Exponential Search
- 18. Selection Algorithms (Quickselect, Median of Medians)
- 19. Range Searching and Nearest Neighbor
- 20. Search Optimizations and Variants

## **Chapter 3. Data Structures in Action**

- 21. Arrays, Linked Lists, Stacks, Queues
- 22. Hash Tables and Variants (Cuckoo, Robin Hood, Consistent)
- 23. Heaps (Binary, Fibonacci, Pairing)
- 24. Balanced Trees (AVL, Red-Black, Splay, Treap)
- 25. Segment Trees and Fenwick Trees
- 26. Disjoint Set Union (Union-Find)
- 27. Probabilistic Data Structures (Bloom, Count-Min, HyperLogLog)
- 28. Skip Lists and B-Trees
- 29. Persistent and Functional Data Structures

- 30. Advanced Trees and Range Queries

## **Chapter 4. Graph Algorithms**

- 31. Traversals (DFS, BFS, Iterative Deepening)
- 32. Strongly Connected Components (Tarjan, Kosaraju)
- 33. Shortest Paths (Dijkstra, Bellman-Ford, A\*, Johnson)
- 34. Shortest Path Variants (0–1 BFS, Bidirectional, Heuristic A\*)
- 35. Minimum Spanning Trees (Kruskal, Prim, Borůvka)
- 36. Flows (Ford–Fulkerson, Edmonds–Karp, Dinic)
- 37. Cuts (Stoer–Wagner, Karger, Gomory–Hu)
- 38. Matchings (Hopcroft–Karp, Hungarian, Blossom)
- 39. Tree Algorithms (LCA, HLD, Centroid Decomposition)
- 40. Advanced Graph Algorithms and Tricks

## **Chapter 5. Dynamic Programming**

- 41. DP Basics and State Transitions
- 42. Classic Problems (Knapsack, Subset Sum, Coin Change)
- 43. Sequence Problems (LIS, LCS, Edit Distance)
- 44. Matrix and Chain Problems
- 45. Bitmask DP and Traveling Salesman
- 46. Digit DP and SOS DP
- 47. DP Optimizations (Divide & Conquer, Convex Hull Trick, Knuth)

- 48. Tree DP and Rerooting
- 49. DP Reconstruction and Traceback
- 50. Meta-DP and Optimization Templates

## **Chapter 6. Mathematics for Algorithms**

- 51. Number Theory (GCD, Modular Arithmetic, CRT)
- 52. Primality and Factorization (Miller–Rabin, Pollard Rho)
- 53. Combinatorics (Permutations, Combinations, Subsets)
- 54. Probability and Randomized Algorithms
- 55. Sieve Methods and Modular Math
- 56. Linear Algebra (Gaussian Elimination, LU, SVD)
- 57. FFT and NTT (Fast Transforms)
- 58. Numerical Methods (Newton, Simpson, Runge–Kutta)
- 59. Mathematical Optimization (Simplex, Gradient, Convex)
- 60. Algebraic Tricks and Transform Techniques

## **Chapter 7. Strings and Text Algorithms**

- 61. String Matching (KMP, Z, Rabin–Karp, Boyer–Moore)
- 62. Multi-Pattern Search (Aho–Corasick)
- 63. Suffix Structures (Suffix Array, Suffix Tree, LCP)
- 64. Palindromes and Periodicity (Manacher)
- 65. Edit Distance and Alignment

- 66. Compression (Huffman, Arithmetic, LZ77, BWT)
- 67. Cryptographic Hashes and Checksums
- 68. Approximate and Streaming Matching
- 69. Bioinformatics Alignment (Needleman–Wunsch, Smith–Waterman)
- 70. Text Indexing and Search Structures

## **Chapter 8. Geometry, Graphics, and Spatial Algorithms**

- 71. Convex Hull (Graham, Andrew, Chan)
- 72. Closest Pair and Segment Intersection
- 73. Line Sweep and Plane Sweep Algorithms
- 74. Delaunay and Voronoi Diagrams
- 75. Point in Polygon and Polygon Triangulation
- 76. Spatial Data Structures (KD, R-tree)
- 77. Rasterization and Scanline Techniques
- 78. Computer Vision (Canny, Hough, SIFT)
- 79. Pathfinding in Space (A\*, RRT, PRM)
- 80. Computational Geometry Variants and Applications

## **Chapter 9. Systems, Databases, and Distributed Algorithms**

- 81. Concurrency Control (2PL, MVCC, OCC)
- 82. Logging, Recovery, and Commit Protocols
- 83. Scheduling (Round Robin, EDF, Rate-Monotonic)

- 84. Caching and Replacement (LRU, LFU, CLOCK)
- 85. Networking (Routing, Congestion Control)
- 86. Distributed Consensus (Paxos, Raft, PBFT)
- 87. Load Balancing and Rate Limiting
- 88. Search and Indexing (Inverted, BM25, WAND)
- 89. Compression and Encoding in Systems
- 90. Fault Tolerance and Replication

## **Chapter 10. AI, ML, and Optimization**

- 91. Classical ML (k-means, Naive Bayes, SVM, Decision Trees)
- 92. Ensemble Methods (Bagging, Boosting, Random Forests)
- 93. Gradient Methods (SGD, Adam, RMSProp)
- 94. Deep Learning (Backpropagation, Dropout, Normalization)
- 95. Sequence Models (Viterbi, Beam Search, CTC)
- 96. Metaheuristics (GA, SA, PSO, ACO)
- 97. Reinforcement Learning (Q-learning, Policy Gradients)
- 98. Approximation and Online Algorithms
- 99. Fairness, Causal Inference, and Robust Optimization
- 100. AI Planning, Search, and Learning Systems

# The Cheatsheet

## Page 1. Big Picture and Complexity

A quick reference for understanding algorithms, efficiency, and growth rates. Keep this sheet beside you as you read or code.

### What Is an Algorithm?

An algorithm is a clear, step-by-step process that solves a problem.

Property	Description
Precise	Each step is unambiguous
Finite	Must stop after a certain number of steps
Effective	Every step is doable by machine or human
Deterministic	Same input, same output (usually)

Think of it like a recipe:

- Input: ingredients
- Steps: instructions
- Output: final dish

### Core Qualities

Concept	Question to Ask
Correctness	Does it always solve the problem
Termination	Does it eventually stop
Complexity	How much time and space it needs
Clarity	Is it easy to understand and implement

## Why Complexity Matters

Different algorithms grow differently as input size  $n$  increases.

Growth Rate	Example Algorithm	Effect When $n$ Doubles
$O(1)$	Hash lookup	No change
$O(\log n)$	Binary search	Slight increase
$O(n)$	Linear scan	Doubled
$O(n \log n)$	Merge sort	Slightly more than 2×
$O(n^2)$	Bubble sort	Quadrupled
$O(2^n)$	Subset generation	Explodes
$O(n!)$	Brute-force permutations	Unusable beyond $n = 10$

## Measuring Time and Space

Measure	Meaning	Example
Time	Number of operations	Loop from 1 to $n$ : $O(n)$
Complexity		
Space	Memory usage (stack, heap, data structures)	Recursive call depth: $O(n)$
Complexity		

Simple rules:

- Sequential steps: sum of costs
- Nested loops: product of sizes
- Recursion: use recurrence relations

## Common Patterns

Pattern	Cost Formula	Complexity
Single Loop (1 to $n$ )	$T(n) = n$	$O(n)$
Nested Loops ( $n \times n$ )	$T(n) = n^2$	$O(n^2)$
Halving Each Step	$T(n) = \log_2 n$	$O(\log n)$
Divide and Conquer (2 halves)	$T(n) = 2T(n/2) + n$	$O(n \log n)$

## Doubling Rule

Run algorithm for  $n$  and  $2n$ :

Observation	Likely Complexity
Constant time	$O(1)$
Time doubles	$O(n)$
Time quadruples	$O(n^2)$
Time $\times$ log factor	$O(n \log n)$

## Tiny Code: Binary Search

```
def binary_search(arr, x):
    lo, hi = 0, len(arr) - 1
    while lo <= hi:
        mid = (lo + hi) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] < x:
            lo = mid + 1
        else:
            hi = mid - 1
    return -1
```

Complexity:

$$T(n) = T(n/2) + 1 \Rightarrow O(\log n)$$

## Common Pitfalls

Issue	Tip
Off-by-one error	Check loop bounds carefully
Infinite loop	Ensure termination condition is reachable
Midpoint overflow (C/C++)	Use $mid = lo + (hi - lo) / 2$
Unsorted data in search	Binary search only works on sorted input

## Quick Growth Summary

Type	Formula	Example	Description
Constant	1		Fixed time
Logarithmic	$\log n$		Divide each time
Linear	$n$		Step through all items
Linearithmic	$n \log n$		Sort-like complexity
Quadratic	$n^2$		Double loop
Cubic	$n^3$		Triple nested loops
Exponential	$2^n$		All subsets
Factorial	$n!$		All permutations

## Simple Rule of Thumb

Trace small examples by hand. Count steps, memory, and recursion depth. You'll see how growth behaves before running code.

## Page 2. Recurrences and Master Theorem

This page helps you break down recursive algorithms and estimate their runtime using recurrences.

### What Is a Recurrence?

A recurrence relation expresses a problem's cost  $T(n)$  in terms of smaller subproblems.

Typical structure:

$$T(n) = a, T\left(\frac{n}{b}\right) + f(n)$$

where:

- $a$  = number of subproblems
- $b$  = factor by which input shrinks
- $f(n)$  = extra work per call (merge, combine, etc.)

### Common Recurrences

Algorithm	Recurrence Form	Solution
Binary Search	$T(n) = T(n/2) + 1$	$O(\log n)$
Merge Sort	$T(n) = 2T(n/2) + n$	$O(n \log n)$
Quick Sort (avg)	$T(n) = 2T(n/2) + O(n)$	$O(n \log n)$
Quick Sort (worst)	$T(n) = T(n-1) + O(n)$	$O(n^2)$
Matrix Multiply	$T(n) = 8T(n/2) + O(n^2)$	$O(n^3)$
Karatsuba	$T(n) = 3T(n/2) + O(n)$	$O(n^{\log_2 3})$

## Solving Recurrences

There are several methods to solve them:

Method	Description	Best For
Iteration	Expand step by step	Simple recurrences
Substitution	Guess and prove with induction	Verification
Recursion Tree	Visualize total work per level	Divide and conquer
Master Theorem	Shortcut for $T(n) = aT(n/b) + f(n)$	Standard forms

## The Master Theorem

Given

$$T(n) = aT(n/b) + f(n)$$

Let

$$n^{\log_b a}$$

be the “critical term”

Case Condition		Result
1	If $f(n) = O(n^{\log_b a - \varepsilon})$	$T(n) = \Theta(n^{\log_b a})$
2	If $f(n) = \Theta(n^{\log_b a} \log^k n)$	$T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
3	If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ and regularity holds	$T(n) = \Theta(f(n))$

## Examples

Algorithm	$a$	$b$	$f(n)$	Case	$T(n)$
Merge Sort	2	2	$n$	2	$\Theta(n \log n)$

Algorithm	$a$	$b$	$f(n)$	Case	$T(n)$
Binary Search	1	2	1	1	$\Theta(\log n)$
Strassen Multiply	7	2	$n^2$	2	$\Theta(n^{\log_2 7})$
Quick Sort (avg)	2	2	$n$	2	$\Theta(n \log n)$

## Recursion Tree Visualization

Break cost into levels:

Example:  $T(n) = 2T(n/2) + n$

Level	#Nodes	Work per Node	Total Work
0	1	$n$	$n$
1	2	$n/2$	$n$
2	4	$n/4$	$n$
...	...	...	...

Sum across  $\log_2 n$  levels:

$$T(n) = n \log_2 n$$

## Tiny Code: Fast Exponentiation

Compute  $a^n$  efficiently.

```
def power(a, n):
    res = 1
    while n > 0:
        if n % 2 == 1:
            res *= a
        a *= a
        n //= 2
    return res
```

Recurrence:

$$T(n) = T(n/2) + O(1) \Rightarrow O(\log n)$$

## Iteration Method Example

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

Expand:

$$T(n) = T(n/2) + n = T(n/4) + n/2 + n = T(n/8) + n/4 + n/2 + n = \dots + n(1 + 1/2 + 1/4 + \dots) = O(n)$$

## Common Forms

Form	Result
$T(n) = T(n - 1) + O(1)$	$O(n)$
$T(n) = T(n/2) + O(1)$	$O(\log n)$
$T(n) = 2T(n/2) + O(1)$	$O(n)$
$T(n) = 2T(n/2) + O(n)$	$O(n \log n)$
$T(n) = T(n/2) + O(n)$	$O(n)$

## Quick Checklist

1. Identify  $a$ ,  $b$ , and  $f(n)$
2. Compare  $f(n)$  to  $n^{\log_b a}$
3. Apply correct case
4. Confirm assumptions (regularity)
5. State final complexity

Understanding recurrences helps you estimate performance before coding. Always look for subproblem count, size, and merge cost.

## Page 3. Sorting at a Glance

Sorting is one of the most common algorithmic tasks. This page helps you quickly compare sorting methods, their complexity, stability, and when to use them.

### Why Sorting Matters

Sorting organizes data so that searches, merges, and analyses become efficient. Many problems become simpler once the input is sorted.

## Quick Comparison Table

Algorithm	Best Case	Average Case	Worst Case	Space	Stable	In-Place	Notes
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Yes	Simple, educational
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No	Yes	Few swaps
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Yes	Great for small/partial sort
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes	No	Stable, divide and conquer
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	No	Yes	Fast average, in place
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	No	Yes	Not stable
Counting Sort	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(n + k)$	Yes	No	Integer keys only
Radix Sort	$O(d(n + k))$	$O(d(n + k))$	$O(d(n + k))$	$O(n + k)$	Yes	No	Sort by digits
Bucket Sort	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n)$	Yes	No	Uniform distribution needed

## Choosing a Sorting Algorithm

Situation	Best Choice
Small array or nearly sorted data	Insertion Sort
Stable required, general case	Merge Sort or Timsort
In-place and fast on average	Quick Sort
Guarantee worst-case $O(n \log n)$	Heap Sort
Small integer keys or limited range	Counting or Radix
External sorting (large data)	External Merge Sort

## Tiny Code: Insertion Sort

Simple and intuitive for beginners.

```
def insertion_sort(a):
    for i in range(1, len(a)):
```

```

key = a[i]
j = i - 1
while j >= 0 and a[j] > key:
    a[j + 1] = a[j]
    j -= 1
a[j + 1] = key
return a

```

Complexity:

$$T(n) = O(n^2)$$

average,

$$O(n)$$

best (already sorted)

## Divide and Conquer Sorts

### Merge Sort

Splits list, sorts halves, merges results.

Recurrence:

$$T(n) = 2T(n/2) + O(n) = O(n \log n)$$

Tiny Code:

```

def merge_sort(a):
    if len(a) <= 1:
        return a
    mid = len(a)//2
    L = merge_sort(a[:mid])
    R = merge_sort(a[mid:])
    i = j = 0
    res = []
    while i < len(L) and j < len(R):
        if L[i] <= R[j]:
            res.append(L[i]); i += 1
        else:
            res.append(R[j]); j += 1
    res.extend(L[i:]); res.extend(R[j:])
    return res

```

## Quick Sort

Pick pivot, partition, sort subarrays.

Recurrence:

$$T(n) = T(k) + T(n - k - 1) + O(n)$$

Average case:

$$O(n \log n)$$

Worst case:

$$O(n^2)$$

Tiny Code:

```
def quick_sort(a):
    if len(a) <= 1:
        return a
    pivot = a[len(a)//2]
    left  = [x for x in a if x < pivot]
    mid   = [x for x in a if x == pivot]
    right = [x for x in a if x > pivot]
    return quick_sort(left) + mid + quick_sort(right)
```

## Stable vs Unstable

Property	Description	Example
Stable	Equal elements keep original order	Merge Sort, Insertion
Unstable	May reorder equal elements	Quick, Heap

## Visualization Tips

Pattern	Description
Bubble	Compare and swap adjacent
Selection	Select min each pass
Insertion	Grow sorted region step by step
Merge	Divide, conquer, merge
Quick	Partition and recurse
Heap	Build heap, extract repeatedly

## Summary Table

Type	Category	Complexity	Stable	Space
Simple	Bubble, Selection	$O(n^2)$	Varies	$O(1)$
Insertion	Incremental	$O(n^2)$	Yes	$O(1)$
Divide/Conquer	Merge, Quick	$O(n \log n)$	Merge yes	Merge no
Distribution	Counting, Radix	$O(n + k)$	Yes	$O(n + k)$
Hybrid	Timsort, IntroSort	$O(n \log n)$	Yes	Varies

When in doubt, start with Timsort (Python) or std::sort (C++) which adapt dynamically.

## Page 4. Searching and Selection

Searching means finding what you need from a collection. Selection means picking specific elements such as the smallest, largest, or k-th element. This page summarizes both.

### Searching Basics

Type	Description	Data Requirement	Complexity
Linear Search	Check one by one	None	$O(n)$
Binary Search	Divide range by 2 each step	Sorted	$O(\log n)$
Jump Search	Skip ahead fixed steps	Sorted	$O(\sqrt{n})$
Interpolation	Guess position based on value	Sorted, uniform	$O(\log \log n)$ avg
Exponential	Expand window, then binary search	Sorted	$O(\log n)$

### Linear Search

Simple but slow for large inputs.

```
def linear_search(a, x):
    for i, v in enumerate(a):
        if v == x:
            return i
    return -1
```

Complexity:

$$T(n) = O(n)$$

## Binary Search

Fast on sorted lists.

```
def binary_search(a, x):
    lo, hi = 0, len(a) - 1
    while lo <= hi:
        mid = (lo + hi) // 2
        if a[mid] == x:
            return mid
        elif a[mid] < x:
            lo = mid + 1
        else:
            hi = mid - 1
    return -1
```

Complexity:

$$T(n) = T(n/2) + 1 \Rightarrow O(\log n)$$

## Binary Search Variants

Variant	Goal	Return Value
Lower Bound	First index where $a[i] \geq x$	Position of first $\geq x$
Upper Bound	First index where $a[i] > x$	Position of first $> x$
Count Range	<code>upper_bound - lower_bound</code>	Count of $x$ in sorted array

## Common Binary Search Pitfalls

Problem	Fix
Infinite loop	Update bounds correctly
Off-by-one	Check mid inclusion carefully
Unsuitable for unsorted data	Sort or use hash-based search
Overflow (C/C++)	<code>mid = lo + (hi - lo) / 2</code>

## Exponential Search

Used for unbounded or large sorted lists.

1. Check positions 1, 2, 4, 8, ... until  $a[i] \geq x$
2. Binary search in last found interval

Complexity:

$$O(\log n)$$

## Selection Problems

Find the  $k$ -th smallest or largest element.

Task	Example Use Case	Algorithm	Complexity
Min / Max	Smallest / largest element	Linear Scan	$O(n)$
$k$ -th Smallest	Order statistic	Quickselect	Avg $O(n)$
Median	Middle element	Quickselect	Avg $O(n)$
Top- $k$ Elements	Partial sort	Heap / Partition	$O(n \log k)$
Median of Medians	Worst-case linear selection	Deterministic	$O(n)$

### Tiny Code: Quickselect ( $k$ -th smallest)

```
import random

def quickselect(a, k):
    if len(a) == 1:
        return a[0]
    pivot = random.choice(a)
    left  = [x for x in a if x < pivot]
    mid   = [x for x in a if x == pivot]
    right = [x for x in a if x > pivot]

    if k < len(left):
        return quickselect(left, k)
    elif k < len(left) + len(mid):
        return pivot
    else:
        return quickselect(right, k - len(left) - len(mid))
```

Complexity: Average  $O(n)$ , Worst  $O(n^2)$

## Tiny Code: Lower Bound

```
def lower_bound(a, x):
    lo, hi = 0, len(a)
    while lo < hi:
        mid = (lo + hi) // 2
        if a[mid] < x:
            lo = mid + 1
        else:
            hi = mid
    return lo
```

## Hash-Based Searching

When order does not matter, hashing gives near constant lookup.

Operation	Average	Worst
Insert	$O(1)$	$O(n)$
Search	$O(1)$	$O(n)$
Delete	$O(1)$	$O(n)$

Best for large, unsorted collections.

## Summary Table

Scenario	Recommended Approach	Complexity
Small array	Linear Search	$O(n)$
Large, sorted array	Binary Search	$O(\log n)$
Unbounded range	Exponential Search	$O(\log n)$
Need k-th smallest element	Quickselect	Avg $O(n)$
Many lookups	Hash Table	Avg $O(1)$

## Quick Tips

- Always check whether data is sorted before applying binary search.
- Quickselect is great when you only need the k-th element, not a full sort.
- Use hash maps for fast lookups on unsorted data.

## Page 5. Core Data Structures

Data structures organize data for efficient access and modification. Choosing the right one often makes an algorithm simple and fast.

### Arrays and Lists

Structure	Access	Search	Insert End	Insert Middle	Delete	Notes
Static Array	$O(1)$	$O(n)$	N/A	$O(n)$	$O(n)$	Fixed size
Dynamic Array	$O(1)$	$O(n)$	Amortized $O(1)$	$O(n)$	$O(n)$	Auto-resizing
Linked List (S)	$O(n)$	$O(n)$	$O(1)$ head	$O(1)$ if node known	$O(1)$ if node known	Sequential access
Linked List (D)	$O(n)$	$O(n)$	$O(1)$ head/tail	$O(1)$ if node known	$O(1)$ if node known	Two-way traversal

- Singly linked lists: next pointer only
- Doubly linked lists: next and prev pointers
- Dynamic arrays use *doubling* to grow capacity

### Tiny Code: Dynamic Array Resize (Python-like)

```
def resize(arr, new_cap):
    new = [None] * new_cap
    for i in range(len(arr)):
        new[i] = arr[i]
    return new
```

Doubling capacity keeps amortized append  $O(1)$ .

### Stacks and Queues

Structure	Push	Pop	Peek	Notes
Stack (LIFO)	$O(1)$	$O(1)$	$O(1)$	Undo operations, recursion
Queue (FIFO)	$O(1)$	$O(1)$	$O(1)$	Scheduling, BFS

Structure	Push	Pop	Peek	Notes
Deque	$O(1)$	$O(1)$	$O(1)$	Insert/remove both ends

### Tiny Code: Stack

```
stack = []
stack.append(x)    # push
x = stack.pop()   # pop
```

### Tiny Code: Queue

```
from collections import deque

q = deque()
q.append(x)    # enqueue
x = q.popleft()  # dequeue
```

### Priority Queue (Heap)

Stores elements so the smallest (or largest) is always on top.

Operation	Complexity
Insert	$O(\log n)$
Extract min	$O(\log n)$
Peek min	$O(1)$
Build heap	$O(n)$

Tiny Code:

```
import heapq
heap = []
heapq.heappush(heap, value)
x = heapq.heappop(heap)
```

Heaps are used in Dijkstra, Prim, and scheduling.

## Hash Tables

Operation	Average	Worst	Notes
Insert	$O(1)$	$O(n)$	Hash collisions increase cost
Search	$O(1)$	$O(n)$	Good hash + low load factor helps
Delete	$O(1)$	$O(n)$	Usually open addressing or chaining

Key ideas:

- Compute index using hash function: `index = hash(key) % capacity`
- Resolve collisions by chaining or probing

### Tiny Code: Hash Map (Simplified)

```
table = [[] for _ in range(8)]
def put(key, value):
    i = hash(key) % len(table)
    for kv in table[i]:
        if kv[0] == key:
            kv[1] = value
            return
    table[i].append([key, value])
```

## Sets

A hash-based collection of unique elements.

Operation	Average Complexity
Add	$O(1)$
Search	$O(1)$
Remove	$O(1)$

Used for membership checks and duplicates removal.

## Union-Find (Disjoint Set)

Keeps track of connected components. Two main operations:

- $\text{find}(x)$ : get representative of  $x$
- $\text{union}(a,b)$ : merge sets of  $a$  and  $b$

With path compression + union by rank  $\rightarrow$  nearly  $O(1)$ .

Tiny Code:

```
class DSU:  
    def __init__(self, n):  
        self.p = list(range(n))  
        self.r = [0]*n  
    def find(self, x):  
        if self.p[x] != x:  
            self.p[x] = self.find(self.p[x])  
        return self.p[x]  
    def union(self, a, b):  
        ra, rb = self.find(a), self.find(b)  
        if ra == rb: return  
        if self.r[ra] < self.r[rb]: ra, rb = rb, ra  
        self.p[rb] = ra  
        if self.r[ra] == self.r[rb]:  
            self.r[ra] += 1
```

## Summary Table

Category	Structure	Use Case
Sequence	Array, List	Ordered data
LIFO/FIFO	Stack, Queue	Recursion, scheduling
Priority	Heap	Best-first selection, PQ problems
Hash-based	Hash Table, Set	Fast lookups, uniqueness
Connectivity	Union-Find	Graph components, clustering

## Quick Tips

- Choose array when random access matters.
- Choose list when insertions/deletions frequent.
- Choose stack or queue for control flow.

- Choose heap for priority.
- Choose hash table for constant lookups.
- Choose DSU for disjoint sets or graph merging.

## Page 6. Graphs Quick Use

Graphs model connections between objects. They appear everywhere: maps, networks, dependencies, and systems. This page gives you a compact view of common graph algorithms.

### Graph Basics

A graph has vertices (nodes) and edges (connections).

Type	Description
Undirected	Edges go both ways
Directed (Digraph)	Edges have direction
Weighted	Edges carry cost or distance
Unweighted	All edges cost 1

### Representations

Representation	Space	Best For	Notes
Adjacency List	$O(V + E)$	Sparse graphs	Common in practice
Adjacency Matrix	$O(V^2)$	Dense graphs	Constant-time edge lookup
Edge List	$O(E)$	Edge-based algorithms	Easy to iterate over edges

Adjacency List Example (Python):

```
graph = {
    0: [(1, 2), (2, 5)],
    1: [(2, 1)],
    2: []
}
```

Each tuple (`neighbor, weight`) represents an edge.

## Traversals

### Breadth-First Search (BFS)

Visits layer by layer (good for shortest paths in unweighted graphs).

```
from collections import deque
def bfs(adj, s):
    dist = {s: 0}
    q = deque([s])
    while q:
        u = q.popleft()
        for v in adj[u]:
            if v not in dist:
                dist[v] = dist[u] + 1
                q.append(v)
    return dist
```

Complexity:  $O(V + E)$

### Depth-First Search (DFS)

Explores deeply before backtracking.

```
def dfs(adj, u, visited):
    visited.add(u)
    for v in adj[u]:
        if v not in visited:
            dfs(adj, v, visited)
```

Complexity:  $O(V + E)$

## Shortest Path Algorithms

Algorithm	Works On	Negative Edges	Complexity	Notes
BFS	Unweighted	No	$O(V + E)$	Shortest hops
Dijkstra	Weighted (nonneg)	No	$O((V + E) \log V)$	Uses priority queue
Bellman-Ford	Weighted	Yes	$O(VE)$	Detects negative cycles

Algorithm	Works On	Negative Edges	Complexity	Notes
Floyd-Warshall	All pairs	Yes	$O(V^3)$	DP approach

### Tiny Code: Dijkstra's Algorithm

```
import heapq

def dijkstra(adj, s):
    INF = 1018
    dist = [INF] * len(adj)
    dist[s] = 0
    pq = [(0, s)]
    while pq:
        d, u = heapq.heappop(pq)
        if d != dist[u]:
            continue
        for v, w in adj[u]:
            nd = d + w
            if nd < dist[v]:
                dist[v] = nd
                heapq.heappush(pq, (nd, v))
    return dist
```

### Topological Sort (DAGs only)

Orders nodes so every edge  $(u, v)$  goes from earlier to later.

Method	Idea	Complexity
DFS-based	Post-order stack reversal	$O(V + E)$
Kahn's Algo	Remove nodes with indegree 0	$O(V + E)$

### Minimum Spanning Tree (MST)

Connect all nodes with minimum total weight.

Algorithm	Idea	Complexity	Notes
Kruskal	Sort edges, use Union-Find	$O(E \log E)$	Works well with edge list
Prim	Grow tree using PQ	$O(E \log V)$	Starts from any vertex

### Tiny Code: Kruskal MST

```
def kruskal(edges, n):
    parent = list(range(n))
    def find(x):
        if parent[x] != x:
            parent[x] = find(parent[x])
        return parent[x]
    res = 0
    for w, u, v in sorted(edges):
        ru, rv = find(u), find(v)
        if ru != rv:
            res += w
            parent[rv] = ru
    return res
```

### Strongly Connected Components (SCC)

Subsets where every node can reach every other. Use Kosaraju or Tarjan algorithm, both  $O(V + E)$ .

### Cycle Detection

Graph Type	Method	Notes
Undirected	DFS with parent	Edge to non-parent visited
Directed	DFS with color/state	Back edge found = cycle

### Summary Table

Task	Algorithm	Complexity	Notes
Visit all nodes	DFS / BFS	$O(V + E)$	Traversal

Task	Algorithm	Complexity	Notes
Shortest path (unweighted)	BFS	$O(V + E)$	Counts edges
Shortest path (weighted)	Dijkstra	$O(E \log V)$	No negative weights
Negative edges allowed	Bellman-Ford	$O(VE)$	Detects negative cycles
All-pairs shortest path	Floyd-Warshall	$O(V^3)$	DP matrix
MST	Kruskal / Prim	$O(E \log V)$	Minimal connection cost
DAG order	Topological Sort	$O(V + E)$	Only for DAGs

## Quick Tips

- Use BFS for shortest path in unweighted graphs.
- Use Dijkstra if weights are nonnegative.
- Use Union-Find for Kruskal MST.
- Use Topological Sort for dependency resolution.
- Always check for negative edges before using Dijkstra.

## Page 7. Dynamic Programming Quick Use

Dynamic Programming (DP) is about solving big problems by breaking them into overlapping subproblems and reusing their solutions. This page helps you see patterns quickly.

### When to Use DP

You can usually apply DP if:

Symptom	Meaning
Optimal Substructure	Best solution uses best of subparts
Overlapping Subproblems	Same subresults appear again
Decision + Recurrence	State transitions can be defined

### DP Styles

Style	Description	Example
Top-down (Memo)	Recursion + cache results	Fibonacci with memoization
Bottom-up (Tabular)	Iterative fill table	Knapsack table
Space-optimized	Reuse previous row/state	Rolling arrays

## Fibonacci Example

Recurrence:

$$F(n) = F(n - 1) + F(n - 2), \quad F(0) = 0, F(1) = 1$$

### Top-down (Memoization)

```
def fib(n, memo={}):
    if n <= 1:
        return n
    if n not in memo:
        memo[n] = fib(n-1, memo) + fib(n-2, memo)
    return memo[n]
```

### Bottom-up (Tabulation)

```
def fib(n):
    dp = [0, 1]
    for i in range(2, n + 1):
        dp.append(dp[i-1] + dp[i-2])
    return dp[n]
```

## Steps to Solve DP Problems

1. Define State Example:  $dp[i]$  = best answer for first  $i$  items
2. Define Transition Example:  $dp[i] = \max(dp[i - 1], value[i] + dp[i - weight[i]])$
3. Set Base Cases Example:  $dp[0] = 0$
4. Choose Order Bottom-up or Top-down
5. Return Answer Often  $dp[n]$  or  $dp[target]$

## Common DP Categories

Category	Example Problems	State Form
Sequence	LIS, LCS, Edit Distance	$dp[i][j]$ over prefixes
Subset	Knapsack, Subset Sum	$dp[i][w]$ capacity-based
Partition	Palindrome Partitioning, Equal Sum	$dp[i]$ cut-based
Grid	Min Path Sum, Unique Paths	$dp[i][j]$ over cells
Counting	Coin Change Count, Stairs	Add ways from subproblems

Category	Example Problems	State Form
Interval	Matrix Chain, Burst Balloons	$dp[i][j]$ range subproblem
Bitmask	TSP, Assignment	$dp[mask][i]$ subset states
Digit	Count numbers with constraint	$dp[pos][tight][sum]$ digits
Tree	Rerooting, Subtree DP	$dp[u]$ over children

## Classic Problems

Problem	State Definition	Transition
Climbing Stairs	$dp[i]$ = ways to reach step i	$dp[i] = dp[i - 1] + dp[i - 2]$
Coin Change (Count Ways)	$dp[x]$ = ways to make sum x	$dp[x] += dp[x - coin]$
0/1 Knapsack	$dp[w]$ = max value under weight w	$dp[w] = \max(dp[w], dp[w - w_i] + v_i)$
Longest Increasing Subseq.	$dp[i]$ = LIS ending at i	if $a[j] < a[i]$ , $dp[i] = dp[j] + 1$
Edit Distance	$dp[i][j]$ = edit cost	min(insert, delete, replace)
Matrix Chain Multiplication	$dp[i][j]$ = min cost mult subchain	$dp[i][j] = \min_k(dp[i][k] + dp[k + 1][j])$

## Tiny Code: 0/1 Knapsack (1D optimized)

```
def knapsack(weights, values, W):
    dp = [0]*(W+1)
    for i in range(len(weights)):
        for w in range(W, weights[i]-1, -1):
            dp[w] = max(dp[w], dp[w-weights[i]] + values[i])
    return dp[W]
```

## Sequence Alignment Example

Edit Distance Recurrence:

$$dp[i][j] = \begin{cases} dp[i - 1][j - 1], & \text{if } s[i] = t[j], \\ 1 + \min(dp[i - 1][j], dp[i][j - 1], dp[i - 1][j - 1]), & \text{otherwise.} \end{cases}$$

## Optimization Techniques

Technique	When to Use	Example
Space Optimization	2D → 1D states reuse	Knapsack, LCS
Prefix/Suffix Precomp	Range aggregates	Sum/Min queries
Divide & Conquer DP	Monotonic decisions	Matrix Chain
Convex Hull Trick	Linear transition minima	DP on lines
Bitset DP	Large boolean states	Subset sum optimization

## Debugging Tips

- Print partial dp arrays to see progress.
- Check base cases carefully.
- Ensure loops match transition dependencies.
- Always confirm the recurrence before coding.

## Page 8. Mathematics for Algorithms Quick Use

Mathematics builds the foundation for algorithmic reasoning. This page collects essential formulas and methods every programmer should know.

### Number Theory Essentials

Topic	Description	Formula / Idea
GCD (Euclidean)	Greatest common divisor	$gcd(a, b) = gcd(b, a)$
Extended GCD	Solve $ax + by = gcd(a, b)$	Backtrack coefficients
LCM	Least common multiple	$lcm(a, b) = \frac{a \cdot b}{gcd(a, b)}$
Modular Addition	Add under modulo M	$(a + b) \bmod M$
Modular Multiply	Multiply under modulo M	$(a \cdot b) \bmod M$
Modular Inverse	$a^{-1} \bmod M$	$a^{M-2} \bmod M$ if M is prime
Modular Exponent	Fast exponentiation	Square and multiply
CRT	Combine congruences	Solve system $x \equiv a_i \pmod{m_i}$

Tiny Code (Modular Exponentiation):

```

def modpow(a, n, M):
    res = 1
    while n:
        if n & 1:
            res = res * a % M
        a = a * a % M
        n >= 1
    return res

```

## Primality and Factorization

Algorithm	Use Case	Complexity	Notes
Trial Division	Small n	$O(\sqrt{n})$	Simple
Sieve of Eratosthenes	Generate primes	$O(n \log \log n)$	Classic prime sieve
Miller–Rabin	Probabilistic primality	$O(k \log^3 n)$	Fast for big n
Pollard Rho	Factor composite	$O(n^{1/4})$	Randomized
Sieve of Atkin	Faster variant	$O(n)$	Complex implementation

## Combinatorics

Formula	Description
$n! = n \cdot (n - 1) \cdots 1$	Factorial
$\binom{n}{k} = \frac{n!}{k!(n - k)!}$	Number of combinations
$P(n, k) = \frac{n!}{(n - k)!}$	Number of permutations
Pascal's Rule: $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$	Build Pascal's Triangle
Catalan: $C_n = \frac{1}{n+1} \binom{2n}{n}$	Parentheses counting

Tiny Code (nCr with factorials mod M):

```

def nCr(n, r, fact, inv):
    return fact[n]*inv[r]%M*inv[n-r]%M

```

## Probability Basics

Concept	Formula or Idea
Probability	$P(A) = \frac{\text{favorable}}{\text{total}}$
Complement	$P(\bar{A}) = 1 - P(A)$
Union	$P(A \cup B) = P(A) + P(B) - P(A \cap B)$
Conditional	$P(A B) = \frac{P(A \cap B)}{P(B)}$
Bayes' Theorem	$P(A B) = \frac{P(B A)P(A)}{P(B)}$
Expected Value	$E[X] = \sum x_i P(x_i)$
Variance	$Var(X) = E[X^2] - E[X]^2$

## Linear Algebra Core

Operation	Formula / Method	Complexity
Gaussian Elimination	Solve $Ax = b$	$O(n^3)$
Determinant	Product of pivots	$O(n^3)$
Matrix Multiply	$(AB) * ij = \sum_k A * ik B_{kj}$	$O(n^3)$
Transpose	$A_{ij}^T = A_{ji}$	$O(n^2)$
LU Decomposition	$A = LU$ (lower, upper)	$O(n^3)$
Cholesky	$A = LL^T$ (symmetric pos. def.)	$O(n^3)$
Power Method	Dominant eigenvalue estimation	iterative

Tiny Code (Gaussian Elimination Skeleton):

```
for i in range(n):
    pivot = a[i][i]
    for j in range(i, n+1):
        a[i][j] /= pivot
    for k in range(n):
        if k != i:
            ratio = a[k][i]
            for j in range(i, n+1):
                a[k][j] -= ratio*a[i][j]
```

## Fast Transforms

Transform	Use Case	Complexity	Notes
FFT	Polynomial convolution	$O(n \log n)$	Complex numbers
NTT	Modular convolution	$O(n \log n)$	Prime modulus
FWT (XOR)	XOR-based convolution	$O(n \log n)$	Subset DP

FFT Equation:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-2\pi i k n / N}$$

## Numerical Methods

Method	Purpose	Formula or Idea
Bisection	Root-finding	Midpoint halve until $f(x) = 0$
Newton–Raphson	Fast convergence	$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$
Secant Method	Approx derivative	$x_{n+1} = x_n - f(x_n) \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}$
Simpson's Rule	Integration	$\int_a^b f(x) dx \approx \frac{h}{3} (f(a) + 4f(m) + f(b))$

## Optimization and Calculus

Concept	Formula / Idea
Derivative	$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$
Gradient Descent	$x_{k+1} = x_k - \eta \nabla f(x_k)$
Lagrange Multipliers	$\nabla f = \lambda \nabla g$
Convex Function	$f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda)f(y)$

Tiny Code (Gradient Descent):

```
x = x0
for _ in range(1000):
    grad = df(x)
    x -= lr * grad
```

## Algebraic Tricks

Topic	Formula / Use
Exponentiation	$a^n$ via square-multiply
Polynomial Deriv.	$(ax^n)' = n \cdot ax^{n-1}$
Integration	$\int x^n dx = \frac{x^{n+1}}{n+1} + C$
Möbius	$f(n) = \sum_{d n} g(d) \Rightarrow g(n) = \sum_{d n} \mu(d) \cdot f(n/d)$
Inversion	

## Quick Reference Table

Domain	Must-Know Algorithm
Number Theory	GCD, Mod Exp, CRT
Combinatorics	Pascal, Factorial, Catalan
Probability	Bayes, Expected Value
Linear Algebra	Gaussian Elimination
Transforms	FFT, NTT
Optimization	Gradient Descent

## Page 9. Strings and Text Algorithms Quick Use

Strings are sequences of characters used in text search, matching, and transformation. This page gives quick references to classical and modern string techniques.

### String Fundamentals

Concept	Description	Example
Alphabet	Set of symbols	{a, b, c}
String Length	Number of characters	"hello" → 5
Substring	Continuous part of string	"ell" in "hello"
Subsequence	Ordered subset (not necessarily cont.)	"hlo" from "hello"
Prefix / Suffix	Starts / ends part of string	"he", "lo"

Indexing: Most algorithms use 0-based indexing.

## String Search Overview

Algorithm	Complexity	Description
Naive Search	$O(nm)$	Check all positions
KMP	$O(n + m)$	Prefix-suffix skip table
Z-Algorithm	$O(n + m)$	Precompute match lengths
Rabin-Karp	$O(n + m)$ avg	Rolling hash check
Boyer-Moore	$O(n/m)$ avg	Backward scan, skip mismatches

## KMP Prefix Function

Compute prefix-suffix matches for pattern.

Step	Meaning
$pi[i]$	Longest proper prefix that is also suffix for $pattern[0 : i]$

Tiny Code:

```
def prefix_function(p):
    pi = [0]*len(p)
    j = 0
    for i in range(1, len(p)):
        while j > 0 and p[i] != p[j]:
            j = pi[j-1]
        if p[i] == p[j]:
            j += 1
        pi[i] = j
    return pi
```

Search uses  $pi$  to skip mismatches.

## Z-Algorithm

Computes length of substring starting at  $i$  matching prefix.

Step	Meaning
$Z[i]$	Longest substring starting at $i$ matching prefix

Use  $\$S = \text{pattern} + '\$' + \text{text}$  to find pattern occurrences.

## Rabin–Karp Rolling Hash

---

Idea Compute hash for window of text, slide, compare

---

Hash Function:

$$h(s) = (s_0 p^{n-1} + s_1 p^{n-2} + \dots + s_{n-1}) \bmod M$$

Update efficiently when sliding one character.

Tiny Code:

```
def rolling_hash(s, base=257, mod=109+7):
    h = 0
    for ch in s:
        h = (h*base + ord(ch)) % mod
    return h
```

## Advanced Pattern Matching

Algorithm	Use Case	Complexity
Boyer–Moore	Large alphabet	$O(n/m)$ avg
Sunday	Last char shift heuristic	$O(n)$ avg
Bitap	Approximate match	$O(nm/w)$
Aho–Corasick	Multi-pattern search	$O(n + z)$

## Aho–Corasick Automaton

Build a trie from patterns and compute failure links.

Step	Description
Build Trie	Add all patterns
Failure Link	Fallback to next prefix
Output Link	Record pattern match

Tiny Code Sketch:

```

from collections import deque

def build_ac(patterns):
    trie = [{}]
    fail = [0]
    for pat in patterns:
        node = 0
        for c in pat:
            node = trie[node].setdefault(c, len(trie))
            if node == len(trie):
                trie.append({})
                fail.append(0)
    # compute failure links
    q = deque()
    for c in trie[0]:
        q.append(trie[0][c])
    while q:
        u = q.popleft()
        for c, v in trie[u].items():
            f = fail[u]
            while f and c not in trie[f]:
                f = fail[f]
            fail[v] = trie[f].get(c, 0)
            q.append(v)
    return trie, fail

```

## Suffix Structures

Structure	Purpose	Build Time
Suffix Array	Sorted list of suffix indices	$O(n \log n)$
LCP Array	Longest Common Prefix of suffix	$O(n)$
Suffix Tree	Trie of suffixes	$O(n)$ (Ukkonen)
Suffix Automaton	Minimal DFA of substrings	$O(n)$

Suffix Array Doubling Approach:

- Rank substrings of length  $2^k$
- Sort and merge using pairs of ranks

LCP via Kasai's Algorithm:

$$LCP[i] = \text{common prefix of } S[SA[i] :], S[SA[i-1] :]$$

## Palindrome Detection

Algorithm	Description	Complexity
Manacher's Algorithm	Longest palindromic substring	$O(n)$
DP Table	Check substring palindrome	$O(n^2)$
Center Expansion	Expand around center	$O(n^2)$

Manacher's Core:

- Transform with separators (#)
- Track radius of palindrome around each center

## Edit Distance Family

Algorithm	Description	Complexity
Levenshtein Distance	Insert/Delete/Replace	$O(nm)$
Damerau–Levenshtein	Add transposition	$O(nm)$
Hirschberg	Space-optimized LCS	$O(nm)$ time, $O(n)$ space

Recurrence:

$$dp[i][j] = \min \{ dp[i-1][j] + 1, dp[i][j-1] + 1, dp[i-1][j-1] + (s_i \neq t_j) \}$$

## Compression Techniques

Algorithm	Type	Idea
Huffman Coding	Prefix code	Shorter codes for frequent chars
Arithmetic Coding	Range encoding	Fractional interval representation
LZ77 / LZ78	Dictionary-based	Reuse earlier substrings
BWT + MTF + RLE	Block sorting	Group similar chars before coding

Huffman Principle: Shorter bit strings assigned to higher frequency symbols.

## Hashing and Checksums

Algorithm	Use Case	Notes
CRC32	Error detection	Simple polynomial mod
MD5	Hash (legacy)	Not secure
SHA-256	Secure hash	Cryptographic
Rolling Hash	Substring compare	Used in Rabin–Karp

## Quick Reference

Task	Algorithm	Complexity
Single pattern search	KMP / Z	$O(n + m)$
Multi-pattern search	Aho–Corasick	$O(n + z)$
Approximate search	Bitap / Wu–Manber	$O(kn)$
Substring queries	Suffix Array + LCP	$O(\log n)$
Palindromes	Manacher	$O(n)$
Compression	Huffman / LZ77	variable
Edit distance	DP table	$O(nm)$

## Page 10. Geometry, Graphics, and Spatial Algorithms Quick Use

Geometry helps us solve problems about shapes, distances, and spatial relationships. This page summarizes core computational geometry techniques with simple formulas and examples.

### Coordinate Basics

Concept	Description	Formula / Example
Point	Distance between $(x_1, y_1)$ and $(x_2, y_2)$	$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$
Distance	Between two points	$(\frac{x_1+x_2}{2}, \frac{y_1+y_2}{2})$
Midpoint	Angle & projection	$\vec{a} \cdot \vec{b} =  a  b  \cos \theta$
Dot	Signed area, orientation	$a \times b = a_x b_y - a_y b_x$
Product	CCW, CW, collinear check	$\text{sign}(a \times b)$
Cross Product	Orientation Test	
(2D)		

Tiny Code (Orientation Test):

```

def orient(a, b, c):
    val = (b[0]-a[0])*(c[1]-a[1]) - (b[1]-a[1])*(c[0]-a[0])
    return 0 if val == 0 else (1 if val > 0 else -1)

```

## Convex Hull

Find the smallest convex polygon enclosing all points.

Algorithm	Complexity	Notes
Graham Scan	$O(n \log n)$	Sort by angle, use stack
Andrew's Monotone	$O(n \log n)$	Sort by x, build upper/lower
Jarvis March	$O(nh)$	Wrap hull, h = hull size
Chan's Algorithm	$O(n \log h)$	Output-sensitive hull

Steps:

1. Sort points
2. Build lower hull
3. Build upper hull
4. Concatenate

## Closest Pair of Points

Divide-and-conquer approach.

Step	Description
Split by x	Divide points into halves
Recurse and merge	Track min distance across strip

Complexity:  $O(n \log n)$

Formula:

$$d(p, q) = \sqrt{(x_p - x_q)^2 + (y_p - y_q)^2}$$

## Line Intersection

Two segments  $(p_1, p_2)$  and  $(q_1, q_2)$  intersect if:

1. Orientations differ
2. Segments overlap on line if collinear

Tiny Code:

```
def intersect(p1, p2, q1, q2):  
    o1 = orient(p1, p2, q1)  
    o2 = orient(p1, p2, q2)  
    o3 = orient(q1, q2, p1)  
    o4 = orient(q1, q2, p2)  
    return o1 != o2 and o3 != o4
```

## Polygon Area (Shoelace Formula)

For vertices  $(x_i, y_i)$  in order:

$$A = \frac{1}{2} \left| \sum_{i=0}^{n-1} (x_i y_{i+1} - x_{i+1} y_i) \right|$$

Tiny Code:

```
def area(poly):  
    s = 0  
    n = len(poly)  
    for i in range(n):  
        x1, y1 = poly[i]  
        x2, y2 = poly[(i+1)%n]  
        s += x1*y2 - x2*y1  
    return abs(s)/2
```

## Point in Polygon

Method	Idea	Complexity
Ray Casting	Count edge crossings	$O(n)$
Winding Number	Track signed rotations	$O(n)$
Convex Test	Check all orientations	$O(n)$

Ray Casting: Odd number of crossings → inside.

## Rotating Calipers

Used for:

- Polygon diameter (farthest pair)
- Minimum bounding box
- Width and antipodal pairs

Idea: Sweep around convex hull using tangents. Complexity:  $O(n)$  after hull.

## Sweep Line Techniques

Problem	Method	Complexity
Closest Pair	Active set by y	$O(n \log n)$
Segment Intersection	Event-based sweeping	$O((n + k) \log n)$
Rectangle Union Area	Vertical edge events	$O(n \log n)$
Skyline Problem	Merge by height	$O(n \log n)$

Use balanced trees or priority queues for active sets.

## Circle Geometry

Concept	Formula
Equation	$(x - x_c)^2 + (y - y_c)^2 = r^2$
Tangent Length	$\sqrt{d^2 - r^2}$
Two-Circle Intersection	Distance-based geometry

## Spatial Data Structures

Structure	Use Case	Notes
KD-Tree	Nearest neighbor search	Axis-aligned splits
R-Tree	Range queries	Bounding boxes hierarchy
Quadtree	2D recursive subdivision	Graphics, collision detection
Octree	3D extension	Volumetric partitioning

Structure	Use Case	Notes
BSP Tree	Planar splits	Rendering, collision

## Rasterization and Graphics

Algorithm	Purpose	Notes
Bresenham Line	Draw line integer grid	No floating point
Midpoint Circle	Circle rasterization	Symmetry exploitation
Scanline Fill	Polygon fill algorithm	Sort edges, horizontal sweep
Z-Buffer	Hidden surface removal	Per-pixel depth comparison
Phong Shading	Smooth lighting	Interpolate normals

## Pathfinding in Space

Algorithm	Description	Notes
A*	Heuristic shortest path	$f(n) = g(n) + h(n)$
Theta*	Any-angle path	Shortcut-based
RRT / RRT*	Random exploration	Robotics planning
PRM	Probabilistic roadmap	Sampled graph
Visibility Graph	Connect visible points	Geometric planning

## Quick Summary

Task	Algorithm	Complexity
Convex Hull	Graham / Andrew	$O(n \log n)$
Closest Pair	Divide and Conquer	$O(n \log n)$
Segment Intersection Detection	Sweep Line	$O(n \log n)$
Point in Polygon	Ray Casting	$O(n)$
Polygon Area	Shoelace Formula	$O(n)$
Nearest Neighbor Search	KD-Tree	$O(\log n)$
Pathfinding	A*	$O(E \log V)$

## Tip

- Always sort points for geometry preprocessing.
- Use cross product for orientation tests.
- Prefer integer arithmetic when possible to avoid floating errors.

## Page 11. Systems, Databases, and Distributed Algorithms Quick Use

Systems and databases rely on algorithms that manage memory, concurrency, persistence, and coordination. This page gives an overview of the most important ones.

### Concurrency Control

Ensures correctness when multiple transactions or threads run at once.

Method	Idea	Notes
Two-Phase Locking (2PL)	Acquire locks, then release after commit	Guarantees serializability
Strict 2PL	Hold all locks until commit	Prevents cascading aborts
Conservative 2PL	Lock all before execution	Deadlock-free but less parallel
Timestamp Ordering	Order by timestamps	May abort late transactions
Multiversion CC (MVCC)	Readers get snapshots	Used in PostgreSQL, InnoDB
Optimistic CC (OCC)	Validate at commit	Best for low conflict workloads

### Tiny Code: Timestamp Ordering

```
# Simplified
if write_ts[x] > txn_ts or read_ts[x] > txn_ts:
    abort()
else:
    write_ts[x] = txn_ts
```

Each object tracks read and write timestamps.

## Deadlocks

Circular waits among transactions.

Detection | Build Wait-For Graph, detect cycle |

Prevention | Wait-Die (old waits) / Wound-Wait (young aborts) |

Detection Complexity:  $O(V + E)$

Tiny Code (Wait-For Graph Cycle Check):

```
def has_cycle(graph):
    visited, stack = set(), set()
    def dfs(u):
        visited.add(u)
        stack.add(u)
        for v in graph[u]:
            if v not in visited and dfs(v): return True
            if v in stack: return True
        stack.remove(u)
    return False
return any(dfs(u) for u in graph)
```

## Logging and Recovery

Technique	Description	Notes
Write-Ahead Log	Log before data	Ensures durability
ARIES	Analysis, Redo, Undo phases	Industry standard
Checkpointing	Save consistent snapshot	Speeds recovery
Shadow Paging	Copy-on-write updates	Simpler but less flexible

Recovery after crash:

1. Analysis: find active transactions
2. Redo: reapply committed changes
3. Undo: revert uncommitted ones

## Indexing

Accelerates lookups and range queries.

Index Type	Description	Notes
B-Tree / B+Tree	Balanced multiway tree	Disk-friendly
Hash Index	Exact match only	No range queries
GiST / R-Tree	Spatial data	Bounding box hierarchy
Inverted Index	Text search	Maps token to document list

B+Tree Complexity:  $O(\log_B N)$  (B = branching factor)

Tiny Code (Binary Search in Index):

```
def search(node, key):
    i = bisect_left(node.keys, key)
    if i < len(node.keys) and node.keys[i] == key:
        return node.values[i]
    if node.is_leaf:
        return None
    return search(node.children[i], key)
```

## Query Processing

Step	Description
Parsing	Build abstract syntax tree
Optimization	Reorder joins, pick indices
Execution Plan	Choose algorithm per operator
Execution	Evaluate iterators or pipelines

Common join strategies:

Join Type	Complexity	Notes
Nested Loop	$O(nm)$	Simple, slow
Hash Join	$O(n + m)$	Build + probe
Sort-Merge Join	$O(n \log n + m \log m)$	Sorted inputs

## Caching and Replacement

Policy	Description	Notes
LRU	Evict least recently used	Simple, temporal locality
LFU	Evict least frequently used	Good for stable patterns
ARC / LIRS	Adaptive hybrid	Handles mixed workloads
Random	Random eviction	Simple, fair

Tiny Code (LRU using OrderedDict):

```
from collections import OrderedDict

class LRU:
    def __init__(self, cap):
        self.cap = cap
        self.cache = OrderedDict()
    def get(self, k):
        if k not in self.cache: return -1
        self.cache.move_to_end(k)
        return self.cache[k]
    def put(self, k, v):
        if k in self.cache: self.cache.move_to_end(k)
        self.cache[k] = v
        if len(self.cache) > self.cap: self.cache.popitem(last=False)
```

## Distributed Systems Core

Problem	Description	Typical Solution
Consensus	Agree on value across nodes	Paxos, Raft
Leader Election	Pick coordinator	Bully, Raft
Replication	Maintain copies	Log replication
Partitioning	Split data	Consistent hashing
Membership	Detect nodes	Gossip protocols

## Raft Consensus (Simplified)

Phase	Action
Election	Nodes vote, elect leader
Replication	Leader appends log entries

Phase	Action
Commitment	Once majority acknowledge

Safety: Committed entries never change. Liveness: New leader elected on failure.

Tiny Code Sketch:

```
if vote_request.term > term:
    term = vote_request.term
    voted_for = candidate
```

### Consistent Hashing

Distributes keys across nodes smoothly.

Step	Description
Hash each node to ring	e.g. hash(node_id)
Hash each key	Find next node clockwise
Add/remove node	Only nearby keys move

Used in: Dynamo, Cassandra, Memcached.

### Fault Tolerance Patterns

Pattern	Description	Example
Replication	Multiple copies	Primary-backup
Checkpointing	Save progress periodically	ML training
Heartbeats	Liveness detection	Cluster managers
Retry + Backoff	Handle transient failures	API calls
Quorum Reads/Writes	Require majority agreement	Cassandra

### Distributed Coordination

Tool / Protocol	Description	Example Use
ZooKeeper	Centralized coordination	Locks, config
Raft	Distributed consensus	Log replication

Tool / Protocol	Description	Example Use
Etcd	Key-value store on Raft	Cluster metadata

**Summary Table**

Topic	Algorithm / Concept	Complexity	Notes
Locking	2PL, MVCC, OCC	varies	Transaction isolation
Deadlock	Wait-Die, Detection	$O(V + E)$	Graph-based check
Recovery	ARIES, WAL	varies	Crash recovery
Indexing	B+Tree, Hash Index	$O(\log N)$	Faster queries
Join	Hash / Sort-Merge	varies	Query optimization
Cache	LRU, LFU	$O(1)$	Data locality
Consensus	Raft, Paxos	$O(n)$ msg	Fault tolerance
Partitioning	Consistent Hashing	$O(1)$ avg	Scalability

### Quick Tips

- Always ensure serializability in concurrency.
- Use MVCC for read-heavy workloads.
- ARIES ensures durability via WAL.
- For scalability, partition and replicate wisely.
- Consensus is required for shared state correctness.

## Page 12. Algorithms for AI, ML, and Optimization Quick Use

This page gathers classical algorithms that power modern AI and machine learning systems, from clustering and classification to gradient-based learning and metaheuristics.

### Classical Machine Learning Algorithms

Category	Algorithm	Core Idea	Complexity
Clustering	k-Means	Assign to nearest centroid, update centers	$O(nkt)$
Clustering	k-Medoids (PAM)	Representative points as centers	$O(k(n - k)^2)$
Clustering	Gaussian Mixture (EM)	Soft assignments via probabilities	$O(nkd)$ per iter

Category	Algorithm	Core Idea	Complexity
Classification	Naive Bayes	Apply Bayes rule with feature independence	$O(nd)$
Classification	Logistic Regression	Linear + sigmoid activation	$O(nd)$
Classification	SVM (Linear)	Maximize margin via convex optimization	$O(nd)$ approx
Classification	k-NN	Vote from nearest neighbors	$O(nd)$ per query
Trees	Decision Tree (CART)	Recursive splitting by impurity	$O(nd \log n)$
Projection	LDA / PCA	Find projection maximizing variance or class separation	$O(d^3)$

### Tiny Code: k-Means

```

import random, math

def kmeans(points, k, iters=100):
    centroids = random.sample(points, k)
    for _ in range(iters):
        groups = [[] for _ in range(k)]
        for p in points:
            idx = min(range(k), key=lambda i: (p[0]-centroids[i][0])**2 + (p[1]-centroids[i][1])**2)
            groups[idx].append(p)
        new_centroids = []
        for g in groups:
            if g:
                x = sum(p[0] for p in g)/len(g)
                y = sum(p[1] for p in g)/len(g)
                new_centroids.append((x,y))
            else:
                new_centroids.append(random.choice(points))
        if centroids == new_centroids: break
        centroids = new_centroids
    return centroids

```

### Linear Models

Model	Formula	Loss Function
Linear Regression	$\hat{y} = w^T x + b$	MSE: $\frac{1}{n} \sum (y - \hat{y})^2$
Logistic Regression	$\hat{y} = \sigma(w^T x + b)$	Cross-Entropy
Ridge Regression	Linear + $L_2$ penalty	$L = \text{MSE} + \lambda  w ^2$
Lasso Regression	Linear + $L_1$ penalty	$L = \text{MSE} + \lambda  w _1$

Tiny Code (Gradient Descent for Linear Regression):

```
def train(X, y, lr=0.01, epochs=1000):
    w = [0]*len(X[0])
    b = 0
    for _ in range(epochs):
        for i in range(len(y)):
            y_pred = sum(w[j]*X[i][j] for j in range(len(w))) + b
            err = y_pred - y[i]
            for j in range(len(w)):
                w[j] -= lr * err * X[i][j]
            b -= lr * err
    return w, b
```

## Decision Trees and Ensembles

Algorithm	Description	Notes
ID3 / C4.5 / CART	Split by info gain or Gini	Recursive, interpretable
Random Forest	Bagging + Decision Trees	Reduces variance
Gradient Boosting	Sequential residual fitting	XGBoost, LightGBM, CatBoost
AdaBoost	Weighted weak learners	Sensitive to noise

Impurity Measures:

- Gini:  $1 - \sum p_i^2$
- Entropy:  $-\sum p_i \log_2 p_i$

## Support Vector Machines (SVM)

Finds a maximum margin hyperplane.

Objective:

$$\min_{w,b} \frac{1}{2} |w|^2 + C \sum \xi_i$$

subject to  $y_i(w^T x_i + b) \geq 1 - \xi_i$

Kernel trick enables nonlinear separation:

$$K(x_i, x_j) = \phi(x_i) \cdot \phi(x_j)$$

## Neural Network Fundamentals

Component	Description
Neuron	$y = \sigma(w \cdot x + b)$
Activation	Sigmoid, ReLU, Tanh
Loss	MSE, Cross-Entropy
Training	Gradient Descent + Backprop
Optimizers	SGD, Adam, RMSProp

Forward Propagation:

$$a^{(l)} = \sigma(W^{(l)}a^{(l-1)} + b^{(l)})$$

Backpropagation computes gradients layer by layer.

## Gradient Descent Variants

Variant	Idea	Notes
Batch	Use all data each step	Stable but slow
Stochastic	Update per sample	Noisy, fast
Mini-batch	Group updates	Common practice
Momentum	Add velocity term	Faster convergence
Adam	Adaptive moment estimates	Most popular

Update Rule:

$$w = w - \eta \cdot \frac{\partial L}{\partial w}$$

## Unsupervised Learning

Algorithm	Description	Notes
PCA	Variance-based projection	Eigen decomposition
ICA	Independent components	Signal separation

Algorithm	Description	Notes
t-SNE	Preserve local structure	Visualization only
Autoencoder	NN reconstruction model	Dimensionality red.

PCA Formula: Covariance  $C = \frac{1}{n}X^T X$ , eigenvectors of  $C$  are principal axes.

## Probabilistic Models

Model	Description	Notes
Naive Bayes	Independence assumption	$P(y x) \propto P(y) \prod P(x_i y)$
HMM	Sequential hidden states	Viterbi for decoding
Markov Chains	Transition probabilities	$P(x_t x_{t-1})$
Gaussian Mixture	Soft clustering	EM algorithm

## Optimization and Metaheuristics

Algorithm	Category	Notes
Gradient Descent	Convex Opt.	Differentiable objectives
Newton's Method	Second-order	Uses Hessian
Simulated Annealing	Prob. search	Escape local minima
Genetic Algorithm	Evolutionary	Population-based search
PSO (Swarm)	Collective move	Inspired by flocking behavior
Hill Climbing	Greedy search	Local optimization

## Reinforcement Learning Core

Concept	Description	Example
Agent	Learner/decision maker	Robot, policy
Environment	Provides states, rewards	Game, simulation
Policy	Mapping state $\rightarrow$ action	$\pi(s) = a$
Value Function	Expected return	$V(s), Q(s, a)$

Q-Learning Update:

$$Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a))$$

Tiny Code:

```
Q[s][a] += alpha * (r + gamma * max(Q[s_next]) - Q[s][a])
```

## AI Search Algorithms

Algorithm	Description	Complexity	Notes
BFS	Shortest path unweighted	$O(V + E)$	Level order search
DFS	Deep exploration	$O(V + E)$	Backtracking
A* Search	Informed, uses heuristic	$O(E \log V)$	$f(n) = g(n) + h(n)$
IDA*	Iterative deepening A*	Memory efficient	Optimal if $h$ admissible
Beam Search	Keep best k states	Approximate	NLP decoding

## Evaluation Metrics

Task	Metric	Formula / Meaning
Classification	Accuracy, Precision, Recall	$\frac{TP}{TP+FP}$ , $\frac{TP}{TP+FN}$
Regression	RMSE, MAE, $R^2$	Fit and error magnitude
Clustering	Silhouette Score	Cohesion vs separation
Ranking	MAP, NDCG	Order-sensitive

Confusion Matrix:

		Pred +	Pred -
Actual +	TP	FN	
	FP	TN	

## Summary

Category	Algorithm Example	Notes
Clustering	k-Means, GMM	Unsupervised grouping
Classification	Logistic, SVM, Trees	Supervised labeling
Regression	Linear, Ridge, Lasso	Predict continuous value
Optimization	GD, Adam, Simulated Annealing	Minimize loss
Probabilistic	Bayes, HMM, EM	Uncertainty modeling
Reinforcement	Q-Learning, SARSA	Reward-based learning

# The Book

- [Download PDF](#) - print-ready
- [Download EPUB](#) - e-reader friendly
- [View LaTex](#) - `.tex` source
- [Source code \(Github\)](#) - Markdown source
- [Read on GitHub Pages](#) - view online

Licensed under CC BY-NC-SA 4.0.

## Chapter 1. Foundations of algorithms

### 1. What Is an Algorithm?

Let's start at the beginning. Before code, data, or performance, we need a clear idea of what an algorithm really is.

An algorithm is a clear, step-by-step procedure to solve a problem. Think of it like a recipe: you have inputs (ingredients), a series of steps (instructions), and an output (the finished dish).

At its core, an algorithm should be:

- Precise: every step is well defined and unambiguous
- Finite: it finishes after a limited number of steps
- Effective: each step is simple enough to carry out
- Deterministic (usually): the same input gives the same output

When you write an algorithm, you are describing how to get from question to answer, not just what the answer is.

### **Example: Sum from 1 to (n)**

Suppose you want the sum of the numbers from 1 to (n).

Natural language steps

1. Set `total = 0`
2. For each `i` from 1 to `n`, add `i` to `total`
3. Return `total`

Pseudocode

```
Algorithm SumToN(n):
    total ← 0
    for i ← 1 to n:
        total ← total + i
    return total
```

C code

```
int sum_to_n(int n) {
    int total = 0;
    for (int i = 1; i <= n; i++) {
        total += i;
    }
    return total;
}
```

### **Tiny Code**

Try a quick run by hand with ( $n = 5$ ):

- start `total = 0`
- add 1 → `total = 1`
- add 2 → `total = 3`
- add 3 → `total = 6`
- add 4 → `total = 10`
- add 5 → `total = 15`

Output is 15.

You will also see this closed-form formula soon:

$$1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2}$$

## Why It Matters

Algorithms are the blueprints of computation. Every program, from a calculator to an AI model, is built from algorithms. Computers are fast at following instructions. Algorithms give those instructions structure and purpose.

Algorithms are the language of problem solving.

## Try It Yourself

1. Write an algorithm to find the maximum number in a list
2. Write an algorithm to reverse a string
3. Describe your morning routine as an algorithm: list the inputs, the steps, and the final output

Tip: the best way to learn is to think in small, clear steps. Break a problem into simple actions you can execute one by one.

## 2. Measuring Time and Space

Now that you know what an algorithm is, it's time to ask a deeper question:

How do we know if one algorithm is better than another?

It's not enough for an algorithm to be correct. It should also be efficient. We measure efficiency in two key ways: time and space.

### Time Complexity

Time measures how long an algorithm takes to run, relative to its input size. We don't measure in seconds, because hardware speed varies. Instead, we count steps or operations.

Example:

```
for (int i = 0; i < n; i++) {  
    printf("Hi\n");  
}
```

This loop runs  $n$  times, so it has time complexity  $O(n)$ . The time grows linearly with input size.

Another example:

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        printf("*");
```

This runs  $n \times n = n^2$  times, so it has  $O(n^2)$  time complexity.

These Big-O symbols describe how runtime grows as the input grows.

## Space Complexity

Space measures how much memory an algorithm uses.

Example:

```
int sum = 0; // O(1) space
```

This uses a constant amount of memory, regardless of input size.

But if we allocate an array:

```
int arr[n]; // O(n) space
```

This uses space proportional to  $n$ .

Often, we trade time for space:

- Using a hash table speeds up lookups (more memory, less time)
- Using a streaming algorithm saves memory (less space, more time)

## Tiny Code

Compare two ways to compute the sum from 1 to  $n$ :

Method 1: Loop

```
int sum_loop(int n) {
    int total = 0;
    for (int i = 1; i <= n; i++) total += i;
    return total;
}
```

Time:  $O(n)$  Space:  $O(1)$

Method 2: Formula

```
int sum_formula(int n) {
    return n * (n + 1) / 2;
}
```

Time:  $O(1)$  Space:  $O(1)$

Both are correct, but one is faster. Analyzing time and space helps you understand why.

## Why It Matters

When data grows huge (millions or billions), small inefficiencies explode.

An algorithm that takes  $O(n^2)$  time might feel fine for 10 elements, but impossible for 1,000,000.

Measuring time and space helps you:

- Predict performance
- Compare different solutions
- Optimize intelligently

It's your compass for navigating complexity.

## Try It Yourself

1. Write a simple algorithm to find the minimum in an array. Estimate its time and space complexity.
2. Compare two algorithms that solve the same problem. Which one scales better?
3. Think of a daily task that feels like  $O(n)$ . Can you imagine one that's  $O(1)$ ?

Understanding these measurements early makes every future algorithm more meaningful.

## 3. Big-O, Big-Theta, Big-Omega

Now that you can measure time and space, let's learn the language used to describe those measurements.

When we say an algorithm is  $O(n)$ , we're using asymptotic notation, a way to describe how an algorithm's running time or memory grows as input size  $n$  increases.

It's not about exact steps, but about how the cost scales for very large  $n$ .

## The Big-O (Upper Bound)

Big-O answers the question: “*How bad can it get?*” It gives an upper bound on growth, the worst-case scenario.

If an algorithm takes at most  $5n + 20$  steps, we write  $O(n)$ . We drop constants and lower-order terms because they don’t matter at scale.

Common Big-O notations:

Name	Notation	Growth	Example
Constant	$O(1)$	Flat	Accessing array element
Logarithmic	$O(\log n)$	Very slow growth	Binary search
Linear	$O(n)$	Proportional	Single loop
Quadratic	$O(n^2)$	Grows quickly	Double loop
Exponential	$O(2^n)$	Explodes	Recursive subset generation

If your algorithm is  $O(n)$ , doubling input size roughly doubles runtime. If it’s  $O(n^2)$ , doubling input size makes it about four times slower.

## The Big-Theta (Tight Bound)

Big-Theta ( $\Theta$ ) gives a tight bound, when you know the algorithm’s growth from above and below.

If runtime is roughly  $3n + 2$ , then  $T(n) = \Theta(n)$ . That means it’s both  $O(n)$  and  $\Omega(n)$ .

## The Big-Omega (Lower Bound)

Big-Omega ( $\Omega$ ) answers: “*How fast can it possibly be?*” It’s the best-case growth, the lower limit.

Example:

- Linear search:  $\Omega(1)$  if the element is at the start
- $O(n)$  in the worst case if it’s at the end

So we might say:

$$T(n) = \Omega(1), \quad T(n) = O(n)$$

## Tiny Code

Let's see Big-O in action.

```
int sum_pairs(int n) {
    int total = 0;
    for (int i = 0; i < n; i++)          // O(n)
        for (int j = 0; j < n; j++)      // O(n)
            total += i + j;             // O(1)
    return total;
}
```

Total steps  $n \times n = n^2$ . So  $T(n) = O(n^2)$ .

If we added a constant-time operation before or after the loops, it wouldn't matter. Constants vanish in asymptotic notation.

## Why It Matters

Big-O, Big-Theta, and Big-Omega let you talk precisely about performance. They are the grammar of efficiency.

When you can write:

Algorithm A runs in  $O(n \log n)$  time,  $O(n)$  space

you've captured its essence clearly and compared it meaningfully.

They help you:

- Predict behavior at scale
- Choose better data structures
- Communicate efficiency in interviews and papers

It's not about exact timing, it's about growth.

## Try It Yourself

1. Analyze this code:

```
for (int i = 1; i <= n; i *= 2)
    printf("%d", i);
```

What's the time complexity?

2. Write an algorithm that's  $O(n \log n)$  (hint: merge sort).
3. Identify the best, worst, and average-case complexities for linear search and binary search.

Learning Big-O is like learning a new language, once you're fluent, you can see how code grows before you even run it.

## 4. Algorithmic Paradigms (Greedy, Divide and Conquer, DP)

Once you can measure performance, it's time to explore how algorithms are designed. Behind every clever solution is a guiding paradigm, a way of thinking about problems.

Three of the most powerful are:

1. Greedy Algorithms
2. Divide and Conquer
3. Dynamic Programming (DP)

Each represents a different mindset for problem solving.

### 1. Greedy Algorithms

A greedy algorithm makes the best local choice at each step, hoping it leads to a global optimum.

Think of it like:

“Take what looks best right now, and don't worry about the future.”

They are fast and simple, but not always correct. They only work when the greedy choice property holds.

Example: Coin Change (Greedy version) Suppose you want to make 63 cents using US coins (25, 10, 5, 1). The greedy approach:

- Take 25 → 38 left
- Take 25 → 13 left
- Take 10 → 3 left
- Take 1 × 3

This works here, but not always (try coins 1, 3, 4 for amount 6). Simple, but not guaranteed optimal.

Common greedy algorithms:

- Kruskal's Minimum Spanning Tree

- Prim's Minimum Spanning Tree
- Dijkstra's Shortest Path (non-negative weights)
- Huffman Coding

## 2. Divide and Conquer

This is a classic paradigm. You break the problem into smaller subproblems, solve each recursively, and then combine the results.

It's like splitting a task among friends, then merging their answers.

Formally:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Examples:

- Merge Sort: divide the array, sort halves, merge
- Quick Sort: partition around a pivot
- Binary Search: halve the range each step

Elegant and powerful, but recursion overhead can add cost if poorly structured.

## 3. Dynamic Programming (DP)

DP is for problems with overlapping subproblems and optimal substructure. You solve smaller subproblems once and store the results to avoid recomputation.

It's like divide and conquer with memory.

Example: Fibonacci Naive recursion is exponential. DP with memoization is linear.

```
int fib(int n) {
    if (n <= 1) return n;
    static int memo[1000] = {0};
    if (memo[n]) return memo[n];
    memo[n] = fib(n-1) + fib(n-2);
    return memo[n];
}
```

Efficient reuse, but requires insight into subproblem structure.

## Tiny Code

Quick comparison using Fibonacci:

Naive (Divide and Conquer)

```
int fib_dc(int n) {
    if (n <= 1) return n;
    return fib_dc(n-1) + fib_dc(n-2); // exponential
}
```

DP (Memoization)

```
int fib_dp(int n, int memo[]) {
    if (n <= 1) return n;
    if (memo[n]) return memo[n];
    return memo[n] = fib_dp(n-1, memo) + fib_dp(n-2, memo);
}
```

## Why It Matters

Algorithmic paradigms give you patterns for design:

- Greedy: when local choices lead to a global optimum
- Divide and Conquer: when the problem splits naturally
- Dynamic Programming: when subproblems overlap

Once you recognize a problem's structure, you'll instantly know which mindset fits best.

Think of paradigms as templates for reasoning, not just techniques but philosophies.

## Try It Yourself

1. Write a greedy algorithm to make change using coins [1, 3, 4] for amount 6. Does it work?
2. Implement merge sort using divide and conquer.
3. Solve Fibonacci both ways (naive vs DP) and compare speeds.
4. Think of a real-life task you solve greedily.

Learning paradigms is like learning styles of thought. Once you know them, every problem starts to look familiar.

## 5. Recurrence Relations

Every time you break a problem into smaller subproblems, you create a recurrence, a mathematical way to describe how the total cost grows.

Recurrence relations are the backbone of analyzing recursive algorithms. They tell us how much time or space an algorithm uses, based on the cost of its subproblems.

### What Is a Recurrence?

A recurrence relation expresses  $T(n)$ , the total cost for input size  $n$ , in terms of smaller instances.

Example (Merge Sort):

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

That means:

- It divides the problem into 2 halves ( $2T(n/2)$ )
- Merges results in  $O(n)$  time

You will often see recurrences like:

- $T(n) = T(n - 1) + O(1)$
- $T(n) = 2T(n/2) + O(n)$
- $T(n) = T(n/2) + O(1)$

Each one represents a different structure of recursion.

### Example 1: Simple Linear Recurrence

Consider this code:

```
int count_down(int n) {
    if (n == 0) return 0;
    return 1 + count_down(n - 1);
}
```

This calls itself once for each smaller input:

$$T(n) = T(n - 1) + O(1)$$

Solve it:

$$T(n) = O(n)$$

Because it runs once per level.

### Example 2: Binary Recurrence

For binary recursion:

```
int sum_tree(int n) {
    if (n == 1) return 1;
    return sum_tree(n/2) + sum_tree(n/2) + 1;
}
```

Here we do two subcalls on  $n/2$  and a constant amount of extra work:

$$T(n) = 2T(n/2) + O(1)$$

Solve it:  $T(n) = O(n)$

Why? Each level doubles the number of calls but halves the size. There are  $\log n$  levels, and total work adds up to  $O(n)$ .

## Solving Recurrences

There are several ways to solve them:

- Substitution Method Guess the solution, then prove it by induction.
- Recursion Tree Method Expand the recurrence into a tree and sum the cost per level.
- Master Theorem Use a formula when the recurrence matches:

$$T(n) = aT(n/b) + f(n)$$

## Master Theorem (Quick Summary)

If  $T(n) = aT(n/b) + f(n)$ , then:

- If  $f(n) = O(n^{\log_b a - \epsilon})$ , then  $T(n) = \Theta(n^{\log_b a})$
- If  $f(n) = \Theta(n^{\log_b a})$ , then  $T(n) = \Theta(n^{\log_b a} \log n)$
- If  $f(n) = \Omega(n^{\log_b a + \epsilon})$ , and the regularity condition holds, then  $T(n) = \Theta(f(n))$

Example (Merge Sort):  $a = 2$ ,  $b = 2$ ,  $f(n) = O(n)$

$$T(n) = 2T(n/2) + O(n) = O(n \log n)$$

## Tiny Code

Let's write a quick recursive sum:

```
int sum_array(int arr[], int l, int r) {
    if (l == r) return arr[l];
    int mid = (l + r) / 2;
    return sum_array(arr, l, mid) + sum_array(arr, mid+1, r);
}
```

Recurrence:

$$T(n) = 2T(n/2) + O(1)$$

$$\rightarrow O(n)$$

If you added merging (like in merge sort), you would get  $+O(n)$ :

$$\rightarrow O(n \log n)$$

## Why It Matters

Recurrence relations let you predict the cost of recursive solutions.

Without them, recursion feels like magic. With them, you can quantify efficiency.

They are key to understanding:

- Divide and Conquer
- Dynamic Programming
- Backtracking

Once you can set up a recurrence, solving it becomes a game of algebra and logic.

### Try It Yourself

1. Write a recurrence for binary search. Solve it.
2. Write a recurrence for merge sort. Solve it.
3. Analyze this function:

```
void fun(int n) {  
    if (n <= 1) return;  
    fun(n/2);  
    fun(n/3);  
    fun(n/6);  
}
```

What's the recurrence? Approximate the complexity.

4. Expand  $T(n) = T(n - 1) + 1$  into its explicit sum.

Learning recurrences helps you see inside recursion. They turn code into equations.

## 6. Searching Basics

Before we sort or optimize, we need a way to find things. Searching is one of the most fundamental actions in computing, whether it's looking up a name, finding a key, or checking if something exists.

A search algorithm takes a collection (array, list, tree, etc.) and a target, and returns whether the target is present (and often its position).

Let's begin with two foundational techniques: Linear Search and Binary Search.

### 1. Linear Search

Linear search is the simplest method:

- Start at the beginning
- Check each element in turn
- Stop if you find the target

It works on any list, sorted or not, but can be slow for large data.

```
int linear_search(int arr[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key) return i;
    }
    return -1;
}
```

Example: If `arr` = [2, 4, 6, 8, 10] and `key` = 6, it finds it at index 2.

Complexity:

- Time:  $O(n)$
- Space:  $O(1)$

Linear search is simple and guaranteed to find the target if it exists, but slow when lists are large.

## 2. Binary Search

When the list is sorted, we can do much better. Binary search repeatedly divides the search space in half.

Steps:

1. Check the middle element
2. If it matches, you're done
3. If `target < mid`, search the left half
4. Else, search the right half

```
int binary_search(int arr[], int n, int key) {
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] == key) return mid;
        else if (arr[mid] < key) low = mid + 1;
        else high = mid - 1;
    }
    return -1;
}
```

Example: `arr` = [2, 4, 6, 8, 10], `key` = 8

- $\text{mid} = 6 \rightarrow \text{key} > \text{mid} \rightarrow$  search right half
- $\text{mid} = 8 \rightarrow$  found

Complexity:

- Time:  $O(\log n)$
- Space:  $O(1)$

Binary search is a massive improvement; doubling input only adds one extra step.

### 3. Recursive Binary Search

Binary search can also be written recursively:

```
int binary_search_rec(int arr[], int low, int high, int key) {
    if (low > high) return -1;
    int mid = (low + high) / 2;
    if (arr[mid] == key) return mid;
    else if (arr[mid] > key) return binary_search_rec(arr, low, mid - 1, key);
    else return binary_search_rec(arr, mid + 1, high, key);
}
```

Same logic, different structure. Both iterative and recursive forms are equally efficient.

### 4. Choosing Between Them

Method	Works On	Time	Space	Needs Sorting
Linear Search	Any list	$O(n)$	$O(1)$	No
Binary Search	Sorted list	$O(\log n)$	$O(1)$	Yes

If data is unsorted or very small, linear search is fine. If data is sorted and large, binary search is far superior.

### Tiny Code

Compare the steps: For  $n = 16$ :

- Linear search  $\rightarrow$  up to 16 comparisons
- Binary search  $\rightarrow \log_2 16 = 4$  comparisons

That's a huge difference.

## Why It Matters

Searching is the core of information retrieval. Every database, compiler, and system relies on it.

Understanding simple searches prepares you for:

- Hash tables (constant-time lookups)
- Tree searches (ordered structures)
- Graph traversals (structured exploration)

It's not just about finding values; it's about learning how data structure and algorithm design fit together.

## Try It Yourself

1. Write a linear search that returns all indices where a target appears.
2. Modify binary search to return the first occurrence of a target in a sorted array.
3. Compare runtime on arrays of size 10, 100, 1000.
4. What happens if you run binary search on an unsorted list?

Search is the foundation. Once you master it, you'll recognize its patterns everywhere.

## 7. Sorting Basics

Sorting is one of the most studied problems in computer science. Why? Because order matters. It makes searching faster, patterns clearer, and data easier to manage.

A sorting algorithm arranges elements in a specific order (usually ascending or descending). Once sorted, many operations (like binary search, merging, or deduplication) become much simpler.

Let's explore the foundational sorting methods and the principles behind them.

### 1. What Makes a Sort Algorithm

A sorting algorithm should define:

- Input: a sequence of elements
- Output: the same elements, in sorted order
- Stability: keeps equal elements in the same order (important for multi-key sorts)
- In-place: uses only a constant amount of extra space

Different algorithms balance speed, memory, and simplicity.

## 2. Bubble Sort

Idea: repeatedly “bubble up” the largest element to the end by swapping adjacent pairs.

```
void bubble_sort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

Each pass moves the largest remaining item to its final position.

- Time:  $O(n^2)$
- Space:  $O(1)$
- Stable: Yes

Simple but inefficient for large data.

## 3. Selection Sort

Idea: repeatedly select the smallest element and put it in the correct position.

```
void selection_sort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int min_idx = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_idx]) min_idx = j;
        }
        int temp = arr[i];
        arr[i] = arr[min_idx];
        arr[min_idx] = temp;
    }
}
```

- Time:  $O(n^2)$
- Space:  $O(1)$

- Stable: No

Fewer swaps, but still quadratic in time.

#### 4. Insertion Sort

Idea: build the sorted list one item at a time, inserting each new item in the right place.

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

- Time:  $O(n^2)$  (best case  $O(n)$  when nearly sorted)
- Space:  $O(1)$
- Stable: Yes

Insertion sort is great for small or nearly sorted datasets. It is often used as a base in hybrid sorts like Timsort.

#### 5. Comparing the Basics

Algorithm	Best Case	Average Case	Worst Case	Stable	In-place
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	Yes
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	No	Yes
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	Yes

All three are quadratic in time, but Insertion Sort performs best on small or partially sorted data.

## Tiny Code

Quick check with `arr = [5, 3, 4, 1, 2]`:

Insertion Sort (step by step)

- Insert 3 before 5 → [3, 5, 4, 1, 2]
- Insert 4 → [3, 4, 5, 1, 2]
- Insert 1 → [1, 3, 4, 5, 2]
- Insert 2 → [1, 2, 3, 4, 5]

Sorted!

## Why It Matters

Sorting is a gateway algorithm. It teaches you about iteration, swapping, and optimization.

Efficient sorting is critical for:

- Preprocessing data for binary search
- Organizing data for analysis
- Building indexes and ranking systems

It's the first step toward deeper concepts like divide and conquer and hybrid optimization.

## Try It Yourself

1. Implement all three: bubble, selection, insertion
2. Test them on arrays of size 10, 100, 1000, and note timing differences
3. Try sorting an array that's already sorted. Which one adapts best?
4. Modify insertion sort to sort in descending order

Sorting may seem simple, but it's a cornerstone. Mastering it will shape your intuition for almost every other algorithm.

## 8. Data Structures Overview

Algorithms and data structures are two sides of the same coin. An algorithm is how you solve a problem. A data structure is where you store and organize data so that your algorithm can work efficiently.

You can think of data structures as containers, each one shaped for specific access patterns, trade-offs, and performance needs. Choosing the right one is often the key to designing a fast algorithm.

## 1. Why Data Structures Matter

Imagine you want to find a book quickly.

- If all books are piled randomly → you must scan every one ( $O(n)$ )
- If they're sorted on a shelf → you can use binary search ( $O(\log n)$ )
- If you have an index or catalog → you can find it instantly ( $O(1)$ )

Different structures unlock different efficiencies.

## 2. The Core Data Structures

Let's walk through the most essential ones:

Type	Description	Key Operations	Typical Use
Array	Fixed-size contiguous memory	Access ( $O(1)$ ), Insert/Delete ( $O(n)$ )	Fast index access
Linked List	Sequence of nodes with pointers	Insert/Delete ( $O(1)$ ), Access ( $O(n)$ )	Dynamic sequences
Stack	LIFO (last-in, first-out)	push(), pop() in $O(1)$	Undo, recursion
Queue	FIFO (first-in, first-out)	enqueue(), dequeue() in $O(1)$	Scheduling, buffers
Hash Table	Key-value pairs via hashing	Average $O(1)$ , Worst $O(n)$	Lookup, caching
Heap	Partially ordered tree	Insert $O(\log n)$ , Extract-Min $O(\log n)$	Priority queues
Tree	Hierarchical structure	Access $O(\log n)$ (balanced)	Sorted storage
Graph	Nodes + edges	Traversal $O(V + E)$	Networks, paths
Set / Map	Unique keys or key-value pairs	$O(\log n)$ or $O(1)$	Membership tests

Each comes with trade-offs. Arrays are fast but rigid, linked lists are flexible but slower to access, and hash tables are lightning-fast but unordered.

## 3. Abstract Data Types (ADTs)

An ADT defines what operations you can do, not how they're implemented. For example, a Stack ADT promises:

- `push(x)`

- `pop()`
- `peek()`

It can be implemented with arrays or linked lists, the behavior stays the same.

Common ADTs:

- Stack
- Queue
- Deque
- Priority Queue
- Map / Dictionary

This separation of interface and implementation helps design flexible systems.

#### 4. The Right Tool for the Job

Choosing the correct data structure often decides the performance of your algorithm:

Problem	Good Choice	Reason
Undo feature	Stack	LIFO fits history
Scheduling tasks	Queue	FIFO order
Dijkstra's algorithm	Priority Queue	Extract smallest distance
Counting frequencies	Hash Map	Fast key lookup
Dynamic median	Heap + Heap	Balance two halves
Search by prefix	Trie	Fast prefix lookups

Good programmers don't just write code. They pick the right structure.

#### Tiny Code

Example: comparing array vs linked list

Array:

```
int arr[5] = {1, 2, 3, 4, 5};
printf("%d", arr[3]); // O(1)
```

Linked List:

```
struct Node { int val; struct Node* next; };
```

To get the 4th element, you must traverse  $\rightarrow O(n)$

Different structures, different access costs.

## Why It Matters

Every efficient algorithm depends on the right data structure.

- Searching, sorting, and storing all rely on structure
- Memory layout affects cache performance
- The wrong choice can turn  $O(1)$  into  $O(n^2)$

Understanding these structures is like knowing the tools in a workshop. Once you recognize their shapes, you'll instinctively know which to grab.

## Try It Yourself

1. Implement a stack using an array. Then implement it using a linked list.
2. Write a queue using two stacks.
3. Try storing key-value pairs in a hash table (hint: mod by table size).
4. Compare access times for arrays vs linked lists experimentally.

Data structures aren't just storage. They are the skeletons your algorithms stand on.

## 9. Graphs and Trees Overview

Now that you've seen linear structures like arrays and linked lists, it's time to explore non-linear structures, graphs and trees. These are the shapes behind networks, hierarchies, and relationships.

They appear everywhere: family trees, file systems, maps, social networks, and knowledge graphs all rely on them.

## 1. Trees

A tree is a connected structure with no cycles. It's a hierarchy, and every node (except the root) has one parent.

- Root: the top node
- Child: a node directly connected below
- Leaf: a node with no children
- Height: the longest path from root to a leaf

A binary tree is one where each node has at most two children. A binary search tree (BST) keeps elements ordered:

- Left child < parent < right child

Basic operations:

- Insert
- Search
- Delete
- Traverse (preorder, inorder, postorder, level-order)

Example:

```
struct Node {  
    int val;  
    struct Node *left, *right;  
};
```

Insert in BST:

```
struct Node* insert(struct Node* root, int val) {  
    if (!root) return newNode(val);  
    if (val < root->val) root->left = insert(root->left, val);  
    else root->right = insert(root->right, val);  
    return root;  
}
```

## 2. Common Tree Types

Type	Description	Use Case
Binary Tree	Each node has 2 children	General hierarchy
Binary Search Tree	Left < Root < Right	Ordered data
AVL / Red-Black Tree	Self-balancing BST	Fast search/insert
Heap	Complete binary tree, parent or children	Priority queues
Trie	Tree of characters	Prefix search
Segment Tree	Tree over ranges	Range queries
Fenwick Tree	Tree with prefix sums	Efficient updates

Balanced trees keep height  $O(\log n)$ , guaranteeing fast operations.

### 3. Graphs

A graph generalizes the idea of trees. In graphs, nodes (vertices) can connect freely.

A graph is a set of vertices ( $V$ ) and edges ( $E$ ):

$$G = (V, E)$$

Directed vs Undirected:

- Directed: edges have direction (A → B)
- Undirected: edges connect both ways (A, B)

Weighted vs Unweighted:

- Weighted: each edge has a cost
- Unweighted: all edges are equal

Representation:

1. Adjacency Matrix:  $n \times n$  matrix; entry  $(i, j) = 1$  if edge exists
2. Adjacency List: array of lists; each vertex stores its neighbors

Example adjacency list:

```
vector<int> graph[n];
graph[0].push_back(1);
graph[0].push_back(2);
```

## 4. Common Graph Types

Graph Type	Description	Example
Undirected	Edges without direction	Friendship network
Directed	Arrows indicate direction	Web links
Weighted	Edges have costs	Road network
Cyclic	Contains loops	Task dependencies
Acyclic	No loops	Family tree
DAG (Directed Acyclic)	Directed, no cycles	Scheduling, compilers
Complete	All pairs connected	Dense networks
Sparse	Few edges	Real-world graphs

## 5. Basic Graph Operations

- Add Vertex / Edge
- Traversal: Depth-First Search (DFS), Breadth-First Search (BFS)
- Path Finding: Dijkstra, Bellman-Ford
- Connectivity: Union-Find, Tarjan (SCC)
- Spanning Trees: Kruskal, Prim

Each graph problem has its own flavor, from finding shortest paths to detecting cycles.

### Tiny Code

Breadth-first search (BFS):

```
void bfs(int start, vector<int> graph[], int n) {
    bool visited[n];
    memset(visited, false, sizeof(visited));
    queue<int> q;
    visited[start] = true;
    q.push(start);
    while (!q.empty()) {
        int node = q.front(); q.pop();
        printf("%d ", node);
        for (int neighbor : graph[node]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
            }
        }
    }
}
```

```
    }  
}  
}
```

This explores level by level, perfect for shortest paths in unweighted graphs.

## Why It Matters

Trees and graphs model relationships and connections, not just sequences. They are essential for:

- Search engines (web graph)
- Compilers (syntax trees, dependency DAGs)
- AI (state spaces, decision trees)
- Databases (indexes, joins, relationships)

Understanding them unlocks an entire world of algorithms, from DFS and BFS to Dijkstra, Kruskal, and beyond.

## Try It Yourself

1. Build a simple binary search tree and implement inorder traversal.
2. Represent a graph with adjacency lists and print all edges.
3. Write a DFS and BFS for a small graph.
4. Draw a directed graph with a cycle and detect it manually.

Graphs and trees move you beyond linear thinking. They let you explore *connections*, not just collections.

## 10. Algorithm Design Patterns

By now, you've seen what algorithms are and how they're analyzed. You've explored searches, sorts, structures, and recursion. The next step is learning patterns, reusable strategies that guide how you build new algorithms from scratch.

Just like design patterns in software architecture, algorithmic design patterns give structure to your thinking. Once you recognize them, many problems suddenly feel familiar.

## 1. Brute Force

Start simple. Try every possibility and pick the best result. Brute force is often your baseline, clear but inefficient.

Example: Find the maximum subarray sum by checking all subarrays.

- Time:  $O(n^2)$
- Advantage: easy to reason about
- Disadvantage: explodes for large input

Sometimes, brute force helps you see the structure needed for a better approach.

## 2. Divide and Conquer

Split the problem into smaller parts, solve each, and combine. Ideal for problems with self-similarity.

Classic examples:

- Merge Sort → split and merge
- Binary Search → halve the search space
- Quick Sort → partition and sort

General form:

$$T(n) = aT(n/b) + f(n)$$

Use recurrence relations and the Master Theorem to analyze them.

## 3. Greedy

Make the best local decision at each step. Works only when local optimal choices lead to a global optimum.

Examples:

- Activity Selection
- Huffman Coding
- Dijkstra (for non-negative weights)

Greedy algorithms are simple and fast when they fit.

## 4. Dynamic Programming (DP)

When subproblems overlap, store results and reuse them. Think recursion plus memory.

Two main styles:

- Top-Down (Memoization): recursive with caching
- Bottom-Up (Tabulation): iterative filling table

Used in:

- Fibonacci numbers
- Knapsack
- Longest Increasing Subsequence (LIS)
- Matrix Chain Multiplication

DP transforms exponential recursion into polynomial time.

## 5. Backtracking

Explore all possibilities, but prune when constraints fail. It is brute force with early exits.

Perfect for:

- N-Queens
- Sudoku
- Permutation generation
- Subset sums

Backtracking builds solutions incrementally, abandoning paths that cannot lead to a valid result.

## 6. Two Pointers

Move two indices through a sequence to find patterns or meet conditions.

Common use:

- Sorted arrays (sum pairs, partitions)
- String problems (palindromes, sliding windows)
- Linked lists (slow and fast pointers)

Simple, but surprisingly powerful.

## 7. Sliding Window

Maintain a window over data, expand or shrink it as needed.

Used for:

- Maximum sum subarray (Kadane's algorithm)
- Substrings of length  $k$
- Longest substring without repeating characters

Helps reduce  $O(n^2)$  to  $O(n)$  in sequence problems.

## 8. Binary Search on Answer

Sometimes, the input is not sorted, but the answer space is. If you can define a function `check(mid)` that is monotonic (true or false changes once), you can apply binary search on possible answers.

Examples:

- Minimum capacity to ship packages in D days
- Smallest feasible value satisfying a constraint

Powerful for optimization under monotonic conditions.

## 9. Graph-Based

Think in terms of nodes and edges, paths and flows.

Patterns include:

- BFS and DFS (exploration)
- Topological Sort (ordering)
- Dijkstra and Bellman-Ford (shortest paths)
- Union-Find (connectivity)
- Kruskal and Prim (spanning trees)

Graphs often reveal relationships hidden in data.

## 10. Meet in the Middle

Split the problem into two halves, compute all possibilities for each, and combine efficiently.  
Used in problems where brute force  $O(2^n)$  is too large but  $O(2^{n/2})$  is manageable.

Examples:

- Subset sum (divide into two halves)
- Search problems in combinatorics

A clever compromise between brute force and efficiency.

### Tiny Code

Example: Two Pointers to find a pair sum

```
int find_pair_sum(int arr[], int n, int target) {  
    int i = 0, j = n - 1;  
    while (i < j) {  
        int sum = arr[i] + arr[j];  
        if (sum == target) return 1;  
        else if (sum < target) i++;  
        else j--;  
    }  
    return 0;  
}
```

Works in  $O(n)$  for sorted arrays, elegant and fast.

### Why It Matters

Patterns are mental shortcuts. They turn “blank page” problems into “I’ve seen this shape before.”

Once you recognize the structure, you can choose a suitable pattern and adapt it. This is how top coders solve complex problems under time pressure, not by memorizing algorithms, but by seeing patterns.

## Try It Yourself

1. Write a brute-force and a divide-and-conquer solution for maximum subarray sum. Compare speed.
2. Solve the coin change problem using both greedy and DP.
3. Implement N-Queens with backtracking.
4. Use two pointers to find the smallest window with a given sum.
5. Pick a problem you've solved before. Can you reframe it using a different design pattern?

The more patterns you practice, the faster you will map new problems to known strategies, and the more powerful your algorithmic intuition will become.

## Chapter 2. Sorting and Searching

### 11. Elementary Sorting (Bubble, Insertion, Selection)

Before diving into advanced sorts like mergesort or heapsort, it's important to understand the elementary sorting algorithms , the building blocks. They're simple, intuitive, and great for learning how sorting works under the hood.

In this section, we'll cover three classics:

- Bubble Sort - swap adjacent out-of-order pairs- Selection Sort - select the smallest element each time- Insertion Sort - insert elements one by one in order These algorithms share (  $O(n^2)$  ) time complexity but differ in behavior and stability.

#### 1. Bubble Sort

Idea: Compare adjacent pairs and swap if they're out of order. Repeat until the array is sorted. Each pass “bubbles” the largest element to the end.

Steps:

1. Compare `arr[j]` and `arr[j+1]`
2. Swap if `arr[j] > arr[j+1]`
3. Continue passes until no swaps are needed

Code:

```

void bubble_sort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int swapped = 0;
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = 1;
            }
        }
        if (!swapped) break;
    }
}

```

Complexity:

- Best: ( O(n) ) (already sorted)- Worst: (  $O(n^2)$  )- Space: ( O(1) )- Stable: Yes Intuition: Imagine bubbles rising , after each pass, the largest “bubble” settles at the top.

## 2. Selection Sort

Idea: Find the smallest element and place it at the front.

Steps:

1. For each position  $i$ , find the smallest element in the remainder of the array
2. Swap it with  $\text{arr}[i]$

Code:

```

void selection_sort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int min_idx = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_idx])
                min_idx = j;
        }
        int temp = arr[i];
        arr[i] = arr[min_idx];
        arr[min_idx] = temp;
    }
}

```

Complexity:

- Best: (  $O(n)$  )- Worst: (  $O(n^2)$  )- Space: (  $O(1)$  )- Stable: No Intuition: Selection sort “selects” the next correct element and fixes it. It minimizes swaps but still scans all elements.

### 3. Insertion Sort

Idea: Build a sorted array one element at a time by inserting each new element into its correct position.

Steps:

1. Start from index 1
2. Compare with previous elements
3. Shift elements greater than key to the right
4. Insert key into the correct place

Code:

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

Complexity:

- Best: (  $O(n)$  ) (nearly sorted)- Worst: (  $O(n^2)$  )- Space: (  $O(1)$  )- Stable: Yes Intuition: It's like sorting cards in your hand , take the next card and slide it into the right place.

### 4. Comparing the Three

Algorithm	Best Case	Average Case	Worst Case	Stable	In-Place	Notes
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	Yes	Early exit possible
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	No	Yes	Few swaps
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	Yes	Great on small or nearly sorted data

## Tiny Code

Let's see how insertion sort works on [5, 3, 4, 1, 2]:

- Start with 3 → insert before 5 → [3, 5, 4, 1, 2]- Insert 4 → [3, 4, 5, 1, 2]- Insert 1 → [1, 3, 4, 5, 2]- Insert 2 → [1, 2, 3, 4, 5] Sorted in five passes.

## Why It Matters

Elementary sorts teach you:

- How comparisons and swaps drive order- The trade-off between simplicity and efficiency- How to reason about stability and adaptability While these aren't used for large datasets in practice, they're used *inside* hybrid algorithms like Timsort and IntroSort, which switch to insertion sort for small chunks.

## Try It Yourself

- Implement all three and print the array after each pass.
- Test on arrays: already sorted, reversed, random, partially sorted.
- Modify bubble sort to sort descending.
- Try insertion sort on 10,000 elements and note its behavior.
- Can you detect when the list is already sorted and stop early?

Start simple. Master these patterns. They'll be your foundation for everything from merge sort to radix sort.

## 12. Divide-and-Conquer Sorting (Merge, Quick, Heap)

Elementary sorts are great for learning, but their ( $O(n^2)$ ) runtime quickly becomes a bottleneck. To scale beyond small arrays, we need algorithms that divide problems into smaller parts, sort them independently, and combine the results.

This is the essence of divide and conquer , break it down, solve subproblems, merge solutions. In sorting, this approach yields some of the fastest general-purpose algorithms: Merge Sort, Quick Sort, and Heap Sort.

### 1. Merge Sort

Idea: Split the array in half, sort each half recursively, then merge the two sorted halves.

Merge sort is stable, works well with linked lists, and guarantees ( $O(n \log n)$ ) time.

Steps:

1. Divide the array into halves
2. Recursively sort each half
3. Merge two sorted halves into one

Code:

```
void merge(int arr[], int l, int m, int r) {  
    int n1 = m - l + 1;  
    int n2 = r - m;  
    int L[n1], R[n2];  
    for (int i = 0; i < n1; i++) L[i] = arr[l + i];  
    for (int j = 0; j < n2; j++) R[j] = arr[m + 1 + j];  
    int i = 0, j = 0, k = l;  
    while (i < n1 && j < n2) {  
        if (L[i] <= R[j]) arr[k++] = L[i++];  
        else arr[k++] = R[j++];  
    }  
    while (i < n1) arr[k++] = L[i++];  
    while (j < n2) arr[k++] = R[j++];  
}  
  
void merge_sort(int arr[], int l, int r) {  
    if (l < r) {  
        int m = (l + r) / 2;  
        merge_sort(arr, l, m);  
        merge_sort(arr, m + 1, r);  
    }  
}
```

```

        merge(arr, l, m, r);
    }
}

```

Complexity:

- Time:  $(O(n \log n))$  (always)- Space:  $(O(n))$  (temporary arrays)- Stable: Yes Merge sort is predictable, making it ideal for external sorting (like sorting data on disk).

## 2. Quick Sort

Idea: Pick a pivot, partition the array so smaller elements go left and larger go right, then recursively sort both sides.

Quick sort is usually the fastest in practice due to good cache locality and low constant factors.

Steps:

1. Choose a pivot (often middle or random)
2. Partition: move smaller elements to left, larger to right
3. Recursively sort the two partitions

Code:

```

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            int tmp = arr[i]; arr[i] = arr[j]; arr[j] = tmp;
        }
    }
    int tmp = arr[i + 1]; arr[i + 1] = arr[high]; arr[high] = tmp;
    return i + 1;
}

void quick_sort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quick_sort(arr, low, pi - 1);
        quick_sort(arr, pi + 1, high);
    }
}

```

```
    }
}
```

Complexity:

- Best / Average:  $(O(n \log n))$ - Worst:  $(O(n^2))$  (bad pivot, e.g. sorted input with naive pivot)- Space:  $(O(\log n))$  (recursion)- Stable: No (unless modified) Quick sort is often used in standard libraries due to its efficiency in real-world workloads.

### 3. Heap Sort

Idea: Turn the array into a heap, repeatedly extract the largest element, and place it at the end.

A heap is a binary tree where every parent is greater than its children (max-heap).

Steps:

1. Build a max-heap
2. Swap the root (max) with the last element
3. Reduce heap size, re-heapify
4. Repeat until sorted

Code:

```
void heapify(int arr[], int n, int i) {
    int largest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;
    if (l < n && arr[l] > arr[largest]) largest = l;
    if (r < n && arr[r] > arr[largest]) largest = r;
    if (largest != i) {
        int tmp = arr[i]; arr[i] = arr[largest]; arr[largest] = tmp;
        heapify(arr, n, largest);
    }
}

void heap_sort(int arr[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);
    for (int i = n - 1; i > 0; i--) {
        int tmp = arr[0]; arr[0] = arr[i]; arr[i] = tmp;
        heapify(arr, i, 0);
    }
}
```

```

    }
}

```

Complexity:

- Time: ( $O(n \log n)$ )- Space: ( $O(1)$ )- Stable: No Heap sort is reliable and space-efficient but less cache-friendly than quicksort.

#### 4. Comparison

Algorithm	Best Case	Average Case	Worst Case	Space	Stable	Notes
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes	Predictable, stable
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	No	Fast in practice
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	No	In-place, robust

Each one fits a niche:

- Merge Sort → stability and guarantees- Quick Sort → speed and cache performance- Heap Sort → low memory usage and simplicity

#### Tiny Code

Try sorting [5, 1, 4, 2, 8] with merge sort:

1. Split → [5,1,4], [2,8]
2. Sort each → [1,4,5], [2,8]
3. Merge → [1,2,4,5,8]

Each recursive split halves the problem, yielding ( $O(\log n)$ ) depth with ( $O(n)$ ) work per level.

#### Why It Matters

Divide-and-conquer sorting is the foundation for efficient order processing. It introduces ideas you'll reuse in:

- Binary search (halving)- Matrix multiplication- Fast Fourier Transform- Dynamic programming These sorts teach how recursion, partitioning, and merging combine into scalable solutions.

## Try It Yourself

1. Implement merge sort, quick sort, and heap sort.
2. Test all three on the same random array. Compare runtime.
3. Modify quick sort to use a random pivot.
4. Build a stable version of heap sort.
5. Visualize merge sort's recursion tree and merging process.

Mastering these sorts gives you a template for solving any divide-and-conquer problem efficiently.

## 13. Counting and Distribution Sorts (Counting, Radix, Bucket)

So far, we've seen comparison-based sorts like merge sort and quicksort. These rely on comparing elements and are bounded by the  $O(n \log n)$  lower limit for comparisons.

But what if you don't need to compare elements directly , what if they're integers or values from a limited range?

That's where counting and distribution sorts come in. They exploit structure, not just order, to achieve linear-time sorting in the right conditions.

### 1. Counting Sort

Idea: If your elements are integers in a known range ( $[0, k]$ ), you can count occurrences of each value, then reconstruct the sorted output.

Counting sort doesn't compare , it counts.

Steps:

1. Find the range of input (max value (k))
2. Count occurrences in a frequency array
3. Convert counts to cumulative counts
4. Place elements into their sorted positions

Code:

```
void counting_sort(int arr[], int n, int k) {  
    int count[k + 1];  
    int output[n];  
    for (int i = 0; i <= k; i++) count[i] = 0;  
    for (int i = 0; i < n; i++) count[arr[i]]++;  
    for (int i = 1; i <= k; i++) count[i] += count[i - 1];  
    for (int i = n - 1; i >= 0; i--) output[i] = arr[count[i] - 1];  
    for (int i = 0; i < n; i++) arr[i] = output[i];  
}
```

```

    for (int i = n - 1; i >= 0; i--) {
        output[count[arr[i]] - 1] = arr[i];
        count[arr[i]]--;
    }
    for (int i = 0; i < n; i++) arr[i] = output[i];
}

```

Example:  $\text{arr} = [4, 2, 2, 8, 3, 3, 1]$ ,  $k = 8 \rightarrow \text{count} = [0,1,2,2,1,0,0,0,1] \rightarrow \text{cumulative} = [0,1,3,5,6,6,6,6,7] \rightarrow \text{sorted} = [1,2,2,3,3,4,8]$

Complexity:

- Time:  $(O(n + k))$ - Space:  $(O(k))$ - Stable: Yes When to use:
- Input is integers- Range ( $k$ ) not much larger than ( $n$ )

## 2. Radix Sort

Idea: Sort digits one at a time, from least significant (LSD) or most significant (MSD), using a stable sub-sort like counting sort.

Radix sort works best when all elements have fixed-length representations (e.g., integers, strings of equal length).

Steps (LSD method):

1. For each digit position (from rightmost to leftmost)
2. Sort all elements by that digit using a stable sort (like counting sort)

Code:

```

int get_max(int arr[], int n) {
    int mx = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > mx) mx = arr[i];
    return mx;
}

void counting_sort_digit(int arr[], int n, int exp) {
    int output[n];
    int count[10] = {0};
    for (int i = 0; i < n; i++)
        count[(arr[i] / exp) % 10]++;
    for (int i = 1; i < 10; i++)

```

```

        count[i] += count[i - 1];
    for (int i = n - 1; i >= 0; i--) {
        int digit = (arr[i] / exp) % 10;
        output[count[digit] - 1] = arr[i];
        count[digit]--;
    }
    for (int i = 0; i < n; i++)
        arr[i] = output[i];
}

void radix_sort(int arr[], int n) {
    int m = get_max(arr, n);
    for (int exp = 1; m / exp > 0; exp *= 10)
        counting_sort_digit(arr, n, exp);
}

```

Example: arr = [170, 45, 75, 90, 802, 24, 2, 66] → sort by 1s → 10s → 100s → final = [2, 24, 45, 66, 75, 90, 170, 802]

Complexity:

- Time:  $(O(d \times (n + b)))$ , where
  - (d): number of digits - (b): base (10 for decimal)- Space:  $(O(n + b))$ - Stable: Yes When to use:
- Fixed-length numbers- Bounded digits (e.g., base 10 or 2)

### 3. Bucket Sort

Idea: Divide elements into buckets based on value ranges, sort each bucket individually, then concatenate.

Works best when data is uniformly distributed in a known interval.

Steps:

1. Create (k) buckets for value ranges
2. Distribute elements into buckets
3. Sort each bucket (often using insertion sort)
4. Merge buckets

Code:

```

void bucket_sort(float arr[], int n) {
    vector<float> buckets[n];
    for (int i = 0; i < n; i++) {
        int idx = n * arr[i]; // assuming 0 <= arr[i] < 1
        buckets[idx].push_back(arr[i]);
    }
    for (int i = 0; i < n; i++)
        sort(buckets[i].begin(), buckets[i].end());
    int idx = 0;
    for (int i = 0; i < n; i++)
        for (float val : buckets[i])
            arr[idx++] = val;
}

```

Complexity:

- Average:  $(O(n + k))$ - Worst:  $(On^2)$  (if all fall in one bucket)- Space:  $(O(n + k))$ - Stable: Depends on bucket sort method When to use:
- Real numbers uniformly distributed in  $([0,1])$

#### 4. Comparison

Algorithm	Time	Space	Stable	Type	Best Use
Counting Sort	$O(n + k)$	$O(k)$	Yes	Non-comparison	Small integer range
Radix Sort	$O(d(n + b))$	$O(n + b)$	Yes	Non-comparison	Fixed-length numbers
Bucket Sort	$O(n + k)$ avg	$O(n + k)$	Often	Distribution-based	Uniform floats

These algorithms achieve  $O(n)$  behavior when assumptions hold , they're specialized but incredibly fast when applicable.

#### Tiny Code

Let's walk counting sort on  $\text{arr} = [4, 2, 2, 8, 3, 3, 1]$ :

- Count occurrences  $\rightarrow [1,2,2,1,0,0,0,1]$ - Cumulative count  $\rightarrow$  positions- Place elements  $\rightarrow [1,2,2,3,3,4,8]$  Sorted , no comparisons.

## Why It Matters

Distribution sorts teach a key insight:

If you know the structure of your data, you can sort faster than comparison allows.

They show how data properties , range, distribution, digit length , can drive algorithm design.

You'll meet these ideas again in:

- Hashing (bucketing)- Indexing (range partitioning)- Machine learning (binning, histogramming)

## Try It Yourself

1. Implement counting sort for integers from 0 to 100.
2. Extend radix sort to sort strings by character.
3. Visualize bucket sort for values between 0 and 1.
4. What happens if you use counting sort on negative numbers? Fix it.
5. Compare counting vs quick sort on small integer arrays.

These are the first glimpses of linear-time sorting , harnessing knowledge about data to break the ( $O(n \log n)$ ) barrier.

## 14. Hybrid Sorts (IntroSort, Timsort)

In practice, no single sorting algorithm is perfect for all cases. Some are fast on average but fail in worst cases (like Quick Sort). Others are consistent but slow due to overhead (like Merge Sort). Hybrid sorting algorithms combine multiple techniques to get the *best of all worlds* , practical speed, stability, and guaranteed performance.

Two of the most widely used hybrids in modern systems are IntroSort and Timsort , both power the sorting functions in major programming languages.

### 1. The Idea Behind Hybrid Sorting

Real-world data is messy: sometimes nearly sorted, sometimes random, sometimes pathological. A smart sorting algorithm should adapt to the data.

Hybrids switch between different strategies based on:

- Input size- Recursion depth- Degree of order- Performance thresholds So, the algorithm “introspects” or “adapts” while running.

## 2. IntroSort

IntroSort (short for *introspective sort*) begins like Quick Sort, but when recursion gets too deep , which means Quick Sort's worst case may be coming , it switches to Heap Sort to guarantee ( $O(n \log n)$ ) time.

Steps:

1. Use Quick Sort as long as recursion depth  $< 2 \log n$
2. If depth exceeds limit  $\rightarrow$  switch to Heap Sort
3. For very small subarrays  $\rightarrow$  switch to Insertion Sort

This triple combo ensures:

- Fast average case (Quick Sort)- Guaranteed upper bound (Heap Sort)- Efficiency on small arrays (Insertion Sort) Code Sketch:

```
void intro_sort(int arr[], int n) {
    int depth_limit = 2 * log(n);
    intro_sort_util(arr, 0, n - 1, depth_limit);
}

void intro_sort_util(int arr[], int begin, int end, int depth_limit) {
    int size = end - begin + 1;
    if (size < 16) {
        insertion_sort(arr + begin, size);
        return;
    }
    if (depth_limit == 0) {
        heap_sort_range(arr, begin, end);
        return;
    }
    int pivot = partition(arr, begin, end);
    intro_sort_util(arr, begin, pivot - 1, depth_limit - 1);
    intro_sort_util(arr, pivot + 1, end, depth_limit - 1);
}
```

Complexity:

- Average: ( $O(n \log n)$ )- Worst: ( $O(n \log n)$ )- Space: ( $O(\log n)$ )- Stable: No (depends on partition scheme) Used in:
- C++ STL's `std::sort`- Many systems where performance guarantees matter

### 3. Timsort

Timsort is a stable hybrid combining Insertion Sort and Merge Sort. It was designed to handle real-world data, which often has runs (already sorted segments).

Developed by Tim Peters (Python core dev), Timsort is now used in:

- Python's `sorted()` and `.sort()`- Java's `Arrays.sort()` for objects Idea:
- Identify runs , segments already ascending or descending- Reverse descending runs (to make them ascending)- Sort small runs with Insertion Sort- Merge runs with Merge Sort Timsort adapts beautifully to partially ordered data.

Steps:

1. Scan array, detect runs (sequences already sorted)
2. Push runs to a stack
3. Merge runs using a carefully balanced merge strategy

Pseudocode (simplified):

```
def timsort(arr):  
    RUN = 32  
    n = len(arr)  
  
    # Step 1: sort small chunks  
    for i in range(0, n, RUN):  
        insertion_sort(arr, i, min((i + RUN - 1), n - 1))  
  
    # Step 2: merge sorted runs  
    size = RUN  
    while size < n:  
        for start in range(0, n, size * 2):  
            mid = start + size - 1  
            end = min(start + size * 2 - 1, n - 1)  
            merge(arr, start, mid, end)  
        size *= 2
```

Complexity:

- Best: ( $O(n)$ ) (already sorted data)- Average: ( $O(n \log n)$ )- Worst: ( $O(n \log n)$ )- Space: ( $O(n)$ )- Stable: Yes Key Strengths:
- Excellent for real-world, partially sorted data- Stable (keeps equal keys in order)- Optimized merges (adaptive merging)

## 4. Comparison

Algorithm	Base Methods	Stability	Best	Average	Worst	Real Use
IntroSort	Quick + Heap + Insertion	No	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	C++ STL
Timsort	Merge + Insertion	Yes	$O(n)$	$O(n \log n)$	$O(n \log n)$	Python, Java

IntroSort prioritizes performance guarantees. Timsort prioritizes adaptivity and stability.

Both show that “one size fits all” sorting doesn’t exist , great systems detect *what’s going on* and adapt.

### Tiny Code

Suppose we run Timsort on [1, 2, 3, 7, 6, 5, 8, 9]:

- Detect runs: [1,2,3], [7,6,5], [8,9]- Reverse [7,6,5] → [5,6,7]- Merge runs → [1,2,3,5,6,7,8,9] Efficient because it leverages the existing order.

### Why It Matters

Hybrid sorts are the real-world heroes , they combine theory with practice. They teach an important principle:

When one algorithm’s weakness shows up, switch to another’s strength.

These are not academic curiosities , they’re in your compiler, your browser, your OS, your database. Understanding them means you understand how modern languages optimize fundamental operations.

### Try It Yourself

1. Implement IntroSort and test on random, sorted, and reverse-sorted arrays.
2. Simulate Timsort’s run detection on nearly sorted input.
3. Compare sorting speed of Insertion Sort vs Timsort for small arrays.
4. Add counters to Quick Sort and see when IntroSort should switch.
5. Explore Python’s `sorted()` with different input shapes , guess when it uses merge vs insertion.

Hybrid sorts remind us: good algorithms adapt , they’re not rigid, they’re smart.

## 15. Special Sorts (Cycle, Gnome, Comb, Pancake)

Not all sorting algorithms follow the mainstream divide-and-conquer or distribution paradigms. Some were designed to solve niche problems, to illustrate elegant ideas, or simply to experiment with different mechanisms of ordering.

These special sorts, Cycle Sort, Gnome Sort, Comb Sort, and Pancake Sort, are fascinating not because they're the fastest, but because they reveal creative ways to think about permutation, local order, and in-place operations.

### 1. Cycle Sort

Idea: Minimize the number of writes. Cycle sort rearranges elements into cycles, placing each value directly in its correct position. It performs exactly as many writes as there are misplaced elements, making it ideal for flash memory or systems where writes are expensive.

Steps:

1. For each position  $i$ , find where  $\text{arr}[i]$  belongs (its rank).
2. If it's not already there, swap it into position.
3. Continue the cycle until the current position is correct.
4. Move to the next index.

Code:

```
void cycle_sort(int arr[], int n) {
    for (int cycle_start = 0; cycle_start < n - 1; cycle_start++) {
        int item = arr[cycle_start];
        int pos = cycle_start;

        for (int i = cycle_start + 1; i < n; i++)
            if (arr[i] < item) pos++;

        if (pos == cycle_start) continue;

        while (item == arr[pos]) pos++;
        int temp = arr[pos];
        arr[pos] = item;
        item = temp;

        while (pos != cycle_start) {
            pos = cycle_start;
            for (int i = cycle_start + 1; i < n; i++)
```

```

        if (arr[i] < item) pos++;
    while (item == arr[pos]) pos++;
    temp = arr[pos];
    arr[pos] = item;
    item = temp;
}
}
}

```

Complexity:

- Time: ( $O(n^2)$ )- Writes: minimal (exactly  $n-c$ , where  $c = \#cycles$ )- Stable: No Use Case: When minimizing writes is more important than runtime.

## 2. Gnome Sort

Idea: A simpler variation of insertion sort. Gnome sort moves back and forth like a “gnome” tidying flower pots: if two adjacent pots are out of order, swap and step back; otherwise, move forward.

Steps:

1. Start at index 1
2. If  $\text{arr}[i] \geq \text{arr}[i-1]$ , move forward
3. Else, swap and step back (if possible)
4. Repeat until the end

Code:

```

void gnome_sort(int arr[], int n) {
    int i = 1;
    while (i < n) {
        if (i == 0 || arr[i] >= arr[i - 1]) i++;
        else {
            int temp = arr[i]; arr[i] = arr[i - 1]; arr[i - 1] = temp;
            i--;
        }
    }
}

```

Complexity:

- Time: ( $O(n^2)$ )- Space: ( $O(1)$ )- Stable: Yes Use Case: Educational simplicity. It's a readable form of insertion logic without nested loops.

### 3. Comb Sort

Idea: An improvement over Bubble Sort by introducing a gap between compared elements, shrinking it gradually. By jumping farther apart early, Comb Sort helps eliminate small elements that are “stuck” near the end.

Steps:

1. Start with gap = n
2. On each pass, shrink gap = gap / 1.3
3. Compare and swap items gap apart
4. Stop when gap = 1 and no swaps occur

Code:

```
void comb_sort(int arr[], int n) {  
    int gap = n;  
    int swapped = 1;  
    while (gap > 1 || swapped) {  
        gap = (gap * 10) / 13;  
        if (gap == 9 || gap == 10) gap = 11;  
        if (gap < 1) gap = 1;  
        swapped = 0;  
        for (int i = 0; i + gap < n; i++) {  
            if (arr[i] > arr[i + gap]) {  
                int temp = arr[i]; arr[i] = arr[i + gap]; arr[i + gap] = temp;  
                swapped = 1;  
            }  
        }  
    }  
}
```

Complexity:

- Average:  $(On \log n)$ - Worst:  $(On^2)$ - Space:  $(O(1))$ - Stable: No Use Case: When a simple, in-place, nearly linear-time alternative to bubble sort is desired.

### 4. Pancake Sort

Idea: Sort an array using only one operation: flip (reversing a prefix). It’s like sorting pancakes on a plate, flip the stack so the largest pancake goes to the bottom, then repeat for the rest.

Steps:

1. Find the maximum unsorted element
2. Flip it to the front
3. Flip it again to its correct position
4. Reduce the unsorted portion by one

Code:

```

void flip(int arr[], int i) {
    int start = 0;
    while (start < i) {
        int temp = arr[start];
        arr[start] = arr[i];
        arr[i] = temp;
        start++;
        i--;
    }
}

void pancake_sort(int arr[], int n) {
    for (int curr_size = n; curr_size > 1; curr_size--) {
        int mi = 0;
        for (int i = 1; i < curr_size; i++)
            if (arr[i] > arr[mi]) mi = i;
        if (mi != curr_size - 1) {
            flip(arr, mi);
            flip(arr, curr_size - 1);
        }
    }
}

```

Complexity:

- Time:  $(On^2)$ - Space:  $(O(1))$ - Stable: No Fun Fact: Pancake sort is the only known algorithm whose operations mimic a kitchen utensil, and inspired the Burnt Pancake Problem in combinatorics and genome rearrangement theory.

## 5. Comparison

Algorithm	Time	Space	Stable	Distinctive Trait
Cycle Sort	$O(n^2)$	$O(1)$	No	Minimal writes
Gnome Sort	$O(n^2)$	$O(1)$	Yes	Simple insertion-like behavior

Algorithm	Time	Space	Stable	Distinctive Trait
Comb Sort	$O(n \log n)$ avg	$O(1)$	No	Shrinking gap, improved bubble
Pancake Sort	$O(n^2)$	$O(1)$	No	Prefix reversals only

Each highlights a different design goal:

- Cycle: minimize writes- Gnome: simplify logic- Comb: optimize comparisons- Pancake: restrict operations

### Tiny Code

Example (Pancake Sort on [3, 6, 1, 9]):

1. Max = 9 at index 3 → flip(3) → [9, 1, 6, 3]
2. flip(3) → [3, 6, 1, 9] (9 fixed)
3. Max = 6 → flip(1) → [6, 3, 1, 9]
4. flip(2) → [1, 3, 6, 9]

Sorted using only flips.

### Why It Matters

Special sorts show there's more than one way to think about ordering. They're laboratories for exploring new ideas: minimizing swaps, limiting operations, or optimizing stability. Even if they're not the go-to in production, they deepen your intuition about sorting mechanics.

### Try It Yourself

1. Implement each algorithm and visualize their operations step-by-step.
2. Measure how many writes Cycle Sort performs vs. others.
3. Compare Gnome and Insertion sort on nearly sorted arrays.
4. Modify Comb Sort's shrink factor, how does performance change?
5. Write Pancake Sort with printouts of every flip to see the “stack” in motion.

These quirky algorithms prove that sorting isn't just science, it's also art and experimentation.

## 16. Linear and Binary Search

Searching is the process of finding a target value within a collection of data. Depending on whether the data is sorted or unsorted, you'll use different strategies.

In this section, we revisit two of the most fundamental searching methods , Linear Search and Binary Search , and see how they underpin many higher-level algorithms and data structures.

### 1. Linear Search

Idea: Check each element one by one until you find the target. This is the simplest possible search and works on unsorted data.

Steps:

1. Start from index 0
2. Compare `arr[i]` with the target
3. If match, return index
4. If end reached, return -1

Code:

```
int linear_search(int arr[], int n, int key) {  
    for (int i = 0; i < n; i++) {  
        if (arr[i] == key) return i;  
    }  
    return -1;  
}
```

Example:  $\text{arr} = [7, 2, 4, 9, 1]$ ,  $\text{key} = 9$

- Compare 7, 2, 4, then 9 → found at index 3 Complexity:
- Time: ( O(n) )- Space: ( O(1) )- Best case: ( O(1) ) (first element)- Worst case: ( O(n) )  
Pros:
  - Works on any data (sorted or unsorted)- Simple to implement Cons:
    - Inefficient on large arrays Use it when data is small or unsorted, or when simplicity matters more than speed.

## 2. Binary Search

Idea: If the array is sorted, you can repeatedly halve the search space. Compare the middle element to the target , if it's greater, search left; if smaller, search right.

Steps:

1. Find the midpoint
2. If  $\text{arr}[\text{mid}] == \text{key}$ , done
3. If  $\text{arr}[\text{mid}] > \text{key}$ , search left
4. If  $\text{arr}[\text{mid}] < \text{key}$ , search right
5. Repeat until range is empty

Iterative Version:

```
int binary_search(int arr[], int n, int key) {  
    int low = 0, high = n - 1;  
    while (low <= high) {  
        int mid = (low + high) / 2;  
        if (arr[mid] == key) return mid;  
        else if (arr[mid] < key) low = mid + 1;  
        else high = mid - 1;  
    }  
    return -1;  
}
```

Recursive Version:

```
int binary_search_rec(int arr[], int low, int high, int key) {  
    if (low > high) return -1;  
    int mid = (low + high) / 2;  
    if (arr[mid] == key) return mid;  
    else if (arr[mid] > key)  
        return binary_search_rec(arr, low, mid - 1, key);  
    else  
        return binary_search_rec(arr, mid + 1, high, key);  
}
```

Example:  $\text{arr} = [1, 3, 5, 7, 9, 11]$ ,  $\text{key} = 7$

- $\text{mid} = 5 \rightarrow \text{key} > \text{mid} \rightarrow$  move right-  $\text{mid} = 7 \rightarrow$  found Complexity:
- Time: (  $O(\log n)$  )- Space: (  $O(1)$  ) (iterative) or (  $O(\log n)$  ) (recursive)- Best case: (  $O(1)$  ) (middle element) Requirements:

- Must be sorted- Must have random access (array, not linked list) Pros:
- Very fast for large sorted arrays- Foundation for advanced searches (e.g. interpolation, exponential) Cons:
- Needs sorted data- Doesn't adapt to frequent insertions/deletions

### 3. Binary Search Variants

Binary search is a *pattern* as much as a single algorithm. You can tweak it to find:

- First occurrence: move left if `arr[mid] == key`- Last occurrence: move right if `arr[mid] == key`- Lower bound: first index `key`- Upper bound: first index `> key` Example (Lower Bound):

```
int lower_bound(int arr[], int n, int key) {
    int low = 0, high = n;
    while (low < high) {
        int mid = (low + high) / 2;
        if (arr[mid] < key) low = mid + 1;
        else high = mid;
    }
    return low;
}
```

Usage: These variants power functions like `std::lower_bound()` in C++ and binary search trees' lookup logic.

### 4. Comparison

Algorithm	Works				Notes
	On	Time	Space	Sorted Data Needed	
Linear Search	Any	$O(n)$	$O(1)$	No	Best for small/unsorted
Binary Search	Sorted	$O(\log n)$	$O(1)$	Yes	Fastest on ordered arrays

Binary search trades simplicity for power , once your data is sorted, you unlock sublinear search.

## Tiny Code

Compare on array [2, 4, 6, 8, 10], key = 8:

- Linear: 4 steps- Binary: 2 steps This gap grows huge with size , for  $n = 10^6$ , linear takes up to a million steps, binary about 20.

## Why It Matters

These two searches form the foundation of retrieval. Linear search shows brute-force iteration; binary search shows how structure (sorted order) leads to exponential improvement.

From databases to compiler symbol tables to tree lookups, this principle , *divide to search faster* , is everywhere.

## Try It Yourself

1. Implement linear and binary search.
2. Count comparisons for ( n = 10, 100, 1000 ).
3. Modify binary search to return the first occurrence of a duplicate.
4. Try binary search on unsorted data , what happens?
5. Combine with sorting: sort array, then search.

Mastering these searches builds intuition for all lookup operations , they are the gateway to efficient data retrieval.

## 17. Interpolation and Exponential Search

Linear and binary search work well across many scenarios, but they don't take into account how data is distributed. When values are uniformly distributed, we can *estimate* where the target lies, instead of always splitting the range in half. This leads to Interpolation Search, which "jumps" close to where the value should be.

For unbounded or infinite lists, we can't even know the size of the array up front , that's where Exponential Search shines, by quickly expanding its search window before switching to binary search.

Let's dive into both.

## 1. Interpolation Search

Idea: If data is sorted and uniformly distributed, you can *predict* where a key might be using linear interpolation. Instead of splitting at the middle, estimate the position based on the value's proportion in the range.

Formula:

$$\text{pos} = \text{low} + \frac{(\text{key} - \text{arr}[\text{low}]) \times (\text{high} - \text{low})}{\text{arr}[\text{high}] - \text{arr}[\text{low}]}$$

This “guesses” where the key lies. If ( $\text{key} = \text{arr}[\text{pos}]$ ), we’re done. Otherwise, adjust `low` or `high` and repeat.

Steps:

1. Compute estimated position `pos`
2. Compare `arr[pos]` with `key`
3. Narrow range accordingly
4. Repeat while `low <= high` and `key` within range

Code:

```
int interpolation_search(int arr[], int n, int key) {
    int low = 0, high = n - 1;

    while (low <= high && key >= arr[low] && key <= arr[high]) {
        if (low == high) {
            if (arr[low] == key) return low;
            return -1;
        }
        int pos = low + ((double)(key - arr[low]) * (high - low)) / (arr[high] - arr[low]);

        if (arr[pos] == key)
            return pos;
        if (arr[pos] < key)
            low = pos + 1;
        else
            high = pos - 1;
    }
    return -1;
}
```

Example:  $\text{arr} = [10, 20, 30, 40, 50]$ ,  $\text{key} = 40$   $\text{pos} = 0 + ((40 - 10) * (4 - 0)) / (50 - 10) = 3 \rightarrow$  found at index 3

Complexity:

- Best: ( $O(1)$ )- Average: ( $O(\log \log n)$ ) (uniform data)- Worst: ( $O(n)$ ) (non-uniform or skewed data)- Space: ( $O(1)$ ) When to Use:
  - Data is sorted and nearly uniform- Numeric data where values grow steadily Note: Interpolation search is adaptive , faster when data is predictable, slower when data is irregular.

## 2. Exponential Search

Idea: When you don't know the array size (e.g., infinite streams, linked data, files), you can't just binary search from 0 to  $n-1$ . Exponential search finds a search range dynamically by doubling its step size until it overshoots the target, then does binary search within that range.

Steps:

1. If  $\text{arr}[0] == \text{key}$ , return 0
2. Find a range  $[\text{bound}/2, \text{bound}]$  such that  $\text{arr}[\text{bound}] \geq \text{key}$
3. Perform binary search in that range

Code:

```
int exponential_search(int arr[], int n, int key) {  
    if (arr[0] == key) return 0;  
    int bound = 1;  
    while (bound < n && arr[bound] < key)  
        bound *= 2;  
    int low = bound / 2;  
    int high = (bound < n) ? bound : n - 1;  
    // Binary search in [low, high]  
    while (low <= high) {  
        int mid = (low + high) / 2;  
        if (arr[mid] == key) return mid;  
        else if (arr[mid] < key) low = mid + 1;  
        else high = mid - 1;  
    }  
    return -1;  
}
```

Example:  $\text{arr} = [2, 4, 6, 8, 10, 12, 14, 16]$ ,  $\text{key} = 10$

- Step:  $\text{bound} = 1$  (4), 2 (6), 4 (10 - key)- Binary search  $[2,4] \rightarrow$  found Complexity:

- Time:  $(O(\log i))$ , where  $i$  is index of the target- Space:  $(O(1))$ - Best:  $(O(1))$  When to Use:
- Unbounded or streamed data- Unknown array size but sorted order

### 3. Comparison

Algorithm	Best Case	Average Case	Worst Case	Data Requirement	Notes
Linear Search	$O(1)$	$O(n)$	$O(n)$	Unsorted	Works everywhere
Binary Search	$O(1)$	$O(\log n)$	$O(\log n)$	Sorted	Predictable halving
Interpolation Search	$O(1)$	$O(\log \log n)$	$O(n)$	Sorted + Uniform	Adaptive, fast on uniform data
Exponential Search	$O(1)$	$O(\log n)$	$O(\log n)$	Sorted	Great for unknown size

Interpolation improves on binary *if* data is smooth. Exponential shines when size is unknown.

### Tiny Code

Interpolation intuition: If your data is evenly spaced (10, 20, 30, 40, 50), the value 40 should be roughly 75% along. Instead of halving every time, we jump *right near it*. It's data-aware searching.

Exponential intuition: When size is unknown, “expand until you find the wall,” then search within.

### Why It Matters

These two searches show how context shapes algorithm design:

- *Distribution* (Interpolation Search)- *Boundaries* (Exponential Search) They teach that performance depends not only on structure (sortedness) but also metadata , how much you know about data spacing or limits.

These principles resurface in skip lists, search trees, and probabilistic indexing.

## Try It Yourself

1. Test interpolation search on [10, 20, 30, 40, 50] , note how few steps it takes.
2. Try the same on [1, 2, 4, 8, 16, 32, 64] , note slowdown.
3. Implement exponential search and simulate an “infinite” array by stopping at n.
4. Compare binary vs interpolation search on random vs uniform data.
5. Extend exponential search to linked lists , how does complexity change?

Understanding these searches helps you tailor lookups to the shape of your data , a key skill in algorithmic thinking.

## 18. Selection Algorithms (Quickselect, Median of Medians)

Sometimes you don’t need to sort an entire array , you just want the k-th smallest (or largest) element. Sorting everything is overkill when you only need one specific rank. Selection algorithms solve this problem efficiently, often in linear time.

They’re the backbone of algorithms for median finding, percentiles, and order statistics, and they underpin operations like *pivot selection* in Quick Sort.

### 1. The Selection Problem

Given an unsorted array of ( n ) elements and a number ( k ), find the element that would be at position ( k ) if the array were sorted.

For example: arr = [7, 2, 9, 4, 6], (k = 3) → Sorted = [2, 4, 6, 7, 9] → 3rd smallest = 6

We can solve this without sorting everything.

### 2. Quickselect

Idea: Quickselect is a selection variant of Quick Sort. It partitions the array around a pivot, but recurses only on the side that contains the k-th element.

It has average-case O(n) time because each partition roughly halves the search space.

Steps:

1. Choose a pivot (random or last element)
2. Partition array into elements < pivot and > pivot
3. Let pos be the pivot’s index after partition
4. If pos == k-1 → done
5. If pos > k-1 → recurse left
6. If pos < k-1 → recurse right

Code:

```
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low;
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            int temp = arr[i]; arr[i] = arr[j]; arr[j] = temp;
            i++;
        }
    }
    int temp = arr[i]; arr[i] = arr[high]; arr[high] = temp;
    return i;
}

int quickselect(int arr[], int low, int high, int k) {
    if (low == high) return arr[low];
    int pos = partition(arr, low, high);
    int rank = pos - low + 1;
    if (rank == k) return arr[pos];
    if (rank > k) return quickselect(arr, low, pos - 1, k);
    return quickselect(arr, pos + 1, high, k - rank);
}
```

Example: arr = [7, 2, 9, 4, 6], ( k = 3 )

- Pivot = 6- Partition → [2, 4, 6, 9, 7], pos = 2- rank = 3 → found (6) Complexity:
- Average: (O(n))- Worst: ( $O(n^2)$ ) (bad pivots)- Space: (O(1))- In-place When to Use:
- Fast average case- You don't need full sorting Quickselect is used in C++'s `nth_element()` and many median-finding implementations.

### 3. Median of Medians

Idea: Guarantee worst-case ( O(n) ) time by choosing a good pivot deterministically.

This method ensures the pivot divides the array into reasonably balanced parts every time.

Steps:

1. Divide array into groups of 5
2. Find the median of each group (using insertion sort)
3. Recursively find the median of these medians → pivot

4. Partition array around this pivot
5. Recurse into the side containing the k-th element

This guarantees at least 30% of elements are eliminated each step → linear time in worst case.

Code Sketch:

```
int select_pivot(int arr[], int low, int high) {
    int n = high - low + 1;
    if (n <= 5) {
        insertion_sort(arr + low, n);
        return low + n / 2;
    }

    int medians[(n + 4) / 5];
    int i;
    for (i = 0; i < n / 5; i++) {
        insertion_sort(arr + low + i * 5, 5);
        medians[i] = arr[low + i * 5 + 2];
    }
    if (i * 5 < n) {
        insertion_sort(arr + low + i * 5, n % 5);
        medians[i] = arr[low + i * 5 + (n % 5) / 2];
        i++;
    }
    return select_pivot(medians, 0, i - 1);
}
```

You'd then partition around pivot and recurse just like Quickselect.

Complexity:

- Worst:  $(O(n))$ - Space:  $(O(1))$  (in-place version)- Stable: No (doesn't preserve order)  
Why It Matters: Median of Medians is slower in practice than Quickselect but provides theoretical guarantees , vital in real-time or critical systems.

#### 4. Special Cases

- Min / Max: trivial , just scan once  $((O(n)))$ - Median:  $k = \lceil n/2 \rceil$  , can use Quickselect or Median of Medians- Top-k Elements: use partial selection or heaps (k smallest/largest)  
Example: To get top 5 scores from a million entries, use Quickselect to find 5th largest, then filter threshold.

## 5. Comparison

Algorithm	Best	Average	Worst	Stable	In-Place	Notes
Quickselect	$O(n)$	$O(n)$	$O(n^2)$	No	Yes	Fast in practice
Median of Medians	$O(n)$	$O(n)$	$O(n)$	No	Yes	Deterministic
Sorting	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Depends	Depends	Overkill for single element

Quickselect is fast and simple; Median of Medians is safe and predictable.

### Tiny Code

Find 4th smallest in [9, 7, 2, 5, 4, 3]:

- Pivot = 4 → partition [2,3,4,9,7,5]- 4 at position 2 → rank = 3 < 4 → recurse right- New range [9,7,5], ( k = 1 ) → smallest = 5 Result: 5

### Why It Matters

Selection algorithms reveal a key insight:

Sometimes you don't need everything , just what matters.

They form the basis for:

- Median filters in signal processing- Partitioning steps in sorting- k-th order statistics- Robust statistics and quantile computation They embody a “partial work, full answer” philosophy , do exactly enough.

### Try It Yourself

1. Implement Quickselect and find k-th smallest for various k.
2. Compare runtime vs full sorting.
3. Modify Quickselect to find k-th largest.
4. Implement Median of Medians pivot selection.
5. Use Quickselect to find median of 1,000 random elements.

Mastering selection algorithms helps you reason about efficiency , you'll learn when to stop sorting and start selecting.

## 19. Range Searching and Nearest Neighbor

Searching isn't always about finding a single key. Often, you need to find all elements within a given range , or the closest match to a query point.

These problems are central to databases, computational geometry, and machine learning (like k-NN classification). This section introduces algorithms for range queries (e.g. find all values between L and R) and nearest neighbor searches (e.g. find the point closest to query q).

### 1. Range Searching

Idea: Given a set of data points (1D or multidimensional), quickly report all points within a specified range.

In 1D (simple arrays), range queries can be handled by binary search and prefix sums. In higher dimensions, we need trees designed for efficient spatial querying.

#### A. 1D Range Query (Sorted Array)

Goal: Find all elements in [L, R].

Steps:

1. Use lower bound to find first element L
2. Use upper bound to find first element > R
3. Output all elements in between

Code (C++-style pseudo):

```
int l = lower_bound(arr, arr + n, L) - arr;
int r = upper_bound(arr, arr + n, R) - arr;
for (int i = l; i < r; i++)
    printf("%d ", arr[i]);
```

Time Complexity:

- Binary search bounds: ( $O(\log n)$ ) - Reporting results: ( $O(k)$ ) where ( $k$ ) = number of elements in range → Total: ( $O(\log n + k)$ )

## B. Prefix Sum Range Query (For sums)

If you just need the sum (not the actual elements), use prefix sums:

$$\text{prefix}[i] = a_0 + a_1 + \dots + a_i$$

Then range sum:

$$\text{sum}(L, R) = \text{prefix}[R] - \text{prefix}[L - 1]$$

Code:

```
int prefix[n];
prefix[0] = arr[0];
for (int i = 1; i < n; i++)
    prefix[i] = prefix[i - 1] + arr[i];

int range_sum(int L, int R) {
    return prefix[R] - (L > 0 ? prefix[L - 1] : 0);
}
```

Time: ( $O(1)$ ) per query after ( $O(n)$ ) preprocessing.

Used in:

- Databases for fast range aggregation- Fenwick trees, segment trees

## C. 2D Range Queries (Rectangular Regions)

For points  $((x, y))$ , queries like:

“Find all points where  $L_x \leq x \leq R_x$  and  $L_y \leq y \leq R_y$ ”

Use specialized structures:

- Range Trees (balanced BSTs per dimension)- Fenwick Trees / Segment Trees (for 2D arrays)- KD-Trees (spatial decomposition) Time:  $(O(\log^2 n + k))$  typical for 2D Space:  $(O(n \log n))$

## 2. Nearest Neighbor Search

Idea: Given a set of points, find the one closest to query (q). Distance is often Euclidean, but can be any metric.

Brute Force: Check all points →  $O(n)$  per query. Too slow for large datasets.

We need structures that let us prune far regions fast.

### A. KD-Tree

KD-tree = K-dimensional binary tree. Each level splits points by one coordinate, alternating axes. Used for efficient nearest neighbor search in low dimensions (2D-10D).

Construction:

1. Choose axis = depth % k
2. Sort points by axis
3. Pick median → root
4. Recursively build left and right

Query (Nearest Neighbor):

1. Traverse down tree based on query position
2. Backtrack , check whether hypersphere crosses splitting plane
3. Keep track of best (closest) distance

Complexity:

- Build:  $(O(n \log n))$ - Query:  $(O(\log n))$  avg,  $(O(n))$  worst Use Cases:
- Nearest city lookup- Image / feature vector matching- Game AI spatial queries Code Sketch (2D Example):

```
struct Point { double x, y; };

double dist(Point a, Point b) {
    return sqrt((a.x - b.x)*(a.x - b.x) + (a.y - b.y)*(a.y - b.y));
}
```

(Full KD-tree implementation omitted for brevity , idea is recursive partitioning.)

## B. Ball Tree / VP-Tree

For high-dimensional data, KD-trees degrade. Alternatives like Ball Trees (split by hyperspheres) or VP-Trees (Vantage Point Trees) perform better.

They split based on distance metrics, not coordinate axes.

## C. Approximate Nearest Neighbor (ANN)

For large-scale, high-dimensional data (e.g. embeddings, vectors):

- Locality Sensitive Hashing (LSH)- HNSW (Hierarchical Navigable Small World Graphs)  
These trade exactness for speed, common in:
- Vector databases- Recommendation systems- AI model retrieval

## 3. Summary

Problem	Brute Force	Optimized	Time (Query)	Notes
1D Range Query	Scan $O(n)$	Binary Search	$O(\log n + k)$	Sorted data
Range Sum	$O(n)$	Prefix Sum	$O(1)$	Static data
2D Range Query	$O(n)$	Range Tree	$O(\log^2 n + k)$	Spatial filtering
Nearest Neighbor	$O(n)$	KD-Tree	$O(\log n)$ avg	Exact, low-dim
Nearest Neighbor (high-dim)	$O(n)$	HNSW / LSH	$\sim O(1)$	Approximate

## Tiny Code

Simple 1D range query:

```
int arr[] = {1, 3, 5, 7, 9, 11};
int L = 4, R = 10;
int l = lower_bound(arr, arr + 6, L) - arr;
int r = upper_bound(arr, arr + 6, R) - arr;
for (int i = l; i < r; i++)
    printf("%d ", arr[i]); // 5 7 9
```

Output: 5 7 9

## Why It Matters

Range and nearest-neighbor queries power:

- Databases (SQL range filters, BETWEEN)- Search engines (spatial indexing)- ML (k-NN classifiers, vector similarity)- Graphics / Games (collision detection, spatial queries) These are not just searches , they're geometric lookups, linking algorithms to spatial reasoning.

## Try It Yourself

1. Write a function to return all numbers in [L, R] using binary search.
2. Build a prefix sum array and answer 5 range-sum queries in O(1).
3. Implement a KD-tree for 2D points and query nearest neighbor.
4. Compare brute-force vs KD-tree search on 1,000 random points.
5. Explore Python's `scipy.spatial.KDTree` or `sklearn.neighbors`.

These algorithms bridge searching with geometry and analytics, forming the backbone of spatial computation.

## 20. Search Optimizations and Variants

We've explored the main search families , linear, binary, interpolation, exponential , each fitting a different data shape or constraint. Now let's move one step further: optimizing search for performance and adapting it to specialized scenarios.

This section introduces practical variants and enhancements used in real systems, databases, and competitive programming, including jump search, fibonacci search, ternary search, and exponential + binary combinations.

### 1. Jump Search

Idea: If data is sorted, we can “jump” ahead by fixed steps instead of scanning linearly. It's like hopping through the array in blocks , when you overshoot the target, you step back and linearly search that block.

It strikes a balance between linear and binary search , fewer comparisons without the recursion or halving of binary search.

Steps:

1. Choose jump size =  $\sqrt{n}$
2. Jump by blocks until `arr[step] > key`
3. Linear search in previous block

Code:

```
int jump_search(int arr[], int n, int key) {
    int step = sqrt(n);
    int prev = 0;

    while (arr[min(step, n) - 1] < key) {
        prev = step;
        step += sqrt(n);
        if (prev >= n) return -1;
    }

    for (int i = prev; i < min(step, n); i++) {
        if (arr[i] == key) return i;
    }
    return -1;
}
```

Example: arr = [1, 3, 5, 7, 9, 11, 13, 15], key = 11

- step = 2- Jump 5, 7, 9, 11 → found Complexity:
- Time:  $(O\sqrt{n})$ - Space:  $(O(1))$ - Works on sorted data When to Use: For moderately sized sorted lists when you want fewer comparisons but minimal overhead.

## 2. Fibonacci Search

Idea: Similar to binary search, but it splits the array based on Fibonacci numbers instead of midpoints. This allows using only addition and subtraction (no division), useful on hardware where division is costly.

Also, like binary search, it halves (roughly) the search space each iteration.

Steps:

1. Find the smallest Fibonacci number n
2. Use it to compute probe index
3. Compare and move interval accordingly

Code (Sketch):

```

int fibonacci_search(int arr[], int n, int key) {
    int fibMMm2 = 0; // (m-2)'th Fibonacci
    int fibMMm1 = 1; // (m-1)'th Fibonacci
    int fibM = fibMMm2 + fibMMm1; // m'th Fibonacci

    while (fibM < n) {
        fibMMm2 = fibMMm1;
        fibMMm1 = fibM;
        fibM = fibMMm2 + fibMMm1;
    }

    int offset = -1;
    while (fibM > 1) {
        int i = min(offset + fibMMm2, n - 1);
        if (arr[i] < key) {
            fibM = fibMMm1;
            fibMMm1 = fibMMm2;
            fibMMm2 = fibM - fibMMm1;
            offset = i;
        } else if (arr[i] > key) {
            fibM = fibMMm2;
            fibMMm1 = fibMMm1 - fibMMm2;
            fibMMm2 = fibM - fibMMm1;
        } else return i;
    }
    if (fibMMm1 && arr[offset + 1] == key)
        return offset + 1;
    return -1;
}

```

Complexity:

- Time: ( $O(\log n)$ )- Space: ( $O(1)$ )- Sorted input required Fun Fact: Fibonacci search was originally designed for tape drives , where random access is expensive, and predictable jumps matter.

### 3. Ternary Search

Idea: When the function or sequence is unimodal (strictly increasing then decreasing), you can locate a maximum or minimum by splitting the range into three parts instead of two.

Used not for discrete lookup but for optimization on sorted functions.

Steps:

1. Divide range into thirds
2. Evaluate at two midpoints  $m_1, m_2$
3. Eliminate one-third based on comparison
4. Repeat until range is small

Code:

```
double ternary_search(double low, double high, double (*f)(double)) {  
    for (int i = 0; i < 100; i++) {  
        double m1 = low + (high - low) / 3;  
        double m2 = high - (high - low) / 3;  
        if (f(m1) < f(m2))  
            low = m1;  
        else  
            high = m2;  
    }  
    return (low + high) / 2;  
}
```

Example: Find minimum of ( $f(x) = (x-3)^2$ ) between [0,10]. After iterations, converges to (x-3).

Complexity:

- Time:  $O(\log \text{range})$
- Space:  $O(1)$
- Works for unimodal functions

Used in:

- Mathematical optimization
- Search-based tuning
- Game AI decision models

#### 4. Binary Search Variants (Review)

Binary search can be tailored to answer richer queries:

- Lower Bound: first index - key- Upper Bound: first index > key- Equal Range: range of all equal elements- Rotated Arrays: find element in rotated sorted array- Infinite Arrays: use exponential expansion Rotated Example: arr = [6,7,9,1,3,4], key = 3 → Find pivot, then binary search correct side.

## 5. Combined Searches

Real systems often chain algorithms:

- Exponential + Binary Search → when bounds unknown- Interpolation + Linear Search  
→ when near target- Jump + Linear Search → hybrid iteration These hybrids use context switching , pick a fast search, then fall back to simple scan in a narrowed window.

## 6. Summary

Algorithm	Time	Space	Data Requirement	Special Strength
Jump Search	$O(\sqrt{n})$	$O(1)$	Sorted	Fewer comparisons
Fibonacci Search	$O(\log n)$	$O(1)$	Sorted	Division-free
Ternary Search	$O(\log \text{range})$	$O(1)$	Unimodal	Optimization
Binary Variants	$O(\log n)$	$O(1)$	Sorted	Bound finding
Combined Searches	Adaptive	$O(1)$	Mixed	Practical hybrids

### Tiny Code

Jump Search intuition:

```
// Blocks of size sqrt(n)
[1, 3, 5, 7, 9, 11, 13, 15]
Step: 3 → 7 > 6 → search previous block
```

Jumps reduce comparisons dramatically vs linear scan.

### Why It Matters

Search optimization is about adapting structure to context. You don't always need a fancy data structure , sometimes a tweak like fixed-step jumping or Fibonacci spacing yields massive gains.

These ideas influence:

- Indexing in databases- Compilers' symbol resolution- Embedded systems with low-level constraints They embody the principle: search smarter, not harder.

## Try It Yourself

1. Implement Jump Search and test vs Binary Search on 1M elements.
2. Write a Fibonacci Search , compare steps taken.
3. Use Ternary Search to find min of a convex function.
4. Modify binary search to find element in rotated array.
5. Combine Jump + Linear , how does it behave for small n?

Understanding these variants arms you with flexibility , the heart of algorithmic mastery.

## Chapter 3. Data Structures in Actions

### 21. Arrays, Linked Lists, Stacks, Queues

Every data structure is built on top of a few core foundations , the ones that teach you how data is stored, accessed, and moved. In this section, we'll revisit the essentials: arrays, linked lists, stacks, and queues.

They're simple, but they show you the most important design trade-offs in algorithms:

- Contiguity vs. flexibility- Speed vs. dynamic growth- Last-in-first-out vs. first-in-first-out access

#### 1. Arrays

Idea: A contiguous block of memory storing elements of the same type. Access by index in  $O(1)$  time , that's their superpower.

Operations:

- Access `arr[i]`: ( $O(1)$ )- Update `arr[i]`: ( $O(1)$ )- Insert at end: ( $O(1)$ ) (amortized for dynamic arrays)- Insert in middle: ( $O(n)$ )- Delete: ( $O(n)$ ) Example:

```
int arr[5] = {10, 20, 30, 40, 50};  
printf("%d", arr[2]); // 30
```

Strengths:

- Fast random access- Cache-friendly (contiguous memory)- Simple, predictable Weaknesses:
- Fixed size (unless using dynamic array)- Costly inserts/deletes Dynamic Arrays: Languages provide resizable arrays (like `vector` in C++ or `ArrayList` in Java) using doubling strategy , when full, allocate new array twice as big and copy. This gives amortized ( $O(1)$ ) insertion at end.

## 2. Linked Lists

Idea: A chain of nodes, where each node stores a value and a pointer to the next. No contiguous memory required.

Operations:

- Access:  $(O(n))$ - Insert/Delete at head:  $(O(1))$ - Search:  $(O(n))$  Example:

```
typedef struct Node {
    int data;
    struct Node* next;
} Node;

Node* head = NULL;
```

Types:

- Singly Linked List: one pointer (next)- Doubly Linked List: two pointers (next, prev)- Circular Linked List: last node points back to first Strengths:
- Dynamic size- Fast insert/delete (no shifting) Weaknesses:
- Slow access- Extra memory for pointers- Poor cache locality Linked lists shine when memory is fragmented or frequent insertions/deletions are needed.

## 3. Stack

Idea: A Last-In-First-Out (LIFO) structure , the most recently added element is the first to be removed.

Used in:

- Function call stacks- Expression evaluation- Undo operations Operations:
- `push(x)`: add element on top- `pop()`: remove top element- `peek()`: view top element Example (Array-based Stack):

```
#define MAX 100
int stack[MAX], top = -1;

void push(int x) { stack[++top] = x; }
int pop() { return stack[top--]; }
int peek() { return stack[top]; }
```

Complexity: All ( $O(1)$ ): push, pop, peek

Variants:

- Linked-list-based stack (no fixed size)- Min-stack (tracks minimums) Stacks also appear implicitly , in recursion and backtracking algorithms.

## 4. Queue

Idea: A First-In-First-Out (FIFO) structure , the first added element leaves first.

Used in:

- Task scheduling- BFS traversal- Producer-consumer pipelines Operations:
- `enqueue(x)`: add to rear- `dequeue()`: remove from front- `front()`: view front Example (Array-based Queue):

```
#define MAX 100
int queue[MAX], front = 0, rear = 0;

void enqueue(int x) { queue[rear++] = x; }
int dequeue() { return queue[front++]; }
```

This simple implementation can waste space. A circular queue fixes that by wrapping indices modulo MAX:

```
rear = (rear + 1) % MAX;
```

Complexity: All ( $O(1)$ ): enqueue, dequeue

Variants:

- Deque (double-ended queue): push/pop from both ends- Priority Queue: dequeue highest priority (not strictly FIFO)

Structure	Access	Insert	Delete	Order	Memory	Notes
-----------	--------	--------	--------	-------	--------	-------

## 5. Comparison

Structure	Access	Insert	Delete	Order	Memory	Notes
Array	O(1)	O(n)	O(n)	Indexed	Contiguous	Fast access
Linked List	O(n)	O(1)*	O(1)*	Sequential	Pointers	Flexible size
Stack	O(1)	O(1)	O(1)	LIFO	Minimal	Call stack, parsing
Queue	O(1)	O(1)	O(1)	FIFO	Minimal	Scheduling, BFS

(\* at head or tail with pointer)

## Tiny Code

Simple stack example:

```
push(10);
push(20);
printf("%d", pop()); // 20
```

Simple queue example:

```
enqueue(5);
enqueue(8);
printf("%d", dequeue()); // 5
```

These short routines appear in almost every algorithm , from recursion stacks to graph traversals.

## Why It Matters

These four structures form the spine of data structures:

- Arrays teach indexing and memory- Linked lists teach pointers and dynamic allocation- Stacks teach recursion and reversal- Queues teach scheduling and order maintenance Every complex structure (trees, heaps, graphs) builds on these.

Master them, and every algorithm will feel more natural.

## Try It Yourself

1. Implement a linked list with `insert_front` and `delete_value`.
2. Build a stack and use it to reverse an array.
3. Implement a queue for a round-robin scheduler.
4. Convert infix expression to postfix using a stack.
5. Compare time taken to insert 1000 elements in array vs linked list.

Understanding these foundations gives you the vocabulary of structure , the way algorithms organize their thoughts in memory.

## 22. Hash Tables and Variants (Cuckoo, Robin Hood, Consistent)

When you need lightning-fast lookups, insertions, and deletions, few data structures match the raw efficiency of a hash table. They're everywhere , from symbol tables and caches to compilers and databases , powering average-case  $O(1)$  access.

In this section, we'll unpack how hash tables work, their collision strategies, and explore modern variants like Cuckoo Hashing, Robin Hood Hashing, and Consistent Hashing, each designed to handle different real-world needs.

### 1. The Core Idea

A hash table maps keys to values using a hash function that transforms the key into an index in an array.

$$\text{index} = h(\text{key}) \bmod \text{table\_size}$$

If no two keys hash to the same index, all operations are ( $O(1)$ ). But in practice, collisions happen , two keys may map to the same slot , and we must handle them smartly.

### 2. Collision Resolution Strategies

A. Separate Chaining Each table slot holds a linked list (or dynamic array) of entries with the same hash.

Pros: Simple, handles load factor  $> 1$  Cons: Extra pointers, memory overhead

Code Sketch:

```

typedef struct Node {
    int key, value;
    struct Node* next;
} Node;

Node* table[SIZE];

int hash(int key) { return key % SIZE; }

void insert(int key, int value) {
    int idx = hash(key);
    Node* node = malloc(sizeof(Node));
    node->key = key; node->value = value;
    node->next = table[idx];
    table[idx] = node;
}

```

B. Open Addressing All keys live directly in the table. On collision, find another slot.

Three main strategies:

- Linear probing: try next slot (+1)- Quadratic probing: step size increases quadratically- Double hashing: second hash decides step size Example (Linear Probing):

```

int hash(int key) { return key % SIZE; }
int insert(int key, int value) {
    int idx = hash(key);
    while (table[idx].used)
        idx = (idx + 1) % SIZE;
    table[idx] = (Entry){key, value, 1};
}

```

Load Factor  $\alpha = \frac{n}{m}$  affects performance , when too high, rehash to larger size.

### 3. Modern Variants

Classic hash tables can degrade under heavy collisions. Modern variants reduce probe chains and balance load more evenly.

## A. Cuckoo Hashing

Idea: Each key has two possible locations , if both full, evict one (“kick out the cuckoo”) and reinsert. Ensures constant lookup , at most two probes.

Steps:

1. Compute two hashes ( $h\_1(key)$ ), ( $h\_2(key)$ )
2. If slot 1 empty → place
3. Else evict occupant, reinsert it using alternate hash
4. Repeat until placed or cycle detected (rehash if needed)

Code Sketch (Conceptual):

```
int h1(int key) { return key % SIZE; }
int h2(int key) { return (key / SIZE) % SIZE; }

void insert(int key) {
    int pos1 = h1(key);
    if (!table1[pos1]) { table1[pos1] = key; return; }
    int displaced = table1[pos1]; table1[pos1] = key;

    int pos2 = h2(displaced);
    if (!table2[pos2]) { table2[pos2] = displaced; return; }
    // continue evicting if needed
}
```

Pros:

- Worst-case O(1) lookup (constant probes)- Predictable latency Cons:
- Rehash needed on insertion failure- More complex logic Used in high-performance caches and real-time systems.

## B. Robin Hood Hashing

Idea: Steal slots from richer (closer) keys to ensure fairness. When inserting, if you find someone with smaller probe distance, swap , “steal from the rich, give to the poor.”

This balances probe lengths and improves variance and average lookup time.

Key Principle:

If  $\text{new\_probe\_distance} > \text{existing\_probe\_distance} \Rightarrow \text{swap}$

Code Sketch:

```
int insert(int key) {
    int idx = hash(key);
    int dist = 0;
    while (table[idx].used) {
        if (table[idx].dist < dist) {
            // swap entries
            Entry tmp = table[idx];
            table[idx] = (Entry){key, dist, 1};
            key = tmp.key;
            dist = tmp.dist;
        }
        idx = (idx + 1) % SIZE;
        dist++;
    }
    table[idx] = (Entry){key, dist, 1};
}
```

Pros:

- Reduced variance- Better performance under high load Cons:
- Slightly slower insertion Used in modern languages like Rust (`hashbrown`) and Swift.

## C. Consistent Hashing

Idea: When distributing keys across multiple nodes, you want minimal movement when adding/removing a node. Consistent hashing maps both keys and nodes onto a circular hash ring.

Steps:

1. Hash nodes into a ring
2. Hash keys into same ring
3. Each key belongs to the next node clockwise

When a node is added or removed, only nearby keys move.

Used in:

- Distributed caches (Memcached, DynamoDB)- Load balancing- Sharding in databases

Code (Conceptual):

```

Ring: 0 ----- 2^32
Nodes: N1 at hash("A"), N2 at hash("B")
Key: hash("User42") → assign to next node clockwise

```

Pros:

- Minimal rebalancing- Scalable Cons:
- More complex setup- Requires virtual nodes for even distribution

#### 4. Complexity Overview

Variant	Insert	Search	Delete	Memory	Notes
Chaining	O(1) avg	O(1) avg	O(1) avg	High	Simple, dynamic
Linear Probing	O(1) avg	O(1) avg	O(1) avg	Low	Clustering risk
Cuckoo	O(1)	O(1)	O(1)	Medium	Two tables, predictable
Robin Hood	O(1)	O(1)	O(1)	Low	Balanced probes
Consistent	O(log n)	O(log n)	O(log n)	De- pends	Distributed keys

#### Tiny Code

Simple hash table with linear probing:

```

#define SIZE 10
int keys[SIZE], values[SIZE], used[SIZE];

int hash(int key) { return key % SIZE; }

void insert(int key, int value) {
    int idx = hash(key);
    while (used[idx]) idx = (idx + 1) % SIZE;
    keys[idx] = key; values[idx] = value; used[idx] = 1;
}

```

Lookup:

```

int get(int key) {
    int idx = hash(key);
    while (used[idx]) {
        if (keys[idx] == key) return values[idx];
        idx = (idx + 1) % SIZE;
    }
    return -1;
}

```

## Why It Matters

Hash tables show how structure and randomness combine for speed. They embody the idea that a good hash function + smart collision handling = near-constant performance.

Variants like Cuckoo and Robin Hood are examples of modern engineering trade-offs , balancing performance, memory, and predictability. Consistent hashing extends these ideas to distributed systems.

## Try It Yourself

1. Implement a hash table with chaining and test collision handling.
2. Modify it to use linear probing , measure probe lengths.
3. Simulate Cuckoo hashing with random inserts.
4. Implement Robin Hood swapping logic , observe fairness.
5. Draw a consistent hash ring with 3 nodes and 10 keys , track movement when adding one node.

Once you master these, you'll see hashing everywhere , from dictionaries to distributed databases.

## 23. Heaps (Binary, Fibonacci, Pairing)

Heaps are priority-driven data structures , they always give you fast access to the most important element, typically the minimum or maximum. They're essential for priority queues, scheduling, graph algorithms (like Dijkstra), and streaming analytics.

In this section, we'll start from the basic binary heap and then explore more advanced ones like Fibonacci and pairing heaps, which trade off simplicity, speed, and amortized guarantees.

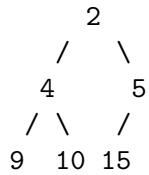
## 1. The Heap Property

A heap is a tree-based structure (often represented as an array) that satisfies:

- Min-Heap: Every node < its children- Max-Heap: Every node > its children This ensures the root always holds the smallest (or largest) element.

Complete Binary Tree: All levels filled except possibly the last, which is filled left to right.

Example (Min-Heap):



Here, the smallest element (2) is at the root.

## 2. Binary Heap

Storage: Stored compactly in an array. For index  $i$  (0-based):

- Parent =  $(i - 1) / 2$ - Left child =  $2i + 1$ - Right child =  $2i + 2$  Operations:

Operation	Description	Time
push(x)	Insert element	$O(\log n)$
pop()	Remove root	$O(\log n)$
peek()	Get root	$O(1)$
heapify()	Build heap	$O(n)$

### A. Insertion (Push)

Insert at the end, then bubble up until heap property is restored.

Code:

```

void push(int heap[], int *n, int x) {
    int i = (*n)++;
    heap[i] = x;
    while (i > 0 && heap[(i - 1)/2] > heap[i]) {
        int tmp = heap[i];
        heap[i] = heap[(i - 1)/2];
        heap[(i - 1)/2] = tmp;
        i = (i - 1) / 2;
    }
}

```

## B. Removal (Pop)

Replace root with last element, then bubble down (heapify).

Code:

```

void heapify(int heap[], int n, int i) {
    int smallest = i, l = 2*i + 1, r = 2*i + 2;
    if (l < n && heap[l] < heap[smallest]) smallest = l;
    if (r < n && heap[r] < heap[smallest]) smallest = r;
    if (smallest != i) {
        int tmp = heap[i]; heap[i] = heap[smallest]; heap[smallest] = tmp;
        heapify(heap, n, smallest);
    }
}

```

Pop:

```

int pop(int heap[], int *n) {
    int root = heap[0];
    heap[0] = heap[--(*n)];
    heapify(heap, *n, 0);
    return root;
}

```

## C. Building a Heap

Heapify bottom-up from last non-leaf: ( $O(n)$ )

```

for (int i = n/2 - 1; i >= 0; i--)
    heapify(heap, n, i);

```

## D. Applications

- Heapsort: Repeatedly pop min ( $O(n \log n)$ )- Priority Queue: Fast access to smallest/largest- Graph Algorithms: Dijkstra, Prim- Streaming: Median finding using two heaps

### 3. Fibonacci Heap

Idea: A heap optimized for algorithms that do many decrease-key operations (like Dijkstra's). It stores a collection of trees with lazy merging, giving amortized bounds:

Operation	Amortized Time
Insert	(O(1))
Find-Min	(O(1))
Extract-Min	(Olog n)
Decrease-Key	(O(1))
Merge	(O(1))

It achieves this by delaying structural fixes until absolutely necessary (using potential method in amortized analysis).

Structure:

- A circular linked list of roots- Each node can have multiple children- Consolidation on extract-min ensures minimal degree duplication Used in theoretical optimizations where asymptotic complexity matters (e.g. Dijkstra in  $(OE + V\log V)$  vs  $(OE\log V)$ ).

### 4. Pairing Heap

Idea: A simpler, practical alternative to Fibonacci heaps. Self-adjusting structure using a tree with multiple children.

Operations:

- Insert: (O(1))- Extract-Min: (Olog n) amortized- Decrease-Key: (Olog n) amortized Steps:

- **merge** two heaps: attach one as child of the other- **extract-min**: remove root, merge children in pairs, then merge all results Why It's Popular:
- Easier to implement- Great real-world performance- Used in functional programming and priority schedulers

## 5. Comparison

Heap Type	Insert	Extract-Min	Decrease-Key	Merge	Simplicity	Use Case
Binary Heap	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	Easy	General-purpose
Fibonacci Heap	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$	Complex	Theoretical optimality
Pairing Heap	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$	Moderate	Practical alternative

## Tiny Code

Binary Heap Demo:

```
int heap[100], n = 0;
push(heap, &n, 10);
push(heap, &n, 4);
push(heap, &n, 7);
printf("%d ", pop(heap, &n)); // 4
```

Output: 4

## Why It Matters

Heaps show how to prioritize elements dynamically. From sorting to scheduling, they're the backbone of many “choose the best next” algorithms. Variants like Fibonacci and Pairing Heaps demonstrate how amortized analysis can unlock deeper efficiency , crucial in graph theory and large-scale optimization.

## Try It Yourself

1. Implement a binary min-heap with `push`, `pop`, and `peek`.
2. Use a heap to sort a list (Heapsort).
3. Build a priority queue for task scheduling.
4. Study how Dijkstra changes when replacing arrays with heaps.
5. Explore Fibonacci heap pseudo-code , trace `decrease-key`.

Mastering heaps gives you a deep sense of priority-driven design , how to keep “the best” element always within reach.

## 24. Balanced Trees (AVL, Red-Black, Splay, Treap)

Unbalanced trees can degrade into linear lists, turning your beautiful ( $O(\log n)$ ) search into a sad ( $O(n)$ ) crawl. Balanced trees solve this , they keep the height logarithmic, guaranteeing fast lookups, insertions, and deletions.

In this section, you’ll learn how different balancing philosophies work , AVL (strict balance), Red-Black (relaxed balance), Splay (self-adjusting), and Treap (randomized balance).

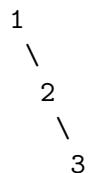
### 1. The Idea of Balance

For a binary search tree (BST):

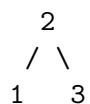
$$\text{height} = O(\log n)$$

only if it’s balanced , meaning the number of nodes in left and right subtrees differ by a small factor.

Unbalanced BST (bad):



Balanced BST (good):



Balance ensures efficient:

- $\text{search}(x) \rightarrow (\text{Olog } n)$ -  $\text{insert}(x) \rightarrow (\text{Olog } n)$ -  $\text{delete}(x) \rightarrow (\text{Olog } n)$

## 2. AVL Tree (Adelson-Velsky & Landis)

Invented in 1962, AVL is the first self-balancing BST. It enforces strict balance:

$$|\text{height}(\text{left}) - \text{height}(\text{right})| \leq 1$$

Whenever this condition breaks, rotations fix it.

Rotations:

- LL (Right Rotation): imbalance on left-left- RR (Left Rotation): imbalance on right-right- LR / RL: double rotation cases Code (Rotation Example):

```
Node* rotateRight(Node* y) {
    Node* x = y->left;
    Node* T = x->right;
    x->right = y;
    y->left = T;
    return x;
}
```

Height & Balance Factor:

```
int height(Node* n) { return n ? n->h : 0; }
int balance(Node* n) { return height(n->left) - height(n->right); }
```

Properties:

- Strict height bound: ( $\text{Olog } n$ )- More rotations (slower insertions)- Excellent lookup speed  
Used when lookups  $>$  updates (databases, indexing).

### 3. Red-Black Tree

Idea: A slightly looser balance for faster insertions. Each node has a color (Red/Black) with these rules:

1. Root is black
2. Red node's children are black
3. Every path has same number of black nodes
4. Null nodes are black

Balance through color flips + rotations

Compared to AVL:

- Fewer rotations (faster insert/delete)- Slightly taller (slower lookup)- Simpler amortized balance Used in:
- C++ `std::map`, `std::set`- Java `TreeMap`, Linux scheduler Complexity: All major operations ( $O(\log n)$ )

### 4. Splay Tree

Idea: Bring recently accessed node to root via splaying (rotations). It adapts to access patterns , the more you access a key, the faster it becomes.

Splaying Steps:

- Zig: one rotation (root child)- Zig-Zig: two rotations (same side)- Zig-Zag: two rotations (different sides) Code (Conceptual):

```
Node* splay(Node* root, int key) {
    if (!root || root->key == key) return root;
    if (key < root->key) {
        if (!root->left) return root;
        // splay in left subtree
        if (key < root->left->key)
            root->left->left = splay(root->left->left, key),
            root = rotateRight(root);
        else if (key > root->left->key)
            root->left->right = splay(root->left->right, key),
            root->left = rotateLeft(root->left);
        return rotateRight(root);
    } else {
        if (!root->right) return root;
```

```

        // symmetric
    }
}

```

Why It's Cool: No strict balance, but amortized ( $O(\log n)$ ). Frequently accessed elements stay near top.

Used in self-adjusting caches, rope data structures, memory allocators.

## 5. Treap (Tree + Heap)

Idea: Each node has two keys:

- BST key → order property- Priority → heap property Insertion = normal BST insert + heap fix via rotation.

Balance comes from randomization , random priorities ensure expected ( $O(\log n)$ ) height.

Code Sketch:

```

typedef struct Node {
    int key, priority;
    struct Node *left, *right;
} Node;

Node* insert(Node* root, int key) {
    if (!root) return newNode(key, rand());
    if (key < root->key) root->left = insert(root->left, key);
    else root->right = insert(root->right, key);

    if (root->left && root->left->priority > root->priority)
        root = rotateRight(root);
    if (root->right && root->right->priority > root->priority)
        root = rotateLeft(root);
    return root;
}

```

Advantages:

- Simple logic- Random balancing- Expected ( $O(\log n)$ ) Used in randomized algorithms and functional programming.

## 6. Comparison

Tree	Balance Type	Rotations	Height	Insert/Delete	Lookup	Notes
AVL	Strict	More	$(O \log n)$	Medium	Fast	Lookup-heavy
Red-Black	Relaxed	Fewer	$(O \log n)$	Fast	Medium	Library std
Splay	Adaptive	Variable	Amortized $(O \log n)$	Fast	Fast (amortized)	Access patterns
Treap	Randomized	Avg few	$(O \log n)$ expected	Simple	Simple	Probabilistic

## Tiny Code

AVL Insert (Skeleton):

```
Node* insert(Node* root, int key) {
    if (!root) return newNode(key);
    if (key < root->key) root->left = insert(root->left, key);
    else root->right = insert(root->right, key);
    root->h = 1 + max(height(root->left), height(root->right));
    int b = balance(root);
    if (b > 1 && key < root->left->key) return rotateRight(root);
    if (b < -1 && key > root->right->key) return rotateLeft(root);
    // other cases...
    return root;
}
```

## Why It Matters

Balanced trees guarantee predictable performance under dynamic updates. Each variant represents a philosophy:

- AVL: precision- Red-Black: practicality- Splay: adaptability- Treap: randomness Together, they teach one core idea , keep height in check, no matter the operations.

## Try It Yourself

1. Implement an AVL tree and visualize rotations.
2. Insert keys [10, 20, 30, 40, 50] and trace Red-Black color changes.
3. Splay after each access , see which keys stay near top.
4. Build a Treap with random priorities , measure average height.
5. Compare performance of BST vs AVL on sorted input.

Balanced trees are the architects of order , always keeping chaos one rotation away.

## 25. Segment Trees and Fenwick Trees

When you need to answer range queries quickly (like sum, min, max) and support updates to individual elements, simple prefix sums won't cut it anymore.

You need something smarter , data structures that can divide and conquer over ranges, updating and combining results efficiently.

That's exactly what Segment Trees and Fenwick Trees (Binary Indexed Trees) do:

- Query over a range in ( $O(\log n)$ ) - Update elements in ( $O(\log n)$ ) They're the backbone of competitive programming, signal processing, and database analytics.

### 1. The Problem

Given an array  $A[0..n-1]$ , support:

1. `update(i, x)` → change  $A[i]$  to  $x$
2. `query(L, R)` → compute sum (or min, max) of  $A[L..R]$

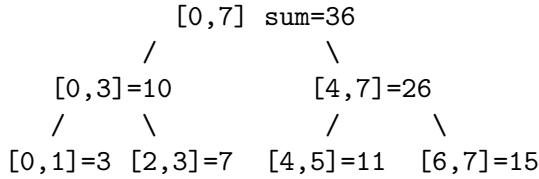
Naive approach:

- Update: ( $O(1)$ )- Query: ( $O(n)$ ) Prefix sums fix one but not both. Segment and Fenwick trees fix both.

### 2. Segment Tree

Idea: Divide the array into segments (intervals) recursively. Each node stores an aggregate (sum, min, max) of its range. You can combine child nodes to get any range result.

Structure (Sum Example):



Each node represents a range [L,R]. Leaf nodes = single elements.

## A. Build

Recursive Construction: Time: (O(n))

```

void build(int node, int L, int R) {
    if (L == R) tree[node] = arr[L];
    else {
        int mid = (L + R) / 2;
        build(2*node, L, mid);
        build(2*node+1, mid+1, R);
        tree[node] = tree[2*node] + tree[2*node+1];
    }
}

```

## B. Query (Range Sum)

Query [l, r] recursively:

- If current range [L, R] fully inside [l, r], return node value- If disjoint, return 0- Else combine children

```

int query(int node, int L, int R, int l, int r) {
    if (r < L || R < l) return 0;
    if (l <= L && R <= r) return tree[node];
    int mid = (L + R) / 2;
    return query(2*node, L, mid, l, r)
        + query(2*node+1, mid+1, R, l, r);
}

```

## C. Update

Change  $\text{arr}[i] = x$  and update tree nodes covering  $i$ .

```
void update(int node, int L, int R, int i, int x) {
    if (L == R) tree[node] = x;
    else {
        int mid = (L + R)/2;
        if (i <= mid) update(2*node, L, mid, i, x);
        else update(2*node+1, mid+1, R, i, x);
        tree[node] = tree[2*node] + tree[2*node+1];
    }
}
```

Complexities:

- Build:  $(O(n))$ - Query:  $(O\log n)$ - Update:  $(O\log n)$ - Space:  $(O(4n))$

## D. Variants

Segment trees are flexible:

- Range minimum/maximum- Range GCD- Lazy propagation → range updates- 2D segment tree for grids

### 3. Fenwick Tree (Binary Indexed Tree)

Idea: Stores cumulative frequencies using bit manipulation. Each node covers a range size = LSB(index).

Simpler, smaller, but supports only associative ops (sum, xor, etc.)

Indexing:

- Parent:  $i + (i \& -i)$ - Child:  $i - (i \& -i)$  Build: Initialize with zero, then add elements one by one.

Add / Update:

```
void add(int i, int x) {
    for (; i <= n; i += i & -i)
        bit[i] += x;
}
```

Prefix Sum:

```
int sum(int i) {
    int res = 0;
    for (; i > 0; i -= i & -i)
        res += bit[i];
    return res;
}
```

Range Sum [L, R]:

$$\text{sum}(R) - \text{sum}(L - 1)$$

Complexities:

- Build: ( $O(n \log n)$ )- Query: ( $O(\log n)$ )- Update: ( $O(\log n)$ )- Space: ( $O(n)$ )

#### 4. Comparison

Feature	Segment Tree	Fenwick Tree
Space	$O(4n)$	$O(n)$
Build	$O(n)$	$O(n \log n)$
Query	$O(\log n)$	$O(\log n)$
Update	$O(\log n)$	$O(\log n)$
Range Update	With Lazy	Tricky
Range Query	Flexible	Sum/XOR only
Implementation	Moderate	Simple

#### 5. Applications

- Sum / Min / Max / XOR queries- Frequency counts- Inversions counting- Order statistics- Online problems where array updates over time Used in:
- Competitive programming- Databases (analytics on changing data)- Time series queries- Games (damage/range updates)

#### Tiny Code

Fenwick Tree Example:

```

int bit[1001], n;

void update(int i, int val) {
    for (; i <= n; i += i & -i)
        bit[i] += val;
}

int query(int i) {
    int res = 0;
    for (; i > 0; i -= i & -i)
        res += bit[i];
    return res;
}

// range sum
int range_sum(int L, int R) { return query(R) - query(L - 1); }

```

## Why It Matters

Segment and Fenwick trees embody divide-and-conquer over data , balancing dynamic updates with range queries. They're how modern systems aggregate live data efficiently.

They teach a powerful mindset:

“If you can split a problem, you can solve it fast.”

## Try It Yourself

1. Build a segment tree for sum queries.
2. Add range minimum queries (RMQ).
3. Implement a Fenwick tree , test with prefix sums.
4. Solve: number of inversions in array using Fenwick tree.
5. Add lazy propagation to segment tree for range updates.

Once you master these, range queries will never scare you again , you'll slice through them in logarithmic time.

## 26. Disjoint Set Union (Union-Find)

Many problems involve grouping elements into sets and efficiently checking whether two elements belong to the same group , like connected components in a graph, network connectivity, Kruskal's MST, or even social network clustering.

For these, the go-to structure is the Disjoint Set Union (DSU), also called Union-Find. It efficiently supports two operations:

1. `find(x)` → which set does `x` belong to?
2. `union(x, y)` → merge the sets containing `x` and `y`.

With path compression and union by rank, both operations run in near-constant time, specifically  $O(\alpha(n))$ , where  $\alpha$  is the inverse Ackermann function (practically  $< 4$ ).

### 1. The Problem

Suppose you have  $(n)$  elements initially in separate sets. Over time, you want to:

- Merge two sets- Check if two elements share the same set Example:

```
Sets: {1}, {2}, {3}, {4}, {5}
Union(1,2) → {1,2}, {3}, {4}, {5}
Union(3,4) → {1,2}, {3,4}, {5}
Find(2) == Find(1)? Yes
Find(5) == Find(3)? No
```

### 2. Basic Implementation

Each element has a parent pointer. Initially, every node is its own parent.

Parent array representation:

```
int parent[N];

void make_set(int v) {
    parent[v] = v;
}

int find(int v) {
    if (v == parent[v]) return v;
    return find(parent[v]);
}
```

```

void union_sets(int a, int b) {
    a = find(a);
    b = find(b);
    if (a != b)
        parent[b] = a;
}

```

This works, but deep trees can form , making `find` slow. We fix that with path compression.

### 3. Path Compression

Every time we call `find(v)`, we make all nodes along the path point directly to the root. This flattens the tree dramatically.

Optimized Find:

```

int find(int v) {
    if (v == parent[v]) return v;
    return parent[v] = find(parent[v]);
}

```

So next time, lookups will be ( $O(1)$ ) for those nodes.

### 4. Union by Rank / Size

When merging, always attach the smaller tree to the larger to keep depth small.

Union by Rank:

```

int parent[N], rank[N];

void make_set(int v) {
    parent[v] = v;
    rank[v] = 0;
}

void union_sets(int a, int b) {
    a = find(a);
    b = find(b);
    if (a != b) {
        if (rank[a] < rank[b])

```

```

        swap(a, b);
parent[b] = a;
if (rank[a] == rank[b])
    rank[a]++;
}
}

```

Union by Size (Alternative): Track size of each set and attach smaller to larger.

```

int size[N];
void union_sets(int a, int b) {
    a = find(a);
    b = find(b);
    if (a != b) {
        if (size[a] < size[b]) swap(a, b);
        parent[b] = a;
        size[a] += size[b];
    }
}

```

## 5. Complexity

With both path compression and union by rank, all operations are effectively constant time:

$$O(\alpha(n)) \approx O(1)$$

For all practical ( $n$ ),  $O(n)$ .

Operation	Time
Make set	$O(1)$
Find	$O(\log n)$
Union	$O(\log n)$

## 6. Applications

- Graph Connectivity: determine connected components- Kruskal's MST: add edges, avoid cycles- Dynamic connectivity- Image segmentation- Network clustering- Cycle detection in undirected graphs Example: Kruskal's Algorithm

```

sort(edges.begin(), edges.end());
for (edge e : edges)
    if (find(e.u) != find(e.v)) {
        union_sets(e.u, e.v);
        mst_weight += e.w;
    }
}

```

## 7. Example

```

int parent[6], rank[6];

void init() {
    for (int i = 1; i <= 5; i++) {
        parent[i] = i;
        rank[i] = 0;
    }
}

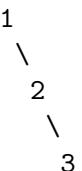
int main() {
    init();
    union_sets(1, 2);
    union_sets(3, 4);
    union_sets(2, 3);
    printf("%d\n", find(4)); // prints representative of {1,2,3,4}
}

```

Result: {1,2,3,4}, {5}

## 8. Visualization

Before compression:



After compression:

1

2  
3

Every `find` call makes future queries faster.

## 9. Comparison

Variant	Find	Union	Notes
Basic	$O(n)$	$O(n)$	Deep trees
Path Compression	$O(\log n)$	$O(\log n)$	Very fast
+ Rank / Size	$O(\log n)$	$O(\log n)$	Balanced
Persistent DSU	$O(\log n)$	$O(\log n)$	Undo/rollback support

## Tiny Code

Full DSU with path compression + rank:

```
int parent[1000], rank[1000];

void make_set(int v) {
    parent[v] = v;
    rank[v] = 0;
}

int find(int v) {
    if (v != parent[v])
        parent[v] = find(parent[v]);
    return parent[v];
}

void union_sets(int a, int b) {
    a = find(a);
    b = find(b);
    if (a != b) {
        if (rank[a] < rank[b]) swap(a, b);
        parent[b] = a;
        if (rank[a] == rank[b])
            rank[a]++;
    }
}
```

## Why It Matters

Union-Find embodies structural sharing and lazy optimization , you don't balance eagerly, but just enough. It's one of the most elegant demonstrations of how constant-time algorithms are possible through clever organization.

It teaches a key algorithmic lesson:

“Work only when necessary, and fix structure as you go.”

## Try It Yourself

1. Implement DSU and test `find/union`.
2. Build a program that counts connected components.
3. Solve Kruskal's MST using DSU.
4. Add `get_size(v)` to return component size.
5. Try rollback DSU (keep stack of changes).

Union-Find is the quiet powerhouse behind many graph and connectivity algorithms , simple, fast, and deeply elegant.

## 27. Probabilistic Data Structures (Bloom, Count-Min, HyperLogLog)

When you work with massive data streams , billions of elements, too big for memory , you can't store everything. But what if you don't need *perfect* answers, just *fast and tiny approximate ones*?

That's where probabilistic data structures shine. They trade a bit of accuracy for huge space savings and constant-time operations.

In this section, we'll explore three of the most famous:

- Bloom Filters → membership queries- Count-Min Sketch → frequency estimation- HyperLogLog → cardinality estimation Each of them answers “How likely is X?” or “How many?” efficiently , perfect for modern analytics, caching, and streaming systems.

### 1. Bloom Filter , “Is this element probably in the set?”

A Bloom filter answers:

“Is x in the set?” with either maybe yes or definitely no.

No false negatives, but *some* false positives.

## A. Idea

Use an array of bits (size  $m$ ), all initialized to 0. Use  $k$  different hash functions.

To insert an element:

1. Compute  $k$  hashes: (  $h_1(x)$ ,  $h_2(x)$ , ...,  $h_k(x)$  )
2. Set each bit position  $b_i = 1$

To query an element:

1. Compute same  $k$  hashes
2. If all bits are 1  $\rightarrow$  maybe yes
3. If any bit is 0  $\rightarrow$  definitely no

## B. Example

Insert **dog**:

- ( $h_1(\text{dog})=2$ ,  $h_2(\text{dog})=5$ ,  $h_3(\text{dog})=9$ ) Set bits 2, 5, 9  $\rightarrow 1$

Check **cat**:

- If any hash bit = 0  $\rightarrow$  not present

## C. Complexity

Operation	Time	Space	Accuracy
Insert	$O(k)$	$O(m)$	Tunable
Query	$O(k)$	$O(m)$	False positives

False positive rate  $(1 - e^{-kn/m})^k$

Choose  $m$  and  $k$  based on expected  $n$  and acceptable error.

## D. Code

```

#define M 1000
int bitset[M];

int hash1(int x) { return (x * 17) % M; }
int hash2(int x) { return (x * 31 + 7) % M; }

void add(int x) {
    bitset[hash1(x)] = 1;
    bitset[hash2(x)] = 1;
}

bool contains(int x) {
    return bitset[hash1(x)] && bitset[hash2(x)];
}

```

Used in:

- Caches (check before disk lookup)- Spam filters- Databases (join filtering)- Blockchain and peer-to-peer networks

## 2. Count-Min Sketch , “How often has this appeared?”

Tracks frequency counts in a stream, using sub-linear memory.

Instead of a full table, it uses a 2D array of counters, each row hashed with a different hash function.

### A. Insert

For each row  $i$ :

- Compute hash ( $h_i(x)$ )- Increment `count[i][h_i(x)]++ #####`

For element  $x$ :

- Compute all ( $h_i(x)$ )- Take `min(count[i][h_i(x)])` across rows  $\rightarrow$  gives an upper-bounded estimate of true frequency

### C. Code

```

#define W 1000
#define D 5
int count[D][W];

int hash(int i, int x) {
    return (x * (17*i + 3)) % W;
}

void add(int x) {
    for (int i = 0; i < D; i++)
        count[i][hash(i, x)]++;
}

int query(int x) {
    int res = INT_MAX;
    for (int i = 0; i < D; i++)
        res = min(res, count[i][hash(i, x)]);
    return res;
}

```

## D. Complexity

	Operation	Time	Space
Insert	O(D)	O(W×D)	
Query	O(D)	O(W×D)	

Error controlled by:

$$\varepsilon = \frac{1}{W}, \quad \delta = 1 - e^{-D}$$

Used in:

- Frequency counting in streams- Hot-key detection- Network flow analysis- Trending topics

## 3. HyperLogLog , “How many unique items?”

Estimates cardinality (number of distinct elements) with very small memory (~1.5 KB for millions).

## A. Idea

Hash each element uniformly → 32-bit value. Split hash into:

- Prefix bits → bucket index- Suffix bits → count leading zeros Each bucket stores the max leading zero count seen. At the end, use harmonic mean of counts to estimate distinct values.

## B. Formula

$$E = \alpha_m \cdot m^2 \cdot \left( \sum_{i=1}^m 2^{-M[i]} \right)^{-1}$$

where  $M[i]$  is the zero count in bucket  $i$ , and  $\alpha_m$  is a correction constant.

Accuracy:  $\sim 1.04 / \sqrt{m}$

## C. Complexity

Operation	Time	Space	Error
Add	$O(1)$	$O(m)$	$\sim 1.04 / \sqrt{m}$
Merge	$O(m)$	$O(m)$	,

Used in:

- Web analytics (unique visitors)- Databases (COUNT DISTINCT)- Distributed systems (mergeable estimates)

## 4. Comparison

Structure	Purpose	Query	Memory	Error	Notes
Bloom	Membership	$O(k)$	Tiny	False positives	No deletions
Count-Min	Frequency	$O(D)$	Small	Overestimate	Streaming counts
Hyper-LogLog	Cardinality	$O(1)$	Very small	$\sim 1\%$	Mergeable

## Tiny Code

Bloom Filter Demo:

```
add(42);
add(17);
printf("%d\n", contains(42)); // 1 (maybe yes)
printf("%d\n", contains(99)); // 0 (definitely no)
```

## Why It Matters

Probabilistic data structures show how approximation beats impossibility when resources are tight. They make it feasible to process massive streams in real time, when storing everything is impossible.

They teach a deeper algorithmic truth:

“A bit of uncertainty can buy you a world of scalability.”

## Try It Yourself

1. Implement a Bloom filter with 3 hash functions.
2. Measure false positive rate for 10K elements.
3. Build a Count-Min Sketch and test frequency estimation.
4. Approximate unique elements using HyperLogLog logic.
5. Explore real-world systems: Redis (Bloom/CM Sketch), PostgreSQL (HyperLogLog).

These tiny probabilistic tools are how big data becomes tractable.

## 28. Skip Lists and B-Trees

When you want fast search, insert, and delete but need a structure that's easier to code than trees or optimized for disk and memory blocks, two clever ideas step in:

- Skip Lists → randomized, layered linked lists that behave like balanced BSTs- B-Trees → multi-way trees that minimize disk I/O and organize large data blocks Both guarantee  $O(\log n)$  operations, but they shine in very different environments , Skip Lists in-memory, B-Trees on disk.

## 1. Skip Lists

Invented by: William Pugh (1990) Goal: Simulate binary search using linked lists with probabilistic shortcuts.

### A. Idea

A skip list is a stack of linked lists, each level skipping over more elements.

Example:

```
Level 3:          > 50
Level 2:    > 10   > 30   > 50
Level 1:  5  > 10  > 20  > 30  > 40  > 50
```

Higher levels are sparser and let you “skip” large chunks of the list.

You search top-down:

- Move right while next target- Drop down when you can't go further This mimics binary search , logarithmic layers, logarithmic hops.

### B. Construction

Each inserted element is given a random height, with geometric distribution:

- Level 1 (base) always exists- Level 2 with probability  $\frac{1}{2}$ - Level 3 with  $\frac{1}{4}$ , etc. Expected total nodes =  $2n$ , Expected height =  $(O \log n)$

### C. Operations

Operation	Time	Space	Notes
Search	$(O \log n)$	$O(n)$	Randomized balance
Insert	$(O \log n)$	$O(n)$	Rebuild towers
Delete	$(O \log n)$	$O(n)$	Rewire pointers

Search Algorithm:

```

Node* search(SkipList* sl, int key) {
    Node* cur = sl->head;
    for (int lvl = sl->level; lvl >= 0; lvl--) {
        while (cur->forward[lvl] && cur->forward[lvl]->key < key)
            cur = cur->forward[lvl];
    }
    cur = cur->forward[0];
    if (cur && cur->key == key) return cur;
    return NULL;
}

```

Skip Lists are simple, fast, and probabilistically balanced , no rotations, no rebalancing.

## D. Why Use Skip Lists?

- Easier to implement than balanced trees- Support concurrent access well- Randomized, not deterministic , but highly reliable Used in:
- Redis (sorted sets)- LevelDB / RocksDB internals- Concurrent maps

## 2. B-Trees

Invented by: Rudolf Bayer & Ed McCreight (1972) Goal: Reduce disk access by grouping data in blocks.

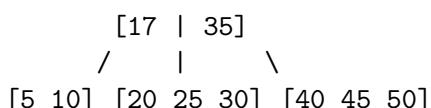
A B-Tree is a generalization of a BST:

- Each node holds multiple keys and children- Keys are kept sorted- Child subtrees span ranges between keys

## A. Structure

A B-Tree of order  $m$ :

- Each node has  $m$  children- Each internal node has  $k-1$  keys if it has  $k$  children- All leaves at the same depth Example (order 3):



## B. Operations

1. Search
  - Traverse from root - Binary search in each node's key array - Follow appropriate child  $\rightarrow (O\log_m n)$
2. Insert
  - Insert in leaf - If overflow  $\rightarrow$  split node - Promote median key to parent
3. Delete
  - Borrow or merge if node underflows Each split or merge keeps height minimal.

## C. Complexity

Operation	Time	Disk Accesses	Notes
Search	$(O\log_m n)$	$(O\log_m n)$	$m = \text{branching factor}$
Insert	$(O\log_m n)$	$(O(1))$ splits	Balanced
Delete	$(O\log_m n)$	$(O(1))$ merges	Balanced

Height  $\log_m n \rightarrow$  very shallow when ( $m$ ) large (e.g. 100).

## D. B+ Tree Variant

In B+ Trees:

- All data in leaves (internal nodes = indexes)- Leaves linked  $\rightarrow$  efficient range queries  
Used in:
  - Databases (MySQL, PostgreSQL)- File systems (NTFS, HFS+)- Key-value stores

## E. Example Flow

Insert 25:

```
[10 | 20 | 30] → overflow
Split → [10] [30]
Promote 20
Root: [20]
```

### 3. Comparison

Feature	Skip List	B-Tree
Balancing	Randomized	Deterministic
Fanout	2 (linked)	m-way
Environment	In-memory	Disk-based
Search	$O(\log n)$	$Olog_m n$
Insert/Delete	$O(\log n)$	$Olog_m n$
Concurrency	Easy	Complex
Range Queries	Sequential scan	Linked leaves (B+)

### Tiny Code

Skip List Search (Conceptual):

```
Node* search(SkipList* list, int key) {
    Node* cur = list->head;
    for (int lvl = list->level; lvl >= 0; lvl--) {
        while (cur->next[lvl] && cur->next[lvl]->key < key)
            cur = cur->next[lvl];
    }
    cur = cur->next[0];
    return (cur && cur->key == key) ? cur : NULL;
}
```

B-Tree Node (Skeleton):

```
#define M 4
typedef struct {
    int keys[M-1];
    Node* child[M];
    int n;
} Node;
```

### Why It Matters

Skip Lists and B-Trees show two paths to balance:

- Randomized simplicity (Skip List)- Block-based order (B-Tree) Both offer logarithmic guarantees, but one optimizes pointer chasing, the other I/O.

They're fundamental to:

- In-memory caches (Skip List)- On-disk indexes (B-Tree, B+ Tree)- Sorted data structures across systems

### Try It Yourself

1. Build a basic skip list and insert random keys.
2. Trace a search path across levels.
3. Implement B-Tree insert and split logic.
4. Compare height of BST vs B-Tree for 1,000 keys.
5. Explore how Redis and MySQL use these internally.

Together, they form the bridge between linked lists and balanced trees, uniting speed, structure, and scalability.

## 29. Persistent and Functional Data Structures

Most data structures are ephemeral , when you update them, the old version disappears. But sometimes, you want to keep all past versions, so you can go back in time, undo operations, or run concurrent reads safely.

That's the magic of persistent data structures: every update creates a new version while sharing most of the old structure.

This section introduces the idea of persistence, explores how to make classic structures like arrays and trees persistent, and explains why functional programming loves them.

### 1. What Is Persistence?

A persistent data structure preserves previous versions after updates. You can access any version , past or present , without side effects.

Three levels:

Type	Description	Example
Partial	Can access past versions, but only modify the latest	Undo stack
Full	Can access and modify any version	Immutable map
Confluent	Can combine different versions	Git-like merges

This is essential in functional programming, undo systems, version control, persistent segment trees, and immutable databases.

## 2. Ephemeral vs Persistent

Ephemeral:

```
arr[2] = 7; // old value lost forever
```

Persistent:

```
new_arr = update(arr, 2, 7); // old_arr still exists
```

Persistent structures use structural sharing , unchanged parts are reused, not copied.

## 3. Persistent Linked List

Easiest example: each update creates a new head, reusing the tail.

```
struct Node { int val; Node* next; };

Node* push(Node* head, int x) {
    Node* newHead = malloc(sizeof(Node));
    newHead->val = x;
    newHead->next = head;
    return newHead;
}
```

Now both `old_head` and `new_head` coexist. Each version is immutable , you never change existing nodes.

Access: old and new lists share most of their structure:

```
v0: 1 → 2 → 3
v1: 0 → 1 → 2 → 3
```

Only one new node was created.

## 4. Persistent Binary Tree

For trees, updates create new paths from the root to the modified node, reusing the rest.

```

typedef struct Node {
    int key;
    struct Node *left, *right;
} Node;

Node* update(Node* root, int pos, int val) {
    if (!root) return newNode(val);
    Node* node = malloc(sizeof(Node));
    *node = *root; // copy
    if (pos < root->key) node->left = update(root->left, pos, val);
    else node->right = update(root->right, pos, val);
    return node;
}

```

Each `update` creates a new version , only ( $O(\log n)$ ) new nodes per change.

This is the core of persistent segment trees used in competitive programming.

## 5. Persistent Array (Functional Trick)

Arrays are trickier because of random access. Solutions:

- Use balanced binary trees as array replacements- Each update replaces one node- Persistent vector = tree of small arrays (used in Clojure, Scala) This gives:
- Access: ( $O(\log n)$ )- Update: ( $O(\log n)$ )- Space: ( $O(\log n)$ ) per update

## 6. Persistent Segment Tree

Used for versioned range queries:

- Each update = new root- Each version = snapshot of history Example: Track how array changes over time, query “sum in range [L,R] at version t”.

Build:

```

Node* build(int L, int R) {
    if (L == R) return newNode(arr[L]);
    int mid = (L+R)/2;
    return newNode(
        build(L, mid),
        build(mid+1, R),

```

```

        sum
    );
}

```

Update: only  $O(\log n)$  new nodes

```

Node* update(Node* prev, int L, int R, int pos, int val) {
    if (L == R) return newNode(val);
    int mid = (L+R)/2;
    if (pos <= mid)
        return newNode(update(prev->left, L, mid, pos, val), prev->right);
    else
        return newNode(prev->left, update(prev->right, mid+1, R, pos, val));
}

```

Each version = new root; old ones still valid.

## 7. Functional Perspective

In functional programming, data is immutable by default. Instead of mutating, you create a new version.

This allows:

- Thread-safety (no races)- Time-travel debugging- Undo/redo systems- Concurrency without locks Languages like Haskell, Clojure, and Elm build everything this way.

For example, Clojure's `persistent vector` uses path copying and branching factor 32 for  $(O(\log_{32} n))$  access.

## 8. Applications

- Undo / Redo stacks (text editors, IDEs)- Version control (Git trees)- Immutable databases (Datomic)- Segment trees over time (competitive programming)- Snapshots in memory allocators or games

## 9. Complexity

Structure	Update	Access	Space per Update	Notes
Persistent Linked List	O(1)	O(1)	O(1)	Simple sharing
Persistent Tree	O(log n)	O(log n)	O(log n)	Path copying
Persistent Array	O(log n)	O(log n)	O(log n)	Tree-backed
Persistent Segment Tree	O(log n)	O(log n)	O(log n)	Versioned queries

## Tiny Code

Persistent Linked List Example:

```
Node* v0 = NULL;
v0 = push(v0, 3);
v0 = push(v0, 2);
Node* v1 = push(v0, 1);
// v0 = [2,3], v1 = [1,2,3]
```

## Why It Matters

Persistence is about time as a first-class citizen. It lets you:

- Roll back- Compare versions- Work immutably and safely It's the algorithmic foundation behind functional programming, time-travel debugging, and immutable data systems.

It teaches this powerful idea:

“Never destroy , always build upon what was.”

## Try It Yourself

1. Implement a persistent stack using linked lists.
2. Write a persistent segment tree for range sums.
3. Track array versions after each update and query old states.
4. Compare space/time with an ephemeral one.
5. Explore persistent structures in Clojure (`conj`, `assoc`) or Rust (`im` crate).

Persistence transforms data from fleeting state into a history you can navigate , a timeline of structure and meaning.

## 30. Advanced Trees and Range Queries

So far, you've seen balanced trees (AVL, Red-Black, Treap) and segment-based structures (Segment Trees, Fenwick Trees). Now it's time to combine those ideas and step into advanced trees , data structures that handle dynamic sets, order statistics, intervals, ranges, and geometry-like queries in logarithmic time.

This chapter is about trees that go beyond search , they store order, track ranges, and answer complex queries efficiently.

We'll explore:

- Order Statistic Trees (k-th element, rank queries)- Interval Trees (range overlaps)- Range Trees (multi-dimensional search)- KD-Trees (spatial partitioning)- Merge Sort Trees (offline range queries)

### 1. Order Statistic Tree

Goal: find the k-th smallest element, or the rank of an element, in  $(O\log n)$ .

Built on top of a balanced BST (e.g. Red-Black) by storing subtree sizes.

#### A. Augmented Tree Nodes

Each node keeps:

- **key**: element value- **left**, **right**: children- **size**: number of nodes in subtree

```
typedef struct Node {
    int key, size;
    struct Node *left, *right;
} Node;
```

Whenever you rotate or insert, update **size**:

```
int get_size(Node* n) { return n ? n->size : 0; }
void update_size(Node* n) {
    if (n) n->size = get_size(n->left) + get_size(n->right) + 1;
}
```

## B. Find k-th Element

Recursively use subtree sizes:

```
Node* kth(Node* root, int k) {
    int left = get_size(root->left);
    if (k == left + 1) return root;
    else if (k <= left) return kth(root->left, k);
    else return kth(root->right, k - left - 1);
}
```

Time:  $(O \log n)$

## C. Find Rank

Find position of a key (number of smaller elements):

```
int rank(Node* root, int key) {
    if (!root) return 0;
    if (key < root->key) return rank(root->left, key);
    if (key > root->key) return get_size(root->left) + 1 + rank(root->right, key);
    return get_size(root->left) + 1;
}
```

Used in:

- Databases (ORDER BY, pagination)- Quantile queries- Online median maintenance

## 2. Interval Tree

Goal: find all intervals overlapping with a given point or range.

Used in computational geometry, scheduling, and genomic data.

## A. Structure

BST ordered by interval low endpoint. Each node stores:

- `low, high`: interval bounds- `max`: maximum `high` in its subtree

```

typedef struct {
    int low, high, max;
    struct Node *left, *right;
} Node;

```

## B. Query Overlap

Check if  $x$  overlaps  $\text{node}\rightarrow\text{interval}$ : If not, go left or right based on  $\text{max}$  values.

```

bool overlap(Interval a, Interval b) {
    return a.low <= b.high && b.low <= a.high;
}

Node* overlap_search(Node* root, Interval q) {
    if (!root) return NULL;
    if (overlap(root->interval, q)) return root;
    if (root->left && root->left->max >= q.low)
        return overlap_search(root->left, q);
    return overlap_search(root->right, q);
}

```

Time:  $(O \log n)$  average

## C. Use Cases

- Calendar/schedule conflict detection- Collision detection- Genome region lookup- Segment intersection

## 3. Range Tree

Goal: multi-dimensional queries like

“How many points fall inside rectangle  $[x_1, x_2] \times [y_1, y_2]$ ? ”

Structure:

- Primary BST on  $x$ - Each node stores secondary BST on  $y$  Query time:  $(O \log^2 n)$  Space:  $(O n \log n)$

Used in:

- 2D search- Computational geometry- Databases (spatial joins)

## 4. KD-Tree

Goal: efficiently search points in k-dimensional space.

Alternate splitting dimensions at each level:

- Level 0 → split by x- Level 1 → split by y- Level 2 → split by z Each node stores:
- Point (vector)- Split axis Used for:
- Nearest neighbor search- Range queries- ML (k-NN classifiers) Time:
- Build:  $(O(n \log n))$ - Query:  $(O(\sqrt{n}))$  average in 2D

## 5. Merge Sort Tree

Goal: query “number of elements  $k$  in range  $[L, R]$ ”

Built like a segment tree, but each node stores a sorted list of its range.

Build: merge children lists Query: binary search in node lists

Time:

- Build:  $(O(n \log n))$ - Query:  $(O(\log^2 n))$  Used in offline queries and order-statistics over ranges.

## 6. Comparison

Tree Type	Use Case	Query	Update	Notes
Order Statistic	k-th, rank	$O(\log n)$	$O(\log n)$	Augmented BST
Interval	Overlaps	$O(\log n + k)$	$O(\log n)$	Store intervals
Range Tree	2D range	$O(\log^2 n + k)$	$O(\log^2 n)$	Multi-dim
KD-Tree	Spatial	$O(\sqrt{n})$ avg	$O(\log n)$	Nearest neighbor
Merge Sort Tree	Offline rank	$O(\log^2 n)$	Static	Built from sorted segments

## Tiny Code

Order Statistic Example:

```

Node* root = NULL;
root = insert(root, 10);
root = insert(root, 20);
root = insert(root, 30);
printf("%d", kth(root, 2)->key); // 20

```

Interval Query:

```

Interval q = {15, 17};
Node* res = overlap_search(root, q);
if (res) printf("Overlap: [%d, %d]\n", res->low, res->high);

```

## Why It Matters

These trees extend balance into dimensions and ranges. They let you query ordered data efficiently: “How many?”, “Which overlaps?”, “Where is k-th smallest?”.

They teach a deeper design principle:

“Augment structure with knowledge , balance plus metadata equals power.”

## Try It Yourself

1. Implement an order statistic tree , test rank/k-th queries.
2. Insert intervals and test overlap detection.
3. Build a simple KD-tree for 2D points.
4. Solve rectangle counting with a range tree.
5. Precompute a merge sort tree for offline queries.

These advanced trees form the final evolution of structured queries , blending geometry, order, and logarithmic precision.

## Chapter 4. Graph Algorithms

### 31. Traversals (DFS, BFS, Iterative Deepening)

Graphs are everywhere , maps, networks, dependencies, state spaces. Before you can analyze them, you need a way to visit their vertices , systematically, without getting lost or looping forever.

That's where graph traversals come in. They're the foundation for everything that follows: connected components, shortest paths, spanning trees, topological sorts, and more.

This section walks through the three pillars:

- DFS (Depth-First Search) , explore deeply before backtracking- BFS (Breadth-First Search) , explore level by level- Iterative Deepening , a memory-friendly hybrid

## 1. Representing Graphs

Before traversal, you need a good structure.

Adjacency List (most common):

```
#define MAX 1000
vector<int> adj[MAX];
```

Add edges:

```
void add_edge(int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u); // omit if directed
}
```

Track visited vertices:

```
bool visited[MAX];
```

## 2. Depth-First Search (DFS)

DFS dives deep, following one branch fully before exploring others. It's recursive, like exploring a maze by always turning left until you hit a wall.

### A. Recursive Form

```

void dfs(int u) {
    visited[u] = true;
    for (int v : adj[u]) {
        if (!visited[v])
            dfs(v);
    }
}

```

Start it:

```
dfs(start_node);
```

## B. Iterative Form (with Stack)

```

void dfs_iter(int start) {
    stack<int> s;
    s.push(start);
    while (!s.empty()) {
        int u = s.top(); s.pop();
        if (visited[u]) continue;
        visited[u] = true;
        for (int v : adj[u]) s.push(v);
    }
}

```

## C. Complexity

Graph Type	Time	Space
Adjacency List	$O(V + E)$	$O(V)$

DFS is used in:

- Connected components- Cycle detection- Topological sort- Backtracking & search- Articulation points / bridges

### 3. Breadth-First Search (BFS)

BFS explores neighbors first , it's like expanding in waves. This guarantees shortest path in unweighted graphs.

#### A. BFS with Queue

```
void bfs(int start) {
    queue<int> q;
    q.push(start);
    visited[start] = true;
    while (!q.empty()) {
        int u = q.front(); q.pop();
        for (int v : adj[u]) {
            if (!visited[v]) {
                visited[v] = true;
                q.push(v);
            }
        }
    }
}
```

#### B. Track Distance

```
int dist[MAX];
void bfs_dist(int s) {
    fill(dist, dist + MAX, -1);
    dist[s] = 0;
    queue<int> q; q.push(s);
    while (!q.empty()) {
        int u = q.front(); q.pop();
        for (int v : adj[u]) {
            if (dist[v] == -1) {
                dist[v] = dist[u] + 1;
                q.push(v);
            }
        }
    }
}
```

Now  $\text{dist}[v]$  gives shortest distance from  $s$ .

### C. Complexity

Same as DFS:

Time	Space
$O(V + E)$	$O(V)$

Used in:

- Shortest paths (unweighted)- Level-order traversal- Bipartite check- Connected components

## 4. Iterative Deepening Search (IDS)

DFS is memory-light but might go too deep. BFS is optimal but can use a lot of memory. Iterative Deepening Search (IDS) combines both.

It performs DFS with increasing depth limits:

```
bool dls(int u, int target, int depth) {
    if (u == target) return true;
    if (depth == 0) return false;
    for (int v : adj[u])
        if (dls(v, target, depth - 1)) return true;
    return false;
}

bool ids(int start, int target, int max_depth) {
    for (int d = 0; d <= max_depth; d++)
        if (dls(start, target, d)) return true;
    return false;
}
```

Used in:

- AI search problems (state spaces)- Game trees (chess, puzzles)

## 5. Traversal Order Examples

For a graph:

```
1 - 2 - 3  
|   |  
4 - 5
```

DFS (starting at 1):  $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 4$  BFS (starting at 1):  $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 5$

## 6. Directed vs Undirected

- Undirected: mark both directions- Directed: follow edge direction only DFS on directed graphs is core to:
- SCC (Strongly Connected Components)- Topological Sorting- Reachability analysis

## 7. Traversal Trees

Each traversal implicitly builds a spanning tree:

- DFS Tree: based on recursion- BFS Tree: based on levels Use them to:
- Detect cross edges, back edges- Classify edges (important for algorithms like Tarjan's)

## 8. Comparison

Aspect	DFS	BFS
Strategy	Deep first	Level-wise
Space	$O(V)$ (stack)	$O(V)$ (queue)
Path Optimality	Not guaranteed	Yes (unweighted)
Applications	Cycle detection, backtracking	Shortest path, level order

## Tiny Code

DFS + BFS Combo:

```

void traverse(int n) {
    for (int i = 1; i <= n; i++) visited[i] = false;
    dfs(1);
    for (int i = 1; i <= n; i++) visited[i] = false;
    bfs(1);
}

```

## Why It Matters

DFS and BFS are the roots of graph theory in practice. Every algorithm you'll meet later , shortest paths, flows, SCCs , builds upon them.

They teach you how to navigate structure, how to systematically explore unknowns, and how search lies at the heart of computation.

## Try It Yourself

1. Build an adjacency list and run DFS/BFS from vertex 1.
2. Track discovery and finish times in DFS.
3. Use BFS to compute shortest paths in an unweighted graph.
4. Modify DFS to count connected components.
5. Implement IDS for a puzzle like the 8-puzzle.

Graph traversal is the art of exploration , once you master it, the rest of graph theory falls into place.

## 32. Strongly Connected Components (Tarjan, Kosaraju)

In directed graphs, edges have direction, so connectivity gets tricky. It's not enough for vertices to be reachable , you need mutual reachability.

That's the essence of a strongly connected component (SCC):

A set of vertices where every vertex can reach every other vertex.

Think of SCCs as islands of mutual connectivity , inside, you can go anywhere; outside, you can't. They're the building blocks for simplifying directed graphs into condensation DAGs (no cycles).

We'll explore two classic algorithms:

- Kosaraju's Algorithm , clean, intuitive, two-pass- Tarjan's Algorithm , one-pass, stack-based elegance

## 1. Definition

A Strongly Connected Component (SCC) in a directed graph (  $G = (V, E)$  ) is a maximal subset of vertices  $C \subseteq V$  such that for every pair (  $(u, v) \in C$  ):  $u \rightarrow v$  and  $v \rightarrow u$ .

In other words, every node in (C) is reachable from every other node in (C).

Example:

```
1 → 2 → 3 → 1    forms an SCC
4 → 5              separate SCCs
```

## 2. Applications

- Condensation DAG: compress SCCs into single nodes , no cycles remain.- Component-based reasoning: topological sorting on DAG of SCCs.- Program analysis: detecting cycles, dependencies.- Web graphs: find clusters of mutually linked pages.- Control-flow: loops and strongly connected subroutines.

## 3. Kosaraju's Algorithm

A simple two-pass algorithm using DFS and graph reversal.

Steps:

1. Run DFS and push nodes onto a stack in finish-time order.
2. Reverse the graph (edges flipped).
3. Pop nodes from stack; DFS on reversed graph; each DFS = one SCC.

## A. Implementation

```
vector<int> adj[MAX] , rev[MAX];
bool visited[MAX];
stack<int> st;
vector<vector<int>> sccs;

void dfs1(int u) {
    visited[u] = true;
    for (int v : adj[u])
        if (!visited[v])
            dfs1(v);
```

```

        st.push(u);
    }

void dfs2(int u, vector<int>& comp) {
    visited[u] = true;
    comp.push_back(u);
    for (int v : rev[u])
        if (!visited[v])
            dfs2(v, comp);
}

void kosaraju(int n) {
    // Pass 1: order by finish time
    for (int i = 1; i <= n; i++)
        if (!visited[i]) dfs1(i);

    // Reverse graph
    for (int u = 1; u <= n; u++)
        for (int v : adj[u])
            rev[v].push_back(u);

    // Pass 2: collect SCCs
    fill(visited, visited + n + 1, false);
    while (!st.empty()) {
        int u = st.top(); st.pop();
        if (!visited[u]) {
            vector<int> comp;
            dfs2(u, comp);
            sccs.push_back(comp);
        }
    }
}

```

Time Complexity:  $(O(V + E))$  , two DFS passes.

Space Complexity:  $(O(V + E))$

## B. Example

Graph:

```

1 → 2 → 3
↑   ↓   ↓
5 ← 4 ← 6

```

SCCs:

- {1,2,4,5}- {3,6}

#### 4. Tarjan's Algorithm

More elegant: one DFS pass, no reversal, stack-based. It uses discovery times and low-link values to detect SCC roots.

##### A. Idea

- `disc[u]`: discovery time of node u- `low[u]`: smallest discovery time reachable from u- A node is root of an SCC if `disc[u] == low[u]` Maintain a stack of active nodes (in current DFS path).

##### B. Implementation

```

vector<int> adj[MAX];
int disc[MAX], low[MAX], timer;
bool inStack[MAX];
stack<int> st;
vector<vector<int>> sccs;

void dfs_tarjan(int u) {
    disc[u] = low[u] = ++timer;
    st.push(u);
    inStack[u] = true;

    for (int v : adj[u]) {
        if (!disc[v]) {
            dfs_tarjan(v);
            low[u] = min(low[u], low[v]);
        } else if (inStack[v]) {
            low[u] = min(low[u], disc[v]);
        }
    }
}

```

```

    }

    if (disc[u] == low[u]) {
        vector<int> comp;
        while (true) {
            int v = st.top(); st.pop();
            inStack[v] = false;
            comp.push_back(v);
            if (v == u) break;
        }
        sccs.push_back(comp);
    }
}

void tarjan(int n) {
    for (int i = 1; i <= n; i++)
        if (!disc[i])
            dfs_tarjan(i);
}

```

Time Complexity:  $(O(V + E))$

Space Complexity:  $(O(V))$

## C. Walkthrough

Graph:

```

1 → 2 → 3
↑   ↓   ↓
5 ← 4 ← 6

```

DFS visits nodes in order; when it finds a node whose `disc == low`, it pops from the stack to form an SCC.

Result:

```

SCC1: 1 2 4 5
SCC2: 3 6

```

## 5. Comparison

Feature	Kosaraju	Tarjan
DFS Passes	2	1
Reversal Needed	Yes	No
Stack	Yes (finish order)	Yes (active path)
Implementation	Simple conceptually	Compact, efficient
Time	$O(V + E)$	$O(V + E)$

## 6. Condensation Graph

Once SCCs are found, you can build a DAG: Each SCC becomes a node, edges represent cross-SCC connections. Topological sorting now applies.

Used in:

- Dependency analysis- Strong component compression- DAG dynamic programming

### Tiny Code

Print SCCs (Tarjan):

```
tarjan(n);
for (auto &comp : sccs) {
    for (int x : comp) printf("%d ", x);
    printf("\n");
}
```

### Why It Matters

SCC algorithms turn chaotic directed graphs into structured DAGs. They're the key to reasoning about cycles, dependencies, and modularity.

Understanding them reveals a powerful truth:

“Every complex graph can be reduced to a simple hierarchy , once you find its strongly connected core.”

## Try It Yourself

1. Implement both Kosaraju and Tarjan , verify they match.
2. Build SCC DAG and run topological sort on it.
3. Detect cycles via SCC size > 1.
4. Use SCCs to solve 2-SAT (boolean satisfiability).
5. Visualize condensation of a graph with 6 nodes.

Once you can find SCCs, you can tame directionality , transforming messy networks into ordered systems.

## 33. Shortest Paths (Dijkstra, Bellman-Ford, A\*, Johnson)

Once you can traverse a graph, the next natural question is:

“What is the shortest path between two vertices?”

Shortest path algorithms are the heart of routing, navigation, planning, and optimization. They compute minimal cost paths , whether distance, time, or weight , and adapt to different edge conditions (non-negative, negative, heuristic).

This section covers the most essential algorithms:

- Dijkstra’s Algorithm , efficient for non-negative weights- Bellman-Ford Algorithm , handles negative edges- A\* , best-first with heuristics- Johnson’s Algorithm , all-pairs shortest paths in sparse graphs

### 1. The Shortest Path Problem

Given a weighted graph (  $G = (V, E)$  ) and a source (  $s$  ), find  $\text{dist}[v]$ , the minimum total weight to reach every vertex (  $v$  ).

Variants:

- Single-source shortest path (SSSP) , one source to all- Single-pair , one source to one target- All-pairs shortest path (APSP) , every pair- Dynamic shortest path , with updates

### 2. Dijkstra’s Algorithm

Best for non-negative weights. Idea: explore vertices in increasing distance order, like water spreading.

## A. Steps

1. Initialize all distances to infinity.
2. Set source distance = 0.
3. Use a priority queue to always pick the node with smallest tentative distance.
4. Relax all outgoing edges.

## B. Implementation (Adjacency List)

```
#include <bits/stdc++.h>
using namespace std;

const int INF = 1e9;
vector<pair<int,int>> adj[1000]; // (neighbor, weight)
int dist[1000];

void dijkstra(int n, int s) {
    fill(dist, dist + n + 1, INF);
    dist[s] = 0;
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<>> pq;
    pq.push({0, s});

    while (!pq.empty()) {
        auto [d, u] = pq.top(); pq.pop();
        if (d != dist[u]) continue;
        for (auto [v, w] : adj[u]) {
            if (dist[v] > dist[u] + w) {
                dist[v] = dist[u] + w;
                pq.push({dist[v], v});
            }
        }
    }
}
```

Complexity:

- Using priority queue (binary heap):  $O((V + E) \log V)$
- Space:  $O(V + E)$

### C. Example

Graph:

```
1 →(2) 2 →(3) 3  
↓(4)      ↑(1)  
4 →(2)
```

`dijkstra(1)` gives shortest distances:

```
dist[1] = 0  
dist[2] = 2  
dist[3] = 5  
dist[4] = 4
```

### D. Properties

- Works only if all edges  $w \geq 0$ - Can reconstruct path via `parent[v]`- Used in:
  - GPS and routing systems - Network optimization - Scheduling with positive costs

## 3. Bellman-Ford Algorithm

Handles negative edge weights, and detects negative cycles.

### A. Idea

Relax all edges  $(V-1)$  times. If on  $(V)$ -th iteration you can still relax  $\rightarrow$  negative cycle exists.

### B. Implementation

```
struct Edge { int u, v, w; };  
vector<Edge> edges;  
int dist[1000];  
  
bool bellman_ford(int n, int s) {  
    fill(dist, dist + n + 1, INF);  
    dist[s] = 0;
```

```

for (int i = 1; i <= n - 1; i++) {
    for (auto e : edges) {
        if (dist[e.u] + e.w < dist[e.v])
            dist[e.v] = dist[e.u] + e.w;
    }
}
// Check for negative cycle
for (auto e : edges)
    if (dist[e.u] + e.w < dist[e.v])
        return false; // negative cycle
return true;
}

```

Complexity:  $(O(VE))$  Works even when  $(w < 0)$ .

### C. Example

Graph:

$1 \rightarrow (2) \quad 2 \rightarrow (-5) \quad 3 \rightarrow (2) \quad 4$

Bellman-Ford finds path  $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$  with total cost  $(-1)$ .

If a cycle reduces total weight indefinitely, algorithm detects it.

### D. Use Cases

- Currency exchange arbitrage- Game graphs with penalties- Detecting impossible constraints

## 4. A\* Search Algorithm

Heuristic-guided shortest path, perfect for pathfinding (AI, maps, games).

It combines actual cost and estimated cost:

$$f(v) = g(v) + h(v)$$

where

- $(g(v))$ : known cost so far-  $(h(v))$ : heuristic estimate (must be admissible)

## A. Pseudocode

```
priority_queue<pair<int,int>, vector<pair<int,int>>, greater<>> pq;
g[start] = 0;
pq.push({h[start], start});

while (!pq.empty()) {
    auto [f, u] = pq.top(); pq.pop();
    if (u == goal) break;
    for (auto [v, w] : adj[u]) {
        int new_g = g[u] + w;
        if (new_g < g[v]) {
            g[v] = new_g;
            pq.push({g[v] + h[v], v});
        }
    }
}
}
```

Heuristic Example:

- Euclidean distance (for grids)- Manhattan distance (for 4-direction movement)

## B. Use Cases

- Game AI (pathfinding)- Robot motion planning- Map navigation Complexity: ( $O(E)$ ) in best case, depends on heuristic quality.

## 5. Johnson's Algorithm

Goal: All-Pairs Shortest Path in sparse graphs with negative edges (no negative cycles).

Idea:

1. Add new vertex  $q$  connected to all others with edge weight 0
2. Run Bellman-Ford from  $q$  to get potential  $h(v)$
3. Reweight edges:  $(w'(u, v) = w(u, v) + h(u) - h(v))$  (now all weights  $\geq 0$ )
4. Run Dijkstra from each vertex

Complexity:  $(OVE + V^2 \log V)$

## 6. Summary

Algorithm	Handles Negative Weights	Detects Negative Cycle	Heuristic	Complexity	Use Case
Dijkstra	No	No	No	$O((V+E) \log V)$	Non-negative weights
Bellman-Ford	Yes	Yes	No	$O(VE)$	Negative edges
A*	No (unless careful)	No	Yes	Depends on $V$	Pathfinding
Johnson	Yes (no neg. cycles)	Yes	No	$O(VE + V \log V)$	All-pairs, sparse

## Tiny Code

Dijkstra Example:

```
dijkstra(n, 1);
for (int i = 1; i <= n; i++)
    printf("dist[%d] = %d\n", i, dist[i]);
```

## Why It Matters

Shortest paths are the essence of optimization , not just in graphs, but in reasoning: finding minimal cost, minimal distance, minimal risk.

These algorithms teach:

“The path to a goal isn’t random , it’s guided by structure, weight, and knowledge.”

## Try It Yourself

1. Build a weighted graph and compare Dijkstra vs Bellman-Ford.
2. Introduce a negative edge and observe Bellman-Ford detecting it.
3. Implement A\* on a grid with obstacles.
4. Use Dijkstra to plan routes in a city map dataset.
5. Try Johnson’s algorithm for all-pairs shortest paths.

Master these, and you master direction + cost = intelligence in motion.

## 34. Shortest Path Variants (0-1 BFS, Bidirectional, Heuristic A\*)

Sometimes the classic shortest path algorithms aren't enough. You might have special edge weights (only 0 or 1), a need for faster searches, or extra structure you can exploit.

That's where shortest path variants come in , they're optimized adaptations of the big three (BFS, Dijkstra, A\*) for specific scenarios.

In this section, we'll explore:

- 0-1 BFS → when edge weights are only 0 or 1- Bidirectional Search → meet-in-the-middle for speed- Heuristic A\* → smarter exploration guided by estimates Each shows how structure in your problem can yield speed-ups.

### 1. 0-1 BFS

If all edge weights are either 0 or 1, you don't need a priority queue. A deque (double-ended queue) is enough for ( $O(V + E)$ ) time.

Why? Because edges with weight 0 should be processed immediately, while edges with weight 1 can wait one step longer.

#### A. Algorithm

Use a deque.

- When relaxing an edge with weight 0, push to front.- When relaxing an edge with weight 1, push to back.

```
const int INF = 1e9;
vector<pair<int,int>> adj[1000]; // (v, w)
int dist[1000];

void zero_one_bfs(int n, int s) {
    fill(dist, dist + n + 1, INF);
    deque<int> dq;
    dist[s] = 0;
    dq.push_front(s);

    while (!dq.empty()) {
        int u = dq.front(); dq.pop_front();
        for (auto [v, w] : adj[u]) {
            if (dist[v] > dist[u] + w) {
```

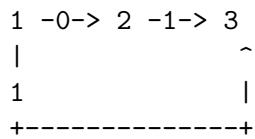
```

        dist[v] = dist[u] + w;
        if (w == 0) dq.push_front(v);
        else dq.push_back(v);
    }
}
}
}

```

## B. Example

Graph:



Shortest path from 1 to 3 = 1 (via edge 1-2-3). Deque ensures weight-0 edges don't get delayed.

## C. Complexity

Time	Space	Notes
$O(V + E)$	$O(V)$	Optimal for binary weights

Used in:

- Layered BFS- Grid problems with binary costs- BFS with teleportation (weight 0 edges)

## 2. Bidirectional Search

Sometimes you just need one path , from source to target , in an unweighted graph. Instead of expanding from one side, expand from both ends and stop when they meet.

This reduces search depth from  $(Ob^d)$  to  $(Ob^{d/2})$  (huge gain for large graphs).

## A. Idea

Run BFS from both source and target simultaneously. When their frontiers intersect, you've found the shortest path.

## B. Implementation

```
bool visited_from_s[MAX], visited_from_t[MAX];
queue<int> qs, qt;

int bidirectional_bfs(int s, int t) {
    qs.push(s); visited_from_s[s] = true;
    qt.push(t); visited_from_t[t] = true;

    while (!qs.empty() && !qt.empty()) {
        if (step(qs, visited_from_s, visited_from_t)) return 1;
        if (step(qt, visited_from_t, visited_from_s)) return 1;
    }
    return 0;
}

bool step(queue<int>& q, bool vis[], bool other[]) {
    int size = q.size();
    while (size--) {
        int u = q.front(); q.pop();
        if (other[u]) return true;
        for (int v : adj[u]) {
            if (!vis[v]) {
                vis[v] = true;
                q.push(v);
            }
        }
    }
    return false;
}
```

## C. Complexity

Time	Space	Notes
$Ob^{d/2}$	$Ob^{d/2}$	Doubly fast in practice

Used in:

- Maze solvers- Shortest paths in large sparse graphs- Social network “degrees of separation”

### 3. Heuristic A\* (Revisited)

A\* generalizes Dijkstra with goal-directed search using heuristics. We revisit it here to show how heuristics change exploration order.

#### A. Cost Function

$$f(v) = g(v) + h(v)$$

- ( $g(v)$ ): cost so far- ( $h(v)$ ): estimated cost to goal- ( $h(v)$ ) must be admissible (( $h(v)$  true cost))

#### B. Implementation

```

struct Node {
    int v; int f, g;
    bool operator>(const Node& o) const { return f > o.f; }
};

priority_queue<Node, vector<Node>, greater<Node>> pq;

void astar(int start, int goal) {
    g[start] = 0;
    h[start] = heuristic(start, goal);
    pq.push({start, g[start] + h[start], g[start]});

    while (!pq.empty()) {
        auto [u, f_u, g_u] = pq.top(); pq.pop();
        if (u == goal) break;
        for (auto [v, w] : adj[u]) {
            int new_g = g[u] + w;
            if (new_g < g[v]) {

```

```

        g[v] = new_g;
        int f_v = new_g + heuristic(v, goal);
        pq.push({v, f_v, new_g});
    }
}
}
}

```

### C. Example Heuristics

- Grid map: Manhattan distance ( $h(x, y) = |x - x_g| + |y - y_g|$ )
- Navigation: straight-line (Euclidean)- Game tree: evaluation function

### D. Performance

Heuristic	Effect
Perfect ( $h = \text{true cost}$ )	Optimal, visits minimal nodes
Admissible but weak	Still correct, more nodes
Overestimate	May fail (non-admissible)

## 4. Comparison

Algorithm	Weight Type	Strategy	Time	Space	Notes
0-1 BFS	0 or 1	Deque-based	$O(V+E)$	$O(V)$	No heap
Bidirectional BFS	Unweighted	Two-way search	$Ob^{d/2}$	$Ob^{d/2}$	Meets in middle
A*	Non- negative	Heuristic search	Depends	$O(V)$	Guided

## 5. Example Scenario

Problem	Variant
Grid with teleport (cost 0)	0-1 BFS
Huge social graph (find shortest chain)	Bidirectional BFS
Game AI pathfinding	A* with Manhattan heuristic

## Tiny Code

0-1 BFS Quick Demo:

```
add_edge(1, 2, 0);
add_edge(2, 3, 1);
zero_one_bfs(3, 1);
printf("%d\n", dist[3]); // shortest = 1
```

## Why It Matters

Special cases deserve special tools. These variants show that understanding structure (like edge weights or symmetry) can yield huge gains.

They embody a principle:

“Don’t just run faster , run smarter, guided by what you know.”

## Try It Yourself

1. Implement 0-1 BFS for a grid with cost 0 teleports.
2. Compare BFS vs Bidirectional BFS on a large maze.
3. Write A\* for an 8x8 chessboard knight’s move puzzle.
4. Tune heuristics , see how overestimating breaks A\*.
5. Combine A\* and 0-1 BFS for hybrid search.

With these in hand, you can bend shortest path search to the shape of your problem , efficient, elegant, and exact.

## 35. Minimum Spanning Trees (Kruskal, Prim, Boruvka)

When a graph connects multiple points with weighted edges, sometimes you don’t want the *shortest path*, but the *cheapest network* that connects everything.

That’s the Minimum Spanning Tree (MST) problem:

Given a connected, weighted, undirected graph, find a subset of edges that connects all vertices with minimum total weight and no cycles.

MSTs are everywhere , from building networks and designing circuits to clustering and approximation algorithms.

Three cornerstone algorithms solve it beautifully:

- Kruskal's , edge-based, union-find- Prim's , vertex-based, greedy expansion- Boruvka's , component merging in parallel

## **1. What Is a Spanning Tree?**

A spanning tree connects all vertices with exactly  $(V-1)$  edges. Among all spanning trees, the one with minimum total weight is the MST.

Properties:

- Contains no cycles- Connects all vertices- Edge count =  $(V - 1)$ - Unique if all weights distinct

## **2. MST Applications**

- Network design (roads, cables, pipelines)- Clustering (e.g., hierarchical clustering)- Image segmentation- Approximation (e.g., TSP  $\sim 2 \times$  MST)- Graph simplification

## **3. Kruskal's Algorithm**

Build the MST edge-by-edge, in order of increasing weight. Use Union-Find (Disjoint Set Union) to avoid cycles.

### **A. Steps**

1. Sort all edges by weight.
2. Initialize each vertex as its own component.
3. For each edge  $(u, v)$ :
  - If  $u$  and  $v$  are in different components  $\rightarrow$  include edge - Union their sets Stop when  $(V-1)$  edges chosen.

### **B. Implementation**

```

struct Edge { int u, v, w; };
vector<Edge> edges;
int parent[MAX], rank_[MAX];

int find(int x) {
    return parent[x] == x ? x : parent[x] = find(parent[x]);
}

bool unite(int a, int b) {
    a = find(a); b = find(b);
    if (a == b) return false;
    if (rank_[a] < rank_[b]) swap(a, b);
    parent[b] = a;
    if (rank_[a] == rank_[b]) rank_[a]++;
    return true;
}

int kruskal(int n) {
    iota(parent, parent + n + 1, 0);
    sort(edges.begin(), edges.end(), [](Edge a, Edge b){ return a.w < b.w; });
    int total = 0;
    for (auto &e : edges)
        if (unite(e.u, e.v))
            total += e.w;
    return total;
}

```

Complexity:

- Sorting edges: ( $O(E \log E)$ )- Union-Find operations: ( $O(V)$ ) (almost constant)- Total: ( $O(E \log E)$ )

### C. Example

Graph:

```

1 -4- 2
|     |
2     3
\--/

```

Edges sorted: (1-3,1), (1-2,4), (2-3,3)

Pick 1-3, 2-3 → MST weight = 1 + 3 = 4

#### 4. Prim's Algorithm

Grow MST from a starting vertex, adding the smallest outgoing edge each step.

Similar to Dijkstra , but pick edges, not distances.

##### A. Steps

1. Start with one vertex, mark as visited.
2. Use priority queue for candidate edges.
3. Pick smallest edge that connects to an unvisited vertex.
4. Add vertex to MST, repeat until all visited.

##### B. Implementation

```
vector<pair<int,int>> adj[MAX]; // (v, w)
bool used[MAX];
int prim(int n, int start) {
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<>> pq;
    pq.push({0, start});
    int total = 0;

    while (!pq.empty()) {
        auto [w, u] = pq.top(); pq.pop();
        if (used[u]) continue;
        used[u] = true;
        total += w;
        for (auto [v, w2] : adj[u])
            if (!used[v]) pq.push({w2, v});
    }
    return total;
}
```

Complexity:

- $O((V + E) \log V)$  with binary heap

Used when:

- Graph is dense
- Easier to grow tree than sort all edges

### C. Example

Graph:

```
1 -2- 2
|     |
4     1
\-3-/
```

Start at 1 → choose (1-2), (1-3) → MST weight =  $2 + 3 = 5$

### 5. Borůvka's Algorithm

Less famous, but elegant , merges cheapest outgoing edge per component in parallel.

Each component picks one cheapest outgoing edge, adds it, merges components. Repeat until one component left.

Complexity:  $(OE \log V)$

Used in parallel/distributed MST computations.

### 6. Comparison

Algorithm	Strategy	Time	Space	Best For
Kruskal	Edge-based, sort all edges	$O(E \log E)$	$O(E)$	Sparse graphs
Prim	Vertex-based, grow tree	$O(E \log V)$	$O(V+E)$	Dense graphs
Borůvka	Component merging	$O(E \log V)$	$O(E)$	Parallel MST

### 7. MST Properties

- Cut Property: For any cut, smallest crossing edge
- MST.- Cycle Property: For any cycle, largest edge not
- MST.- MST may not be unique if equal weights.

## 8. Building the Tree

Store MST edges:

```
vector<Edge> mst_edges;
if (unite(e.u, e.v)) mst_edges.push_back(e);
```

Then use MST for:

- Path queries- Clustering (remove largest edge)- Approximation TSP (preorder traversal)

### Tiny Code

Kruskal Example:

```
edges.push_back({1,2,4});
edges.push_back({1,3,1});
edges.push_back({2,3,3});
printf("MST = %d\n", kruskal(3)); // 4
```

### Why It Matters

MSTs model connection without redundancy. They're about efficiency , connecting everything at minimal cost, a principle that appears in infrastructure, data, and even ideas.

They teach:

“You can connect the whole with less , if you choose wisely.”

### Try It Yourself

1. Implement Kruskal's algorithm using union-find.
2. Run Prim's algorithm and compare output.
3. Build MST on random weighted graph , visualize tree.
4. Remove heaviest edge from MST to form two clusters.
5. Explore Boruvka for parallel execution.

MSTs are how you span complexity with minimal effort , a tree of balance, economy, and order.

## 36. Flows (Ford-Fulkerson, Edmonds-Karp, Dinic)

Some graphs don't just connect , they *carry* something. Imagine water flowing through pipes, traffic through roads, data through a network. Each edge has a capacity, and you want to know:

“How much can I send from source to sink before the system clogs?”

That's the Maximum Flow problem , a cornerstone of combinatorial optimization, powering algorithms for matching, cuts, scheduling, and more.

This section covers the big three:

- Ford-Fulkerson , the primal idea- Edmonds-Karp , BFS-based implementation- Dinic's Algorithm , layered speed

### 1. Problem Definition

Given a directed graph (  $G = (V, E)$  ), each edge (  $(u, v)$  ) has a capacity (  $c(u, v) \geq 0$  ).

We have:

- Source (  $s$  )- Sink (  $t$  ) We want the maximum flow from (  $s$  ) to (  $t$  ): a function (  $f(u, v)$  ) that satisfies:
  1. Capacity constraint: (  $0 \leq f(u, v) \leq c(u, v)$  )
  2. Flow conservation: For every vertex  $v \neq s, t$ : (  $\sum_{u \in N^-(v)} f(u, v) = \sum_{w \in N^+(v)} f(v, w)$  )

Total flow = (  $f(s, v)$  )

### 2. The Big Picture

Max Flow - Min Cut Theorem:

The value of the maximum flow equals the capacity of the minimum cut.

So finding a max flow is equivalent to finding the bottleneck.

### 3. Ford-Fulkerson Method

The idea:

- While there exists a path from (s) to (t) with available capacity, push flow along it.

Each step:

1. Find augmenting path
2. Send flow = min residual capacity along it
3. Update residual capacities

Repeat until no augmenting path.

#### A. Residual Graph

Residual capacity:

$$r(u, v) = c(u, v) - f(u, v)$$

If (  $f(u, v) > 0$  ), then add reverse edge ( (v, u) ) with capacity (  $f(u, v)$  ).

This allows undoing flow if needed.

#### B. Implementation (DFS-style)

```
const int INF = 1e9;
vector<pair<int,int>> adj[MAX];
int cap[MAX][MAX];

int dfs(int u, int t, int flow, vector<int>& vis) {
    if (u == t) return flow;
    vis[u] = 1;
    for (auto [v, _] : adj[u]) {
        if (!vis[v] && cap[u][v] > 0) {
            int pushed = dfs(v, t, min(flow, cap[u][v]), vis);
            if (pushed > 0) {
                cap[u][v] -= pushed;
                cap[v][u] += pushed;
                return pushed;
            }
        }
    }
}
```

```

        return 0;
    }

int ford_fulkerson(int s, int t, int n) {
    int flow = 0;
    while (true) {
        vector<int> vis(n + 1, 0);
        int pushed = dfs(s, t, INF, vis);
        if (pushed == 0) break;
        flow += pushed;
    }
    return flow;
}

```

Complexity:  $(OE \cdot \text{max flow})$ , depends on flow magnitude.

#### 4. Edmonds-Karp Algorithm

A refinement:

Always choose shortest augmenting path (by edges) using BFS.

Guarantees polynomial time.

##### A. Implementation (BFS + parent tracking)

```

int bfs(int s, int t, vector<int>& parent, int n) {
    fill(parent.begin(), parent.end(), -1);
    queue<pair<int,int>> q;
    q.push({s, INF});
    parent[s] = -2;
    while (!q.empty()) {
        auto [u, flow] = q.front(); q.pop();
        for (auto [v, _] : adj[u]) {
            if (parent[v] == -1 && cap[u][v] > 0) {
                int new_flow = min(flow, cap[u][v]);
                parent[v] = u;
                if (v == t) return new_flow;
                q.push({v, new_flow});
            }
        }
    }
}

```

```

        }
    }

    return 0;
}

int edmonds_karp(int s, int t, int n) {
    int flow = 0;
    vector<int> parent(n + 1);
    int new_flow;
    while ((new_flow = bfs(s, t, parent, n))) {
        flow += new_flow;
        int v = t;
        while (v != s) {
            int u = parent[v];
            cap[u][v] -= new_flow;
            cap[v][u] += new_flow;
            v = u;
        }
    }
    return flow;
}

```

Complexity: ( $OVE^2$ ) Always terminates (no dependence on flow values).

## 5. Dinic's Algorithm

A modern classic , uses BFS to build level graph, and DFS to send blocking flow.

It works layer-by-layer, avoiding useless exploration.

### A. Steps

1. Build level graph via BFS (assign levels to reachable nodes).
2. DFS sends flow along level-respecting paths.
3. Repeat until no path remains.

### B. Implementation

```

vector<int> level, ptr;

bool bfs_level(int s, int t, int n) {
    fill(level.begin(), level.end(), -1);
    queue<int> q;
    q.push(s);
    level[s] = 0;
    while (!q.empty()) {
        int u = q.front(); q.pop();
        for (auto [v, _] : adj[u])
            if (level[v] == -1 && cap[u][v] > 0) {
                level[v] = level[u] + 1;
                q.push(v);
            }
    }
    return level[t] != -1;
}

int dfs_flow(int u, int t, int pushed) {
    if (u == t || pushed == 0) return pushed;
    for (int &cid = ptr[u]; cid < (int)adj[u].size(); cid++) {
        int v = adj[u][cid].first;
        if (level[v] == level[u] + 1 && cap[u][v] > 0) {
            int tr = dfs_flow(v, t, min(pushed, cap[u][v]));
            if (tr > 0) {
                cap[u][v] -= tr;
                cap[v][u] += tr;
                return tr;
            }
        }
    }
    return 0;
}

int dinic(int s, int t, int n) {
    int flow = 0;
    level.resize(n + 1);
    ptr.resize(n + 1);
    while (bfs_level(s, t, n)) {
        fill(ptr.begin(), ptr.end(), 0);
        while (int pushed = dfs_flow(s, t, INF))
            flow += pushed;
    }
}

```

```

    }
    return flow;
}

```

Complexity:  $(OEV^2)$  worst case,  $(OE\sqrt{V})$  in practice.

## 6. Comparison

Algorithm	Strategy	Handles	Time	Notes
Ford-Fulkerson	DFS augmenting paths	Integral capacities	$OE \times \max_{flow}$	Simple, may loop on reals
Edmonds-Karp	BFS augmenting paths	All capacities	$O(VE^2)$	Always terminates
Dinic	Level graph + DFS	All capacities	$O(V^2E)$	Fast in practice

## 7. Applications

- Network routing- Bipartite matching- Task assignment (flows = people  $\rightarrow$  jobs)- Image segmentation (min-cut)- Circulation with demands- Data pipelines, max throughput systems

### Tiny Code

Ford-Fulkerson Example:

```

add_edge(1, 2, 3);
add_edge(1, 3, 2);
add_edge(2, 3, 5);
add_edge(2, 4, 2);
add_edge(3, 4, 3);
printf("Max flow = %d\n", ford_fulkerson(1, 4, 4)); // 5

```

## Why It Matters

Flow algorithms transform capacity constraints into solvable systems. They reveal the deep unity between optimization and structure: every maximum flow defines a minimum bottleneck cut.

They embody a timeless truth:

“To understand limits, follow the flow.”

## Try It Yourself

1. Implement Ford-Fulkerson using DFS.
2. Switch to Edmonds-Karp and observe performance gain.
3. Build Dinic’s level graph and visualize layers.
4. Model job assignment as bipartite flow.
5. Verify Max Flow = Min Cut on small examples.

Once you master flows, you’ll see them hidden in everything that moves , from data to decisions.

## 37. Cuts (Stoer-Wagner, Karger, Gomory-Hu)

Where flow problems ask “*How much can we send?*”, cut problems ask “*Where does it break?*”

A cut splits a graph into two disjoint sets. The minimum cut is the smallest set of edges whose removal disconnects the graph , the tightest “bottleneck” holding it together.

This chapter explores three major algorithms:

- Stoer-Wagner , deterministic min-cut for undirected graphs- Karger’s Randomized Algorithm , fast, probabilistic- Gomory-Hu Tree , compress all-pairs min-cuts into one tree Cuts reveal hidden structure , clusters, vulnerabilities, boundaries , and form the dual to flows via the Max-Flow Min-Cut Theorem.

### 1. The Min-Cut Problem

Given a weighted undirected graph (  $G = (V, E)$  ): Find the minimum total weight of edges whose removal disconnects the graph.

Equivalent to:

The smallest sum of edge weights crossing any partition (  $S, V \setminus S$  ).

For directed graphs, you use max-flow methods; For undirected graphs, specialized algorithms exist.

## 2. Applications

- Network reliability , weakest link detection- Clustering , partition graph by minimal interconnection- Circuit design , splitting components- Image segmentation , separating regions- Community detection , sparse connections between groups

## 3. Stoer-Wagner Algorithm (Deterministic)

A clean, deterministic method for global minimum cut in undirected graphs.

### A. Idea

1. Start with the full vertex set ( V ).
2. Repeatedly run Maximum Adjacency Search:
  - Start from a vertex - Grow a set by adding the most tightly connected vertex - The last added vertex defines a cut3. Contract the last two added vertices into one.
3. Keep track of smallest cut seen.

Repeat until one vertex remains.

### B. Implementation (Adjacency Matrix)

```
const int INF = 1e9;
int g[MAX][MAX], w[MAX];
bool added[MAX], exist[MAX];

int stoer_wagner(int n) {
    int best = INF;
    vector<int> v(n);
    iota(v.begin(), v.end(), 0);

    while (n > 1) {
        fill(w, w + n, 0);
        fill(added, added + n, false);
```

```

int prev = 0;
for (int i = 0; i < n; i++) {
    int sel = -1;
    for (int j = 0; j < n; j++)
        if (!added[j] && (sel == -1 || w[j] > w[sel])) sel = j;
    if (i == n - 1) {
        best = min(best, w[sel]);
        for (int j = 0; j < n; j++)
            g[prev][j] = g[j][prev] += g[sel][j];
        v.erase(v.begin() + sel);
        n--;
        break;
    }
    added[sel] = true;
    for (int j = 0; j < n; j++) w[j] += g[sel][j];
    prev = sel;
}
}
return best;
}

```

Complexity:  $(OV^3)$ , or  $(OVE + V^2 \log V)$  with heaps  
Input: weighted undirected graph  
Output: global min cut value

### C. Example

Graph:

```

1 -3- 2
|     |
4     2
\--/

```

Cuts:

- $\{1,2\}|\{3\} \rightarrow 7$ -  $\{1,3\}|\{2\} \rightarrow 5$  Min cut = 5

#### 4. Karger's Algorithm (Randomized)

A simple, elegant probabilistic method. Repeatedly contract random edges until two vertices remain; the remaining crossing edges form a cut.

Run multiple times → high probability of finding min cut.

#### A. Algorithm

1. While (  $|V| > 2$  ):

- Choose random edge  $((u, v))$  - Contract  $(u, v)$  into one node - Remove self-loops2. Return number of edges between remaining nodes

Repeat  $(On^2 \log n)$  times for high confidence.

#### B. Implementation Sketch

```
struct Edge { int u, v; };
vector<Edge> edges;
int parent[MAX];

int find(int x) { return parent[x] == x ? x : parent[x] = find(parent[x]); }
void unite(int a, int b) { parent[find(b)] = find(a); }

int karger(int n) {
    int m = edges.size();
    iota(parent, parent + n, 0);
    int vertices = n;
    while (vertices > 2) {
        int i = rand() % m;
        int u = find(edges[i].u), v = find(edges[i].v);
        if (u == v) continue;
        unite(u, v);
        vertices--;
    }
    int cuts = 0;
    for (auto e : edges)
        if (find(e.u) != find(e.v)) cuts++;
    return cuts;
}
```

Expected Time:  $(On^2)$  per run Probability of success:  $(2 / (n(n-1)))$  per run Run multiple trials and take minimum.

### C. Use Case

Great for large sparse graphs, or when approximate solutions are acceptable. Intuitive: the min cut survives random contractions if chosen carefully enough.

## 5. Gomory-Hu Tree

A compact way to store all-pairs min-cuts. It compresses  $(OV^2)$  flow computations into  $V-1$  cuts.

### A. Idea

- Build a tree where the min cut between any two vertices = the minimum edge weight on their path in the tree.

### B. Algorithm

1. Pick vertex ( $s$ ).
2. For each vertex  $t \neq s$ ,
  - Run max flow to find min cut between ( $s, t$ ). - Partition vertices accordingly.
  - Connect partitions to form a tree.

Result: Gomory-Hu tree ( $V-1$  edges).

Now any pair's min cut = smallest edge on path between them.

Complexity:  $(O(V))$  max flow runs.

### C. Uses

- Quickly answer all-pairs cut queries- Network reliability- Hierarchical clustering

## 6. Comparison

Algorithm	Type	Random-ized	Graph	Complexity	Output
Stoer-Wagner	Deterministic	No	Undirected	$O(V^3)$	Global min cut
Karger	Random-ized	Yes	Undirected	$O(n^2 \log n)$ (multi-run)	Probabilistic min cut
Gomory-Hu	Deterministic	No	Undirected	$O(V \times \text{MaxFlow})$	All-pairs min cuts

## 7. Relationship to Flows

By Max-Flow Min-Cut, min-cut capacity = max-flow value.

So you can find:

- s-t min cut = via max flow- global min cut = min over all (s, t) pairs Specialized algorithms just make it faster.

### Tiny Code

Stoer-Wagner Example:

```
printf("Global Min Cut = %d\n", stoer_wagner(n));
```

Karger Multi-Run:

```
int ans = INF;
for (int i = 0; i < 100; i++)
    ans = min(ans, karger(n));
printf("Approx Min Cut = %d\n", ans);
```

### Why It Matters

Cuts show you fragility , the weak seams of connection. While flows tell you *how much can pass*, cuts reveal *where it breaks first*.

They teach:

“To understand strength, study what happens when you pull things apart.”

## Try It Yourself

1. Implement Stoer-Wagner and test on small graphs.
2. Run Karger 100 times and track success rate.
3. Build a Gomory-Hu tree and answer random pair queries.
4. Verify Max-Flow = Min-Cut equivalence on examples.
5. Use cuts for community detection in social graphs.

Mastering cuts gives you both grip and insight , where systems hold, and where they give way.

## 38. Matchings (Hopcroft-Karp, Hungarian, Blossom)

In many problems, we need to pair up elements efficiently: students to schools, jobs to workers, tasks to machines.

These are matching problems , find sets of edges with no shared endpoints that maximize cardinality or weight.

Depending on graph type, different algorithms apply:

- Hopcroft-Karp , fast matching in bipartite graphs- Hungarian Algorithm , optimal weighted assignment- Edmonds' Blossom Algorithm , general graphs (non-bipartite) Matching is a fundamental combinatorial structure, appearing in scheduling, flow networks, and resource allocation.

### 1. Terminology

- Matching: set of edges with no shared vertices- Maximum Matching: matching with largest number of edges- Perfect Matching: covers all vertices (each vertex matched once)- Maximum Weight Matching: matching with largest total edge weight Graph Types:
- Bipartite: vertices split into two sets (L, R); edges only between sets- General: arbitrary connections (may contain odd cycles)

### 2. Applications

- Job assignment- Network flows- Resource allocation- Student-project pairing- Stable marriages (with preferences)- Computer vision (feature correspondence)

### 3. Hopcroft-Karp Algorithm (Bipartite Matching)

A highly efficient algorithm for maximum cardinality matching in bipartite graphs.

It uses layered BFS + DFS to find multiple augmenting paths simultaneously.

#### A. Idea

1. Initialize matching empty.
2. While augmenting paths exist:
  - BFS builds layer graph (shortest augmenting paths). - DFS finds all augmenting paths along those layers. Each phase increases matching size significantly.

#### B. Complexity

$$O(E\sqrt{V})$$

Much faster than augmenting one path at a time (like Ford-Fulkerson).

#### C. Implementation

Let `pairU[u]` = matched vertex in R, or 0 if unmatched `pairV[v]` = matched vertex in L, or 0 if unmatched

```
vector<int> adjL[MAX];
int pairU[MAX], pairV[MAX], dist[MAX];
int nL, nR;

bool bfs() {
    queue<int> q;
    for (int u = 1; u <= nL; u++) {
        if (!pairU[u]) dist[u] = 0, q.push(u);
        else dist[u] = INF;
    }
    int found = INF;
    while (!q.empty()) {
        int u = q.front(); q.pop();
        if (dist[u] < found) {
            for (int v : adjL[u]) {
                if (!pairV[v]) found = dist[u] + 1;
```

```

        else if (dist[pairV[v]] == INF) {
            dist[pairV[v]] = dist[u] + 1;
            q.push(pairV[v]);
        }
    }
}

return found != INF;
}

bool dfs(int u) {
    for (int v : adjL[u]) {
        if (!pairV[v] || (dist[pairV[v]] == dist[u] + 1 && dfs(pairV[v]))) {
            pairU[u] = v;
            pairV[v] = u;
            return true;
        }
    }
    dist[u] = INF;
    return false;
}

int hopcroft_karp() {
    int matching = 0;
    while (bfs()) {
        for (int u = 1; u <= nL; u++)
            if (!pairU[u] && dfs(u)) matching++;
    }
    return matching;
}

```

## D. Example

Graph:

$U = \{1, 2, 3\}$ ,  $V = \{a, b\}$   
 Edges: 1-a, 2-a, 3-b

Matching:  $\{1-a, 3-b\}$  (size 2)

## 4. Hungarian Algorithm (Weighted Bipartite Matching)

Solves assignment problem , given cost matrix  $c_{ij}$ , assign each (i) to one (j) minimizing total cost (or maximizing profit).

### A. Idea

Subtract minimums row- and column-wise → expose zeros → find minimal zero-cover → adjust matrix → repeat.

Equivalent to solving min-cost perfect matching on a bipartite graph.

### B. Complexity

$$O(V^3)$$

Works for dense graphs, moderate sizes.

### C. Implementation Sketch (Matrix Form)

```
int hungarian(const vector<vector<int>>& cost) {
    int n = cost.size();
    vector<int> u(n+1), v(n+1), p(n+1), way(n+1);
    for (int i = 1; i <= n; i++) {
        p[0] = i; int j0 = 0;
        vector<int> minv(n+1, INF);
        vector<char> used(n+1, false);
        do {
            used[j0] = true;
            int i0 = p[j0], delta = INF, j1;
            for (int j = 1; j <= n; j++) if (!used[j]) {
                int cur = cost[i0-1][j-1] - u[i0] - v[j];
                if (cur < minv[j]) minv[j] = cur, way[j] = j0;
                if (minv[j] < delta) delta = minv[j], j1 = j;
            }
            for (int j = 0; j <= n; j++)
                if (used[j]) u[p[j]] += delta, v[j] -= delta;
                else minv[j] -= delta;
            j0 = j1;
        } while (p[j0]);
    }
}
```

```

        do { int j1 = way[j0]; p[j0] = p[j1]; j0 = j1; } while (j0);
    }
    return -v[0]; // minimal cost
}

```

## D. Example

Cost matrix:

	a	b	c
1	3	2	1
2	2	3	2
3	3	2	3

Optimal assignment = 1-c, 2-a, 3-b Cost =  $1 + 2 + 2 = 5$

## 5. Edmonds' Blossom Algorithm (General Graphs)

For non-bipartite graphs, simple augmenting path logic breaks down (odd cycles). Blossom algorithm handles this via contraction of blossoms (odd cycles).

### A. Idea

- Find augmenting paths- When odd cycle encountered (blossom), shrink it into one vertex- Continue search- Expand blossoms at end

### B. Complexity

$$O(V^3)$$

Though complex to implement, it's the general-purpose solution for matchings.

### C. Use Cases

- Non-bipartite job/task assignments- General pairing problems- Network design

## 6. Comparison

Algorithm	Graph Type	Weighted	Complexity	Output
Hopcroft-Karp	Bipartite	No	$O(E\sqrt{V})$	Max cardinality
Hungarian	Bipartite	Yes	$O(V^3)$	Min/Max cost matching
Blossom	General	Yes	$O(V^3)$	Max cardinality or weight

## 7. Relation to Flows

Bipartite matching = max flow on network:

- Left → Source edges (capacity 1)- Right → Sink edges (capacity 1)- Between sets → edges (capacity 1) Matching size = flow value

### Tiny Code

Hopcroft-Karp Demo:

```
nL = 3; nR = 2;
adjL[1] = {1};
adjL[2] = {1};
adjL[3] = {2};
printf("Max Matching = %d\n", hopcroft_karp()); // 2
```

### Why It Matters

Matchings are the language of pairing and assignment. They express cooperation without overlap , a structure of balance.

They reveal a deep duality:

“Every match is a flow, every assignment an optimization.”

## Try It Yourself

1. Build a bipartite graph and run Hopcroft-Karp.
2. Solve an assignment problem with Hungarian algorithm.
3. Explore Blossom's contraction idea conceptually.
4. Compare max-flow vs matching approach.
5. Use matching to model scheduling (people tasks).

Matching teaches how to pair without conflict, a lesson both mathematical and universal.

## 39. Tree Algorithms (LCA, HLD, Centroid Decomposition)

Trees are the backbone of many algorithms , they are connected, acyclic, and wonderfully structured.

Because of their simplicity, they allow elegant divide-and-conquer, dynamic programming, and query techniques. This section covers three fundamental patterns:

- Lowest Common Ancestor (LCA) , answer ancestor queries fast- Heavy-Light Decomposition (HLD) , break trees into chains for segment trees / path queries- Centroid Decomposition , recursively split tree by balance for divide-and-conquer Each reveals a different way to reason about trees , by depth, by chains, or by balance.

### 1. Lowest Common Ancestor (LCA)

Given a tree, two nodes (u, v). The LCA is the lowest node (farthest from root) that is an ancestor of both.

Applications:

- Distance queries- Path decomposition- RMQ / binary lifting- Tree DP and rerooting

#### A. Naive Approach

Climb ancestors until they meet. But this is ( $O(n)$ ) per query , too slow for many queries.

## B. Binary Lifting

Precompute ancestors at powers of 2. Then jump up by powers to align depths.

Preprocessing:

1. DFS to record depth
2.  $\text{up}[v][k] = 2^k$ -th ancestor of v

Answering query:

1. Lift deeper node up to same depth
2. Lift both together while  $\text{up}[u][k] \neq \text{up}[v][k]$
3. Return parent

Code:

```
const int LOG = 20;
vector<int> adj[MAX];
int up[MAX][LOG], depth[MAX];

void dfs(int u, int p) {
    up[u][0] = p;
    for (int k = 1; k < LOG; k++)
        up[u][k] = up[up[u][k-1]][k-1];
    for (int v : adj[u]) if (v != p) {
        depth[v] = depth[u] + 1;
        dfs(v, u);
    }
}

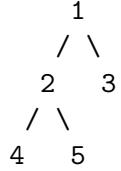
int lca(int u, int v) {
    if (depth[u] < depth[v]) swap(u, v);
    int diff = depth[u] - depth[v];
    for (int k = 0; k < LOG; k++)
        if (diff & (1 << k)) u = up[u][k];
    if (u == v) return u;
    for (int k = LOG-1; k >= 0; k--)
        if (up[u][k] != up[v][k])
            u = up[u][k], v = up[v][k];
    return up[u][0];
}
```

Complexity:

- Preprocess: ( $O(n \log n)$ ) - Query: ( $O(\log n)$ )

### C. Example

Tree:



- $\text{LCA}(4,5) = 2$ -  $\text{LCA}(4,3) = 1$

## 2. Heavy-Light Decomposition (HLD)

When you need to query paths (sum, max, min, etc.) on trees efficiently, you can use Heavy-Light Decomposition.

### A. Idea

Decompose the tree into chains:

- Heavy edge = edge to child with largest subtree- Light edges = others Result: Every path from root to leaf crosses at most ( $O(\log n)$ ) light edges.

So, a path query can be broken into ( $O(\log^2 n)$ ) segment tree queries.

### B. Steps

1. DFS to compute subtree sizes and identify heavy child
2. Decompose into chains
3. Assign IDs for segment tree
4. Use Segment Tree / BIT on linearized array

Key functions:

- `dfs_sz(u)` → compute subtree sizes- `decompose(u, head)` → assign chain heads Code (core):

```

int parent[MAX], depth[MAX], heavy[MAX], head[MAX], pos[MAX];
int cur_pos = 0;

int dfs_sz(int u) {
    int size = 1, max_sz = 0;
    for (int v : adj[u]) if (v != parent[u]) {
        parent[v] = u;
        depth[v] = depth[u] + 1;
        int sz = dfs_sz(v);
        if (sz > max_sz) max_sz = sz, heavy[u] = v;
        size += sz;
    }
    return size;
}

void decompose(int u, int h) {
    head[u] = h;
    pos[u] = cur_pos++;
    if (heavy[u] != -1) decompose(heavy[u], h);
    for (int v : adj[u])
        if (v != parent[u] && v != heavy[u])
            decompose(v, v);
}

```

Query path(u, v):

- While heads differ, move up chain by chain- Query segment tree in [pos[head[u]], pos[u]]- When in same chain, query segment [pos[v], pos[u]] Complexity:
- Build:  $(O(n))$ - Query/Update:  $(O\log^2 n)$

## C. Use Cases

- Path sums- Path maximums- Edge updates- Subtree queries

### 3. Centroid Decomposition

Centroid = node that splits tree into subtrees  $n/2$  each. By removing centroids recursively, we form a centroid tree.

Used for divide-and-conquer on trees.

## A. Steps

### 1. Find centroid

- DFS to compute subtree sizes - Choose node where largest subtree  $n/22$ . Decompose:
- Remove centroid - Recurse on subtrees Code (core):

```
int subtree[MAX];
bool removed[MAX];
vector<int> adj[MAX];

int dfs_size(int u, int p) {
    subtree[u] = 1;
    for (int v : adj[u])
        if (v != p && !removed[v])
            subtree[u] += dfs_size(v, u);
    return subtree[u];
}

int find_centroid(int u, int p, int n) {
    for (int v : adj[u])
        if (v != p && !removed[v])
            if (subtree[v] > n / 2)
                return find_centroid(v, u, n);
    return u;
}

void decompose(int u, int p) {
    int n = dfs_size(u, -1);
    int c = find_centroid(u, -1, n);
    removed[c] = true;
    // process centroid here
    for (int v : adj[c])
        if (!removed[v])
            decompose(v, c);
}
```

Complexity:  $(On \log n)$

## B. Applications

- Distance queries (decompose + store distance to centroid)- Tree problems solvable by divide-and-conquer- Dynamic queries (add/remove nodes)

## 4. Comparison

Algorithm	Purpose	Query	Preprocess	Complexity	Notes
LCA	Ancestor query	(Olog n)	(On log n)	Fast ancestor lookup	
HLD	Path queries	(Olog <sup>2</sup> n)	(O(n))	Segment tree-friendly	
Centroid Decomposition	Divide tree	-	(On log n)	Balanced splits	

## 5. Interconnections

- HLD often uses LCA internally.- Centroid decomposition may use distance to ancestor (via LCA).- All exploit tree structure to achieve sublinear queries.

### Tiny Code

LCA(4,5):

```
dfs(1,1);
printf("%d\n", lca(4,5)); // 2
```

HLD Path Sum: Build segment tree on pos[u] order, query along chains.

Centroid: decompose(1, -1);

### Why It Matters

Tree algorithms show how structure unlocks efficiency. They transform naive traversals into fast, layered, or recursive solutions.

To master data structures, you must learn to “climb” and “cut” trees intelligently.

“Every rooted path hides a logarithm.”

## Try It Yourself

1. Implement binary lifting LCA and test queries.
2. Add segment tree over HLD and run path sums.
3. Decompose tree by centroid and count nodes at distance k.
4. Combine LCA + HLD for path min/max.
5. Draw centroid tree of a simple graph.

Master these, and trees will stop being “just graphs” , they’ll become *tools*.

## 40. Advanced Graph Algorithms and Tricks

By now you’ve seen the big families , traversals, shortest paths, flows, matchings, cuts, and trees. But real-world graphs often bring extra constraints: dynamic updates, multiple sources, layered structures, or special properties (planar, DAG, sparse).

This section gathers powerful advanced graph techniques , tricks and patterns that appear across problems once you’ve mastered the basics.

We’ll explore:

- Topological Sorting & DAG DP- Strongly Connected Components (Condensation Graphs)- Articulation Points & Bridges (2-Edge/Vertex Connectivity)- Eulerian & Hamiltonian Paths- Graph Coloring & Bipartiteness Tests- Cycle Detection & Directed Acyclic Reasoning- Small-to-Large Merging, DSU on Tree, Mo’s Algorithm on Trees- Bitmask DP on Graphs- Dynamic Graphs (Incremental/Decremental BFS/DFS)- Special Graphs (Planar, Sparse, Dense) These aren’t just algorithms , they’re patterns that let you attack harder graph problems with insight.

### 1. Topological Sorting & DAG DP

In a DAG (Directed Acyclic Graph), edges always point forward. This makes it possible to order vertices linearly so all edges go from left to right , a topological order.

Use cases:

- Task scheduling- Dependency resolution- DP on DAG (longest/shortest path, counting paths) Algorithm (Kahn’s):

```

vector<int> topo_sort(int n) {
    vector<int> indeg(n+1), res;
    queue<int> q;
    for (int u = 1; u <= n; u++)
        for (int v : adj[u]) indeg[v]++;
    for (int u = 1; u <= n; u++)
        if (!indeg[u]) q.push(u);
    while (!q.empty()) {
        int u = q.front(); q.pop();
        res.push_back(u);
        for (int v : adj[u])
            if (--indeg[v] == 0) q.push(v);
    }
    return res;
}

```

DAG DP:

```

vector<int> dp(n+1, 0);
for (int u : topo_order)
    for (int v : adj[u])
        dp[v] = max(dp[v], dp[u] + weight(u,v));

```

Complexity:  $O(V + E)$

## 2. Strongly Connected Components (Condensation)

In directed graphs, vertices may form SCCs (mutually reachable components). Condensing SCCs yields a DAG, often easier to reason about.

Use:

- Component compression- Meta-graph reasoning- Cycle condensation Tarjan's Algorithm: DFS with low-link values, single pass.

Kosaraju's Algorithm: Two passes , DFS on graph and reversed graph.

Complexity:  $O(V + E)$

Once SCCs are built, you can run DP or topological sort on the condensed DAG.

### 3. Articulation Points & Bridges

Find critical vertices/edges whose removal disconnects the graph.

- Articulation point: vertex whose removal increases component count- Bridge: edge whose removal increases component count Algorithm: Tarjan's DFS Track discovery time `tin[u]` and lowest reachable ancestor `low[u]`.

```
void dfs(int u, int p) {
    tin[u] = low[u] = ++timer;
    for (int v : adj[u]) {
        if (v == p) continue;
        if (!tin[v]) {
            dfs(v, u);
            low[u] = min(low[u], low[v]);
            if (low[v] > tin[u]) bridge(u, v);
            if (low[v] >= tin[u] && p != -1) cut_vertex(u);
        } else low[u] = min(low[u], tin[v]);
    }
}
```

Applications:

- Network reliability- Biconnected components- 2-edge/vertex connectivity tests

### 4. Eulerian & Hamiltonian Paths

- Eulerian Path: visits every edge exactly once
  - Exists if graph is connected and 0 or 2 vertices have odd degree- Hamiltonian Path: visits every vertex exactly once (NP-hard) Euler Tour Construction: Hierholzer's algorithm ( $O(E)$ )

Applications:

- Route reconstruction (e.g., word chains)- Postman problems

### 5. Graph Coloring & Bipartiteness

Bipartite Check: DFS/ BFS alternating colors Fails if odd cycle found.

```

bool bipartite(int n) {
    vector<int> color(n+1, -1);
    for (int i = 1; i <= n; i++) if (color[i] == -1) {
        queue<int> q; q.push(i); color[i] = 0;
        while (!q.empty()) {
            int u = q.front(); q.pop();
            for (int v : adj[u]) {
                if (color[v] == -1)
                    color[v] = color[u] ^ 1, q.push(v);
                else if (color[v] == color[u])
                    return false;
            }
        }
    }
    return true;
}

```

Applications:

- 2-SAT reduction- Planar graph coloring- Conflict-free assignment

## 6. Cycle Detection

- DFS + recursion stack for directed graphs- Union-Find for undirected graphs Used to test acyclicity, detect back edges, or find cycles for rollback or consistency checks.

## 7. DSU on Tree (Small-to-Large Merging)

For queries like “count distinct colors in subtree,” merge results from smaller to larger subtrees to maintain  $O(n \log n)$ .

Pattern:

1. DFS through children
2. Keep large child’s data structure
3. Merge small child’s data in

Applications:

- Offline subtree queries- Heavy subproblem caching

## 8. Mo's Algorithm on Trees

Offline algorithm to answer path queries efficiently:

- Convert path queries to ranges via Euler Tour- Use Mo's ordering to process in  $O((N + Q)\sqrt{N})$  Useful when online updates aren't required.

## 9. Bitmask DP on Graphs

For small graphs ( $n = 20$ ): State = subset of vertices e.g., Traveling Salesman Problem (TSP)

```
dp[mask][u] = min cost to visit mask, end at u
```

Transition:

```
dp[mask | (1<<v)][v] = min(dp[mask][u] + cost[u][v])
```

Complexity:  $O(n^2 \cdot 2^n)$

## 10. Dynamic Graphs

Graphs that change:

- Incremental BFS: maintain distances as edges added- Decremental connectivity: union-find rollback or dynamic trees Used in online queries, evolving networks, or real-time systems.

## 11. Special Graph Classes

- Planar graphs:  $3V - 6E$ ; use face counting- Sparse graphs: adjacency lists best- Dense graphs: adjacency matrix / bitset Optimizations often hinge on density.

### Tiny Code

Topological Order:

```
auto order = topo_sort(n);
for (int u : order) printf("%d ", u);
```

Bridge Check: if ( $\text{low}[v] > \text{tin}[u]$ ) edge is a bridge.

Euler Path Check: Count odd-degree nodes == 0 or 2.

## Why It Matters

These advanced techniques complete your toolkit. They're not isolated , they combine to solve real-world puzzles: dependency graphs, robust networks, optimized paths, compressed states.

They teach a mindset:

“Graphs are not obstacles , they’re shapes of possibility.”

## Try It Yourself

1. Implement topological sort and DAG DP.
2. Find SCCs and build condensation graph.
3. Detect articulation points and bridges.
4. Check Euler path conditions on random graphs.
5. Try DSU on tree for subtree statistics.
6. Solve TSP via bitmask DP for n = 15.

Once you can mix and match these tools, you’re no longer just navigating graphs , you’re shaping them.

# Chapter 5. Dynamic Programming

## 41. DP Basics and State Transitions

Dynamic Programming (DP) is one of the most powerful ideas in algorithm design. It’s about breaking a big problem into smaller overlapping subproblems, solving each once, and reusing their answers.

When brute force explodes exponentially, DP brings it back under control. This section introduces the mindset, the mechanics, and the math behind DP.

### 1. The Core Idea

Many problems have two key properties:

- Overlapping subproblems: The same smaller computations repeat many times.
- Optimal substructure: The optimal solution to a problem can be built from optimal solutions to its subproblems.

DP solves each subproblem once, stores the result, and reuses it. This saves exponential time , often reducing (  $O(2^n)$  ) to (  $O(n^2)$  ) or (  $O(n)$  ).

## 2. The Recipe

When approaching a DP problem, follow this pattern:

1. Define the state. Decide what subproblems you'll solve. Example: `dp[i] = best answer for first i elements.`
2. Write the recurrence. Express each state in terms of smaller ones. Example: `dp[i] = dp[i-1] + cost(i)`
3. Set the base cases. Where does the recursion start? Example: `dp[0] = 0`
4. Decide the order. Bottom-up (iterative) or top-down (recursive with memoization).
5. Return the final answer. Often `dp[n]` or `max(dp[i])`.

## 3. Example: Fibonacci Numbers

Let's begin with a classic , the nth Fibonacci number (  $F(n) = F(n-1) + F(n-2)$  ).

Recursive (slow):

```
int fib(int n) {
    if (n <= 1) return n;
    return fib(n - 1) + fib(n - 2);
}
```

This recomputes the same values over and over , exponential time.

Top-Down DP (Memoization):

```
int dp[MAX];
int fib(int n) {
    if (n <= 1) return n;
    if (dp[n] != -1) return dp[n];
    return dp[n] = fib(n-1) + fib(n-2);
}
```

Bottom-Up DP (Tabulation):

```

int fib(int n) {
    int dp[n+1];
    dp[0] = 0; dp[1] = 1;
    for (int i = 2; i <= n; i++)
        dp[i] = dp[i-1] + dp[i-2];
    return dp[n];
}

```

Space Optimized:

```

int fib(int n) {
    int a = 0, b = 1, c;
    for (int i = 2; i <= n; i++) {
        c = a + b;
        a = b;
        b = c;
    }
    return b;
}

```

#### 4. States, Transitions, and Dependencies

A DP table is a map from states to answers. Each state depends on others via a transition function.

Think of it like a graph , each edge represents a recurrence relation.

Example:

- State:  $dp[i]$  = number of ways to reach step  $i$ - Transition:  $dp[i] = dp[i-1] + dp[i-2]$  (like stairs)- Base:  $dp[0] = 1$

#### 5. Common DP Patterns

1. 1D Linear DP
  - Problems like Fibonacci, climbing stairs, LIS.
2. 2D DP
  - Grids, sequences, or combinations (LCS, knapsack).
3. Bitmask DP

- Subsets, TSP, combinatorial optimization.
4. DP on Trees
    - Subtree computations (sum, diameter).
  5. Digit DP
    - Counting numbers with properties in a range.
  6. Segment DP
    - Matrix chain multiplication, interval merges.

## 6. Top-Down vs Bottom-Up

Approach	Method	Pros	Cons
Top-Down	Recursion + Memoization	Easy to write, intuitive	Stack overhead, needs memo
Bottom-Up	Iteration	Fast, space-optimizable	Harder to derive order

When dependencies are simple and acyclic, bottom-up shines. When they're complex, top-down is easier.

## 7. Example 2: Climbing Stairs

You can climb 1 or 2 steps at a time. How many distinct ways to reach step ( n )?

State: `dp[i] = ways to reach step i` Transition: `dp[i] = dp[i-1] + dp[i-2]` Base: `dp[0] = 1, dp[1] = 1`

Code:

```
int climb(int n) {
    int dp[n+1];
    dp[0] = dp[1] = 1;
    for (int i = 2; i <= n; i++)
        dp[i] = dp[i-1] + dp[i-2];
    return dp[n];
}
```

## 8. Debugging DP

To debug DP:

- Print intermediate states.- Visualize table (especially 2D).- Check base cases.- Trace one small example by hand.

## 9. Complexity

Most DP algorithms are linear or quadratic in number of states:

- Time = (#states)  $\times$  (work per transition)- Space = (#states) Example: Fibonacci: (  $O(n)$  ) time, (  $O(1)$  ) space Knapsack: (  $O(n \times W)$  ) LCS: (  $O(n \times m)$  )

## Tiny Code

Fibonacci (tabulated):

```
int dp[100];
dp[0] = 0; dp[1] = 1;
for (int i = 2; i <= n; i++)
    dp[i] = dp[i-1] + dp[i-2];
printf("%d", dp[n]);
```

## Why It Matters

DP is the art of remembering. It transforms recursion into iteration, chaos into order.

From optimization to counting, from paths to sequences , once you see substructure, DP becomes your hammer.

“Every repetition hides a recurrence.”

## Try It Yourself

1. Write top-down and bottom-up Fibonacci.
2. Count ways to climb stairs with steps {1,2,3}.
3. Compute number of paths in an  $n \times m$  grid.
4. Try to spot state, recurrence, base in each problem.
5. Draw dependency graphs to visualize transitions.

DP isn't a formula , it's a mindset: break problems into parts, remember the past, and build from it.

## 42. Classic Problems (Knapsack, Subset Sum, Coin Change)

Now that you know what dynamic programming *is*, let's dive into the classic trio , problems that every programmer meets early on:

- Knapsack (maximize value under weight constraint)- Subset Sum (can we form a given sum?)- Coin Change (how many ways or fewest coins to reach a total) These are the training grounds of DP: each shows how to define states, transitions, and base cases clearly.

### 1. 0/1 Knapsack Problem

Problem: You have  $n$  items, each with weight  $w[i]$  and value  $v[i]$ . A knapsack with capacity  $W$ . Pick items (each at most once) to maximize total value, without exceeding weight.

#### A. State

$dp[i][w] = \text{max value using first } i \text{ items with capacity } w$

#### B. Recurrence

For item  $i$ :

- If we don't take it:  $dp[i-1][w]$ - If we take it (if  $w[i] \leq w$ ):  $dp[i-1][w - w[i]] + v[i]$   
So,

$$dp[i][w] = \max(dp[i-1][w], dp[i-1][w - w[i]] + v[i])$$

#### C. Base Case

$dp[0][w] = 0$  for all  $w$  (no items = no value)

#### D. Implementation

```

int knapsack(int n, int W, int w[], int v[]) {
    int dp[n+1][W+1];
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= W; j++) {
            if (i == 0 || j == 0) dp[i][j] = 0;
            else if (w[i-1] <= j)
                dp[i][j] = max(dp[i-1][j], dp[i-1][j - w[i-1]] + v[i-1]);
            else
                dp[i][j] = dp[i-1][j];
        }
    }
    return dp[n][W];
}

```

Complexity: Time:  $(O(nW))$  Space:  $(O(nW))$  (can be optimized to 1D  $(O(W))$ )

## E. Space Optimization (1D DP)

```

int dp[W+1] = {0};
for (int i = 0; i < n; i++)
    for (int w = W; w >= weight[i]; w--)
        dp[w] = max(dp[w], dp[w - weight[i]] + value[i]);

```

## F. Example

Items:

```

w = [2, 3, 4, 5]
v = [3, 4, 5, 6]
W = 5

```

Best: take items 1 + 2 → value 7

## 2. Subset Sum

Problem: Given a set  $S$  of integers, can we pick some to sum to  $\text{target}$ ?

## A. State

$dp[i][sum]$  = true if we can form sum  $sum$  using first  $i$  elements.

## B. Recurrence

- Don't take:  $dp[i-1][sum]$ - Take (if  $a[i] \leq sum$ ):  $dp[i-1][sum - a[i]]$  So,

$$dp[i][sum] = dp[i-1][sum]; ||; dp[i-1][sum - a[i]]$$

## C. Base Case

$dp[0][0]$  = true (sum 0 possible with no elements)  $dp[0][sum]$  = false for  $sum > 0$

## D. Implementation

```
bool subset_sum(int a[], int n, int target) {
    bool dp[n+1][target+1];
    for (int i = 0; i <= n; i++) dp[i][0] = true;
    for (int j = 1; j <= target; j++) dp[0][j] = false;

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= target; j++) {
            if (a[i-1] > j) dp[i][j] = dp[i-1][j];
            else dp[i][j] = dp[i-1][j] || dp[i-1][j - a[i-1]];
        }
    }
    return dp[n][target];
}
```

Complexity: Time:  $(O(n \cdot target))$

## E. Example

$S = [3, 34, 4, 12, 5, 2]$ , target = 9 Yes  $\rightarrow 4 + 5$

## 3. Coin Change

Two variants:

### (a) Count Ways (Unbounded Coins)

“How many ways to make total T with coins c [] ?”

Order doesn't matter.

State:  $dp[i][t] = \text{ways using first } i \text{ coins for total } t$

Recurrence:

- Skip coin:  $dp[i-1][t]$ - Take coin (unlimited):  $dp[i][t - c[i]]$

$$dp[i][t] = dp[i-1][t] + dp[i][t - c[i]]$$

Base:  $dp[0][0] = 1$

1D Simplified:

```
int dp[T+1] = {0};
dp[0] = 1;
for (int coin : coins)
    for (int t = coin; t <= T; t++)
        dp[t] += dp[t - coin];
```

### (b) Min Coins (Fewest Coins to Reach Total)

State:  $dp[t] = \min \text{ coins to reach } t$

Recurrence:

$$dp[t] = \min_{c_i \leq t} (dp[t - c_i] + 1)$$

Base:  $dp[0] = 0$ , rest = INF

```
int dp[T+1];
fill(dp, dp+T+1, INF);
dp[0] = 0;
for (int t = 1; t <= T; t++)
    for (int c : coins)
        if (t >= c) dp[t] = min(dp[t], dp[t - c] + 1);
```

### Example

Coins = [1,2,5], Total = 5

- Ways: 4 (5; 2+2+1; 2+1+1+1; 1+1+1+1+1)- Min Coins: 1 (5)

## 4. Summary

Problem	Type	State	Transition	Complexity
0/1 Knapsack	Max value	dp[i][w]	max(take, skip)	O(nW)
Subset Sum	Feasibility	dp[i][sum]	OR of include/exclude	O(n * sum)
Coin Change (ways)	Counting	dp[t]	dp[t] + dp[t - coin]	O(nT)
Coin Change (min)	Optimization	dp[t]	min(dp[t - coin] + 1)	O(nT)

## Tiny Code

Min Coin Change (1D):

```
int dp[T+1];
fill(dp, dp+T+1, INF);
dp[0] = 0;
for (int c : coins)
    for (int t = c; t <= T; t++)
        dp[t] = min(dp[t], dp[t - c] + 1);
printf("%d\n", dp[T]);
```

## Why It Matters

These three are archetypes:

- Knapsack: optimize under constraint- Subset Sum: choose feasibility- Coin Change: count or minimize Once you master them, you can spot their patterns in harder problems , from resource allocation to pathfinding.

“Every constraint hides a choice; every choice hides a state.”

## Try It Yourself

1. Implement 0/1 Knapsack (2D and 1D).
2. Solve Subset Sum for target 30 with random list.
3. Count coin combinations for amount 10.
4. Compare “min coins” vs “ways to form.”
5. Write down state-transition diagram for each.

These three form your DP foundation , the grammar for building more complex algorithms.

## 43. Sequence Problems (LIS, LCS, Edit Distance)

Sequence problems form the *heart* of dynamic programming. They appear in strings, arrays, genomes, text comparison, and version control. Their power comes from comparing prefixes , building large answers from aligned smaller ones.

This section explores three cornerstones:

- LIS (Longest Increasing Subsequence)- LCS (Longest Common Subsequence)- Edit Distance (Levenshtein Distance) Each teaches a new way to think about subproblems, transitions, and structure.

### 1. Longest Increasing Subsequence (LIS)

Problem: Given an array, find the length of the longest subsequence that is *strictly increasing*.

A subsequence isn't necessarily contiguous , you can skip elements.

Example: [10, 9, 2, 5, 3, 7, 101, 18] → LIS is [2, 3, 7, 18] → length 4

#### A. State

$dp[i]$  = length of LIS ending at index  $i$

#### B. Recurrence

$$dp[i] = 1 + \max_{j < i \wedge a[j] < a[i]} dp[j]$$

If no smaller  $a[j]$ , then  $dp[i] = 1$ .

#### C. Base

$dp[i] = 1$  for all  $i$  (each element alone is an LIS)

#### D. Implementation

```

int lis(int a[], int n) {
    int dp[n], best = 0;
    for (int i = 0; i < n; i++) {
        dp[i] = 1;
        for (int j = 0; j < i; j++)
            if (a[j] < a[i])
                dp[i] = max(dp[i], dp[j] + 1);
        best = max(best, dp[i]);
    }
    return best;
}

```

Complexity:  $(On^2)$

## E. Binary Search Optimization

Use a tail array:

- `tail[len]` = min possible ending value of LIS of length `len` For each `x`:
- Replace `tail[idx]` via `lower_bound`

```

int lis_fast(vector<int>& a) {
    vector<int> tail;
    for (int x : a) {
        auto it = lower_bound(tail.begin(), tail.end(), x);
        if (it == tail.end()) tail.push_back(x);
        else *it = x;
    }
    return tail.size();
}

```

Complexity:  $(O(n \log n))$

## 2. Longest Common Subsequence (LCS)

Problem: Given two strings, find the longest subsequence present in both.

Example: `s1 = "ABCBDAB", s2 = "BDCABA"` LCS = “BCBA” → length 4

## A. State

$dp[i][j]$  = LCS length between  $s_1[0..i-1]$  and  $s_2[0..j-1]$

## B. Recurrence

$$dp[i][j] = \begin{cases} dp[i-1][j-1] + 1, & \text{if } s_1[i-1] = s_2[j-1], \\ \max(dp[i-1][j], dp[i][j-1]), & \text{otherwise.} \end{cases}$$

## C. Base

$dp[0][*] = dp[*][0] = 0$  (empty string)

## D. Implementation

```
int lcs(string a, string b) {
    int n = a.size(), m = b.size();
    int dp[n+1][m+1];
    for (int i = 0; i <= n; i++)
        for (int j = 0; j <= m; j++)
            if (i == 0 || j == 0) dp[i][j] = 0;
            else if (a[i-1] == b[j-1])
                dp[i][j] = dp[i-1][j-1] + 1;
            else
                dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
    return dp[n][m];
}
```

Complexity: ( $O(nm)$ )

## E. Reconstruct LCS

Trace back from  $dp[n][m]$ :

- If chars equal → take it and move diagonally- Else move toward larger neighbor

## F. Example

a = “AGGTAB”, b = “GXTXAYB” LCS = “GTAB” → 4

### 3. Edit Distance (Levenshtein Distance)

Problem: Minimum operations (insert, delete, replace) to convert string  $a \rightarrow b$ .

Example: kitten  $\rightarrow$  sitting = 3 (replace k $\rightarrow$ s, insert i, insert g)

#### A. State

$dp[i][j] = \min \text{ edits to convert } a[0..i-1] \rightarrow b[0..j-1]$

#### B. Recurrence

If  $a[i-1] == b[j-1]$ :

$$dp[i][j] = dp[i-1][j-1]$$

Else:

$$dp[i][j] = 1 + \min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1])$$

(Delete, Insert, Replace)

#### C. Base

- $dp[0][j] = j$  (insert all)-  $dp[i][0] = i$  (delete all)

#### D. Implementation

```
int edit_distance(string a, string b) {
    int n = a.size(), m = b.size();
    int dp[n+1][m+1];
    for (int i = 0; i <= n; i++)
        for (int j = 0; j <= m; j++) {
            if (i == 0) dp[i][j] = j;
            else if (j == 0) dp[i][j] = i;
            else if (a[i-1] == b[j-1])
                dp[i][j] = dp[i-1][j-1];
            else
                dp[i][j] = 1 + min({dp[i-1][j], dp[i][j-1], dp[i-1][j-1]});
        }
    return dp[n][m];
}
```

Complexity: ( $O(nm)$ )

## E. Example

a = “horse”, b = “ros”

- replace h→r, delete r, delete e → 3

## 4. Summary

Problem	Type	State	Transition	Complexity
LIS	Single seq	$dp[i]$	$1 + \max(dp[j])$	$O(n^2) / O(n \log n)$
LCS	Two seqs	$dp[i][j]$	if match +1 else max	$O(nm)$
Edit Distance	Two seqs	$dp[i][j]$	if match 0 else 1 + min	$O(nm)$

## 5. Common Insights

- LIS builds upward , from smaller sequences.- LCS aligns two sequences , compare prefixes.- Edit Distance quantifies *difference* , minimal edits. They’re templates for bioinformatics, text diffing, version control, and more.

## Tiny Code

LCS:

```
if (a[i-1] == b[j-1])
    dp[i][j] = dp[i-1][j-1] + 1;
else
    dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
```

## Why It Matters

Sequence DPs teach you how to compare progressions , how structure and similarity evolve over time.

They transform vague “compare these” tasks into crisp recurrence relations.

“To align is to understand.”

## Try It Yourself

1. Implement LIS ( $O(n^2)$  and  $O(n \log n)$ )
2. Find LCS of two given strings
3. Compute edit distance between “intention” and “execution”
4. Modify LCS to print one valid subsequence
5. Try to unify LCS and Edit Distance in a single table

Master these, and you can handle any DP on sequences , the DNA of algorithmic thinking.

## 44. Matrix and Chain Problems

Dynamic programming shines when a problem involves choices over intervals , which order, which split, which parenthesis. This chapter explores a class of problems built on chains and matrices, where order matters and substructure is defined by intervals.

We'll study:

- Matrix Chain Multiplication (MCM) - optimal parenthesization- Polygon Triangulation - divide shape into minimal-cost triangles- Optimal BST / Merge Patterns - weighted merging decisions These problems teach interval DP, where each state represents a segment  $([i, j])$ .

### 1. Matrix Chain Multiplication (MCM)

Problem: Given matrices  $A_1, A_2, \dots, A_n$ , find the parenthesization that minimizes total scalar multiplications.

Matrix  $A_i$  has dimensions  $p[i - 1] \times p[i]$ . We can multiply  $A_i \cdot A_{i+1}$  only if inner dimensions match.

Goal: Minimize operations:

$$\text{cost}(i, j) = \min_k (\text{cost}(i, k) + \text{cost}(k + 1, j) + p[i - 1] \cdot p[k] \cdot p[j])$$

#### A. State

$\text{dp}[i][j] = \min$  multiplications to compute  $A_i \dots A_j$

#### B. Base

$\text{dp}[i][i] = 0$  (single matrix needs no multiplication)

## C. Recurrence

$$dp[i][j] = \min_{i \leq k < j} dp[i][k] + dp[k+1][j] + p[i-1] \times p[k] \times p[j]$$

## D. Implementation

```
int matrix_chain(int p[], int n) {
    int dp[n][n];
    for (int i = 1; i < n; i++) dp[i][i] = 0;

    for (int len = 2; len < n; len++) {
        for (int i = 1; i + len - 1 < n; i++) {
            int j = i + len - 1;
            dp[i][j] = INT_MAX;
            for (int k = i; k < j; k++)
                dp[i][j] = min(dp[i][j],
                               dp[i][k] + dp[k+1][j] + p[i-1]*p[k]*p[j]);
        }
    }
    return dp[1][n-1];
}
```

Complexity:  $(On^3)$  time,  $(On^2)$  space

## E. Example

$p = [10, 20, 30, 40, 30]$  Optimal order:  $((A1A2)A3)A4 \rightarrow \text{cost } 30000$

## 2. Polygon Triangulation

Given a convex polygon with  $n$  vertices, connect non-intersecting diagonals to minimize total cost. Cost of a triangle = perimeter or product of side weights.

This is the same structure as MCM , divide polygon by diagonals.

## A. State

$dp[i][j] = \min$  triangulation cost for polygon vertices from  $i$  to  $j$ .

## B. Recurrence

$$dp[i][j] = \min_{i < k < j} (dp[i][k] + dp[k][j] + cost(i, j, k))$$

Base:  $dp[i][i+1] = 0$  (fewer than 3 points)

## C. Implementation

```
double polygon_triangulation(vector<Point> &p) {
    int n = p.size();
    double dp[n][n];
    for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) dp[i][j] = 0;
    for (int len = 2; len < n; len++) {
        for (int i = 0; i + len < n; i++) {
            int j = i + len;
            dp[i][j] = 1e18;
            for (int k = i+1; k < j; k++)
                dp[i][j] = min(dp[i][j],
                               dp[i][k] + dp[k][j] + dist(p[i], p[k]) + dist(p[k], p[j]) + dist(p[j], p[i]));
        }
    }
    return dp[0][n-1];
}
```

Complexity:  $(On^3)$

## 3. Optimal Binary Search Tree (OBST)

Given sorted keys  $k_1 < k_2 < \dots < k_n$  with search frequencies ( $f[i]$ ), construct a BST with minimal expected search cost.

The more frequently accessed nodes should be nearer the root.

### A. State

$dp[i][j] = \min$  cost to build BST from keys  $i \dots j$   $\text{sum}[i][j] = \text{sum of frequencies from } i \text{ to } j$  (precomputed)

## B. Recurrence

$$dp[i][j] = \min_{k=i}^j (dp[i][k-1] + dp[k+1][j] + sum[i][j])$$

Each root adds one to depth of its subtrees → extra cost = `sum[i][j]`

## C. Implementation

```

int optimal_bst(int freq[], int n) {
    int dp[n][n], sum[n][n];
    for (int i = 0; i < n; i++) {
        dp[i][i] = freq[i];
        sum[i][i] = freq[i];
        for (int j = i+1; j < n; j++)
            sum[i][j] = sum[i][j-1] + freq[j];
    }
    for (int len = 2; len <= n; len++) {
        for (int i = 0; i+len-1 < n; i++) {
            int j = i + len - 1;
            dp[i][j] = INT_MAX;
            for (int r = i; r <= j; r++) {
                int left = (r > i) ? dp[i][r-1] : 0;
                int right = (r < j) ? dp[r+1][j] : 0;
                dp[i][j] = min(dp[i][j], left + right + sum[i][j]);
            }
        }
    }
    return dp[0][n-1];
}

```

Complexity:  $(On^3)$

## 4. Merge Pattern Problems

Many problems , merging files, joining ropes, Huffman coding , involve repeatedly combining elements with minimal total cost.

All follow this template:

$$dp[i][j] = \min_k (dp[i][k] + dp[k+1][j] + \text{merge cost})$$

Same structure as MCM.

## 5. Key Pattern: Interval DP

State:  $dp[i][j] = \text{best answer for subarray } [i..j]$  Transition: Try all splits  $k$  between  $i$  and  $j$

Template:

```
for (len = 2; len <= n; len++) {
    for (i = 0; i + len - 1 < n; i++) {
        j = i + len - 1;
        dp[i][j] = INF;
        for (k = i; k < j; k++)
            dp[i][j] = min(dp[i][j], dp[i][k] + dp[k+1][j] + cost(i, j, k));
    }
}
```

## 6. Summary

Problem	State	Recurrence	Complexity
MCM	$dp[i][j]$	$\min(dp[i][k] + dp[k+1][j] + p[i-1]/p[k]/p[j])$	$O(n^3)$
Polygon Triangulation	$dp[i][j]$	$\min(dp[i][k] + dp[k][j] + \text{cost})$	$O(n^3)$
OBST	$dp[i][j]$	$\min(dp[i][k-1] + dp[k+1][j] + \text{sum}[i][j])$	$O(n^3)$
Merge Problems	$dp[i][j]$	$\min(dp[i][k] + dp[k+1][j] + \text{merge cost})$	$O(n^3)$

## Tiny Code

Matrix Chain (Compact):

```
for (len = 2; len < n; len++) {
    for (i = 1; i + len - 1 < n; i++) {
        j = i + len - 1; dp[i][j] = INF;
        for (k = i; k < j; k++)
            dp[i][j] = min(dp[i][j], dp[i][k] + dp[k+1][j] + p[i-1]*p[k]*p[j]);
    }
}
```

## Why It Matters

These problems are DP in 2D, reasoning over intervals and splits. They train your ability to “cut the problem” at every possible point.

“Between every start and end lies a choice of where to divide.”

## Try It Yourself

1. Implement MCM and print parenthesization.
2. Solve polygon triangulation with edge weights.
3. Build OBST for frequencies [34, 8, 50].
4. Visualize DP table diagonally.
5. Generalize to merging  $k$  segments at a time.

Master these, and you'll see interval DP patterns hiding in parsing, merging, and even AI planning.

## 45. Bitmask DP and Traveling Salesman

Some dynamic programming problems require you to track which items have been used, or which subset of elements is active at a given point. This is where Bitmask DP shines. It encodes subsets as binary masks, allowing you to represent state space efficiently.

This technique is a must-know for:

- Traveling Salesman Problem (TSP)- Subset covering / visiting problems- Permutations and combinations of sets- Game states and toggles

### 1. The Idea of Bitmask DP

A bitmask is an integer whose binary representation encodes a subset.

For ( $n$ ) elements:

- There are  $2^n$  subsets.- A subset is represented by a mask from 0 to  $(1 \ll n) - 1$ . Example for  $n = 4$ :

Subset	Mask (binary)	Mask (decimal)
	0000	0
{0}	0001	1
{1}	0010	2
{0,1,3}	1011	11

We can check membership:

- `mask & (1 << i)` → whether element  $i$  is in subset
- We can add elements:
- `mask | (1 << i)` → add element  $i$
- We can remove elements:
- `mask & ~(1 << i)` → remove element  $i$

## 2. Example: Traveling Salesman Problem (TSP)

Problem: Given  $n$  cities and cost matrix  $\text{cost}[i][j]$ , find the minimum cost Hamiltonian cycle visiting all cities exactly once and returning to start.

### A. State

$\text{dp}[\text{mask}][i] = \text{minimum cost to reach city } i \text{ having visited subset } \text{mask}$

- $\text{mask} \rightarrow$  set of visited cities-  $i \rightarrow$  current city

### B. Base Case

$\text{dp}[1 << 0][0] = 0$  (start at city 0, only 0 visited)

### C. Transition

For each subset  $\text{mask}$  and city  $i$  in  $\text{mask}$ , try moving from  $i$  to  $j$  not in  $\text{mask}$ :

$$\text{dp}[\text{mask} \cup (1 << j)][j] = \min (\text{dp}[\text{mask} \cup (1 << j)][j], \text{dp}[\text{mask}][i] + \text{cost}[i][j])$$

### D. Implementation

```
int tsp(int n, int cost[20][20]) {
    int N = 1 << n;
    const int INF = 1e9;
    int dp[N][n];
    for (int m = 0; m < N; m++)
        for (int i = 0; i < n; i++)
            dp[m][i] = INF;

    dp[1][0] = 0; // start at city 0

    for (int mask = 1; mask < N; mask++) {
        for (int i = 0; i < n; i++) {
            if (!(mask & (1 << i))) continue;
            for (int j = 0; j < n; j++) {
                if (mask & (1 << j)) continue;
                int next = mask | (1 << j);
```

```

        dp[next][j] = min(dp[next][j], dp[mask][i] + cost[i][j]);
    }
}

int ans = INF;
for (int i = 1; i < n; i++)
    ans = min(ans, dp[N-1][i] + cost[i][0]);
return ans;
}

```

Complexity:

- States: (  $O(n \cdot 2^n)$  )- Transitions: (  $O(n)$  )- Total: (  $O(n^2 \cdot 2^n)$  )

## E. Example

```

n = 4
cost = {
{0, 10, 15, 20},
{10, 0, 35, 25},
{15, 35, 0, 30},
{20, 25, 30, 0}
}

```

Optimal path:  $0 \rightarrow 1 \rightarrow 3 \rightarrow 2 \rightarrow 0$  Cost = 80

## 3. Other Common Bitmask DP Patterns

1. Subset Sum / Partition `dp[mask] = true if subset represented by mask satisfies property`
2. Counting Set Bits `__builtin_popcount(mask)` gives number of elements in subset.
3. Iterating Over Submasks

```

for (int sub = mask; sub; sub = (sub-1) & mask)
    // handle subset sub

```

4. Assigning Tasks (Assignment Problem)

- Each mask represents set of workers assigned.- State:  $dp[mask] = \min \text{ cost for assigned tasks.}$

```
for (mask) for (task)
if (!(mask & (1 << task)))
dp[mask | (1 << task)] = min(dp[mask | (1 << task)], 
    dp[mask] + cost[__builtin_popcount(mask)][task]);
```

#### 4. Memory Tricks

- If only previous masks needed, use rolling arrays:

```
dp[next][j] = ...
swap(dp, next_dp)
```

- Compress dimensions:  $(O2^n)$  memory for small  $n$

#### 5. Summary

Problem	State	Transition	Complexity
TSP	$dp[mask][i]$	$\min(dp[mask][i] + cost[i][j])$	$O(n^2 \cdot 2)$
Assignment	$dp[mask]$	add one new element	$O(n^2 \cdot 2)$
Subset Sum	$dp[mask]$	union of valid subsets	$O(2^n \cdot n)$

#### Tiny Code

Core Transition:

```
for (mask)
    for (i)
        if (mask & (1<<i))
            for (j)
                if (!(mask & (1<<j)))
                    dp[mask | (1<<j)][j] = min(dp[mask | (1<<j)][j], dp[mask][i] + cost[i][j]);
```

## Why It Matters

Bitmask DP is how you enumerate subsets efficiently. It bridges combinatorics and optimization, solving exponential problems with manageable constants.

“Every subset is a story, and bits are its alphabet.”

## Try It Yourself

1. Solve TSP with 4 cities (hand-trace the table).
2. Implement Assignment Problem using bitmask DP.
3. Count subsets with even sum.
4. Use bitmask DP to find maximum compatible set of tasks.
5. Explore how to optimize memory with bit tricks.

Bitmask DP unlocks the world of subset-based reasoning , the foundation of combinatorial optimization.

## 46. Digit DP and SOS DP

In some problems, you don't iterate over indices or subsets , you iterate over digits or masks to count or optimize over structured states. Two major flavors stand out:

- Digit DP - counting numbers with certain properties (e.g. digit sum, constraints)- SOS DP (Sum Over Subsets) - efficiently computing functions over all subsets These are essential techniques when brute force would require enumerating every number or subset, which quickly becomes impossible.

### 1. Digit DP (Counting with Constraints)

Digit DP is used to count or sum over all numbers  $N$  that satisfy a condition, such as:

- The sum of digits equals a target.- The number doesn't contain a forbidden digit.- The number has certain parity or divisibility. Instead of iterating over all numbers (up to  $10^1$  !), we iterate digit-by-digit.

## A. State Design

Typical DP state:

```
dp[pos] [sum] [tight] [leading_zero]
```

- **pos**: current digit index (from most significant to least)- **sum**: property tracker (e.g. sum of digits, remainder)- **tight**: whether we're still restricted by N's prefix- **leading\_zero**: whether we've started placing nonzero digits

## B. Transition

At each digit position, we choose a digit d:

```
limit = tight ? (digit at pos in N) : 9
for (d = 0; d <= limit; d++) {
    new_tight = tight && (d == limit)
    new_sum = sum + d
    // or new_mod = (mod * 10 + d) % M
}
```

Transition accumulates results across valid choices.

## C. Base Case

When **pos == len(N)** (end of digits):

- Return 1 if condition holds (e.g. **sum == target**), else 0

## D. Example: Count numbers N with digit sum = S

```
long long dp[20][200][2];

long long solve(string s, int pos, int sum, bool tight) {
    if (pos == s.size()) return sum == 0;
    if (sum < 0) return 0;
    if (dp[pos][sum][tight] != -1) return dp[pos][sum][tight];

    int limit = tight ? (s[pos] - '0') : 9;
    long long res = 0;
```

```

    for (int d = 0; d <= limit; d++)
        res += solve(s, pos+1, sum-d, tight && (d==limit));

    return dp[pos][sum][tight] = res;
}

```

Usage:

```

string N = "12345";
int S = 9;
memset(dp, -1, sizeof dp);
cout << solve(N, 0, S, 1);

```

Complexity:  $O(\text{number of digits} \times \text{sum} \times 2) \rightarrow \text{typically } O(20 \times 200 \times 2)$

## E. Example Variants

1. Count numbers divisible by 3 → track remainder: `new_rem = (rem*10 + d) % 3`
2. Count numbers without consecutive equal digits → add `last_digit` to state.
3. Count beautiful numbers (like palindromes, no repeated digits) → track bitmask of used digits.

## F. Summary

Problem	State	Transition	Complexity
Sum of digits = S	dp[pos][sum][tight]	sum-d	$O(\text{len} \cdot S)$
Divisible by k	dp[pos][rem][tight]	$(\text{rem} * 10 + d) \% k$	$O(\text{len} \cdot k)$
No repeated digits	dp[pos][mask][tight]	mask	$O(\text{len} \cdot 2^{\text{len}})$

## Tiny Code

```

for (int d = 0; d <= limit; d++)
    res += solve(pos+1, sum-d, tight && (d==limit));

```

## 2. SOS DP (Sum Over Subsets)

When dealing with functions on subsets, we sometimes need to compute:

$$f(S) = \sum_{T \subseteq S} g(T)$$

Naively  $O(3^n)$ . SOS DP reduces it to  $O(n \cdot 2^n)$ .

### A. Setup

Let  $f[\text{mask}] = g[\text{mask}]$  initially. For each bit  $i$ :

```
for (mask = 0; mask < (1<<n); mask++)
    if (mask & (1<<i))
        f[mask] += f[mask^(1<<i)];
```

After this,  $f[\text{mask}] = \text{sum of } g[\text{sub}] \text{ for all sub } \subseteq \text{mask}$ .

### B. Example

Given array  $a[\text{mask}]$ , compute  $\text{sum}[\text{mask}] = \text{sum}_{\{\text{sub} \subseteq \text{mask}\}} a[\text{sub}]$

```
int n = 3;
int N = 1 << n;
int f[N], a[N];
// initialize a[]
for (int mask = 0; mask < N; mask++) f[mask] = a[mask];
for (int i = 0; i < n; i++)
    for (int mask = 0; mask < N; mask++)
        if (mask & (1 << i))
            f[mask] += f[mask ^ (1 << i)];
```

### C. Why It Works

Each iteration adds contributions from subsets differing by one bit. By processing all bits, every subset's contribution propagates upward.

## D. Variants

- Sum over supersets: reverse direction.- Max instead of sum: replace  $+=$  with  $\max=$ .- XOR convolution: combine values under XOR subset relation.

## E. Applications

- Inclusion-exclusion acceleration- Precomputing subset statistics- DP over masks with subset transitions

## F. Complexity

Problem	Naive	SOS DP
Subset sum	$O(3^n)$	$O(n \cdot 2^n)$
Superset sum	$O(3^n)$	$O(n \cdot 2^n)$

## Why It Matters

Digit DP teaches counting under constraints , thinking digit by digit. SOS DP teaches subset propagation , spreading information efficiently.

Together, they show how to tame exponential state spaces with structure.

“When the search space explodes, symmetry and structure are your compass.”

## Try It Yourself

1. Count numbers  $10^6$  whose digit sum = 10.
2. Count numbers  $10^6$  without repeated digits.
3. Compute  $f[mask] = \sum_{\{sub\ mask\}} a[sub]$  for  $n=4$ .
4. Use SOS DP to find how many subsets of bits have even sum.
5. Modify Digit DP to handle leading zeros explicitly.

Master these, and you can handle structured exponential problems with elegance and speed.

## 47. DP Optimizations (Divide & Conquer, Convex Hull Trick, Knuth)

Dynamic Programming often starts with a simple recurrence, but naïve implementations can be too slow (e.g.,  $O(n^2)$  or worse). When the recurrence has special structure, such as monotonicity or convexity, we can exploit it to reduce time complexity drastically.

This chapter introduces three powerful optimization families:

1. Divide and Conquer DP
2. Convex Hull Trick (CHT)
3. Knuth Optimization

Each one is based on discovering order or geometry hidden inside transitions.

### 1. Divide and Conquer Optimization

If you have a recurrence like:

$$dp[i] = \min_{k < i} dp[k] + C(k, i)$$

and the optimal  $k$  for  $dp[i]$  optimal  $k$  for  $dp[i+1]$ , you can use divide & conquer to compute  $dp$  in  $(O(n \log n))$  or  $(O(n \log^2 n))$ .

This property is called monotonicity of argmin.

#### A. Conditions

Let  $(C(k, i))$  be the cost to transition from  $(k)$  to  $(i)$ . Divide and conquer optimization applies if:

$$opt(i) \leq opt(i+1)$$

and  $(C)$  satisfies quadrangle inequality (or similar convex structure).

#### B. Template

```

void compute(int l, int r, int optL, int optR) {
    if (l > r) return;
    int mid = (l + r) / 2;
    pair<long long,int> best = {INF, -1};
    for (int k = optL; k <= min(mid, optR); k++) {
        long long val = dp_prev[k] + cost(k, mid);
        if (val < best.first) best = {val, k};
    }
    dp[mid] = best.first;
    int opt = best.second;
    compute(l, mid-1, optL, opt);
    compute(mid+1, r, opt, optR);
}

```

You call it as:

```
compute(1, n, 0, n-1);
```

### C. Example: Divide Array into K Segments

Given array  $a[1..n]$ , divide into  $k$  parts to minimize

$$dp[i][k] = \min_{j < i} dp[j][k-1] + cost(j+1, i)$$

If cost satisfies quadrangle inequality, you can optimize each layer in (  $O(n \log n)$  ).

### D. Complexity

Naive: (  $O(n^2)$  ) → Optimized: (  $O(n \log n)$  )

## 2. Convex Hull Trick (CHT)

Applies when DP recurrence is linear in  $i$  and  $k$ :

$$dp[i] = \min_{k < i} (m_k \cdot x_i + b_k)$$

where:

- $m_k$  is slope (depends on  $k$ )- (  $b_k = dp[k] + c(k)$  )-  $x_i$  is known (monotonic) You can maintain lines  $y = m_k x + b_k$  in a convex hull, and query min efficiently.

## A. Conditions

- Slopes  $m_k$  are monotonic (either increasing or decreasing)- Query points  $x_i$  are sorted If both monotonic, we can use pointer walk in  $O(1)$  amortized per query. Otherwise, use Li Chao Tree ( $O(\log n)$ ).

## B. Implementation (Monotonic Slopes)

```

struct Line { long long m, b; };
deque<Line> hull;

bool bad(Line l1, Line l2, Line l3) {
    return (l3.b - l1.b)*(l1.m - l2.m) <= (l2.b - l1.b)*(l1.m - l3.m);
}

void add(long long m, long long b) {
    Line l = {m, b};
    while (hull.size() >= 2 && bad(hull[hull.size()-2], hull.back(), l))
        hull.pop_back();
    hull.push_back(l);
}

long long query(long long x) {
    while (hull.size() >= 2 &&
           hull[0].m*x + hull[0].b >= hull[1].m*x + hull[1].b)
        hull.pop_front();
    return hull.front().m*x + hull.front().b;
}

```

## C. Example: DP for Line-Based Recurrence

$$dp[i] = a_i^2 + \min_{j < i} (dp[j] + b_j \cdot a_i)$$

Here  $m_j = b_j$ ,  $x_i = a_i$ ,  $b_j = dp[j]$

## D. Complexity

- Naive: (  $O(n^2)$  )- CHT: (  $O(n)$  ) or (  $O(n \log n)$  )

### 3. Knuth Optimization

Used in interval DP problems (like Matrix Chain, Merging Stones).

If:

1.  $dp[i][j] = \min_{k=i}^{j-1}(dp[i][k] + dp[k+1][j] + w(i, j))$
2. The cost  $w(i, j)$  satisfies the quadrangle inequality:

$$w(a, c) + w(b, d) \leq w(a, d) + w(b, c)$$

3. And the monotonicity condition:

$$opt[i][j-1] \leq opt[i][j] \leq opt[i+1][j]$$

Then you can reduce the search space from (  $O(n)$  ) to (  $O(1)$  ) per cell, making total complexity (  $On^2$  ) instead of (  $On^3$  ).

### A. Implementation

```
for (int len = 2; len <= n; len++) {
    for (int i = 1; i + len - 1 <= n; i++) {
        int j = i + len - 1;
        dp[i][j] = INF;
        for (int k = opt[i][j-1]; k <= opt[i+1][j]; k++) {
            long long val = dp[i][k] + dp[k+1][j] + cost(i, j);
            if (val < dp[i][j]) {
                dp[i][j] = val;
                opt[i][j] = k;
            }
        }
    }
}
```

### B. Example

Optimal Binary Search Tree or Merging Stones (with additive cost). Typical improvement: (  $On^3 \rightarrow On^2$  )

## 4. Summary

Technique	Applies To	Key Property	Complexity
Divide & Conquer	1D transitions	Monotonic argmin	$O(n \log n)$
DP			
Convex Hull Trick	Linear transitions	Monotonic slopes	$O(n) / O(n \log n)$
Knuth Optimization	Interval DP	Quadrangle + Monotonicity	$O(n^2)$

## Tiny Code

Divide & Conquer Template

```
void compute(int l, int r, int optL, int optR);
```

CHT Query

```
while (size>=2 && f[1](x) < f[0](x)) pop_front();
```

## Why It Matters

These optimizations show that DP isn't just brute force with memory , it's mathematical reasoning on structure.

Once you spot monotonicity or linearity, you can shrink a quadratic solution into near-linear time.

“Optimization is the art of seeing simplicity hiding in structure.”

## Try It Yourself

1. Optimize Matrix Chain DP using Knuth.
2. Apply Divide & Conquer on  $dp[i] = \min_{\{k < i\}}(dp[k] + (i-k)^2)$ .
3. Solve Slope DP with CHT for convex cost functions.
4. Compare runtime vs naive DP on random data.
5. Derive conditions for opt monotonicity in your custom recurrence.

Master these techniques, and you'll turn your DPs from slow prototypes into lightning-fast solutions.

## 48. Tree DP and Rerooting

Dynamic Programming on trees is one of the most elegant and powerful patterns in algorithm design. Unlike linear arrays or grids, trees form hierarchical structures, where each node depends on its children or parent. Tree DP teaches you how to aggregate results up and down the tree, handling problems where subtrees interact.

In this section, we'll cover:

1. Basic Tree DP (rooted trees)
2. DP over children (bottom-up aggregation)
3. Rerooting technique (top-down propagation)
4. Common applications and examples

### 1. Basic Tree DP: The Idea

We define  $dp[u]$  to represent some property of the subtree rooted at  $u$ . Then we combine children's results to compute  $dp[u]$ .

This bottom-up approach is like postorder traversal.

Example structure:

```
function dfs(u, parent):  
    dp[u] = base_value  
    for v in adj[u]:  
        if v == parent: continue  
        dfs(v, u)  
        dp[u] = combine(dp[u], dp[v])
```

#### Example 1: Size of Subtree

Let  $dp[u]$  = number of nodes in subtree rooted at  $u$

```
void dfs(int u, int p) {  
    dp[u] = 1;  
    for (int v : adj[u]) {  
        if (v == p) continue;  
        dfs(v, u);  
        dp[u] += dp[v];  
    }  
}
```

Key idea: Combine children's sizes to get parent size. Complexity: (  $O(n)$  )

### **Example 2: Height of Tree**

Let  $dp[u] = \text{height of subtree rooted at } u$

```
void dfs(int u, int p) {
    dp[u] = 0;
    for (int v : adj[u]) {
        if (v == p) continue;
        dfs(v, u);
        dp[u] = max(dp[u], 1 + dp[v]);
    }
}
```

This gives you the height when rooted at any node.

### **2. DP Over Children (Bottom-Up Aggregation)**

Tree DP is all about merging children.

For example, if you want the number of ways to color or number of independent sets, you compute children's dp and merge results at parent.

### **Example 3: Counting Independent Sets**

In a tree, an independent set is a set of nodes with no two adjacent.

State:

- $dp[u][0] = \text{ways if } u \text{ is not selected}$  -  $dp[u][1] = \text{ways if } u \text{ is selected}$  Recurrence:

$$dp[u][0] = \prod_{v \in \text{children}(u)} (dp[v][0] + dp[v][1])$$

$$dp[u][1] = \prod_{v \in \text{children}(u)} dp[v][0]$$

Implementation:

```

void dfs(int u, int p) {
    dp[u][0] = dp[u][1] = 1;
    for (int v : adj[u]) {
        if (v == p) continue;
        dfs(v, u);
        dp[u][0] *= (dp[v][0] + dp[v][1]);
        dp[u][1] *= dp[v][0];
    }
}

```

Final answer =  $dp[\text{root}][0] + dp[\text{root}][1]$

#### Example 4: Maximum Path Sum in Tree

Let  $dp[u]$  = max path sum starting at  $u$  and going down To find best path anywhere, store a global max over child pairs.

```

int ans = 0;
int dfs(int u, int p) {
    int best1 = 0, best2 = 0;
    for (int v : adj[u]) {
        if (v == p) continue;
        int val = dfs(v, u) + weight(u, v);
        if (val > best1) swap(best1, val);
        if (val > best2) best2 = val;
    }
    ans = max(ans, best1 + best2);
    return best1;
}

```

This gives tree diameter or max path sum.

### 3. Rerooting Technique

Rerooting DP allows you to compute answers for every node as root, without recomputing from scratch (  $O(n^2)$  ). It's also known as DP on trees with re-rooting.

## Idea

1. First, compute  $dp\_down[u]$  = answer for subtree when rooted at  $u$ .
2. Then, propagate info from parent to child ( $dp\_up[u]$ ), so each node gets info from outside its subtree.
3. Combine  $dp\_down$  and  $dp\_up$  to get  $dp\_all[u]$ .

## Example 5: Sum of Distances from Each Node

Let's find  $ans[u] = \text{sum of distances from } u \text{ to all nodes}$ .

1. Root the tree at 0.
2. Compute subtree sizes and total distance from root.
3. Reroot to adjust distances using parent's info.

Step 1: Bottom-up:

```
void dfs1(int u, int p) {
    sz[u] = 1;
    for (int v : adj[u]) {
        if (v == p) continue;
        dfs1(v, u);
        sz[u] += sz[v];
        dp[u] += dp[v] + sz[v];
    }
}
```

Step 2: Top-down:

```
void dfs2(int u, int p) {
    for (int v : adj[u]) {
        if (v == p) continue;
        dp[v] = dp[u] - sz[v] + (n - sz[v]);
        dfs2(v, u);
    }
}
```

After  $dfs2$ ,  $dp[u] = \text{sum of distances from node } u$ .

Complexity: (  $O(n)$  )

#### 4. General Rerooting Template

```
// 1. Postorder: compute dp_down[u] from children
void dfs_down(u, p):
    dp_down[u] = base
    for v in adj[u]:
        if v != p:
            dfs_down(v, u)
            dp_down[u] = merge(dp_down[u], dp_down[v])

// 2. Preorder: use parent's dp_up to compute dp_all[u]
void dfs_up(u, p, dp_up_parent):
    ans[u] = merge(dp_down[u], dp_up_parent)
    prefix, suffix = prefix products of children
    for each child v:
        dp_up_v = merge(prefix[v-1], suffix[v+1], dp_up_parent)
        dfs_up(v, u, dp_up_v)
```

This template generalizes rerooting to many problems:

- Maximum distance from each node- Number of ways to select subtrees- Sum of subtree sizes seen from each root

#### 5. Summary

Pattern	Description	Complexity
Basic Tree DP	Combine child subresults	O(n)
DP Over Children	Each node depends on children	O(n)
Rerooting DP	Compute result for every root	O(n)
Multiple States	Track choices (e.g. include/exclude)	O(n · state)

#### Tiny Code

Subtree Size

```
void dfs(int u, int p) {
    dp[u] = 1;
    for (int v: adj[u]) if (v != p) {
```

```

    dfs(v, u);
    dp[u] += dp[v];
}
}

```

Reroot Sum Distances

```

dp[v] = dp[u] - sz[v] + (n - sz[v]);

```

### Why It Matters

Tree DP is how we think recursively over structure , each node's truth emerges from its children. Rerooting expands this idea globally, giving every node its own perspective.

“In the forest of states, each root sees a different world , yet all follow the same law.”

### Try It Yourself

1. Count number of nodes in each subtree.
2. Compute sum of depths from each node.
3. Find tree diameter using DP.
4. Count number of independent sets modulo 1e9+7.
5. Implement rerooting to find max distance from each node.

Tree DP turns recursive patterns into universal strategies for hierarchical data.

## 49. DP Reconstruction and Traceback

So far, we've focused on computing optimal values (min cost, max score, count of ways). But in most real problems, we don't just want the number , we want to know how we got it.

That's where reconstruction comes in: once you've filled your DP table, you can trace back the decisions that led to the optimal answer.

This chapter shows you how to:

1. Record transitions during DP computation
2. Reconstruct paths, subsets, or sequences
3. Handle multiple reconstructions (paths, sets, parent links)
4. Understand traceback in 1D, 2D, and graph-based DPs

## 1. The Core Idea

Each DP state comes from a choice. If you store *which choice was best*, you can walk backward from the final state to rebuild the solution.

Think of it as:

```
dp[i] = best over options  
choice[i] = argmin or argmax option
```

Then:

```
reconstruction_path = []  
i = n  
while i > 0:  
    reconstruction_path.push(choice[i])  
    i = choice[i].prev
```

You're not just solving , you're remembering the path.

## 2. Reconstruction in 1D DP

### Example: Coin Change (Minimum Coins)

Problem: Find minimum number of coins to make value  $n$ .

Recurrence:

$$dp[x] = 1 + \min_{c \in \text{coins}, c \leq x} dp[x - c]$$

To reconstruct which coins were used:

```
int dp[MAXN], prev_coin[MAXN];  
dp[0] = 0;  
for (int x = 1; x <= n; x++) {  
    dp[x] = INF;  
    for (int c : coins) {  
        if (x >= c && dp[x-c] + 1 < dp[x]) {  
            dp[x] = dp[x-c] + 1;  
            prev_coin[x] = c;  
        }  
    }  
}
```

Reconstruction:

```
vector<int> used;
int cur = n;
while (cur > 0) {
    used.push_back(prev_coin[cur]);
    cur -= prev_coin[cur];
}
```

Output: coins used in one optimal solution.

### Example: LIS Reconstruction

You know how to find LIS length. Now reconstruct the sequence.

```
int dp[n], prev[n];
int best_end = 0;
for (int i = 0; i < n; i++) {
    dp[i] = 1; prev[i] = -1;
    for (int j = 0; j < i; j++)
        if (a[j] < a[i] && dp[j] + 1 > dp[i]) {
            dp[i] = dp[j] + 1;
            prev[i] = j;
        }
    if (dp[i] > dp[best_end]) best_end = i;
}
```

Rebuild LIS:

```
vector<int> lis;
for (int i = best_end; i != -1; i = prev[i])
    lis.push_back(a[i]);
reverse(lis.begin(), lis.end());
```

## 3. Reconstruction in 2D DP

### Example: LCS (Longest Common Subsequence)

We have  $dp[i][j]$  filled using:

$$dp[i][j] = \begin{cases} dp[i-1][j-1] + 1, & \text{if } a[i-1] = b[j-1], \\ \max(dp[i-1][j], dp[i][j-1]), & \text{otherwise.} \end{cases}$$

To reconstruct LCS:

```
int i = n, j = m;
string lcs = "";
while (i > 0 && j > 0) {
    if (a[i-1] == b[j-1]) {
        lcs.push_back(a[i-1]);
        i--; j--;
    }
    else if (dp[i-1][j] > dp[i][j-1]) i--;
    else j--;
}
reverse(lcs.begin(), lcs.end());
```

Output: one valid LCS string.

### Example: Edit Distance

Operations: insert, delete, replace.

You can store the operation:

```
if (a[i-1] == b[j-1]) op[i][j] = "match";
else if (dp[i][j] == dp[i-1][j-1] + 1) op[i][j] = "replace";
else if (dp[i][j] == dp[i-1][j] + 1) op[i][j] = "delete";
else op[i][j] = "insert";
```

Then backtrack to list operations:

```
while (i > 0 || j > 0) {
    if (op[i][j] == "match") i--, j--;
    else if (op[i][j] == "replace") { print("Replace"); i--; j--; }
    else if (op[i][j] == "delete") { print("Delete"); i--; }
    else { print("Insert"); j--; }
}
```

## 4. Reconstruction in Path Problems

When DP tracks shortest paths, you can keep parent pointers.

### Example: Bellman-Ford Path Reconstruction

```
int dist[n], parent[n];
dist[src] = 0;
for (int k = 0; k < n-1; k++)
    for (auto [u,v,w] : edges)
        if (dist[u] + w < dist[v]) {
            dist[v] = dist[u] + w;
            parent[v] = u;
        }
vector<int> path;
for (int v = dest; v != src; v = parent[v])
    path.push_back(v);
path.push_back(src);
reverse(path.begin(), path.end());
```

You now have the actual shortest path.

## 5. Handling Multiple Solutions

Sometimes multiple optimal paths exist. You can:

- Store all predecessors instead of one- Backtrack recursively to enumerate all solutions- Tie-break deterministically (e.g., lexicographically smallest) Example:

```
if (new_val == dp[i]) parents[i].push_back(j);
```

Then recursively generate all possible paths.

## 6. Visualization

DP reconstruction often looks like following arrows in a grid or graph:

- LCS: diagonal ( ), up ( $\uparrow$ ), left ( $\leftarrow$ )- Shortest path: parent edges- LIS: predecessor chain  
You're walking through decisions, not just numbers.

## 7. Summary

Type	State	Reconstruction
1D DP	prev[i]	Trace chain
2D DP	op[i][j]	Follow choices
Graph DP	parent[v]	Follow edges
Counting DP	optional	Recover counts / paths

## Tiny Code

General pattern:

```
for (state)
    for (choice)
        if (better) {
            dp[state] = value;
            parent[state] = choice;
        }
```

Then:

```
while (state != base) {
    path.push_back(parent[state]);
    state = parent[state];
}
```

## Why It Matters

Solving DP gets you the score , reconstructing shows you the story. It's the difference between knowing the answer and understanding the reasoning.

“Numbers tell you the outcome; pointers tell you the path.”

## Try It Yourself

1. Reconstruct one LIS path.
2. Print all LCSs for small strings.
3. Show edit operations to transform “cat” → “cut”.
4. Track subset used in Knapsack to reach exact weight.

## 5. Recover optimal merge order in Matrix Chain DP.

Reconstruction turns DP from a static table into a narrative of decisions , a map back through the maze of optimal choices.

# 50. Meta-DP and Optimization Templates

We've now explored many flavors of dynamic programming , on sequences, grids, trees, graphs, subsets, and digits. This final chapter in the DP arc zooms out to the *meta-level*: how to see DP patterns, generalize them, and turn them into reusable templates.

If classical DP is about solving one problem, meta-DP is about recognizing *families* of problems that share structure. You'll learn how to build your own DP frameworks, use common templates, and reason from first principles.

## 1. What Is Meta-DP?

A *Meta-DP* is a high-level abstraction of a dynamic programming pattern. It encodes:

- State definition pattern- Transition pattern- Optimization structure- Dimensional dependencies By learning these patterns, you can design DPs faster, reuse logic across problems, and spot optimizations early.

Think of Meta-DP as:

“Instead of memorizing 100 DPs, master 10 DP blueprints.”

## 2. The Four Building Blocks

Every DP has the same core ingredients:

### 1. State: what subproblem you're solving

- Often `dp[i]`, `dp[i][j]`, or `dp[mask]` - Represents smallest unit of progress
- Transition: how to build larger subproblems from smaller ones
- E.g. `dp[i] = min(dp[j] + cost(j, i))`
- Base Case: known trivial answers
- E.g. `dp[0] = 0`
- Order: how to fill the states
- E.g. increasing `i`, decreasing `i`, or topological order Once you can describe a problem in these four, it *is* a DP.

### 3. Meta-Templates for Common Structures

Below are generalized templates to use and adapt.

#### A. Line DP (1D Sequential)

Shape: linear progression Examples:

- Fibonacci- Knapsack (1D capacity)- LIS (sequential dependency)

```
for (int i = 1; i <= n; i++) {  
    dp[i] = base;  
    for (int j : transitions(i))  
        dp[i] = min(dp[i], dp[j] + cost(j, i));  
}
```

Visualization: → → → Each state depends on previous positions.

#### B. Grid DP (2D Spatial)

Shape: grid or matrix Examples:

- Paths in a grid- Edit Distance- Counting paths with obstacles

```
for (i = 0; i < n; i++)  
    for (j = 0; j < m; j++)  
        dp[i][j] = combine(dp[i-1][j], dp[i][j-1]);
```

Visualization: Moves from top-left to bottom-right.

#### C. Interval DP

Shape: segments or subarrays Examples:

- Matrix Chain Multiplication- Optimal BST- Merging Stones

```

for (len = 2; len <= n; len++)
    for (i = 0; i + len - 1 < n; i++) {
        j = i + len - 1;
        dp[i][j] = INF;
        for (k = i; k < j; k++)
            dp[i][j] = min(dp[i][j], dp[i][k] + dp[k+1][j] + cost(i,j));
    }
}

```

Key Insight: overlapping intervals, split points.

## D. Subset DP

Shape: subsets of a set Examples:

- Traveling Salesman (TSP)- Assignment problem- SOS DP

```

for (mask = 0; mask < (1<<n); mask++)
    for (sub = mask; sub; sub = (sub-1)&mask)
        dp[mask] = combine(dp[mask], dp[sub]);

```

Key Insight: use bitmasks to represent subsets.

## E. Tree DP

Shape: hierarchical dependencies Examples:

- Subtree sizes- Independent sets- Rerooting

```

void dfs(u, p):
    dp[u] = base
    for (v in children)
        if (v != p)
            dfs(v, u)
            dp[u] = merge(dp[u], dp[v])

```

## F. Graph DP (Topological Order)

Shape: DAG structure Examples:

- Longest path in DAG- Counting paths- DAG shortest path

```
for (u in topo_order)
    for (v in adj[u])
        dp[v] = combine(dp[v], dp[u] + weight(u,v));
```

Key: process nodes in topological order.

## G. Digit DP

Shape: positional digits, constrained transitions Examples:

- Count numbers satisfying digit conditions- Divisibility / digit sum problems

```
dp[pos][sum][tight] = dp[next_pos][new_sum][new_tight];
```

## H. Knuth / Divide & Conquer / Convex Hull Trick

Shape: optimization over monotone or convex transitions Examples:

- Cost-based splits- Line-based transitions

```
dp[i] = min_k (dp[k] + cost(k, i))
```

Key: structure in opt[i] or slope.

Question	Clue
----------	------

#### 4. Recognizing DP Type

Ask these diagnostic questions:

Question	Clue
“Does each step depend on smaller subproblems?”	DP
“Do I split a segment?”	Interval DP
“Do I choose subsets?”	Subset / Bitmask DP
“Do I move along positions?”	Line DP
“Do I merge children?”	Tree DP
“Do I process in a DAG?”	Graph DP
“Do I track digits or constraints?”	Digit DP

#### 5. Optimization Layer

Once you have a working DP, ask:

- Can transitions be reduced (monotonicity)?- Can overlapping cost be cached (prefix sums)?- Can dimensions be compressed (rolling arrays)?- Can you reuse solutions for each segment (Divide & Conquer / Knuth)? This transforms your DP from conceptual to efficient.

#### 6. Meta-DP: Transformations

- Compress dimensions: if only  $dp[i-1]$  needed, use 1D array.- Invert loops: bottom-up top-down.- Change base: prefix-sums for range queries.- State lifting: add dimension for new property (like remainder, parity, bitmask). > “When stuck, add a dimension. When slow, remove one.”

#### 7. Common Template Snippets

Rolling 1D Knapsack:

```
for (c = C; c >= w[i]; c--)
    dp[c] = max(dp[c], dp[c-w[i]] + val[i]);
```

Top-Down Memoization:

```
int solve(state):
    if (visited[state]) return dp[state];
    dp[state] = combine(solve(next_states));
```

Iterative DP:

```
for (state in order)
    dp[state] = merge(prev_states);
```

## 8. Building Your Own DP Framework

You can design a generic DP(state, transition) class:

```
struct DP {
    vector<long long> dp;
    function<long long(int,int)> cost;
    DP(int n, auto cost): dp(n, INF), cost(cost) {}
    void solve() { for (int i=1; i<n; i++) for (int j=0; j<i; j++)
        dp[i] = min(dp[i], dp[j] + cost(j, i)); }
};
```

Reusable, readable, flexible.

## 9. Summary

DP Type	Core State	Shape	Typical Complexity
Line DP	dp[i]	Linear	$O(n^2) \rightarrow O(n)$
Grid DP	dp[i][j]	Matrix	$O(n \cdot m)$
Interval DP	dp[i][j]	Triangular	$O(n^3)$
Subset DP	dp[mask]	Exponential	$O(n \cdot 2^n)$
Tree DP	dp[u]	Tree	$O(n)$
Digit DP	dp[pos][sum]	Recursive	$O(\text{len} \cdot \text{sum})$
Graph DP	dp[v]	DAG	$O(V+E)$

## Tiny Code

```
for (state in order)
    dp[state] = combine(all_prev(state));
```

## Why It Matters

Meta-DP turns your thinking from case-by-case to pattern-by-pattern. You stop memorizing formulas and start *seeing shapes*: lines, grids, intervals, trees, masks.

Once you can name the shape, you can write the DP.

“DP is not about filling tables. It’s about recognizing structure.”

## Try It Yourself

1. Classify each classic DP problem into a type.
2. Write one template per pattern (Line, Grid, Tree, etc.).
3. Create a `dp_solve(state, transitions)` function to generalize logic.
4. For each pattern, practice a small example.
5. Build your own “Little Book of DP Patterns” with code snippets.

This is your bridge from concrete solutions to algorithmic fluency , the foundation for mastering the next 950 algorithms ahead.

# Chapter 6. Strings and Text Algorithms

## 51. Number Theory (GCD, Modular Arithmetic, CRT)

Number theory forms the mathematical backbone of many algorithms , from hashing and cryptography to modular combinatorics and primality testing. In algorithmic problem-solving, it’s all about working with integers, divisibility, and modular systems efficiently.

This section covers the essential toolkit:

- GCD and Extended Euclidean Algorithm- Modular Arithmetic (addition, subtraction, multiplication, inverse)- Modular Exponentiation- Chinese Remainder Theorem (CRT)

### 1. The Greatest Common Divisor (GCD)

The GCD of two integers  $a$  and  $b$ , denoted  $\gcd(a, b)$ , is the largest integer that divides both. It’s the cornerstone for fraction simplification, Diophantine equations, and modular inverses.

## A. Euclidean Algorithm

Based on:

$$\gcd(a, b) = \gcd(b, a \bmod b)$$

```
int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a % b);
}
```

Time complexity:  $O(\log \min(a, b))$

## B. Extended Euclidean Algorithm

Finds integers ( x, y ) such that:

$$ax + by = \gcd(a, b)$$

This is critical for finding modular inverses.

```
int ext_gcd(int a, int b, int &x, int &y) {
    if (b == 0) {
        x = 1; y = 0;
        return a;
    }
    int x1, y1;
    int g = ext_gcd(b, a % b, x1, y1);
    x = y1;
    y = x1 - (a / b) * y1;
    return g;
}
```

## C. Bezout's Identity

If  $g = \gcd(a, b)$ , then  $ax + by = g$  has integer solutions. If  $g = 1$ ,  $x$  is the modular inverse of  $a$  mod  $b$ .

## 2. Modular Arithmetic

A modular system “wraps around” after a certain value (  $m$  ).

We write:

$$a \equiv b \pmod{m} \quad \text{if } m \mid (a - b)$$

It behaves like ordinary arithmetic, with the rule:

- $(a + b) \bmod m = ((a \bmod m) + (b \bmod m)) \bmod m$
- $(a \cdot b) \bmod m = ((a \bmod m) \cdot (b \bmod m)) \bmod m$
- $(a - b) \bmod m = ((a \bmod m) - (b \bmod m) + m) \bmod m$

### A. Modular Exponentiation

Compute  $a^b \bmod m$  efficiently using binary exponentiation.

```
long long modpow(long long a, long long b, long long m) {
    long long res = 1;
    a %= m;
    while (b > 0) {
        if (b & 1) res = (res * a) % m;
        a = (a * a) % m;
        b >>= 1;
    }
    return res;
}
```

Complexity: (  $O(\log b)$  )

### B. Modular Inverse

Given (  $a$  ), find  $a^{-1}$  such that:

$$a \cdot a^{-1} \equiv 1 \pmod{m}$$

Case 1: If (  $m$  ) is prime, use Fermat’s Little Theorem:

$$a^{-1} \equiv a^{m-2} \pmod{m}$$

```

int modinv(int a, int m) {
    return modpow(a, m-2, m);
}

```

Case 2: If ( a ) and ( m ) are coprime, use Extended GCD:

```

int inv(int a, int m) {
    int x, y;
    int g = ext_gcd(a, m, x, y);
    if (g != 1) return -1; // no inverse
    return (x % m + m) % m;
}

```

## C. Modular Division

To divide  $a/b \bmod m$ :

$$a/b \equiv a \cdot b^{-1} \pmod{m}$$

So compute the inverse and multiply.

## 3. Chinese Remainder Theorem (CRT)

The CRT solves systems of congruences:

$$x \equiv a_1 \pmod{m_1}$$

$$x \equiv a_2 \pmod{m_2}$$

If moduli  $m_1, m_2, \dots, m_k$  are pairwise coprime, there exists a unique solution modulo  $M = m_1 m_2 \dots m_k$ .

## A. 2-Equation Example

Solve:

$$x \equiv a_1 \pmod{m_1}, \quad x \equiv a_2 \pmod{m_2}$$

Let:

- $M = m_1 m_2$ -  $M_1 = M/m_1$ -  $M_2 = M/m_2$  Find inverses  $inv_1 = M_1^{-1} \bmod m_1$ ,  $inv_2 = M_2^{-1} \bmod m_2$

Then:

$$x = (a_1 \cdot M_1 \cdot inv_1 + a_2 \cdot M_2 \cdot inv_2) \bmod M$$

## B. Implementation

```
long long crt(vector<int> a, vector<int> m) {
    long long M = 1;
    for (int mod : m) M *= mod;
    long long res = 0;
    for (int i = 0; i < a.size(); i++) {
        long long Mi = M / m[i];
        long long inv = modinv(Mi % m[i], m[i]);
        res = (res + 1LL * a[i] * Mi % M * inv % M) % M;
    }
    return (res % M + M) % M;
}
```

## C. Example

Solve:

x    2 (mod 3)  
x    3 (mod 5)  
x    2 (mod 7)

Solution: ( x = 23 ) (mod 105)

Check:

- ( 23 % 3 = 2 )- ( 23 % 5 = 3 )- ( 23 % 7 = 2 )

## 4. Tiny Code

GCD

```
int gcd(int a, int b) { return b ? gcd(b, a % b) : a; }
```

Modular Power

```
modpow(a, b, m)
```

Modular Inverse

```
modinv(a, m)
```

CRT

```
crt(a[], m[])
```

## 5. Summary

Concept	Formula	Purpose
GCD	$\gcd(a, b) = \gcd(b, a \bmod b)$	Simplify ratios
Extended GCD	$ax + by = \gcd(a, b)$	Find modular inverse
Modular Inverse	$a^{-1} \equiv a^{m-2} \pmod{m}$	Solve modular equations
Modular Exp	$a^b \bmod m$	Fast exponentiation
CRT	Combine congruences	Multi-mod system

## Why It Matters

Number theory lets algorithms speak the language of integers , turning huge computations into modular games. From hashing to RSA, from combinatorics to cryptography, it's everywhere.

“When numbers wrap around, math becomes modular , and algorithms become magical.”

## Try It Yourself

1. Compute  $\gcd(48, 180)$ .
2. Find inverse of 7 mod 13.
3. Solve  $x \equiv 1 \pmod{2}$ ,  $x \equiv 2 \pmod{3}$ ,  $x \equiv 3 \pmod{5}$ .
4. Implement modular division  $a/b \bmod m$ .
5. Use modpow to compute  $3^{200} \bmod 13$ .

These basics unlock higher algorithms in cryptography, combinatorics, and beyond.

## 52. Primality and Factorization (Miller-Rabin, Pollard Rho)

Primality and factorization are core to number theory, cryptography, and competitive programming. Many modern systems (RSA, ECC) rely on the hardness of factoring large numbers. Here, we learn how to test if a number is prime and break it into factors efficiently.

We'll cover:

- Trial Division
- Sieve of Eratosthenes (for precomputation)
- Probabilistic Primality Test (Miller-Rabin)
- Integer Factorization (Pollard Rho)

### 1. Trial Division

The simplest way to test primality is by dividing by all integers up to  $\sqrt{n}$ .

```
bool is_prime(long long n) {
    if (n < 2) return false;
    if (n % 2 == 0) return n == 2;
    for (long long i = 3; i * i <= n; i += 2)
        if (n % i == 0) return false;
    return true;
}
```

Time Complexity: (  $O(\sqrt{n})$  ) Good for  $n \leq 10^6$ , impractical for large (  $n$  ).

### 2. Sieve of Eratosthenes

For checking many numbers at once, use a sieve.

Idea: Mark all multiples of each prime starting from 2.

```
vector<bool> sieve(int n) {
    vector<bool> is_prime(n+1, true);
    is_prime[0] = is_prime[1] = false;
    for (int i = 2; i * i <= n; i++)
        if (is_prime[i])
            for (int j = i * i; j <= n; j += i)
                is_prime[j] = false;
    return is_prime;
}
```

Time Complexity: (  $O(n \log \log n)$  )

Useful for generating primes up to  $10^7$ .

### 3. Modular Multiplication

Before we do probabilistic tests or factorization, we need safe modular multiplication for large numbers.

```
long long modmul(long long a, long long b, long long m) {
    __int128 res = (__int128)a * b % m;
    return (long long)res;
}
```

Avoid overflow for  $n \approx 10^{18}$ .

### 4. Miller-Rabin Primality Test

A probabilistic test that can check if (  $n$  ) is prime or composite in (  $O(k \log^3 n)$  ).

Idea: For a prime (  $n$  ):

$$a^{n-1} \equiv 1 \pmod{n}$$

But for composites, most (  $a$  ) fail this.

We write  $n - 1 = 2^s \cdot d$ , (  $d$  ) odd.

For each base (  $a$  ):

- Compute  $x = a^d \pmod{n}$ - If (  $x = 1$  ) or (  $x = n - 1$  ), pass- Else, square (  $s-1$  ) times- If none equal (  $n - 1$  ), composite

```
bool miller_rabin(long long n) {
    if (n < 2) return false;
    for (long long p : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37})
        if (n % p == 0) return n == p;
    long long d = n - 1, s = 0;
    while ((d & 1) == 0) d >>= 1, s++;
    auto modpow = [&](long long a, long long b) {
        long long r = 1;
        while (b) {
            if (b & 1) r = modmul(r, a, n);
            a = modmul(a, a, n);
            b >>= 1;
        }
    };
    long long x = modpow(a, d);
    for (int i = 0; i < s; i++) {
        x = modmul(x, x, n);
        if (x == 1 || x == n - 1) return true;
    }
    return false;
}
```

```

    }
    return r;
};

for (long long a : {2, 325, 9375, 28178, 450775, 9780504, 1795265022}) {
    if (a % n == 0) continue;
    long long x = modpow(a, d);
    if (x == 1 || x == n - 1) continue;
    bool composite = true;
    for (int r = 1; r < s; r++) {
        x = modmul(x, x, n);
        if (x == n - 1) {
            composite = false;
            break;
        }
    }
    if (composite) return false;
}
return true;
}

```

Deterministic for:

- $n < 2^{64}$  with bases above. Complexity: (  $O(k \log^3 n)$  )

## 5. Pollard Rho Factorization

Efficient for finding nontrivial factors of large composites. Based on Floyd's cycle detection (Tortoise and Hare).

Idea: Define a pseudo-random function:

$$f(x) = (x^2 + c) \bmod n$$

Then find  $\gcd(|x - y|, n)$  where  $x, y$  move at different speeds.

```

long long pollard_rho(long long n) {
    if (n % 2 == 0) return 2;
    auto f = [&](long long x, long long c) {
        return (modmul(x, x, n) + c) % n;
    };
    while (true) {
        long long x = rand() % (n - 2) + 2;

```

```

long long y = x;
long long c = rand() % (n - 1) + 1;
long long d = 1;
while (d == 1) {
    x = f(x, c);
    y = f(f(y, c), c);
    d = gcd(abs(x - y), n);
}
if (d != n) return d;
}
}

```

Use:

1. Check if ( n ) is prime (Miller-Rabin)
2. If not, find a factor using Pollard Rho
3. Recurse on factors

Complexity:  $\sim (O(n^{1/4}))$  average

## 6. Example

Factorize ( n = 8051 ):

1. Miller-Rabin  $\rightarrow$  composite
2. Pollard Rho  $\rightarrow$  factor 83
3. ( 8051 / 83 = 97 )
4. Both primes ( 8051 = 83  $\times$  97 )

## 7. Tiny Code

```

void factor(long long n, vector<long long> &f) {
    if (n == 1) return;
    if (miller_rabin(n)) {
        f.push_back(n);
        return;
    }
    long long d = pollard_rho(n);
    factor(d, f);
    factor(n / d, f);
}

```

Call `factor(n, f)` to get prime factors.

## 8. Summary

Algorithm	Purpose	Complexity	Type
Trial Division	Small primes	( $O\sqrt{n}$ )	Deterministic
Sieve	Precompute primes	( $O n \log \log n$ )	Deterministic
Miller-Rabin	Primality test	( $O k \log^3 n$ )	Probabilistic
Pollard Rho	Factorization	( $O n^{1/4}$ )	Probabilistic

## Why It Matters

Modern security, number theory problems, and many algorithmic puzzles depend on knowing when a number is prime and factoring it quickly when it isn't. These tools are the entry point to RSA, modular combinatorics, and advanced cryptography.

## Try It Yourself

1. Check if 97 is prime using trial division and Miller-Rabin.
2. Factorize 5959 (should yield  $59 \times 101$ ).
3. Generate all primes  $\leq 100$  using a sieve.
4. Write a recursive factorizer using Pollard Rho + Miller-Rabin.
5. Measure performance difference between  $\sqrt{n}$  trial and Pollard Rho for  $n \approx 10^{12}$ .

These techniques make huge numbers approachable , one factor at a time.

## 53. Combinatorics (Permutations, Combinations, Subsets)

Combinatorics is the art of counting structures , how many ways can we arrange, select, or group things? In algorithms, it's everywhere: DP transitions, counting paths, bitmask enumeration, and probabilistic reasoning. Here we'll build a toolkit for computing factorials, nCr, nPr, and subset counts, both exactly and under a modulus.

## 1. Factorials and Precomputation

Most combinatorial formulas rely on factorials:

$$n! = 1 \times 2 \times 3 \times \cdots \times n$$

We can precompute them modulo ( m ) (often  $10^9 + 7$ ) for efficiency.

```
const int MOD = 1e9 + 7;
const int MAXN = 1e6;
long long fact[MAXN + 1], invfact[MAXN + 1];

long long modpow(long long a, long long b) {
    long long res = 1;
    while (b > 0) {
        if (b & 1) res = res * a % MOD;
        a = a * a % MOD;
        b >>= 1;
    }
    return res;
}

void init_factorials() {
    fact[0] = 1;
    for (int i = 1; i <= MAXN; i++)
        fact[i] = fact[i - 1] * i % MOD;
    invfact[MAXN] = modpow(fact[MAXN], MOD - 2);
    for (int i = MAXN - 1; i >= 0; i--)
        invfact[i] = invfact[i + 1] * (i + 1) % MOD;
}
```

Now you can compute ( nCr ) and ( nPr ) in ( O(1) ) time.

## 2. Combinations ( nCr )

The number of ways to choose r items from ( n ) items:

$$C(n, r) = \frac{n!}{r!(n-r)!}$$

```

long long nCr(int n, int r) {
    if (r < 0 || r > n) return 0;
    return fact[n] * invfact[r] % MOD * invfact[n - r] % MOD;
}

```

Properties:

- $(C(n, 0) = 1), (C(n, n) = 1)$
- $C(n, r) = C(n, n - r)$
- Pascal's Rule:  $C(n, r) = C(n - 1, r - 1) + C(n - 1, r)$

### **Example**

(  $C(5, 2) = 10$  ) There are 10 ways to pick 2 elements from a 5-element set.

### **3. Permutations ( nPr )**

Number of ways to arrange r elements chosen from ( n ):

$$P(n, r) = \frac{n!}{(n - r)!}$$

```

long long nPr(int n, int r) {
    if (r < 0 || r > n) return 0;
    return fact[n] * invfact[n - r] % MOD;
}

```

### **Example**

(  $P(5, 2) = 20$  ) Choosing 2 out of 5 elements and arranging them yields 20 orders.

### **4. Subsets and Power Set**

Each element has 2 choices: include or exclude. Hence, number of subsets:

$$2^n$$

```

long long subsets_count(int n) {
    return modpow(2, n);
}

```

Enumerating subsets using bitmasks:

```
for (int mask = 0; mask < (1 << n); mask++) {
    for (int i = 0; i < n; i++)
        if (mask & (1 << i))
            ; // include element i
}
```

Total:  $2^n$  subsets, (  $O(n2^n)$  ) time to enumerate.

## 5. Multisets and Repetition

Number of ways to choose ( r ) items from ( n ) with repetition:

$$C(n + r - 1, r)$$

For example, number of ways to give 5 candies to 3 kids (each can get 0): (  $C(3+5-1, 5) = C(7,5) = 21$  )

## 6. Modular Combinatorics

When working modulo ( p ): - Use modular inverse for division. -  $C(n, r) \bmod p = fact[n] \cdot invfact[r] \cdot invfact[n - r] \bmod p$

When  $n \geq p$ , use Lucas' Theorem:

$$C(n, r) \bmod p = C(n/p, r/p) \cdot C(n \bmod p, r \bmod p) \bmod p$$

## 7. Stirling and Bell Numbers (Advanced)

- Stirling Numbers of 2nd Kind: ways to partition ( n ) items into ( k ) non-empty subsets

$$S(n, k) = k \cdot S(n - 1, k) + S(n - 1, k - 1)$$

- Bell Numbers: total number of partitions

$$B(n) = \sum_{k=0}^n S(n, k)$$

Used in set partition and grouping problems.

## 8. Tiny Code

```

init_factorials();
printf("%lld\n", nCr(10, 3)); // 120
printf("%lld\n", nPr(10, 3)); // 720
printf("%lld\n", subsets_count(5)); // 32

```

## 9. Summary

Concept	Formula	Meaning	Example
Factorial	$n!$	All arrangements	$5! = 120$
Combination	$C(n, r) = \frac{n!}{r!(n-r)!}$	Choose	$C(5, 2) = 10$
Permutation	$P(n, r) = \frac{n!}{(n-r)!}$	Arrange	$P(5, 2) = 20$
Subsets	$2^n$	All combinations	$2^3 = 8$
Multisets	$C(n + r - 1, r)$	Repetition allowed	$C(4, 2) = 6$

## Why It Matters

Combinatorics underlies probability, DP counting, and modular problems. You can't master competitive programming or algorithm design without counting possibilities correctly. It teaches how structure emerges from choice , and how to count it efficiently.

## Try It Yourself

1. Compute  $C(1000, 500) \bmod (10^9 + 7)$ .
2. Count the number of 5-bit subsets with exactly 3 bits on, i.e.  $C(5, 3)$ .
3. Write a loop to print all subsets of  $\{a, b, c, d\}$ .
4. Use Lucas' theorem for  $C(10^6, 1000) \bmod 13$ .
5. Implement Stirling recursion and print  $S(5, 2)$ .

Every algorithmic counting trick , from Pascal's triangle to binomial theorem , starts right here.

## 54. Probability and Randomized Algorithms

Probability introduces controlled randomness into computation. Instead of deterministic steps, randomized algorithms use random choices to achieve speed, simplicity, or robustness. This section bridges probability theory and algorithm design , teaching how to model, analyze, and exploit randomness.

We'll cover:

- Probability Basics
- Expected Value
- Monte Carlo and Las Vegas Algorithms
- Randomized Data Structures and Algorithms

## 1. Probability Basics

Every event has a probability between 0 and 1.

If a sample space has  $n$  equally likely outcomes and  $k$  of them satisfy a condition, then

$$P(E) = \frac{k}{n}$$

Examples

- Rolling a fair die:  $P(\text{even}) = \frac{3}{6} = \frac{1}{2}$
- Drawing an ace from a deck:  $P(\text{ace}) = \frac{4}{52} = \frac{1}{13}$

Key Rules

- Complement:  $P(\bar{E}) = 1 - P(E)$
- Addition:  $P(A \cup B) = P(A) + P(B) - P(A \cap B)$
- Multiplication:  $P(A \cap B) = P(A) \cdot P(B | A)$

## 2. Expected Value

The expected value is the weighted average of outcomes.

$$E[X] = \sum_i P(x_i) \cdot x_i$$

Example: Expected value of a die:

$$E[X] = \frac{1+2+3+4+5+6}{6} = 3.5$$

Properties:

- Linearity:  $E[aX + bY] = aE[X] + bE[Y]$

- Useful for average-case analysis

In algorithms:

- Expected number of comparisons in QuickSort is  $O(n \log n)$
- Expected time for hash table lookup is  $O(1)$

### 3. Monte Carlo vs Las Vegas

Randomized algorithms are broadly two types:

Type	Output	Runtime	Example
Monte Carlo	May be wrong (probabilistically)	Fixed	Miller-Rabin Primality
Las Vegas	Always correct	Random runtime	Randomized QuickSort

Monte Carlo:

- Faster, approximate
- You can control error probability
- E.g. primality test returns “probably prime”

Las Vegas:

- Output guaranteed correct
- Runtime varies by luck
- E.g. QuickSort with random pivot

### 4. Randomization in Algorithms

Randomization helps break worst-case patterns.

#### A. Randomized QuickSort

Pick a random pivot instead of first element. Expected time becomes ( $O(n \log n)$ ) regardless of input order.

```
int partition(int a[], int l, int r) {
    int pivot = a[l + rand() % (r - l + 1)];
    // move pivot to end, then normal partition
}
```

This avoids adversarial inputs like sorted arrays.

## B. Randomized Hashing

Hash collisions can be exploited by adversaries. Using random coefficients in hash functions makes attacks infeasible.

```
long long hash(long long x, long long a, long long b, long long p) {
    return (a * x + b) % p;
}
```

Pick random ( a, b ) for robustness.

## C. Randomized Data Structures

1. Skip List: uses random levels for nodes Expected ( Olog n ) search/insert/delete
2. Treap: randomized heap priority + BST order Maintains balance in expectation

```
struct Node {
    int key, priority;
    Node *l, *r;
};
```

Randomized balancing gives good average performance without rotation logic.

## D. Random Sampling

Pick random elements efficiently:

- Reservoir Sampling: sample ( k ) items from a stream of unknown size- Shuffle: Fisher-Yates Algorithm

```
for (int i = n - 1; i > 0; i--) {
    int j = rand() % (i + 1);
    swap(a[i], a[j]);
}
```

## 5. Probabilistic Guarantees

Randomized algorithms often use Chernoff bounds and Markov's inequality to bound errors:

- Markov:  $P(X \geq kE[X]) \leq \frac{1}{k}$
- Chebyshev:  $P(|X - E[X]| \geq c\sigma) \leq \frac{1}{c^2}$
- Chernoff: Exponentially small tail bounds

These ensure “with high probability”  $(1 - \frac{1}{n^c})$  guarantees.

## 6. Tiny Code

Randomized QuickSort:

```
int partition(int arr[], int low, int high) {
    int pivotIdx = low + rand() % (high - low + 1);
    swap(arr[pivotIdx], arr[high]);
    int pivot = arr[high], i = low;
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) swap(arr[i++], arr[j]);
    }
    swap(arr[i], arr[high]);
    return i;
}

void quicksort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quicksort(arr, low, pi - 1);
        quicksort(arr, pi + 1, high);
    }
}
```

## 7. Summary

Concept	Key Idea	Use Case
Expected Value	Weighted average outcome	Analyze average case
Monte Carlo	Probabilistic correctness	Primality test
Las Vegas	Probabilistic runtime	QuickSort
Random Pivot	Break worst-case	Sorting
Skip List / Treap	Random balancing	Data Structures

Concept	Key Idea	Use Case
Reservoir Sampling	Stream selection	Large data

## Why It Matters

Randomization is not “luck”, it’s a design principle. It transforms rigid algorithms into adaptive, robust systems. In complexity theory, randomness helps achieve bounds impossible deterministically.

“A bit of randomness turns worst cases into best friends.”

## Try It Yourself

1. Simulate rolling two dice and compute expected sum.
2. Implement randomized QuickSort and measure average runtime.
3. Write a Monte Carlo primality checker.
4. Create a random hash function for integers.
5. Implement reservoir sampling for a large input stream.

These experiments show how uncertainty can become a powerful ally in algorithm design.

## 55. Sieve Methods and Modular Math

Sieve methods are essential tools in number theory for generating prime numbers, prime factors, and function values ( $, \phi$ ) efficiently. Combined with modular arithmetic, they form the backbone of algorithms in cryptography, combinatorics, and competitive programming.

This section introduces:

- Sieve of Eratosthenes- Optimized Linear Sieve- Sieve for Smallest Prime Factor (SPF)- Euler’s Totient Function ( $\phi$ )- Modular Applications

### 1. The Sieve of Eratosthenes

The classic algorithm to find all primes ( $n$ ).

Idea: Start from 2, mark all multiples as composite. Continue to  $\sqrt{n}$ .

```

vector<int> sieve(int n) {
    vector<int> primes;
    vector<bool> is_prime(n + 1, true);
    is_prime[0] = is_prime[1] = false;
    for (int i = 2; i * i <= n; i++)
        if (is_prime[i])
            for (int j = i * i; j <= n; j += i)
                is_prime[j] = false;
    for (int i = 2; i <= n; i++)
        if (is_prime[i]) primes.push_back(i);
    return primes;
}

```

Time Complexity: (  $O(n \log \log n)$  )

Space: (  $O(n)$  )

Example: Primes up to 20 → 2, 3, 5, 7, 11, 13, 17, 19

## 2. Linear Sieve ( $O(n)$ )

Unlike the basic sieve, each number is marked exactly once by its smallest prime factor (SPF).

Idea:

- For each prime (  $p$  ), mark  $p \times i$  only once.- Use  $\text{spf}[i]$  to store smallest prime factor.

```

const int N = 1e6;
int spf[N + 1];
vector<int> primes;

void linear_sieve() {
    for (int i = 2; i <= N; i++) {
        if (!spf[i]) {
            spf[i] = i;
            primes.push_back(i);
        }
        for (int p : primes) {
            if (p > spf[i] || 1LL * i * p > N) break;
            spf[i * p] = p;
        }
    }
}

```

Benefits:

- Get primes, SPF, and factorizations in ( O(n) ).- Ideal for problems needing many factorizations.

### 3. Smallest Prime Factor (SPF) Table

With `spf[]`, factorization becomes ( Olog n ).

```
vector<int> factorize(int x) {
    vector<int> f;
    while (x != 1) {
        f.push_back(spf[x]);
        x /= spf[x];
    }
    return f;
}
```

Example:  $\text{spf}[12] = 2 \rightarrow \text{factors} = [2, 2, 3]$

### 4. Euler's Totient Function ( (n) )

The number of integers ( n ) that are coprime with ( n ).

Formula:

$$\varphi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right)$$

Properties:

- $\varphi(p) = p - 1$  if  $p$  is prime
- Multiplicative: if  $\gcd(a, b) = 1$ , then  $\varphi(ab) = \varphi(a)\varphi(b)$

Implementation (Linear Sieve):

```
const int N = 1e6;
int phi[N + 1];
bool is_comp[N + 1];
vector<int> primes;

void phi_sieve() {
    phi[1] = 1;
```

```

for (int i = 2; i <= N; i++) {
    if (!is_comp[i]) {
        primes.push_back(i);
        phi[i] = i - 1;
    }
    for (int p : primes) {
        if (1LL * i * p > N) break;
        is_comp[i * p] = true;
        if (i % p == 0) {
            phi[i * p] = phi[i] * p;
            break;
        } else {
            phi[i * p] = phi[i] * (p - 1);
        }
    }
}

```

Example:

- $\varphi(6) = 6(1 - \frac{1}{2})(1 - \frac{1}{3}) = 2$
- Numbers coprime with 6: 1, 5

## 5. Modular Math Applications

Once we have primes and totients, we can do many modular computations.

### A. Fermat's Little Theorem

If ( p ) is prime,

$$a^{p-1} \equiv 1 \pmod{p}$$

Hence,

$$a^{-1} \equiv a^{p-2} \pmod{p}$$

Used in: modular inverses, combinatorics.

## B. Euler's Theorem

If  $\gcd(a, n) = 1$ , then

$$a^{\varphi(n)} \equiv 1 \pmod{n}$$

Generalizes Fermat's theorem to composite moduli.

## C. Modular Exponentiation with Totient Reduction

For very large powers:

$$a^b \pmod{n} = a^{b \pmod{\varphi(n)}} \pmod{n}$$

(when  $a$  and  $n$  are coprime)

## 6. Tiny Code

Primes up to n:

```
auto primes = sieve(100);
```

Totients up to n:

```
phi_sieve();
cout << phi[10]; // 4
```

Factorization:

```
auto f = factorize(60); // [2, 2, 3, 5]
```

Concept	Description	Time	Use
---------	-------------	------	-----

## 7. Summary

Concept	Description	Time	Use
Eratosthenes	Mark multiples	( $O(n \log \log n)$ )	Simple prime gen
Linear Sieve	Mark once	( $O(n)$ )	Prime + SPF
SPF Table (n)	Smallest prime factor Coprime count	( $O(1)$ ) query ( $O(n)$ )	Fast factorization Modular exponent
Fermat / Euler	Inverses, reduction	( $O(\log n)$ )	Modular arithmetic

## Why It Matters

Sieve methods are the fastest way to preprocess arithmetic information. They unlock efficient solutions to problems involving primes, divisors, modular equations, and cryptography.

“Before you can reason about numbers, you must first sieve them clean.”

## Try It Yourself

1. Generate all primes  $\leq 10^6$  using a linear sieve.
2. Factorize 840 using the SPF array.
3. Compute  $\varphi(n)$  for  $n = 1..20$ .
4. Verify  $a^{\varphi(n)} \equiv 1 \pmod{n}$  for  $a = 3, n = 10$ .
5. Solve  $a^b \pmod{n}$  with  $b$  very large using  $\varphi(n)$ .

Sieve once, and modular math becomes effortless forever after.

## 56. Linear Algebra (Gaussian Elimination, LU, SVD)

Linear algebra gives algorithms their mathematical backbone. From solving equations to powering ML models, it’s the hidden engine behind optimization, geometry, and numerical computation.

In this section, we’ll focus on the algorithmic toolkit:

- Gaussian Elimination (solve systems, invert matrices)
- LU Decomposition (efficient repeated solving)
- SVD (Singular Value Decomposition) overview

You'll see how algebra becomes code , step by step.

## 1. Systems of Linear Equations

We want to solve:

$$A \cdot x = b$$

where ( A ) is an  $n \times n$  matrix, and ( x, b ) are vectors.

For example:

$$\begin{cases} 2x + 3y = 8 \\ x + 2y = 5 \end{cases}$$

The solution is the intersection of two lines. In general,  $A^{-1}b$  gives ( x ), but we usually solve it more directly using Gaussian elimination.

## 2. Gaussian Elimination (Row Reduction)

Idea: Transform ( [A|b] ) (augmented matrix) into upper-triangular form, then back-substitute.

Steps:

1. For each row, select a pivot (non-zero leading element).
2. Eliminate below it using row operations.
3. After all pivots, back-substitute to get the solution.

## A. Implementation (C)

```
const double EPS = 1e-9;

vector<double> gauss(vector<vector<double>> A, vector<double> b) {
    int n = A.size();
    for (int i = 0; i < n; i++) {
        // 1. Find pivot
        int pivot = i;
        for (int j = i + 1; j < n; j++)
            if (fabs(A[j][i]) > fabs(A[pivot][i]))
                pivot = j;
        swap(A[i], A[pivot]);
        swap(b[i], b[pivot]);
```

```

// 2. Normalize pivot row
double div = A[i][i];
if (fabs(div) < EPS) continue;
for (int k = i; k < n; k++) A[i][k] /= div;
b[i] /= div;

// 3. Eliminate below
for (int j = i + 1; j < n; j++) {
    double factor = A[j][i];
    for (int k = i; k < n; k++) A[j][k] -= factor * A[i][k];
    b[j] -= factor * b[i];
}
}

// 4. Back substitution
vector<double> x(n);
for (int i = n - 1; i >= 0; i--) {
    x[i] = b[i];
    for (int j = i + 1; j < n; j++)
        x[i] -= A[i][j] * x[j];
}
return x;
}

```

Time complexity: (  $O(n^3)$  )

## B. Example

Solve:

$$\begin{cases} 2x + 3y = 8 \\ x + 2y = 5 \end{cases}$$

Augmented matrix:

$$\left[ \begin{array}{cc|cc} 2 & 3 & 8 & 1 \\ 1 & 2 & 5 & 0 \end{array} \right]$$

Reduce:

- Row2  $\leftarrow$  Row2  $- \frac{1}{2}$  Row1  $\rightarrow [1, 2|5] \rightarrow [0, 0.5|1]$  - Back substitute  $\rightarrow (y = 2, x = 1)$

### 3. LU Decomposition

LU factorization expresses:

$$A = L \cdot U$$

where ( L ) is lower-triangular (1s on diagonal), ( U ) is upper-triangular.

This allows solving ( A x = b ) in two triangular solves:

1. Solve ( L y = b )
2. Solve ( U x = y )

Efficient when solving for multiple b's (same A).

#### A. Decomposition Algorithm

```
void lu_decompose(vector<vector<double>>& A, vector<vector<double>>& L, vector<vector<double>>& U) {
    int n = A.size();
    L.assign(n, vector<double>(n, 0));
    U.assign(n, vector<double>(n, 0));

    for (int i = 0; i < n; i++) {
        // Upper
        for (int k = i; k < n; k++) {
            double sum = 0;
            for (int j = 0; j < i; j++)
                sum += L[i][j] * U[j][k];
            U[i][k] = A[i][k] - sum;
        }
        // Lower
        for (int k = i; k < n; k++) {
            if (i == k) L[i][i] = 1;
            else {
                double sum = 0;
                for (int j = 0; j < i; j++)
                    sum += L[k][j] * U[j][i];
                L[k][i] = (A[k][i] - sum) / U[i][i];
            }
        }
    }
}
```

Solve with forward + backward substitution.

## 4. Singular Value Decomposition (SVD)

SVD generalizes diagonalization for non-square matrices:

$$A = U\Sigma V^T$$

Where:

- ( $U$ ): left singular vectors (orthogonal)-  $\Sigma$ : diagonal of singular values-  $V^T$ : right singular vectors Applications:
- Data compression (PCA)- Noise reduction- Rank estimation- Pseudoinverse  $A^+ = V\Sigma^{-1}U^T$  In practice, use libraries (e.g. LAPACK, Eigen).

## 5. Numerical Stability and Pivoting

In floating-point math:

- Always pick largest pivot (partial pivoting)- Avoid dividing by small numbers- Use EPS = 1e-9 threshold Small numerical errors can amplify quickly , stability is key.

## 6. Tiny Code

```
vector<vector<double>> A = {{2, 3}, {1, 2}};
vector<double> b = {8, 5};
auto x = gauss(A, b);
// Output: x = [1, 2]
```

## 7. Summary

Algorithm	Purpose	Complexity	Notes
Gaussian Elimination	Solve $Ax=b$	$(On^3)$	Direct method
LU Decomposition	Repeated solves	$(On^3)$	Triangular factorization
SVD	General decomposition	$(On^3)$	Robust, versatile

## Why It Matters

Linear algebra is the language of algorithms , it solves equations, optimizes functions, and projects data. Whether building solvers or neural networks, these methods are your foundation.

“Every algorithm lives in a vector space , it just needs a basis to express itself.”

## Try It Yourself

1. Solve a  $3 \times 3$  linear system with Gaussian elimination.
2. Implement LU decomposition and test  $L \cdot U = A$ .
3. Use LU to solve multiple ( b ) vectors.
4. Explore SVD using a math library; compute singular values of a  $2 \times 2$  matrix.
5. Compare results between naive and pivoted elimination for unstable systems.

Start with row operations , and you’ll see how geometry and algebra merge into code.

## 57. FFT and NTT (Fast Transforms)

The Fast Fourier Transform (FFT) is one of the most beautiful and practical algorithms ever invented. It converts data between time (or coefficient) domain and frequency (or point) domain efficiently. The Number Theoretic Transform (NTT) is its modular counterpart for integer arithmetic , ideal for polynomial multiplication under a modulus.

This section covers:

- Why we need transforms- Discrete Fourier Transform (DFT)- Cooley-Tukey FFT (complex numbers)- NTT (modular version)- Applications (polynomial multiplication, convolution)

### 1. Motivation

Suppose you want to multiply two polynomials:

$$A(x) = a_0 + a_1x + a_2x^2$$

$$B(x) = b_0 + b_1x + b_2x^2$$

Their product has coefficients:

$$c_k = \sum_{i+j=k} a_i \cdot b_j$$

This is convolution:

$$C = A * B$$

Naively, this takes (  $O(n^2)$  ). FFT reduces it to (  $O(n \log n)$  ).

## 2. Discrete Fourier Transform (DFT)

The DFT maps coefficients  $a_0, a_1, \dots, a_{n-1}$  to evaluations at (  $n$  )-th roots of unity:

$$A_k = \sum_{j=0}^{n-1} a_j \cdot e^{-2\pi i \cdot jk/n}$$

and the inverse transform recovers  $a_j$  from  $A_k$ .

## 3. Cooley-Tukey FFT

Key idea: recursively split the sum into even and odd parts:

$$A_k = A_{even}(w_n^2) + w_n^k \cdot A_{odd}(w_n^2)$$

Where  $w_n = e^{-2\pi i/n}$  is an (  $n$  )-th root of unity.

### Implementation (C++)

```
#include <complex>
#include <vector>
#include <cmath>
using namespace std;

using cd = complex<double>;
const double PI = acos(-1);

void fft(vector<cd> &a, bool invert) {
    int n = a.size();
    for (int i = 1, j = 0; i < n; i++) {
        int bit = n >> 1;
        for (; j & bit; bit >>= 1) j ^= bit;
        j ^= bit;
        if (i < j) swap(a[i], a[j]);
    }
    if (invert) {
        for (int i = 0; i < n; i++) {
            a[i] /= sqrt(n);
            if (a[i].real() < 0) a[i].real() *= -1;
        }
    }
}
```

```

    }

    for (int len = 2; len <= n; len <= 1) {
        double ang = 2 * PI / len * (invert ? -1 : 1);
        cd wlen(cos(ang), sin(ang));
        for (int i = 0; i < n; i += len) {
            cd w(1);
            for (int j = 0; j < len / 2; j++) {
                cd u = a[i + j], v = a[i + j + len / 2] * w;
                a[i + j] = u + v;
                a[i + j + len / 2] = u - v;
                w *= wlen;
            }
        }
    }

    if (invert) {
        for (cd &x : a) x /= n;
    }
}

```

## Polynomial Multiplication with FFT

```

vector<long long> multiply(vector<int> const& a, vector<int> const& b) {
    vector<cd> fa(a.begin(), a.end()), fb(b.begin(), b.end());
    int n = 1;
    while (n < (int)a.size() + (int)b.size()) n <= 1;
    fa.resize(n);
    fb.resize(n);

    fft(fa, false);
    fft(fb, false);
    for (int i = 0; i < n; i++) fa[i] *= fb[i];
    fft(fa, true);

    vector<long long> result(n);
    for (int i = 0; i < n; i++)
        result[i] = llround(fa[i].real());
    return result;
}

```

Complexity: (  $O(n \log n)$  )

#### 4. Number Theoretic Transform (NTT)

FFT uses complex numbers , NTT uses modular arithmetic with roots of unity mod p. We need a prime ( p ) such that:

$$p = c \cdot 2^k + 1$$

so a primitive root ( g ) exists.

Popular choices:

- (  $p = 998244353, g = 3$  )- (  $p = 7340033, g = 3$  )

#### Implementation (NTT)

```
const int MOD = 998244353;
const int G = 3;

int modpow(int a, int b) {
    long long res = 1;
    while (b) {
        if (b & 1) res = res * a % MOD;
        a = 1LL * a * a % MOD;
        b >>= 1;
    }
    return res;
}

void ntt(vector<int> &a, bool invert) {
    int n = a.size();
    for (int i = 1, j = 0; i < n; i++) {
        int bit = n >> 1;
        for (; j & bit; bit >>= 1) j ^= bit;
        j ^= bit;
        if (i < j) swap(a[i], a[j]);
    }
    for (int len = 2; len <= n; len <= 1) {
        int wlen = modpow(G, (MOD - 1) / len);
        if (invert) wlen = modpow(wlen, MOD - 2);
        for (int i = 0; i < n; i += len) {
            long long w = 1;
```

```

        for (int j = 0; j < len / 2; j++) {
            int u = a[i + j];
            int v = (int)(a[i + j + len / 2] * w % MOD);
            a[i + j] = u + v < MOD ? u + v : u + v - MOD;
            a[i + j + len / 2] = u - v >= 0 ? u - v : u - v + MOD;
            w = w * wlen % MOD;
        }
    }
}

if (invert) {
    int inv_n = modpow(n, MOD - 2);
    for (int &x : a) x = 1LL * x * inv_n % MOD;
}
}

```

## 5. Applications

1. Polynomial Multiplication: (  $O(n \log n)$  )
2. Convolution: digital signal processing
3. Big Integer Multiplication (Karatsuba, FFT)
4. Subset Convolution and combinatorial transforms
5. Number-theoretic sums (NTT-friendly modulus)

## 6. Tiny Code

```

vector<int> A = {1, 2, 3};
vector<int> B = {4, 5, 6};
// Result = {4, 13, 28, 27, 18}
auto C = multiply(A, B);

```

## 7. Summary

Algorithm	Domain	Complexity	Type
DFT	Complex	( $O(n^2)$ )	Naive
FFT	Complex	( $O(n \log n)$ )	Recursive
NTT	Modular	( $O(n \log n)$ )	Integer arithmetic

## Why It Matters

FFT and NTT bring polynomial algebra to life. They turn slow convolutions into lightning-fast transforms. From multiplying huge integers to compressing signals, they're the ultimate divide-and-conquer on structure.

“To multiply polynomials fast, you first turn them into music , then back again.”

## Try It Yourself

1. Multiply  $(1 + 2x + 3x^2)$  and  $(4 + 5x + 6x^2)$  using FFT.
2. Implement NTT over 998244353 and verify results mod p.
3. Compare (  $On^2$  ) vs FFT performance for n = 1024.
4. Experiment with inverse FFT (invert = true).
5. Explore circular convolution for signal data.

Once you master FFT/NTT, you hold the power of speed in algebraic computation.

## 58. Numerical Methods (Newton, Simpson, Runge-Kutta)

Numerical methods let us approximate solutions when exact algebraic answers are hard or impossible. They are the foundation of scientific computing, simulation, and optimization , bridging the gap between continuous math and discrete computation.

In this section, we'll explore three classics:

- Newton-Raphson: root finding- Simpson's Rule: numerical integration- Runge-Kutta (RK4): solving differential equations These algorithms showcase how iteration, approximation, and convergence build powerful tools.

### 1. Newton-Raphson Method

Used to find a root of (  $f(x) = 0$  ). Starting from a guess  $x_0$ , iteratively refine:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Convergence is quadratic if ( f ) is smooth and  $x_0$  is close enough.

## A. Example

Solve (  $f(x) = x^2 - 2 = 0$  ) We know root =  $\sqrt{2}$

Start  $x_0 = 1$

Iter	$x_n$	$(fx_n)$	$(f'x_n)$	$x_{n+1}$
0	1.000	-1.000	2.000	1.500
1	1.500	0.250	3.000	1.417
2	1.417	0.006	2.834	1.414

Converged:  $1.414 \approx \sqrt{2}$

## B. Implementation

```
#include <math.h>
#include <stdio.h>

double f(double x) { return x * x - 2; }
double df(double x) { return 2 * x; }

double newton(double x0) {
    for (int i = 0; i < 20; i++) {
        double fx = f(x0);
        double dfx = df(x0);
        if (fabs(fx) < 1e-9) break;
        x0 = x0 - fx / dfx;
    }
    return x0;
}

int main() {
    printf("Root: %.6f\n", newton(1.0)); // 1.414214
}
```

Time Complexity: (  $O(k)$  ) iterations, each (  $O(1)$  )

## 2. Simpson's Rule (Numerical Integration)

When you can't integrate (  $f(x)$  ) analytically, approximate the area under the curve.

Divide interval (  $[a, b]$  ) into even (  $n$  ) subintervals (width (  $h$  )).

$$I \approx \frac{h}{3} \left( f(a) + 4 \sum f(x_{odd}) + 2 \sum f(x_{even}) + f(b) \right)$$

### A. Implementation

```
#include <math.h>
#include <stdio.h>

double f(double x) { return x * x; } // integrate x^2

double simpson(double a, double b, int n) {
    double h = (b - a) / n;
    double s = f(a) + f(b);
    for (int i = 1; i < n; i++) {
        double x = a + i * h;
        s += f(x) * (i % 2 == 0 ? 2 : 4);
    }
    return s * h / 3;
}

int main() {
    printf("  x^2 dx  %.6f\n", simpson(0, 1, 100)); // ~0.333333
}
```

Accuracy: (  $O(h^4)$  ) Note: (  $n$  ) must be even.

### B. Example

$$\int_0^1 x^2 dx = \frac{1}{3}$$

With (  $n = 100$  ), Simpson gives ( 0.333333 ).

### 3. Runge-Kutta (RK4)

Used to solve first-order ODEs:

$$y' = f(x, y), \quad y(x_0) = y_0$$

RK4 Formula:

$$k_1 = f(x_n, y_n) \quad k_2 = f\left(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_1\right) \quad k_3 = f\left(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_2\right) \quad k_4 = f(x_n + h, y_n + hk_3) \quad y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

Accuracy: (  $O(h^4)$  )

#### A. Example

Solve (  $y' = x + y$  ), (  $y(0) = 1$  ), step (  $h = 0.1$  ).

Each iteration refines (  $y$  ) with weighted slope average.

#### B. Implementation

```
#include <stdio.h>

double f(double x, double y) {
    return x + y;
}

double runge_kutta(double x0, double y0, double h, double xn) {
    double x = x0, y = y0;
    while (x < xn) {
        double k1 = f(x, y);
        double k2 = f(x + h / 2, y + h * k1 / 2);
        double k3 = f(x + h / 2, y + h * k2 / 2);
        double k4 = f(x + h, y + h * k3);
        y += h * (k1 + 2*k2 + 2*k3 + k4) / 6;
        x += h;
    }
    return y;
}
```

```

int main() {
    printf("y(0.1) %.6f\n", runge_kutta(0, 1, 0.1, 0.1));
}

```

#### 4. Tiny Code Summary

Method	Purpose	Formula	Accuracy	Type
Newton-Raphson	Root finding	$x_{n+1} = x_n - \frac{f}{f'}$	Quadratic	Iterative
Simpson's Rule	Integration	( $h/3(\dots)$ )	$(Oh^4)$	Deterministic
Runge-Kutta (RK4)	ODEs	Weighted slope avg	$(Oh^4)$	Iterative

#### 5. Numerical Stability

- Small step ( $h$ ): better accuracy, more cost- Large ( $h$ ): faster, less stable- Always check convergence ( $|x_{n+1} - x_n| < \varepsilon$ )- Avoid division by small derivatives in Newton's method

#### Why It Matters

Numerical methods let computers simulate the continuous world. From physics to AI training, they solve what calculus cannot symbolically.

“When equations won’t talk, iterate , and they’ll whisper their answers.”

#### Try It Yourself

1. Use Newton's method for  $\cos x - x = 0$ .
2. Approximate  $\int_0^{\pi/2} \sin x dx$  with Simpson's rule.
3. Solve  $y' = y - x^2 + 1$ ,  $y(0) = 0.5$  using RK4.
4. Compare RK4 with Euler's method for the same ODE.
5. Experiment with step sizes  $h \in \{0.1, 0.01, 0.001\}$  and observe convergence.

Numerical thinking turns continuous problems into iterative algorithms , precise enough to power every simulation and solver you'll ever write.

## 59. Mathematical Optimization (Simplex, Gradient, Convex)

Mathematical optimization is about finding the best solution , smallest cost, largest profit, shortest path , under given constraints. It's the heart of machine learning, operations research, and engineering design.

In this section, we'll explore three pillars:

- Simplex Method , for linear programs
- Gradient Descent , for continuous optimization
- Convex Optimization , the theory ensuring global optima

### 1. What Is Optimization?

A general optimization problem looks like:

$$\min_x f(x)$$

subject to constraints:

$$g_i(x) \leq 0, \quad h_j(x) = 0$$

When (  $f$  ) and  $g_i, h_j$  are linear, it's a Linear Program (LP). When (  $f$  ) is differentiable, we can use gradients. When (  $f$  ) is convex, every local minimum is global , the ideal world.

### 2. The Simplex Method (Linear Programming)

A linear program has the form:

$$\max c^T x$$

subject to

$$Ax \leq b, \quad x \geq 0$$

Geometrically, each constraint forms a half-space. The feasible region is a convex polytope, and the optimum lies at a vertex.

#### A. Example

Maximize (  $z = 3x + 2y$  ) subject to

$$\begin{cases} 2x + y \leq 18 \\ 2x + 3y \leq 42 \\ x, y \geq 0 \end{cases}$$

Solution: (  $x=9, y=8$  ), (  $z=43$  )

## B. Algorithm (Sketch)

1. Convert inequalities to equalities by adding slack variables.
2. Initialize at a vertex (basic feasible solution).
3. At each step:
  - Choose entering variable (most negative coefficient in objective). - Choose leaving variable (min ratio test). - Pivot to new vertex.
  - 4. Repeat until optimal.

## C. Implementation (Simplified Pseudocode)

```
// Basic simplex-like outline
while (exists negative coefficient in objective row) {
    choose entering column j;
    choose leaving row i (min b[i]/a[i][j]);
    pivot(i, j);
}
```

Libraries (like GLPK or Eigen) handle full implementations.

Time Complexity: worst (  $O2^n$  ), but fast in practice.

## 3. Gradient Descent

For differentiable (  $f(x)$  ), we move opposite the gradient:

$$x_{k+1} = x_k - \eta \nabla f(x_k)$$

where  $\eta$  is the learning rate.

Intuition: (  $f(x)$  ) points uphill, so step opposite it.

## A. Example

Minimize (  $f(x) = (x-3)^2$  )

$$f'(x) = 2(x - 3)$$

Start  $x_0 = 0$ ,  $\eta = 0.1$

Iter	$x_k$	$(fx_k)$	Gradient	New (x)
0	0	9	-6	0.6
1	0.6	5.76	-4.8	1.08
2	1.08	3.69	-3.84	1.46
...	$\rightarrow 3$	$\rightarrow 0$	$\rightarrow 0$	$\rightarrow 3$

Converges to (  $x = 3$  )

## B. Implementation

```
#include <math.h>
#include <stdio.h>

double f(double x) { return (x - 3) * (x - 3); }
double df(double x) { return 2 * (x - 3); }

double gradient_descent(double x0, double lr) {
    for (int i = 0; i < 100; i++) {
        double g = df(x0);
        if (fabs(g) < 1e-6) break;
        x0 -= lr * g;
    }
    return x0;
}

int main() {
    printf("Min at x = %.6f\n", gradient_descent(0, 0.1));
}
```

## C. Variants

- Momentum: ( $v = v + 1 - \beta f(x)$ ) - Adam: adaptive learning rates- Stochastic Gradient Descent (SGD): random subset of data All used heavily in machine learning.

## 4. Convex Optimization

A function ( $f$ ) is convex if:

$$f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda)f(y)$$

This means any local minimum is global.

Examples:

- ( $f(x) = x^2$ ) (convex)- ( $f(x) = x^3$ ) (not convex) For convex functions with linear constraints, gradient-based methods always converge to the global optimum.

## A. Checking Convexity

- 1D: ( $f'(x) \geq 0$ ) - Multivariate: Hessian ( $\nabla^2 f(x)$ ) is positive semidefinite

## 5. Applications

- Linear Programming (Simplex): logistics, scheduling- Quadratic Programming: portfolio optimization- Gradient Methods: ML, curve fitting- Convex Programs: control systems, regularization

## 6. Tiny Code

Simple gradient descent to minimize ( $f(x,y)=x^2+y^2$ ):

```
double f(double x, double y) { return x*x + y*y; }
void grad(double x, double y, double *gx, double *gy) {
    *gx = 2*x; *gy = 2*y;
}

void optimize() {
    double x=5, y=3, lr=0.1;
    for(int i=0; i<100; i++){
        double gx, gy;
```

```

        grad(x, y, &gx, &gy);
        x -= lr * gx;
        y -= lr * gy;
    }
    printf("Min at (%.3f, %.3f)\n", x, y);
}

```

## 7. Summary

Algorithm	Domain	Complexity	Notes
Simplex	Linear	Polynomial (average case)	LP solver
Gradient Descent	Continuous	$O(k)$	Needs step size
Convex Methods	Convex	$O(k \log \frac{1}{\epsilon})$	Global optima guaranteed

## Why It Matters

Optimization turns math into decisions. From fitting curves to planning resources, it formalizes trade-offs and efficiency. It's where computation meets purpose , finding the best in all possible worlds.

“Every algorithm is, at heart, an optimizer , searching for something better.”

## Try It Yourself

1. Solve a linear program with 2 constraints manually via Simplex.
2. Implement gradient descent for  $f(x) = (x - 5)^2 + 2$ .
3. Add momentum to your gradient descent loop.
4. Check convexity by plotting  $f(x) = x^4 - 3x^2$ .
5. Experiment with learning rates: too small leads to slow convergence; too large can diverge.

Mastering optimization means mastering how algorithms improve themselves , step by step, iteration by iteration.

## 60. Algebraic Tricks and Transform Techniques

In algorithm design, algebra isn't just theory , it's a toolbox for transforming problems. By expressing computations algebraically, we can simplify, accelerate, or generalize solutions. This section surveys common algebraic techniques that turn hard problems into manageable ones.

We'll explore:

- Algebraic identities and factorizations
- Generating functions and transforms
- Convolution tricks
- Polynomial methods and FFT applications
- Matrix and linear transforms for acceleration

### 1. Algebraic Identities

These let you rewrite or decompose expressions to reveal structure or reduce complexity.

Classic Forms:

- Difference of squares:

$$a^2 - b^2 = (a - b)(a + b)$$

- Sum of cubes:

$$a^3 + b^3 = (a + b)(a^2 - ab + b^2)$$

- Square of sum:

$$(a + b)^2 = a^2 + 2ab + b^2$$

Used in dynamic programming, geometry, and optimization when simplifying recurrence terms or constraints.

Example: Transforming  $(x + y)^2$  lets you compute both  $x^2 + y^2$  and cross terms efficiently.

### 2. Generating Functions

A generating function encodes a sequence  $a_0, a_1, a_2, \dots$  into a formal power series:

$$G(x) = a_0 + a_1x + a_2x^2 + \dots$$

They turn recurrence relations and counting problems into algebraic equations.

Example: Fibonacci sequence

$$F(x) = F_0 + F_1x + F_2x^2 + \dots$$

with recurrence  $F_n = F_{n-1} + F_{n-2}$

Solve algebraically:

$$F(x) = \frac{x}{1-x-x^2}$$

Applications: combinatorics, probability, counting partitions.

### 3. Convolution Tricks

Convolution arises in combining sequences:

$$(c_n) = (a * b) * n = \sum *i = 0^n a_i b_{n-i}$$

Naive computation: (  $O(n^2)$  ) Using Fast Fourier Transform (FFT): (  $O(n \log n)$  )

Example: Polynomial multiplication Let

$$A(x) = a_0 + a_1x + a_2x^2, \quad B(x) = b_0 + b_1x + b_2x^2$$

Then (  $C(x) = A(x)B(x)$  ) gives coefficients by convolution.

This trick is used in:

- Large integer multiplication- Pattern matching (cross-correlation)- Subset sum acceleration

### 4. Polynomial Methods

Many algorithmic problems can be represented as polynomials, where coefficients encode combinatorial structure.

#### A. Polynomial Interpolation

Given (  $n+1$  ) points, there's a unique degree-(  $n$  ) polynomial passing through them.

Used in error correction, FFT-based reconstruction, and number-theoretic transforms.

Lagrange Interpolation:

$$P(x) = \sum_i y_i \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}$$

## B. Root Representation

Solve equations or check identities by working modulo a polynomial. Used in finite fields and coding theory (e.g., Reed-Solomon).

## 5. Transform Techniques

Transforms convert problems to simpler domains where operations become efficient.

Transform	Converts	Key Property	Used In
FFT / NTT	Time Frequency	Convolution → Multiplication	Signal, polynomial mult
Z-Transform	Sequence Function	Recurrence solving	DSP, control
Laplace Transform	Function Algebraic	Diff. eq. → Algebraic eq.	Continuous systems
Walsh-Hadamard Transform	Boolean vectors	XOR convolution	Subset sum, SOS DP

Example: Subset Convolution via FWT

For all subsets ( S ):

$$f'(S) = \sum_{T \subseteq S} f(T)$$

Use Fast Walsh-Hadamard Transform (FWHT) to compute in (  $O(n2^n)$  ) instead of (  $O(3^n)$  ).

## 6. Matrix Tricks

Matrix algebra enables transformations and compact formulations.

- Matrix exponentiation: solve recurrences in  $O(\log n)$
- Diagonalization:  $A = PDP^{-1}$ , then  $A^k = PD^kP^{-1}$
- Fast power: speeds up Fibonacci, linear recurrences, Markov chains

Example: Fibonacci

$$\begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

## 7. Tiny Code

Polynomial Multiplication via FFT (Pseudo-C):

```
// Outline using complex FFT library
fft(A, false);
fft(B, false);
for (int i = 0; i < n; i++)
    C[i] = A[i] * B[i];
fft(C, true); // inverse
```

Matrix Power (Fibonacci):

```
void matmul(long long A[2][2], long long B[2][2]) {
    long long C[2][2] = {{0}};
    for (int i=0;i<2;i++)
        for (int j=0;j<2;j++)
            for (int k=0;k<2;k++)
                C[i][j] += A[i][k]*B[k][j];
    memcpy(A, C, sizeof(C));
}

void matpow(long long A[2][2], int n) {
    long long R[2][2] = {{1,0},{0,1}};
    while(n){
        if(n&1) matmul(R,A);
        matmul(A,A);
        n>>=1;
    }
    memcpy(A, R, sizeof(R));
}
```

## 8. Summary

Technique	Purpose	Speedup
Algebraic Identities	Simplify expressions	Constant factor
Generating Functions	Solve recurrences	Conceptual
FFT / Convolution	Combine sequences fast	( $O(n^2) \rightarrow O(n \log n)$ )
Polynomial Interpolation	Reconstruction	( $O(n^2) \rightarrow O(n \log^2 n)$ )
Matrix Tricks	Accelerate recurrences	( $O(n) \rightarrow O(\log n)$ )

## Why It Matters

Algebra turns computation into structure. By rewriting problems in algebraic form, you reveal hidden symmetries, exploit fast transforms, and find elegant solutions. It's not magic , it's the math beneath performance.

“The smartest code is often the one that solves itself on paper first.”

## Try It Yourself

1. Multiply two polynomials using FFT.
2. Represent Fibonacci as a matrix and compute  $F_{100}$ .
3. Use generating functions to count coin change ways.
4. Implement subset sum via Walsh-Hadamard transform.
5. Derive a recurrence and solve it algebraically.

Understanding algebraic tricks makes you not just a coder, but a mathematical engineer , bending structure to will.

# Chapter 7. Strings and Text Algorithms

## 61. String Matching (KMP, Z, Rabin-Karp, Boyer-Moore)

String matching is one of the oldest and most fundamental problems in computer science: given a text ( T ) of length ( n ) and a pattern ( P ) of length ( m ), find all positions where ( P ) appears in ( T ).

This section walks you through both naive and efficient algorithms , from the straightforward brute-force method to elegant linear-time solutions like KMP and Z-algorithm, and clever heuristics like Boyer-Moore and Rabin-Karp.

### 1. Problem Setup

We're given:

- Text:  $T = t_1 t_2 \dots t_n$ - Pattern:  $P = p_1 p_2 \dots p_m$  Goal: find all ( i ) such that

$$T[i \dots i + m - 1] = P[1 \dots m]$$

Naive solution: compare ( P ) with every substring of ( T ) Time complexity: ( O(nm) )

We'll now see how to reduce it to ( O(n + m) ) or close.

## 2. Knuth-Morris-Pratt (KMP)

KMP avoids rechecking characters by precomputing overlaps within the pattern.

It builds a prefix-function (also called failure function), which tells how much to shift when a mismatch happens.

### A. Prefix Function

For each position ( $i$ ), compute  $\pi[i] = \text{length of longest prefix that's also a suffix of } (P[1..i])$ .

Example: Pattern ababc

i	P[i]	[i]
1	a	0
2	b	0
3	a	1
4	b	2
5	c	0

### B. Search Phase

Use  $\pi[]$  to skip mismatched prefixes in the text.

Time Complexity: (  $O(n + m)$  ) Space: (  $O(m)$  )

Tiny Code (C)

```
void compute_pi(char *p, int m, int pi[]) {
    pi[0] = 0;
    for (int i = 1, k = 0; i < m; i++) {
        while (k > 0 && p[k] != p[i]) k = pi[k-1];
        if (p[k] == p[i]) k++;
        pi[i] = k;
    }
}

void kmp_search(char *t, char *p) {
    int n = strlen(t), m = strlen(p);
    int pi[m]; compute_pi(p, m, pi);
    for (int i = 0, k = 0; i < n; i++) {
```

```

while (k > 0 && p[k] != t[i]) k = pi[k-1];
if (p[k] == t[i]) k++;
if (k == m) {
    printf("Found at %d\n", i - m + 1);
    k = pi[k-1];
}
}
}

```

### 3. Z-Algorithm

Z-algorithm computes the Z-array,  
where  $Z[i]$  = length of the longest substring starting at  $i$  that matches the prefix of  $P$ .

To match  $P$  in  $T$ , build the string:

$$S = P + \# + T$$

Then every  $i$  where  $Z[i] = |P|$  corresponds to a match.

Time:  $O(n + m)$

Simple and elegant.

Example:

```

P = "aba", T = "ababa"
S = "aba#ababa"
Z = [0,0,1,0,3,0,1,0]
Match at index 0, 2

```

### 4. Rabin-Karp (Rolling Hash)

Instead of comparing strings character-by-character, compute a hash for each window in (  $T$  ), and compare hashes.

$$h(s_1 s_2 \dots s_m) = (s_1 b^{m-1} + s_2 b^{m-2} + \dots + s_m) \bmod M$$

Use a rolling hash to update in (  $O(1)$  ) per shift.

Time: average (  $O(n + m)$  ), worst (  $O(nm)$  ) Good for multiple pattern search.

Tiny Code (Rolling Hash)

```

#define B 256
#define M 101

void rabin_karp(char *t, char *p) {
    int n = strlen(t), m = strlen(p);
    int h = 1, pHash = 0, tHash = 0;
    for (int i = 0; i < m-1; i++) h = (h*B) % M;
    for (int i = 0; i < m; i++) {
        pHash = (B*pHash + p[i]) % M;
        tHash = (B*tHash + t[i]) % M;
    }
    for (int i = 0; i <= n-m; i++) {
        if (pHash == tHash && strncmp(&t[i], p, m) == 0)
            printf("Found at %d\n", i);
        if (i < n-m)
            tHash = (B*(tHash - t[i]*h) + t[i+m]) % M;
        if (tHash < 0) tHash += M;
    }
}

```

## 5. Boyer-Moore (Heuristic Skipping)

Boyer-Moore compares from right to left and uses two heuristics:

1. Bad Character Rule When mismatch at ( j ), shift pattern so next occurrence of ( T[i] ) in ( P ) aligns.
2. Good Suffix Rule Shift pattern so a suffix of matched portion aligns with another occurrence.

Time: ( O(n/m) ) on average Practical and fast, especially for English text.

## 6. Summary

Algorithm	Time	Space	Idea	Best For
Naive	(O(nm))	(O(1))	Direct compare	Simple cases
KMP	(O(n+m))	(O(m))	Prefix overlap	General use
Z	(O(n+m))	(O(n+m))	Prefix matching	Pattern prep
Rabin-Karp	(O(n+m)) avg	(O(1))	Hashing	Multi-pattern
Boyer-Moore	(O(n/m)) avg	(Om + σ)	Right-to-left skip	Long texts

## Why It Matters

String matching powers text editors, DNA search, spam filters, and search engines. These algorithms show how structure and clever preprocessing turn brute force into elegance.

“To find is human, to match efficiently is divine.”

## Try It Yourself

1. Implement KMP and print all matches in a sentence.
2. Use Rabin-Karp to find multiple keywords.
3. Compare running times on large text files.
4. Modify KMP for case-insensitive matching.
5. Visualize prefix function computation step-by-step.

By mastering these, you’ll wield the foundation of pattern discovery , the art of finding order in streams of symbols.

## 62. Multi-Pattern Search (Aho-Corasick)

So far, we’ve matched one pattern against a text. But what if we have many patterns , say, a dictionary of keywords , and we want to find all occurrences of all patterns in a single pass?

That’s where the Aho-Corasick algorithm shines. It builds a trie with failure links, turning multiple patterns into one efficient automaton. Think of it as “KMP for many words at once.”

### 1. Problem Setup

Given:

- A text ( T ) of length ( n )- A set of patterns  $P_1, P_2, \dots, P_k$  with total length  $m = \sum |P_i|$

Goal: find all occurrences of every  $P_i$  in ( T ).

Naive solution: Run KMP for each pattern , (  $O(kn)$  )

Better idea: Merge all patterns into a trie, and use failure links to transition on mismatches.

Aho-Corasick achieves  $O(n + m + z)$ , where ( z ) = number of matches reported.

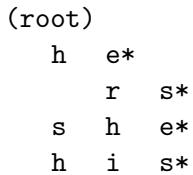
## 2. Trie Construction

Each pattern is inserted into a trie node-by-node.

Example Patterns:

he, she, his, hers

Trie:



Each node may mark an output (end of pattern).

## 3. Failure Links

Failure link of a node points to the longest proper suffix that's also a prefix in the trie.

These links let us “fall back” like KMP.

When mismatch happens, follow failure link to find next possible match.

### Building Failure Links (BFS)

1. Root’s failure = null
2. Children of root → failure = root
3. BFS over nodes:
  - For each edge ( (u, c) → v ): follow failure links from ( u ) until you find ( f ) with edge ( c ) then  $v.\text{fail} = f.c$

### Example

For “he”, “she”, “his”, “hers”:

- `fail("he") = root-fail("hers") = "rs"` path invalid → fallback to "s" if exists So failure links connect partial suffixes.

#### 4. Matching Phase

Now we can process the text in one pass:

```
state = root
for each character c in text:
    while state has no child c and state != root:
        state = state.fail
    if state has child c:
        state = state.child[c]
    else:
        state = root
    if state.output:
        report matches at this position
```

Each transition costs  $O(1)$  amortized. No backtracking , fully linear time.

#### 5. Example Walkthrough

Patterns: he, she, his, hers Text: ahishers

At each character:

```
a → root (no match)
h → go to h
i → go to hi
s → go to his → output "his"
h → fallback → h
e → he → output "he"
r → her → continue
s → hers → output "hers"
```

Outputs: "his", "he", "hers"

#### 6. Tiny Code (C Implementation Sketch)

```

#define ALPHA 26

typedef struct Node {
    struct Node *next[ALPHA];
    struct Node *fail;
    int out;
} Node;

Node* newNode() {
    Node *n = malloc(sizeof(Node));
    return n;
}

void insert(Node *root, char *p) {
    for (int i = 0; p[i]; i++) {
        int c = p[i] - 'a';
        if (!root->next[c]) root->next[c] = newNode();
        root = root->next[c];
    }
    root->out = 1;
}

void build_failures(Node *root) {
    Node *q[10000];
    int front=0, back=0;
    root->fail = root;
    q[back++] = root;
    while (front < back) {
        Node *u = q[front++];
        for (int c=0; c<ALPHA; c++) {
            Node *v = u->next[c];
            if (!v) continue;
            Node *f = u->fail;
            while (f != root && !f->next[c]) f = f->fail;
            if (f->next[c] && f->next[c] != v) v->fail = f->next[c];
            else v->fail = root;
            if (v->fail->out) v->out = 1;
            q[back++] = v;
        }
    }
}

```

## 7. Complexity

Phase	Time	Space
Trie Build	( O(m) )	( O(m) )
Failure Links	( O(m) )	( O(m) )
Search	( O(n + z) )	( O(1) )

Total:  $O(n + m + z)$

## 8. Summary

Step	Purpose
Trie	Merge patterns
Fail Links	Handle mismatches
Outputs	Collect matches
BFS	Build efficiently
One Pass	Match all patterns

## Why It Matters

Aho-Corasick is the core of:

- Spam filters- Intrusion detection (e.g., Snort IDS)- Keyword search in compilers- DNA sequence scanners It's a masterclass in blending automata theory with practical efficiency.

“Why search one word at a time when your algorithm can read the whole dictionary?”

## Try It Yourself

1. Build an automaton for words {"he", "she", "hers"} and trace it manually.
2. Modify code for uppercase letters.
3. Extend to report overlapping matches.
4. Measure runtime vs. naive multi-search.
5. Visualize the failure links in a graph.

Once you grasp Aho-Corasick, you'll see pattern search not as a loop , but as a machine that reads and recognizes.

### 63. Suffix Structures (Suffix Array, Suffix Tree, LCP)

Suffix-based data structures are among the most powerful tools in string algorithms. They enable fast searching, substring queries, pattern matching, and lexicographic operations , all from one fundamental idea:

Represent all suffixes of a string in a structured form.

In this section, we explore three key constructs:

- Suffix Array (SA) - lexicographically sorted suffix indices- Longest Common Prefix (LCP) array - shared prefix lengths between neighbors- Suffix Tree - compressed trie of all suffixes Together, they power many advanced algorithms in text processing, bioinformatics, and compression.

#### 1. Suffix Array (SA)

A suffix array stores all suffixes of a string in lexicographic order, represented by their starting indices.

Example: String banana\$ All suffixes:

Index	Suffix
0	banana\$
1	anana\$
2	nana\$
3	ana\$
4	na\$
5	a\$
6	\$

Sort them:

Sorted Order	Suffix	Index
0	\$	6
1	a\$	5
2	ana\$	3
3	anana\$	1
4	banana\$	0
5	na\$	4
6	nana\$	2

Suffix Array: [6, 5, 3, 1, 0, 4, 2]

### Construction (Prefix Doubling)

We iteratively sort suffixes by first 2 characters, using radix sort on pairs of ranks.

Steps:

1. Assign initial rank by character.
2. Sort by (rank[i], rank[i+k]).
3. Repeat doubling  $k \leftarrow 2k$  until all ranks distinct.

Time Complexity: (  $O(n \log n)$  ) Space: (  $O(n)$  )

Tiny Code (C, Sketch)

```
typedef struct { int idx, rank[2]; } Suffix;
int cmp(Suffix a, Suffix b) {
    return (a.rank[0]==b.rank[0]) ? (a.rank[1]-b.rank[1]) : (a.rank[0]-b.rank[0]);
}

void buildSA(char *s, int n, int sa[]) {
    Suffix suf[n];
    for (int i = 0; i < n; i++) {
        suf[i].idx = i;
        suf[i].rank[0] = s[i];
        suf[i].rank[1] = (i+1<n) ? s[i+1] : -1;
    }
    for (int k = 2; k < 2*n; k *= 2) {
        qsort(suf, n, sizeof(Suffix), cmp);
        int r = 0, rank[n]; rank[suf[0].idx]=0;
        for (int i=1;i<n;i++) {
            if (suf[i].rank[0]!=suf[i-1].rank[0] || suf[i].rank[1]!=suf[i-1].rank[1]) r++;
            rank[suf[i].idx]=r;
        }
        for (int i=0;i<n;i++){
            suf[i].rank[0] = rank[suf[i].idx];
            suf[i].rank[1] = (suf[i].idx+k/2<n)?rank[suf[i].idx+k/2]:-1;
        }
    }
    for (int i=0;i<n;i++) sa[i]=suf[i].idx;
}
```

## 2. Longest Common Prefix (LCP)

The LCP array stores the length of the longest common prefix between consecutive suffixes in SA order.

Example: banana\$

SA	Suffix	LCP
6	\$	0
5	a\$	0
3	ana\$	1
1	anana\$	3
0	banana\$	0
4	na\$	0
2	nana\$	2

So LCP = [0,0,1,3,0,0,2]

## Kasai's Algorithm (Build in O(n))

We compute LCP in one pass using inverse SA:

```
void buildLCP(char *s, int n, int sa[], int lcp[]) {
    int rank[n];
    for (int i=0;i<n;i++) rank[sa[i]]=i;
    int k=0;
    for (int i=0;i<n;i++) {
        if (rank[i]==n-1) { k=0; continue; }
        int j = sa[rank[i]+1];
        while (i+k<n && j+k<n && s[i+k]==s[j+k]) k++;
        lcp[rank[i]]=k;
        if (k>0) k--;
    }
}
```

Time Complexity: ( O(n) )

### 3. Suffix Tree

A suffix tree is a compressed trie of all suffixes.

Each edge holds a substring interval, not individual characters. This gives:

- Construction in ( O(n) ) (Ukkonen's algorithm)- Pattern search in ( O(m) )- Many advanced uses (e.g., longest repeated substring) Example: String: banana\$ Suffix tree edges:

```
(root)
b[0:0] → ...
a[1:1] → ...
n[2:2] → ...
```

Edges compress consecutive letters into intervals like [start:end].

### Comparison

Structure	Space	Build Time	Search
Suffix Array	( O(n) )	( O(n log n) )	( O(m log n) )
LCP Array	( O(n) )	( O(n) )	Range queries
Suffix Tree	( O(n) )	( O(n) )	( O(m) )

Suffix Array + LCP compact Suffix Tree.

### 4. Applications

1. Substring search - binary search in SA
2. Longest repeated substring - max(LCP)
3. Lexicographic order - direct from SA
4. Distinct substrings count = ( n(n+1)/2 - LCP[i] )
5. Pattern frequency - range query in SA using LCP

### 5. Tiny Code (Search via SA)

```

int searchSA(char *t, int n, char *p, int sa[]) {
    int l=0, r=n-1, m=strlen(p);
    while (l <= r) {
        int mid = (l+r)/2;
        int cmp = strncmp(t+sa[mid], p, m);
        if (cmp==0) return sa[mid];
        else if (cmp<0) l=mid+1;
        else r=mid-1;
    }
    return -1;
}

```

## 6. Summary

Concept	Purpose	Complexity
Suffix Array	Sorted suffix indices	( $O(n \log n)$ )
LCP Array	Adjacent suffix overlap	( $O(n)$ )
Suffix Tree	Compressed trie of suffixes	( $O(n)$ )

Together they form the core of advanced string algorithms.

## Why It Matters

Suffix structures reveal hidden order in strings. They turn raw text into searchable, analyzable data , ideal for compression, search engines, and DNA analysis.

“All suffixes, perfectly sorted , the DNA of text.”

## Try It Yourself

1. Build suffix array for `banana$` by hand.
2. Write code to compute LCP and longest repeated substring.
3. Search multiple patterns using binary search on SA.
4. Count distinct substrings from SA + LCP.
5. Compare SA-based vs. tree-based search performance.

Mastering suffix structures equips you to tackle problems that were once “too big” for brute force , now solvable with elegance and order.

## 64. Palindromes and Periodicity (Manacher)

Palindromes are symmetric strings that read the same forwards and backwards , like “level”, “racecar”, or “madam”. They arise naturally in text analysis, bioinformatics, and even in data compression.

This section introduces efficient algorithms to detect and analyze palindromic structure and periodicity in strings, including the legendary Manacher’s Algorithm, which finds all palindromic substrings in linear time.

### 1. What Is a Palindrome?

A string ( S ) is a palindrome if:

$$S[i] = S[n - i + 1] \quad \text{for all } i$$

Examples:

- "abba" is even-length palindrome- "aba" is odd-length palindrome A string may contain many palindromic substrings , our goal is to find all centers efficiently.

### 2. Naive Approach

For each center (between characters or at characters), expand outward while characters match.

```
for each center c:  
    expand left, right while S[l] == S[r]
```

Complexity: (  $O(n^2)$  ), too slow for large strings.

We need something faster , that’s where Manacher’s Algorithm steps in.

### 3. Manacher’s Algorithm ( $O(n)$ )

Manacher’s Algorithm finds the radius of the longest palindrome centered at each position in linear time.

It cleverly reuses previous computations using mirror symmetry and a current right boundary.

## Step-by-Step

1. Preprocess string to handle even-length palindromes: Insert # between characters.

Example:

```
S = "abba"  
T = "^#a#b#b#a#$"  
(^ and $ are sentinels)
```

2. Maintain:

- C: center of rightmost palindrome - R: right boundary - P[i]: palindrome radius at i

3. For each position i:

- mirror position mirror = 2\*C - i - initialize P[i] = min(R - i, P[mirror]) - expand around i while characters match - if new palindrome extends past R, update C and R

4. The maximum value of P[i] gives the longest palindrome.

## Example

```
S = "abba"  
T = "^#a#b#b#a#$"  
P = [0,0,1,0,3,0,3,0,1,0,0]  
Longest radius = 3 → "abba"
```

Tiny Code (C Implementation)

```
int manacher(char *s) {  
    int n = strlen(s);  
    char t[2*n + 3];  
    int p[2*n + 3];  
    int m = 0;  
    t[m++] = '^';  
    for (int i=0; i<n; i++) {  
        t[m++] = '#';  
        t[m++] = s[i];  
    }  
    t[m++] = '#'; t[m++] = '$';  
    t[m] = '\0';
```

```

int c = 0, r = 0, maxLen = 0;
for (int i=1; i<m-1; i++) {
    int mirror = 2*c - i;
    if (i < r)
        p[i] = (r - i < p[mirror]) ? (r - i) : p[mirror];
    else p[i] = 0;
    while (t[i + 1 + p[i]] == t[i - 1 - p[i]])
        p[i]++;
    if (i + p[i] > r) {
        c = i;
        r = i + p[i];
    }
    if (p[i] > maxLen) maxLen = p[i];
}
return maxLen;
}

```

Time Complexity: ( O(n) ) Space: ( O(n) )

#### 4. Periodicity and Repetition

A string ( S ) has a period ( p ) if:

$$S[i] = S[i + p] \text{ for all valid } i$$

Example: abcabcabc has period 3 (abc).

Checking Periodicity:

1. Build prefix function (as in KMP).
2. Let ( n = |S| ),  $p = n - \pi[n - 1]$ .
3. If  $n \bmod p = 0$ , period = ( p ).

Example:

```

S = "ababab"
  = [0,0,1,2,3,4]
p = 6 - 4 = 2
6 mod 2 = 0 → periodic

```

## Tiny Code (Check Periodicity)

```
int period(char *s) {
    int n = strlen(s), pi[n];
    pi[0]=0;
    for(int i=1,k=0;i<n;i++){
        while(k>0 && s[k]!=s[i]) k=pi[k-1];
        if(s[k]==s[i]) k++;
        pi[i]=k;
    }
    int p = n - pi[n-1];
    return (n % p == 0) ? p : n;
}
```

## 5. Applications

- Palindrome Queries: is substring ( $S[l:r]$ ) palindrome? → precompute radii- Longest Palindromic Substring- DNA Symmetry Analysis- Pattern Compression / Period Detection- String Regularity Tests

## 6. Summary

Concept	Purpose	Time
Naive Expand	Simple palindrome check	( $O(n^2)$ )
Manacher	Longest palindromic substring	( $O(n)$ )
KMP Prefix	Period detection	( $O(n)$ )

## Why It Matters

Palindromes reveal hidden symmetries. Manacher's algorithm is a gem , a linear-time mirror-based solution to a quadratic problem.

“In every word, there may hide a reflection.”

## Try It Yourself

1. Run Manacher's algorithm on "abacdfgdcaba".
2. Modify code to print all palindromic substrings.
3. Use prefix function to find smallest period.

4. Combine both to find palindromic periodic substrings.
5. Compare runtime vs. naive expand method.

Understanding palindromes and periodicity teaches how structure emerges from repetition , a central theme in all of algorithmic design.

## 65. Edit Distance and Alignment

Edit distance measures how different two strings are , the minimal number of operations needed to turn one into the other. It's a cornerstone of spell checking, DNA sequence alignment, plagiarism detection, and fuzzy search.

The most common form is the Levenshtein distance, using:

- Insertion (add a character)- Deletion (remove a character)- Substitution (replace a character) We'll also touch on alignment, which generalizes this idea with custom scoring and penalties.

### 1. Problem Definition

Given two strings ( A ) and ( B ), find the minimum number of edits to convert  $A \rightarrow B$ .

If ( A = “kitten” ) ( B = “sitting” )

One optimal sequence:

```
kitten → sitten (substitute 'k'→'s')
sitten → sittin (substitute 'e'→'i')
sittin → sitting (insert 'g')
```

So edit distance = 3.

### 2. Dynamic Programming Solution

Let  $dp[i][j]$  be the minimum edits to convert  $A[0..i - 1] \rightarrow B[0..j - 1]$ .

Recurrence:

$$dp[i][j] = \begin{cases} dp[i - 1][j - 1], & \text{if } A[i - 1] = B[j - 1], \\ 1 + \min (dp[i - 1][j], dp[i][j - 1], dp[i - 1][j - 1]), & \text{otherwise} \end{cases}$$

Where: -  $dp[i - 1][j]$ : delete from A -  $dp[i][j - 1]$ : insert into A -  $dp[i - 1][j - 1]$ : substitute

Base cases:

$$dp[0][j] = j, \quad dp[i][0] = i$$

Time complexity:  $O(|A||B|)$

### Example

A = kitten, B = sitting

	“ ”	s	i	t	t	i	n	g
“ ”	0	1	2	3	4	5	6	7
k	1	1	2	3	4	5	6	7
i	2	2	1	2	3	4	5	6
t	3	3	2	1	2	3	4	5
t	4	4	3	2	1	2	3	4
e	5	5	4	3	2	2	3	4
n	6	6	5	4	3	3	2	3

Edit distance = 3

Tiny Code (C)

```
#include <stdio.h>
#include <string.h>
#define MIN3(a,b,c) ((a<b)?((a<c)?a:c):((b<c)?b:c))

int edit_distance(char *A, char *B) {
    int n = strlen(A), m = strlen(B);
    int dp[n+1][m+1];
    for (int i=0;i<=n;i++) dp[i][0]=i;
    for (int j=0;j<=m;j++) dp[0][j]=j;
    for (int i=1;i<=n;i++)
        for (int j=1;j<=m;j++)
            if (A[i-1]==B[j-1])
                dp[i][j]=dp[i-1][j-1];
            else
                dp[i][j]=1+MIN3(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]);
    return dp[n][m];
}

int main() {
```

```

    printf("%d\n", edit_distance("kitten","sitting")); // 3
}

```

### 3. Space Optimization

We only need the previous row to compute the current row.

So,

Space complexity:  $O(\min(|A|, |B|))$

```

int edit_distance_opt(char *A, char *B) {
    int n=strlen(A), m=strlen(B);
    int prev[m+1], curr[m+1];
    for(int j=0;j<=m;j++) prev[j]=j;
    for(int i=1;i<=n;i++){
        curr[0]=i;
        for(int j=1;j<=m;j++){
            if(A[i-1]==B[j-1]) curr[j]=prev[j-1];
            else curr[j]=1+MIN3(prev[j], curr[j-1], prev[j-1]);
        }
        memcpy(prev,curr,sizeof(curr));
    }
    return prev[m];
}

```

### 4. Alignment

Alignment shows which characters correspond between two strings. Used in bioinformatics (e.g., DNA sequence alignment).

Each operation has a cost:

- Match: 0
- Mismatch: 1
- Gap (insert/delete): 1 We fill the DP table similarly, but track choices to trace back alignment.

## Example Alignment

A: kitten-  
B: sitt-ing

We can visualize the transformation path by backtracking dp table.

## Scoring Alignment (General Form)

We can generalize:

$$dp[i][j] = \min \{ dp[i - 1][j - 1] + cost(A_i, B_j), dp[i - 1][j] + gap, dp[i][j - 1] + gap \}$$

Used in Needleman-Wunsch (global alignment) and Smith-Waterman (local alignment).

## 5. Variants

- Damerau-Levenshtein: adds transposition (swap adjacent chars)- Hamming Distance: only substitutions, equal-length strings- Weighted Distance: different operation costs- Local Alignment: only best matching substrings

## 6. Summary

Method	Operations	Time	Use
Levenshtein	insert, delete, replace	(O(nm))	Spell check, fuzzy search
Hamming	substitution only	(O(n))	DNA, binary strings
Alignment (Needleman-Wunsch)	with scoring	(O(nm))	Bioinformatics
Local Alignment (Smith-Waterman)	best substring	(O(nm))	DNA regions

## Why It Matters

Edit distance transforms “difference” into data. It quantifies how far apart two strings are, enabling flexible, robust comparisons.

“Similarity isn’t perfection , it’s the cost of becoming alike.”

## Try It Yourself

1. Compute edit distance between “intention” and “execution”.
2. Trace back operations to show alignment.
3. Modify costs (insertion=2, deletion=1, substitution=2) and compare results.
4. Implement Hamming distance for equal-length strings.
5. Explore Smith-Waterman for longest common substring.

Once you master edit distance, you can build tools that understand typos, align genomes, and search imperfectly , perfectly.

## 66. Compression (Huffman, Arithmetic, LZ77, BWT)

Compression algorithms let us encode information efficiently, reducing storage or transmission cost without losing meaning. They turn patterns and redundancy into shorter representations , the essence of data compression.

This section introduces the key families of lossless compression algorithms that form the backbone of formats like ZIP, PNG, and GZIP.

We'll explore:

- Huffman Coding (prefix-free variable-length codes)
- Arithmetic Coding (fractional interval encoding)
- LZ77 / LZ78 (dictionary-based methods)
- Burrows-Wheeler Transform (BWT) (reversible sorting transform)

### 1. Huffman Coding

Huffman coding assigns shorter codes to frequent symbols, and longer codes to rare ones , achieving optimal compression among prefix-free codes.

#### A. Algorithm

1. Count frequencies of all symbols.
2. Build a min-heap of nodes (`symbol, freq`).
3. While heap size > 1:
  - Extract two smallest nodes `a, b`. - Create new node with `freq = a.freq + b.freq`.  
- Push back into heap.
4. Traverse tree , collect codes.

Each symbol gets a unique prefix code (no code is prefix of another).

## B. Example

Text: ABRACADABRA

Frequencies:

Symbol	Count
A	5
B	2
R	2
C	1
D	1

Building tree gives codes like:

A: 0  
B: 101  
R: 100  
C: 1110  
D: 1111

Encoded text: 0 101 100 0 1110 0 1111 0 101 100 0 Compression achieved!

Tiny Code (C, Sketch)

```
typedef struct Node {  
    char ch;  
    int freq;  
    struct Node *left, *right;  
} Node;
```

Use a min-heap (priority queue) to build the tree. Traverse recursively to print codewords.

Complexity: ( $O(n \log n)$ )

## 2. Arithmetic Coding

Instead of mapping symbols to bit strings, arithmetic coding maps the entire message to a single number in  $[0,1]$ .

We start with interval  $([0,1])$ , and iteratively narrow it based on symbol probabilities.

### **Example**

Symbols: {A: 0.5, B: 0.3, C: 0.2} Message: ABC

Intervals:

```
Start: [0, 1)
A → [0, 0.5)
B → [0.25, 0.4)
C → [0.34, 0.37)
```

Final code = any number in [0.34, 0.37) (e.g. 0.35)

Decoding reverses this process.

Advantage: achieves near-optimal entropy compression. Used in: JPEG2000, H.264

Time Complexity: ( O(n) )

### **3. LZ77 (Sliding Window Compression)**

LZ77 replaces repeated substrings with back-references (`offset`, `length`, `next_char`) pointing into a sliding window.

### **Example**

Text: abcabcabcx

Window slides; when `abc` repeats:

```
(0,0,'a'), (0,0,'b'), (0,0,'c'),
(3,3,'x') // "abc" repeats from 3 chars back
```

So sequence is compressed as references to earlier substrings.

Used in: DEFLATE (ZIP, GZIP), PNG

Time: ( O(n) ), Space: proportional to window size.

### **Tiny Code (Conceptual)**

```
struct Token { int offset, length; char next; };
```

Search previous window for longest match before emitting token.

#### 4. LZ78 (Dictionary-Based)

Instead of sliding window, LZ78 builds an explicit dictionary of substrings.

Algorithm:

- Start with empty dictionary.- Read input, find longest prefix in dictionary.- Output (`index`, `next_char`) and insert new entry. Example:

Input: ABAABABAABAB

Output: (0,A), (0,B), (1,B), (2,A), (4,A), (3,B)

Used in: LZW (GIF, TIFF)

#### 5. Burrows-Wheeler Transform (BWT)

BWT is not compression itself , it permutes text to cluster similar characters, making it more compressible by run-length or Huffman coding.

##### Steps

1. Generate all rotations of string.
2. Sort them lexicographically.
3. Take last column as output.

Example: banana\$

Rotations	Sorted
banana\$	<i>banana  ananab</i>
abanan nabana	
banana nanaba	

Last column: annb\$aa BWT("banana") = "annbaa"

Reversible with index of original row.

Used in: bzip2, FM-index (bioinformatics)

## 6. Summary

Algorithm	Idea	Complexity	Use
Huffman	Variable-length prefix codes	$(On \log n)$	General compression
Arithmetic	Interval encoding	$(O(n))$	Near-optimal entropy
LZ77	Sliding window matches	$(O(n))$	ZIP, PNG
LZ78	Dictionary building	$(O(n))$	GIF, TIFF
BWT	Permute for clustering	$(On \log n)$	bzip2

### Why It Matters

Compression algorithms reveal structure in data , they exploit patterns that humans can't see. They're also a window into information theory, showing how close we can get to the entropy limit.

“To compress is to understand , every bit saved is a pattern found.”

### Try It Yourself

1. Build a Huffman tree for MISSISSIPPI.
2. Implement a simple LZ77 encoder for repeating patterns.
3. Apply BWT and observe clustering of symbols.
4. Compare Huffman and Arithmetic outputs on same input.
5. Explore DEFLATE format combining LZ77 + Huffman.

Understanding compression means learning to see redundancy , the key to efficient storage, transmission, and understanding itself.

## 67. Cryptographic Hashes and Checksums

In algorithms, hashing helps us map data to fixed-size values. But when used for security and verification, hashing becomes a cryptographic tool. This section explores cryptographic hashes and checksums , algorithms that verify integrity, detect corruption, and secure data.

We'll look at:

- Simple checksums (parity, CRC)- Cryptographic hash functions (MD5, SHA family, BLAKE3)- Properties like collision resistance and preimage resistance- Practical uses in verification, signing, and storage

## 1. Checksums

Checksums are lightweight methods to detect accidental errors in data (not secure against attackers). They're used in filesystems, networking, and storage to verify integrity.

### A. Parity Bit

Adds one bit to make total 1s even or odd. Used in memory or communication to detect single-bit errors.

Example: Data = 1011 → has three 1s. Add parity bit 1 to make total 4 (even parity).

Limitation: Only detects odd number of bit errors.

### B. Modular Sum (Simple Checksum)

Sum all bytes (mod 256 or 65536).

Tiny Code (C)

```
uint8_t checksum(uint8_t *data, int n) {
    uint32_t sum = 0;
    for (int i = 0; i < n; i++) sum += data[i];
    return (uint8_t)(sum % 256);
}
```

Use: Simple file or packet validation.

### C. CRC (Cyclic Redundancy Check)

CRCs treat bits as coefficients of a polynomial. Divide by a generator polynomial, remainder = CRC code.

Used in Ethernet, ZIP, and PNG.

Example: CRC-32, CRC-16.

Fast hardware and table-driven implementations available.

Key Property:

- Detects most burst errors- Not cryptographically secure

## 2. Cryptographic Hash Functions

A cryptographic hash function (  $h(x)$  ) maps any input to a fixed-size output such that:

1. Deterministic: same input → same output
2. Fast computation
3. Preimage resistance: hard to find (  $x$  ) given (  $h(x)$  )
4. Second-preimage resistance: hard to find  $x' \neq x$  with (  $h(x') = h(x)$  )
5. Collision resistance: hard to find any two distinct inputs with same hash

Algorithm	Output (bits)	Notes
MD5	128	Broken (collisions found)
SHA-1	160	Deprecated
SHA-256	256	Standard (SHA-2 family)
SHA-3	256	Keccak-based sponge
BLAKE3	256	Fast, parallel, modern

### Example

```
h("hello") = 2cf24dba5fb0a... (SHA-256)
```

Change one letter, hash changes completely (avalanche effect):

```
h("Hello") = 185f8db32271f...
```

Even small changes → big differences.

### Tiny Code (C, using pseudo-interface)

```
#include <openssl/sha.h>

unsigned char hash[SHA256_DIGEST_LENGTH];
SHA256((unsigned char*)"hello", 5, hash);
```

Print hash as hex string to verify.

### 3. Applications

- Data integrity: verify files (e.g., SHA256SUM)- Digital signatures: sign hashes, not raw data- Password storage: store hashes, not plaintext- Deduplication: detect identical files via hashes- Blockchain: link blocks with hash pointers- Git: stores objects via SHA-1 identifiers

### 4. Hash Collisions

A collision occurs when (  $h(x) = h(y)$  ) for  $x \neq y$ . Good cryptographic hashes make this computationally infeasible.

By the birthday paradox, collisions appear after  $2^{n/2}$  operations for an ( n )-bit hash.

Hence, SHA-256  $\rightarrow \sim 2^{128}$  effort to collide.

### 5. Checksums vs Hashes

Feature	Checksum	Cryptographic Hash
Goal	Detect errors	Ensure integrity and authenticity
Resistance	Low	High
Output Size	Small	128-512 bits
Performance	Very fast	Fast but secure
Example	CRC32	SHA-256, BLAKE3

### Why It Matters

Checksums catch accidental corruption, hashes protect against malicious tampering. Together, they guard the trustworthiness of data , the foundation of secure systems.

“Integrity is invisible , until it’s lost.”

### Try It Yourself

1. Compute CRC32 of a text file, flip one bit, and recompute.
2. Use `sha256sum` to verify file integrity.
3. Experiment: change one character in input, observe avalanche.
4. Compare performance of SHA-256 and BLAKE3.
5. Research how Git uses SHA-1 to track versions.

By learning hashes, you master one of the pillars of security , proof that something hasn't changed, even when everything else does.

## 68. Approximate and Streaming Matching

Exact string matching (like KMP or Boyer-Moore) demands perfect alignment between pattern and text. But what if errors, noise, or incomplete data exist?

That's where approximate matching and streaming matching come in. These algorithms let you search efficiently even when:

- The pattern might contain typos or mutations- The text arrives in a stream (too large to store entirely)- You want to match “close enough,” not “exactly” They’re crucial in search engines, spell checkers, bioinformatics, and real-time monitoring systems.

### 1. Approximate String Matching

Approximate string matching finds occurrences of a pattern in a text allowing mismatches, insertions, or deletions , often measured by edit distance.

#### A. Dynamic Programming (Levenshtein Distance)

Given two strings  $A$  and  $B$ , the edit distance is the minimum number of insertions, deletions, or substitutions to turn  $A$  into  $B$ .

We can build a DP table  $dp[i][j]$ :

- $dp[i][0] = i$  (delete all characters)
- $dp[0][j] = j$  (insert all characters)
- If  $A[i] = B[j]$ , then  $dp[i][j] = dp[i - 1][j - 1]$
- Else  $dp[i][j] = 1 + \min(dp[i - 1][j], dp[i][j - 1], dp[i - 1][j - 1])$

Tiny Code (C)

```
int edit_distance(char *a, char *b) {
    int n = strlen(a), m = strlen(b);
    int dp[n+1][m+1];
    for (int i = 0; i <= n; i++) dp[i][0] = i;
    for (int j = 0; j <= m; j++) dp[0][j] = j;
```

```

for (int i = 1; i <= n; i++)
    for (int j = 1; j <= m; j++)
        if (a[i-1] == b[j-1]) dp[i][j] = dp[i-1][j-1];
        else dp[i][j] = 1 + fmin(fmin(dp[i-1][j], dp[i][j-1]), dp[i-1][j-1]);
return dp[n][m];
}

```

This computes Levenshtein distance in ( O(nm) ) time.

### B. Bitap Algorithm (Shift-Or)

When pattern length is small, Bitap uses bitmasks to track mismatches. It efficiently supports up to k errors and runs in near linear time for small patterns.

Used in grep -E, ag, and fuzzy searching systems.

Idea: Maintain a bitmask where 1 = mismatch, 0 = match. Shift and OR masks as we scan text.

### C. k-Approximate Matching

Find all positions where edit distance  $\leq k$ . Efficient for small (  $k$  ) (e.g., spell correction: edit distance 2).

Applications:

- Typo-tolerant search- DNA sequence matching- Autocomplete systems

## 2. Streaming Matching

In streaming, the text is too large or unbounded, so we must process input online. We can't store everything , only summaries or sketches.

### A. Rolling Hash (Rabin-Karp style)

Maintains a moving hash of recent characters. When new character arrives, update hash in ( $O(1)$ ). Compare with pattern's hash for possible match.

Good for sliding window matching.

Example:

```
hash = (base * (hash - old_char * base^(m-1)) + new_char) % mod;
```

### B. Fingerprinting (Karp-Rabin Fingerprint)

A compact representation of a substring. If fingerprints match, do full verification (avoid false positives). Used in streaming algorithms and chunking.

### C. Sketch-Based Matching

Algorithms like Count-Min Sketch or SimHash build summaries of large data. They help approximate similarity between streams.

Applications:

- Near-duplicate detection (SimHash in Google)- Network anomaly detection- Real-time log matching

## 3. Approximate Matching in Practice

Domain	Use Case	Algorithm
Spell Checking	“recieve” → “receive”	Edit Distance
DNA Alignment	Find similar sequences	Smith-Waterman
Autocomplete	Suggest close matches	Fuzzy Search
Logs & Streams	Online pattern alerts	Streaming Bitap, Karp-Rabin
Near-Duplicate	Detect similar text	SimHash, MinHash

Algorithm	Time	Space	Notes
-----------	------	-------	-------

#### 4. Complexity

Algorithm	Time	Space	Notes
Levenshtein DP	(O(nm))	(O(nm))	Exact distance
Bitap	(O(n))	(O(1))	For small patterns
Rolling Hash	(O(n))	(O(1))	Probabilistic match
SimHash	(O(n))	(O(1))	Approximate similarity

#### Why It Matters

Real-world data is messy , typos, noise, loss, corruption. Approximate matching lets you build algorithms that forgive errors and adapt to streams. It powers everything from search engines to genomics, ensuring your algorithms stay practical in an imperfect world.

#### Try It Yourself

1. Compute edit distance between “kitten” and “sitting.”
2. Implement fuzzy search that returns words with 1 typo.
3. Use rolling hash to detect repeated substrings in a stream.
4. Experiment with SimHash to compare document similarity.
5. Observe how small typos affect fuzzy vs exact search.

### 69. Bioinformatics Alignment (Needleman-Wunsch, Smith-Waterman)

In bioinformatics, comparing DNA, RNA, or protein sequences is like comparing strings , but with biological meaning. Each sequence is made of letters (A, C, G, T for DNA; amino acids for proteins). To analyze similarity, scientists use sequence alignment algorithms that handle insertions, deletions, and substitutions.

Two fundamental methods dominate:

- Needleman-Wunsch for global alignment- Smith-Waterman for local alignment

## 1. Sequence Alignment

Alignment means placing two sequences side by side to maximize matches and minimize gaps or mismatches.

For example:

```
A C G T G A
| | |   |
A C G A G A
```

Here, mismatches and gaps may occur, but the alignment finds the best possible match under a scoring system.

### Scoring System

Alignment uses scores instead of just counts. Typical scheme:

- Match: +1- Mismatch: -1- Gap (insertion or deletion): -2 You can adjust weights depending on the biological context.

## 2. Needleman-Wunsch (Global Alignment)

Used when you want to align entire sequences , from start to end.

It uses dynamic programming to build a score table (  $dp[i][j]$  ), where each cell represents the best score for aligning prefixes (  $A[1..i]$  ) and (  $B[1..j]$  ).

Recurrence:

$$dp[i][j] = \max \left\{ dp[i-1][j-1] + \text{score}(A_i, B_j), dp[i-1][j] + \text{gap penalty}, dp[i][j-1] + \text{gap penalty} \right\}$$

Base cases:

$$dp[0][j] = j \times \text{gap penalty}, \quad dp[i][0] = i \times \text{gap penalty}$$

Tiny Code (C)

```

int max3(int a, int b, int c) {
    return a > b ? (a > c ? a : c) : (b > c ? b : c);
}

int needleman_wunsch(char *A, char *B, int match, int mismatch, int gap) {
    int n = strlen(A), m = strlen(B);
    int dp[n+1][m+1];
    for (int i = 0; i <= n; i++) dp[i][0] = i * gap;
    for (int j = 0; j <= m; j++) dp[0][j] = j * gap;

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            int s = (A[i-1] == B[j-1]) ? match : mismatch;
            dp[i][j] = max3(dp[i-1][j-1] + s, dp[i-1][j] + gap, dp[i][j-1] + gap);
        }
    }
    return dp[n][m];
}

```

Example:

```

A = "ACGT"
B = "AGT"
match = +1, mismatch = -1, gap = -2

```

Produces optimal alignment:

```

A C G T
A - G T

```

### 3. Smith-Waterman (Local Alignment)

Used when sequences may have similar segments, not full-length similarity. Perfect for finding motifs or conserved regions.

Recurrence is similar, but with local reset to zero:

$$dp[i][j] = \max \{0, dp[i-1][j-1] + \text{score}(A_i, B_j), dp[i-1][j] + \text{gap penalty}, dp[i][j-1] + \text{gap penalty}\}$$

Final answer = maximum value in the table (not necessarily at the end).

It finds the best substring alignment.

## **Example**

```
A = "ACGTTG"  
B = "CGT"
```

Smith-Waterman finds best local match:

```
A C G T  
| | |  
C G T
```

Unlike global alignment, extra prefixes or suffixes are ignored.

## **4. Variants and Extensions**

Algorithm	Type	Notes
Needleman-Wunsch	Global	Aligns full sequences
Smith-Waterman	Local	Finds similar subsequences
Gotoh Algorithm	Global	Uses affine gap penalty (opening + extension)
BLAST	Heuristic	Speeds up search for large databases

BLAST (Basic Local Alignment Search Tool) uses word seeds and extension, trading exactness for speed , essential for large genome databases.

## **5. Complexity**

Both Needleman-Wunsch and Smith-Waterman run in:

- Time: ( O(nm) )- Space: ( O(nm) ) But optimized versions use banded DP or Hirschberg's algorithm to cut memory to ( O(n + m) ).

## **Why It Matters**

Sequence alignment bridges computer science and biology. It's how we:

- Compare species- Identify genes- Detect mutations- Trace ancestry- Build phylogenetic trees The idea of “minimum edit cost” echoes everywhere , from spell checkers to DNA analysis.

“In biology, similarity is a story. Alignment is how we read it.”

## Try It Yourself

1. Implement Needleman-Wunsch for short DNA sequences.
2. Change gap penalties , see how alignment shifts.
3. Compare outputs from global and local alignment.
4. Use real sequences from GenBank to test.
5. Explore BLAST online and compare to exact alignment results.

## 70. Text Indexing and Search Structures

When text becomes large , think books, databases, or the entire web , searching naively for patterns ( $O(nm)$ ) is far too slow. We need indexing structures that let us search fast, often in  $O(m)$  or  $O(\log n)$  time.

This section covers the backbone of search engines and string processing:

- Suffix Arrays- Suffix Trees- Inverted Indexes- Tries and Prefix Trees- Compressed Indexes like FM-Index (Burrows-Wheeler)

### 1. Why Index?

A text index is like a table of contents , it doesn't store the book, but lets you jump straight to what you want.

If you have a text of length (  $n$  ), and you'll run many queries, it's worth building an index (even if it costs (  $O(n \log n)$  ) to build).

Without indexing: each query takes (  $O(nm)$  ). With indexing: each query can take (  $O(m)$  ) or less.

### 2. Suffix Array

A suffix array is a sorted array of all suffixes of a string.

For text "banana", suffixes are:

```
0: banana
1: anana
2: nana
3: ana
4: na
5: a
```

Sorted lexicographically:

```
5: a
3: ana
1: anana
0: banana
4: na
2: nana
```

Suffix Array = [5, 3, 1, 0, 4, 2]

To search, binary search over the suffix array using your pattern , (  $O(m \log n)$  ).

Tiny Code (C) (naive construction)

```
int cmp(const void *a, const void *b, void *txt) {
    int i = *(int*)a, j = *(int*)b;
    return strcmp((char*)txt + i, (char*)txt + j);
}

void build_suffix_array(char *txt, int n, int sa[]) {
    for (int i = 0; i < n; i++) sa[i] = i;
    qsort_r(sa, n, sizeof(int), cmp, txt);
}
```

Modern methods like prefix doubling or radix sort build it in (  $O(n \log n)$  ).

Applications:

- Fast substring search- Longest common prefix (LCP) array- Pattern matching in DNA sequences- Plagiarism detection

### 3. Suffix Tree

A suffix tree is a compressed trie of all suffixes , each edge stores multiple characters.

For "banana", you'd build a tree where each leaf corresponds to a suffix index.

Advantages:

- Pattern search in (  $O(m)$  )- Space (  $O(n)$  ) (with compression) Built using Ukkonen's algorithm in (  $O(n)$  ).

Use Suffix Array + LCP as a space-efficient alternative.

#### **4. FM-Index (Burrows-Wheeler Transform)**

Used in compressed full-text search (e.g., Bowtie, BWA). Combines:

- Burrows-Wheeler Transform (BWT)- Rank/select bitvectors Supports pattern search in  $O(m)$  time with very low memory.

Idea: transform text so similar substrings cluster together, enabling compression and backward search.

Applications:

- DNA alignment- Large text archives- Memory-constrained search

#### **5. Inverted Index**

Used in search engines. Instead of suffixes, it indexes words.

For example, text corpus:

```
doc1: quick brown fox  
doc2: quick red fox
```

Inverted index:

```
"quick" → [doc1, doc2]  
"brown" → [doc1]  
"red" → [doc2]  
"fox" → [doc1, doc2]
```

Now searching “quick fox” becomes set intersection of lists.

Used with ranking functions (TF-IDF, BM25).

## 6. Tries and Prefix Trees

A trie stores strings character by character. Each node = prefix.

```
typedef struct Node {
    struct Node *child[26];
    int end;
} Node;
```

Perfect for:

- Autocomplete- Prefix search- Spell checkers Search:  $O(m)$ , where  $m$  = pattern length.

Compressed tries (Patricia trees) reduce space.

## 7. Comparing Structures

Structure	Search Time	Build Time	Space	Notes
Trie	$O(m)$	$O(n)$	High	Prefix queries
Suffix Array	$O(m \log n)$	$O(n \log n)$	Medium	Sorted suffixes
Suffix Tree	$O(m)$	$O(n)$	High	Rich structure
FM-Index	$O(m)$	$O(n)$	Low	Compressed
Inverted Index	$O(k)$	$O(N)$	Medium	Word-based

### Why It Matters

Text indexing is the backbone of search engines, DNA alignment, and autocomplete systems. Without it, Google searches, code lookups, or genome scans would take minutes, not milliseconds.

“Indexing turns oceans of text into navigable maps.”

### Try It Yourself

1. Build a suffix array for “banana” and perform binary search for “ana.”
2. Construct a trie for a dictionary and query prefixes.
3. Write a simple inverted index for a few documents.
4. Compare memory usage of suffix tree vs suffix array.
5. Experiment with FM-index using an online demo (like BWT explorer).

## Chapter 8. Geometry, Graphics, and Spatial Algorithms

### 71. Convex Hull (Graham, Andrew, Chan)

In computational geometry, the convex hull of a set of points is the smallest convex polygon that contains all the points. Intuitively, imagine stretching a rubber band around a set of nails on a board , the shape the band takes is the convex hull.

Convex hulls are foundational for many geometric algorithms, like closest pair, Voronoi diagrams, and collision detection.

In this section, we'll explore three classical algorithms:

- Graham Scan - elegant and simple ( $O(n \log n)$ )- Andrew's Monotone Chain - robust and practical ( $O(n \log n)$ )- Chan's Algorithm - advanced and optimal ( $O(n \log h)$ , where  $h =$  number of hull points)

#### 1. Definition

Given a set of points  $P = p_1, p_2, \dots, p_n$ , the convex hull, (  $\text{CH}(P)$  ), is the smallest convex polygon enclosing all points.

Formally:

$$\text{CH}(P) = \bigcap C \subseteq \mathbb{R}^2 \mid C \text{ is convex and } P \subseteq C$$

A polygon is convex if every line segment between two points of the polygon lies entirely inside it.

#### 2. Orientation Test

All convex hull algorithms rely on an orientation test using cross product: Given three points ( a, b, c ):

$$\text{cross}(a, b, c) = (b_x - a_x)(c_y - a_y) - (b_y - a_y)(c_x - a_x)$$

- $> 0 \rightarrow$  counter-clockwise turn-  $< 0 \rightarrow$  clockwise turn-  $= 0 \rightarrow$  collinear

### 3. Graham Scan

One of the earliest convex hull algorithms.

Idea:

1. Pick the lowest point (and leftmost if tie).
2. Sort all other points by polar angle with respect to it.
3. Traverse points and maintain a stack:
  - Add point - While last three points make a right turn, pop middle one4. Remaining points form convex hull in CCW order.

Tiny Code (C)

```
typedef struct { double x, y; } Point;

double cross(Point a, Point b, Point c) {
    return (b.x - a.x)*(c.y - a.y) - (b.y - a.y)*(c.x - a.x);
}

int cmp(const void *p1, const void *p2) {
    Point *a = (Point*)p1, *b = (Point*)p2;
    // Compare by polar angle or distance
    return (a->y != b->y) ? (a->y - b->y) : (a->x - b->x);
}

int graham_scan(Point pts[], int n, Point hull[]) {
    qsort(pts, n, sizeof(Point), cmp);
    int top = 0;
    for (int i = 0; i < n; i++) {
        while (top >= 2 && cross(hull[top-2], hull[top-1], pts[i]) <= 0)
            top--;
        hull[top++] = pts[i];
    }
    return top; // number of hull points
}
```

Complexity:

- Sorting: (  $O(n \log n)$  )- Scanning: (  $O(n)$  )  $\rightarrow$  Total:  $O(n \log n)$

## Example

Input:

(0, 0), (1, 1), (2, 2), (2, 0), (0, 2)

Hull (CCW):

(0,0) → (2,0) → (2,2) → (0,2)

## 4. Andrew's Monotone Chain

Simpler and more robust for floating-point coordinates. Builds lower and upper hulls separately.

Steps:

1. Sort points lexicographically (x, then y).
2. Build lower hull (left-to-right)
3. Build upper hull (right-to-left)
4. Concatenate (excluding duplicates)

Tiny Code (C)

```
int monotone_chain(Point pts[], int n, Point hull[]) {
    qsort(pts, n, sizeof(Point), cmp);
    int k = 0;
    // Lower hull
    for (int i = 0; i < n; i++) {
        while (k >= 2 && cross(hull[k-2], hull[k-1], pts[i]) <= 0) k--;
        hull[k++] = pts[i];
    }
    // Upper hull
    for (int i = n-2, t = k+1; i >= 0; i--) {
        while (k >= t && cross(hull[k-2], hull[k-1], pts[i]) <= 0) k--;
        hull[k++] = pts[i];
    }
    return k-1; // last point == first point
}
```

Time Complexity: (  $O(n \log n)$  )

## 5. Chan's Algorithm

When  $h \ll n$ , Chan's method achieves (  $O(n \log h)$  ):

1. Partition points into groups of size (  $m$  ).
2. Compute hulls for each group (Graham).
3. Merge hulls with Jarvis March (gift wrapping).
4. Choose (  $m$  ) cleverly ( $m = 2^k$ ) to ensure (  $O(n \log h)$  ).

Used in: large-scale geometric processing.

## 6. Applications

Domain	Use
Computer Graphics	Shape boundary, hitboxes
GIS / Mapping	Region boundaries
Robotics	Obstacle envelopes
Clustering	Outlier detection
Data Analysis	Minimal bounding shape

## 7. Complexity Summary

Algorithm	Time	Space	Notes
Graham Scan	( $O(n \log n)$ )	( $O(n)$ )	Simple, classic
Monotone Chain	( $O(n \log n)$ )	( $O(n)$ )	Stable, robust
Chan's Algorithm	( $O(n \log h)$ )	( $O(n)$ )	Best asymptotic

## Why It Matters

Convex hulls are one of the cornerstones of computational geometry. They teach sorting, cross products, and geometric reasoning , and form the basis for many spatial algorithms.

“Every scattered set hides a simple shape. The convex hull is that hidden simplicity.”

## Try It Yourself

1. Implement Graham Scan for 10 random points.
2. Plot the points and verify the hull.
3. Compare results with Andrew's Monotone Chain.
4. Test with collinear and duplicate points.
5. Explore 3D convex hulls (QuickHull, Gift Wrapping) next.

## 72. Closest Pair and Segment Intersection

Geometric problems often ask: *what's the shortest distance between two points?* or *do these segments cross?* These are classic building blocks in computational geometry , essential for collision detection, graphics, clustering, and path planning.

This section covers two foundational problems:

- Closest Pair of Points - find two points with minimum Euclidean distance- Segment Intersection - determine if (and where) two line segments intersect

### 1. Closest Pair of Points

Given ( n ) points in 2D, find a pair with the smallest distance. The brute force solution is (  $O(n^2)$  ), but using Divide and Conquer, we can solve it in  $O(n \log n)$ .

#### A. Divide and Conquer Algorithm

Idea:

1. Sort points by x-coordinate.
2. Split into left and right halves.
3. Recursively find closest pairs in each half (distance = ( d )).
4. Merge step: check pairs across the split line within ( d ).

In merge step, we only need to check at most 6 neighbors per point (by geometric packing).

Tiny Code (C, Sketch)

```
#include <math.h>
typedef struct { double x, y; } Point;

double dist(Point a, Point b) {
    double dx = a.x - b.x, dy = a.y - b.y;
```

```

    return sqrt(dx*dx + dy*dy);
}

double brute_force(Point pts[], int n) {
    double d = 1e9;
    for (int i = 0; i < n; i++)
        for (int j = i + 1; j < n; j++)
            d = fmin(d, dist(pts[i], pts[j]));
    return d;
}

```

Recursive divide and merge:

```

double closest_pair(Point pts[], int n) {
    if (n <= 3) return brute_force(pts, n);
    int mid = n / 2;
    double d = fmin(closest_pair(pts, mid),
                    closest_pair(pts + mid, n - mid));
    // merge step: check strip points within distance d
    // sort by y, check neighbors
    return d;
}

```

Time Complexity: (  $O(n \log n)$  )

Example:

Points:

(2,3), (12,30), (40,50), (5,1), (12,10), (3,4)

Closest pair: (2,3) and (3,4), distance =  $\sqrt{2}$

## B. Sweep Line Variant

Another method uses a line sweep and a balanced tree to keep active points. As you move from left to right, maintain a window of recent points within (  $d$  ).

Used in large-scale spatial systems.

## Applications

Domain	Use
Clustering	Find nearest neighbors
Robotics	Avoid collisions
GIS	Nearest city search
Networking	Sensor proximity

## 2. Segment Intersection

Given two segments ( AB ) and ( CD ), determine whether they intersect. It's the core of geometry engines and vector graphics systems.

### A. Orientation Test

We use the cross product (orientation) test again. Two segments ( AB ) and ( CD ) intersect if and only if:

1. The segments straddle each other:

$$\text{orient}(A, B, C) \neq \text{orient}(A, B, D)$$

$$\text{orient}(C, D, A) \neq \text{orient}(C, D, B)$$

2. Special cases for collinear points (check bounding boxes).

Tiny Code (C)

```
double cross(Point a, Point b, Point c) {
    return (b.x - a.x)*(c.y - a.y) - (b.y - a.y)*(c.x - a.x);
}

int on_segment(Point a, Point b, Point c) {
    return fmin(a.x, b.x) <= c.x && c.x <= fmax(a.x, b.x) &&
           fmin(a.y, b.y) <= c.y && c.y <= fmax(a.y, b.y);
}

int intersect(Point a, Point b, Point c, Point d) {
    double o1 = cross(a, b, c);
    double o2 = cross(a, b, d);
```

```

    double o3 = cross(c, d, a);
    double o4 = cross(c, d, b);
    if (o1*o2 < 0 && o3*o4 < 0) return 1; // general case
    if (o1 == 0 && on_segment(a,b,c)) return 1;
    if (o2 == 0 && on_segment(a,b,d)) return 1;
    if (o3 == 0 && on_segment(c,d,a)) return 1;
    if (o4 == 0 && on_segment(c,d,b)) return 1;
    return 0;
}

```

## B. Line Sweep Algorithm (Bentley-Ottmann)

For multiple segments, check all intersections efficiently. Algorithm:

1. Sort all endpoints by x-coordinate.
2. Sweep from left to right.
3. Maintain active set (balanced BST).
4. Check neighboring segments for intersections.

Time complexity:  $O((n + k) \log n)$ , where  $k$  is the number of intersections.

Used in CAD, map rendering, and collision systems.

## 3. Complexity Summary

Problem	Naive	Optimal	Technique
Closest Pair	$O(n^2)$	$O(n \log n)$	Divide & Conquer
Segment Intersection	$O(n^2)$	$O((n + k) \log n)$	Sweep Line

## Why It Matters

Geometric algorithms like these teach how to reason spatially , blending math, sorting, and logic. They power real-world systems where precision matters: from self-driving cars to game engines.

“Every point has a neighbor; every path may cross another , geometry finds the truth in space.”

## Try It Yourself

1. Implement the closest pair algorithm using divide and conquer.
2. Visualize all pairwise distances , see which pairs are minimal.
3. Test segment intersection on random pairs.
4. Modify for 3D line segments using vector cross products.
5. Try building a line sweep visualizer to catch intersections step-by-step.

## 73. Line Sweep and Plane Sweep Algorithms

The sweep line (or plane sweep) technique is one of the most powerful paradigms in computational geometry. It transforms complex spatial problems into manageable one-dimensional events , by sweeping a line (or plane) across the input and maintaining a dynamic set of active elements.

This method underlies many geometric algorithms:

- Event sorting → handle things in order- Active set maintenance → track current structure- Updates and queries → respond as the sweep progresses Used for intersection detection, closest pair, rectangle union, computational geometry in graphics and GIS.

### 1. The Core Idea

Imagine a vertical line sweeping from left to right across the plane. At each “event” (like a point or segment endpoint), we update the set of objects the line currently touches , the active set.

Each event may trigger queries, insertions, or removals.

This approach works because geometry problems often depend only on local relationships between nearby elements as the sweep advances.

### A. Sweep Line Template

A general structure looks like this:

```
struct Event { double x; int type; Object *obj; };
sort(events.begin(), events.end());

ActiveSet S;

for (Event e : events) {
```

```

    if (e.type == START) S.insert(e.obj);
    else if (e.type == END) S.erase(e.obj);
    else if (e.type == QUERY) handle_query(S, e.obj);
}

```

Sorting ensures events are processed in order of increasing x (or another dimension).

## 2. Classic Applications

Let's explore three foundational problems solvable by sweep techniques.

### A. Segment Intersection (Bentley-Ottmann)

Goal: detect all intersections among ( n ) line segments.

Steps:

1. Sort endpoints by x-coordinate.
2. Sweep from left to right.
3. Maintain an ordered set of active segments (sorted by y).
4. When a new segment starts, check intersection with neighbors above and below.
5. When segments intersect, record intersection and insert a new event at the x-coordinate of intersection.

Complexity:  $O((n + k) \log n)$ , where  $k$  is the number of intersections.

### B. Closest Pair of Points

Sweep line version sorts by x, then slides a vertical line while maintaining active points within a strip of width ( d ) (current minimum). Only need to check at most 6-8 nearby points in strip.

Complexity: (  $O(n \log n)$  )

### C. Rectangle Union Area

Given axis-aligned rectangles, compute total area covered.

Idea:

- Treat vertical edges as events (entering/exiting rectangles).- Sweep line moves along x-axis.- Maintain y-intervals in active set (using a segment tree or interval tree).- At each step, multiply current width  $\times$  height of union of active intervals. Complexity: (  $O(n \log n)$  )

Tiny Code Sketch (C)

```

typedef struct { double x, y1, y2; int type; } Event;
Event events[MAX];
int n_events;

qsort(events, n_events, sizeof(Event), cmp_by_x);

double prev_x = events[0].x, area = 0;
SegmentTree T;

for (int i = 0; i < n_events; i++) {
    double dx = events[i].x - prev_x;
    area += dx * T.total_length(); // current union height
    if (events[i].type == START)
        T.insert(events[i].y1, events[i].y2);
    else
        T.remove(events[i].y1, events[i].y2);
    prev_x = events[i].x;
}

```

### 3. Other Applications

Problem	Description	Time
K-closest points	Maintain top $k$ in active set	$O(n \log n)$
Union of rectangles	Compute covered area	$O(n \log n)$
Point location	Locate point in planar subdivision	$O(\log n)$
Visibility graph	Track visible edges	$O(n \log n)$

## 4. Plane Sweep Extensions

While line sweep moves in one dimension (x), plane sweep handles 2D or higher-dimensional spaces, where:

- Events are 2D cells or regions.- Sweep front is a plane instead of a line. Used in 3D collision detection, computational topology, and CAD systems.

### Conceptual Visualization

1. Sort events by one axis (say, x).
2. Maintain structure (set, tree, or heap) of intersecting or active elements.
3. Update at each event and record desired output (intersection, union, coverage).

The key is the locality principle: only neighbors in the sweep structure can change outcomes.

## 5. Complexity

Phase	Complexity
Sorting events	$O(n \log n)$
Processing events	$O(n \log n)$
Total	$O(n \log n)$ (typical)

### Why It Matters

The sweep line method transforms geometric chaos into order , turning spatial relationships into sorted sequences. It's the bridge between geometry and algorithms, blending structure with motion.

“A sweep line sees everything , not all at once, but just in time.”

### Try It Yourself

1. Implement a sweep-line segment intersection finder.
2. Compute the union area of 3 rectangles with overlaps.
3. Animate the sweep line to visualize event processing.
4. Modify for circular or polygonal objects.
5. Explore how sweep-line logic applies to time-based events in scheduling.

## 74. Delaunay and Voronoi Diagrams

In geometry and spatial computing, Delaunay triangulations and Voronoi diagrams are duals , elegant structures that capture proximity, territory, and connectivity among points.

They're used everywhere: from mesh generation, pathfinding, geospatial analysis, to computational biology. This section introduces both, their relationship, and algorithms to construct them efficiently.

### 1. Voronoi Diagram

Given a set of sites (points)  $P = p_1, p_2, \dots, p_n$ , the Voronoi diagram partitions the plane into regions , one per point , so that every location in a region is closer to its site than to any other.

Formally, the Voronoi cell for  $p_i$  is:

$$V(p_i) = x \in \mathbb{R}^2 \mid d(x, p_i) \leq d(x, p_j), \forall j \neq i$$

Each region is convex, and boundaries are formed by perpendicular bisectors.

#### Example

For points ( A, B, C ):

- Draw bisectors between each pair.- Intersection points define Voronoi vertices.- Resulting polygons cover the plane, one per site. Used to model nearest neighbor regions , “which tower serves which area?”

#### Properties

- Every cell is convex.- Neighboring cells share edges.- The diagram's vertices are centers of circumcircles through three sites.- Dual graph = Delaunay triangulation.

### 2. Delaunay Triangulation

The Delaunay triangulation (DT) connects points so that no point lies inside the circumcircle of any triangle.

Equivalently, it's the dual graph of the Voronoi diagram.

It tends to avoid skinny triangles , maximizing minimum angles, creating well-shaped meshes.

## Formal Definition

A triangulation ( $T$ ) of ( $P$ ) is Delaunay if for every triangle  $\triangle abc \in T$ , no point  $p \in P \setminus \{a, b, c\}$  lies inside the circumcircle of  $\triangle abc$ .

Why It Matters:

- Avoids sliver triangles.- Used in finite element meshes, terrain modeling, and path planning.- Leads to natural neighbor interpolation and smooth surfaces.

## 3. Relationship

Voronoi and Delaunay are geometric duals:

Voronoi	Delaunay
Regions = proximity zones	Triangles = neighbor connections
Edges = bisectors	Edges = neighbor pairs
Vertices = circumcenters	Faces = circumcircles

Connecting neighboring Voronoi cells gives Delaunay edges.

## 4. Algorithms

Several algorithms can build these diagrams efficiently.

### A. Incremental Insertion

1. Start with a super-triangle enclosing all points.
2. Insert points one by one.
3. Remove triangles whose circumcircle contains the point.
4. Re-triangulate the resulting polygonal hole.

Time Complexity: ( $O(n^2)$ ), improved to ( $O(n \log n)$ ) with randomization.

### B. Divide and Conquer

1. Sort points by x.
2. Recursively build DT for left and right halves.
3. Merge by finding common tangents.

Time Complexity: ( $O(n \log n)$ ) Elegant, structured, and deterministic.

### C. Fortune's Sweep Line Algorithm

For Voronoi diagrams, Fortune's algorithm sweeps a line from top to bottom. Maintains a beach line of parabolic arcs and event queue.

Each event (site or circle) updates the structure , building Voronoi edges incrementally.

Time Complexity: ( On  $\log n$  )

### D. Bowyer-Watson (Delaunay via Circumcircle Test)

A practical incremental version widely used in graphics and simulation.

Steps:

- Start with supertriangle- Insert point- Remove all triangles whose circumcircle contains point- Reconnect the resulting cavity

Tiny Code (Conceptual)

```
typedef struct { double x, y; } Point;

typedef struct { Point a, b, c; } Triangle;

bool in_circle(Point a, Point b, Point c, Point p) {
    double A[3][3] = {
        {a.x - p.x, a.y - p.y, (a.x*a.x + a.y*a.y) - (p.x*p.x + p.y*p.y)},
        {b.x - p.x, b.y - p.y, (b.x*b.x + b.y*b.y) - (p.x*p.x + p.y*p.y)},
        {c.x - p.x, c.y - p.y, (c.x*c.x + c.y*c.y) - (p.x*p.x + p.y*p.y)}
    };
    return determinant(A) > 0;
}
```

This test ensures Delaunay property.

## 5. Applications

Domain	Application
GIS	Nearest facility, region partition
Mesh Generation	Finite element methods
Robotics	Visibility graphs, navigation
Computer Graphics	Terrain triangulation

Domain	Application
Clustering	Spatial neighbor structure

## 6. Complexity Summary

Algorithm	Type	Time	Notes
Fortune	Voronoi	( On $\log n$ )	Sweep line
Bowyer-Watson	Delaunay	( On $\log n$ )	Incremental
Divide & Conquer	Delaunay	( On $\log n$ )	Recursive

## Why It Matters

Voronoi and Delaunay diagrams reveal natural structure in point sets. They convert distance into geometry, showing how space is divided and connected. If geometry is the shape of space, these diagrams are its skeleton.

“Every point claims its territory; every territory shapes its network.”

## Try It Yourself

1. Draw Voronoi regions for 5 random points by hand.
2. Build Delaunay triangles (connect neighboring sites).
3. Verify the empty circumcircle property.
4. Use a library (CGAL / SciPy) to visualize both structures.
5. Explore how adding new points reshapes the diagrams.

## 75. Point in Polygon and Polygon Triangulation

Geometry often asks two fundamental questions:

1. Is a point inside or outside a polygon?
2. How can a complex polygon be broken into triangles for computation?

These are the building blocks of spatial analysis, computer graphics, and computational geometry.

## 1. Point in Polygon (PIP)

Given a polygon defined by vertices ( $x_1, y_1, x_2, y_2, \dots, x_n, y_n$ ) and a test point (( $x, y$ )), we want to determine if the point lies inside, on the boundary, or outside the polygon.

### Methods

#### A. Ray Casting Algorithm

Shoot a ray horizontally to the right of the point. Count how many times it intersects polygon edges.

- Odd count → Inside- Even count → Outside This is based on the even-odd rule.

Tiny Code (Ray Casting in C)

```
bool point_in_polygon(Point p, Point poly[], int n) {
    bool inside = false;
    for (int i = 0, j = n - 1; i < n; j = i++) {
        if (((poly[i].y > p.y) != (poly[j].y > p.y)) &&
            (p.x < (poly[j].x - poly[i].x) *
             (p.y - poly[i].y) /
             (poly[j].y - poly[i].y + poly[i].x)))
            inside = !inside;
    }
    return inside;
}
```

This toggles `inside` every time a crossing is found.

#### B. Winding Number Algorithm

Counts how many times the polygon winds around the point.

- Nonzero winding number → Inside- Zero → Outside More robust for complex polygons with holes or self-intersections.

Method	Time Complexity	Robustness
Ray Casting	(O(n))	Simple, may fail on edge cases
Winding Number	(O(n))	More accurate for complex shapes

## **Edge Cases**

Handle:

- Points on edges or vertices- Horizontal edges (special treatment to avoid double counting)  
Numerical precision is key.

## **Applications**

- Hit testing in computer graphics- GIS spatial queries- Collision detection

## **2. Polygon Triangulation**

A polygon triangulation divides a polygon into non-overlapping triangles whose union equals the polygon.

Why triangulate?

- Triangles are simple, stable, and efficient for rendering and computation.- Used in graphics pipelines, area computation, physics, and mesh generation.

### **A. Triangulation Basics**

For a simple polygon with ( n ) vertices,

- Always possible- Always yields ( n - 2 ) triangles Goal: Find a triangulation efficiently and stably.

### **B. Ear Clipping Algorithm**

An intuitive and widely used method for triangulation.

## Idea

1. Find an ear: a triangle formed by three consecutive vertices ( $v_{i-1}, v_i, v_{i+1}$ ) such that:
  - It is convex - Contains no other vertex inside
2. Clip the ear (remove vertex  $v_i$ )
3. Repeat until only one triangle remains

Time Complexity: (  $O(n^2)$  )

Tiny Code (Ear Clipping Sketch)

```
while (n > 3) {  
    for (i = 0; i < n; i++) {  
        if (is_ear(i)) {  
            add_triangle(i-1, i, i+1);  
            remove_vertex(i);  
            break;  
        }  
    }  
}
```

Helper `is_ear()` checks convexity and emptiness.

## C. Dynamic Programming for Convex Polygons

If the polygon is convex, use DP triangulation:

$$dp[i][j] = \min_{k \in (i,j)} dp[i][k] + dp[k][j] + cost(i, j, k)$$

Cost: perimeter or area (for minimum-weight triangulation)

Time Complexity: (  $O(n^3)$  ) Space: (  $O(n^2)$  )

## D. Divide and Conquer

Recursively split polygon and triangulate sub-polygons. Useful for convex or near-convex shapes.

Algorithm	Time	Notes
Ear Clipping	$(On^2)$	Simple polygons
DP Triangulation	$(On^3)$	Weighted cost
Convex Polygon	$(O(n))$	Straightforward

### 3. Applications

Domain	Usage
Computer Graphics	Rendering, rasterization
Computational Geometry	Area computation, integration
Finite Element Analysis	Mesh subdivision
Robotics	Path planning, map decomposition

### Why It Matters

Point-in-polygon answers where you are. Triangulation tells you how space is built. Together, they form the foundation of geometric reasoning.

“From a single point to a thousand triangles, geometry turns space into structure.”

### Try It Yourself

1. Draw a non-convex polygon and test random points using the ray casting rule.
2. Implement the ear clipping algorithm for a simple polygon.
3. Visualize how each step removes an ear and simplifies the shape.
4. Compare triangulation results for convex vs concave shapes.

## 76. Spatial Data Structures (KD, R-tree)

When working with geometric data, points, rectangles, or polygons, efficient lookup and organization are crucial. Spatial data structures are designed to answer queries like:

- Which objects are near a given point?- Which shapes intersect a region?- What's the nearest neighbor? They form the backbone of computational geometry, computer graphics, GIS, and search systems.

## 1. Motivation

Brute force approaches that check every object have (  $O(n)$  ) or worse performance. Spatial indexing structures, like KD-Trees and R-Trees, enable efficient range queries, nearest neighbor searches, and spatial joins.

## 2. KD-Tree (k-dimensional tree)

A KD-tree is a binary tree that recursively partitions space using axis-aligned hyperplanes.

Each node splits the data by one coordinate axis (x, y, z, ...).

### Structure

- Each node represents a point.- Each level splits by a different axis (x, y, x, y, ...).- Left child contains points with smaller coordinate.- Right child contains larger coordinate.

Tiny Code (KD-tree Construction in 2D)

```
typedef struct {
    double x, y;
} Point;

int axis; // 0 for x, 1 for y

KDNode* build(Point points[], int n, int depth) {
    if (n == 0) return NULL;
    axis = depth % 2;
    int mid = n / 2;
    nth_element(points, points + mid, points + n, compare_by_axis);
    KDNode* node = new_node(points[mid]);
    node->left = build(points, mid, depth + 1);
    node->right = build(points + mid + 1, n - mid - 1, depth + 1);
    return node;
}
```

Search Complexity:

- Average: (  $O(\log n)$  )- Worst-case: (  $O(n)$  )

## Queries

- Range query: Find points in a region.- Nearest neighbor: Search branches that might contain closer points.- K-nearest neighbors: Use priority queues.

## Pros & Cons

Pros	Cons
Efficient for static data	Costly updates
Good for low dimensions	Degrades with high dimensions

## Applications

- Nearest neighbor in ML- Collision detection- Clustering (e.g., k-means acceleration)

### 3. R-Tree (Rectangle Tree)

An R-tree is a height-balanced tree for rectangular bounding boxes. It's the spatial analog of a B-tree.

#### Idea

- Store objects or bounding boxes in leaf nodes.- Internal nodes store MBRs (Minimum Bounding Rectangles) that cover child boxes.- Query by traversing overlapping MBRs.

Tiny Code (R-Tree Node Sketch)

```
typedef struct {
    Rectangle mbr;
    Node* children[MAX_CHILDREN];
    int count;
} Node;
```

Insertion chooses the child whose MBR expands least to accommodate the new entry.

## Operations

- Insert: Choose subtree → Insert → Adjust MBRs- Search: Descend into nodes whose MBR intersects query- Split: When full, use heuristics (linear, quadratic, R\*-Tree)  
Complexity:
  - Query: ( Olog n )- Insert/Delete: ( Olog n ) average

## Pros & Cons

Pros	Cons
Supports dynamic data	Overlaps can degrade performance
Ideal for rectangles	Complex split rules

## Variants

- R\*-Tree: Optimized reinsertion, better packing- R+ Tree: Non-overlapping partitions  
Hilbert R-Tree: Uses space-filling curves

## 4. Comparison

Feature	KD-Tree	R-Tree
Data Type	Points	Rectangles / Regions
Dimensionality	Low (2-10)	Medium
Use Case	NN, range queries	Spatial joins, overlap queries
Updates	Expensive	Dynamic-friendly
Balance	Recursive median	B-tree-like

## 5. Other Spatial Structures

Structure	Description
Quadtree	Recursive 2D subdivision into 4 quadrants
Octree	3D analog of quadtree
BSP Tree	Binary partition using arbitrary planes
Grid Index	Divide space into uniform grid cells

## 6. Applications

Domain	Usage
GIS	Region queries, map intersections
Graphics	Ray tracing acceleration
Robotics	Collision and path planning
ML	Nearest neighbor search
Databases	Spatial indexing

### Why It Matters

Spatial structures turn geometry into searchable data. They enable efficient algorithms for where and what's near, vital for real-time systems.

“Divide space wisely, and queries become whispers instead of shouts.”

### Try It Yourself

1. Build a KD-tree for 10 random 2D points.
2. Implement nearest neighbor search.
3. Insert rectangles into a simple R-tree and query intersection with a bounding box.
4. Compare query time vs brute force.

## 77. Rasterization and Scanline Techniques

When you draw shapes on a screen, triangles, polygons, circles, they must be converted into pixels. This conversion is called rasterization. It's the bridge between geometric math and visible images.

Rasterization and scanline algorithms are foundational to computer graphics, game engines, and rendering pipelines.

### 1. What Is Rasterization?

Rasterization transforms vector shapes (continuous lines and surfaces) into discrete pixels on a grid.

For example, a triangle defined by vertices  $(x_1, y_1)$ ,  $(x_2, y_2)$ ,  $(x_3, y_3)$  must be filled pixel by pixel.

## 2. Core Idea

Each shape (line, polygon, circle) is sampled over a grid. The algorithm decides which pixels are inside, on, or outside the shape.

A rasterizer answers:

- Which pixels should be lit?- What color or depth should each pixel have?

## 3. Line Rasterization (Bresenham's Algorithm)

A classic method for drawing straight lines with integer arithmetic.

Key Idea: Move from one pixel to the next, choosing the pixel closest to the true line path.

```
void draw_line(int x0, int y0, int x1, int y1) {
    int dx = abs(x1 - x0), dy = abs(y1 - y0);
    int sx = (x0 < x1) ? 1 : -1;
    int sy = (y0 < y1) ? 1 : -1;
    int err = dx - dy;
    while (true) {
        plot(x0, y0); // draw pixel
        if (x0 == x1 && y0 == y1) break;
        int e2 = 2 * err;
        if (e2 > -dy) { err -= dy; x0 += sx; }
        if (e2 < dx) { err += dx; y0 += sy; }
    }
}
```

Why it works: Bresenham avoids floating-point math and keeps the line visually continuous.

## 4. Polygon Rasterization

To fill shapes, we need scanline algorithms, they sweep a horizontal line (y-axis) across the shape and fill pixels in between edges.

## Scanline Fill Steps

1. Sort edges by their y-coordinates.
2. Scan each line (y).
3. Find intersections with polygon edges.
4. Fill between intersection pairs.

This guarantees correct filling for convex and concave polygons.

## Example (Simple Triangle Rasterization)

```
for (int y = y_min; y <= y_max; y++) {  
    find all x-intersections with polygon edges;  
    sort x-intersections;  
    for (int i = 0; i < count; i += 2)  
        draw_line(x[i], y, x[i+1], y);  
}
```

## 5. Circle Rasterization (Midpoint Algorithm)

Use symmetry, a circle is symmetric in 8 octants.

Each step calculates the error term to decide whether to move horizontally or diagonally.

```
void draw_circle(int xc, int yc, int r) {  
    int x = 0, y = r, d = 3 - 2 * r;  
    while (y >= x) {  
        plot_circle_points(xc, yc, x, y);  
        x++;  
        if (d > 0) { y--; d += 4 * (x - y) + 10; }  
        else d += 4 * x + 6;  
    }  
}
```

## 6. Depth and Shading

In 3D graphics, rasterization includes depth testing (Z-buffer) and color interpolation. Each pixel stores its depth; new pixels overwrite only if closer.

Interpolated shading (Gouraud, Phong) computes smooth color transitions across polygons.

## 7. Hardware Rasterization

Modern GPUs perform rasterization in parallel:

- Vertex Shader → Projection- Rasterizer → Pixel Grid- Fragment Shader → Color & Depth Each pixel is processed in fragment shaders for lighting, texture, and effects.

## 8. Optimizations

Technique	Purpose
Bounding Box Clipping	Skip off-screen regions
Early Z-Culling	Discard hidden pixels early
Edge Functions	Fast inside-test for triangles
Barycentric Coordinates	Interpolate depth/color smoothly

## 9. Why It Matters

Rasterization turns math into imagery. It's the foundation of all visual computing, renderers, CAD, games, and GUIs. Even with ray tracing rising, rasterization remains dominant for real-time rendering.

“Every pixel you see began as math, it’s just geometry painted by light.”

## 10. Try It Yourself

1. Implement Bresenham's algorithm for lines.
2. Write a scanline polygon fill for triangles.
3. Extend it with color interpolation using barycentric coordinates.
4. Compare performance vs brute force (looping over all pixels).

## 78. Computer Vision (Canny, Hough, SIFT)

Computer vision is where algorithms learn to see, to extract structure, shape, and meaning from images. Behind every object detector, edge map, and keypoint matcher lies a handful of powerful geometric algorithms.

In this section, we explore four pillars of classical vision: Canny edge detection, Hough transform, and SIFT (Scale-Invariant Feature Transform).

## 1. The Vision Pipeline

Most vision algorithms follow a simple pattern:

1. Input: Raw pixels (grayscale or color)
2. Preprocess: Smoothing or filtering
3. Feature extraction: Edges, corners, blobs
4. Detection or matching: Shapes, keypoints
5. Interpretation: Object recognition, tracking

Canny, Hough, and SIFT live in the feature extraction and detection stages.

## 2. Canny Edge Detector

Edges mark places where intensity changes sharply, the outlines of objects. The Canny algorithm (1986) is one of the most robust and widely used edge detectors.

### Steps

1. Smoothing: Apply Gaussian blur to reduce noise.
2. Gradient computation:
  - Compute  $G_x$  and  $G_y$  via Sobel filters
  - Gradient magnitude:  $G = \sqrt{G_x^2 + G_y^2}$
  - Gradient direction:  $\theta = \tan^{-1} \frac{G_y}{G_x}$
3. Non-maximum suppression:
  - Keep only local maxima along the gradient direction
4. Double thresholding:
  - Strong edges (high gradient)
  - Weak edges (connected to strong ones)
5. Edge tracking by hysteresis:
  - Connect weak edges linked to strong edges

## Tiny Code (Pseudocode)

```
Image canny(Image input) {
    Image smoothed = gaussian_blur(input);
    Gradient grad = sobel(smoothed);
    Image suppressed = non_max_suppression(grad);
    Image edges = hysteresis_threshold(suppressed, low, high);
    return edges;
}
```

## Why Canny Works

Canny maximizes three criteria:

1. Good detection (low false negatives)
2. Good localization (edges close to true edges)
3. Single response (no duplicates)

It's a careful balance between sensitivity and stability.

## 3. Hough Transform

Canny finds edge points, Hough connects them into shapes.

The Hough transform detects lines, circles, and other parametric shapes using voting in parameter space.

### Line Detection

Equation of a line:

$$\rho = x \cos \theta + y \sin \theta$$

Each edge point votes for all  $(\rho, \theta)$  combinations it could belong to. Peaks in the accumulator array correspond to strong lines.

Tiny Code (Hough Transform)

```
for each edge point (x, y):
    for theta in [0, 180]:
        rho = x*cos(theta) + y*sin(theta);
        accumulator[rho, theta]++;
```

Then pick  $(\rho, \theta)$  with highest votes.

### Circle Detection

Use 3D accumulator  $center_x, center_y, radius$ . Each edge pixel votes for possible circle centers.

### Applications

- Lane detection in self-driving- Shape recognition (circles, ellipses)- Document analysis (lines, grids)

## 4. SIFT (Scale-Invariant Feature Transform)

SIFT finds keypoints that remain stable under scale, rotation, and illumination changes.

It's widely used for image matching, panoramas, 3D reconstruction, and object recognition.

### Steps

#### 1. Scale-space extrema detection

- Use Difference of Gaussians (DoG) across scales. - Detect maxima/minima in space-scale neighborhood.
- 2. Keypoint localization
- Refine keypoint position and discard unstable ones.
- 3. Orientation assignment
- Assign dominant gradient direction.
- 4. Descriptor generation
- Build a 128D histogram of gradient orientations in a local patch.

Tiny Code (Outline)

```
for each octave:  
    build scale-space pyramid  
    find DoG extrema  
    localize keypoints  
    assign orientations  
    compute 128D descriptor
```

### Properties

Property	Description
Scale Invariant	Detects features at multiple scales
Rotation Invariant	Uses local orientation
Robust	Handles lighting, noise, affine transforms

## 5. Comparison

Algorithm	Purpose	Output	Robustness
Canny	Edge detection	Binary edge map	Sensitive to thresholds
Hough	Shape detection	Lines, circles	Needs clean edges
SIFT	Feature detection	Keypoints, descriptors	Very robust

## 6. Applications

Domain	Use Case
Robotics	Visual SLAM, localization
AR / VR	Marker tracking
Search	Image matching
Medical	Edge segmentation
Industry	Quality inspection

## 7. Modern Successors

- ORB (FAST + BRIEF): Efficient for real-time- SURF: Faster SIFT alternative- Harris / FAST: Corner detectors- Deep features: CNN-based descriptors

## Why It Matters

These algorithms gave machines their first eyes, before deep learning, they were how computers recognized structure. Even today, they're used in preprocessing, embedded systems, and hybrid pipelines.

“Before neural nets could dream, vision began with gradients, geometry, and votes.”

## Try It Yourself

1. Implement Canny using Sobel and hysteresis.
2. Use Hough transform to detect lines in a synthetic image.
3. Try OpenCV SIFT to match keypoints between two rotated images.
4. Compare edge maps before and after Gaussian blur.

## 79. Pathfinding in Space (A\*, RRT, PRM)

When navigating a maze, driving an autonomous car, or moving a robot arm, the question is the same: How do we find a path from start to goal efficiently and safely?

Pathfinding algorithms answer this question, balancing optimality, speed, and adaptability. In this section, we explore three foundational families:

- A\*: Heuristic search in grids and graphs- RRT (Rapidly-Exploring Random Tree): Sampling-based exploration- PRM (Probabilistic Roadmap): Precomputed navigation networks

### 1. The Pathfinding Problem

Given:

- A space (grid, graph, or continuous)- A start node and goal node- A cost function (distance, time, energy)- Optional obstacles Find a collision-free, low-cost path.

### 2. A\* (A-star) Search

A\* combines Dijkstra's algorithm with a heuristic that estimates remaining cost. It's the most popular graph-based pathfinding algorithm.

#### Key Idea

Each node ( n ) has:

$$f(n) = g(n) + h(n)$$

- ( g(n) ): cost so far- ( h(n) ): estimated cost to goal- ( f(n) ): total estimated cost

Algorithm

1. Initialize priority queue with start node

2. While queue not empty:

- Pop node with smallest (  $f(n)$  ) - If goal reached → return path - For each neighbor:
  - Compute new (  $g$  ), (  $f$  ) - Update queue if better

Tiny Code (Grid A\*)

```

typedef struct { int x, y; double g, f; } Node;

double heuristic(Node a, Node b) {
    return fabs(a.x - b.x) + fabs(a.y - b.y); // Manhattan
}

void a_star(Node start, Node goal) {
    PriorityQueue open;
    push(open, start);
    while (!empty(open)) {
        Node cur = pop_min(open);
        if (cur == goal) return reconstruct_path();
        for (Node n : neighbors(cur)) {
            double tentative_g = cur.g + dist(cur, n);
            if (tentative_g < n.g) {
                n.g = tentative_g;
                n.f = n.g + heuristic(n, goal);
                push(open, n);
            }
        }
    }
}

```

## Complexity

- Time: (  $O(E \log V)$  ) - Space: (  $O(V)$  ) - Optimal if (  $h(n)$  ) is admissible (never overestimates)

## Variants

Variant	Description
Dijkstra	A* with ( $h(n) = 0$ )
Greedy Best-First	Uses ( $h(n)$ ) only
Weighted A*	Speeds up with tradeoff on optimality

Variant	Description
Jump Point Search	Optimized for uniform grids

### 3. RRT (Rapidly-Exploring Random Tree)

A\* struggles in continuous or high-dimensional spaces (e.g. robot arms). RRT tackles this with randomized exploration.

#### Core Idea

- Grow a tree from the start by randomly sampling points.- Extend tree toward each sample (step size  $\epsilon$ ).- Stop when near the goal.

Tiny Code (RRT Sketch)

```
Tree T = {start};
for (int i = 0; i < MAX_ITERS; i++) {
    Point q_rand = random_point();
    Point q_near = nearest(T, q_rand);
    Point q_new = steer(q_near, q_rand, step_size);
    if (collision_free(q_near, q_new))
        add_edge(T, q_near, q_new);
    if (distance(q_new, goal) < eps)
        return path;
}
```

#### Pros & Cons

Pros	Cons
Works in continuous space	Paths are suboptimal
Handles high dimensions	Randomness may miss narrow passages
Simple and fast	Needs post-processing (smoothing)

#### Variants

Variant	Description
RRT*	Asymptotically optimal
Bi-RRT	Grow from both start and goal
Informed RRT*	Focus on promising regions

## 4. PRM (Probabilistic Roadmap)

PRM builds a graph of feasible configurations, a roadmap, then searches it.

### Steps

1. Sample random points in free space
2. Connect nearby points with collision-free edges
3. Search roadmap (e.g., with A\*)

Tiny Code (PRM Sketch)

```
Graph G = {};
for (int i = 0; i < N; i++) {
    Point p = random_free_point();
    G.add_vertex(p);
}
for each p in G:
    for each q near p:
        if (collision_free(p, q))
            G.add_edge(p, q);
path = a_star(G, start, goal);
```

### Pros & Cons

Pros	Cons
Precomputes reusable roadmap	Needs many samples for coverage
Good for multiple queries	Poor for single-query planning
Works in high-dim spaces	May need post-smoothing

## 5. Comparison

Algorithm	Space	Nature	Optimal	Use Case
A*	Discrete	Deterministic	Yes	Grids, graphs
RRT	Continuous	Randomized	No (RRT* = Yes)	Robotics, motion planning
PRM	Continuous	Randomized	Approx.	Multi-query planning

## 6. Applications

Domain	Use Case
Robotics	Arm motion, mobile navigation
Games	NPC pathfinding, AI navigation mesh
Autonomous vehicles	Route planning
Aerospace	Drone and spacecraft trajectory
Logistics	Warehouse robot movement

### Why It Matters

Pathfinding is decision-making in space, it gives agents the ability to move, explore, and act purposefully. From Pac-Man to Mars rovers, every journey starts with an algorithm.

“To move with purpose, one must first see the paths that are possible.”

### Try It Yourself

1. Implement A\* on a 2D grid with walls.
2. Generate an RRT in a 2D obstacle field.
3. Build a PRM for a continuous space and run A\* on the roadmap.
4. Compare speed and path smoothness across methods.

## 80. Computational Geometry Variants and Applications

Computational geometry is the study of algorithms on geometric data, points, lines, polygons, circles, and higher-dimensional shapes. By now, you’ve seen core building blocks: convex hulls, intersections, nearest neighbors, triangulations, and spatial indexing.

This final section brings them together through variants, generalizations, and real-world applications, showing how geometry quietly powers modern computing.

## 1. Beyond the Plane

Most examples so far assumed 2D geometry. But real systems often live in 3D or N-D spaces.

Dimension	Example Problems	Typical Uses
2D	Convex hull, polygon area, line sweep	GIS, CAD, mapping
3D	Convex polyhedra, mesh intersection, visibility	Graphics, simulation
N-D	Voronoi in high-D, KD-trees, optimization	ML, robotics, data science

Higher dimensions add complexity (and sometimes impossibility):

- Exact geometry often replaced by approximations.- Volume, distance, and intersection tests become more expensive.

## 2. Approximate and Robust Geometry

Real-world geometry faces numerical errors (floating point) and degenerate cases (collinear, overlapping). To handle this, algorithms adopt robustness and approximation strategies.

- Epsilon comparisons: treat values within tolerance as equal- Orientation tests: robustly compute turn direction via cross product- Exact arithmetic: rational or symbolic computation- Grid snapping: quantize space for stability Approximate geometry accepts small error for large speed-up, essential in graphics and machine learning.

## 3. Geometric Duality

A powerful tool for reasoning about problems: map points to lines, lines to points. For example:

- A point  $((a, b))$  maps to line  $(y = ax - b)$ .- A line  $(y = mx + c)$  maps to point  $((m, -c))$ . Applications:
- Transforming line intersection problems into point location problems- Simplifying half-plane intersections- Enabling arrangement algorithms in computational geometry Duality is a common trick: turn geometry upside-down to make it simpler.

## 4. Geometric Data Structures

Recap of core spatial structures and what they're best at:

Structure	Stores	Queries	Use Case
KD-Tree	Points	NN, range	Low-D search
R-Tree	Rectangles	Overlaps	Spatial DB
Quad/Octree	Space partitions	Point lookup	Graphics, GIS
BSP Tree	Polygons	Visibility	Rendering
Delaunay Triangulation	Points	Neighbors	Mesh generation
Segment Tree	Intervals	Range	Sweep-line events

## 5. Randomized Geometry

Randomness simplifies deterministic geometry:

- Randomized incremental construction (Convex Hulls, Delaunay)- Random sampling for approximation ( -nets, VC dimension)- Monte Carlo geometry for probabilistic intersection and coverage Example: randomized incremental convex hull builds expected (  $O(n \log n)$  ) structures with elegant proofs.

## 6. Computational Topology

Beyond geometry lies shape connectivity, studied by topology. Algorithms compute connected components, holes, homology, and Betti numbers.

Applications include:

- 3D printing (watertightness)- Data analysis (persistent homology)- Robotics (free space topology) Geometry meets topology in alpha-shapes, simplicial complexes, and manifold reconstruction.

## 7. Geometry Meets Machine Learning

Many ML methods are geometric at heart:

- Nearest neighbor → Voronoi diagram- SVM → hyperplane separation- K-means → Voronoi partitioning- Manifold learning → low-dim geometry- Convex optimization → geometric feasibility Visualization tools (t-SNE, UMAP) rely on spatial embedding and distance geometry.

## 8. Applications Across Fields

Field	Application	Geometric Core
Graphics	Rendering, collision	Triangulation, ray tracing
GIS	Maps, roads	Polygons, point-in-region
Robotics	Path planning	Obstacles, configuration space
Architecture	Modeling	Mesh operations
Vision	Object boundaries	Contours, convexity
AI	Clustering, similarity	Distance metrics
Physics	Simulation	Particle collision
Databases	Spatial joins	R-Trees, indexing

Geometry underpins structure, position, and relationship, the backbone of spatial reasoning.

## 9. Complexity and Open Problems

Some problems still challenge efficient solutions:

- Point location in dynamic settings- Visibility graphs in complex polygons- Motion planning in high dimensions- Geometric median / center problems- Approximation guarantees in robust settings These remain active areas in computational geometry research.

### Tiny Code (Point-in-Polygon via Ray Casting)

```
bool inside(Point p, Polygon poly) {
    int cnt = 0;
    for (int i = 0; i < poly.n; i++) {
        Point a = poly[i], b = poly[(i + 1) % poly.n];
        if (intersect_ray(p, a, b)) cnt++;
    }
    return cnt % 2 == 1; // odd crossings = inside
}
```

This small routine appears everywhere, maps, games, GUIs, and physics engines.

## 10. Why It Matters

Computational geometry is more than shape, it's the mathematics of space, powering visual computing, spatial data, and intelligent systems. Everywhere something moves, collides, maps, or recognizes form, geometry is the invisible hand guiding it.

“All computation lives somewhere, and geometry is how we understand the where.”

## Try It Yourself

1. Implement point-in-polygon and test on convex vs concave shapes.
2. Visualize a Delaunay triangulation and its Voronoi dual.
3. Experiment with KD-trees for nearest neighbor queries.
4. Write a small convex hull in 3D using incremental insertion.
5. Sketch an RRT path over a geometric map.

# Chapter 9. Systems, Databases, and Distributed Algorithms

## 81. Concurrency Control (2PL, MVCC, OCC)

In multi-user or multi-threaded systems, many operations want to read or write shared data at the same time. Without discipline, this leads to chaos, lost updates, dirty reads, or even inconsistent states.

Concurrency control ensures correctness under parallelism, so that the result is as if each transaction ran alone (a property called serializability).

This section introduces three foundational techniques:

- 2PL - Two-Phase Locking- MVCC - Multi-Version Concurrency Control- OCC - Optimistic Concurrency Control

### 1. The Goal: Serializability

We want transactions to behave as if executed in some serial order, even though they're interleaved.

A schedule is *serializable* if it yields the same result as some serial order of transactions.

Concurrency control prevents problems like:

- Lost Update: Two writes overwrite each other.- Dirty Read: Read uncommitted data.- Non-repeatable Read: Data changes mid-transaction.- Phantom Read: New rows appear after a query.

## 2. Two-Phase Locking (2PL)

Idea: Use locks to coordinate access. Each transaction has two phases:

1. Growing phase: acquire locks (shared or exclusive)
2. Shrinking phase: release locks (no new locks allowed after release)

This ensures conflict-serializability.

### Lock Types

Type	Operation	Shared?	Exclusive?
Shared (S)	Read	Yes	No
Exclusive (X)	Write	No	No

If a transaction needs to read: request S-lock. If it needs to write: request X-lock.

Tiny Code (Lock Manager Sketch)

```
void acquire_lock(Transaction *T, Item *X, LockType type) {
    while (conflict_exists(X, type))
        wait();
    add_lock(X, T, type);
}

void release_all(Transaction *T) {
    for (Lock *l in T->locks)
        unlock(l);
}
```

### Example

```
T1: read(A); write(A)
T2: read(A); write(A)
```

Without locks → race condition. With 2PL → one must wait → consistent.

### Variants

Variant	Description
Strict 2PL	Holds all locks until commit → avoids cascading aborts
Rigorous 2PL	Same as Strict, all locks released at end
Conservative 2PL	Acquires all locks before execution

### Pros & Cons

Pros	Cons
Guarantees serializability	Can cause deadlocks
Simple concept	Blocking, contention under load

## 3. Multi-Version Concurrency Control (MVCC)

Idea: Readers don't block writers, and writers don't block readers. Each write creates a new version of data with a timestamp.

Transactions read from a consistent snapshot based on their start time.

### Snapshot Isolation

- Readers see the latest committed version at transaction start.- Writers produce new versions; conflicts detected at commit time. Each record stores:
  - `value- created_at- deleted_at` (if applicable)

Tiny Code (Version Chain)

```
struct Version {
    int value;
    Timestamp created;
    Timestamp deleted;
    Version *next;
};
```

Read finds version with `created <= tx.start && deleted > tx.start`.

### Pros & Cons

Pros	Cons
No read locks	Higher memory (multiple versions)
Readers never block	Write conflicts at commit
Great for OLTP systems	GC of old versions needed

## Used In

- PostgreSQL- Oracle- MySQL (InnoDB)- Spanner

## 4. Optimistic Concurrency Control (OCC)

Idea: Assume conflicts are rare. Let transactions run without locks. At commit time, validate, if conflicts exist, rollback.

### Phases

1. Read phase - execute, read data, buffer writes.
2. Validation phase - check if conflicts occurred.
3. Write phase - apply changes if valid, else abort.

Tiny Code (OCC Validation)

```
bool validate(Transaction *T) {
    for (Transaction *U in committed_since(T.start))
        if (conflict(T, U))
            return false;
    return true;
}
```

### Pros & Cons

Pros	Cons
No locks → no deadlocks	High abort rate under contention
Great for low-conflict workloads	Wasted work on abort

## Used In

- In-memory DBs- Distributed systems- STM (Software Transactional Memory)

## 5. Choosing a Strategy

System Type	Preferred Control
OLTP (many reads/writes)	MVCC
OLAP (read-heavy)	MVCC or OCC
Real-time systems	2PL (predictable)
Low contention	OCC
High contention	2PL / MVCC

## 6. Why It Matters

Concurrency control is the backbone of consistency in databases, distributed systems, and even multi-threaded programs. It enforces correctness amid chaos, ensuring your data isn't silently corrupted.

“Without order, parallelism is noise. Concurrency control is its conductor.”

### Try It Yourself

1. Simulate 2PL with two transactions updating shared data.
2. Implement a toy MVCC table with version chains.
3. Write an OCC validator for three concurrent transactions.
4. Experiment: under high conflict, which model performs best?

## 82. Logging, Recovery, and Commit Protocols

No matter how elegant your algorithms or how fast your storage, failures happen. Power cuts, crashes, and network splits are inevitable. What matters is recovery, restoring the system to a consistent state without losing committed work.

Logging, recovery, and commit protocols form the backbone of reliable transactional systems, ensuring durability and correctness in the face of crashes.

### 1. The Problem

We need to guarantee the ACID properties:

- Atomicity - all or nothing- Consistency - valid before and after- Isolation - no interference- Durability - once committed, always safe If a crash occurs mid-transaction, how do we roll back or redo correctly?

The answer: Log everything, then replay or undo after failure.

## 2. Write-Ahead Logging (WAL)

The golden rule:

“Write log entries before modifying the database.”

Every action is recorded in a sequential log on disk, ensuring the system can reconstruct the state.

### Log Record Format

Each log entry typically includes:

- LSN (Log Sequence Number)- Transaction ID- Operation (update, insert, delete)- Before image (old value)- After image (new value)

```
struct LogEntry {  
    int lsn;  
    int tx_id;  
    char op[10];  
    Value before, after;  
};
```

When a transaction commits, the system first flushes logs to disk (`fsync`). Only then is the commit acknowledged.

## 3. Recovery Actions

When the system restarts, it reads logs and applies a recovery algorithm.

### Three Phases (ARIES Model)

1. Analysis - determine state at crash (active vs committed)
2. Redo - repeat all actions from last checkpoint
3. Undo - rollback incomplete transactions

ARIES (Algorithm for Recovery and Isolation Exploiting Semantics) is the most widely used approach (IBM DB2, PostgreSQL, SQL Server).

## **Redo Rule**

If the system committed before crash → redo all updates so data is preserved.

## **Undo Rule**

If the system didn't commit → undo to maintain atomicity.

Tiny Code (Simplified Recovery Sketch)

```
void recover(Log log) {
    for (Entry e : log) {
        if (e.committed)
            apply(e.after);
        else
            apply(e.before);
    }
}
```

## **4. Checkpointing**

Instead of replaying the entire log, systems take checkpoints, periodic snapshots marking a safe state.

Type	Description
Sharp checkpoint	Stop all transactions briefly, flush data + log
Fuzzy checkpoint	Mark consistent LSN; continue running

Checkpoints reduce recovery time: only replay after the last checkpoint.

## **5. Commit Protocols**

In distributed systems, multiple nodes must agree to commit or abort together. This is handled by atomic commit protocols.

## Two-Phase Commit (2PC)

Goal: All participants either commit or abort in unison.

Steps:

1. Prepare phase (voting):

- Coordinator asks all participants to “prepare” - Each replies yes/no
- Commit phase (decision):
- If all say yes → commit - Else → abort

```
Coordinator: PREPARE
Participants: VOTE YES / NO
Coordinator: COMMIT / ABORT
```

If the coordinator crashes after prepare, participants must wait → blocking protocol.

Tiny Code (2PC Pseudocode)

```
bool two_phase_commit(Participants P) {
    for each p in P:
        if (!p.prepare()) return abort_all();
    for each p in P:
        p.commit();
    return true;
}
```

## Three-Phase Commit (3PC)

Improves on 2PC by adding an intermediate phase to avoid indefinite blocking. More complex, used in systems with reliable failure detection.

## 6. Logging in Distributed Systems

Each participant maintains its own WAL. To recover globally:

- Use coordinated checkpoints- Maintain global commit logs- Consensus-based protocols (Paxos Commit, Raft) can replace 2PC for high availability

## 7. Example Timeline

Step	Action
T1 updates record A	WAL entry written
T1 updates record B	WAL entry written
T1 commits	WAL flush, commit record
Crash!	Disk may be inconsistent
Restart	Recovery scans log, redoes T1

## 8. Pros and Cons

Approach	Strength	Weakness
WAL	Simple, durable	Write overhead
Checkpointing	Faster recovery	I/O spikes
2PC	Global atomicity	Blocking
3PC / Consensus	Non-blocking	Complex, slower

## 9. Real Systems

System	Strategy
PostgreSQL	WAL + ARIES + Checkpoint
MySQL (InnoDB)	WAL + Fuzzy checkpoint
Spanner	WAL + 2PC + TrueTime
Kafka	WAL for durability
RocksDB	WAL + LSM checkpoints

## 10. Why It Matters

Logging and commit protocols make data survive crashes and stay consistent across machines. Without them, every failure risks corruption.

“Persistence is not about never failing, it’s about remembering how to stand back up.”

## Try It Yourself

1. Write a toy WAL system that logs before writes.
2. Simulate a crash mid-transaction and replay the log.
3. Implement a simple 2PC coordinator with two participants.

4. Compare recovery time with vs without checkpoints.

## 83. Scheduling (Round Robin, EDF, Rate-Monotonic)

In operating systems and real-time systems, scheduling determines the order in which tasks or processes run. Since resources like CPU time are limited, a good scheduler aims to balance fairness, efficiency, and responsiveness.

### 1. The Goal of Scheduling

Every system has tasks competing for the CPU. Scheduling decides:

- Which task runs next- How long it runs- When it yields or preempts Different goals apply in different domains:

Domain	Objective
General-purpose OS	Fairness, responsiveness
Real-time systems	Meeting deadlines
Embedded systems	Predictability
High-performance servers	Throughput, latency balance

A scheduler's policy can be preemptive (interrupts tasks) or non-preemptive (waits for voluntary yield).

### 2. Round Robin Scheduling

Round Robin (RR) is one of the simplest preemptive schedulers. Each process gets a fixed time slice (quantum) and runs in a circular queue.

If a process doesn't finish, it's put back at the end of the queue.

#### Tiny Code: Round Robin (Pseudocode)

```
queue processes;
while (!empty(processes)) {
    process = dequeue(processes);
    run_for_quantum(process);
    if (!process.finished)
```

```

    enqueue(processes, process);
}

```

## Characteristics

- Fair: Every process gets CPU time.- Responsive: Short tasks don't starve.- Downside: Context switching overhead if quantum is too small. ##### Example

Process	Burst Time
P1	4
P2	3
P3	2

Quantum = 1 Order: P1, P2, P3, P1, P2, P3, P1, P2 → all finish fairly.

## 3. Priority Scheduling

Each task has a priority. The scheduler always picks the highest-priority ready task.

- Preemptive: A higher-priority task can interrupt a lower one.- Non-preemptive: The CPU is released voluntarily. ##### Problems
- Starvation: Low-priority tasks may never run.- Solution: Aging - gradually increase waiting task priority.

## 4. Earliest Deadline First (EDF)

EDF is a dynamic priority scheduler for real-time systems. Each task has a deadline, and the task with the earliest deadline runs first.

### Rule

At any time, run the ready task with the closest deadline.

### Example

Task	Execution Time	Deadline
T1	1	3
T2	2	5
T3	1	2

Order: T3 → T1 → T2

EDF is optimal for preemptive scheduling of independent tasks on a single processor.

## 5. Rate-Monotonic Scheduling (RMS)

In periodic real-time systems, tasks repeat at fixed intervals. RMS assigns higher priority to tasks with shorter periods.

Task	Period	Priority
T1	2 ms	High
T2	5 ms	Medium
T3	10 ms	Low

It's static (priorities don't change) and optimal among fixed-priority schedulers.

## Utilization Bound

For n tasks, RMS is guaranteed schedulable if:

$$U = \sum_{i=1}^n \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$$

For example, for 3 tasks,  $U \leq 0.78$ .

## 6. Shortest Job First (SJF)

Run the task with the shortest burst time first.

- Non-preemptive SJF: Once started, runs to completion.- Preemptive SJF (Shortest Remaining Time First): Preempts if a shorter job arrives. Advantage: Minimizes average waiting time. Disadvantage: Needs knowledge of future job lengths.

## 7. Multilevel Queue Scheduling

Divide processes into classes (interactive, batch, system). Each class has its own queue with own policy, e.g.:

- Queue 1: System → RR (quantum = 10ms)- Queue 2: Interactive → RR (quantum = 50ms)- Queue 3: Batch → FCFS (First-Come-First-Serve) CPU is assigned based on queue priority.

## 8. Multilevel Feedback Queue (MLFQ)

Processes move between queues based on behavior.

- CPU-bound → move down (lower priority)- I/O-bound → move up (higher priority) Goal: Adaptive scheduling that rewards interactive tasks.

Used in modern OS kernels (Linux, Windows).

## 9. Scheduling Metrics

Metric	Meaning
Turnaround Time	Completion – Arrival
Waiting Time	Time spent in ready queue
Response Time	Time from arrival to first execution
Throughput	Completed tasks per unit time
CPU Utilization	% of time CPU is busy

Schedulers balance these based on design goals.

## 10. Why It Matters

Schedulers shape how responsive, efficient, and fair a system feels. In operating systems, they govern multitasking. In real-time systems, they ensure deadlines are met. In servers, they keep latency low and throughput high.

“Scheduling is not just about time. It’s about fairness, foresight, and flow.”

## Try It Yourself

1. Simulate Round Robin with quantum = 2, compare average waiting time.
2. Implement EDF for a set of periodic tasks with deadlines.
3. Check schedulability under RMS for 3 periodic tasks.
4. Explore Linux CFS (Completely Fair Scheduler) source code.
5. Compare SJF and RR for CPU-bound vs I/O-bound workloads.

## 84. Caching and Replacement (LRU, LFU, CLOCK)

Caching is the art of remembering the past to speed up the future. In computing, caches store recently used or frequently accessed data to reduce latency and load on slower storage (like disks or networks). The challenge: caches have limited capacity, so when full, we must decide what to evict. That's where replacement policies come in.

### 1. The Need for Caching

Caches appear everywhere:

- CPU: L1, L2, L3 caches speed up memory access- Databases: query results or index pages- Web browsers / CDNs: recently fetched pages- Operating systems: page cache for disk blocks The principle guiding all caches is locality:
- Temporal locality: recently used items are likely used again soon- Spatial locality: nearby items are likely needed next

### 2. Cache Replacement Problem

When the cache is full, which item should we remove?

We want to minimize cache misses (requests not found in cache).

Formally:

Given a sequence of accesses, find a replacement policy that minimizes misses.

Theoretical optimal policy (OPT): always evict the item used farthest in the future. But OPT requires future knowledge, so we rely on heuristics like LRU, LFU, CLOCK.

### 3. Least Recently Used (LRU)

LRU evicts the least recently accessed item. It assumes recently used = likely to be used again.

## Implementation Approaches

- Stack (list): move item to top on access- Hash map + doubly linked list:  $O(1)$  insert, delete, lookup ##### Tiny Code: LRU (Simplified)

```
typedef struct Node {
    int key;
    struct Node *prev, *next;
} Node;

HashMap cache;
List lru_list;

void access(int key) {
    if (in_cache(key)) move_to_front(key);
    else {
        if (cache_full()) remove_lru();
        insert_front(key);
    }
}
```

## Pros

- Good for workloads with strong temporal locality ##### Cons
- Costly in hardware or massive caches (metadata overhead)

## 4. Least Frequently Used (LFU)

LFU evicts the least frequently accessed item.

Tracks usage count for each item:

- Increment on each access- Evict lowest-count item ##### Example

Item	Accesses	Frequency
A	3	3
B	1	1
C	2	2

Evict B.

## **Variants**

- LFU with aging: gradually reduce counts to adapt to new trends- Approximate LFU: counters in ranges (for memory efficiency) ##### Pros
- Great for stable, repetitive workloads ##### Cons
- Poor for workloads with shifting popularity (slow adaptation)

## **5. FIFO (First In First Out)**

Simple but naive:

- Evict the oldest item, ignoring usage Used in simple hardware caches. Good when access pattern is cyclic, bad otherwise.

## **6. Random Replacement (RR)**

Evict a random entry.

Surprisingly competitive in some high-concurrency systems, and trivial to implement. Used in memcached (as an option).

## **7. CLOCK Algorithm**

A practical approximation of LRU, widely used in OS page replacement.

Each page has a reference bit (R). Pages form a circular list.

Algorithm:

1. Clock hand sweeps over pages.
2. If R = 0, evict page.
3. If R = 1, set R = 0 and skip.

This mimics LRU with O(1) cost and low overhead.

## **8. Second-Chance and Enhanced CLOCK**

Second-Chance: give recently used pages a “second chance” before eviction. Enhanced CLOCK: also uses modify bit (M) to prefer clean pages.

Used in Linux’s page replacement (with Active/Inactive lists).

## **9. Adaptive Algorithms**

Modern systems use hybrid or adaptive policies:

- ARC (Adaptive Replacement Cache) - balances recency and frequency- CAR (Clock with Adaptive Replacement) - CLOCK-style adaptation- TinyLFU - frequency sketch + admission policy- Hyperbolic caching - popularity decay for large-scale systems These adapt dynamically to changing workloads.

## **10. Why It Matters**

Caching is the backbone of system speed:

- OS uses it for paging- Databases for buffer pools- CPUs for memory hierarchies- CDNs for global acceleration Choosing the right eviction policy can mean orders of magnitude improvement in latency and throughput.

“A good cache remembers what matters, and forgets what no longer does.”

## **Try It Yourself**

1. Simulate a cache of size 3 with sequence: A B C A B D A B C D Compare LRU, LFU, and FIFO miss counts.
2. Implement LRU with a doubly-linked list and hash map in C.
3. Try CLOCK with reference bits, simulate a sweep.
4. Experiment with ARC and TinyLFU for dynamic workloads.
5. Measure hit ratios for different access patterns (sequential, random, looping).

## **85. Networking (Routing, Congestion Control)**

Networking algorithms make sure data finds its way through vast, connected systems, efficiently, reliably, and fairly. Two core pillars of network algorithms are routing (deciding *where* packets go) and congestion control (deciding *how fast* to send them).

Together, they ensure the internet functions under heavy load, dynamic topology, and unpredictable demand.

### **1. The Goals of Networking Algorithms**

- Correctness: all destinations are reachable if paths exist- Efficiency: use minimal resources (bandwidth, latency, hops)- Scalability: support large, dynamic networks- Robustness: recover from failures- Fairness: avoid starving flows

## 2. Types of Routing

Routing decides paths packets should follow through a graph-like network.

### Static vs Dynamic Routing

- Static: fixed routes, manual configuration (good for small networks)- Dynamic: routes adjust automatically as topology changes (internet-scale) ##### Unicast, Multicast, Broadcast
- Unicast: one-to-one (most traffic)- Multicast: one-to-many (video streaming, gaming)- Broadcast: one-to-all (local networks)

## 3. Shortest Path Routing

Most routing relies on shortest path algorithms:

### Dijkstra's Algorithm

- Builds shortest paths from one source- Complexity:  $O(E \log V)$  with priority queue  
Used in:
  - OSPF (Open Shortest Path First)- IS-IS (Intermediate System to Intermediate System)  
##### Bellman-Ford Algorithm
  - Handles negative edges- Basis for Distance-Vector routing (RIP) ##### Tiny Code: Dijkstra for Routing

```
#define INF 1e9
int dist[MAX], visited[MAX];
vector<pair<int,int>> adj[MAX];

void dijkstra(int s, int n) {
    for (int i = 0; i < n; i++) dist[i] = INF;
    dist[s] = 0;
    priority_queue<pair<int,int>> pq;
    pq.push({0, s});
    while (!pq.empty()) {
        int u = pq.top().second; pq.pop();
        if (visited[u]) continue;
        visited[u] = 1;
        for (auto [v, w]: adj[u]) {
```

```

        if (dist[v] > dist[u] + w) {
            dist[v] = dist[u] + w;
            pq.push({-dist[v], v});
        }
    }
}

```

#### 4. Distance-Vector vs Link-State

Feature	Distance-Vector (RIP)	Link-State (OSPF)
Info Shared	Distance to neighbors	Full topology map
Convergence	Slower (loops possible)	Fast (SPF computation)
Complexity	Lower	Higher
Examples	RIP, BGP (conceptually)	OSPF, IS-IS

RIP uses Bellman-Ford. OSPF floods link-state updates, runs Dijkstra at each node.

#### 5. Hierarchical Routing

Large-scale networks (like the Internet) use hierarchical routing:

- Routers grouped into Autonomous Systems (AS)- Intra-AS routing: OSPF, IS-IS- Inter-AS routing: BGP (Border Gateway Protocol) BGP exchanges reachability info, not shortest paths, and prefers policy-based routing (e.g., cost, contracts, peering).

#### 6. Congestion Control

Even with good routes, we can't flood links. Congestion control ensures fair and efficient use of bandwidth.

Implemented primarily at the transport layer (TCP).

## TCP Congestion Control

Key components:

- Additive Increase, Multiplicative Decrease (AIMD)- Slow Start: probe capacity- Congestion Avoidance: grow cautiously- Fast Retransmit / Recovery Modern variants:
- TCP Reno: classic AIMD- TCP Cubic: non-linear growth for high-speed networks- BBR (Bottleneck Bandwidth + RTT): model-based control ##### Algorithm Sketch (AIMD)

```
On ACK: cwnd += 1/cwnd // increase slowly  
On loss: cwnd /= 2      // halve window
```

## 7. Queue Management

Routers maintain queues. Too full? => Packet loss, latency spikes, tail drop.

Solutions:

- RED (Random Early Detection) - drop packets early- CoDel (Controlled Delay) - monitor queue delay, drop adaptively These prevent bufferbloat, improving latency for real-time traffic.

## 8. Flow Control vs Congestion Control

- Flow Control: prevent sender from overwhelming receiver- Congestion Control: prevent sender from overwhelming network TCP uses both: receive window (rwnd) and congestion window (cwnd). Actual sending rate =  $\min(rwnd, cwnd)$ .

## 9. Data Plane vs Control Plane

- Control Plane: decides routes (OSPF, BGP)- Data Plane: forwards packets (fast path) Modern networking (e.g. SDN, Software Defined Networking) separates these:
- Controller computes routes- Switches act on flow rules

## 10. Why It Matters

Routing and congestion control shape the performance of:

- The Internet backbone- Data center networks (with load balancing)- Cloud services and microservice meshes- Content delivery networks (CDNs) Every packet's journey, from your laptop to a global data center, relies on these ideas.

“Networking is not magic. It’s algorithms moving data through time and space.”

### Try It Yourself

1. Implement Dijkstra's algorithm for a small network graph.
2. Simulate RIP (Distance Vector): each node updates from neighbors.
3. Model TCP AIMD window growth; visualize with Python.
4. Try RED: drop packets when queue length > threshold.
5. Compare TCP Reno, Cubic, BBR throughput in simulation.

## 86. Distributed Consensus (Paxos, Raft, PBFT)

In a distributed system, multiple nodes must agree on a single value, for example, the state of a log, a database entry, or a blockchain block. This agreement process is called consensus.

Consensus algorithms let distributed systems act as one reliable system, even when some nodes fail, crash, or lie (Byzantine faults).

### 1. Why Consensus?

Imagine a cluster managing a shared log (like in databases or Raft). Each node might:

- See different requests,- Fail and recover,- Communicate over unreliable links. We need all non-faulty nodes to agree on the same order of operations.

A valid consensus algorithm must satisfy:

- Agreement: all correct nodes choose the same value- Validity: the chosen value was proposed by a node- Termination: every correct node eventually decides- Fault Tolerance: works despite failures

## 2. The FLP Impossibility

The FLP theorem (Fischer, Lynch, Paterson, 1985) says:

In an asynchronous system with even one faulty process, no deterministic algorithm can guarantee consensus.

So practical algorithms use:

- Randomization, or- Partial synchrony (timeouts, retries)

## 3. Paxos: The Classical Algorithm

Paxos, by Leslie Lamport, is the theoretical foundation for distributed consensus.

It revolves around three roles:

- Proposers: suggest values- Acceptors: vote on proposals- Learners: learn the final decision  
Consensus proceeds in two phases.

### Phase 1 (Prepare)

1. Proposer picks a proposal number  $n$  and sends (`Prepare, n`) to acceptors.
2. Acceptors respond with their highest accepted proposal (if any).

### Phase 2 (Accept)

1. If proposer receives a majority of responses, it sends (`Accept, n, v`) with value  $v$  (highest seen or new).
2. Acceptors accept if they haven't promised higher  $n$ .

When a majority accept, value  $v$  is chosen.

### Guarantees

- Safety: no two different values chosen- Liveness: possible under stable leadership ##### Drawbacks
- Complex to implement correctly- High messaging overhead > "Paxos is for theorists; Raft is for engineers."

## 4. Raft: Understandable Consensus

Raft was designed to be simpler and more practical than Paxos, focusing on replicated logs.

### Roles

- Leader: coordinates all changes- Followers: replicate leader's log- Candidates: during elections ##### Workflow

#### 1. Leader Election

- Timeout triggers candidate election. - Each follower votes; majority wins.
- Leader appends entries, sends `AppendEntries` RPCs. - Followers acknowledge; leader commits when majority ack.
- Logs are consistent across majority. - Followers accept only valid prefixes. Raft ensures:
- At most one leader per term- Committed entries never lost- Logs stay consistent ##### Pseudocode Sketch

```
on timeout -> become_candidate()
send RequestVote(term, id)
if majority_votes -> become_leader()

on AppendEntries(term, entries):
    if term >= current_term:
        append(entries)
        reply success
```

## 5. PBFT: Byzantine Fault Tolerance

Paxos and Raft assume crash faults (nodes stop, not lie). For Byzantine faults (arbitrary behavior), we use PBFT (Practical Byzantine Fault Tolerance).

Tolerates up to  $f$  faulty nodes out of  $3f + 1$  total.

## **Phases**

1. Pre-Prepare: Leader proposes value
2. Prepare: Nodes broadcast proposal hashes
3. Commit: Nodes confirm receipt by  $2f+1$  votes

Used in blockchains and critical systems (space, finance).

## **6. Quorum Concept**

Consensus often relies on quorums (majorities):

- Two quorums always intersect, ensuring consistency.- Write quorum + read quorum total nodes. In Raft/Paxos:
- Majority =  $N/2 + 1$ - Guarantees overlap even if some nodes fail.

## **7. Log Replication and State Machines**

Consensus underlies Replicated State Machines (RSM):

- Every node applies the same commands in the same order.- Guarantees deterministic, identical states. This model powers:
- Databases (etcd, Spanner, TiKV)- Coordination systems (ZooKeeper, Consul)- Kubernetes control planes

## **8. Leader Election**

All practical consensus systems need leaders:

- Simplifies coordination- Reduces conflicts- Heartbeats detect failures- New elections restore progress Algorithms:
- Raft Election (random timeouts)- Bully Algorithm- Chang-Roberts Ring Election

## **9. Performance and Optimization**

- Batching: amortize RPC overhead- Pipeline: parallelize appends- Read-only optimizations: serve from followers (stale reads)- Witness nodes: participate in quorum without full data Advanced:
- Multi-Paxos: reuse leader, fewer rounds- Fast Paxos: shortcut some phases- Viewstamped Replication: Paxos-like log replication

## **10. Why It Matters**

Consensus is the backbone of reliability in modern distributed systems. Every consistent database, service registry, or blockchain depends on it.

Systems using consensus:

- etcd, Consul, ZooKeeper - cluster coordination- Raft in Kubernetes - leader election- PBFT in blockchains - fault-tolerant ledgers- Spanner, TiDB - consistent databases > “Consensus is how machines learn to agree, and trust.”

### **Try It Yourself**

1. Implement Raft leader election in C or Python.
2. Simulate Paxos on 5 nodes with message drops.
3. Explore PBFT: try failing nodes and Byzantine behavior.
4. Compare performance of Raft vs Paxos under load.
5. Build a replicated key-value store with Raft.

## **87. Load Balancing and Rate Limiting**

When systems scale, no single server can handle all requests alone. Load balancing distributes incoming traffic across multiple servers to improve throughput, reduce latency, and prevent overload. Meanwhile, rate limiting protects systems by controlling how often requests are allowed, ensuring fairness, stability, and security.

These two ideas, spreading the load and controlling the flow, are cornerstones of modern distributed systems and APIs.

### **1. Why Load Balancing Matters**

Imagine a web service receiving thousands of requests per second. If every request went to one machine, it would crash. A load balancer (LB) acts as a traffic director, spreading requests across many backends.

Goals:

- Efficiency - fully utilize servers- Reliability - no single point of failure- Scalability - handle growing workloads- Flexibility - add/remove servers dynamically

## **2. Types of Load Balancers**

### **1. Layer 4 (Transport Layer)**

Balances based on IP and port. Fast and protocol-agnostic (works for TCP/UDP).

Example: Linux IPVS, Envoy, HAProxy

### **2. Layer 7 (Application Layer)**

Understands protocols like HTTP. Can route by URL path, headers, cookies.

Example: Nginx, Envoy, AWS ALB

## **3. Load Balancing Algorithms**

### **Round Robin**

Cycles through backends in order.

```
Req1 → ServerA  
Req2 → ServerB  
Req3 → ServerC
```

Simple, fair (if all nodes equal).

### **Weighted Round Robin**

Assigns weights to reflect capacity. Example: ServerA(2x), ServerB(1x)

### **Least Connections**

Send request to server with fewest active connections.

### **Least Response Time**

Select backend with lowest latency (monitored dynamically).

## **Hash-Based (Consistent Hashing)**

Deterministically route based on request key (like user ID).

- Keeps cache locality- Used in CDNs, distributed caches (e.g. memcached) ##### Random

Pick a random backend, surprisingly effective under uniform load.

## **4. Consistent Hashing (In Depth)**

Used for sharding and sticky sessions.

Key idea:

- Map servers to a hash ring- A request's key is hashed onto the ring- Assigned to next clockwise server When servers join/leave, only small fraction of keys move.

Used in:

- CDNs- Distributed caches (Redis Cluster, DynamoDB)- Load-aware systems

## **5. Health Checks and Failover**

A smart LB monitors health of each server:

- Heartbeat pings (HTTP/TCP)- Auto-remove unhealthy servers- Rebalance traffic instantly Example: If ServerB fails, remove from rotation:

Healthy: [ServerA, ServerC]

Also supports active-passive failover: hot standby servers take over when active fails.

## **6. Global Load Balancing**

Across regions or data centers:

- GeoDNS: route to nearest region- Anycast: advertise same IP globally; routing picks nearest- Latency-based routing: measure and pick lowest RTT Used by CDNs, cloud services, multi-region apps

## 7. Rate Limiting: The Other Side

If load balancing spreads the work, rate limiting keeps total work reasonable.

It prevents:

- Abuse (bots, DDoS)- Overload (too many requests)- Fairness issues (no user dominates resources) Policies:
- Per-user, per-IP, per-API-key- Global or per-endpoint

## 8. Rate Limiting Algorithms

### Token Bucket

- Bucket holds tokens (capacity = burst limit)- Each request consumes 1 token- Tokens refill at constant rate (rate limit)- If empty → reject or delay Good for bursty traffic.

```
if (tokens > 0) {  
    tokens--;  
    allow();  
} else reject();
```

### Leaky Bucket

- Requests flow into a bucket, drain at fixed rate- Excess = overflow = dropped Smooths bursts; used for shaping.

### Fixed Window Counter

- Count requests in fixed interval (e.g. 1s)- Reset every window- Simple but unfair around boundaries ##### Sliding Window Log / Sliding Window Counter
- Maintain timestamps of requests- Remove old ones beyond time window- More accurate and fair

## **9. Combining Both**

A full system might:

- Use rate limiting per user or service- Use load balancing across nodes- Apply circuit breakers when overload persists Together, they form resilient architectures that stay online even under spikes.

## **10. Why It Matters**

These techniques make large-scale systems:

- Scalable - handle millions of users- Stable - prevent cascading failures- Fair - each client gets a fair share- Resilient - recover gracefully from spikes or node loss Used in:
- API Gateways (Kong, Envoy, Nginx)- Cloud Load Balancers (AWS ALB, GCP LB)- Kubernetes Ingress and Service Meshes- Distributed Caches and Databases > “Balance keeps systems alive. Limits keep them sane.”

## **Try It Yourself**

1. Simulate Round Robin and Least Connections balancing across 3 servers.
2. Implement a Token Bucket rate limiter in C or Python.
3. Test burst traffic, observe drops or delays.
4. Combine Consistent Hashing with Token Bucket for user-level control.
5. Visualize how load balancing + rate limiting keep system latency low.

## **88. Search and Indexing (Inverted, BM25, WAND)**

Search engines, whether web-scale like Google or local like SQLite’s FTS, rely on efficient indexing and ranking to answer queries fast. Instead of scanning all documents, they use indexes (structured lookup tables) to quickly find relevant matches.

This section explores inverted indexes, ranking algorithms (TF-IDF, BM25), and efficient retrieval techniques like WAND.

## 1. The Search Problem

Given:

- A corpus of documents- A query (e.g., “machine learning algorithms”) We want to return:
- Relevant documents- Ranked by importance and similarity Naive search →  $O(N \times M)$  comparisons Inverted indexes →  $O(K \log N)$ , where K = terms in query

## 2. Inverted Index: The Heart of Search

An inverted index maps terms → documents containing them.

### Example

Term	Postings List
“data”	[1, 4, 5]
“algorithm”	[2, 3, 5]
“machine”	[1, 2]

Each posting may include:

- docID- term frequency (tf)- positions (for phrase search) ##### Construction Steps
  1. Tokenize documents → words
  2. Normalize (lowercase, stemming, stopword removal)
  3. Build postings: term → [docIDs, tf, positions]
  4. Sort & compress for storage efficiency

Used by:

- Elasticsearch, Lucene, Whoosh, Solr

### 3. Boolean Retrieval

Simplest model:

- Query = Boolean expression e.g. (machine AND learning) OR AI

Use set operations on postings:

- AND → intersection- OR → union- NOT → difference Fast intersection uses merge algorithm on sorted lists.

```
void intersect(int A[], int B[], int n, int m) {  
    int i = 0, j = 0;  
    while (i < n && j < m) {  
        if (A[i] == B[j]) { print(A[i]); i++; j++; }  
        else if (A[i] < B[j]) i++;  
        else j++;  
    }  
}
```

But Boolean search doesn't rank results, so we need scoring models.

### 4. Vector Space Model

Represent documents and queries as term vectors. Each dimension = term weight (tf-idf).

- tf: term frequency in document- idf: inverse document frequency  $idf = \log \frac{N}{df_t}$

Cosine similarity measures relevance:

$$\text{score}(q, d) = \frac{q \cdot d}{|q||d|}$$

Simple, interpretable, forms basis of BM25 and modern embeddings.

## 5. BM25: The Classic Ranking Function

BM25 (Best Match 25) is the de facto standard in information retrieval.

$$\text{score}(q, d) = \sum_{t \in q} \text{IDF}(t) \cdot \frac{f(t, d) \cdot (k_1 + 1)}{f(t, d) + k_1 \cdot (1 - b + b \cdot \frac{|d|}{\text{avgdl}})}$$

Where:

- ( $f(t, d)$ ): term frequency- ( $|d|$ ): doc length- ( $\text{avgdl}$ ): average doc length-  $k_1, b$ : tunable params (typ. 1.2-2.0, 0.75) ##### Advantages
- Balances term frequency, document length, and rarity- Fast and effective baseline- Still used in Elasticsearch, Lucene, OpenSearch

## 6. Efficiency Tricks: WAND, Block-Max WAND

Ranking involves merging multiple postings. We can skip irrelevant documents early with WAND (Weak AND).

### WAND Principle

- Each term has upper-bound score- Maintain pointers in each posting- Compute potential max score- If max < current threshold, skip doc Improves latency for top-k retrieval.

Variants:

- BMW (Block-Max WAND) - uses block-level score bounds- MaxScore - simpler thresholding- Dynamic pruning - skip unpromising candidates

## 7. Index Compression

Postings lists are long, compression is crucial.

Common schemes:

- Delta encoding: store gaps between docIDs- Variable-byte (VB) or Gamma coding- Frame of Reference (FOR) and SIMD-BP128 for vectorized decoding Goal: smaller storage + faster decompression

## **8. Advanced Retrieval**

### **Proximity Search**

Require words appear near each other. Use positional indexes.

### **Phrase Search**

Match exact sequences using positions: “machine learning” “learning machine”

### **Fuzzy / Approximate Search**

Allow typos: Use Levenshtein automata, n-grams, or k-approximate matching

### **Fielded Search**

Score per field (title, body, tags) Weighted combination

## **9. Learning-to-Rank and Semantic Search**

Modern search adds ML-based re-ranking:

- Learning to Rank (LTR): use features (tf, idf, BM25, clicks)- Neural re-ranking: BERT-style embeddings for semantic similarity- Hybrid retrieval: combine BM25 + dense vectors (e.g. ColBERT, RRF) Also: ANN (Approximate Nearest Neighbor) for vector-based search.

## **10. Why It Matters**

Efficient search powers:

- Web search engines- IDE symbol lookup- Log search, code search- Database full-text search- AI retrieval pipelines (RAG) It's where algorithms meet language and scale.

“Search is how we connect meaning to memory.”

## Try It Yourself

1. Build a tiny inverted index in C or Python.
2. Implement Boolean AND and OR queries.
3. Compute TF-IDF and BM25 scores for a toy dataset.
4. Add WAND pruning for top-k retrieval.
5. Compare BM25 vs semantic embeddings for relevance.

## 89. Compression and Encoding in Systems

Compression and encoding algorithms are the quiet workhorses of computing, shrinking data to save space, bandwidth, and time. They allow systems to store more, transmit faster, and process efficiently. From files and databases to networks and logs, compression shapes nearly every layer of system design.

### 1. Why Compression Matters

Compression is everywhere:

- Databases - column stores, indexes, logs- Networks - HTTP, TCP, QUIC payloads- File systems - ZFS, NTFS, btrfs compression- Streaming - video/audio codecs- Logs & telemetry - reduce I/O and storage cost Benefits:
- Smaller data = faster I/O- Less storage = lower cost- Less transfer = higher throughput Trade-offs:
- CPU overhead (compression/decompression)- Latency (especially for small data)- Suitability (depends on entropy and structure)

### 2. Key Concepts

#### Entropy

Minimum bits needed to represent data (Shannon). High entropy → less compressible.

#### Redundancy

Compression exploits repetition and patterns.

## **Lossless vs Lossy**

- Lossless: reversible (ZIP, PNG, LZ4)- Lossy: approximate (JPEG, MP3, H.264) In system contexts, lossless dominates.

### **3. Common Lossless Compression Families**

#### **Huffman Coding**

- Prefix-free variable-length codes- Frequent symbols = short codes- Optimal under symbol-level model Used in: DEFLATE, JPEG, MP3

#### **Arithmetic Coding**

- Encodes sequence as fractional interval- More efficient than Huffman for skewed distributions- Used in: H.264, bzip2, AV1 ##### Dictionary-Based (LZ77, LZ78)
- Replace repeated substrings with references- Core of ZIP, gzip, zlib, LZMA, Snappy ##### LZ77 Sketch

```
while (not EOF) {  
    find longest match in sliding window;  
    output (offset, length, next_char);  
}
```

Variants:

- LZ4 - fast, lower ratio- Snappy - optimized for speed- Zstandard (Zstd) - tunable speed/ratio, dictionary support ##### Burrows-Wheeler Transform (BWT)
- Reorders data to group similar symbols- Followed by Move-To-Front + Huffman- Used in bzip2, BWT-based compressors ##### Run-Length Encoding (RLE)
- Replace consecutive repeats with (symbol, count)- Great for structured or sparse data Example: AAAAABBBC → (A,5)(B,3)(C,2)

## **4. Specialized Compression in Systems**

### **Columnar Databases**

Compress per column:

- Dictionary encoding - map strings → ints- Run-length encoding - good for sorted columns- Delta encoding - store differences (time series)- Bit-packing - fixed-width integers in minimal bits Combine multiple for optimal ratio.

Example (time deltas):

[100, 102, 103, 107] → [100, +2, +1, +4]

### **Log and Telemetry Compression**

- Structured formats → fieldwise encoding- Often Snappy or LZ4 for fast decode- Aggregators (Fluentd, Loki, Kafka) rely heavily on them ##### Data Lakes and Files
- Parquet, ORC, Arrow → columnar + compressed- Choose codec per column: LZ4 for speed, Zstd for ratio

## **5. Streaming and Chunked Compression**

Large data often processed in chunks:

- Enables random access and parallelism- Needed for network streams, distributed files Example: `zlib` block, `Zstd` frame, `gzip` chunk

Used in:

- HTTP chunked encoding- Kafka log segments- MapReduce shuffle

## **6. Encoding Schemes**

Compression encoding. Encoding ensures safe transport.

## Base64

- Maps 3 bytes → 4 chars- 33% overhead- Used for binary → text (emails, JSON APIs)  
#### URL Encoding
- Escape unsafe chars with %xx ##### Delta Encoding
- Store differences, not full values ##### Varint / Zigzag Encoding
- Compact integers (e.g. protobufs)- Smaller numbers → fewer bytes Example:

```
while (x >= 0x80) { emit((x & 0x7F) | 0x80); x >>= 7; }
emit(x);
```

## 7. Adaptive and Context Models

Modern compressors adapt to local patterns:

- PPM (Prediction by Partial Matching)- Context mixing (PAQ)- Zstd uses FSE (Finite State Entropy) coding Balance between speed, memory, and compression ratio.

## 8. Hardware Acceleration

Compression can be offloaded to:

- CPUs with SIMD (AVX2, SSE4.2)- GPUs (parallel encode/decode)- NICs / SmartNICs- ASICs (e.g., Intel QAT) Critical for high-throughput databases, network appliances, storage systems.

## 9. Design Trade-offs

Goal	Best Choice
Max speed	LZ4, Snappy
Max ratio	Zstd, LZMA
Balance	Zstd (tunable)
Column store	RLE, Delta, Dict
Logs / telemetry	Snappy, LZ4
Archival	bzip2, xz
Real-time	LZ4, Brotli (fast mode)

Choose based on CPU budget, I/O cost, latency tolerance.

## **10. Why It Matters**

Compression is a first-class optimization:

- Saves petabytes in data centers- Boosts throughput across networks- Powers cloud storage (S3, BigQuery, Snowflake)- Enables efficient analytics and ML pipelines > “Every byte saved is time earned.”

### **Try It Yourself**

1. Compress text using Huffman coding (build frequency table).
2. Compare gzip, Snappy, and Zstd on a 1GB dataset.
3. Implement delta encoding and RLE for numeric data.
4. Try dictionary encoding on repetitive strings.
5. Measure compression ratio, speed, and CPU usage trade-offs.

## **90. Fault Tolerance and Replication**

Modern systems must survive hardware crashes, network partitions, or data loss without stopping. Fault tolerance ensures that a system continues to function, even when parts fail. Replication underpins this resilience, duplicating data or computation across multiple nodes for redundancy, performance, and consistency.

Together, they form the backbone of reliability in distributed systems.

### **1. Why Fault Tolerance?**

No system is perfect:

- Servers crash- Disks fail- Networks partition- Power goes out The question isn’t *if* failure happens, but *when*. Fault-tolerant systems detect, contain, and recover from failure automatically.

Goals:

- Availability - keep serving requests- Durability - never lose data- Consistency - stay correct across replicas

## **2. Failure Models**

### **Crash Faults**

Node stops responding but doesn't misbehave. Handled by restarts or replication (Raft, Paxos).

### **Omission Faults**

Lost messages or dropped updates. Handled with retries and acknowledgments.

### **Byzantine Faults**

Arbitrary/malicious behavior. Handled by Byzantine Fault Tolerance (PBFT), expensive but robust.

## **3. Redundancy: The Core Strategy**

Fault tolerance = redundancy + detection + recovery

Redundancy types:

- Hardware: multiple power supplies, disks (RAID)- Software: replicated services, retries- Data: multiple copies, erasure codes- Temporal: retry or checkpoint and replay

## **4. Replication Models**

### **1. Active Replication**

All replicas process requests in parallel (lockstep). Results must match. Used in real-time and Byzantine systems.

### **2. Passive (Primary-Backup)**

One leader (primary) handles requests. Backups replicate log, take over on failure. Used in Raft, ZooKeeper, PostgreSQL streaming.

### **3. Quorum Replication**

Writes and reads contact majority of replicas. Ensures overlap → consistency. Used in Cassandra, DynamoDB, Etcd.

### **5. Consistency Models**

Replication introduces a trade-off between consistency and availability (CAP theorem).

#### **Strong Consistency**

All clients see the same value immediately. Example: Raft, Etcd, Spanner.

#### **Eventual Consistency**

Replicas converge over time. Example: DynamoDB, Cassandra.

#### **Causal Consistency**

Preserves causal order of events. Example: Vector clocks, CRDTs.

Choice depends on workload:

- Banking → strong- Social feeds → eventual- Collaborative editing → causal

### **6. Checkpointing and Recovery**

To recover after crash:

- Periodically checkpoint state- On restart, replay log of missed events Example: Databases → Write-Ahead Log (WAL) Stream systems → Kafka checkpoints

- ```
1. Save state to disk
2. Record latest log position
3. On restart + reload + replay
```

## 7. Erasure Coding

Instead of full copies, store encoded fragments. With ( k ) data blocks, ( m ) parity blocks → tolerate ( m ) failures.

Example: Reed-Solomon (used in HDFS, Ceph)

| k | m | Total | Fault Tolerance |
|---|---|-------|-----------------|
| 4 | 2 | 6     | 2 failures      |

Better storage efficiency than 3× replication.

## 8. Failure Detection

Detecting failure is tricky in distributed systems (because of latency). Common techniques:

- Heartbeats - periodic “I’m alive” messages- Timeouts - suspect node if no heartbeat- Gossip protocols - share failure info among peers Used in Consul, Cassandra, Kubernetes health checks.

## 9. Self-Healing Systems

After failure:

1. Detect it
2. Isolate faulty component
3. Replace or restart
4. Rebalance load or re-replicate data

Patterns:

- Supervisor trees (Erlang/Elixir)- Self-healing clusters (Kubernetes)- Rebalancing (Cassandra ring repair) “Never trust a single machine, trust the system.”

## 10. Why It Matters

Fault tolerance turns fragile infrastructure into reliable services.

Used in:

- Databases (replication + WAL)- Distributed storage (HDFS, Ceph, S3)- Orchestration (Kubernetes controllers)- Streaming systems (Kafka, Flink) Without replication and fault tolerance, large-scale systems would collapse under failure.

“Resilience is built, not assumed.”

### Try It Yourself

1. Build a primary-backup key-value store: leader writes, follower replicates.
2. Add heartbeat + timeout detection to trigger failover.
3. Simulate partition: explore behavior under strong vs eventual consistency.
4. Implement checkpoint + replay recovery for a small app.
5. Compare 3× replication vs Reed-Solomon (4+2) for space and reliability.

# Chapter 10. AI, ML and Optimization

## 91. Classical ML (k-means, Naive Bayes, SVM, Decision Trees)

Classical machine learning is built on interpretable mathematics and solid optimization foundations. Long before deep learning, these algorithms powered search engines, spam filters, and recommendation systems. They’re still used today, fast, explainable, and easy to deploy.

This section covers the four pillars of classical ML:

- k-means - unsupervised clustering- Naive Bayes - probabilistic classification- SVM - margin-based classification- Decision Trees - rule-based learning

### 1. The Essence of Classical ML

Classical ML is about learning from data using statistical principles, often without huge compute. Given dataset ( $D = \{x_i, y_i\}$ ), the task is to:

- Predict ( $y$ ) from ( $x$ )- Generalize beyond seen data- Balance bias and variance

## 2. k-means Clustering

Goal: partition data into ( k ) groups (clusters) such that intra-cluster distance is minimized.

### Objective

$$\min_C \sum_{i=1}^k \sum_{x \in C_i} |x - \mu_i|^2$$

Where  $\mu_i$  = centroid of cluster ( i ).

### Algorithm

1. Choose ( k ) random centroids
2. Assign each point to nearest centroid
3. Recompute centroids
4. Repeat until stable

### Tiny Code (C-style)

```
for (iter = 0; iter < max_iter; iter++) {  
    assign_points_to_clusters();  
    recompute_centroids();  
}
```

### Pros

- Simple, fast (( O(nkd) ))- Works well for spherical clusters ##### Cons
- Requires ( k )- Sensitive to initialization, outliers Variants:
- k-means++ (better initialization)- Mini-batch k-means (scalable)

## 3. Naive Bayes Classifier

A probabilistic model using Bayes' theorem under independence assumptions.

$$P(y|x) \propto P(y) \prod_{i=1}^n P(x_i|y)$$

## **Algorithm**

1. Compute prior (  $P(y)$  )
2. Compute likelihood (  $P_{x_i|y}$  )
3. Predict class with max posterior

## **Types**

- Multinomial NB - text (bag of words)- Gaussian NB - continuous features- Bernoulli NB - binary features ##### Example (Spam Detection)

$P(\text{spam} \mid \text{"win money"}) = P(\text{spam}) * P(\text{"win"} \mid \text{spam}) * P(\text{"money"} \mid \text{spam})$

## **Pros**

- Fast, works well for text- Needs little data- Probabilistic interpretation ##### Cons
- Assumes feature independence- Poor for correlated features

## **4. Support Vector Machines (SVM)**

SVM finds the max-margin hyperplane separating classes.

### **Objective**

Maximize margin = distance between boundary and nearest points.

$$\min_{w,b} \frac{1}{2} |w|^2 \quad \text{s.t.} \quad y_i(w \cdot x_i + b) \geq 1$$

Can be solved via Quadratic Programming.

## Intuition

- Each data point → vector- Hyperplane:  $w \cdot x + b = 0$ - Support vectors = boundary points  
#### Kernel Trick

Transform input via kernel (  $Kx_i, x_j = x_i \cdot x_j$  ):

- Linear: dot product- Polynomial:  $(x_i \cdot x_j + c^T d)$ - RBF:  $e^{-\gamma|x_i - x_j|^2}$  #### Pros
- Effective in high dimensions- Can model nonlinear boundaries- Few hyperparameters  
#### Cons
- Slow on large data- Harder to tune kernel parameters

## 5. Decision Trees

If-else structure for classification/regression.

At each node:

- Pick feature (  $f$  ) and threshold (  $t$  )- Split to maximize information gain #### Metrics
- Entropy:  $H = -\sum p_i \log p_i$ - Gini:  $G = 1 - \sum p_i^2$  #### Pseudocode

```
if (feature < threshold)
    go left;
else
    go right;
```

Build recursively until:

- Max depth- Min samples per leaf- Pure nodes #### Pros
- Interpretable- Handles mixed data- No scaling needed #### Cons
- Prone to overfitting- Unstable (small data changes) Fixes:
- Pruning (reduce depth)- Ensembles: Random Forests, Gradient Boosting

| Algorithm | Bias | Variance |
|-----------|------|----------|
|-----------|------|----------|

## 6. Bias-Variance Tradeoff

| Algorithm     | Bias | Variance |
|---------------|------|----------|
| k-means       | High | Low      |
| Naive Bayes   | High | Low      |
| SVM           | Low  | Medium   |
| Decision Tree | Low  | High     |

Balancing both = good generalization.

## 7. Evaluation Metrics

For classification:

- Accuracy, Precision, Recall, F1-score- ROC-AUC, Confusion Matrix For clustering:
- Inertia, Silhouette Score Always use train/test split or cross-validation.

## 8. Scaling to Large Data

Techniques:

- Mini-batch training- Online updates (SGD)- Dimensionality reduction (PCA)- Approximation (Random Projections) Libraries:
- scikit-learn (Python)- liblinear, libsvm (C/C++)- MLlib (Spark)

## 9. When to Use What

| Task                     | Recommended Algorithm    |
|--------------------------|--------------------------|
| Text classification      | Naive Bayes              |
| Clustering               | k-means                  |
| Nonlinear classification | SVM (RBF)                |
| Tabular data             | Decision Tree            |
| Quick baseline           | Logistic Regression / NB |

## 10. Why It Matters

These algorithms are fast, interpretable, and theoretical foundations of modern ML. They remain the go-to choice for:

- Small to medium datasets- Real-time classification- Explainable AI > “Classical ML is the art of solving problems with math you can still write on a whiteboard.”

### Try It Yourself

1. Cluster 2D points with k-means, plot centroids.
2. Train Naive Bayes on a spam/ham dataset.
3. Classify linearly separable data with SVM.
4. Build a Decision Tree from scratch (entropy, Gini).
5. Compare models’ accuracy and interpretability.

## 92. Ensemble Methods (Bagging, Boosting, Random Forests)

Ensemble methods combine multiple weak learners to build a strong predictor. Instead of relying on one model, ensembles vote, average, or boost multiple models, improving stability and accuracy.

They are the bridge between classical and modern ML , simple models, combined smartly, become powerful.

### 1. The Core Idea

“Many weak learners, when combined, can outperform a single strong one.”

Mathematically, if  $f_1, f_2, \dots, f_k$  are weak learners, an ensemble predictor is:

$$F(x) = \frac{1}{k} \sum_{i=1}^k f_i(x)$$

For classification, combine via majority vote. For regression, combine via average.

### 2. Bagging (Bootstrap Aggregating)

Bagging reduces variance by training models on different samples.

## Steps

1. Draw ( B ) bootstrap samples from dataset ( D ).
2. Train one model per sample.
3. Aggregate predictions by averaging or voting.

$$\hat{f} * \text{bag}(x) = \frac{1}{B} \sum *b = 1^B f_b(x)$$

Each  $f_b$  is trained on a random subset (with replacement).

## Example

- Base learner: Decision Tree- Ensemble: Bagged Trees- Famous instance: Random Forest ##### Tiny Code (C-style Pseudocode)

```
for (int b = 0; b < B; b++) {  
    D_b = bootstrap_sample(D);  
    model[b] = train_tree(D_b);  
}  
prediction = average_predictions(model, x);
```

## Pros

- Reduces variance- Works well with high-variance learners- Parallelizable ##### Cons
- Increases computation- Doesn't reduce bias

## 3. Random Forest

A bagging-based ensemble of decision trees with feature randomness.

## Key Ideas

- Each tree trained on bootstrap sample.- At each split, consider random subset of features.- Final prediction = majority vote or average. This decorrelates trees, improving generalization.

$$F(x) = \frac{1}{B} \sum_{b=1}^B T_b(x)$$

## Pros

- Handles large feature sets- Low overfitting- Good default for tabular data ##### Cons
- Less interpretable- Slower on huge datasets OOB (Out-of-Bag) error = internal validation from unused samples.

## 4. Boosting

Boosting focuses on reducing bias by sequentially training models , each one corrects errors from the previous.

### Steps

1. Start with weak learner (  $f_1(x)$  )
2. Train next learner (  $f_2(x)$  ) on residuals/errors
3. Combine with weighted sum

$$F_m(x) = F_{m-1}(x) + \alpha_m f_m(x)$$

Weights  $\alpha_m$  focus on difficult examples.

### Tiny Code (Conceptual)

```
F = 0;
for (int m = 0; m < M; m++) {
    residual = y - predict(F, x);
    f_m = train_weak_learner(x, residual);
    F += alpha[m] * f_m;
}
```

## 5. AdaBoost (Adaptive Boosting)

AdaBoost adapts weights on samples after each iteration.

## Algorithm

1. Initialize weights:  $w_i = \frac{1}{n}$

2. Train weak classifier  $f_t$

3. Compute error:  $\epsilon_t$

4. Update weights:

$$w_i \leftarrow w_i \cdot e^{\alpha_t \cdot I(y_i \neq f_t(x_i))}$$

where  $\alpha_t = \frac{1}{2} \ln \left( \frac{1-\epsilon_t}{\epsilon_t} \right)$

5. Normalize weights

Final classifier:

$$F(x) = \text{sign} \left( \sum_t \alpha_t f_t(x) \right)$$

## Pros

- High accuracy on clean data- Simple and interpretable weights ##### Cons
- Sensitive to outliers- Sequential → not easily parallelizable

## 6. Gradient Boosting

A modern version of boosting using gradient descent on loss.

At each step, fit new model to negative gradient of loss function.

## Objective

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x)$$

where  $h_m(x) \approx -\frac{\partial L(y, F(x))}{\partial F(x)}$

## Common Libraries

- XGBoost
- LightGBM
- CatBoost #### Pros
- High performance on tabular data- Flexible (custom loss)- Handles mixed feature types #### Cons
- Slower to train- Sensitive to hyperparameters

## 7. Stacking (Stacked Generalization)

Combine multiple models (base learners) via a meta-model.

### Steps

1. Train base models (SVM, Tree, NB, etc.)
2. Collect their predictions
3. Train meta-model (e.g. Logistic Regression) on outputs

$$\hat{y} = f_{meta}(f_1(x), f_2(x), \dots, f_k(x))$$

## 8. Bagging vs Boosting

| Feature   | Bagging         | Boosting          |
|-----------|-----------------|-------------------|
| Strategy  | Parallel        | Sequential        |
| Goal      | Reduce variance | Reduce bias       |
| Weighting | Uniform         | Adaptive          |
| Example   | Random Forest   | AdaBoost, XGBoost |

## 9. Bias-Variance Behavior

- Bagging: ↓ variance- Boosting: ↓ bias- Random Forest: balanced- Stacking: flexible but complex

## 10. Why It Matters

Ensemble methods are the workhorses of classical ML competitions and real-world tabular problems. They blend interpretability, flexibility, and predictive power.

“One tree may fall, but a forest stands strong.”

### Try It Yourself

1. Train a Random Forest on the Iris dataset.
2. Implement AdaBoost from scratch using decision stumps.
3. Compare Bagging vs Boosting accuracy.
4. Try XGBoost with different learning rates.
5. Visualize feature importance across models.

## 93. Gradient Methods (SGD, Adam, RMSProp)

Gradient-based optimization is the heartbeat of machine learning. These methods update parameters iteratively by following the negative gradient of the loss function. They power everything from linear regression to deep neural networks.

### 1. The Core Idea

We want to minimize a loss function ( $L(\theta)$ ). Starting from some initial parameters  $\theta_0$ , we move in the opposite direction of the gradient:

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla_{\theta} L(\theta_t)$$

where  $\eta$  is the learning rate (step size).

The gradient tells us which way the function increases fastest , we move the other way.

### 2. Batch Gradient Descent

Uses the entire dataset to compute the gradient.

$$\nabla_{\theta} L(\theta) = \frac{1}{N} \sum_{i=1}^N \nabla_{\theta} \ell_i(\theta)$$

- Accurate but slow for large ( N )- Each update is expensive Tiny Code

```

for (int t = 0; t < T; t++) {
    grad = compute_full_gradient(data, theta);
    theta = theta - eta * grad;
}

```

Good for: small datasets or convex problems

### 3. Stochastic Gradient Descent (SGD)

Instead of full data, use one random sample per step.

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla_{\theta} \ell_i(\theta_t)$$

- Noisy but faster updates- Can escape local minima- Great for online learning Tiny Code

```

for each sample (x_i, y_i):
    grad = grad_loss(theta, x_i, y_i);
    theta -= eta * grad;

```

Pros

- Fast convergence- Works on large datasets Cons
- Noisy updates- Requires learning rate tuning

### 4. Mini-Batch Gradient Descent

Compromise between batch and stochastic.

Use small subset (mini-batch) of samples:

$$\theta_{t+1} = \theta_t - \eta \cdot \frac{1}{m} \sum_{i=1}^m \nabla_{\theta} \ell_i(\theta_t)$$

Usually batch size = 32 or 64. Faster, more stable updates.

## 5. Momentum

Adds velocity to smooth oscillations.

$$v_t = \beta v_{t-1} + (1 - \beta) \nabla_{\theta} L(\theta_t)$$

$$\theta_{t+1} = \theta_t - \eta v_t$$

This accumulates past gradients to speed movement in consistent directions.

Think of it like a heavy ball rolling down a hill.

## 6. Nesterov Accelerated Gradient (NAG)

Improves momentum by looking ahead:

$$v_t = \beta v_{t-1} + \eta \nabla_{\theta} L(\theta_t - \beta v_{t-1})$$

It anticipates the future position before computing the gradient.

Faster convergence in convex settings.

## 7. RMSProp

Adjusts learning rate per parameter using exponential average of squared gradients:

$$E[g^2] * t = \rho E[g^2] * t - 1 + (1 - \rho) g_t^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

This helps when gradients vary in magnitude.

Good for: non-stationary objectives, deep networks

## 8. Adam (Adaptive Moment Estimation)

Combines momentum + RMSProp:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

Bias-corrected estimates:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Update rule:

$$\theta_{t+1} = \theta_t - \frac{\eta \cdot \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Tiny Code (Conceptual)

```
m = 0; v = 0;
for (int t = 1; t <= T; t++) {
    g = grad(theta);
    m = beta1 * m + (1 - beta1) * g;
    v = beta2 * v + (1 - beta2) * g * g;
    m_hat = m / (1 - pow(beta1, t));
    v_hat = v / (1 - pow(beta2, t));
    theta -= eta * m_hat / (sqrt(v_hat) + eps);
}
```

Pros

- Works well out of the box- Adapts learning rate- Great for deep learning Cons
- May not converge exactly- Needs decay schedule for stability

## 9. Learning Rate Schedules

Control  $\eta$  over time:

- Step decay:  $\eta_t = \eta_0 \cdot \gamma^{\lfloor t/s \rfloor}$ - Exponential decay:  $\eta_t = \eta_0 e^{-\lambda t}$ - Cosine annealing: smooth periodic decay- Warm restarts: reset learning rate periodically

## 10. Why It Matters

All modern deep learning is built on gradients. Choosing the right optimizer can mean faster training and better accuracy.

| Optimizer      | Adaptive | Momentum | Common Use         |
|----------------|----------|----------|--------------------|
| SGD            | No       | Optional | Simple tasks       |
| SGD + Momentum | No       | Yes      | ConvNets           |
| RMSProp        | Yes      | No       | RNNs               |
| Adam           | Yes      | Yes      | Transformers, DNNs |

“Optimization is the art of taking small steps in the right direction , many times over.”

### Try It Yourself

1. Implement SGD and Adam on a linear regression task.
2. Compare training curves for SGD, Momentum, RMSProp, and Adam.
3. Experiment with learning rate schedules.
4. Visualize optimization paths on a 2D contour plot.

## 94. Deep Learning (Backpropagation, Dropout, Normalization)

Deep learning is about stacking layers of computation so that the network can learn hierarchical representations. From raw pixels to abstract features, deep nets build meaning through composition of functions.

At the core of this process are three ideas: backpropagation, regularization (dropout), and normalization.

### 1. The Essence of Deep Learning

A neural network is a chain of functions:

$$f(x; \theta) = f_L(f_{L-1}(\dots f_1(x)))$$

Each layer transforms its input and passes it on.

Training involves finding parameters  $\theta$  that minimize a loss (  $L(fx; \theta, y)$  ).

## 2. Backpropagation

Backpropagation is the algorithm that teaches neural networks.

It uses the chain rule of calculus to efficiently compute gradients layer by layer.

For each layer ( i ):

$$\frac{\partial L}{\partial \theta_i} = \frac{\partial L}{\partial a_i} \cdot \frac{\partial a_i}{\partial \theta_i}$$

and propagate backward:

$$\frac{\partial L}{\partial a_{i-1}} = \frac{\partial L}{\partial a_i} \cdot \frac{\partial a_i}{\partial a_{i-1}}$$

So every neuron learns how much it contributed to the error.

Tiny Code

```
// Pseudocode for 2-layer network
forward:
    z1 = W1*x + b1;
    a1 = relu(z1);
    z2 = W2*a1 + b2;
    y_hat = softmax(z2);
    loss = cross_entropy(y_hat, y);

backward:
    dz2 = y_hat - y;
    dW2 = dz2 * a1.T;
    db2 = sum(dz2);
    da1 = W2.T * dz2;
    dz1 = da1 * relu_grad(z1);
    dW1 = dz1 * x.T;
    db1 = sum(dz1);
```

Each gradient is computed by local differentiation and multiplied back.

## 3. Activation Functions

Nonlinear activations let networks approximate nonlinear functions.

| Function | Formula              | Use                  |
|----------|----------------------|----------------------|
| ReLU     | $\max(0, x)$         | Default, fast        |
| Sigmoid  | $\frac{1}{1+e^{-x}}$ | Probabilities        |
| Tanh     | $\tanh(x)$           | Centered activations |
| GELU     | $x \Phi(x)$          | Modern transformers  |

Without nonlinearity, stacking layers is just one big linear transformation.

#### 4. Dropout

Dropout is a regularization technique that prevents overfitting. During training, randomly turn off neurons:

$$\tilde{a}_i = a_i \cdot m_i, \quad m_i \sim \text{Bernoulli}(p)$$

At inference, scale activations by ( p ) (keep probability).

It forces the network to not rely on any single path.

Tiny Code

```
for (int i = 0; i < n; i++) {
    if (rand_uniform() < p) a[i] = 0;
    else a[i] /= p; // scaling
}
```

#### 5. Normalization

Normalization stabilizes and speeds up training by reducing internal covariate shift.

##### Batch Normalization

Normalize activations per batch:

$$\hat{x} = \frac{x - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$y = \gamma \hat{x} + \beta$$

Learnable parameters  $\gamma, \beta$  restore flexibility.

Benefits:

- Smooth gradients- Allows higher learning rates- Acts as regularizer ##### Layer Normalization

Used in transformers (normalizes across features, not batch).

## 6. Initialization

Proper initialization helps gradients flow.

| Scheme | Formula                            | Use  |
|--------|------------------------------------|------|
| Xavier | $\text{Var}(W) = \frac{1}{n_{in}}$ | Tanh |
| He     | $\text{Var}(W) = \frac{2}{n_{in}}$ | ReLU |

Poor initialization can lead to vanishing or exploding gradients.

## 7. Training Pipeline

1. Initialize weights
2. Forward pass
3. Compute loss
4. Backward pass (backprop)
5. Update weights (e.g. with Adam)

Repeat until convergence.

## 8. Deep Architectures

| Model       | Key Idea              | Typical Use       |
|-------------|-----------------------|-------------------|
| MLP         | Fully connected       | Tabular data      |
| CNN         | Convolutions          | Images            |
| RNN         | Sequential recurrence | Time series, text |
| Transformer | Self-attention        | Language, vision  |

Each architecture stacks linear operations and nonlinearities in different ways.

## 9. Overfitting and Regularization

Common fixes:

- Dropout- Weight decay ( $L_2$  regularization)- Data augmentation- Early stopping The key is to improve generalization, not just minimize training loss.

## 10. Why It Matters

Backpropagation turned neural networks from theory to practice. Normalization made them train faster. Dropout made them generalize better.

Together, they unlocked the deep learning revolution.

“Depth gives power, but gradients give life.”

### Try It Yourself

1. Implement a 2-layer network with ReLU and softmax.
2. Add dropout and batch normalization.
3. Visualize training with and without dropout.
4. Compare performance on MNIST with and without normalization.

## 95. Sequence Models (Viterbi, Beam Search, CTC)

Sequence models process data where order matters, text, speech, DNA, time series. They capture dependencies across positions, predicting the next step from context.

This section explores three fundamental tools: Viterbi, Beam Search, and CTC (Connectionist Temporal Classification).

### 1. The Nature of Sequential Data

Sequential data has temporal or structural order. Each element  $x_t$  depends on past inputs  $x_{1:t-1}$ .

Common sequence tasks:

- Tagging (POS tagging, named entity recognition)- Transcription (speech → text)- Decoding (translation, path reconstruction) To handle such problems, we need models that remember.

## 2. Hidden Markov Models (HMMs)

A Hidden Markov Model assumes:

- A sequence of hidden states  $z_1, z_2, \dots, z_T$ . Each state emits an observation  $x_t$ . Transition and emission probabilities govern the process

$$P(z_t|z_{t-1}) = A_{z_{t-1}, z_t}, \quad P(x_t|z_t) = B_{z_t}(x_t)$$

Goal: find the most likely sequence of hidden states given observations.

## 3. The Viterbi Algorithm

Viterbi is a dynamic programming algorithm to decode the most probable path:

$$\delta_t(i) = \max_{z_{1:t-1}} P(z_{1:t-1}, z_t = i, x_{1:t})$$

Recurrence:

$$\delta_t(i) = \max_j (\delta_{t-1}(j) \cdot A_{j,i}) \cdot B_i(x_t)$$

Track backpointers to reconstruct the best sequence.

Time complexity:  $O(T \cdot N^2)$ ,  
where  $N$  = number of states,  $T$  = sequence length.

Tiny Code

```
for (t = 1; t < T; t++) {
    for (i = 0; i < N; i++) {
        double best = -INF;
        int argmax = -1;
        for (j = 0; j < N; j++) {
            double score = delta[t-1][j] * A[j][i];
            if (score > best) { best = score; argmax = j; }
        }
        delta[t][i] = best * B[i][x[t]];
        backptr[t][i] = argmax;
    }
}
```

Use `backptr` to trace back the optimal path.

## 4. Beam Search

For many sequence models (e.g. neural machine translation), exhaustive search is impossible. Beam search keeps only the top-k best hypotheses at each step.

Algorithm:

1. Start with an empty sequence and score 0
2. At each step, expand each candidate with all possible next tokens
3. Keep only k best sequences (beam size)
4. Stop when all sequences end or reach max length

Beam size controls trade-off:

- Larger beam → better accuracy, slower- Smaller beam → faster, riskier

Tiny Code

```
for (step = 0; step < max_len; step++) {  
    vector<Candidate> new_beam;  
    for (c in beam) {  
        probs = model_next(c.seq);  
        for (token, p in probs)  
            new_beam.push({c.seq + token, c.score + log(p)});  
    }  
    beam = top_k(new_beam, k);  
}
```

Use log probabilities to avoid underflow.

## 5. Connectionist Temporal Classification (CTC)

Used in speech recognition and handwriting recognition where input and output lengths differ.

CTC learns to align input frames with output symbols without explicit alignment.

Add a special blank symbol ( ) to allow flexible alignment.

Example (CTC decoding):

| Frame | Output | After Collapse |
|-------|--------|----------------|
| A     | A A    | A A            |
| H     | H H    | H              |

Loss:

$$P(y|x) = \sum_{\pi \in \text{Align}(x,y)} P(\pi|x)$$

where  $\pi$  are all alignments that reduce to ( y ).

CTC uses dynamic programming to compute forward-backward probabilities.

## 6. Comparing Methods

| Method      | Used In          | Key Idea                 | Handles Alignment? |
|-------------|------------------|--------------------------|--------------------|
| Viterbi     | HMMs             | Most probable state path | Yes                |
| Beam Search | Neural decoders  | Approximate search       | Implicit           |
| CTC         | Speech / seq2seq | Sum over alignments      | Yes                |

## 7. Use Cases

- Viterbi: POS tagging, speech decoding- Beam Search: translation, text generation- CTC: ASR, OCR, gesture recognition

## 8. Implementation Tips

- Use log-space for probabilities- In beam search, apply length normalization- In CTC, use dynamic programming tables- Combine CTC + beam search for speech decoding

## 9. Common Pitfalls

- Viterbi assumes Markov property (limited memory)- Beam Search can miss global optimum- CTC can confuse repeated characters without blanks

## 10. Why It Matters

Sequence models are the bridge between structure and time. They show how to decode hidden meaning in ordered data.

From decoding Morse code to transcribing speech, these algorithms give machines the gift of sequence understanding.

## Try It Yourself

1. Implement Viterbi for a 3-state HMM.
2. Compare greedy decoding vs beam search on a toy language model.
3. Build a CTC loss table for a short sequence (like “HELLO”).

## 96. Metaheuristics (GA, SA, PSO, ACO)

Metaheuristics are general-purpose optimization strategies that search through vast, complex spaces when exact methods are too slow or infeasible. They don't guarantee the perfect answer but often find good-enough solutions fast.

This section covers four classics:

- GA (Genetic Algorithm)- SA (Simulated Annealing)- PSO (Particle Swarm Optimization)- ACO (Ant Colony Optimization)

### 1. The Metaheuristic Philosophy

Metaheuristics draw inspiration from nature and physics. They combine exploration (searching widely) and exploitation (refining promising spots).

They're ideal for:

- NP-hard problems (TSP, scheduling)- Continuous optimization (parameter tuning)- Black-box functions (no gradients) They trade mathematical guarantees for practical power.

### 2. Genetic Algorithm (GA)

Inspired by natural selection, GAs evolve a population of solutions.

#### Core Steps

1. Initialize population randomly
2. Evaluate fitness of each
3. Select parents
4. Crossover to produce offspring
5. Mutate to add variation
6. Replace worst with new candidates

Repeat until convergence.

Tiny Code

```
for (gen = 0; gen < max_gen; gen++) {
    evaluate(pop);
    parents = select_best(pop);
    offspring = crossover(parents);
    mutate(offspring);
    pop = select_survivors(pop, offspring);
}
```

Operators

- Selection: tournament, roulette-wheel- Crossover: one-point, uniform- Mutation: bit-flip, Gaussian Strengths: global search, diverse exploration Weakness: may converge slowly

### 3. Simulated Annealing (SA)

Mimics cooling of metals, start hot (high randomness), slowly cool.

At each step:

1. Propose random neighbor
2. Accept if better
3. If worse, accept with probability

$$P = e^{-\frac{\Delta E}{T}}$$

4. Gradually lower ( T )

Tiny Code

```
T = T_init;
state = random_state();
while (T > T_min) {
    next = neighbor(state);
    dE = cost(next) - cost(state);
    if (dE < 0 || exp(-dE/T) > rand_uniform())
        state = next;
    T *= alpha; // cooling rate
}
```

Strengths: escapes local minima Weakness: sensitive to cooling schedule

## 4. Particle Swarm Optimization (PSO)

Inspired by bird flocking. Each particle adjusts velocity based on:

- Its own best position- The global best found

$$v_i \leftarrow w v_i + c_1 r_1 (p_i - x_i) + c_2 r_2 (g - x_i)$$

$$x_i \leftarrow x_i + v_i$$

Tiny Code

```
for each particle i:  
    v[i] = w*v[i] + c1*r1*(pbest[i]-x[i]) + c2*r2*(gbest-x[i]);  
    x[i] += v[i];  
    update_best(i);
```

Strengths: continuous domains, easy Weakness: premature convergence

## 5. Ant Colony Optimization (ACO)

Inspired by ant foraging, ants deposit pheromones on paths. The stronger the trail, the more likely others follow.

Steps:

1. Initialize pheromone on all edges
2. Each ant builds a solution (prob. pheromone)
3. Evaluate paths
4. Evaporate pheromone
5. Reinforce good paths

$$\tau_{ij} \leftarrow (1 - \rho)\tau_{ij} + \sum_k \Delta\tau_{ij}^k$$

Tiny Code

```
for each iteration:  
    for each ant:  
        path = build_solution(pheromone);  
        score = evaluate(path);  
    evaporate(pheromone);  
    deposit(pheromone, best_paths);
```

Strengths: combinatorial problems (TSP) Weakness: parameter tuning, slower convergence

## 6. Comparing the Four

| Method | Inspiration    | Best For            | Key Idea                       |
|--------|----------------|---------------------|--------------------------------|
| GA     | Evolution      | Discrete search     | Selection, crossover, mutation |
| SA     | Thermodynamics | Local optima escape | Cooling + randomness           |
| PSO    | Swarm behavior | Continuous search   | Local + global attraction      |
| ACO    | Ant foraging   | Graph paths         | Pheromone reinforcement        |

## 7. Design Patterns

Common metaheuristic pattern:

- Represent solution- Define fitness / cost function- Define neighbor / mutation operators- Balance randomness and greediness Tuning parameters often matters more than equations.

## 8. Hybrid Metaheuristics

Combine strengths:

- GA + SA: evolve population, fine-tune locally- PSO + DE: use swarm + differential evolution- ACO + Local Search: reinforce with hill-climbing These hybrids often outperform single methods.

## 9. Common Pitfalls

- Poor representation → weak search- Over-exploitation → stuck in local optima- Bad parameters → chaotic or stagnant behavior Always visualize progress (fitness over time).

## 10. Why It Matters

Metaheuristics give us adaptive intelligence, searching without gradients, equations, or complete knowledge. They reflect nature's way of solving complex puzzles: iterate, adapt, survive.

“Optimization is not about perfection. It’s about progress guided by curiosity.”

## Try It Yourself

1. Implement Simulated Annealing for the Traveling Salesman Problem.
2. Create a Genetic Algorithm for knapsack optimization.
3. Tune PSO parameters to fit a function  $f(x) = x^2 + 10 \sin x$ .
4. Compare ACO paths for TSP at different evaporation rates.

## 97. Reinforcement Learning (Q-learning, Policy Gradients)

Reinforcement Learning (RL) is about learning through interaction , an agent explores an environment, takes actions, and learns from rewards. Unlike supervised learning (where correct labels are given), RL learns what to do by trial and error.

This section introduces two core approaches:

- Q-learning (value-based)- Policy Gradient (policy-based)

### 1. The Reinforcement Learning Setting

An RL problem is modeled as a Markov Decision Process (MDP):

- States  $S$
- Actions  $A$
- Transition  $P(s' | s, a)$
- Reward  $R(s, a)$
- Discount factor  $\gamma$

The agent's goal is to find a policy  $\pi(a | s)$  that maximizes expected return:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

### 2. Value Functions

The value function measures how good a state (or state-action pair) is.

- State-value:

$$V^\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$$

- Action-value (Q-function):

$$Q^\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$$

### 3. Bellman Equation

The Bellman equation relates a state's value to its neighbors:

$$Q^*(s, a) = R(s, a) + \gamma \max_{a'} Q^*(s', a')$$

This recursive definition drives value iteration and Q-learning.

### 4. Q-Learning

Q-learning learns the optimal action-value function off-policy (independent of behavior policy):

Update Rule:

$$Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

Tiny Code

```
Q[s][a] += alpha * (r + gamma * max(Q[s_next]) - Q[s][a]);  
s = s_next;
```

Repeat while exploring (e.g.,  $\varepsilon$ -greedy):

- With probability  $\varepsilon$ , choose a random action
- With probability  $1 - \varepsilon$ , choose the best action

Over time,  $Q$  converges to  $Q^*$ .

### 5. Exploration vs Exploitation

RL is a balancing act:

- Exploration: try new actions to gather knowledge- Exploitation: use current best knowledge to maximize reward Strategies:
  - -greedy- Softmax action selection- Upper Confidence Bound (UCB)

## 6. Policy Gradient Methods

Instead of learning Q-values, learn the policy directly. Represent policy with parameters  $\theta$ :

$$\pi_\theta(a|s) = P(a|s; \theta)$$

Goal: maximize expected return

$$J(\theta) = \mathbb{E} * \pi * \theta[G_t]$$

Gradient ascent update:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$

REINFORCE Algorithm:

$$\nabla_\theta J(\theta) = \mathbb{E}[G_t \nabla_\theta \log \pi_\theta(a_t|s_t)]$$

Tiny Code

```
theta += alpha * G_t * grad_logpi(a_t, s_t);
```

## 7. Actor-Critic Architecture

Combines policy gradient (actor) + value estimation (critic).

- Actor: updates policy- Critic: estimates value (baseline) Update:

$$\theta \leftarrow \theta + \alpha_\theta \delta_t \nabla_\theta \log \pi_\theta(a_t|s_t)$$

$$w \leftarrow w + \alpha_w \delta_t \nabla_w V_w(s_t)$$

with TD error:

$$\delta_t = r + \gamma V(s') - V(s)$$

## 8. Comparing Methods

| Method          | Type         | Learns    | On/Off Policy | Continuous? |
|-----------------|--------------|-----------|---------------|-------------|
| Q-learning      | Value-based  | $Q(s, a)$ | Off-policy    | No          |
| Policy Gradient | Policy-based | $(a   s)$ | On-policy     | Yes         |
| Actor-Critic    | Hybrid       | Both      | On-policy     | Yes         |

## 9. Extensions

- Deep Q-Networks (DQN): use neural nets for  $Q(s, a)$ - PPO / A3C: advanced actor-critic methods- TD( ): tradeoff between MC and TD learning- Double Q-learning: reduce overestimation- Entropy regularization: encourage exploration

## 10. Why It Matters

Reinforcement learning powers autonomous agents, game AIs, and control systems. It's the foundation of AlphaGo, robotics control, and adaptive decision systems.

“An agent learns not from instruction but from experience.”

### Try It Yourself

1. Implement Q-learning for a grid-world maze.
2. Add -greedy exploration.
3. Visualize the learned policy.
4. Try REINFORCE with a simple policy (e.g. softmax over actions).
5. Compare convergence of Q-learning vs Policy Gradient.

## 98. Approximation and Online Algorithms

In the real world, we often can't wait for a perfect solution , data arrives on the fly, or the problem is too hard to solve exactly. That's where approximation and online algorithms shine. They aim for good-enough results, fast and adaptively, under uncertainty.

### 1. The Big Picture

- Approximation algorithms: Solve NP-hard problems with provable bounds.- Online algorithms: Make immediate decisions without knowing the future. Both trade optimality for efficiency or adaptability.

## 2. Approximation Algorithms

An approximation algorithm finds a solution within a factor  $\rho$  of the optimal.

If  $(C)$  is cost of the algorithm, and  $C^*$  is optimal cost:

$$\rho = \max\left(\frac{C}{C^*}, \frac{C^*}{C}\right)$$

Example:  $\rho = 2 \rightarrow$  solution at most twice worse than optimal.

## 3. Example: Vertex Cover

Problem: Given graph ( $G(V,E)$ ), choose smallest set of vertices covering all edges.

Algorithm (2-approximation):

1. Initialize cover =
  2. While edges remain:
    - Pick any edge  $(u, v)$  - Add both  $u, v$  to cover - Remove all edges incident on  $u$  or  $v$
- Guarantee: At most  $2 \times$  optimal size.

Tiny Code

```
cover = {};
while (!edges.empty()) {
    (u, v) = edges.pop();
    cover.add(u);
    cover.add(v);
    remove_incident_edges(u, v);
}
```

## 4. Example: Metric TSP (Triangle Inequality)

Algorithm (Christofides):

1. Find MST
2. Find odd-degree vertices
3. Find min perfect matching
4. Combine + shortcut to get tour

Guarantee:  $1.5 \times$  optimal.

## 5. Greedy Approximation: Set Cover

Goal: Cover universe ( U ) with minimum sets  $S_i$ .

Greedy Algorithm: Pick set covering most uncovered elements each time. Guarantee:  $H_n \approx \ln n$  factor approximation.

## 6. Online Algorithms

Online algorithms must decide now, before future input is known.

Goal: Minimize competitive ratio:

$$CR = \max_{\text{input}} \frac{\text{Cost} * \text{online}}{\text{Cost} * \text{optimal offline}}$$

Lower CR → better adaptability.

## 7. Classic Example: Online Paging

You have k pages in cache, sequence of page requests.

- If page in cache → hit- Else → miss, must evict one page Strategies:
- LRU (Least Recently Used): evict oldest- FIFO: evict first loaded- Random: pick randomly Competitive Ratio:
- LRU: ( k )- Random: ( 2k-1 )

Tiny Code

```
cache = LRUCache(k);
for (page in requests) {
    if (!cache.contains(page))
        cache.evict_oldest();
    cache.add(page);
}
```

## 8. Online Bipartite Matching (Karp-Vazirani-Vazirani)

Given offline set U and online set V (arrives one by one), match greedily. Competitive ratio:  $1 - \frac{1}{e}$

Used in ad allocation and resource assignment.

## **9. Approximation + Online Together**

Modern algorithms blend both:

- Streaming algorithms: One pass, small memory (Count-Min, reservoir sampling)- Online learning: Update models incrementally (SGD, perceptron)- Approximate dynamic programming: RL and heuristic search These are approximate online solvers , both quick and adaptive.

## **10. Why It Matters**

Approximation algorithms give us provable near-optimal answers. Online algorithms give us real-time adaptivity. Together, they model intelligence under limits , when time and information are scarce.

“Sometimes, good and on time beats perfect and late.”

### **Try It Yourself**

1. Implement 2-approx vertex cover on a small graph.
2. Simulate online paging with LRU vs Random.
3. Build a greedy set cover solver.
4. Measure competitive ratio on test sequences.
5. Combine ideas: streaming + approximation for big data filtering.

## **99. Fairness, Causal Inference, and Robust Optimization**

As algorithms increasingly shape decisions , from hiring to lending to healthcare , we must ensure they're fair, causally sound, and robust to uncertainty. This section blends ideas from ethics, statistics, and optimization to make algorithms not just efficient, but responsible and reliable.

### **1. Why Fairness Matters**

Machine learning systems often inherit biases from data. Without intervention, they can amplify inequality or discrimination.

Fairness-aware algorithms explicitly measure and correct these effects.

Common sources of bias:

- Historical bias (biased data)- Measurement bias (imprecise features)- Selection bias (skewed samples) The goal: equitable treatment across sensitive groups (gender, race, region, etc.)

## 2. Formal Fairness Criteria

Several fairness notions exist, often conflicting:

| Criterion          | Description                                   | Example                                          |
|--------------------|-----------------------------------------------|--------------------------------------------------|
| Demographic Parity | ( $P\hat{Y} = 1 A = a = P\hat{Y} = 1 A = b$ ) | Equal positive rate                              |
| Equal Opportunity  | Equal true positive rates                     | Same recall for all groups                       |
| Equalized Odds     | Equal TPR & FPR                               | Balanced errors                                  |
| Calibration        | Same predicted probability meaning            | If model says 70%, all groups should achieve 70% |

No single measure fits all , fairness depends on context and trade-offs.

## 3. Algorithmic Fairness Techniques

1. Pre-processing Rebalance or reweight data before training. Example: reweighing, sampling.
2. In-processing Add fairness constraints to loss function. Example: adversarial debiasing.
3. Post-processing Adjust predictions after training. Example: threshold shifting.

Tiny Code (Adversarial Debiasing Skeleton)

```
for x, a, y in data:
    y_pred = model(x)
    loss_main = loss_fn(y_pred, y)
    loss_adv = adv_fn(y_pred, a)
    loss_total = loss_main - * loss_adv
    update(loss_total)
```

Here, the adversary tries to predict sensitive attribute, encouraging invariance.

## 4. Causal Inference Basics

Correlation causation. To reason about fairness and robustness, we need causal understanding , what *would* happen if we changed something.

Causal inference models relationships via Directed Acyclic Graphs (DAGs):

- Nodes: variables- Edges: causal influence

## 5. Counterfactual Reasoning

A counterfactual asks:

“What would the outcome be if we intervened differently?”

Formally:

$$P(Y_{do(X=x)})$$

Used in:

- Fairness (counterfactual fairness)- Policy evaluation- Robust decision making

## 6. Counterfactual Fairness

An algorithm is counterfactually fair if prediction stays the same under hypothetical changes to sensitive attributes.

$$\hat{Y} * A \leftarrow a(U) = \hat{Y} * A \leftarrow a'(U)$$

This requires causal models , not just data.

## 7. Robust Optimization

In uncertain environments, we want solutions that hold up under worst-case conditions.

Formulation:

$$\min_x \max_{\xi \in \Xi} f(x, \xi)$$

where  $\Xi$  is the uncertainty set.

Example: Design a portfolio that performs well under varying market conditions.

Tiny Code

```

double robust_objective(double x[], Scenario Xi[], int N) {
    double worst = -INF;
    for (i=0; i<N; i++)
        worst = max(worst, f(x, Xi[i]));
    return worst;
}

```

This searches for a solution minimizing worst-case loss.

## 8. Distributional Robustness

Instead of worst-case instances, protect against worst-case distributions:

$$\min_{\theta} \sup_{Q \in \mathcal{B}(P)} \mathbb{E}_{x \sim Q}[L(\theta, x)]$$

Used in adversarial training and domain adaptation.

Example: Add noise or perturbations to improve resilience:

```
x_adv = x + * sign(grad(loss, x))
```

## 9. Balancing Fairness, Causality, and Robustness

| Goal       | Method                              | Challenge                  |
|------------|-------------------------------------|----------------------------|
| Fairness   | Parity, Adversarial, Counterfactual | Competing definitions      |
| Causality  | DAGs, do-calculus, SCMs             | Identifying true structure |
| Robustness | Min-max, DRO, Adversarial Training  | Trade-off with accuracy    |

Real-world design involves balancing trade-offs.

Sometimes improving fairness reduces accuracy, or robustness increases conservatism.

## 10. Why It Matters

Algorithms don't exist in isolation , they affect people. Embedding fairness, causality, and robustness ensures systems are trustworthy, interpretable, and just.

“The goal is not just intelligent algorithms , but responsible ones.”

## Try It Yourself

1. Train a simple classifier on biased data.
2. Apply reweighing or adversarial debiasing.
3. Draw a causal DAG of your data features.
4. Compute counterfactual fairness for a sample.
5. Implement a robust loss using adversarial perturbations.

## 100. AI Planning, Search, and Learning Systems

AI systems are not just pattern recognizers , they are decision makers. They plan, search, and learn in structured environments, choosing actions that lead to long-term goals. This section explores how modern AI combines planning, search, and learning to solve complex tasks.

### 1. What Is AI Planning?

AI planning is about finding a sequence of actions that transforms an initial state into a goal state.

Formally, a planning problem consists of:

- States ( S )- Actions ( A )- Transition function (  $T(s, a) \rightarrow s'$  )- Goal condition  $G \subseteq S$ - Cost function (  $c(a)$  ) The objective: Find a plan  $\pi = [a_1, a_2, \dots, a_n]$  minimizing total cost or maximizing reward.

### 2. Search-Based Planning

At the heart of planning lies search. Search explores possible action sequences, guided by heuristics.

| Algorithm | Type       | Description                    |
|-----------|------------|--------------------------------|
| DFS       | Uninformed | Deep exploration, no guarantee |
| BFS       | Uninformed | Finds shortest path            |
| Dijkstra  | Weighted   | Optimal if costs 0             |
| A*        | Heuristic  | Combines cost + heuristic      |

A\* Search Formula:

$$f(n) = g(n) + h(n)$$

where:

- ( $g(n)$ ): cost so far- ( $h(n)$ ): heuristic estimate to goal If ( $h$ ) is admissible, A\* is optimal.

Tiny Code (A\* Skeleton)

```

priority_queue<Node> open;
g[start] = 0;
open.push({start, h(start)});
```

```

while (!open.empty()) {
    n = open.pop_min();
    if (goal(n)) break;
    for (a in actions(n)) {
        s = step(n, a);
        cost = g[n] + c(n, a);
        if (cost < g[s]) {
            g[s] = cost;
            f[s] = g[s] + h(s);
            open.push({s, f[s]});
        }
    }
}
```

### 3. Heuristics and Admissibility

A heuristic ( $h(s)$ ) estimates distance to the goal.

- Admissible: never overestimates- Consistent: satisfies triangle inequality Examples:
- Manhattan distance (grids)- Euclidean distance (geometry)- Pattern databases (puzzles)  
Good heuristics = faster convergence.

### 4. Classical Planning (STRIPS)

In symbolic AI, states are represented by facts (predicates), and actions have preconditions and effects.

Example:

```

Action: Move(x, y)
Precondition: At(x), Clear(y)
Effect: ¬At(x), At(y)
```

Search happens in logical state space.

Planners:

- Forward search (progression)- Backward search (regression)- Heuristic planners (FF, HSP)

## 5. Hierarchical Planning

Break complex goals into subgoals.

- HTN (Hierarchical Task Networks): Define high-level tasks broken into subtasks.

Example: “Make dinner” → [Cook rice, Stir-fry vegetables, Set table]

Hierarchy makes planning modular and interpretable.

## 6. Probabilistic Planning

When actions are uncertain:

- MDPs: full observability, stochastic transitions- POMDPs: partial observability Use value iteration, policy iteration, or Monte Carlo planning.

## 7. Learning to Plan

Combine learning with search:

- Learned heuristics: neural networks approximate ( $h(s)$ )- AlphaZero-style planning: learn value + policy, guide tree search- Imitation learning: mimic expert demonstrations This bridges classical AI and modern ML.

Tiny Code (Learning-Guided A\*)

```
f = g + alpha * learned_heuristic(s)
```

Neural net learns ( $h_{\text{--}}(s)$ ) from solved examples.

## **8. Integrated Systems**

Modern AI stacks combine:

- Search (planning backbone)- Learning (policy, heuristic, model)- Simulation (data generation) Examples:
- AlphaZero: self-play + MCTS + neural nets- MuZero: learns model + value + policy jointly- Large Language Agents: use reasoning + memory + search

## **9. Real-World Applications**

- Robotics: motion planning, pathfinding- Games: Go, Chess, strategy games- Logistics: route optimization- Autonomy: drones, vehicles, AI assistants- Synthesis: program and query generation Each blends symbolic reasoning and statistical learning.

## **10. Why It Matters**

Planning, search, and learning form the triad of intelligence:

- Search explores possibilities- Planning sequences actions toward goals- Learning adapts heuristics from experience Together, they power systems that think, adapt, and act.

“Intelligence is not just knowing , it is choosing wisely under constraints.”

## **Try It Yourself**

1. Implement A\* search on a grid maze.
2. Add a Manhattan heuristic.
3. Extend to probabilistic transitions (simulate noise).
4. Build a simple planner with preconditions and effects.
5. Train a neural heuristic to guide search on puzzles.

# The Plan

## Chapter 1. Foundations of Algorithms

### 1. What Is an Algorithm?

| #  | Algorithm                  | Note                                               |
|----|----------------------------|----------------------------------------------------|
| 1  | Euclid's GCD               | Oldest known algorithm for greatest common divisor |
| 2  | Sieve of Eratosthenes      | Generate primes efficiently                        |
| 3  | Binary Search              | Divide and conquer search                          |
| 4  | Exponentiation by Squaring | Fast power computation                             |
| 5  | Long Division              | Classic step-by-step arithmetic                    |
| 6  | Modular Addition Algorithm | Wrap-around arithmetic                             |
| 7  | Base Conversion Algorithm  | Convert between number systems                     |
| 8  | Factorial Computation      | Recursive and iterative approaches                 |
| 9  | Fibonacci Sequence         | Recursive vs. dynamic computation                  |
| 10 | Tower of Hanoi             | Recursive problem-solving pattern                  |

### 2. Measuring Time and Space

| #  | Algorithm                  | Note                                |
|----|----------------------------|-------------------------------------|
| 11 | Counting Operations        | Manual step-counting for complexity |
| 12 | Loop Analysis              | Evaluate time cost of loops         |
| 13 | Recurrence Expansion       | Analyze recursive costs             |
| 14 | Amortized Analysis         | Average per-operation cost          |
| 15 | Space Counting             | Stack and heap tracking             |
| 16 | Memory Footprint Estimator | Track per-variable usage            |
| 17 | Time Complexity Table      | Map O(1)...O( $n^2$ )...O(2)        |
| 18 | Space-Time Tradeoff        | Cache vs. recomputation             |
| 19 | Profiling Algorithm        | Empirical time measurement          |
| 20 | Benchmarking Framework     | Compare algorithm performance       |

### 3. Big-O, Big-Theta, Big-Omega

| #  | Algorithm                      | Note                                  |
|----|--------------------------------|---------------------------------------|
| 21 | Growth Rate Comparator         | Compare asymptotic behaviors          |
| 22 | Dominant Term Extractor        | Simplify runtime expressions          |
| 23 | Limit-Based Complexity Test    | Using limits for asymptotics          |
| 24 | Summation Simplifier           | Sum of arithmetic/geometric sequences |
| 25 | Recurrence Tree Method         | Visualize recursive costs             |
| 26 | Master Theorem Evaluator       | Solve $T(n)$ recurrences              |
| 27 | Big-Theta Proof Builder        | Bounding upper and lower limits       |
| 28 | Big-Omega Case Finder          | Best-case scenario analysis           |
| 29 | Empirical Complexity Estimator | Measure via doubling experiments      |
| 30 | Complexity Class Identifier    | Match runtime to known class          |

### 4. Algorithmic Paradigms (Greedy, Divide and Conquer, DP)

| #  | Algorithm                   | Note                       |
|----|-----------------------------|----------------------------|
| 31 | Greedy Coin Change          | Local optimal step-by-step |
| 32 | Huffman Coding              | Greedy compression tree    |
| 33 | Merge Sort                  | Divide and conquer sort    |
| 34 | Binary Search               | Divide and conquer search  |
| 35 | Karatsuba Multiplication    | Recursive divide & conquer |
| 36 | Matrix Chain Multiplication | DP with substructure       |
| 37 | Longest Common Subsequence  | Classic DP problem         |
| 38 | Rod Cutting                 | DP optimization            |
| 39 | Activity Selection          | Greedy scheduling          |
| 40 | Optimal Merge Patterns      | Greedy file merging        |

### 5. Recurrence Relations

| #  | Algorithm                | Note                               |
|----|--------------------------|------------------------------------|
| 41 | Linear Recurrence Solver | Closed-form for linear recurrences |
| 42 | Master Theorem           | Divide-and-conquer complexity      |
| 43 | Substitution Method      | Inductive proof approach           |
| 44 | Iteration Method         | Expand recurrence step-by-step     |
| 45 | Generating Functions     | Transform recurrences              |
| 46 | Matrix Exponentiation    | Solve linear recurrences fast      |

| #  | Algorithm                 | Note                              |
|----|---------------------------|-----------------------------------|
| 47 | Recurrence to DP Table    | Tabulation approach               |
| 48 | Divide & Combine Template | Convert recurrence into algorithm |
| 49 | Memoized Recursive Solver | Store overlapping results         |
| 50 | Characteristic Polynomial | Solve homogeneous recurrence      |

## 6. Searching Basics

| #  | Algorithm               | Note                           |
|----|-------------------------|--------------------------------|
| 51 | Linear Search           | Sequential element scan        |
| 52 | Binary Search           | Midpoint halving               |
| 53 | Jump Search             | Block skip linear              |
| 54 | Exponential Search      | Doubling step size             |
| 55 | Interpolation Search    | Estimate position by value     |
| 56 | Ternary Search          | Divide into thirds             |
| 57 | Fibonacci Search        | Golden ratio search            |
| 58 | Sentinel Search         | Early termination optimization |
| 59 | Bidirectional Search    | Meet-in-the-middle             |
| 60 | Search in Rotated Array | Adapted binary search          |

## 7. Sorting Basics

| #  | Algorithm      | Note                     |
|----|----------------|--------------------------|
| 61 | Bubble Sort    | Adjacent swap sort       |
| 62 | Selection Sort | Find minimum each pass   |
| 63 | Insertion Sort | Incremental build sort   |
| 64 | Shell Sort     | Gap-based insertion      |
| 65 | Merge Sort     | Divide-and-conquer       |
| 66 | Quick Sort     | Partition-based          |
| 67 | Heap Sort      | Binary heap order        |
| 68 | Counting Sort  | Integer key distribution |
| 69 | Radix Sort     | Digit-by-digit           |
| 70 | Bucket Sort    | Group into ranges        |

## 8. Data Structures Overview

| #  | Algorithm                  | Note                    |
|----|----------------------------|-------------------------|
| 71 | Stack Push/Pop             | LIFO operations         |
| 72 | Queue Enqueue/Dequeue      | FIFO operations         |
| 73 | Singly Linked List         | Linear node chain       |
| 74 | Doubly Linked List         | Bidirectional traversal |
| 75 | Hash Table Insertion       | Key-value indexing      |
| 76 | Binary Search Tree Insert  | Ordered node placement  |
| 77 | Heapify                    | Build heap in-place     |
| 78 | Union-Find Operations      | Disjoint-set management |
| 79 | Graph Adjacency List Build | Sparse representation   |
| 80 | Trie Insertion/Search      | Prefix tree for strings |

## 9. Graphs and Trees Overview

| #  | Algorithm                    | Note                          |
|----|------------------------------|-------------------------------|
| 81 | DFS Traversal                | Depth-first exploration       |
| 82 | BFS Traversal                | Level-order exploration       |
| 83 | Topological Sort             | DAG ordering                  |
| 84 | Minimum Spanning Tree        | Kruskal/Prim overview         |
| 85 | Dijkstra's Shortest Path     | Weighted graph shortest route |
| 86 | Bellman-Ford                 | Handle negative edges         |
| 87 | Floyd-Warshall               | All-pairs shortest path       |
| 88 | Union-Find for MST           | Edge grouping                 |
| 89 | Tree Traversals              | Inorder, Preorder, Postorder  |
| 90 | LCA (Lowest Common Ancestor) | Common node in tree           |

## 10. Algorithm Design Patterns

| #  | Algorithm               | Note                   |
|----|-------------------------|------------------------|
| 91 | Brute Force             | Try all possibilities  |
| 92 | Greedy Choice           | Local optimum per step |
| 93 | Divide and Conquer      | Break and merge        |
| 94 | Dynamic Programming     | Reuse subproblems      |
| 95 | Backtracking            | Explore with undo      |
| 96 | Branch and Bound        | Prune search space     |
| 97 | Randomized Algorithm    | Inject randomness      |
| 98 | Approximation Algorithm | Near-optimal solution  |
| 99 | Online Algorithm        | Step-by-step decision  |

| #   | Algorithm       | Note              |
|-----|-----------------|-------------------|
| 100 | Hybrid Strategy | Combine paradigms |

## Chapter 2. Sorting and Searching

### 11. Elementary Sorting (Bubble, Insertion, Selection)

| #   | Algorithm             | Note                                  |
|-----|-----------------------|---------------------------------------|
| 101 | Bubble Sort           | Swap adjacent out-of-order elements   |
| 102 | Improved Bubble Sort  | Stop early if already sorted          |
| 103 | Cocktail Shaker Sort  | Bidirectional bubble pass             |
| 104 | Selection Sort        | Select smallest element each pass     |
| 105 | Double Selection Sort | Find both min and max each pass       |
| 106 | Insertion Sort        | Insert each element into correct spot |
| 107 | Binary Insertion Sort | Use binary search for position        |
| 108 | Gnome Sort            | Simple insertion-like with swaps      |
| 109 | Odd-Even Sort         | Parallel-friendly comparison sort     |
| 110 | Stooge Sort           | Recursive quirky educational sort     |

### 12. Divide-and-Conquer Sorting (Merge, Quick, Heap)

| #   | Algorithm               | Note                             |
|-----|-------------------------|----------------------------------|
| 111 | Merge Sort              | Recursive divide and merge       |
| 112 | Iterative Merge Sort    | Bottom-up non-recursive version  |
| 113 | Quick Sort              | Partition-based recursive sort   |
| 114 | Hoare Partition Scheme  | Classic quicksort partition      |
| 115 | Lomuto Partition Scheme | Simpler but less efficient       |
| 116 | Randomized Quick Sort   | Avoid worst-case pivot           |
| 117 | Heap Sort               | Heapify + extract max repeatedly |
| 118 | 3-Way Quick Sort        | Handle duplicates efficiently    |
| 119 | External Merge Sort     | Disk-based merge for large data  |
| 120 | Parallel Merge Sort     | Divide work among threads        |

### 13. Counting and Distribution Sorts (Counting, Radix, Bucket)

| #   | Algorithm                | Note                                  |
|-----|--------------------------|---------------------------------------|
| 121 | Counting Sort            | Count key occurrences                 |
| 122 | Stable Counting Sort     | Preserve order of equals              |
| 123 | Radix Sort (LSD)         | Least significant digit first         |
| 124 | Radix Sort (MSD)         | Most significant digit first          |
| 125 | Bucket Sort              | Distribute into buckets               |
| 126 | Pigeonhole Sort          | Simple bucket variant                 |
| 127 | Flash Sort               | Distribution with in-place correction |
| 128 | Postman Sort             | Stable multi-key sort                 |
| 129 | Address Calculation Sort | Hash-like distribution                |
| 130 | Spread Sort              | Hybrid radix/quick strategy           |

## 14. Hybrid Sorts (IntroSort, Timsort)

| #   | Algorithm            | Note                             |
|-----|----------------------|----------------------------------|
| 131 | IntroSort            | Quick + Heap fallback            |
| 132 | TimSort              | Merge + Insertion + Runs         |
| 133 | Dual-Pivot QuickSort | Modern quicksort optimization    |
| 134 | SmoothSort           | Heap-like adaptive sort          |
| 135 | Block Merge Sort     | Cache-efficient merge variant    |
| 136 | Adaptive Merge Sort  | Adjusts to partially sorted data |
| 137 | PDQSort              | Pattern-defeating quicksort      |
| 138 | WikiSort             | Stable in-place merge            |
| 139 | GrailSort            | In-place stable mergesort        |
| 140 | Adaptive Hybrid Sort | Dynamically selects strategy     |

## 15. Special Sorts (Cycle, Gnome, Comb, Pancake)

| #   | Algorithm           | Note                      |
|-----|---------------------|---------------------------|
| 141 | Cycle Sort          | Minimal writes            |
| 142 | Comb Sort           | Shrinking gap bubble      |
| 143 | Gnome Sort          | Insertion-like with swaps |
| 144 | Cocktail Sort       | Two-way bubble            |
| 145 | Pancake Sort        | Flip-based sorting        |
| 146 | Bitonic Sort        | Parallel network sorting  |
| 147 | Odd-Even Merge Sort | Sorting network design    |
| 148 | Sleep Sort          | Uses timing as order key  |
| 149 | Bead Sort           | Simulates gravity         |

| #   | Algorithm | Note                          |
|-----|-----------|-------------------------------|
| 150 | Bogo Sort | Randomly permute until sorted |

## 16. Linear and Binary Search

| #   | Algorithm                   | Note                         |
|-----|-----------------------------|------------------------------|
| 151 | Linear Search               | Scan sequentially            |
| 152 | Linear Search (Sentinel)    | Guard element at end         |
| 153 | Binary Search (Iterative)   | Halve interval each loop     |
| 154 | Binary Search (Recursive)   | Halve interval via recursion |
| 155 | Binary Search (Lower Bound) | First $\geq$ target          |
| 156 | Binary Search (Upper Bound) | First $>$ target             |
| 157 | Exponential Search          | Double step size             |
| 158 | Jump Search                 | Jump fixed steps then linear |
| 159 | Fibonacci Search            | Golden-ratio style jumps     |
| 160 | Uniform Binary Search       | Avoid recomputing midpoints  |

## 17. Interpolation and Exponential Search

| #   | Algorithm                      | Note                         |
|-----|--------------------------------|------------------------------|
| 161 | Interpolation Search           | Estimate index by value      |
| 162 | Recursive Interpolation Search | Divide by estimated midpoint |
| 163 | Exponential Search             | Double and binary refine     |
| 164 | Doubling Search                | Generic exponential pattern  |
| 165 | Galloping Search               | Used in TimSort merges       |
| 166 | Unbounded Binary Search        | Find bounds dynamically      |
| 167 | Root-Finding Bisection         | Search zero-crossing         |
| 168 | Golden Section Search          | Optimize unimodal function   |
| 169 | Fibonacci Search (Optimum)     | Similar to golden search     |
| 170 | Jump + Binary Hybrid           | Combined probing strategy    |

## 18. Selection Algorithms (Quickselect, Median of Medians)

| #   | Algorithm         | Note                      |
|-----|-------------------|---------------------------|
| 171 | Quickselect       | Partition-based selection |
| 172 | Median of Medians | Deterministic pivot       |

| #   | Algorithm                 | Note                    |
|-----|---------------------------|-------------------------|
| 173 | Randomized Select         | Random pivot version    |
| 174 | Binary Search on Answer   | Range-based selection   |
| 175 | Order Statistics Tree     | BST with rank queries   |
| 176 | Tournament Tree Selection | Hierarchical comparison |
| 177 | Heap Select (Min-Heap)    | Maintain top-k elements |
| 178 | Partial QuickSort         | Sort partial prefix     |
| 179 | BFPRT Algorithm           | Linear-time selection   |
| 180 | Kth Largest Stream        | Streaming selection     |

## 19. Range Searching and Nearest Neighbor

| #   | Algorithm                  | Note                        |
|-----|----------------------------|-----------------------------|
| 181 | Binary Search Range        | Find lower and upper bounds |
| 182 | Segment Tree Query         | Sum/min/max over interval   |
| 183 | Fenwick Tree Query         | Efficient prefix sums       |
| 184 | Interval Tree Search       | Overlap queries             |
| 185 | KD-Tree Search             | Spatial nearest neighbor    |
| 186 | R-Tree Query               | Range search in geometry    |
| 187 | Range Minimum Query (RMQ)  | Sparse table approach       |
| 188 | Mo's Algorithm             | Offline query reordering    |
| 189 | Sweep Line Range Search    | Sort + scan technique       |
| 190 | Ball Tree Nearest Neighbor | Metric-space search         |

## 20. Search Optimizations and Variants

| #   | Algorithm                      | Note                     |
|-----|--------------------------------|--------------------------|
| 191 | Binary Search with Tolerance   | For floating values      |
| 192 | Ternary Search                 | Unimodal optimization    |
| 193 | Hash-Based Search              | O(1) expected lookup     |
| 194 | Bloom Filter Lookup            | Probabilistic membership |
| 195 | Cuckoo Hash Search             | Dual-hash relocation     |
| 196 | Robin Hood Hashing             | Equalize probe lengths   |
| 197 | Jump Consistent Hashing        | Stable hash assignment   |
| 198 | Prefix Search in Trie          | Auto-completion lookup   |
| 199 | Pattern Search in Suffix Array | Fast substring lookup    |
| 200 | Search in Infinite Array       | Dynamic bound finding    |

## Chapter 3. Data Structures in Action

### 21. Arrays, Linked Lists, Stacks, Queues

| #   | Algorithm                        | Note                              |
|-----|----------------------------------|-----------------------------------|
| 201 | Dynamic Array Resize             | Doubling strategy for capacity    |
| 202 | Circular Array Implementation    | Wrap-around indexing              |
| 203 | Singly Linked List Insert/Delete | Basic node manipulation           |
| 204 | Doubly Linked List Insert/Delete | Two-way linkage                   |
| 205 | Stack Push/Pop                   | LIFO structure                    |
| 206 | Queue Enqueue/Dequeue            | FIFO structure                    |
| 207 | Deque Implementation             | Double-ended queue                |
| 208 | Circular Queue                   | Fixed-size queue with wrap-around |
| 209 | Stack via Queue                  | Implement stack using two queues  |
| 210 | Queue via Stack                  | Implement queue using two stacks  |

### 22. Hash Tables and Variants (Cuckoo, Robin Hood, Consistent)

| #   | Algorithm            | Note                             |
|-----|----------------------|----------------------------------|
| 211 | Hash Table Insertion | Key-value pair with modulo       |
| 212 | Linear Probing       | Resolve collisions sequentially  |
| 213 | Quadratic Probing    | Nonlinear probing sequence       |
| 214 | Double Hashing       | Alternate hash on collision      |
| 215 | Cuckoo Hashing       | Two-table relocation strategy    |
| 216 | Robin Hood Hashing   | Equalize probe length fairness   |
| 217 | Chained Hash Table   | Linked list buckets              |
| 218 | Perfect Hashing      | No-collision mapping             |
| 219 | Consistent Hashing   | Stable distribution across nodes |
| 220 | Dynamic Rehashing    | Resize on load factor threshold  |

### 23. Heaps (Binary, Fibonacci, Pairing)

| #   | Algorithm            | Note                     |
|-----|----------------------|--------------------------|
| 221 | Binary Heap Insert   | Bubble-up maintenance    |
| 222 | Binary Heap Delete   | Heapify-down maintenance |
| 223 | Build Heap (Heapify) | Bottom-up O(n) build     |
| 224 | Heap Sort            | Extract max repeatedly   |

| #   | Algorithm                    | Note                           |
|-----|------------------------------|--------------------------------|
| 225 | Min Heap Implementation      | For smallest element access    |
| 226 | Max Heap Implementation      | For largest element access     |
| 227 | Fibonacci Heap Insert/Delete | Amortized efficient operations |
| 228 | Pairing Heap Merge           | Lightweight mergeable heap     |
| 229 | Binomial Heap Merge          | Merge trees of equal order     |
| 230 | Leftist Heap Merge           | Maintain rank-skewed heap      |

## 24. Balanced Trees (AVL, Red-Black, Splay, Treap)

| #   | Algorithm              | Note                         |
|-----|------------------------|------------------------------|
| 231 | AVL Tree Insert        | Rotate to maintain balance   |
| 232 | AVL Tree Delete        | Balance after deletion       |
| 233 | Red-Black Tree Insert  | Color fix and rotations      |
| 234 | Red-Black Tree Delete  | Maintain invariants          |
| 235 | Splay Tree Access      | Move accessed node to root   |
| 236 | Treap Insert           | Priority-based rotation      |
| 237 | Treap Delete           | Randomized balance           |
| 238 | Weight Balanced Tree   | Maintain subtree weights     |
| 239 | Scapegoat Tree Rebuild | Rebalance on size threshold  |
| 240 | AA Tree                | Simplified red-black variant |

## 25. Segment Trees and Fenwick Trees

| #   | Algorithm               | Note                         |
|-----|-------------------------|------------------------------|
| 241 | Build Segment Tree      | Recursive construction       |
| 242 | Range Sum Query         | Recursive or iterative query |
| 243 | Range Update            | Lazy propagation technique   |
| 244 | Point Update            | Modify single element        |
| 245 | Fenwick Tree Build      | Incremental binary index     |
| 246 | Fenwick Update          | Update cumulative sums       |
| 247 | Fenwick Query           | Prefix sum retrieval         |
| 248 | Segment Tree Merge      | Combine child results        |
| 249 | Persistent Segment Tree | Maintain history of versions |
| 250 | 2D Segment Tree         | For matrix range queries     |

## 26. Disjoint Set Union (Union-Find)

| #   | Algorithm            | Note                          |
|-----|----------------------|-------------------------------|
| 251 | Make-Set             | Initialize each element       |
| 252 | Find                 | Locate representative         |
| 253 | Union                | Merge two sets                |
| 254 | Union by Rank        | Attach smaller tree to larger |
| 255 | Path Compression     | Flatten tree structure        |
| 256 | DSU with Rollback    | Support undo operations       |
| 257 | DSU on Tree          | Track subtree connectivity    |
| 258 | Kruskal's MST        | Edge selection with DSU       |
| 259 | Connected Components | Group graph nodes             |
| 260 | Offline Query DSU    | Handle dynamic unions         |

## 27. Probabilistic Data Structures (Bloom, Count-Min, HyperLogLog)

| #   | Algorithm             | Note                           |
|-----|-----------------------|--------------------------------|
| 261 | Bloom Filter Insert   | Hash to bit array              |
| 262 | Bloom Filter Query    | Probabilistic membership check |
| 263 | Counting Bloom Filter | Support deletions via counters |
| 264 | Cuckoo Filter         | Space-efficient alternative    |
| 265 | Count-Min Sketch      | Approximate frequency table    |
| 266 | HyperLogLog           | Cardinality estimation         |
| 267 | Flajolet-Martin       | Early probabilistic counting   |
| 268 | MinHash               | Estimate Jaccard similarity    |
| 269 | Reservoir Sampling    | Random k-sample stream         |
| 270 | Skip Bloom Filter     | Range queries on Bloom         |

## 28. Skip Lists and B-Trees

| #   | Algorithm        | Note                       |
|-----|------------------|----------------------------|
| 271 | Skip List Insert | Probabilistic layered list |
| 272 | Skip List Delete | Adjust pointers            |
| 273 | Skip List Search | Jump via tower levels      |
| 274 | B-Tree Insert    | Split on overflow          |
| 275 | B-Tree Delete    | Merge on underflow         |
| 276 | B+ Tree Search   | Leaf-based sequential scan |

| #   | Algorithm           | Note                          |
|-----|---------------------|-------------------------------|
| 277 | B+ Tree Range Query | Efficient ordered access      |
| 278 | B* Tree             | More space-efficient variant  |
| 279 | Adaptive Radix Tree | Byte-wise branching           |
| 280 | Trie Compression    | Path compression optimization |

## 29. Persistent and Functional Data Structures

| #   | Algorithm                 | Note                     |
|-----|---------------------------|--------------------------|
| 281 | Persistent Stack          | Keep all versions        |
| 282 | Persistent Array          | Copy-on-write segments   |
| 283 | Persistent Segment Tree   | Versioned updates        |
| 284 | Persistent Linked List    | Immutable nodes          |
| 285 | Functional Queue          | Amortized reverse lists  |
| 286 | Finger Tree               | Fast concat and split    |
| 287 | Zipper Structure          | Localized mutation       |
| 288 | Red-Black Persistent Tree | Immutable balanced tree  |
| 289 | Trie with Versioning      | Historical string lookup |
| 290 | Persistent Union-Find     | Time-travel connectivity |

## 30. Advanced Trees and Range Queries

| #   | Algorithm              | Note                      |
|-----|------------------------|---------------------------|
| 291 | Sparse Table Build     | Static range min/max      |
| 292 | Cartesian Tree         | RMQ to LCA transformation |
| 293 | Segment Tree Beats     | Handle complex queries    |
| 294 | Merge Sort Tree        | Range count queries       |
| 295 | Wavelet Tree           | Rank/select by value      |
| 296 | KD-Tree                | Multidimensional queries  |
| 297 | Range Tree             | Orthogonal range queries  |
| 298 | Fenwick 2D Tree        | Matrix prefix sums        |
| 299 | Treap Split/Merge      | Range-based treap ops     |
| 300 | Mo's Algorithm on Tree | Offline subtree queries   |

## Chapter 4. Graph Algorithms

### 31. Traversals (DFS, BFS, Iterative Deepening)

| #   | Algorithm                           | Note                                |
|-----|-------------------------------------|-------------------------------------|
| 301 | Depth-First Search (Recursive)      | Explore deeply before backtracking  |
| 302 | Depth-First Search (Iterative)      | Stack-based exploration             |
| 303 | Breadth-First Search (Queue)        | Level-order traversal               |
| 304 | Iterative Deepening DFS             | Combine depth-limit + completeness  |
| 305 | Bidirectional BFS                   | Search from both ends               |
| 306 | DFS on Grid                         | Maze solving / connected components |
| 307 | BFS on Grid                         | Shortest path in unweighted graph   |
| 308 | Multi-Source BFS                    | Parallel layer expansion            |
| 309 | Topological Sort (DFS-based)        | DAG ordering                        |
| 310 | Topological Sort (Kahn's Algorithm) | In-degree tracking                  |

### 32. Strongly Connected Components (Tarjan, Kosaraju)

| #   | Algorithm                   | Note                      |
|-----|-----------------------------|---------------------------|
| 311 | Kosaraju's Algorithm        | Two-pass DFS              |
| 312 | Tarjan's Algorithm          | Low-link discovery        |
| 313 | Gabow's Algorithm           | Stack pair tracking       |
| 314 | SCC DAG Construction        | Condensed component graph |
| 315 | SCC Online Merge            | Incremental condensation  |
| 316 | Component Label Propagation | Iterative labeling        |
| 317 | Path-Based SCC              | DFS with path stack       |
| 318 | Kosaraju Parallel Version   | SCC via parallel DFS      |
| 319 | Dynamic SCC Maintenance     | Add/remove edges          |
| 320 | SCC for Weighted Graph      | Combine with edge weights |

### 33. Shortest Paths (Dijkstra, Bellman-Ford, A\*, Johnson)

| #   | Algorithm                 | Note                        |
|-----|---------------------------|-----------------------------|
| 321 | Dijkstra (Binary Heap)    | Greedy edge relaxation      |
| 322 | Dijkstra (Fibonacci Heap) | Improved priority queue     |
| 323 | Bellman-Ford              | Negative weights support    |
| 324 | SPFA (Queue Optimization) | Faster average Bellman-Ford |
| 325 | A* Search                 | Heuristic-guided path       |
| 326 | Floyd-Warshall            | All-pairs shortest path     |
| 327 | Johnson's Algorithm       | All-pairs using reweighting |
| 328 | 0-1 BFS                   | Deque-based shortest path   |
| 329 | Dial's Algorithm          | Integer weight buckets      |

| #   | Algorithm             | Note                     |
|-----|-----------------------|--------------------------|
| 330 | Multi-Source Dijkstra | Multiple starting points |

### 34. Shortest Path Variants (0–1 BFS, Bidirectional, Heuristic A\*)

| #   | Algorithm                   | Note                               |
|-----|-----------------------------|------------------------------------|
| 331 | 0–1 BFS                     | For edges with weight 0 or 1       |
| 332 | Bidirectional Dijkstra      | Meet in the middle                 |
| 333 | A* with Euclidean Heuristic | Spatial shortest path              |
| 334 | ALT Algorithm               | A* landmarks + triangle inequality |
| 335 | Contraction Hierarchies     | Preprocessing for road networks    |
| 336 | CH Query Algorithm          | Shortcut-based routing             |
| 337 | Bellman-Ford Queue Variant  | Early termination                  |
| 338 | Dijkstra with Early Stop    | Halt on target                     |
| 339 | Goal-Directed Search        | Restrict expansion direction       |
| 340 | Yen's K Shortest Paths      | Enumerate multiple best paths      |

### 35. Minimum Spanning Trees (Kruskal, Prim, Borůvka)

| #   | Algorithm                          | Note                    |
|-----|------------------------------------|-------------------------|
| 341 | Kruskal's Algorithm                | Sort edges + union-find |
| 342 | Prim's Algorithm (Heap)            | Grow MST from seed      |
| 343 | Prim's Algorithm (Adj Matrix)      | Dense graph variant     |
| 344 | Borůvka's Algorithm                | Component merging       |
| 345 | Reverse-Delete MST                 | Remove heavy edges      |
| 346 | MST via Dijkstra Trick             | For positive weights    |
| 347 | Dynamic MST Maintenance            | Handle edge updates     |
| 348 | Minimum Bottleneck Spanning Tree   | Max edge minimization   |
| 349 | Manhattan MST                      | Grid graph optimization |
| 350 | Euclidean MST (Kruskal + Geometry) | Use Delaunay graph      |

### 36. Flows (Ford–Fulkerson, Edmonds–Karp, Dinic)

| #   | Algorithm      | Note                     |
|-----|----------------|--------------------------|
| 351 | Ford–Fulkerson | Augmenting path method   |
| 352 | Edmonds–Karp   | BFS-based Ford–Fulkerson |

| #   | Algorithm                        | Note                         |
|-----|----------------------------------|------------------------------|
| 353 | Dinic's Algorithm                | Level graph + blocking flow  |
| 354 | Push–Relabel                     | Local preflow push           |
| 355 | Capacity Scaling                 | Speed-up with capacity tiers |
| 356 | Cost Scaling                     | Min-cost optimization        |
| 357 | Min-Cost Max-Flow (Bellman-Ford) | Costed augmenting paths      |
| 358 | Min-Cost Max-Flow (SPFA)         | Faster average               |
| 359 | Circulation with Demands         | Generalized flow formulation |
| 360 | Successive Shortest Path         | Incremental min-cost updates |

### 37. Cuts (Stoer–Wagner, Karger, Gomory–Hu)

| #   | Algorithm                      | Note                         |
|-----|--------------------------------|------------------------------|
| 361 | Stoer–Wagner Minimum Cut       | Global min cut               |
| 362 | Karger’s Randomized Cut        | Contract edges randomly      |
| 363 | Karger–Stein                   | Recursive randomized cut     |
| 364 | Gomory–Hu Tree                 | All-pairs min-cut            |
| 365 | Max-Flow Min-Cut               | Duality theorem application  |
| 366 | Stoer–Wagner Repeated Phase    | Multiple passes              |
| 367 | Dynamic Min Cut                | Maintain on edge update      |
| 368 | Minimum s–t Cut (Edmonds–Karp) | Based on flow                |
| 369 | Approximate Min Cut            | Random sampling              |
| 370 | Min k-Cut                      | Partition graph into k parts |

### 38. Matchings (Hopcroft–Karp, Hungarian, Blossom)

| #   | Algorithm                      | Note                              |
|-----|--------------------------------|-----------------------------------|
| 371 | Bipartite Matching (DFS)       | Simple augmenting path            |
| 372 | Hopcroft–Karp                  | $O(E\sqrt{V})$ bipartite matching |
| 373 | Hungarian Algorithm            | Weighted assignment               |
| 374 | Kuhn–Munkres                   | Max-weight matching               |
| 375 | Blossom Algorithm              | General graph matching            |
| 376 | Edmonds’ Blossom Shrinking     | Odd cycle contraction             |
| 377 | Greedy Matching                | Fast approximate                  |
| 378 | Stable Marriage (Gale–Shapley) | Stable pairing                    |
| 379 | Weighted b-Matching            | Capacity-constrained              |
| 380 | Maximal Matching               | Local greedy maximal set          |

## 39. Tree Algorithms (LCA, HLD, Centroid Decomposition)

| #   | Algorithm                  | Note                        |
|-----|----------------------------|-----------------------------|
| 381 | Euler Tour LCA             | Flatten tree to array       |
| 382 | Binary Lifting LCA         | Jump powers of two          |
| 383 | Tarjan's LCA (Offline DSU) | Query via union-find        |
| 384 | Heavy-Light Decomposition  | Decompose paths             |
| 385 | Centroid Decomposition     | Recursive split on centroid |
| 386 | Tree Diameter (DFS Twice)  | Farthest pair               |
| 387 | Tree DP                    | Subtree-based optimization  |
| 388 | Rerooting DP               | Compute all roots' answers  |
| 389 | Binary Search on Tree      | Edge weight constraints     |
| 390 | Virtual Tree               | Build on query subset       |

## 40. Advanced Graph Algorithms and Tricks

| #   | Algorithm                           | Note                          |
|-----|-------------------------------------|-------------------------------|
| 391 | Topological DP                      | DP on DAG order               |
| 392 | SCC Condensed Graph DP              | Meta-graph processing         |
| 393 | Eulerian Path                       | Trail covering all edges      |
| 394 | Hamiltonian Path                    | NP-complete exploration       |
| 395 | Chinese Postman                     | Eulerian circuit with repeats |
| 396 | Hierholzer's Algorithm              | Construct Eulerian circuit    |
| 397 | Johnson's Cycle Finding             | Enumerate all cycles          |
| 398 | Transitive Closure (Floyd–Warshall) | Reachability matrix           |
| 399 | Graph Coloring (Backtracking)       | Constraint satisfaction       |
| 400 | Articulation Points & Bridges       | Critical structure detection  |

## Chapter 5. Dynamic Programming

### 41. DP Basics and State Transitions

| #   | Algorithm       | Note                          |
|-----|-----------------|-------------------------------|
| 401 | Fibonacci DP    | Classic top-down vs bottom-up |
| 402 | Climbing Stairs | Count paths with small steps  |
| 403 | Grid Paths      | DP over 2D lattice            |
| 404 | Min Cost Path   | Accumulate minimal sums       |

| #   | Algorithm                           | Note                            |
|-----|-------------------------------------|---------------------------------|
| 405 | Coin Change (Count Ways)            | Combinatorial sums              |
| 406 | Coin Change (Min Coins)             | Minimize step count             |
| 407 | Knapsack 0/1                        | Select items under weight limit |
| 408 | Knapsack Unbounded                  | Repeatable items                |
| 409 | Longest Increasing Subsequence (DP) | Subsequence optimization        |
| 410 | Edit Distance (Levenshtein)         | Measure similarity step-by-step |

## 42. Classic Problems (Knapsack, Subset Sum, Coin Change)

| #   | Algorithm                  | Note                              |
|-----|----------------------------|-----------------------------------|
| 411 | 0/1 Knapsack               | Value maximization under capacity |
| 412 | Subset Sum                 | Boolean feasibility DP            |
| 413 | Equal Partition            | Divide set into equal halves      |
| 414 | Count of Subsets with Sum  | Counting variant                  |
| 415 | Target Sum                 | DP with +/- transitions           |
| 416 | Unbounded Knapsack         | Reuse items                       |
| 417 | Fractional Knapsack        | Greedy + DP comparison            |
| 418 | Coin Change (Min Coins)    | DP shortest path                  |
| 419 | Coin Change (Count Ways)   | Combinatorial counting            |
| 420 | Multi-Dimensional Knapsack | Capacity in multiple dimensions   |

## 43. Sequence Problems (LIS, LCS, Edit Distance)

| #   | Algorithm                           | Note                          |
|-----|-------------------------------------|-------------------------------|
| 421 | Longest Increasing Subsequence      | $O(n^2)$ DP                   |
| 422 | LIS (Patience Sorting)              | $O(n \log n)$ optimized       |
| 423 | Longest Common Subsequence          | Two-sequence DP               |
| 424 | Edit Distance (Levenshtein)         | Transform operations          |
| 425 | Longest Palindromic Subsequence     | Symmetric DP                  |
| 426 | Shortest Common Supersequence       | Merge sequences               |
| 427 | Longest Repeated Subsequence        | DP with overlap               |
| 428 | String Interleaving                 | Merge with order preservation |
| 429 | Sequence Alignment (Bioinformatics) | Gap penalties                 |
| 430 | Diff Algorithm (Myers/DP)           | Minimal edit path             |

## 44. Matrix and Chain Problems

| #   | Algorithm                      | Note                  |
|-----|--------------------------------|-----------------------|
| 431 | Matrix Chain Multiplication    | Parenthesization cost |
| 432 | Boolean Parenthesization       | Count true outcomes   |
| 433 | Burst Balloons                 | Interval DP           |
| 434 | Optimal BST                    | Weighted search cost  |
| 435 | Polygon Triangulation          | DP over partitions    |
| 436 | Matrix Path Sum                | DP on 2D grid         |
| 437 | Largest Square Submatrix       | Dynamic growth check  |
| 438 | Max Rectangle in Binary Matrix | Histogram + DP        |
| 439 | Submatrix Sum Queries          | Prefix sum DP         |
| 440 | Palindrome Partitioning        | DP with cuts          |

## 45. Bitmask DP and Traveling Salesman

| #   | Algorithm                 | Note                            |
|-----|---------------------------|---------------------------------|
| 441 | Traveling Salesman (TSP)  | Visit all cities                |
| 442 | Subset DP                 | Over subsets of states          |
| 443 | Hamiltonian Path DP       | State compression               |
| 444 | Assignment Problem DP     | Mask over tasks                 |
| 445 | Partition into Two Sets   | Balanced load                   |
| 446 | Count Hamiltonian Cycles  | Bitmask enumeration             |
| 447 | Steiner Tree DP           | Minimal connection of terminals |
| 448 | SOS DP (Sum Over Subsets) | Precompute sums                 |
| 449 | Bitmask Knapsack          | State compression               |
| 450 | Bitmask Independent Set   | Graph subset optimization       |

## 46. Digit DP and SOS DP

| #   | Algorithm                         | Note                     |
|-----|-----------------------------------|--------------------------|
| 451 | Count Numbers with Property       | Digit-state transitions  |
| 452 | Count Without Adjacent Duplicates | Adjacent constraints     |
| 453 | Sum of Digits in Range            | Carry-dependent states   |
| 454 | Count with Mod Condition          | DP over digit sum mod M  |
| 455 | Count of Increasing Digits        | Ordered constraint       |
| 456 | Count with Forbidden Digits       | Exclusion transitions    |
| 457 | SOS DP Subset Sum                 | Sum over bitmask subsets |

| #   | Algorithm                | Note                     |
|-----|--------------------------|--------------------------|
| 458 | SOS DP Superset Sum      | Sum over supersets       |
| 459 | XOR Basis DP             | Combine digit and bit DP |
| 460 | Digit DP for Palindromes | Symmetric digit state    |

## 47. DP Optimizations (Divide & Conquer, Convex Hull Trick, Knuth)

| #   | Algorithm                    | Note                           |
|-----|------------------------------|--------------------------------|
| 461 | Divide & Conquer DP          | Monotone decision property     |
| 462 | Knuth Optimization           | DP with quadrangle inequality  |
| 463 | Convex Hull Trick            | Linear recurrence min queries  |
| 464 | Li Chao Tree                 | Segment-based hull maintenance |
| 465 | Slope Trick                  | Piecewise-linear optimization  |
| 466 | Monotonic Queue Optimization | Sliding DP state               |
| 467 | Bitset DP                    | Speed via bit-parallel         |
| 468 | Offline DP Queries           | Preprocessing state            |
| 469 | DP + Segment Tree            | Range-based optimization       |
| 470 | Divide & Conquer Knapsack    | Split-space DP                 |

## 48. Tree DP and Rerooting

| #   | Algorithm                | Note                      |
|-----|--------------------------|---------------------------|
| 471 | Subtree Sum DP           | Aggregate values          |
| 472 | Diameter DP              | Max path via child states |
| 473 | Independent Set DP       | Choose or skip nodes      |
| 474 | Vertex Cover DP          | Tree constraint problem   |
| 475 | Path Counting DP         | Count root-leaf paths     |
| 476 | DP on Rooted Tree        | Bottom-up aggregation     |
| 477 | Rerooting Technique      | Compute for all roots     |
| 478 | Distance Sum Rerooting   | Efficient recomputation   |
| 479 | Tree Coloring DP         | Combinatorial counting    |
| 480 | Binary Search on Tree DP | Monotonic transitions     |

## 49. DP Reconstruction and Traceback

| #   | Algorithm                   | Note                           |
|-----|-----------------------------|--------------------------------|
| 481 | Reconstruct LCS             | Backtrack table                |
| 482 | Reconstruct LIS             | Track predecessors             |
| 483 | Reconstruct Knapsack        | Recover selected items         |
| 484 | Edit Distance Alignment     | Trace insert/delete/substitute |
| 485 | Matrix Chain Parentheses    | Rebuild parenthesization       |
| 486 | Coin Change Reconstruction  | Backtrack last used coin       |
| 487 | Path Reconstruction DP      | Trace minimal route            |
| 488 | Sequence Reconstruction     | Rebuild from states            |
| 489 | Multi-Choice Reconstruction | Combine best subpaths          |
| 490 | Traceback Visualization     | Visual DP backtrack tool       |

## 50. Meta-DP and Optimization Templates

| #   | Algorithm                        | Note                        |
|-----|----------------------------------|-----------------------------|
| 491 | State Compression Template       | Represent subsets compactly |
| 492 | Transition Optimization Template | Precompute transitions      |
| 493 | Space Optimization Template      | Rolling arrays              |
| 494 | Multi-Dimensional DP Template    | Nested loops version        |
| 495 | Decision Monotonicity            | Optimization hint           |
| 496 | Monge Array Optimization         | Matrix property leverage    |
| 497 | Divide & Conquer Template        | Half-split recursion        |
| 498 | Rerooting Template               | Generalized tree DP         |
| 499 | Iterative DP Pattern             | Bottom-up unrolling         |
| 500 | Memoization Template             | Recursive caching skeleton  |

## Chapter 6. Mathematics for Algorithms

### 51. Number Theory (GCD, Modular Arithmetic, CRT)

| #   | Algorithm                    | Note                               |
|-----|------------------------------|------------------------------------|
| 501 | Euclidean Algorithm          | Compute gcd(a, b)                  |
| 502 | Extended Euclidean Algorithm | Solve $ax + by = \text{gcd}(a, b)$ |
| 503 | Modular Addition             | Add under modulo M                 |
| 504 | Modular Multiplication       | Multiply under modulo M            |
| 505 | Modular Exponentiation       | Fast power mod M                   |
| 506 | Modular Inverse              | Compute $a^{-1} \bmod M$           |
| 507 | Chinese Remainder Theorem    | Combine modular systems            |

| #   | Algorithm                      | Note                         |
|-----|--------------------------------|------------------------------|
| 508 | Binary GCD (Stein's Algorithm) | Bitwise gcd                  |
| 509 | Modular Reduction              | Normalize residues           |
| 510 | Modular Linear Equation Solver | Solve $ax \equiv b \pmod{m}$ |

## 52. Primality and Factorization (Miller–Rabin, Pollard Rho)

| #   | Algorithm                   | Note                          |
|-----|-----------------------------|-------------------------------|
| 511 | Trial Division              | Simple prime test             |
| 512 | Sieve of Eratosthenes       | Generate primes up to n       |
| 513 | Sieve of Atkin              | Faster sieve variant          |
| 514 | Miller–Rabin Primality Test | Probabilistic primality       |
| 515 | Fermat Primality Test       | Modular power check           |
| 516 | Pollard's Rho               | Randomized factorization      |
| 517 | Pollard's p–1 Method        | Factor using smoothness       |
| 518 | Wheel Factorization         | Skip known composites         |
| 519 | AKS Primality Test          | Deterministic polynomial test |
| 520 | Segmented Sieve             | Prime generation for large n  |

## 53. Combinatorics (Permutations, Combinations, Subsets)

| #   | Algorithm                | Note                       |
|-----|--------------------------|----------------------------|
| 521 | Factorial Precomputation | Build $n!$ table           |
| 522 | $nCr$ Computation        | Use Pascal's or factorials |
| 523 | Pascal's Triangle        | Binomial coefficients      |
| 524 | Multiset Combination     | Repetition allowed         |
| 525 | Permutation Generation   | Lexicographic order        |
| 526 | Next Permutation         | STL-style increment        |
| 527 | Subset Generation        | Bitmask or recursion       |
| 528 | Gray Code Generation     | Single-bit flips           |
| 529 | Catalan Number DP        | Count valid parentheses    |
| 530 | Stirling Numbers         | Partition counting         |

## 54. Probability and Randomized Algorithms

| #   | Algorithm                   | Note                          |
|-----|-----------------------------|-------------------------------|
| 531 | Monte Carlo Simulation      | Approximate via randomness    |
| 532 | Las Vegas Algorithm         | Always correct, variable time |
| 533 | Reservoir Sampling          | Uniform sampling from stream  |
| 534 | Randomized QuickSort        | Expected $O(n \log n)$        |
| 535 | Randomized QuickSelect      | Random pivot                  |
| 536 | Birthday Paradox Simulation | Probability collision         |
| 537 | Random Hashing              | Reduce collision chance       |
| 538 | Random Walk Simulation      | State transitions             |
| 539 | Coupon Collector Estimation | Expected trials               |
| 540 | Markov Chain Simulation     | Transition matrix sampling    |

## 55. Sieve Methods and Modular Math

| #   | Algorithm                         | Note                             |
|-----|-----------------------------------|----------------------------------|
| 541 | Sieve of Eratosthenes             | Base prime sieve                 |
| 542 | Linear Sieve                      | $O(n)$ sieve variant             |
| 543 | Segmented Sieve                   | Range prime generation           |
| 544 | SPF (Smallest Prime Factor) Table | Factorization via sieve          |
| 545 | Möbius Function Sieve             | Multiplicative function calc     |
| 546 | Euler's Totient Sieve             | Compute $(n)$ for all $n$        |
| 547 | Divisor Count Sieve               | Count divisors efficiently       |
| 548 | Modular Precomputation            | Store inverses, factorials       |
| 549 | Fermat Little Theorem             | $a^{(p-1)} \equiv 1 \pmod{p}$    |
| 550 | Wilson's Theorem                  | Prime test via factorial mod $p$ |

## 56. Linear Algebra (Gaussian Elimination, LU, SVD)

| #   | Algorithm                       | Note                            |
|-----|---------------------------------|---------------------------------|
| 551 | Gaussian Elimination            | Solve $Ax = b$                  |
| 552 | Gauss-Jordan Elimination        | Reduced row echelon             |
| 553 | LU Decomposition                | Factor $A$ into $L \cdot U$     |
| 554 | Cholesky Decomposition          | $A = L \cdot L^T$ for SPD       |
| 555 | QR Decomposition                | Orthogonal factorization        |
| 556 | Matrix Inversion (Gauss-Jordan) | Find $A^{-1}$                   |
| 557 | Determinant by Elimination      | Product of pivots               |
| 558 | Rank of Matrix                  | Count non-zero rows             |
| 559 | Eigenvalue Power Method         | Approximate dominant eigenvalue |

| #   | Algorithm                    | Note            |
|-----|------------------------------|-----------------|
| 560 | Singular Value Decomposition | $A = U\Sigma V$ |

## 57. FFT and NTT (Fast Transforms)

| #   | Algorithm                        | Note                         |
|-----|----------------------------------|------------------------------|
| 561 | Discrete Fourier Transform (DFT) | $O(n^2)$ baseline            |
| 562 | Fast Fourier Transform (FFT)     | $O(n \log n)$ convolution    |
| 563 | Cooley–Tukey FFT                 | Recursive divide and conquer |
| 564 | Iterative FFT                    | In-place bit reversal        |
| 565 | Inverse FFT                      | Recover time-domain          |
| 566 | Convolution via FFT              | Polynomial multiplication    |
| 567 | Number Theoretic Transform (NTT) | Modulo prime FFT             |
| 568 | Inverse NTT                      | Modular inverse transform    |
| 569 | Bluestein's Algorithm            | FFT of arbitrary size        |
| 570 | FFT-Based Multiplication         | Big integer product          |

## 58. Numerical Methods (Newton, Simpson, Runge–Kutta)

| #   | Algorithm             | Note                          |
|-----|-----------------------|-------------------------------|
| 571 | Newton–Raphson        | Root finding via tangent      |
| 572 | Bisection Method      | Interval halving              |
| 573 | Secant Method         | Approximate derivative        |
| 574 | Fixed-Point Iteration | $x = f(x)$ convergence        |
| 575 | Gaussian Quadrature   | Weighted integration          |
| 576 | Simpson's Rule        | Piecewise quadratic integral  |
| 577 | Trapezoidal Rule      | Linear interpolation integral |
| 578 | Runge–Kutta (RK4)     | ODE solver                    |
| 579 | Euler's Method        | Step-by-step ODE              |
| 580 | Gradient Descent (1D) | Numerical optimization        |

## 59. Mathematical Optimization (Simplex, Gradient, Convex)

| #   | Algorithm           | Note                      |
|-----|---------------------|---------------------------|
| 581 | Simplex Method      | Linear programming solver |
| 582 | Dual Simplex Method | Solve dual constraints    |

| #   | Algorithm                      | Note                        |
|-----|--------------------------------|-----------------------------|
| 583 | Interior-Point Method          | Convex optimization         |
| 584 | Gradient Descent               | Unconstrained optimization  |
| 585 | Stochastic Gradient Descent    | Sample-based updates        |
| 586 | Newton's Method (Multivariate) | Quadratic convergence       |
| 587 | Conjugate Gradient             | Solve SPD systems           |
| 588 | Lagrange Multipliers           | Constrained optimization    |
| 589 | KKT Conditions Solver          | Convex constraint handling  |
| 590 | Coordinate Descent             | Sequential variable updates |

## 60. Algebraic Tricks and Transform Techniques

| #   | Algorithm                       | Note                        |
|-----|---------------------------------|-----------------------------|
| 591 | Polynomial Multiplication (FFT) | Fast convolution            |
| 592 | Polynomial Inversion            | Newton iteration            |
| 593 | Polynomial Derivative           | Term-wise multiply by index |
| 594 | Polynomial Integration          | Divide by index+1           |
| 595 | Formal Power Series Composition | Substitute series           |
| 596 | Exponentiation by Squaring      | Fast powering               |
| 597 | Modular Exponentiation          | Fast power mod M            |
| 598 | Fast Walsh–Hadamard Transform   | XOR convolution             |
| 599 | Zeta Transform                  | Subset summation            |
| 600 | Möbius Inversion                | Recover original from sums  |

## Chapter 7. Strings and Text Algorithms

### 61. String Matching (KMP, Z, Rabin–Karp, Boyer–Moore)

| #   | Algorithm                 | Note                            |
|-----|---------------------------|---------------------------------|
| 601 | Naive String Matching     | Compare every position          |
| 602 | Knuth–Morris–Pratt (KMP)  | Prefix function skipping        |
| 603 | Z-Algorithm               | Match using Z-values            |
| 604 | Rabin–Karp                | Rolling hash comparison         |
| 605 | Boyer–Moore               | Backward skip based on mismatch |
| 606 | Boyer–Moore–Horspool      | Simplified shift table          |
| 607 | Sunday Algorithm          | Last-character shift            |
| 608 | Finite Automaton Matching | DFA-based matching              |
| 609 | Bitap Algorithm           | Bitmask approximate matching    |

| #   | Algorithm         | Note                    |
|-----|-------------------|-------------------------|
| 610 | Two-Way Algorithm | Optimal linear matching |

## 62. Multi-Pattern Search (Aho–Corasick)

| #   | Algorithm                  | Note                        |
|-----|----------------------------|-----------------------------|
| 611 | Aho–Corasick Automaton     | Trie + failure links        |
| 612 | Trie Construction          | Prefix tree build           |
| 613 | Failure Link Computation   | BFS for transitions         |
| 614 | Output Link Management     | Handle overlapping patterns |
| 615 | Multi-Pattern Search       | Find all keywords           |
| 616 | Dictionary Matching        | Find multiple substrings    |
| 617 | Dynamic Aho–Corasick       | Add/remove patterns         |
| 618 | Parallel AC Search         | Multi-threaded traversal    |
| 619 | Compressed AC Automaton    | Memory-optimized            |
| 620 | Extended AC with Wildcards | Flexible matching           |

## 63. Suffix Structures (Suffix Array, Suffix Tree, LCP)

| #   | Algorithm                | Note                        |
|-----|--------------------------|-----------------------------|
| 621 | Suffix Array (Naive)     | Sort all suffixes           |
| 622 | Suffix Array (Doubling)  | $O(n \log n)$ rank-based    |
| 623 | Kasai's LCP Algorithm    | Longest common prefix       |
| 624 | Suffix Tree (Ukkonen)    | Linear-time online          |
| 625 | Suffix Automaton         | Minimal DFA of substrings   |
| 626 | SA-IS Algorithm          | $O(n)$ suffix array         |
| 627 | LCP RMQ Query            | Range minimum for substring |
| 628 | Generalized Suffix Array | Multiple strings            |
| 629 | Enhanced Suffix Array    | Combine SA + LCP            |
| 630 | Sparse Suffix Tree       | Space-efficient variant     |

## 64. Palindromes and Periodicity (Manacher)

| #   | Algorithm              | Note                      |
|-----|------------------------|---------------------------|
| 631 | Naive Palindrome Check | Expand around center      |
| 632 | Manacher's Algorithm   | $O(n)$ longest palindrome |

| #   | Algorithm                            | Note                               |
|-----|--------------------------------------|------------------------------------|
| 633 | Longest Palindromic Substring        | Center expansion                   |
| 634 | Palindrome DP Table                  | Substring boolean matrix           |
| 635 | Palindromic Tree (Eertree)           | Track distinct palindromes         |
| 636 | Prefix Function Periodicity          | Detect repetition patterns         |
| 637 | Z-Function Periodicity               | Identify periodic suffix           |
| 638 | KMP Prefix Period Check              | Shortest repeating unit            |
| 639 | Lyndon Factorization                 | Decompose string into Lyndon words |
| 640 | Minimal Rotation (Booth's Algorithm) | Lexicographically minimal shift    |

## 65. Edit Distance and Alignment

| #   | Algorithm                  | Note                       |
|-----|----------------------------|----------------------------|
| 641 | Levenshtein Distance       | Insert/delete/replace cost |
| 642 | Damerau–Levenshtein        | Swap included              |
| 643 | Hamming Distance           | Count differing bits       |
| 644 | Needleman–Wunsch           | Global alignment           |
| 645 | Smith–Waterman             | Local alignment            |
| 646 | Hirschberg's Algorithm     | Memory-optimized alignment |
| 647 | Edit Script Reconstruction | Backtrack operations       |
| 648 | Affine Gap Penalty DP      | Varying gap cost           |
| 649 | Myers Bit-Vector Algorithm | Fast edit distance         |
| 650 | Longest Common Subsequence | Alignment by inclusion     |

## 66. Compression (Huffman, Arithmetic, LZ77, BWT)

| #   | Algorithm                 | Note                        |
|-----|---------------------------|-----------------------------|
| 651 | Huffman Coding            | Optimal prefix tree         |
| 652 | Canonical Huffman         | Deterministic ordering      |
| 653 | Arithmetic Coding         | Interval probability coding |
| 654 | Shannon–Fano Coding       | Early prefix method         |
| 655 | Run-Length Encoding (RLE) | Repeat compression          |
| 656 | LZ77                      | Sliding-window match        |
| 657 | LZ78                      | Dictionary building         |
| 658 | LZW                       | Variant used in GIF         |
| 659 | Burrows–Wheeler Transform | Block reordering            |
| 660 | Move-to-Front Encoding    | Locality boosting transform |

## 67. Cryptographic Hashes and Checksums

| #   | Algorithm                | Note                         |
|-----|--------------------------|------------------------------|
| 661 | Rolling Hash             | Polynomial mod-based         |
| 662 | CRC32                    | Cyclic redundancy check      |
| 663 | Adler-32                 | Lightweight checksum         |
| 664 | MD5                      | Legacy cryptographic hash    |
| 665 | SHA-1                    | Deprecated hash function     |
| 666 | SHA-256                  | Secure hash standard         |
| 667 | SHA-3 (Keccak)           | Sponge construction          |
| 668 | HMAC                     | Keyed message authentication |
| 669 | Merkle Tree              | Hierarchical hashing         |
| 670 | Hash Collision Detection | Birthday bound simulation    |

## 68. Approximate and Streaming Matching

| #   | Algorithm                | Note                             |
|-----|--------------------------|----------------------------------|
| 671 | K-Approximate Matching   | Allow k mismatches               |
| 672 | Bitap Algorithm          | Bitwise dynamic programming      |
| 673 | Landau–Vishkin Algorithm | Edit distance k                  |
| 674 | Filtering Algorithm      | Fast approximate search          |
| 675 | Wu–Manber                | Multi-pattern approximate search |
| 676 | Streaming KMP            | Online prefix updates            |
| 677 | Rolling Hash Sketch      | Sliding window hashing           |
| 678 | Sketch-based Similarity  | MinHash / LSH variants           |
| 679 | Weighted Edit Distance   | Weighted operations              |
| 680 | Online Levenshtein       | Dynamic stream update            |

## 69. Bioinformatics Alignment (Needleman–Wunsch, Smith–Waterman)

| #   | Algorithm                         | Note                      |
|-----|-----------------------------------|---------------------------|
| 681 | Needleman–Wunsch                  | Global sequence alignment |
| 682 | Smith–Waterman                    | Local alignment           |
| 683 | Gotoh Algorithm                   | Affine gap penalties      |
| 684 | Hirschberg Alignment              | Linear-space alignment    |
| 685 | Multiple Sequence Alignment (MSA) | Progressive methods       |
| 686 | Profile Alignment                 | Align sequence to profile |

| #   | Algorithm                     | Note                    |
|-----|-------------------------------|-------------------------|
| 687 | Hidden Markov Model Alignment | Probabilistic alignment |
| 688 | BLAST                         | Heuristic local search  |
| 689 | FASTA                         | Word-based alignment    |
| 690 | Pairwise DP Alignment         | General DP framework    |

## 70. Text Indexing and Search Structures

| #   | Algorithm                          | Note                       |
|-----|------------------------------------|----------------------------|
| 691 | Inverted Index Build               | Word-to-document mapping   |
| 692 | Positional Index                   | Store word positions       |
| 693 | TF-IDF Weighting                   | Importance scoring         |
| 694 | BM25 Ranking                       | Modern ranking formula     |
| 695 | Trie Index                         | Prefix search structure    |
| 696 | Suffix Array Index                 | Substring search           |
| 697 | Compressed Suffix Array            | Space-optimized            |
| 698 | FM-Index                           | BWT-based compressed index |
| 699 | DAWG (Directed Acyclic Word Graph) | Shared suffix graph        |
| 700 | Wavelet Tree for Text              | Rank/select on sequences   |

## Chapter 8. Geometry, Graphics, and Spatial Algorithms

### 71. Convex Hull (Graham, Andrew, Chan)

| #   | Algorithm                    | Note                                     |
|-----|------------------------------|------------------------------------------|
| 701 | Gift Wrapping (Jarvis March) | Wrap hull one point at a time            |
| 702 | Graham Scan                  | Sort by angle, maintain stack            |
| 703 | Andrew's Monotone Chain      | Sort by x, upper + lower hull            |
| 704 | Chan's Algorithm             | Output-sensitive $O(n \log h)$           |
| 705 | QuickHull                    | Divide-and-conquer hull                  |
| 706 | Incremental Convex Hull      | Add points one by one                    |
| 707 | Divide & Conquer Hull        | Merge two partial hulls                  |
| 708 | 3D Convex Hull               | Extend to 3D geometry                    |
| 709 | Dynamic Convex Hull          | Maintain hull with inserts               |
| 710 | Rotating Calipers            | Compute diameter, width, antipodal pairs |

## 72. Closest Pair and Segment Intersection

| #   | Algorithm                       | Note                          |
|-----|---------------------------------|-------------------------------|
| 711 | Closest Pair (Divide & Conquer) | Split, merge minimal distance |
| 712 | Closest Pair (Sweep Line)       | Maintain active window        |
| 713 | Brute Force Closest Pair        | Check all $O(n^2)$ pairs      |
| 714 | Bentley–Ottmann                 | Find all line intersections   |
| 715 | Segment Intersection Test       | Cross product orientation     |
| 716 | Line Sweep for Segments         | Event-based intersection      |
| 717 | Intersection via Orientation    | CCW test                      |
| 718 | Circle Intersection             | Geometry of two circles       |
| 719 | Polygon Intersection            | Clip overlapping polygons     |
| 720 | Nearest Neighbor Pair           | Combine KD-tree + search      |

## 73. Line Sweep and Plane Sweep Algorithms

| #   | Algorithm                              | Note                             |
|-----|----------------------------------------|----------------------------------|
| 721 | Sweep Line for Events                  | Process sorted events            |
| 722 | Interval Scheduling                    | Select non-overlapping intervals |
| 723 | Rectangle Union Area                   | Sweep edges to count area        |
| 724 | Segment Intersection (Bentley–Ottmann) | Detect all crossings             |
| 725 | Skyline Problem                        | Merge height profiles            |
| 726 | Closest Pair Sweep                     | Maintain active set              |
| 727 | Circle Arrangement                     | Sweep and count regions          |
| 728 | Sweep for Overlapping Rectangles       | Detect collisions                |
| 729 | Range Counting                         | Count points in rectangle        |
| 730 | Plane Sweep for Triangles              | Polygon overlay computation      |

## 74. Delaunay and Voronoi Diagrams

| #   | Algorithm                            | Note                          |
|-----|--------------------------------------|-------------------------------|
| 731 | Delaunay Triangulation (Incremental) | Add points, maintain Delaunay |
| 732 | Delaunay (Divide & Conquer)          | Merge triangulations          |
| 733 | Delaunay (Fortune's Sweep)           | $O(n \log n)$ construction    |
| 734 | Voronoi Diagram (Fortune's)          | Sweep line beachline          |
| 735 | Incremental Voronoi                  | Update on insertion           |
| 736 | Bowyer–Watson                        | Empty circle criterion        |

| #   | Algorithm                | Note                             |
|-----|--------------------------|----------------------------------|
| 737 | Duality Transform        | Convert between Voronoi/Delaunay |
| 738 | Power Diagram            | Weighted Voronoi                 |
| 739 | Lloyd's Relaxation       | Smooth Voronoi cells             |
| 740 | Voronoi Nearest Neighbor | Region-based lookup              |

## 75. Point in Polygon and Polygon Triangulation

| #   | Algorithm                              | Note                     |
|-----|----------------------------------------|--------------------------|
| 741 | Ray Casting                            | Count edge crossings     |
| 742 | Winding Number                         | Angle sum method         |
| 743 | Convex Polygon Point Test              | Orientation checks       |
| 744 | Ear Clipping Triangulation             | Remove ears iteratively  |
| 745 | Monotone Polygon Triangulation         | Sweep line triangulation |
| 746 | Delaunay Triangulation                 | Optimal triangle quality |
| 747 | Convex Decomposition                   | Split into convex parts  |
| 748 | Polygon Area (Shoelace Formula)        | Signed area computation  |
| 749 | Minkowski Sum                          | Add shapes geometrically |
| 750 | Polygon Intersection (Weiler–Atherton) | Clip overlapping shapes  |

## 76. Spatial Data Structures (KD, R-tree)

| #   | Algorithm                         | Note                     |
|-----|-----------------------------------|--------------------------|
| 751 | KD-Tree Build                     | Recursive median split   |
| 752 | KD-Tree Search                    | Axis-aligned query       |
| 753 | Range Search KD-Tree              | Orthogonal query         |
| 754 | Nearest Neighbor KD-Tree          | Closest point search     |
| 755 | R-Tree Build                      | Bounding box hierarchy   |
| 756 | R*-Tree                           | Optimized split strategy |
| 757 | Quad Tree                         | Spatial decomposition    |
| 758 | Octree                            | 3D spatial decomposition |
| 759 | BSP Tree (Binary Space Partition) | Split by planes          |
| 760 | Morton Order (Z-Curve)            | Spatial locality index   |

## 77. Rasterization and Scanline Techniques

| #   | Algorithm                     | Note                         |
|-----|-------------------------------|------------------------------|
| 761 | Bresenham's Line Algorithm    | Efficient integer drawing    |
| 762 | Midpoint Circle Algorithm     | Circle rasterization         |
| 763 | Scanline Fill                 | Polygon interior fill        |
| 764 | Edge Table Fill               | Sort edges by y              |
| 765 | Z-Buffer Algorithm            | Hidden surface removal       |
| 766 | Painter's Algorithm           | Sort by depth                |
| 767 | Gouraud Shading               | Vertex interpolation shading |
| 768 | Phong Shading                 | Normal interpolation         |
| 769 | Anti-Aliasing (Supersampling) | Smooth jagged edges          |
| 770 | Scanline Polygon Clipping     | Efficient clipping           |

## 78. Computer Vision (Canny, Hough, SIFT)

| #   | Algorithm                                | Note                           |
|-----|------------------------------------------|--------------------------------|
| 771 | Canny Edge Detector                      | Gradient + hysteresis          |
| 772 | Sobel Operator                           | Gradient magnitude filter      |
| 773 | Hough Transform (Lines)                  | Accumulator for line detection |
| 774 | Hough Transform (Circles)                | Radius-based accumulator       |
| 775 | Harris Corner Detector                   | Eigenvalue-based corners       |
| 776 | FAST Corner Detector                     | Intensity circle test          |
| 777 | SIFT (Scale-Invariant Feature Transform) | Keypoint detection             |
| 778 | SURF (Speeded-Up Robust Features)        | Faster descriptor              |
| 779 | ORB (Oriented FAST + BRIEF)              | Binary robust feature          |
| 780 | RANSAC                                   | Robust model fitting           |

## 79. Pathfinding in Space (A\*, RRT, PRM)

| #   | Algorithm                           | Note                          |
|-----|-------------------------------------|-------------------------------|
| 781 | A* Search                           | Heuristic pathfinding         |
| 782 | Dijkstra for Grid                   | Weighted shortest path        |
| 783 | Theta*                              | Any-angle pathfinding         |
| 784 | Jump Point Search                   | Grid acceleration             |
| 785 | RRT (Rapidly-Exploring Random Tree) | Random sampling tree          |
| 786 | RRT*                                | Optimal variant with rewiring |
| 787 | PRM (Probabilistic Roadmap)         | Graph sampling planner        |
| 788 | Visibility Graph                    | Connect visible vertices      |
| 789 | Potential Field Pathfinding         | Gradient-based navigation     |

| #   | Algorithm      | Note                      |
|-----|----------------|---------------------------|
| 790 | Bug Algorithms | Simple obstacle avoidance |

## 80. Computational Geometry Variants and Applications

| #   | Algorithm                        | Note                     |
|-----|----------------------------------|--------------------------|
| 791 | Convex Polygon Intersection      | Clip convex sets         |
| 792 | Minkowski Sum                    | Shape convolution        |
| 793 | Rotating Calipers                | Closest/farthest pair    |
| 794 | Half-Plane Intersection          | Feasible region          |
| 795 | Line Arrangement                 | Count regions            |
| 796 | Point Location (Trapezoidal Map) | Query region lookup      |
| 797 | Voronoi Nearest Facility         | Region query             |
| 798 | Delaunay Mesh Generation         | Triangulation refinement |
| 799 | Smallest Enclosing Circle        | Welzl's algorithm        |
| 800 | Collision Detection (SAT)        | Separating axis theorem  |

## Chapter 9. Systems, Databases, and Distributed Algorithms

### 81. Concurrency Control (2PL, MVCC, OCC)

| #   | Algorithm                               | Note                          |
|-----|-----------------------------------------|-------------------------------|
| 801 | Two-Phase Locking (2PL)                 | Acquire-then-release locks    |
| 802 | Strict 2PL                              | Hold locks until commit       |
| 803 | Conservative 2PL                        | Prevent deadlocks via prelock |
| 804 | Timestamp Ordering                      | Schedule by timestamps        |
| 805 | Multiversion Concurrency Control (MVCC) | Snapshot isolation            |
| 806 | Optimistic Concurrency Control (OCC)    | Validate at commit            |
| 807 | Serializable Snapshot Isolation         | Merge read/write sets         |
| 808 | Lock-Free Algorithm                     | Atomic CAS updates            |
| 809 | Wait-Die / Wound-Wait                   | Deadlock prevention policies  |
| 810 | Deadlock Detection (Wait-for Graph)     | Cycle detection in waits      |

### 82. Logging, Recovery, and Commit Protocols

| #   | Algorithm                 | Note                      |
|-----|---------------------------|---------------------------|
| 811 | Write-Ahead Logging (WAL) | Log before commit         |
| 812 | ARIES Recovery            | Re-do/undo with LSNs      |
| 813 | Shadow Paging             | Copy-on-write persistence |
| 814 | Two-Phase Commit (2PC)    | Coordinator-driven commit |
| 815 | Three-Phase Commit (3PC)  | Non-blocking variant      |
| 816 | Checkpointing             | Save state for recovery   |
| 817 | Undo Logging              | Rollback uncommitted      |
| 818 | Redo Logging              | Reapply committed         |
| 819 | Quorum Commit             | Majority agreement        |
| 820 | Consensus Commit          | Combine 2PC + Paxos       |

### 83. Scheduling (Round Robin, EDF, Rate-Monotonic)

| #   | Algorithm                       | Note                            |
|-----|---------------------------------|---------------------------------|
| 821 | First-Come First-Served (FCFS)  | Sequential job order            |
| 822 | Shortest Job First (SJF)        | Optimal average wait            |
| 823 | Round Robin (RR)                | Time-slice fairness             |
| 824 | Priority Scheduling             | Weighted selection              |
| 825 | Multilevel Queue                | Tiered priority queues          |
| 826 | Earliest Deadline First (EDF)   | Real-time optimal               |
| 827 | Rate Monotonic Scheduling (RMS) | Fixed periodic priority         |
| 828 | Lottery Scheduling              | Probabilistic fairness          |
| 829 | Multilevel Feedback Queue       | Adaptive behavior               |
| 830 | Fair Queuing (FQ)               | Flow-based proportional sharing |

### 84. Caching and Replacement (LRU, LFU, CLOCK)

| #   | Algorithm                              | Note                       |
|-----|----------------------------------------|----------------------------|
| 831 | LRU (Least Recently Used)              | Evict oldest used          |
| 832 | LFU (Least Frequently Used)            | Evict lowest frequency     |
| 833 | FIFO Cache                             | Simple queue eviction      |
| 834 | CLOCK Algorithm                        | Approximate LRU            |
| 835 | ARC (Adaptive Replacement Cache)       | Mix of recency + frequency |
| 836 | Two-Queue (2Q)                         | Separate recent/frequent   |
| 837 | LIRS (Low Inter-reference Recency Set) | Predict reuse distance     |
| 838 | TinyLFU                                | Frequency sketch admission |
| 839 | Random Replacement                     | Simple stochastic policy   |

| #   | Algorithm        | Note                      |
|-----|------------------|---------------------------|
| 840 | Belady's Optimal | Evict farthest future use |

## 85. Networking (Routing, Congestion Control)

| #   | Algorithm                              | Note                    |
|-----|----------------------------------------|-------------------------|
| 841 | Dijkstra's Routing                     | Shortest path routing   |
| 842 | Bellman–Ford Routing                   | Distance-vector routing |
| 843 | Link-State Routing (OSPF)              | Global view routing     |
| 844 | Distance-Vector Routing (RIP)          | Local neighbor updates  |
| 845 | Path Vector (BGP)                      | Route advertisement     |
| 846 | Flooding                               | Broadcast to all nodes  |
| 847 | Spanning Tree Protocol                 | Loop-free topology      |
| 848 | Congestion Control (AIMD)              | TCP window control      |
| 849 | Random Early Detection (RED)           | Queue preemptive drop   |
| 850 | ECN (Explicit Congestion Notification) | Mark packets early      |

## 86. Distributed Consensus (Paxos, Raft, PBFT)

| #   | Algorithm                                  | Note                              |
|-----|--------------------------------------------|-----------------------------------|
| 851 | Basic Paxos                                | Majority consensus                |
| 852 | Multi-Paxos                                | Sequence of agreements            |
| 853 | Raft                                       | Log replication + leader election |
| 854 | Viewstamped Replication                    | Alternative consensus design      |
| 855 | PBFT (Practical Byzantine Fault Tolerance) | Byzantine safety                  |
| 856 | Zab (Zookeeper Atomic Broadcast)           | Broadcast + ordering              |
| 857 | EPaxos                                     | Leaderless fast path              |
| 858 | VRR (Virtual Ring Replication)             | Log around ring                   |
| 859 | Two-Phase Commit with Consensus            | Transactional commit              |
| 860 | Chain Replication                          | Ordered state replication         |

## 87. Load Balancing and Rate Limiting

| #   | Algorithm                  | Note                    |
|-----|----------------------------|-------------------------|
| 861 | Round Robin Load Balancing | Sequential distribution |
| 862 | Weighted Round Robin       | Proportional to weight  |

| #   | Algorithm              | Note                          |
|-----|------------------------|-------------------------------|
| 863 | Least Connections      | Pick least loaded node        |
| 864 | Consistent Hashing     | Map requests stably           |
| 865 | Power of Two Choices   | Sample and choose lesser load |
| 866 | Random Load Balancing  | Simple uniform random         |
| 867 | Token Bucket           | Rate-based limiter            |
| 868 | Leaky Bucket           | Steady flow shaping           |
| 869 | Sliding Window Counter | Rolling time window           |
| 870 | Fixed Window Counter   | Resettable counter limiter    |

## 88. Search and Indexing (Inverted, BM25, WAND)

| #   | Algorithm                   | Note                        |
|-----|-----------------------------|-----------------------------|
| 871 | Inverted Index Construction | Word → document list        |
| 872 | Positional Index Build      | Store term positions        |
| 873 | TF-IDF Scoring              | Term frequency weighting    |
| 874 | BM25 Ranking                | Modern scoring model        |
| 875 | Boolean Retrieval           | Logical AND/OR/NOT          |
| 876 | WAND Algorithm              | Efficient top-k retrieval   |
| 877 | Block-Max WAND (BMW)        | Early skipping optimization |
| 878 | Impact-Ordered Indexing     | Sort by contribution        |
| 879 | Tiered Indexing             | Prioritize high-score docs  |
| 880 | DAAT vs SAAT Evaluation     | Document vs score-at-a-time |

## 89. Compression and Encoding in Systems

| #   | Algorithm                 | Note                             |
|-----|---------------------------|----------------------------------|
| 881 | Run-Length Encoding (RLE) | Simple repetition encoding       |
| 882 | Huffman Coding            | Optimal variable-length code     |
| 883 | Arithmetic Coding         | Fractional interval coding       |
| 884 | Delta Encoding            | Store differences                |
| 885 | Variable Byte Encoding    | Compact integers                 |
| 886 | Elias Gamma Coding        | Prefix integer encoding          |
| 887 | Rice Coding               | Unary + remainder scheme         |
| 888 | Snappy                    | Fast block compression           |
| 889 | Zstandard (Zstd)          | Modern adaptive codec            |
| 890 | LZ4                       | High-speed dictionary compressor |

## 90. Fault Tolerance and Replication

| #   | Algorithm                  | Note                     |
|-----|----------------------------|--------------------------|
| 891 | Primary–Backup Replication | One leader, one standby  |
| 892 | Quorum Replication         | Majority write/read rule |
| 893 | Chain Replication          | Ordered consistency      |
| 894 | Gossip Protocol            | Epidemic state exchange  |
| 895 | Anti-Entropy Repair        | Periodic reconciliation  |
| 896 | Erasure Coding             | Redundant data blocks    |
| 897 | Checksum Verification      | Detect corruption        |
| 898 | Heartbeat Monitoring       | Liveness detection       |
| 899 | Leader Election (Bully)    | Highest ID wins          |
| 900 | Leader Election (Ring)     | Token-based rotation     |

## Chapter 10. AI, ML, and Optimization

### 91. Classical ML (k-means, Naive Bayes, SVM, Decision Trees)

| #   | Algorithm                          | Note                               |
|-----|------------------------------------|------------------------------------|
| 901 | k-Means Clustering                 | Partition by centroid iteration    |
| 902 | k-Medoids (PAM)                    | Cluster by exemplars               |
| 903 | Gaussian Mixture Model (EM)        | Soft probabilistic clustering      |
| 904 | Naive Bayes Classifier             | Probabilistic feature independence |
| 905 | Logistic Regression                | Sigmoid linear classifier          |
| 906 | Perceptron                         | Online linear separator            |
| 907 | Decision Tree (CART)               | Recursive partition by impurity    |
| 908 | ID3 Algorithm                      | Information gain splitting         |
| 909 | k-Nearest Neighbors (kNN)          | Distance-based classification      |
| 910 | Linear Discriminant Analysis (LDA) | Projection for separation          |

### 92. Ensemble Methods (Bagging, Boosting, Random Forests)

| #   | Algorithm         | Note                        |
|-----|-------------------|-----------------------------|
| 911 | Bagging           | Bootstrap aggregation       |
| 912 | Random Forest     | Ensemble of decision trees  |
| 913 | AdaBoost          | Weighted error correction   |
| 914 | Gradient Boosting | Sequential residual fitting |

| #   | Algorithm         | Note                              |
|-----|-------------------|-----------------------------------|
| 915 | XGBoost           | Optimized gradient boosting       |
| 916 | LightGBM          | Histogram-based leaf growth       |
| 917 | CatBoost          | Ordered boosting for categoricals |
| 918 | Stacking          | Meta-model ensemble               |
| 919 | Voting Classifier | Majority aggregation              |
| 920 | Snapshot Ensemble | Averaged checkpoints              |

### 93. Gradient Methods (SGD, Adam, RMSProp)

| #   | Algorithm                         | Note                        |
|-----|-----------------------------------|-----------------------------|
| 921 | Gradient Descent                  | Batch full-gradient step    |
| 922 | Stochastic Gradient Descent (SGD) | Sample-wise updates         |
| 923 | Mini-Batch SGD                    | Tradeoff speed and variance |
| 924 | Momentum                          | Add velocity to descent     |
| 925 | Nesterov Accelerated Gradient     | Lookahead correction        |
| 926 | AdaGrad                           | Adaptive per-parameter rate |
| 927 | RMSProp                           | Exponential moving average  |
| 928 | Adam                              | Momentum + adaptive rate    |
| 929 | AdamW                             | Decoupled weight decay      |
| 930 | L-BFGS                            | Limited-memory quasi-Newton |

### 94. Deep Learning (Backpropagation, Dropout, Normalization)

| #   | Algorithm                | Note                       |
|-----|--------------------------|----------------------------|
| 931 | Backpropagation          | Gradient chain rule        |
| 932 | Xavier/He Initialization | Scaled variance init       |
| 933 | Dropout                  | Random neuron deactivation |
| 934 | Batch Normalization      | Normalize per batch        |
| 935 | Layer Normalization      | Normalize per feature      |
| 936 | Gradient Clipping        | Prevent explosion          |
| 937 | Early Stopping           | Prevent overfitting        |
| 938 | Weight Decay             | Regularization via penalty |
| 939 | Learning Rate Scheduling | Dynamic LR adjustment      |
| 940 | Residual Connections     | Skip layer improvement     |

## 95. Sequence Models (Viterbi, Beam Search, CTC)

| #   | Algorithm                                   | Note                         |
|-----|---------------------------------------------|------------------------------|
| 941 | Hidden Markov Model (Forward–Backward)      | Probabilistic sequence model |
| 942 | Viterbi Algorithm                           | Most probable path           |
| 943 | Baum–Welch                                  | EM training for HMMs         |
| 944 | Beam Search                                 | Top-k path exploration       |
| 945 | Greedy Decoding                             | Fast approximate decoding    |
| 946 | Connectionist Temporal Classification (CTC) | Unaligned sequence training  |
| 947 | Attention Mechanism                         | Weighted context aggregation |
| 948 | Transformer Decoder                         | Self-attention stack         |
| 949 | Seq2Seq with Attention                      | Encoder-decoder framework    |
| 950 | Pointer Network                             | Output index selection       |

## 96. Metaheuristics (GA, SA, PSO, ACO)

| #   | Algorithm                         | Note                           |
|-----|-----------------------------------|--------------------------------|
| 951 | Genetic Algorithm (GA)            | Evolutionary optimization      |
| 952 | Simulated Annealing (SA)          | Temperature-controlled search  |
| 953 | Tabu Search                       | Memory of forbidden moves      |
| 954 | Particle Swarm Optimization (PSO) | Velocity-based search          |
| 955 | Ant Colony Optimization (ACO)     | Pheromone-guided path          |
| 956 | Differential Evolution (DE)       | Vector-based mutation          |
| 957 | Harmony Search                    | Music-inspired improvisation   |
| 958 | Firefly Algorithm                 | Brightness-attraction movement |
| 959 | Bee Colony Optimization           | Explore-exploit via scouts     |
| 960 | Hill Climbing                     | Local incremental improvement  |

## 97. Reinforcement Learning (Q-learning, Policy Gradients)

| #   | Algorithm                         | Note                   |
|-----|-----------------------------------|------------------------|
| 961 | Monte Carlo Control               | Average returns        |
| 962 | Temporal Difference (TD) Learning | Bootstrap updates      |
| 963 | SARSA                             | On-policy TD learning  |
| 964 | Q-Learning                        | Off-policy TD learning |
| 965 | Double Q-Learning                 | Reduce overestimation  |
| 966 | Deep Q-Network (DQN)              | Neural Q approximator  |

| #   | Algorithm                          | Note                        |
|-----|------------------------------------|-----------------------------|
| 967 | REINFORCE                          | Policy gradient by sampling |
| 968 | Actor–Critic                       | Value-guided policy update  |
| 969 | PPO (Proximal Policy Optimization) | Clipped surrogate objective |
| 970 | DDPG / SAC                         | Continuous action RL        |

## 98. Approximation and Online Algorithms

| #   | Algorithm                        | Note                                   |
|-----|----------------------------------|----------------------------------------|
| 971 | Greedy Set Cover                 | $\ln(n)$ -approximation                |
| 972 | Vertex Cover Approximation       | Double-matching heuristic              |
| 973 | Traveling Salesman Approximation | MST-based 2-approx                     |
| 974 | k-Center Approximation           | Farthest-point heuristic               |
| 975 | Online Paging (LRU)              | Competitive analysis                   |
| 976 | Online Matching (Ranking)        | Adversarial input resilience           |
| 977 | Online Knapsack                  | Ratio-based acceptance                 |
| 978 | Competitive Ratio Evaluation     | Bound worst-case performance           |
| 979 | PTAS / FPTAS Schemes             | Polynomial approximation               |
| 980 | Primal–Dual Method               | Approximate combinatorial optimization |

## 99. Fairness, Causal Inference, and Robust Optimization

| #   | Algorithm                            | Note                          |
|-----|--------------------------------------|-------------------------------|
| 981 | Reweighting for Fairness             | Adjust sample weights         |
| 982 | Demographic Parity Constraint        | Equalize positive rates       |
| 983 | Equalized Odds                       | Align error rates             |
| 984 | Adversarial Debiasing                | Learn fair representations    |
| 985 | Causal DAG Discovery                 | Graphical causal inference    |
| 986 | Propensity Score Matching            | Estimate treatment effect     |
| 987 | Instrumental Variable Estimation     | Handle confounders            |
| 988 | Robust Optimization                  | Worst-case aware optimization |
| 989 | Distributionally Robust Optimization | Minimax over uncertainty sets |
| 990 | Counterfactual Fairness              | Simulate do-interventions     |

## 100. AI Planning, Search, and Learning Systems

| #    | Algorithm                       | Note                          |
|------|---------------------------------|-------------------------------|
| 991  | Breadth-First Search (BFS)      | Uninformed search             |
| 992  | Depth-First Search (DFS)        | Backtracking search           |
| 993  | A* Search                       | Heuristic guided              |
| 994  | Iterative Deepening A* (IDA*)   | Memory-bounded heuristic      |
| 995  | Uniform Cost Search             | Expand by path cost           |
| 996  | Monte Carlo Tree Search (MCTS)  | Exploration vs exploitation   |
| 997  | Minimax                         | Game tree evaluation          |
| 998  | Alpha-Beta Pruning              | Prune unneeded branches       |
| 999  | STRIPS Planning                 | Action-based state transition |
| 1000 | Hierarchical Task Network (HTN) | Structured AI planning        |

# Chapter 1. Foundations of Algorithms

## Section 1. What is an algorithms?

### 1 Euclid's GCD

Euclid's algorithm is one of the oldest and most elegant procedures in mathematics. It computes the greatest common divisor (GCD) of two integers by repeatedly applying a simple rule: replace the larger number with its remainder when divided by the smaller. When the remainder becomes zero, the smaller number at that step is the GCD.

#### What Problem Are We Solving?

We want the greatest common divisor of two integers  $a$  and  $b$ : the largest number that divides both without a remainder.

A naive way would be to check all numbers from  $\min(a, b)$  down to 1. That's  $O(\min(a, b))$  steps, which is too slow for large inputs. Euclid's insight gives a much faster recursive method using division:

$$\gcd(a, b) = \begin{cases} a, & \text{if } b = 0, \\ \gcd(b, a \bmod b), & \text{otherwise.} \end{cases}$$

#### How It Works (Plain Language)

Imagine two sticks of lengths  $a$  and  $b$ . You can keep cutting the longer stick by the shorter one until one divides evenly. The length of the last nonzero remainder is the GCD.

Steps:

1. Take  $a, b$  with  $a \geq b$ .
2. Replace  $a$  by  $b$ , and  $b$  by  $a \bmod b$ .
3. Repeat until  $b = 0$ .
4. Return  $a$ .

This process always terminates, since  $b$  strictly decreases each step.

## Example Step by Step

Find  $\gcd(48, 18)$ :

| Step | a  | b  | a mod b |
|------|----|----|---------|
| 1    | 48 | 18 | 12      |
| 2    | 18 | 12 | 6       |
| 3    | 12 | 6  | 0       |

When  $b = 0$ ,  $a = 6$ . So  $\gcd(48, 18) = 6$ .

## Tiny Code (Python)

```
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

print(gcd(48, 18)) # Output: 6
```

## Why It Matters

- Foundational example of algorithmic thinking
- Core building block in modular arithmetic, number theory, and cryptography
- Efficient: runs in  $O(\log \min(a, b))$  steps
- Easy to implement iteratively or recursively

## A Gentle Proof (Why It Works)

If  $a = bq + r$ , any common divisor of  $a$  and  $b$  also divides  $r$ , since  $r = a - bq$ . Thus, the set of common divisors of  $(a, b)$  and  $(b, r)$  is the same, and their greatest element (the GCD) is unchanged.

Repeatedly applying this property leads to  $b = 0$ , where  $\gcd(a, 0) = a$ .

## Try It Yourself

1. Compute  $\gcd(270, 192)$  step by step.
2. Implement the recursive version:

$$\gcd(a, b) = \gcd(b, a \bmod b)$$

3. Extend to find  $\gcd(a, b, c)$  using  $\gcd(\gcd(a, b), c)$ .

## Test Cases

| Input $(a, b)$ | Expected Output |
|----------------|-----------------|
| (48, 18)       | 6               |
| (270, 192)     | 6               |
| (7, 3)         | 1               |
| (10, 0)        | 10              |

## Complexity

| Operation | Time                 | Space  |
|-----------|----------------------|--------|
| GCD       | $O(\log \min(a, b))$ | $O(1)$ |

Euclid's GCD algorithm is where algorithmic elegance begins, a timeless loop of division that turns mathematics into motion.

## 2 Sieve of Eratosthenes

The Sieve of Eratosthenes is a classic ancient algorithm for finding all prime numbers up to a given limit. It works by iteratively marking the multiples of each prime, starting from 2. The numbers that remain unmarked at the end are primes.

### What Problem Are We Solving?

We want to find all prime numbers less than or equal to  $n$ . A naive method checks each number  $k$  by testing divisibility from 2 to  $\sqrt{k}$ , which is too slow for large  $n$ . The sieve improves this by using elimination instead of repeated checking.

We aim for an algorithm with time complexity close to  $O(n \log \log n)$ .

## How It Works (Plain Language)

1. Create a list `is_prime[0..n]` and mark all as true.
2. Mark 0 and 1 as non-prime.
3. Starting from  $p = 2$ , if  $p$  is still marked prime:
  - Mark all multiples of  $p$  (from  $p^2$  to  $n$ ) as non-prime.
4. Increment  $p$  and repeat until  $p^2 > n$ .
5. All indices still marked true are primes.

This process “filters out” composite numbers step by step, just like passing sand through finer and finer sieves.

## Example Step by Step

Find all primes up to 30:

Start: [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30]

- $p = 2$ : cross out multiples of 2
- $p = 3$ : cross out multiples of 3
- $p = 5$ : cross out multiples of 5

Remaining numbers: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29

## Tiny Code (Python)

```
def sieve(n):  
    is_prime = [True] * (n + 1)  
    is_prime[0] = is_prime[1] = False  
  
    p = 2  
    while p * p <= n:  
        if is_prime[p]:  
            for i in range(p * p, n + 1, p):  
                is_prime[i] = False  
        p += 1  
  
    return [i for i in range(2, n + 1) if is_prime[i]]
```

```
print(sieve(30)) # [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

## Why It Matters

- One of the earliest and most efficient ways to generate primes
- Forms the basis for number-theoretic algorithms and cryptographic systems
- Conceptually simple yet mathematically deep
- Demonstrates elimination instead of brute force

## A Gentle Proof (Why It Works)

Every composite number  $n$  has a smallest prime divisor  $p \leq \sqrt{n}$ . Thus, when we mark multiples of primes up to  $\sqrt{n}$ , every composite number is crossed out by its smallest prime factor. Numbers that remain unmarked are prime by definition.

## Try It Yourself

1. Run the sieve for  $n = 50$  and list primes.
2. Modify to count primes instead of listing them.
3. Compare runtime with naive primality tests for large  $n$ .
4. Extend to a segmented sieve for  $n > 10^7$ .

## Test Cases

| Input $n$ | Expected Primes                      |
|-----------|--------------------------------------|
| 10        | [2, 3, 5, 7]                         |
| 20        | [2, 3, 5, 7, 11, 13, 17, 19]         |
| 30        | [2, 3, 5, 7, 11, 13, 17, 19, 23, 29] |

## Complexity

| Operation | Time               | Space  |
|-----------|--------------------|--------|
| Sieve     | $O(n \log \log n)$ | $O(n)$ |

The Sieve of Eratosthenes turns the search for primes into a graceful pattern of elimination, simple loops revealing the hidden order of numbers.

## 3 Linear Step Trace

A Linear Step Trace is a simple yet powerful visualization tool for understanding how an algorithm progresses line by line. It records each step of execution, showing how variables change over time, helping beginners see the *flow* of computation.

### What Problem Are We Solving?

When learning algorithms, it's easy to lose track of what's happening after each instruction. A Linear Step Trace helps us *see* execution in motion, one step, one update at a time.

Instead of abstract reasoning alone, we follow the exact state changes that occur during the run, making debugging and reasoning far easier.

### How It Works (Plain Language)

1. Write down your pseudocode or code.
2. Create a table with columns for step number, current line, and variable values.
3. Each time a line executes, record the line number and updated variables.
4. Continue until the program finishes.

This method is algorithm-agnostic, it works for loops, recursion, conditionals, and all flow patterns.

### Example Step by Step

Let's trace a simple loop:

```
sum = 0
for i in 1..4:
    sum = sum + i
```

| Step | Line | i | sum | Note           |
|------|------|---|-----|----------------|
| 1    | 1    | - | 0   | Initialize sum |
| 2    | 2    | 1 | 0   | Loop start     |
| 3    | 3    | 1 | 1   | sum = 0 + 1    |
| 4    | 2    | 2 | 1   | Next iteration |
| 5    | 3    | 2 | 3   | sum = 1 + 2    |
| 6    | 2    | 3 | 3   | Next iteration |
| 7    | 3    | 3 | 6   | sum = 3 + 3    |

| Step | Line | i | sum | Note           |
|------|------|---|-----|----------------|
| 8    | 2    | 4 | 6   | Next iteration |
| 9    | 3    | 4 | 10  | sum = 6 + 4    |
| 10   | -    | - | 10  | End            |

Final result:  $sum = 10$ .

### Tiny Code (Python)

```
sum = 0
trace = []

for i in range(1, 5):
    trace.append((i, sum))
    sum += i

trace.append(("final", sum))
print(trace)
# [(1, 0), (2, 1), (3, 3), (4, 6), ('final', 10)]
```

### Why It Matters

- Builds *step-by-step literacy* in algorithm reading
- Great for teaching loops, conditions, and recursion
- Reveals hidden assumptions and logic errors
- Ideal for debugging and analysis

### A Gentle Proof (Why It Works)

Every algorithm can be expressed as a sequence of state transitions. If each transition is recorded, we obtain a complete trace of computation. Thus, correctness can be verified by comparing expected vs. actual state sequences. This is equivalent to an inductive proof: each step matches the specification.

## Try It Yourself

1. Trace a recursive factorial function step by step.
2. Add a “call stack” column to visualize recursion depth.
3. Trace an array-sorting loop and mark swaps.
4. Compare traces before and after optimization.

## Test Cases

| Program      | Expected Final State |
|--------------|----------------------|
| sum of 1..4  | sum = 10             |
| sum of 1..10 | sum = 55             |
| factorial(5) | result = 120         |

## Complexity

| Operation       | Time   | Space  |
|-----------------|--------|--------|
| Trace Recording | $O(n)$ | $O(n)$ |

A Linear Step Trace transforms invisible logic into a visible path, a story of each line’s journey, one state at a time.

## 4 Algorithm Flow Diagram Builder

An Algorithm Flow Diagram Builder turns abstract pseudocode into a visual map, a diagram of control flow that shows where decisions branch, where loops repeat, and where computations end. It’s the bridge between code and comprehension.

### What Problem Are We Solving?

When an algorithm becomes complex, it’s easy to lose track of its structure. We may know what each line does, but not *how control moves* through the program.

A flow diagram lays out that control structure explicitly, revealing loops, branches, merges, and exits at a glance.

## How It Works (Plain Language)

1. Identify actions and decisions
  - Actions: assignments, computations
  - Decisions: if, while, for, switch
2. Represent them with symbols
  - Rectangle → action
  - Diamond → decision
  - Arrow → flow of control
3. Connect nodes based on what happens next
4. Loop back arrows for iterations, and mark exit points

This yields a graph of control, a shape you can follow from start to finish.

## Example Step by Step

Let's draw the flow for finding the sum of numbers 1 to  $n$ :

Pseudocode:

```
sum = 0
i = 1
while i < n:
    sum = sum + i
    i = i + 1
print(sum)
```

Flow Outline:

1. Start
2. Initialize `sum = 0, i = 1`
3. Decision: `i < n?`
  - Yes → Update `sum`, Increment `i` → Loop back
  - No → Print `sum` → End

Textual Diagram:

```

[Start]
|
[sum=0, i=1]
|
(i == n?) ----No----> [Print sum] -> [End]
|
Yes
|
[sum = sum + i]
|
[i = i + 1]
|
(Back to i == n?)

```

### Tiny Code (Python)

```

def sum_to_n(n):
    sum = 0
    i = 1
    while i <= n:
        sum += i
        i += 1
    return sum

```

Use this code to generate flow diagrams automatically with libraries like `graphviz` or `pyflowchart`.

### Why It Matters

- Reveals structure at a glance
- Makes debugging easier by visualizing possible paths
- Helps design before coding
- Universal representation (language-agnostic)

### A Gentle Proof (Why It Works)

Each algorithm's execution path can be modeled as a directed graph:

- Vertices = program states or actions
- Edges = transitions (next step)

A flow diagram is simply this control graph rendered visually. It preserves correctness since each edge corresponds to a valid jump in control flow.

### Try It Yourself

1. Draw a flowchart for binary search.
2. Mark all possible comparison outcomes.
3. Add loopbacks for mid-point updates.
4. Compare with recursive version, note structural difference.

### Test Cases

| Algorithm     | Key Decision Node | Expected Paths         |
|---------------|-------------------|------------------------|
| Sum loop      | $i \leq n$        | 2 (continue, exit)     |
| Binary search | $key == mid?$     | 3 (left, right, found) |

### Complexity

| Operation            | Time         | Space        |
|----------------------|--------------|--------------|
| Diagram Construction | $O(n)$ nodes | $O(n)$ edges |

An Algorithm Flow Diagram is a lens, it turns invisible execution paths into a map you can walk, from “Start” to “End.”

## 5 Long Division

Long Division is a step-by-step algorithm for dividing one integer by another. It’s one of the earliest examples of a systematic computational procedure, showing how large problems can be solved through a sequence of local, repeatable steps.

### What Problem Are We Solving?

We want to compute the quotient and remainder when dividing two integers  $a$  (dividend) and  $b$  (divisor).

Naively, repeated subtraction would take  $O(a/b)$  steps, far too many for large numbers. Long Division improves this by grouping subtractions by powers of 10, performing digit-wise computation efficiently.

## How It Works (Plain Language)

1. Align digits of  $a$  (the dividend).
2. Compare current portion of  $a$  to  $b$ .
3. Find the largest multiple of  $b$  that fits in the current portion.
4. Subtract, write the quotient digit, and bring down the next digit.
5. Repeat until all digits have been processed.
6. The digits written form the quotient; what's left is the remainder.

This method extends naturally to decimals, just continue bringing down zeros.

## Example Step by Step

Compute  $153 \div 7$ :

| Step | Portion              | Quotient<br>Digit | Remain-<br>der | Action                                     |
|------|----------------------|-------------------|----------------|--------------------------------------------|
| 1    | 15                   | 2                 | 1              | $7 \times 2 = 14$ , subtract $15 - 14 = 1$ |
| 2    | Bring down 3 →<br>13 | 1                 | 6              | $7 \times 1 = 7$ , subtract $13 - 7 = 6$   |
| 3    | No more digits       | ,                 | 6              | Done                                       |

Result: Quotient = 21, Remainder = 6 Check:  $7 \times 21 + 6 = 153$

## Tiny Code (Python)

```
def long_division(a, b):
    quotient = 0
    remainder = 0
    for digit in str(a):
        remainder = remainder * 10 + int(digit)
        q = remainder // b
        remainder = remainder % b
        quotient = quotient * 10 + q
    return quotient, remainder

print(long_division(153, 7))  # (21, 6)
```

## Why It Matters

- Introduces loop invariants and digit-by-digit reasoning
- Foundation for division in arbitrary-precision arithmetic
- Core to implementing division in CPUs and big integer libraries
- Demonstrates decomposing a large task into simple, local operations

## A Gentle Proof (Why It Works)

At each step:

- The current remainder  $r_i$  satisfies  $0 \leq r_i < b$ .
- The algorithm maintains the invariant:

$$a = b \times Q_i + r_i$$

where  $Q_i$  is the partial quotient so far.

- Each step reduces the unprocessed part of  $a$ , ensuring termination with correct  $Q$  and  $r$ .

## Try It Yourself

1. Perform  $2345 \div 13$  by hand.
2. Verify with Python's `divmod(2345, 13)`.
3. Extend your code to produce decimal expansions.
4. Compare digit-wise trace with manual process.

## Test Cases

| Dividend $a$ | Divisor $b$ | Expected Output $(Q, R)$ |
|--------------|-------------|--------------------------|
| 153          | 7           | (21, 6)                  |
| 100          | 8           | (12, 4)                  |
| 99           | 9           | (11, 0)                  |
| 23           | 5           | (4, 3)                   |

## Complexity

| Operation     | Time   | Space  |
|---------------|--------|--------|
| Long Division | $O(d)$ | $O(1)$ |

where  $d$  is the number of digits in  $a$ .

Long Division is more than arithmetic, it's the first encounter with algorithmic thinking: state, iteration, and correctness unfolding one digit at a time.

## 6 Modular Addition

Modular addition is arithmetic on a clock, we add numbers, then wrap around when reaching a fixed limit. It's the simplest example of modular arithmetic, a system that underlies cryptography, hashing, and cyclic data structures.

### What Problem Are We Solving?

We want to add two integers  $a$  and  $b$ , but keep the result within a fixed modulus  $m$ . That means we compute the remainder after dividing the sum by  $m$ .

Formally, we want:

$$(a + b) \bmod m$$

This ensures results always lie in the range  $[0, m - 1]$ , regardless of how large  $a$  or  $b$  become.

### How It Works (Plain Language)

1. Compute the sum  $s = a + b$ .
2. Divide  $s$  by  $m$  to find the remainder.
3. The remainder is the modular sum.

If  $s \geq m$ , we “wrap around” by subtracting  $m$  until it fits in the modular range.

This idea is like hours on a clock:  $10 + 5$  hours on a 12-hour clock  $\rightarrow 3$ .

## Example Step by Step

Let  $a = 10$ ,  $b = 7$ ,  $m = 12$ .

1. Compute  $s = 10 + 7 = 17$ .
2.  $17 \bmod 12 = 5$ .
3. So  $(10 + 7) \bmod 12 = 5$ .

Check:  $17 - 12 = 5$ , fits in  $[0, 11]$ .

## Tiny Code (Python)

```
def mod_add(a, b, m):
    return (a + b) % m

print(mod_add(10, 7, 12)) # 5
```

## Why It Matters

- Foundation of modular arithmetic
- Used in hashing, cyclic buffers, and number theory
- Crucial for secure encryption (RSA, ECC)
- Demonstrates wrap-around logic in bounded systems

## A Gentle Proof (Why It Works)

By definition of modulus:

$$x \bmod m = r \quad \text{such that } x = q \times m + r, \quad 0 \leq r < m$$

Thus, for  $a + b = q \times m + r$ , we have  $(a + b) \bmod m = r$ . All equivalent sums differ by a multiple of  $m$ , so modular addition preserves congruence:

$$(a + b) \bmod m \equiv (a \bmod m + b \bmod m) \bmod m$$

## Try It Yourself

1. Compute  $(15 + 8) \bmod 10$ .
2. Verify  $(a + b) \bmod m = ((a \bmod m) + (b \bmod m)) \bmod m$ .
3. Test with negative values:  $(-3 + 5) \bmod 7$ .
4. Apply to time arithmetic: what is  $11 + 5$  on a 12-hour clock?

## Test Cases

| $a$ | $b$ | $m$ | Result |
|-----|-----|-----|--------|
| 10  | 7   | 12  | 5      |
| 5   | 5   | 10  | 0      |
| 8   | 15  | 10  | 3      |
| 11  | 5   | 12  | 4      |

## Complexity

| Operation        | Time   | Space  |
|------------------|--------|--------|
| Modular Addition | $O(1)$ | $O(1)$ |

Modular addition teaches the rhythm of modular arithmetic, every sum wraps back into harmony, always staying within its finite world.

## 7 Base Conversion

Base conversion is the algorithmic process of expressing a number in a different numeral system. It's how we translate between decimal, binary, octal, hexadecimal, or any base, the language of computers and mathematics alike.

### What Problem Are We Solving?

We want to represent an integer  $n$  in base  $b$ . In base 10, digits go from 0 to 9. In base 2, only 0 and 1. In base 16, digits are 0 ... 9 and  $A \dots F$ .

The goal is to find a sequence of digits  $d_k d_{k-1} \dots d_0$  such that:

$$n = \sum_{i=0}^k d_i \cdot b^i$$

where  $0 \leq d_i < b$ .

### How It Works (Plain Language)

1. Start with the integer  $n$ .
2. Repeatedly divide  $n$  by  $b$ .
3. Record the remainder each time (these are the digits).
4. Stop when  $n = 0$ .
5. The base- $b$  representation is the remainders read in reverse order.

This works because division extracts digits starting from the least significant position.

### Example Step by Step

Convert 45 to binary ( $b = 2$ ):

| Step | $n$ | $n \div 2$ | Remainder |
|------|-----|------------|-----------|
| 1    | 45  | 22         | 1         |
| 2    | 22  | 11         | 0         |
| 3    | 11  | 5          | 1         |
| 4    | 5   | 2          | 1         |
| 5    | 2   | 1          | 0         |
| 6    | 1   | 0          | 1         |

Read remainders upward: 101101

So  $45_{10} = 101101_2$ .

Check:  $1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 32 + 0 + 8 + 4 + 0 + 1 = 45$

### Tiny Code (Python)

```

def to_base(n, b):
    digits = []
    while n > 0:
        digits.append(n % b)
        n //= b
    return digits[::-1] or [0]

print(to_base(45, 2))  # [1, 0, 1, 1, 0, 1]

```

## Why It Matters

- Converts numbers between human and machine representations
- Core in encoding, compression, and cryptography
- Builds intuition for positional number systems
- Used in parsing, serialization, and digital circuits

## A Gentle Proof (Why It Works)

Each division step produces one digit  $r_i = n_i \bmod b$ . We have:

$$n_i = b \cdot n_{i+1} + r_i$$

Unfolding the recurrence gives:

$$n = \sum_{i=0}^k r_i b^i$$

So collecting remainders in reverse order reconstructs  $n$  exactly.

## Try It Yourself

1. Convert  $100_{10}$  to base 8.
2. Convert  $255_{10}$  to base 16.
3. Verify by recombining digits via  $\sum d_i b^i$ .
4. Write a reverse converter: base- $b$  to decimal.

## Test Cases

| Decimal $n$ | Base $b$ | Representation |
|-------------|----------|----------------|
| 45          | 2        | 101101         |
| 100         | 8        | 144            |
| 255         | 16       | FF             |
| 31          | 5        | 111            |

## Complexity

| Operation       | Time          | Space         |
|-----------------|---------------|---------------|
| Base Conversion | $O(\log_b n)$ | $O(\log_b n)$ |

Base conversion is arithmetic storytelling, peeling away remainders until only digits remain, revealing the same number through a different lens.

## 8 Factorial Computation

Factorial computation is the algorithmic act of multiplying a sequence of consecutive integers, a simple rule that grows explosively. It lies at the foundation of combinatorics, probability, and mathematical analysis.

### What Problem Are We Solving?

We want to compute the factorial of a non-negative integer  $n$ , written  $n!$ , defined as:

$$n! = n \times (n - 1) \times (n - 2) \times \cdots \times 1$$

with the base case:

$$0! = 1$$

Factorial counts the number of ways to arrange  $n$  distinct objects, the building block of permutations and combinations.

## How It Works (Plain Language)

There are two main ways:

Iterative:

- Start with `result = 1`
- Multiply by each  $i$  from 1 to  $n$
- Return result

Recursive:

- $n! = n \times (n - 1)!$
- Stop when  $n = 0$

Both methods produce the same result; recursion mirrors the mathematical definition, iteration avoids call overhead.

## Example Step by Step

Compute  $5!$ :

| Step | $n$ | Product |
|------|-----|---------|
| 1    | 1   | 1       |
| 2    | 2   | 2       |
| 3    | 3   | 6       |
| 4    | 4   | 24      |
| 5    | 5   | 120     |

So  $5! = 120$

## Tiny Code (Python)

Iterative Version

```
def factorial_iter(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result

print(factorial_iter(5))  # 120
```

## Recursive Version

```
def factorial_rec(n):
    if n == 0:
        return 1
    return n * factorial_rec(n - 1)

print(factorial_rec(5)) # 120
```

## Why It Matters

- Core operation in combinatorics, calculus, and probability
- Demonstrates recursion, iteration, and induction
- Grows rapidly, useful for testing overflow and asymptotics
- Appears in binomial coefficients, Taylor series, and permutations

## A Gentle Proof (Why It Works)

By definition,  $n! = n \times (n - 1)!$ . Assume  $(n - 1)!$  is correctly computed. Then multiplying by  $n$  yields  $n!$ .

By induction:

- Base case:  $0! = 1$
- Step: if  $(n - 1)!$  is correct, so is  $n!$

Thus, the recursive and iterative definitions are equivalent and correct.

## Try It Yourself

1. Compute  $6!$  both iteratively and recursively.
2. Print intermediate products to trace the growth.
3. Compare runtime for  $n = 1000$  using both methods.
4. Explore factorial in floating point (`math.gamma`) for non-integers.

| Input $n$ | Expected Output $n!$ |
|-----------|----------------------|
|-----------|----------------------|

## Test Cases

| Input $n$ | Expected Output $n!$ |
|-----------|----------------------|
| 0         | 1                    |
| 1         | 1                    |
| 3         | 6                    |
| 5         | 120                  |
| 6         | 720                  |

## Complexity

| Operation | Time   | Space          |
|-----------|--------|----------------|
| Iterative | $O(n)$ | $O(1)$         |
| Recursive | $O(n)$ | $O(n)$ (stack) |

Factorial computation is where simplicity meets infinity, a single rule that scales from 1 to astronomical numbers with graceful inevitability.

## 9 Iterative Process Tracer

An Iterative Process Tracer is a diagnostic algorithm that follows each iteration of a loop, recording variable states, conditions, and updates. It helps visualize the evolution of a program's internal state, turning looping logic into a clear timeline.

### What Problem Are We Solving?

When writing iterative algorithms, it's easy to lose sight of what happens at each step. Are variables updating correctly? Are loop conditions behaving as expected? A tracer captures this process, step by step, so we can verify correctness, find bugs, and teach iteration with clarity.

## How It Works (Plain Language)

1. Identify the loop (for or while).
2. Before or after each iteration, record:
  - The iteration number
  - Key variable values
  - Condition evaluations
3. Store these snapshots in a trace table.
4. After execution, review how values evolve over time.

Think of it as an “execution diary”, every iteration gets a journal entry.

## Example Step by Step

Let's trace a simple accumulation:

```
sum = 0
for i in 1..5:
    sum = sum + i
```

| Step | i | sum | Description       |
|------|---|-----|-------------------|
| 1    | 1 | 1   | Add first number  |
| 2    | 2 | 3   | Add second number |
| 3    | 3 | 6   | Add third number  |
| 4    | 4 | 10  | Add fourth number |
| 5    | 5 | 15  | Add fifth number  |

Final result:  $sum = 15$

## Tiny Code (Python)

```

def trace_sum(n):
    sum = 0
    trace = []
    for i in range(1, n + 1):
        sum += i
        trace.append((i, sum))
    return trace

print(trace_sum(5))
# [(1, 1), (2, 3), (3, 6), (4, 10), (5, 15)]

```

## Why It Matters

- Turns hidden state changes into visible data
- Ideal for debugging loops and verifying invariants
- Supports algorithm teaching and step-by-step reasoning
- Useful in profiling, logging, and unit testing

## A Gentle Proof (Why It Works)

An iterative algorithm is a sequence of deterministic transitions:

$$S_{i+1} = f(S_i)$$

Recording  $S_i$  at each iteration yields the complete trajectory of execution. The trace table captures all intermediate states, ensuring reproducibility and clarity, a form of operational proof.

## Try It Yourself

1. Trace variable updates in a multiplication loop.
2. Add condition checks (e.g. early exits).
3. Record both pre- and post-update states.
4. Compare traces of iterative vs recursive versions.

## Test Cases

| Input $n$ | Expected Trace                               |
|-----------|----------------------------------------------|
| 3         | $[(1, 1), (2, 3), (3, 6)]$                   |
| 4         | $[(1, 1), (2, 3), (3, 6), (4, 10)]$          |
| 5         | $[(1, 1), (2, 3), (3, 6), (4, 10), (5, 15)]$ |

## Complexity

| Operation | Time   | Space  |
|-----------|--------|--------|
| Tracing   | $O(n)$ | $O(n)$ |

An Iterative Process Tracer makes thinking visible, a loop's internal rhythm laid out, step by step, until the final note resolves.

## 10 Tower of Hanoi

The Tower of Hanoi is a legendary recursive puzzle that beautifully illustrates how complex problems can be solved through simple repeated structure. It's a timeless example of *divide and conquer* thinking in its purest form.

### What Problem Are We Solving?

We want to move  $n$  disks from a source peg to a target peg, using one auxiliary peg. Rules:

1. Move only one disk at a time.
2. Never place a larger disk on top of a smaller one.

The challenge is to find the minimal sequence of moves that achieves this.

### How It Works (Plain Language)

The key insight: To move  $n$  disks, first move  $n - 1$  disks aside, move the largest one, then bring the smaller ones back.

Steps:

1. Move  $n - 1$  disks from source  $\rightarrow$  auxiliary
2. Move the largest disk from source  $\rightarrow$  target
3. Move  $n - 1$  disks from auxiliary  $\rightarrow$  target

This recursive structure repeats until the smallest disk moves directly.

## Example Step by Step

For  $n = 3$ , pegs: A (source), B (auxiliary), C (target)

| Step | Move  |
|------|-------|
| 1    | A → C |
| 2    | A → B |
| 3    | C → B |
| 4    | A → C |
| 5    | B → A |
| 6    | B → C |
| 7    | A → C |

Total moves:  $2^3 - 1 = 7$

## Tiny Code (Python)

```
def hanoi(n, source, target, aux):
    if n == 1:
        print(f"{source} → {target}")
        return
    hanoi(n - 1, source, aux, target)
    print(f"{source} → {target}")
    hanoi(n - 1, aux, target, source)

hanoi(3, 'A', 'C', 'B')
```

## Why It Matters

- Classic recursive pattern: break → solve → combine
- Demonstrates exponential growth ( $2^n - 1$  moves)
- Trains recursive reasoning and stack visualization
- Appears in algorithm analysis, recursion trees, and combinatorics

## A Gentle Proof (Why It Works)

Let  $T(n)$  be the number of moves for  $n$  disks. We must move  $n - 1$  disks twice and one largest disk once:

$$T(n) = 2T(n - 1) + 1, \quad T(1) = 1$$

Solving the recurrence:

$$T(n) = 2^n - 1$$

Each recursive step preserves rules and reduces the problem size, ensuring correctness by structural induction.

## Try It Yourself

1. Trace  $n = 2$  and  $n = 3$  by hand.
2. Count recursive calls.
3. Modify code to record moves in a list.
4. Extend to display peg states after each move.

## Test Cases

| $n$ | Expected Moves |
|-----|----------------|
| 1   | 1              |
| 2   | 3              |
| 3   | 7              |
| 4   | 15             |

## Complexity

| Operation | Time     | Space                    |
|-----------|----------|--------------------------|
| Moves     | $O(2^n)$ | $O(n)$ (recursion stack) |

The Tower of Hanoi turns recursion into art, every move guided by symmetry, every step revealing how simplicity builds complexity one disk at a time.

## Section 2. Measuring time and space

### 11 Counting Operations

Counting operations is the first step toward understanding time complexity. It's the art of translating code into math by measuring how many *basic steps* an algorithm performs, helping us predict performance before running it.

#### What Problem Are We Solving?

We want to estimate how long an algorithm takes, not by clock time, but by how many fundamental operations it executes. Instead of relying on hardware speed, we count abstract steps, comparisons, assignments, additions, each treated as one unit of work.

This turns algorithms into analyzable formulas.

#### How It Works (Plain Language)

1. Identify the unit step (like one comparison or addition).
2. Break the algorithm into lines or loops.
3. Count repetitions for each operation.
4. Sum all counts to get a total step function  $T(n)$ .
5. Simplify to dominant terms for asymptotic analysis.

We're not measuring *seconds*, we're measuring *structure*.

#### Example Step by Step

Count operations for:

```
sum = 0
for i in range(1, n + 1):
    sum += i
```

Breakdown:

| Line | Operation             | Count   |
|------|-----------------------|---------|
| 1    | Initialization        | 1       |
| 2    | Loop comparison       | $n + 1$ |
| 3    | Addition + assignment | $n$     |

Total:

$$T(n) = 1 + (n + 1) + n = 2n + 2$$

Asymptotically:

$$T(n) = O(n)$$

### Tiny Code (Python)

```
def count_sum_ops(n):
    ops = 0
    ops += 1 # init sum
    for i in range(1, n + 1):
        ops += 1 # loop check
        ops += 1 # sum += i
    ops += 1 # final loop check
    return ops
```

Test: `count_sum_ops(5) → 13`

### Why It Matters

- Builds intuition for algorithm growth
- Reveals hidden costs (nested loops, recursion)
- Foundation for Big-O and runtime proofs
- Language-agnostic: works for any pseudocode

### A Gentle Proof (Why It Works)

Every program can be modeled as a finite sequence of operations parameterized by input size  $n$ . If  $f(n)$  counts these operations exactly, then for large  $n$ , growth rate  $\Theta(f(n))$  matches actual performance up to constant factors. Counting operations therefore predicts asymptotic runtime behavior.

## Try It Yourself

1. Count operations in a nested loop:

```
for i in range(n):
    for j in range(n):
        x += 1
```

2. Derive  $T(n) = n^2 + 2n + 1$ .
3. Simplify to  $O(n^2)$ .
4. Compare iterative vs recursive counting.

## Test Cases

| Algorithm     | Step Function  | Big-O    |
|---------------|----------------|----------|
| Linear Loop   | $2n + 2$       | $O(n)$   |
| Nested Loop   | $n^2 + 2n + 1$ | $O(n^2)$ |
| Constant Work | $c$            | $O(1)$   |

## Complexity

| Operation      | Time              | Space  |
|----------------|-------------------|--------|
| Counting Steps | $O(1)$ (analysis) | $O(1)$ |

Counting operations transforms code into mathematics, a microscope for understanding how loops, branches, and recursion scale with input size.

## 12 Loop Analysis

Loop analysis is the key to unlocking how algorithms grow, it tells us how many times a loop runs and, therefore, how many operations are performed. Every time you see a loop, you're looking at a formula in disguise.

## What Problem Are We Solving?

We want to determine how many iterations a loop executes as a function of input size  $n$ . This helps us estimate total runtime before measuring it empirically.

Whether a loop is linear, nested, logarithmic, or mixed, understanding its iteration count reveals the algorithm's true complexity.

## How It Works (Plain Language)

1. Identify the loop variable (like `i` in `for i in range(...)`).
2. Find its update rule, additive (`i += 1`) or multiplicative (`i *= 2`).
3. Solve for how many times the condition holds true.
4. Multiply by inner loop work if nested.
5. Sum all contributions from independent loops.

This transforms loops into algebraic expressions you can reason about.

## Example Step by Step

Example 1: Linear Loop

```
for i in range(1, n + 1):
    work()
```

$i$  runs from 1 to  $n$ , incrementing by 1. Iterations:  $n$  Work:  $O(n)$

Example 2: Logarithmic Loop

```
i = 1
while i <= n:
    work()
    i *= 2
```

$i$  doubles each step: 1, 2, 4, 8, ...,  $n$  Iterations:  $\log_2 n + 1$  Work:  $O(\log n)$

Example 3: Nested Loop

```
for i in range(n):
    for j in range(n):
        work()
```

Outer loop:  $n$  Inner loop:  $n$  Total work:  $n \times n = n^2$

## Tiny Code (Python)

```
def linear_loop(n):
    count = 0
    for i in range(n):
        count += 1
    return count # n

def log_loop(n):
    count = 0
    i = 1
    while i <= n:
        count += 1
        i *= 2
    return count # log2(n)
```

## Why It Matters

- Reveals complexity hidden inside loops
- Core tool for deriving  $O(n)$ ,  $O(\log n)$ , and  $O(n^2)$
- Makes asymptotic behavior predictable and measurable
- Works for for-loops, while-loops, and nested structures

## A Gentle Proof (Why It Works)

Each loop iteration corresponds to a true condition in its guard. If the loop variable  $i$  evolves monotonically (by addition or multiplication), the total number of iterations is the smallest  $k$  satisfying the exit condition.

For additive updates:

$$i_0 + k \cdot \Delta \geq n \implies k = \frac{n - i_0}{\Delta}$$

For multiplicative updates:

$$i_0 \cdot r^k \geq n \implies k = \log_r \frac{n}{i_0}$$

## Try It Yourself

1. Analyze loop:

```
i = n
while i > 0:
    i /= 2
```

$$\rightarrow O(\log n)$$

2. Analyze double loop:

```
for i in range(n):
    for j in range(i):
        work()
```

$$\rightarrow \frac{n(n-1)}{2} = O(n^2)$$

3. Combine additive + multiplicative loops.

## Test Cases

| Code Pattern                         | Iterations         | Complexity  |
|--------------------------------------|--------------------|-------------|
| for i in range(n)                    | $n$                | $O(n)$      |
| while i < n: i *= 2                  | $\log_2 n$         | $O(\log n)$ |
| for i in range(n): for j in range(n) | $n^2$              | $O(n^2)$    |
| for i in range(n): for j in range(i) | $\frac{n(n-1)}{2}$ | $O(n^2)$    |

## Complexity

| Operation     | Time              | Space  |
|---------------|-------------------|--------|
| Loop Analysis | $O(1)$ (per loop) | $O(1)$ |

Loop analysis turns repetition into arithmetic, every iteration becomes a term, every loop a story in the language of growth.

## 13 Recurrence Expansion

Recurrence expansion is how we *unfold* recursive definitions to see their true cost. Many recursive algorithms (like Merge Sort or Quick Sort) define runtime in terms of smaller copies of themselves. By expanding the recurrence, we reveal the total work step by step.

## What Problem Are We Solving?

Recursive algorithms often express their runtime as:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

Here:

- $a$  = number of recursive calls
- $b$  = factor by which input size is reduced
- $f(n)$  = work done outside recursion (splitting, merging, etc.)

We want to estimate  $T(n)$  by expanding this relation until the base case.

## How It Works (Plain Language)

Think of recurrence expansion as peeling an onion. Each recursive layer contributes some cost, and we add all layers until the base.

Steps:

1. Write the recurrence.
2. Expand one level: replace  $T(\cdot)$  with its formula.
3. Repeat until the argument becomes the base case.
4. Sum the work done at each level.
5. Simplify the sum to get asymptotic form.

## Example Step by Step

Take Merge Sort:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

Expand:

- Level 0:  $T(n) = 2T(n/2) + n$
- Level 1:  $T(n/2) = 2T(n/4) + n/2 \rightarrow$  Substitute  $T(n) = 4T(n/4) + 2n$
- Level 2:  $T(n) = 8T(n/8) + 3n$
- ...
- Level  $\log_2 n$ :  $T(1) = c$

Sum work across levels:

$$T(n) = n \log_2 n + n = O(n \log n)$$

### Tiny Code (Python)

```
def recurrence_expand(a, b, f, n, base=1):
    level = 0
    total = 0
    size = n
    while size >= base:
        cost = (a * level) * f(size)
        total += cost
        size //= b
        level += 1
    return total
```

Use  $f = \lambda x: x$  for Merge Sort.

### Why It Matters

- Core tool for analyzing recursive algorithms
- Builds intuition before applying the Master Theorem
- Turns abstract recurrence into tangible pattern
- Helps visualize total work per recursion level

### A Gentle Proof (Why It Works)

At level  $i$ :

- There are  $a^i$  subproblems.
- Each subproblem has size  $\frac{n}{b^i}$ .
- Work per level:  $a^i \cdot f\left(\frac{n}{b^i}\right)$

Total cost:

$$T(n) = \sum_{i=0}^{\log_b n} a^i f\left(\frac{n}{b^i}\right)$$

Depending on how  $f(n)$  compares to  $n^{\log_b a}$ , either top, bottom, or middle levels dominate.

## Try It Yourself

1. Expand  $T(n) = 3T(n/2) + n^2$ .
2. Expand  $T(n) = T(n/2) + 1$ .
3. Visualize total work per level.
4. Check your result with Master Theorem.

## Test Cases

| Recurrence           | Expansion  | Result | Complexity    |
|----------------------|------------|--------|---------------|
| $T(n) = 2T(n/2) + n$ | $n \log n$ |        | $O(n \log n)$ |
| $T(n) = T(n/2) + 1$  | $\log n$   |        | $O(\log n)$   |
| $T(n) = 4T(n/2) + n$ | $n^2$      |        | $O(n^2)$      |

## Complexity

| Operation | Time               | Space                  |
|-----------|--------------------|------------------------|
| Expansion | $O(\log n)$ levels | $O(\log n)$ tree depth |

Recurrence expansion turns recursion into rhythm, each level adding its verse, the sum revealing the melody of the algorithm's growth.

## 14 Amortized Analysis

Amortized analysis looks beyond the worst case of individual operations to capture the *average cost per operation* over a long sequence. It tells us when “expensive” actions even out, revealing algorithms that are faster than they first appear.

### What Problem Are We Solving?

Some operations occasionally take a long time (like resizing an array), but most are cheap. A naive worst-case analysis exaggerates total cost. Amortized analysis finds the true average cost across a sequence.

We're not averaging across *inputs*, but across *operations in one run*.

## How It Works (Plain Language)

Suppose an operation is usually  $O(1)$ , but sometimes  $O(n)$ . If that expensive case happens rarely enough, the *average per operation* is still small.

Three main methods:

1. Aggregate method, total cost  $\div$  number of operations
2. Accounting method, charge extra for cheap ops, save credit for costly ones
3. Potential method, define potential energy (stored work) and track change

## Example Step by Step

Dynamic Array Resizing

When an array is full, double its size and copy elements.

| Operation    | Cost | Comment             |
|--------------|------|---------------------|
| Insert #1–#1 | 1    | insert directly     |
| Insert #2    | 2    | resize to 2, copy 1 |
| Insert #3    | 3    | resize to 4, copy 2 |
| Insert #5    | 5    | resize to 8, copy 4 |
| ...          | ...  | ...                 |

Total cost after  $n$  inserts  $2n$  Average cost =  $2n/n = O(1)$  So each insert is amortized  $O(1)$ , not  $O(n)$ .

## Tiny Code (Python)

```
def dynamic_array(n):
    arr = []
    capacity = 1
    cost = 0
    for i in range(n):
        if len(arr) == capacity:
            capacity *= 2
            cost += len(arr) # copying cost
        arr.append(i)
        cost += 1 # insert cost
    return cost, cost / n # total, amortized average
```

Try `dynamic_array(10)` → roughly total cost 20, average 2.

## Why It Matters

- Shows average efficiency over sequences
- Key to analyzing stacks, queues, hash tables, and dynamic arrays
- Explains why “occasionally expensive” operations are still efficient overall
- Separates perception (worst-case) from reality (aggregate behavior)

## A Gentle Proof (Why It Works)

Let  $C_i$  = cost of  $i$ th operation, and  $n$  = total operations.

Aggregate Method:

$$\text{Amortized cost} = \frac{\sum_{i=1}^n C_i}{n}$$

If  $\sum C_i = O(n)$ , each operation's average =  $O(1)$ .

Potential Method:

Define potential  $\Phi_i$  representing saved work. Amortized cost =  $C_i + \Phi_i - \Phi_{i-1}$  Summing over all operations telescopes potential away, leaving total cost bounded by initial + final potential.

## Try It Yourself

1. Analyze amortized cost for stack with occasional full pop.
2. Use accounting method to assign “credits” to inserts.
3. Show  $O(1)$  amortized insert in hash table with resizing.
4. Compare amortized vs worst-case time.

## Test Cases

| Operation Type                | Worst Case  | Amortized      |
|-------------------------------|-------------|----------------|
| Array Insert (Doubling)       | $O(n)$      | $O(1)$         |
| Stack Push                    | $O(1)$      | $O(1)$         |
| Queue Dequeue (2-stack)       | $O(n)$      | $O(1)$         |
| Union-Find (Path Compression) | $O(\log n)$ | $O(\alpha(n))$ |

## Complexity

| Analysis Type | Formula                       | Goal         |
|---------------|-------------------------------|--------------|
| Aggregate     | $\frac{\text{Total Cost}}{n}$ | Simplicity   |
| Accounting    | Assign credits                | Intuition    |
| Potential     | $\Delta\Phi$                  | Formal rigor |

Amortized analysis reveals the calm beneath chaos — a few storms don't define the weather, and one  $O(n)$  moment doesn't ruin  $O(1)$  harmony.

## 15 Space Counting

Space counting is the spatial twin of operation counting, instead of measuring time, we measure how much *memory* an algorithm consumes. Every variable, array, stack frame, or temporary buffer adds to the footprint. Understanding it helps us write programs that fit in memory and scale gracefully.

### What Problem Are We Solving?

We want to estimate the space complexity of an algorithm — how much memory it needs as input size  $n$  grows.

This includes:

- Static space (fixed variables)
- Dynamic space (arrays, recursion, data structures)
- Auxiliary space (extra working memory beyond input)

Our goal: express total memory as a function  $S(n)$ .

### How It Works (Plain Language)

1. Count primitive variables (constants, counters, pointers). → constant space  $O(1)$
2. Add data structure sizes (arrays, lists, matrices). → often proportional to  $n$ ,  $n^2$ , etc.
3. Add recursion stack depth, if applicable.
4. Ignore constants for asymptotic space, focus on growth.

In the end,

$$S(n) = S_{\text{static}} + S_{\text{dynamic}} + S_{\text{recursive}}$$

## Example Step by Step

Example 1: Linear Array

```
arr = [0] * n
```

- $n$  integers  $\rightarrow O(n)$  space

Example 2: 2D Matrix

```
matrix = [[0] * n for _ in range(n)]
```

- $n \times n$  elements  $\rightarrow O(n^2)$  space

Example 3: Recursive Factorial

```
def fact(n):
    if n == 0:
        return 1
    return n * fact(n - 1)
```

- Depth =  $n \rightarrow$  Stack =  $O(n)$
- No extra data structures  $\rightarrow S(n) = O(n)$

## Tiny Code (Python)

```
def space_counter(n):
    const = 1           # O(1)
    arr = [0] * n       # O(n)
    matrix = [[0]*n for _ in range(n)]  # O(n^2)
    return const + len(arr) + len(matrix)
```

This simple example illustrates additive contributions.

## Why It Matters

- Memory is a first-class constraint in large systems
- Critical for embedded, streaming, and real-time algorithms
- Reveals tradeoffs between time and space
- Guides design of in-place vs out-of-place solutions

## A Gentle Proof (Why It Works)

Each algorithm manipulates a finite set of data elements. If  $s_i$  is the space allocated for structure  $i$ , total space is:

$$S(n) = \sum_i s_i(n)$$

Asymptotic space is dominated by the largest term, so  $S(n) = \Theta(\max_i s_i(n))$ .

This ensures our analysis scales with data growth.

## Try It Yourself

1. Count space for Merge Sort (temporary arrays).
2. Compare with Quick Sort (in-place).
3. Add recursion cost explicitly.
4. Analyze time–space tradeoff for dynamic programming.

## Test Cases

| Algorithm     | Space       | Reason                  |
|---------------|-------------|-------------------------|
| Linear Search | $O(1)$      | Constant extra memory   |
| Merge Sort    | $O(n)$      | Extra array for merging |
| Quick Sort    | $O(\log n)$ | Stack depth             |
| DP Table (2D) | $O(n^2)$    | Full grid of states     |

## Complexity

| Component       | Example                   | Cost     |
|-----------------|---------------------------|----------|
| Variables       | $a, b, c$                 | $O(1)$   |
| Arrays          | <code>arr[n]</code>       | $O(n)$   |
| Matrices        | <code>matrix[n][n]</code> | $O(n^2)$ |
| Recursion Stack | Depth $n$                 | $O(n)$   |

Space counting turns memory into a measurable quantity, every variable a footprint, every structure a surface, every stack frame a layer in the architecture of an algorithm.

## 16 Memory Footprint Estimator

A Memory Footprint Estimator calculates how much memory an algorithm or data structure truly consumes, not just asymptotically, but in *real bytes*. It bridges the gap between theoretical space complexity and practical implementation.

### What Problem Are We Solving?

Knowing an algorithm is  $O(n)$  in space isn't enough when working close to memory limits. We need actual estimates: how many bytes per element, how much total allocation, and what overheads exist.

A footprint estimator converts theoretical counts into quantitative estimates for real-world scaling.

### How It Works (Plain Language)

1. Identify data types used: `int`, `float`, `pointer`, `struct`, etc.
2. Estimate size per element (language dependent, e.g. `int = 4 bytes`).
3. Multiply by count to find total memory usage.
4. Include overheads from:
  - Object headers or metadata
  - Padding or alignment
  - Pointers or references

Final footprint:

$$\text{Memory} = \sum_i (\text{count}_i \times \text{size}_i) + \text{overhead}$$

### Example Step by Step

Suppose we have a list of  $n = 1,000,000$  integers in Python.

| Component       | Size (Bytes) | Count     | Total      |
|-----------------|--------------|-----------|------------|
| List object     | 64           | 1         | 64         |
| Pointers        | 8            | 1,000,000 | 8,000,000  |
| Integer objects | 28           | 1,000,000 | 28,000,000 |

Total 36 MB (plus interpreter overhead).

If using a fixed `array('i')` (C-style ints):  $4 \text{ bytes} \times 10^6 = 4 \text{ MB}$ , far more memory-efficient.

### Tiny Code (Python)

```
import sys

n = 1_000_000
arr_list = list(range(n))
arr_array = bytearray(n * 4)

print(sys.getsizeof(arr_list))    # list object
print(sys.getsizeof(arr_array))  # raw byte array
```

Compare memory cost using `sys.getsizeof()`.

### Why It Matters

- Reveals true memory requirements
- Critical for large datasets, embedded systems, and databases
- Explains performance tradeoffs in languages with object overhead
- Supports system design and capacity planning

### A Gentle Proof (Why It Works)

Each variable or element consumes a fixed number of bytes depending on type. If  $n_i$  elements of type  $t_i$  are allocated, total memory is:

$$M(n) = \sum_i n_i \cdot s(t_i)$$

Since  $s(t_i)$  is constant, growth rate follows counts:  $M(n) = O(\max_i n_i)$ , matching asymptotic analysis while giving concrete magnitudes.

### Try It Yourself

1. Estimate memory for a matrix of  $1000 \times 1000$  floats (8 bytes each).
2. Compare Python list of lists vs NumPy array.
3. Add overheads for pointers and headers.
4. Repeat for custom `struct` or class with multiple fields.

## Test Cases

| Structure                  | Formula         | Approx Memory          |
|----------------------------|-----------------|------------------------|
| List of $n$ ints           | $n \times 28$ B | 28 MB (1M items)       |
| Array of $n$ ints          | $n \times 4$ B  | 4 MB                   |
| Matrix $n \times n$ floats | $8n^2$ B        | 8 MB for $n = 1000$    |
| Hash Table $n$ entries     | $O(n)$          | Depends on load factor |

## Complexity

| Metric   | Growth | Unit     |
|----------|--------|----------|
| Space    | $O(n)$ | Bytes    |
| Overhead | $O(1)$ | Metadata |

A Memory Footprint Estimator turns abstract “ $O(n)$  space” into tangible bytes, letting you see how close you are to the edge before your program runs out of room.

## 17 Time Complexity Table

A Time Complexity Table summarizes how different algorithms grow as input size increases, it's a map from formula to feeling, showing which complexities are fast, which are dangerous, and how they compare in scale.

### What Problem Are We Solving?

We want a quick reference that links mathematical growth rates to practical performance. Knowing that an algorithm is  $O(n \log n)$  is good; understanding what that means for  $n = 10^6$  is better.

The table helps estimate feasibility: Can this algorithm handle a million inputs? A billion?

### How It Works (Plain Language)

1. List common complexity classes: constant, logarithmic, linear, etc.
2. Write their formulas and interpretations.
3. Estimate operations for various  $n$ .
4. Highlight tipping points, where performance becomes infeasible.

This creates an *intuition grid* for algorithmic growth.

## Example Step by Step

Let  $n = 10^6$  (1 million). Estimate operations per complexity class (approximate scale):

| Complexity    | Formula                  | Operations (n=10 )         | Intuition      |
|---------------|--------------------------|----------------------------|----------------|
| $O(1)$        | constant                 | 1                          | instant        |
| $O(\log n)$   | $\log_2 10^6 \approx 20$ | 20                         | lightning fast |
| $O(n)$        | $10^6$                   | 1,000,000                  | manageable     |
| $O(n \log n)$ | $10^6 \cdot 20$          | 20M                        | still OK       |
| $O(n^2)$      | $(10^6)^2$               | $10^{12}$                  | too slow       |
| $O(2^n)$      | $2^{20} \approx 10^6$    | impossible beyond $n = 30$ |                |
| $O(n!)$       | factorial                | $10^6!$                    | absurdly huge  |

The table makes complexity feel *real*.

## Tiny Code (Python)

```
import math

def ops_estimate(n):
    return {
        "O(1)": 1,
        "O(log n)": math.log2(n),
        "O(n)": n,
        "O(n log n)": n * math.log2(n),
        "O(n^2)": n2
    }

print(ops_estimate(106))
```

## Why It Matters

- Builds *numerical intuition* for asymptotics
- Helps choose the right algorithm for large  $n$
- Explains why  $O(n^2)$  might work for  $n = 1000$  but not  $n = 10^6$
- Connects abstract math to real-world feasibility

## A Gentle Proof (Why It Works)

Each complexity class describes a function  $f(n)$  bounding operations. Comparing  $f(n)$  for common  $n$  values illustrates relative growth rates. Because asymptotic notation suppresses constants, differences in growth dominate as  $n$  grows.

Thus, numerical examples are faithful approximations of asymptotic behavior.

## Try It Yourself

1. Fill the table for  $n = 10^3, 10^4, 10^6$ .
2. Plot growth curves for each  $f(n)$ .
3. Compare runtime if each operation = 1 microsecond.
4. Identify feasible vs infeasible complexities for your hardware.

## Test Cases

| $n$    | $O(1)$ | $O(\log n)$ | $O(n)$        | $O(n^2)$  |
|--------|--------|-------------|---------------|-----------|
| $10^3$ | 1      | 10          | 1,000         | 1,000,000 |
| $10^6$ | 1      | 20          | 1,000,000     | $10^{12}$ |
| $10^9$ | 1      | 30          | 1,000,000,000 | $10^{18}$ |

## Complexity

| Operation        | Type   | Insight          |
|------------------|--------|------------------|
| Table Generation | $O(1)$ | Static reference |
| Evaluation       | $O(1)$ | Analytical       |

A Time Complexity Table turns abstract Big-O notation into a living chart, where  $O(\log n)$  feels tiny,  $O(n^2)$  feels heavy, and  $O(2^n)$  feels impossible.

## 18 Space–Time Tradeoff Explorer

A Space–Time Tradeoff Explorer helps us understand one of the most fundamental balances in algorithm design: using more memory to gain speed, or saving memory at the cost of time. It's the art of finding equilibrium between storage and computation.

## What Problem Are We Solving?

We often face a choice:

- Precompute and store results for instant access (more space, less time)
- Compute on demand to save memory (less space, more time)

The goal is to analyze both sides and choose the best fit for the problem's constraints.

## How It Works (Plain Language)

1. Identify repeated computations that can be stored.
2. Estimate memory cost of storing precomputed data.
3. Estimate time saved per query or reuse.
4. Compare tradeoffs using total cost models:

$$\text{Total Cost} = \text{Time Cost} + \lambda \cdot \text{Space Cost}$$

where  $\lambda$  reflects system priorities.

5. Decide whether caching, tabulation, or recomputation is preferable.

You're tuning performance with two dials, one for memory, one for time.

## Example Step by Step

Example 1: Fibonacci Numbers

- Recursive (no memory):  $O(2^n)$  time,  $O(1)$  space
- Memoized:  $O(n)$  time,  $O(n)$  space
- Iterative (tabulated):  $O(n)$  time,  $O(1)$  space (store only last two)

Different tradeoffs for the same problem.

Example 2: Lookup Tables

Suppose you need  $\sin(x)$  for many  $x$  values:

- Compute each time  $\rightarrow O(n)$  per query
- Store all results  $\rightarrow O(n)$  memory,  $O(1)$  lookup
- Hybrid: store sampled points, interpolate  $\rightarrow$  balance

## Tiny Code (Python)

```

def fib_naive(n):
    if n <= 1: return n
    return fib_naive(n-1) + fib_naive(n-2)

def fib_memo(n, memo={}):
    if n in memo: return memo[n]
    if n <= 1: return n
    memo[n] = fib_memo(n-1, memo) + fib_memo(n-2, memo)
    return memo[n]

```

Compare time vs memory for each version.

## Why It Matters

- Helps design algorithms under memory limits or real-time constraints
- Essential in databases, graphics, compilers, and AI caching
- Connects theory (asymptotics) to engineering (resources)
- Promotes thinking in trade curves, not absolutes

## A Gentle Proof (Why It Works)

Let  $T(n) = \text{time}$ ,  $S(n) = \text{space}$ . If we precompute  $k$  results,

$$T'(n) = T(n) - \Delta T, \quad S'(n) = S(n) + \Delta S$$

Since  $\Delta T$  and  $\Delta S$  are usually monotonic, minimizing one increases the other. Thus, the optimal configuration lies where

$$\frac{dT}{dS} = -\lambda$$

reflecting the system's valuation of time vs memory.

## Try It Yourself

1. Compare naive vs memoized vs iterative Fibonacci.
2. Build a lookup table for factorials modulo  $M$ .
3. Explore DP tabulation (space-heavy) vs rolling array (space-light).
4. Evaluate caching in a recursive tree traversal.

## Test Cases

| Problem      | Space  | Time     | Strategy        |
|--------------|--------|----------|-----------------|
| Fibonacci    | $O(1)$ | $O(2^n)$ | Naive recursion |
| Fibonacci    | $O(n)$ | $O(n)$   | Memoization     |
| Fibonacci    | $O(1)$ | $O(n)$   | Iterative       |
| Lookup Table | $O(n)$ | $O(1)$   | Precompute      |
| Recompute    | $O(1)$ | $O(n)$   | On-demand       |

## Complexity

| Operation           | Dimension | Note           |
|---------------------|-----------|----------------|
| Space–Time Analysis | $O(1)$    | Conceptual     |
| Optimization        | $O(1)$    | Tradeoff curve |

A Space–Time Tradeoff Explorer turns resource limits into creative levers, helping you choose when to remember, when to recompute, and when to balance both in harmony.

## 19 Profiling Algorithm

Profiling an algorithm means measuring how it *actually behaves*, how long it runs, how much memory it uses, how often loops iterate, and where time is really spent. It turns theoretical complexity into real performance data.

### What Problem Are We Solving?

Big-O tells us how an algorithm scales, but not how it *performs in practice*. Constant factors, system load, compiler optimizations, and cache effects all matter.

Profiling answers:

- Where is the time going?
- Which function dominates?
- Are we bound by CPU, memory, or I/O?

It's the microscope for runtime behavior.

## How It Works (Plain Language)

1. Instrument your code, insert timers, counters, or use built-in profilers.
2. Run with representative inputs.
3. Record runtime, call counts, and memory allocations.
4. Analyze hotspots, the 10% of code causing 90% of cost.
5. Optimize only where it matters.

Profiling doesn't guess, it measures.

## Example Step by Step

### Example 1: Timing a Function

```
import time

start = time.perf_counter()
result = algorithm(n)
end = time.perf_counter()

print("Elapsed:", end - start)
```

Measure total runtime for a given input size.

### Example 2: Line-Level Profiling

```
import cProfile, pstats

cProfile.run('algorithm(1000)', 'stats')
p = pstats.Stats('stats')
p.sort_stats('cumtime').print_stats(10)
```

Shows the 10 most time-consuming functions.

## Tiny Code (Python)

```

def slow_sum(n):
    s = 0
    for i in range(n):
        for j in range(i):
            s += j
    return s

import cProfile
cProfile.run('slow_sum(500)')

```

Output lists functions, calls, total time, and cumulative time.

## Why It Matters

- Bridges theory (Big-O) and practice (runtime)
- Identifies bottlenecks for optimization
- Validates expected scaling across inputs
- Prevents premature optimization, measure first, fix later

## A Gentle Proof (Why It Works)

Every algorithm execution is a trace of operations. Profiling samples or counts these operations in real time.

If  $t_i$  is time spent in component  $i$ , then total runtime  $T = \sum_i t_i$ . Ranking  $t_i$  reveals the dominant terms empirically, confirming or refuting theoretical assumptions.

## Try It Yourself

1. Profile a recursive function (like Fibonacci).
2. Compare iterative vs recursive runtimes.
3. Plot  $n$  vs runtime to visualize empirical complexity.
4. Use `memory_profiler` to capture space usage.

| Algorithm | Expected | Observed (example) | Notes |
|-----------|----------|--------------------|-------|
|-----------|----------|--------------------|-------|

## Test Cases

| Algorithm     | Expected      | Observed (example)                     | Notes                     |
|---------------|---------------|----------------------------------------|---------------------------|
| Linear Search | $O(n)$        | runtime $n$                            | scales linearly           |
| Merge Sort    | $O(n \log n)$ | runtime grows slightly faster than $n$ | merge overhead            |
| Naive         | $O(2^n)$      | explodes at $n > 30$                   | confirms exponential cost |
| Fibonacci     |               |                                        |                           |

## Complexity

| Operation         | Time                  | Space  |
|-------------------|-----------------------|--------|
| Profiling Run     | $O(n)$ (per trial)    | $O(1)$ |
| Report Generation | $O(f)$ (per function) | $O(f)$ |

Profiling is where math meets the stopwatch, transforming asymptotic guesses into concrete numbers and revealing the true heartbeat of your algorithm.

## 20 Benchmarking Framework

A Benchmarking Framework provides a structured way to compare algorithms under identical conditions. It measures performance across input sizes, multiple trials, and varying hardware, revealing which implementation truly performs best in practice.

### What Problem Are We Solving?

You've got several algorithms solving the same problem — which one is *actually faster*? Which scales better? Which uses less memory?

Benchmarking answers these questions with fair, repeatable experiments instead of intuition or isolated timing tests.

## How It Works (Plain Language)

1. Define test cases (input sizes, data patterns).
2. Run all candidate algorithms under the same conditions.
3. Repeat trials to reduce noise.
4. Record metrics:
  - Runtime
  - Memory usage
  - Throughput or latency
5. Aggregate results and visualize trends.

Think of it as a “tournament” where each algorithm plays by the same rules.

## Example Step by Step

Suppose we want to benchmark sorting methods:

1. Inputs: random arrays of sizes  $10^3$ ,  $10^4$ ,  $10^5$
2. Algorithms: `bubble_sort`, `merge_sort`, `timsort`
3. Metric: average runtime over 5 runs
4. Result: table or plot

| Size   | Bubble Sort | Merge Sort | Timsort |
|--------|-------------|------------|---------|
| $10^3$ | 0.05s       | 0.001s     | 0.0008s |
| $10^4$ | 5.4s        | 0.02s      | 0.012s  |
| $10^5$ | —           | 0.25s      | 0.15s   |

Timsort wins across all sizes, data confirms theory.

## Tiny Code (Python)

```
import timeit
import random

def bench(func, n, trials=5):
    data = [random.randint(0, n) for _ in range(n)]
```

```

    return min(timeit.repeat(lambda: func(data.copy()), number=1, repeat=trials))

def bubble_sort(arr):
    for i in range(len(arr)):
        for j in range(len(arr)-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]

def merge_sort(arr):
    if len(arr) <= 1: return arr
    mid = len(arr)//2
    return merge(merge_sort(arr[:mid]), merge_sort(arr[mid:]))

def merge(left, right):
    result = []
    while left and right:
        result.append(left.pop(0) if left[0] < right[0] else right.pop(0))
    return result + left + right

print("Bubble:", bench(bubble_sort, 1000))
print("Merge:", bench(merge_sort, 1000))

```

## Why It Matters

- Converts abstract complexity into empirical performance
- Supports evidence-based optimization
- Detects constant factor effects Big-O hides
- Ensures fair comparisons across algorithms

## A Gentle Proof (Why It Works)

Let  $t_{i,j}$  be time of algorithm  $i$  on trial  $j$ . Benchmarking reports min, max, or mean( $t_{i,*}$ ).

By controlling conditions (hardware, input distribution), we treat  $t_{i,j}$  as samples of the same distribution, allowing valid comparisons of  $E[t_i]$  (expected runtime). Hence, results reflect true relative performance.

## Try It Yourself

1. Benchmark linear vs binary search on sorted arrays.

2. Test dynamic array insertion vs linked list insertion.
3. Run across input sizes  $10^3$ ,  $10^4$ ,  $10^5$ .
4. Plot results:  $n$  (x-axis) vs time (y-axis).

## Test Cases

| Comparison              | Expectation                         |
|-------------------------|-------------------------------------|
| Bubble vs Merge         | Merge faster after small $n$        |
| Linear vs Binary Search | Binary faster for $n > 100$         |
| List vs Dict lookup     | Dict $O(1)$ outperforms List $O(n)$ |

## Complexity

| Step              | Time    | Space  |
|-------------------|---------|--------|
| Run Each Trial    | $O(n)$  | $O(1)$ |
| Aggregate Results | $O(k)$  | $O(k)$ |
| Total Benchmark   | $O(nk)$ | $O(1)$ |

( $k$  = number of trials)

A Benchmarking Framework transforms comparison into science, fair tests, real data, and performance truths grounded in experiment, not hunch.

## Section 3. Big-O, Big-Theta, Big-Omega

### 21 Growth Rate Comparator

A Growth Rate Comparator helps us see how functions grow relative to each other, the backbone of asymptotic reasoning. It lets us answer questions like: does  $n^2$  outgrow  $n \log n$ ? How fast is  $2^n$  compared to  $n!$ ?

#### What Problem Are We Solving?

We need a clear way to compare how fast two functions increase as  $n$  becomes large. When analyzing algorithms, runtime functions like  $n$ ,  $n \log n$ , and  $n^2$  all seem similar at small scales, but their growth rates diverge quickly.

A comparator gives us a mathematical and visual way to rank them.

## How It Works (Plain Language)

1. Write the two functions  $f(n)$  and  $g(n)$ .
2. Compute the ratio  $\frac{f(n)}{g(n)}$  as  $n \rightarrow \infty$ .
3. Interpret the result:
  - If  $\frac{f(n)}{g(n)} \rightarrow 0 \rightarrow f(n) = o(g(n))$  (grows slower)
  - If  $\frac{f(n)}{g(n)} \rightarrow c > 0 \rightarrow f(n) = \Theta(g(n))$  (same growth)
  - If  $\frac{f(n)}{g(n)} \rightarrow \infty \rightarrow f(n) = \omega(g(n))$  (grows faster)

This ratio test tells us which function dominates for large  $n$ .

## Example Step by Step

Example 1: Compare  $n \log n$  vs  $n^2$

$$\frac{n \log n}{n^2} = \frac{\log n}{n}$$

As  $n \rightarrow \infty$ ,  $\frac{\log n}{n} \rightarrow 0 \rightarrow n \log n = o(n^2)$

Example 2: Compare  $2^n$  vs  $n!$

$$\frac{2^n}{n!} \rightarrow 0$$

since  $n!$  grows faster than  $2^n$ .  $\rightarrow 2^n = o(n!)$

## Tiny Code (Python)

```
import math

def compare_growth(f, g, ns):
    for n in ns:
        ratio = f(n)/g(n)
        print(f"n={n:6}, ratio={ratio:.6e}")
```

```
compare_growth(lambda n: n*math.log2(n),
               lambda n: n2,
               [10, 100, 1000, 10000])
```

Output shows ratio shrinking → confirms slower growth.

## Why It Matters

- Builds intuition for asymptotic dominance
- Essential for Big-O, Big-Theta, Big-Omega proofs
- Clarifies why some algorithms scale better
- Translates math into visual and numerical comparisons

## A Gentle Proof (Why It Works)

By definition of asymptotic notation:

If  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$ , then for any  $\varepsilon > 0$ ,  $f(n) < \varepsilon g(n)$  for large  $n$ .

Thus,  $f(n)$  grows slower than  $g(n)$ .

This formal limit test underlies Big-O reasoning.

## Try It Yourself

1. Compare  $n^3$  vs  $2^n$
2. Compare  $\sqrt{n}$  vs  $\log n$
3. Compare  $n!$  vs  $n^n$
4. Plot both functions and see where one overtakes the other

## Test Cases

| $f(n)$   | $g(n)$     | Result | Relation      |
|----------|------------|--------|---------------|
| $\log n$ | $\sqrt{n}$ | 0      | $o(\sqrt{n})$ |
| $n$      | $n \log n$ | 0      | $o(n \log n)$ |
| $n^2$    | $2^n$      | 0      | $o(2^n)$      |
| $2^n$    | $n!$       | 0      | $o(n!)$       |

## Complexity

| Operation  | Time            | Space  |
|------------|-----------------|--------|
| Comparison | $O(1)$ per pair | $O(1)$ |

A Growth Rate Comparator turns asymptotic theory into a conversation, showing, with numbers and limits, who really grows faster as  $n$  climbs toward infinity.

## 22 Dominant Term Extractor

A Dominant Term Extractor simplifies complexity expressions by identifying which term matters most as  $n$  grows large. It's how we turn messy runtime formulas into clean Big-O notation, by keeping only what truly drives growth.

### What Problem Are We Solving?

Algorithms often produce composite cost formulas like

$$T(n) = 3n^2 + 10n + 25$$

Not all terms grow equally. The dominant term determines long-run behavior, so we want to isolate it and discard the rest.

This step bridges detailed operation counting and asymptotic notation.

### How It Works (Plain Language)

1. Write the runtime function  $T(n)$  (from counting steps).
2. List all terms by their growth type ( $n^3$ ,  $n^2$ ,  $n$ ,  $\log n$ , constants).
3. Find the fastest-growing term as  $n \rightarrow \infty$ .
4. Drop coefficients and lower-order terms.
5. The result is the Big-O class.

Think of it as zooming out on a curve, smaller waves vanish at infinity.

## Example Step by Step

Example 1:

$$T(n) = 5n^3 + 2n^2 + 7n + 12$$

For large  $n$ ,  $n^3$  dominates.

So:

$$T(n) = O(n^3)$$

Example 2:

$$T(n) = n^2 + n \log n + 10n$$

Compare term by term:

$$n^2 > n \log n > n$$

So dominant term is  $n^2$ .  $\Rightarrow T(n) = O(n^2)$

## Tiny Code (Python)

```
def dominant_term(terms):
    growth_order = {'1': 0, 'logn': 1, 'n': 2, 'nlogn': 3, 'n^2': 4, 'n^3': 5, '2^n': 6}
    return max(terms, key=lambda t: growth_order[t])

print(dominant_term(['n^2', 'nlogn', 'n'])) # n^2
```

You can extend this with symbolic simplification using SymPy.

## Why It Matters

- Simplifies detailed formulas into clean asymptotics
- Focuses attention on scaling behavior, not constants
- Makes performance comparison straightforward
- Core step in deriving Big-O from raw step counts

## A Gentle Proof (Why It Works)

Let

$$T(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_0$$

As  $n \rightarrow \infty$ ,

$$\frac{a_{k-1} n^{k-1}}{a_k n^k} = \frac{a_{k-1}}{a_k} \rightarrow 0$$

All lower-order terms vanish relative to the largest exponent. So  $T(n) = \Theta(n^k)$ .

This generalizes beyond polynomials to any family of functions with strict growth ordering.

## Try It Yourself

1. Simplify  $T(n) = 4n \log n + 10n + 100$ .
2. Simplify  $T(n) = 2n^3 + 50n^2 + 1000$ .
3. Simplify  $T(n) = 5n + 10 \log n + 100$ .
4. Verify using ratio test:  $\frac{\text{lower term}}{\text{dominant term}} \rightarrow 0$ .

## Test Cases

| Expression          | Dominant Term | Big-O         |
|---------------------|---------------|---------------|
| $3n^2 + 4n + 10$    | $n^2$         | $O(n^2)$      |
| $5n + 8 \log n + 7$ | $n$           | $O(n)$        |
| $n \log n + 100n$   | $n \log n$    | $O(n \log n)$ |
| $4n^3 + n^2 + 2n$   | $n^3$         | $O(n^3)$      |

## Complexity

| Operation  | Time   | Space  |
|------------|--------|--------|
| Extraction | $O(k)$ | $O(1)$ |

( $k$  = number of terms)

A Dominant Term Extractor is like a spotlight, it shines on the one term that decides the pace, letting you see the true asymptotic character of your algorithm.

## 23 Limit-Based Complexity Test

The Limit-Based Complexity Test is a precise way to compare how fast two functions grow by using limits. It's a mathematical tool that turns intuition ("this one feels faster") into proof ("this one *is* faster").

### What Problem Are We Solving?

When analyzing algorithms, we often ask: Does  $f(n)$  grow slower, equal, or faster than  $g(n)$ ? Instead of guessing, we use limits to determine the exact relationship and classify them using Big-O,  $\Theta$ , or  $\Omega$ .

This method gives a formal and reliable comparison of growth rates.

### How It Works (Plain Language)

1. Start with two positive functions  $f(n)$  and  $g(n)$ .
2. Compute the ratio:

$$L = \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$$

3. Interpret the limit:

- If  $L = 0$ , then  $f(n) = o(g(n)) \rightarrow f$  grows slower.
- If  $0 < L < \infty$ , then  $f(n) = \Theta(g(n)) \rightarrow$  same growth rate.
- If  $L = \infty$ , then  $f(n) = \omega(g(n)) \rightarrow f$  grows faster.

The ratio tells us how one function "scales" relative to another.

### Example Step by Step

Example 1:

Compare  $f(n) = n \log n$  and  $g(n) = n^2$ .

$$\frac{f(n)}{g(n)} = \frac{n \log n}{n^2} = \frac{\log n}{n}$$

As  $n \rightarrow \infty$ ,  $\frac{\log n}{n} \rightarrow 0$ . So  $n \log n = o(n^2) \rightarrow$  grows slower.

Example 2:

Compare  $f(n) = 3n^2 + 4n$  and  $g(n) = n^2$ .

$$\frac{f(n)}{g(n)} = \frac{3n^2 + 4n}{n^2} = 3 + \frac{4}{n}$$

As  $n \rightarrow \infty$ ,  $\frac{4}{n} \rightarrow 0$ . So  $\lim$  = 3, constant and positive. Therefore,  $f(n) = \Theta(g(n))$ .

### Tiny Code (Python)

```
import sympy as sp

n = sp.symbols('n', positive=True)
f = n * sp.log(n)
g = n**2
L = sp.limit(f/g, n, sp.oo)
print("Limit:", L)
```

Outputs 0, confirming  $n \log n = o(n^2)$ .

### Why It Matters

- Provides formal proof of asymptotic relationships
- Eliminates guesswork in comparing growth rates
- Core step in Big-O proofs and recurrence analysis
- Helps verify if approximations are valid

### A Gentle Proof (Why It Works)

The definition of asymptotic comparison uses limits:

If  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$ , then for any  $\varepsilon > 0$ ,  $\exists N$  such that  $\forall n > N$ ,  $f(n) \leq \varepsilon g(n)$ .

This satisfies the formal condition for  $f(n) = o(g(n))$ . Similarly, constant or infinite limits define  $\Theta$  and  $\omega$ .

### Try It Yourself

1. Compare  $n^3$  vs  $2^n$ .
2. Compare  $\sqrt{n}$  vs  $\log n$ .
3. Compare  $n!$  vs  $n^n$ .
4. Check ratio for  $n^2 + n$  vs  $n^2$ .

## Test Cases

| $f(n)$    | $g(n)$     | Limit    | Relationship   |
|-----------|------------|----------|----------------|
| $n$       | $n \log n$ | 0        | $o(g(n))$      |
| $n^2 + n$ | $n^2$      | 1        | $\Theta(g(n))$ |
| $2^n$     | $n^3$      | $\infty$ | $\omega(g(n))$ |
| $\log n$  | $\sqrt{n}$ | 0        | $o(g(n))$      |

## Complexity

| Operation        | Time            | Space  |
|------------------|-----------------|--------|
| Limit Evaluation | $O(1)$ symbolic | $O(1)$ |

The Limit-Based Complexity Test is your mathematical magnifying glass, a clean, rigorous way to compare algorithmic growth and turn asymptotic intuition into certainty.

## 24 Summation Simplifier

A Summation Simplifier converts loops and recursive cost expressions into closed-form formulas using algebra and known summation rules. It bridges the gap between raw iteration counts and Big-O notation.

### What Problem Are We Solving?

When analyzing loops, we often get total work expressed as a sum:

$$T(n) = \sum_{i=1}^n i \quad \text{or} \quad T(n) = \sum_{i=1}^n \log i$$

But Big-O requires us to simplify these sums into familiar functions of  $n$ . Summation simplification transforms iteration patterns into asymptotic form.

## How It Works (Plain Language)

1. Write down the summation from your loop or recurrence.
2. Apply standard formulas or approximations:

- $\sum_{i=1}^n 1 = n$
- $\sum_{i=1}^n i = \frac{n(n+1)}{2}$
- $\sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6}$
- $\sum_{i=1}^n \log i = O(n \log n)$

3. Drop constants and lower-order terms.
4. Return simplified function  $f(n) \rightarrow$  then apply Big-O.

It's like algebraic compression for iteration counts.

## Example Step by Step

Example 1:

$$T(n) = \sum_{i=1}^n i$$

Use formula:

$$T(n) = \frac{n(n+1)}{2}$$

Simplify:

$$T(n) = O(n^2)$$

Example 2:

$$T(n) = \sum_{i=1}^n \log i$$

Approximate by integral:

$$\int_1^n \log x, dx = n \log n - n + 1$$

So  $T(n) = O(n \log n)$

Example 3:

$$T(n) = \sum_{i=1}^n \frac{1}{i}$$

$\log n$  (Harmonic series)

## Tiny Code (Python)

```
import sympy as sp

i, n = sp.symbols('i n', positive=True)
expr = sp.summation(i, (i, 1, n))
print(sp.simplify(expr)) # n*(n+1)/2
```

Or use `sp.summation(sp.log(i), (i,1,n))` for logarithmic sums.

## Why It Matters

- Converts nested loops into analyzable formulas
- Core tool in time complexity derivation
- Helps visualize how cumulative work builds up
- Connects discrete steps with continuous approximations

## A Gentle Proof (Why It Works)

If  $f(i)$  is positive and increasing, then by the integral test:

$$\int_1^n f(x), dx \leq \sum_{i=1}^n f(i) \leq f(n) + \int_1^n f(x), dx$$

So for asymptotic purposes,  $\sum f(i)$  and  $\int f(x)$  grow at the same rate.

This equivalence justifies approximations like  $\sum \log i = O(n \log n)$ .

## Try It Yourself

1. Simplify  $\sum_{i=1}^n i^3$ .
2. Simplify  $\sum_{i=1}^n \sqrt{i}$ .
3. Simplify  $\sum_{i=1}^n \frac{1}{i^2}$ .
4. Approximate  $\sum_{i=1}^{n/2} i$  using integrals.

## Test Cases

| Summation          | Formula                  | Big-O         |
|--------------------|--------------------------|---------------|
| $\sum 1$           | $n$                      | $O(n)$        |
| $\sum i$           | $\frac{n(n+1)}{2}$       | $O(n^2)$      |
| $\sum i^2$         | $\frac{n(n+1)(2n+1)}{6}$ | $O(n^3)$      |
| $\sum \log i$      | $n \log n$               | $O(n \log n)$ |
| $\sum \frac{1}{i}$ | $\log n$                 | $O(\log n)$   |

## Complexity

| Operation      | Time                    | Space  |
|----------------|-------------------------|--------|
| Simplification | $O(1)$ (formula lookup) | $O(1)$ |

A Summation Simplifier turns looping arithmetic into elegant formulas, the difference between counting steps and *seeing* the shape of growth.

## 25 Recurrence Tree Method

The Recurrence Tree Method is a visual technique for solving divide-and-conquer recurrences. It expands the recursive formula into a tree of subproblems, sums the work done at each level, and reveals the total cost.

### What Problem Are We Solving?

Many recursive algorithms (like Merge Sort or Quick Sort) define their running time as

$$T(n) = a, T! \left( \frac{n}{b} \right) + f(n)$$

where:

- $a$  = number of subproblems,
- $b$  = size reduction factor,
- $f(n)$  = non-recursive work per call.

The recurrence tree lets us see the full cost by summing over levels instead of applying a closed-form theorem immediately.

## How It Works (Plain Language)

1. Draw the recursion tree
    - Root: problem of size  $n$ , cost  $f(n)$ .
    - Each node: subproblem of size  $\frac{n}{b}$  with cost  $f(\frac{n}{b})$ .
  2. Expand levels until base case ( $n = 1$ ).
  3. Sum work per level:
    - Level  $i$  has  $a^i$  nodes, each size  $\frac{n}{b^i}$ .
    - Total work at level  $i$ :
$$W_i = a^i \cdot f! \left( \frac{n}{b^i} \right)$$
  4. Add all levels:
- $$T(n) = \sum_{i=0}^{\log_b n} W_i$$
5. Identify the dominant level (top, middle, or bottom).
  6. Simplify to Big-O form.

## Example Step by Step

Take Merge Sort:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

Level 0:  $1 \times n = n$  Level 1:  $2 \times \frac{n}{2} = n$  Level 2:  $4 \times \frac{n}{4} = n$  Depth:  $\log_2 n$  levels

Total work:

$$T(n) = n \log_2 n + n = O(n \log n)$$

Every level costs  $n$ , total =  $n \times \log n$ .

## Tiny Code (Python)

```

import math

def recurrence_tree(a, b, f, n):
    total = 0
    level = 0
    while n >= 1:
        work = (a * level) * f(n/(b * level))
        total += work
        level += 1
        n /= b
    return total

```

Use  $f = \lambda x: x$  for  $f(n) = n$ .

## Why It Matters

- Makes recurrence structure visible and intuitive
- Explains why Master Theorem results hold
- Highlights dominant levels (top-heavy vs bottom-heavy)
- Great teaching and reasoning tool for recursive cost breakdown

## A Gentle Proof (Why It Works)

Each recursive call contributes  $f(n)$  work plus child subcalls. Because each level's subproblems have equal size, total cost is additive:

$$T(n) = \sum_{i=0}^{\log_b n} a^i f\left(\frac{n}{b^i}\right)$$

Dominant level dictates asymptotic order:

- Top-heavy:  $f(n)$  dominates  $\rightarrow O(f(n))$
- Balanced: all levels equal  $\rightarrow O(f(n) \log n)$
- Bottom-heavy: leaves dominate  $\rightarrow O(n^{\log_b a})$

This reasoning leads directly to the Master Theorem.

## Try It Yourself

1. Build tree for  $T(n) = 3T(n/2) + n$ .
2. Sum each level's work.
3. Compare with Master Theorem result.
4. Try  $T(n) = T(n/2) + 1$  (logarithmic tree).

## Test Cases

| Recurrence    | Level Work                         | Levels           | Total      | Big-O         |
|---------------|------------------------------------|------------------|------------|---------------|
| $2T(n/2) + n$ |                                    | $\log n$         | $n \log n$ | $O(n \log n)$ |
| $T(n/2) + 1$  | 1                                  | $\log n$         | $\log n$   | $O(\log n)$   |
| $4T(n/2) + n$ | $a^i = 4^i$ , work = $n \cdot 2^i$ | bottom dominates | $O(n^2)$   |               |

## Complexity

| Step              | Time               | Space       |
|-------------------|--------------------|-------------|
| Tree Construction | $O(\log n)$ levels | $O(\log n)$ |

The Recurrence Tree Method turns abstract formulas into living diagrams, showing each layer's effort, revealing which level truly drives the algorithm's cost.

## 26 Master Theorem Evaluator

The Master Theorem Evaluator gives a quick, formula-based way to solve divide-and-conquer recurrences of the form

$$T(n) = a, T! \left( \frac{n}{b} \right) + f(n)$$

It tells you the asymptotic behavior of  $T(n)$  without full expansion or summation, a shortcut born from the recurrence tree.

## What Problem Are We Solving?

We want to find the Big-O complexity of divide-and-conquer algorithms quickly. Manually expanding recursions (via recurrence trees) works, but is tedious. The Master Theorem classifies solutions by comparing the recursive work ( $a, T(n/b)$ ) and non-recursive work ( $f(n)$ ).

## How It Works (Plain Language)

Given

$$T(n) = a, T! \left( \frac{n}{b} \right) + f(n)$$

- $a$  = number of subproblems
- $b$  = shrink factor
- $f(n)$  = work done outside recursion

Compute critical exponent:

$$n^{\log_b a}$$

Compare  $f(n)$  to  $n^{\log_b a}$ :

1. Case 1 (Top-heavy): If  $f(n) = O(n^{\log_b a - \varepsilon})$ ,

$$T(n) = \Theta(n^{\log_b a})$$

Recursive part dominates.

2. Case 2 (Balanced): If  $f(n) = \Theta(n^{\log_b a} \log^k n)$ ,

$$T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$$

Both contribute equally.

3. Case 3 (Bottom-heavy): If  $f(n) = \Omega(n^{\log_b a + \varepsilon})$  and regularity condition holds:

$$af(n/b) \leq cf(n)$$

for some  $c < 1$ , then

$$T(n) = \Theta(f(n))$$

Non-recursive part dominates.

## Example Step by Step

Example 1:

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

- $a = 2, b = 2, f(n) = n$
- $n^{\log_2 2} = n$  So  $f(n) = \Theta(n^{\log_2 2}) \rightarrow$  Case 2

$$T(n) = \Theta(n \log n)$$

Example 2:

$$T(n) = 4T(n/2) + n$$

- $a = 4, b = 2 \rightarrow n^{\log_2 4} = n^2$
- $f(n) = n = O(n^{2-\varepsilon}) \rightarrow$  Case 1

$$T(n) = \Theta(n^2)$$

Example 3:

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

- $a = 1, b = 2, n^{\log_2 1} = 1$
- $f(n) = n = \Omega(n^{0+\varepsilon}) \rightarrow$  Case 3

$$T(n) = \Theta(n)$$

### Tiny Code (Python)

```
import math

def master_theorem(a, b, f_exp):
    critical = math.log(a, b)
    if f_exp < critical:
        return f"O(n^{critical:.2f})"
    elif f_exp == critical:
        return f"O(n^{critical:.2f} log n)"
    else:
        return f"O(n^{f_exp})"
```

For  $T(n) = 2T(n/2) + n$ , call `master_theorem(2, 2, 1) → O(n log n)`

### Why It Matters

- Solves recurrences in seconds
- Foundation for analyzing divide-and-conquer algorithms
- Validates intuition from recurrence trees
- Used widely in sorting, searching, matrix multiplication, FFT

## A Gentle Proof (Why It Works)

Each recursion level costs:

$$a^i, f! \left( \frac{n}{b^i} \right)$$

Total cost:

$$T(n) = \sum_{i=0}^{\log_b n} a^i f! \left( \frac{n}{b^i} \right)$$

The relative growth of  $f(n)$  to  $n^{\log_b a}$  determines which level dominates, top, middle, or bottom, yielding the three canonical cases.

## Try It Yourself

1.  $T(n) = 3T(n/2) + n$
2.  $T(n) = 2T(n/2) + n^2$
3.  $T(n) = 8T(n/2) + n^3$
4. Identify  $a, b, f(n)$  and apply theorem.

## Test Cases

| Recurrence           | Case | Result          |
|----------------------|------|-----------------|
| $2T(n/2) + n$        | 2    | $O(n \log n)$   |
| $4T(n/2) + n$        | 1    | $O(n^2)$        |
| $T(n/2) + n$         | 3    | $O(n)$          |
| $3T(n/3) + n \log n$ | 2    | $O(n \log^2 n)$ |

## Complexity

| Step       | Time   | Space  |
|------------|--------|--------|
| Evaluation | $O(1)$ | $O(1)$ |

The Master Theorem Evaluator is your formulaic compass, it points instantly to the asymptotic truth hidden in recursive equations, no tree-drawing required.

## 27 Big-Theta Proof Builder

A Big-Theta Proof Builder helps you formally prove that a function grows at the same rate as another. It's the precise way to show that  $f(n)$  and  $g(n)$  are asymptotically equivalent, growing neither faster nor slower beyond constant factors.

### What Problem Are We Solving?

We often say an algorithm is  $T(n) = \Theta(n \log n)$ , but how do we prove it? A Big-Theta proof uses inequalities to pin  $T(n)$  between two scaled versions of a simpler function  $g(n)$ , confirming tight asymptotic bounds.

This transforms intuition into rigorous evidence.

### How It Works (Plain Language)

We say

$$f(n) = \Theta(g(n))$$

if there exist constants  $c_1, c_2 > 0$  and  $n_0$  such that for all  $n \geq n_0$ :

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

So  $f(n)$  is sandwiched between two constant multiples of  $g(n)$ .

Steps:

1. Identify  $f(n)$  and candidate  $g(n)$ .
2. Find constants  $c_1, c_2$ , and threshold  $n_0$ .
3. Verify inequality for all  $n \geq n_0$ .
4. Conclude  $f(n) = \Theta(g(n))$ .

### Example Step by Step

Example 1:

$$f(n) = 3n^2 + 10n + 5$$

Candidate:  $g(n) = n^2$

For large  $n$ ,  $10n + 5$  is small compared to  $3n^2$ .

We can show:

$$3n^2 \leq 3n^2 + 10n + 5 \leq 4n^2, \quad \text{for } n \geq 10$$

Thus,  $f(n) = \Theta(n^2)$  with  $c_1 = 3$ ,  $c_2 = 4$ ,  $n_0 = 10$ .

Example 2:

$$f(n) = n \log n + 100n$$

Candidate:  $g(n) = n \log n$

For  $n \geq 2$ ,  $\log n \geq 1$ , so  $100n \leq 100n \log n$ . Hence,

$$n \log n \leq f(n) \leq 101n \log n$$

$$\rightarrow f(n) = \Theta(n \log n)$$

### Tiny Code (Python)

```
def big_theta_proof(f, g, n0, c1, c2):
    for n in range(n0, n0 + 5):
        if not (c1*g(n) <= f(n) <= c2*g(n)):
            return False
    return True

f = lambda n: 3*n2 + 10*n + 5
g = lambda n: n2
print(big_theta_proof(f, g, 10, 3, 4)) # True
```

### Why It Matters

- Converts informal claims (“it’s  $n^2$ -ish”) into formal proofs
- Builds rigor in asymptotic reasoning
- Essential for algorithm analysis, recurrence proofs, and coursework
- Reinforces understanding of constants and thresholds

### A Gentle Proof (Why It Works)

By definition,

$$f(n) = \Theta(g(n)) \iff \exists c_1, c_2, n_0 : c_1 g(n) \leq f(n) \leq c_2 g(n)$$

This mirrors how Big-O and Big-Omega combine:

- $f(n) = O(g(n))$  gives upper bound,
- $f(n) = \Omega(g(n))$  gives lower bound. Together, they form a tight bound, hence  $\Theta$ .

## Try It Yourself

1. Prove  $5n^3 + n^2 + 100 = \Theta(n^3)$ .
2. Prove  $4n + 10 = \Theta(n)$ .
3. Show  $n \log n + 100n = \Theta(n \log n)$ .
4. Fail a proof:  $n^2 + 3n = \Theta(n)$  (not true).

## Test Cases

| $f(n)$            | $g(n)$     | $c_1, c_2, n_0$ | Result             |
|-------------------|------------|-----------------|--------------------|
| $3n^2 + 10n + 5$  | $n^2$      | 3, 4, 10        | $\Theta(n^2)$      |
| $n \log n + 100n$ | $n \log n$ | 1, 101, 2       | $\Theta(n \log n)$ |
| $10n + 50$        | $n$        | 10, 11, 5       | $\Theta(n)$        |

## Complexity

| Step         | Time              | Space  |
|--------------|-------------------|--------|
| Verification | $O(1)$ (symbolic) | $O(1)$ |

The Big-Theta Proof Builder is your asymptotic courtroom, you bring evidence, constants, and inequalities, and the proof delivers a verdict:  $\Theta(g(n))$ , beyond reasonable doubt.

## 28 Big-Omega Case Finder

A Big-Omega Case Finder helps you identify lower bounds on an algorithm's growth, the *guaranteed minimum* cost, even in the best-case scenario. It's the mirror image of Big-O, showing what an algorithm must at least do.

### What Problem Are We Solving?

Big-O gives us an upper bound ("it won't be slower than this"), but sometimes we need to know the floor, a complexity it can never beat.

Big-Omega helps us state:

- The fastest possible asymptotic behavior, or
- The minimal cost inherent to the problem itself.

This is key when analyzing best-case performance or complexity limits (like comparison sorting's  $\Omega(n \log n)$  lower bound).

### How It Works (Plain Language)

We say

$$f(n) = \Omega(g(n))$$

if  $\exists c > 0, n_0$  such that

$$f(n) \geq c \cdot g(n) \quad \text{for all } n \geq n_0$$

Steps:

1. Identify candidate lower-bound function  $g(n)$ .
2. Show  $f(n)$  eventually stays above a constant multiple of  $g(n)$ .
3. Find constants  $c$  and  $n_0$ .
4. Conclude  $f(n) = \Omega(g(n))$ .

### Example Step by Step

Example 1:

$$f(n) = 3n^2 + 5n + 10$$

Candidate:  $g(n) = n^2$

For  $n \geq 1$ ,

$$f(n) \geq 3n^2 \geq 3 \cdot n^2$$

So  $f(n) = \Omega(n^2)$  with  $c = 3, n_0 = 1$ .

Example 2:

$$f(n) = n \log n + 100n$$

Candidate:  $g(n) = n$

Since  $\log n \geq 1$  for  $n \geq 2$ ,

$$f(n) = n \log n + 100n \geq n + 100n = 101n$$

$\rightarrow f(n) = \Omega(n)$  with  $c = 101, n_0 = 2$

### Tiny Code (Python)

```

def big_omega_proof(f, g, n0, c):
    for n in range(n0, n0 + 5):
        if f(n) < c * g(n):
            return False
    return True

f = lambda n: 3*n2 + 5*n + 10
g = lambda n: n2
print(big_omega_proof(f, g, 1, 3))  # True

```

## Why It Matters

- Defines best-case performance
- Provides theoretical lower limits (impossible to beat)
- Complements Big-O (upper bound) and Theta (tight bound)
- Key in proving problem hardness or optimality

## A Gentle Proof (Why It Works)

If

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = L > 0,$$

then for any  $c \leq L$ ,  $f(n) \geq c \cdot g(n)$  for large  $n$ . Thus  $f(n) = \Omega(g(n))$ . This mirrors the formal definition of  $\Omega$  and follows directly from asymptotic ratio reasoning.

## Try It Yourself

1. Show  $4n^3 + n^2 = \Omega(n^3)$
2. Show  $n \log n + n = \Omega(n)$
3. Show  $2^n + n^5 = \Omega(2^n)$
4. Compare with their Big-O forms for contrast.

## Test Cases

| $f(n)$            | $g(n)$ | Constants          | Result        |
|-------------------|--------|--------------------|---------------|
| $3n^2 + 10n$      | $n^2$  | $c = 3, n_0 = 1$   | $\Omega(n^2)$ |
| $n \log n + 100n$ | $n$    | $c = 101, n_0 = 2$ | $\Omega(n)$   |
| $n^3 + n^2$       | $n^3$  | $c = 1, n_0 = 1$   | $\Omega(n^3)$ |

## Complexity

| Step         | Time   | Space  |
|--------------|--------|--------|
| Verification | $O(1)$ | $O(1)$ |

The Big-Omega Case Finder shows the *floor beneath the curve*, ensuring every algorithm stands on a solid lower bound, no matter how fast it tries to run.

## 29 Empirical Complexity Estimator

An Empirical Complexity Estimator bridges theory and experiment, it measures actual runtimes for various input sizes and fits them to known growth models like  $O(n)$ ,  $O(n \log n)$ , or  $O(n^2)$ . It's how we *discover* complexity when the math is unclear or the code is complex.

### What Problem Are We Solving?

Sometimes the exact formula for  $T(n)$  is too messy, or the implementation details are opaque. We can still estimate complexity empirically by observing how runtime changes as  $n$  grows.

This approach is especially useful for:

- Black-box code (unknown implementation)
- Experimental validation of asymptotic claims
- Comparing real-world scaling with theoretical predictions

### How It Works (Plain Language)

1. Choose representative input sizes  $n_1, n_2, \dots, n_k$ .
2. Measure runtime  $T(n_i)$  for each size.
3. Normalize or compare ratios:
  - $T(2n)/T(n) \approx 2 \rightarrow O(n)$
  - $T(2n)/T(n) \approx 4 \rightarrow O(n^2)$
  - $T(2n)/T(n) \approx \log 2 \rightarrow O(\log n)$
4. Fit data to candidate models using regression or ratio tests.
5. Visualize trends (e.g., log–log plot) to identify slope = exponent.

## Example Step by Step

Suppose we test input sizes:  $n = 1000, 2000, 4000, 8000$

| $n$  | $T(n)$ (ms) | Ratio $T(2n)/T(n)$ |
|------|-------------|--------------------|
| 1000 | 5           | —                  |
| 2000 | 10          | 2.0                |
| 4000 | 20          | 2.0                |
| 8000 | 40          | 2.0                |

Ratio  $\approx 2 \rightarrow$  linear growth  $\rightarrow T(n) = O(n)$

Now suppose:

| $n$  | $T(n)$ | Ratio |
|------|--------|-------|
| 1000 | 5      | —     |
| 2000 | 20     | 4     |
| 4000 | 80     | 4     |
| 8000 | 320    | 4     |

Ratio  $\approx 4 \rightarrow$  quadratic growth  $\rightarrow O(n^2)$

## Tiny Code (Python)

```
import time, math

def empirical_estimate(f, ns):
    times = []
    for n in ns:
        start = time.perf_counter()
        f(n)
        end = time.perf_counter()
        times.append(end - start)
    for i in range(1, len(ns)):
        ratio = times[i] / times[i-1]
        print(f"n={ns[i]:6}, ratio={ratio:.2f}")
```

Test with different algorithms to see scaling.

## Why It Matters

- Converts runtime data into Big-O form
- Detects bottlenecks or unexpected scaling
- Useful when theoretical analysis is hard
- Helps validate optimizations or refactors

## A Gentle Proof (Why It Works)

If  $T(n) \approx c \cdot f(n)$ , then the ratio test

$$\frac{T(kn)}{T(n)} \approx \frac{f(kn)}{f(n)}$$

reveals the exponent  $p$  if  $f(n) = n^p$ :

$$\frac{f(kn)}{f(n)} = k^p \implies p = \log_k \frac{T(kn)}{T(n)}$$

Repeated over multiple  $n$ , this converges to the true growth exponent.

## Try It Yourself

1. Measure runtime of sorting for increasing  $n$ .
2. Estimate  $p$  using ratio test.
3. Plot  $\log n$  vs  $\log T(n)$ , slope exponent.
4. Compare  $p$  to theoretical value.

## Test Cases

| Algorithm     | Observed Ratio | Estimated Complexity |
|---------------|----------------|----------------------|
| Bubble Sort   | 4              | $O(n^2)$             |
| Merge Sort    | 2.2            | $O(n \log n)$        |
| Linear Search | 2              | $O(n)$               |
| Binary Search | 1.1            | $O(\log n)$          |

## Complexity

| Step        | Time              | Space  |
|-------------|-------------------|--------|
| Measurement | $O(k \cdot T(n))$ | $O(k)$ |
| Estimation  | $O(k)$            | $O(1)$ |

( $k$  = number of sample points)

An Empirical Complexity Estimator transforms stopwatches into science, turning performance data into curves, curves into equations, and equations into Big-O intuition.

## 30 Complexity Class Identifier

A Complexity Class Identifier helps you categorize problems and algorithms into broad complexity classes like constant, logarithmic, linear, quadratic, exponential, or polynomial time. It's a way to understand *where your algorithm lives* in the vast map of computational growth.

### What Problem Are We Solving?

When analyzing an algorithm, we often want to know how big its time cost gets as input grows. Instead of exact formulas, we classify algorithms into families based on their asymptotic growth.

This tells us what is *feasible* (polynomial) and what is *explosive* (exponential), guiding both design choices and theoretical limits.

### How It Works (Plain Language)

We map the growth rate of  $T(n)$  to a known complexity class:

| Class         | Example                  | Description                 |
|---------------|--------------------------|-----------------------------|
| $O(1)$        | Hash lookup              | Constant time, no scaling   |
| $O(\log n)$   | Binary search            | Sublinear, halves each step |
| $O(n)$        | Linear scan              | Work grows with input size  |
| $O(n \log n)$ | Merge sort               | Near-linear with log factor |
| $O(n^2)$      | Nested loops             | Quadratic growth            |
| $O(n^3)$      | Matrix multiplication    | Cubic growth                |
| $O(2^n)$      | Backtracking             | Exponential explosion       |
| $O(n!)$       | Brute-force permutations | Factorial blowup            |

Steps to Identify:

1. Analyze loops and recursion structure.
2. Count dominant operations.
3. Match pattern to table above.
4. Verify with recurrence or ratio test.
5. Assign class: constant → logarithmic → polynomial → exponential.

### Example Step by Step

Example 1: Single loop:

```
for i in range(n):
    work()
```

$\rightarrow O(n)$  → Linear

Example 2: Nested loops:

```
for i in range(n):
    for j in range(n):
        work()
```

$\rightarrow O(n^2)$  → Quadratic

Example 3: Divide and conquer:

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

$\rightarrow O(n \log n)$  → Log-linear

Example 4: Brute force subsets:

$2^n$  possibilities

$\rightarrow O(2^n)$  → Exponential

### Tiny Code (Python)

```
def classify_complexity(code_structure):
    if "nested n" in code_structure:
        return "O(n^2)"
    if "divide and conquer" in code_structure:
        return "O(n log n)"
    if "constant" in code_structure:
        return "O(1)"
    return "O(n)"
```

You can extend this to pattern-match pseudocode shapes.

## Why It Matters

- Gives instant intuition about scalability
- Guides design trade-offs (speed vs. simplicity)
- Connects practical code to theoretical limits
- Helps compare algorithms solving the same problem

## A Gentle Proof (Why It Works)

If an algorithm performs  $f(n)$  fundamental operations for input size  $n$ , and  $f(n)$  is asymptotically similar to a known class  $g(n)$ :

$$f(n) = \Theta(g(n))$$

then it belongs to the same class. Classes form equivalence groups under  $\Theta$  notation, simplifying infinite functions into a finite taxonomy.

## Try It Yourself

Classify each:

1.  $T(n) = 5n + 10$
2.  $T(n) = n \log n + 100$
3.  $T(n) = n^3 + 4n^2$
4.  $T(n) = 2^n$

Identify their Big-O class and interpret feasibility.

## Test Cases

| $T(n)$       | Class         | Description |
|--------------|---------------|-------------|
| $7n + 3$     | $O(n)$        | Linear      |
| $3n^2 + 10n$ | $O(n^2)$      | Quadratic   |
| $n \log n$   | $O(n \log n)$ | Log-linear  |
| $2^n$        | $O(2^n)$      | Exponential |
| 100          | $O(1)$        | Constant    |

## Complexity

| Step           | Time   | Space  |
|----------------|--------|--------|
| Classification | $O(1)$ | $O(1)$ |

The Complexity Class Identifier is your map of the algorithmic universe, helping you locate where your code stands, from calm constant time to the roaring infinity of factorial growth.

## Section 4. Algorithm Paradigms

### 31 Greedy Coin Example

The Greedy Coin Example introduces the greedy algorithm paradigm, solving problems by always taking the best immediate option, hoping it leads to a globally optimal solution. In coin change, we repeatedly pick the largest denomination not exceeding the remaining amount.

#### What Problem Are We Solving?

We want to make change for a target amount using the fewest coins possible. A greedy algorithm always chooses the locally optimal coin, the largest denomination remaining total, and repeats until the target is reached.

This method works for canonical coin systems (like U.S. currency) but fails for some arbitrary denominations.

#### How It Works (Plain Language)

1. Sort available coin denominations in descending order.
2. For each coin:
  - Take as many as possible without exceeding the total.
  - Subtract their value from the remaining amount.
3. Continue with smaller coins until the remainder is 0.

Greedy assumes: local optimum  $\rightarrow$  global optimum.

## Example Step by Step

Let coins = {25, 10, 5, 1}, target = 63

| Step | Coin | Count | Remaining |
|------|------|-------|-----------|
| 1    | 25   | 2     | 13        |
| 2    | 10   | 1     | 3         |
| 3    | 5    | 0     | 3         |
| 4    | 1    | 3     | 0         |

Total =  $2 \times 25 + 1 \times 10 + 3 \times 1 = 63$  Coins used = 6

Greedy solution = optimal (U.S. system is canonical).

Counterexample:

Coins = {4, 3, 1}, target = 6

- Greedy:  $4 + 1 + 1 = 3$  coins
- Optimal:  $3 + 3 = 2$  coins

So greedy may fail for non-canonical systems.

## Tiny Code (Python)

```
def greedy_change(coins, amount):
    coins.sort(reverse=True)
    result = []
    for coin in coins:
        while amount >= coin:
            amount -= coin
            result.append(coin)
    return result

print(greedy_change([25,10,5,1], 63)) # [25, 25, 10, 1, 1]
```

## Why It Matters

- Demonstrates local decision-making
- Fast and simple:  $O(n)$  over denominations
- Foundation for greedy design in spanning trees, scheduling, compression
- Highlights where greedy works and where it fails

## A Gentle Proof (Why It Works)

For canonical systems, greedy satisfies the optimal substructure and greedy-choice property:

- Greedy-choice property: Locally best  $\rightarrow$  part of a global optimum.
- Optimal substructure: Remaining subproblem has optimal greedy solution.

Inductively, greedy yields minimal coin count.

## Try It Yourself

1. Try greedy change with  $\{25, 10, 5, 1\}$  for 68.
2. Try  $\{9, 6, 1\}$  for 11, compare with brute force.
3. Identify when greedy fails, test  $\{4, 3, 1\}$ .
4. Extend algorithm to return both coins and count.

## Test Cases

| Coins           | Amount | Result             | Optimal?     |
|-----------------|--------|--------------------|--------------|
| $\{25,10,5,1\}$ | 63     | $[25,25,10,1,1,1]$ |              |
| $\{9,6,1\}$     | 11     | $[9,1,1]$          |              |
| $\{4,3,1\}$     | 6      | $[4,1,1]$          | (3+3 better) |

## Complexity

| Step      | Time          | Space  |
|-----------|---------------|--------|
| Sorting   | $O(k \log k)$ | $O(1)$ |
| Selection | $O(k)$        | $O(k)$ |

( $k$  = number of denominations)

The Greedy Coin Example is the first mirror of the greedy philosophy, simple, intuitive, and fast, a lens into problems where choosing *best now* means *best overall*.

## 32 Greedy Template Simulator

The Greedy Template Simulator shows how every greedy algorithm follows the same pattern, repeatedly choosing the best local option, updating the state, and moving toward the goal. It's a reusable mental and coding framework for designing greedy solutions.

## **What Problem Are We Solving?**

Many optimization problems can be solved by making local choices without revisiting earlier decisions. Instead of searching all paths (like backtracking) or building tables (like DP), greedy algorithms follow a deterministic path of best-next choices.

We want a general template to simulate this structure, useful for scheduling, coin change, and spanning tree problems.

## **How It Works (Plain Language)**

1. Initialize the problem state (remaining value, capacity, etc.).
2. While goal not reached:
  - Evaluate all local choices.
  - Pick the best immediate option (by some criterion).
  - Update the state accordingly.
3. End when no more valid moves exist.

Greedy depends on a selection rule (which local choice is best) and a feasibility check (is the choice valid?).

## **Example Step by Step**

Problem: Job Scheduling by Deadline (Maximize Profit)

| Job | Deadline | Profit |
|-----|----------|--------|
| A   | 2        | 60     |
| B   | 1        | 100    |
| C   | 3        | 20     |
| D   | 2        | 40     |

Steps:

1. Sort jobs by profit (desc): B(100), A(60), D(40), C(20)
2. Take each job if slot deadline available
3. Fill slots:
  - Day 1: B

- Day 2: A
- Day 3: C → Total Profit = 180

Greedy rule: “Pick highest profit first if deadline allows.”

### Tiny Code (Python)

```
def greedy_template(items, is_valid, select_best, update_state):
    state = initialize(items)
    while not goal_reached(state):
        best = select_best(items, state)
        if is_valid(best, state):
            update_state(best, state)
        else:
            break
    return state
```

Concrete greedy solutions just plug in:

- `select_best`: define local criterion
- `is_valid`: define feasibility condition
- `update_state`: modify problem state

### Why It Matters

- Reveals shared skeleton behind all greedy algorithms
- Simplifies learning, “different bodies, same bones”
- Encourages reusable code via template-based design
- Helps debug logic: if it fails, test greedy-choice property

### A Gentle Proof (Why It Works)

If a problem has:

- Greedy-choice property: local best is part of global best
- Optimal substructure: subproblem solutions are optimal

Then any algorithm following this template produces a global optimum. Formally proved via induction on input size.

## Try It Yourself

1. Implement template for:
  - Coin change
  - Fractional knapsack
  - Interval scheduling
2. Compare with brute-force or DP to confirm optimality.
3. Identify when greedy fails (e.g., non-canonical coin sets).

## Test Cases

| Problem                 | Local Rule             | Works? | Note             |
|-------------------------|------------------------|--------|------------------|
| Fractional Knapsack     | Max value/weight       |        | Continuous       |
| Interval Scheduling     | Earliest finish        |        | Non-overlapping  |
| Coin Change (25,10,5,1) | Largest coin remaining |        | Canonical only   |
| Job Scheduling          | Highest profit first   |        | Sorted by profit |

## Complexity

| Step      | Time                 | Space  |
|-----------|----------------------|--------|
| Selection | $O(n \log n)$ (sort) | $O(n)$ |
| Iteration | $O(n)$               | $O(1)$ |

The Greedy Template Simulator is the skeleton key of greedy design, once you learn its shape, every greedy algorithm looks like a familiar face.

## 33 Divide & Conquer Skeleton

The Divide & Conquer Skeleton captures the universal structure of algorithms that solve big problems by splitting them into smaller, independent pieces, solving each recursively, then combining their results. It's the framework behind mergesort, quicksort, binary search, and more.

## What Problem Are We Solving?

Some problems are too large or complex to handle at once. Divide & Conquer (D&C) solves them by splitting into smaller subproblems of the same type, solving recursively, and combining the results into a whole.

We want a reusable template that reveals this recursive rhythm.

## How It Works (Plain Language)

Every D&C algorithm follows this triplet:

1. Divide: Break the problem into smaller subproblems.
2. Conquer: Solve each subproblem (often recursively).
3. Combine: Merge or assemble partial solutions.

This recursion continues until a base case (small enough to solve directly).

General Recurrence:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

- $a$ : number of subproblems
- $b$ : factor by which size is reduced
- $f(n)$ : cost to divide/combine

## Example Step by Step

Example: Merge Sort

1. Divide: Split array into two halves
2. Conquer: Recursively sort each half
3. Combine: Merge two sorted halves into one

For  $n = 8$ :

- Level 0: size 8
- Level 1: size  $4 + 4$
- Level 2: size  $2 + 2 + 2 + 2$
- Level 3: size 1 (base case)

Each level costs  $O(n)$  → total  $O(n \log n)$ .

## Tiny Code (Python)

```
def divide_and_conquer(problem, base_case, divide, combine):
    if base_case(problem):
        return solve_directly(problem)
    subproblems = divide(problem)
    solutions = [divide_and_conquer(p, base_case, divide, combine) for p in subproblems]
    return combine(solutions)
```

Plug in custom `divide`, `combine`, and base-case logic for different problems.

## Why It Matters

- Models recursive structure of many core algorithms
- Reveals asymptotic pattern via recurrence
- Enables parallelization (subproblems solved independently)
- Balances simplicity (small subproblems) with power (reduction)

## A Gentle Proof (Why It Works)

If each recursive level divides the work evenly and recombines in finite time, then total cost is sum of all level costs:

$$T(n) = \sum_{i=0}^{\log_b n} a^i \cdot f! \left( \frac{n}{b^i} \right)$$

Master Theorem or tree expansion shows convergence to  $O(n^{\log_b a})$  or  $O(n \log n)$ , depending on  $f(n)$ .

Correctness follows by induction: each subproblem solved optimally combined result optimal.

## Try It Yourself

1. Write a D&C template for:
  - Binary Search
  - Merge Sort
  - Karatsuba Multiplication
2. Identify  $a$ ,  $b$ ,  $f(n)$  for each.
3. Solve their recurrences with Master Theorem.

## Test Cases

| Algorithm     | a | b | $f(n)$         | Complexity        |
|---------------|---|---|----------------|-------------------|
| Binary Search | 1 | 2 | 1              | $O(\log n)$       |
| Merge Sort    | 2 | 2 | $n$            | $O(n \log n)$     |
| Quick Sort    | 2 | 2 | $n$ (expected) | $O(n \log n)$     |
| Karatsuba     | 3 | 2 | $n$            | $O(n^{\log_2 3})$ |

## Complexity

| Step            | Time                    | Space               |
|-----------------|-------------------------|---------------------|
| Recursive calls | $O(n)$ to $O(n \log n)$ | $O(\log n)$ (stack) |
| Combine         | $O(f(n))$               | depends on merging  |

The Divide & Conquer Skeleton is the heartbeat of recursion, a rhythm of divide, solve, combine, pulsing through the core of algorithmic design.

## 34 Backtracking Maze Solver

The Backtracking Maze Solver illustrates the backtracking paradigm, exploring all possible paths through a search space, stepping forward when valid, and undoing moves when a dead end is reached. It's the classic model for recursive search and constraint satisfaction.

### What Problem Are We Solving?

We want to find a path from start to goal in a maze or search space filled with constraints. Brute force would try every path blindly; backtracking improves on this by pruning paths as soon as they become invalid.

This approach powers solvers for mazes, Sudoku, N-Queens, and combinatorial search problems.

## How It Works (Plain Language)

1. Start at the initial position.
2. Try a move (north, south, east, west).
3. If move is valid, mark position and recurse from there.
4. If stuck, backtrack: undo last move and try a new one.
5. Stop when goal is reached or all paths are explored.

The algorithm is depth-first in nature, it explores one branch fully before returning.

## Example Step by Step

Maze (Grid Example)

```
S . . #
# . # .
. . . G
```

- Start at S (0,0), Goal at G (2,3)
- Move right, down, or around obstacles (#)
- Mark visited cells
- When trapped, step back and try another path

Path Found: S → (0,1) → (1,1) → (2,1) → (2,2) → G

## Tiny Code (Python)

```
def solve_maze(maze, x, y, goal):
    if (x, y) == goal:
        return True
    if not valid_move(maze, x, y):
        return False
    maze[x][y] = 'V' # Mark visited
    for dx, dy in [(0,1), (1,0), (0,-1), (-1,0)]:
        if solve_maze(maze, x+dx, y+dy, goal):
            return True
    maze[x][y] = '.' # Backtrack
    return False
```

The recursion explores all paths, marking and unmarking as it goes.

## Why It Matters

- Demonstrates search with undoing
- Foundational for DFS, constraint satisfaction, puzzle solving
- Illustrates state exploration and recursive pruning
- Framework for N-Queens, Sudoku, graph coloring

## A Gentle Proof (Why It Works)

By exploring all valid moves recursively:

- Every feasible path is eventually checked.
- Infeasible branches terminate early due to validity checks.
- Backtracking guarantees all combinations are explored once.

Thus, completeness is ensured, and if a path exists, it will be found.

## Try It Yourself

1. Draw a  $4 \times 4$  maze with one solution.
2. Run backtracking manually, marking path and undoing wrong turns.
3. Modify rules (e.g., diagonal moves allowed).
4. Compare runtime with BFS (which finds shortest path).

## Test Cases

| Maze            | Solution Found | Notes                 |
|-----------------|----------------|-----------------------|
| Open grid       | Yes            | Path straight to goal |
| Maze with block | Yes            | Backs up and reroutes |
| No path         | No             | Exhausts all options  |

## Complexity

| Step   | Time                | Space                  |
|--------|---------------------|------------------------|
| Search | $O(4^n)$ worst-case | $O(n)$ recursion stack |

( $n$  = number of cells)

Pruning and constraints reduce practical cost.

The Backtracking Maze Solver is a journey of trial and error, a guided wanderer exploring paths, retreating gracefully, and finding solutions hidden in the labyrinth.

## 35 Karatsuba Multiplication

The Karatsuba Multiplication algorithm is a divide-and-conquer technique that multiplies two large numbers faster than the classical grade-school method. It reduces the multiplication count from 4 to 3 per recursive step, improving complexity from  $O(n^2)$  to approximately  $O(n^{1.5})$ .

### What Problem Are We Solving?

When multiplying large numbers (or polynomials), the standard approach performs every pairwise digit multiplication,  $O(n^2)$  work for  $n$ -digit numbers. Karatsuba observed that some of this work is redundant. By reusing partial results cleverly, we can cut down the number of multiplications and gain speed.

This is the foundation of many fast arithmetic algorithms and symbolic computation libraries.

### How It Works (Plain Language)

Given two  $n$ -digit numbers:

$$x = 10^m \cdot a + b \quad y = 10^m \cdot c + d$$

where  $(a, b, c, d)$  are roughly  $n/2$ -digit halves of  $x$  and  $y$ .

1. Compute three products:

- $(p_1 = a \cdot c)$
- $(p_2 = b \cdot d)$
- $(p_3 = (a + b)(c + d))$

2. Combine results using:

$$x \cdot y = 10^{2m} \cdot p_1 + 10^m \cdot (p_3 - p_1 - p_2) + p_2$$

This reduces recursive multiplications from 4 to 3.

## Example Step by Step

Multiply  $12 \times 34$ .

Split:

- $a = 1, b = 2$
- $c = 3, d = 4$

Compute:

- $(p_1 = 1 \times 3 = 3)$
- $(p_2 = 2 \times 4 = 8)$
- $(p_3 = (1 + 2)(3 + 4) = 3 \times 7 = 21)$

Combine:

$$(10^2) \cdot 3 + 10 \cdot (21 - 3 - 8) + 8 = 300 + 100 + 8 = 408$$

So  $12 \times 34 = 408$  (correct).

## Tiny Code (Python)

```
def karatsuba(x, y):  
    if x < 10 or y < 10:  
        return x * y  
    n = max(len(str(x)), len(str(y)))  
    m = n // 2  
    a, b = divmod(x, 10m)  
    c, d = divmod(y, 10m)  
    p1 = karatsuba(a, c)  
    p2 = karatsuba(b, d)  
    p3 = karatsuba(a + b, c + d)  
    return p1 * 10(2*m) + (p3 - p1 - p2) * 10m + p2
```

## Why It Matters

- First sub-quadratic multiplication algorithm
- Basis for advanced methods (Toom–Cook, FFT-based)
- Applies to integers, polynomials, big-number arithmetic
- Showcases power of divide and conquer

## A Gentle Proof (Why It Works)

The product expansion is:

$$(a \cdot 10^m + b)(c \cdot 10^m + d) = ac \cdot 10^{2m} + (ad + bc)10^m + bd$$

Observe:

$$(a + b)(c + d) = ac + ad + bc + bd$$

Thus:

$$ad + bc = (a + b)(c + d) - ac - bd$$

Karatsuba leverages this identity to compute ( ad + bc ) without a separate multiplication.

Recurrence:

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

Solution:  $T(n) = O(n^{\log_2 3}) \approx O(n^{1.585})$

## Try It Yourself

1. Multiply  $1234 \times 5678$  using Karatsuba steps.
2. Compare with grade-school multiplication count.
3. Visualize recursive calls as a tree.
4. Derive recurrence and verify complexity.

## Test Cases

| x    | y    | Result   | Method |
|------|------|----------|--------|
| 12   | 34   | 408      | Works  |
| 123  | 456  | 56088    | Works  |
| 9999 | 9999 | 99980001 | Works  |

## Complexity

| Step           | Time                 | Space  |
|----------------|----------------------|--------|
| Multiplication | $O(n^1 \cdot \dots)$ | $O(n)$ |
| Base case      | $O(1)$               | $O(1)$ |

Karatsuba Multiplication reveals the magic of algebraic rearrangement, using one clever identity to turn brute-force arithmetic into an elegant, faster divide-and-conquer dance.

## 36 DP State Diagram Example

The DP State Diagram Example introduces the idea of representing dynamic programming (DP) problems as graphs of states connected by transitions. It's a visual and structural way to reason about overlapping subproblems, dependencies, and recurrence relations.

### What Problem Are We Solving?

Dynamic programming problems often involve a set of subproblems that depend on one another. Without a clear mental model, it's easy to lose track of which states rely on which others.

A state diagram helps us:

- Visualize states as nodes
- Show transitions as directed edges
- Understand dependency order for iteration or recursion

This builds intuition for state definition, transition logic, and evaluation order.

### How It Works (Plain Language)

1. Define the state, what parameters represent a subproblem (e.g., index, capacity, sum).
2. Draw each state as a node.
3. Add edges to show transitions between states.
4. Assign recurrence along edges:

$$dp[\text{state}] = \text{combine}(dp[\text{previous states}])$$

5. Solve by topological order (bottom-up) or memoized recursion (top-down).

## Example Step by Step

Example: Fibonacci Sequence

$$F(n) = F(n - 1) + F(n - 2)$$

State diagram:

F(5)

F(4) F(3)

F(3)F(2)F(2)F(1)

Each node = state  $F(k)$  Edges = dependencies on  $F(k-1)$  and  $F(k-2)$

Observation: Many states repeat, shared subproblems suggest memoization or bottom-up DP.

Another Example: 0/1 Knapsack

State:  $dp[i][w]$  = max value using first  $i$  items, capacity  $w$ . Transitions:

- Include item  $i$ :  $dp[i-1][w-weight[i]] + value[i]$
- Exclude item  $i$ :  $dp[i-1][w]$

Diagram: a grid of states, each cell connected from previous row and shifted left.

## Tiny Code (Python)

```
def fib_dp(n):
    dp = [0, 1]
    for i in range(2, n + 1):
        dp.append(dp[i-1] + dp[i-2])
    return dp[n]
```

Each entry  $dp[i]$  represents a state, filled based on prior dependencies.

## Why It Matters

- Makes DP visual and tangible
- Clarifies dependency direction (acyclic structure)
- Ensures correct order of computation
- Serves as blueprint for bottom-up or memoized implementation

## A Gentle Proof (Why It Works)

If a problem's structure can be represented as a DAG of states, and:

- Every state's value depends only on earlier states
- Base states are known

Then by evaluating nodes in topological order, we guarantee correctness, each subproblem is solved after its dependencies.

This matches mathematical induction over recurrence depth.

## Try It Yourself

1. Draw state diagram for Fibonacci.
2. Draw grid for 0/1 Knapsack (rows = items, cols = capacity).
3. Visualize transitions for Coin Change (ways to make sum).
4. Trace evaluation order bottom-up.

## Test Cases

| Problem     | State      | Transition                   | Diagram Shape |
|-------------|------------|------------------------------|---------------|
| Fibonacci   | $dp[i]$    | $dp[i-1] + dp[i-2]$          | Chain         |
| Knapsack    | $dp[i][w]$ | max(include, exclude)        | Grid          |
| Coin Change | $dp[i][s]$ | $dp[i-1][s] + dp[i][s-coin]$ | Lattice       |

## Complexity

| Step                 | Time                   | Space          |
|----------------------|------------------------|----------------|
| Diagram construction | $O(n^2)$ (visual)      | $O(n^2)$       |
| DP evaluation        | $O(n \cdot m)$ typical | $O(n \cdot m)$ |

The DP State Diagram turns abstract recurrences into maps of reasoning, every arrow a dependency, every node a solved step, guiding you from base cases to the final solution.

## 37 DP Table Visualization

The DP Table Visualization is a way to make dynamic programming tangible, turning states and transitions into a clear table you can fill, row by row or column by column. Each cell represents a subproblem, and the process of filling it shows the algorithm's structure.

### What Problem Are We Solving?

Dynamic programming can feel abstract when written as recurrences. A table transforms that abstraction into something concrete:

- Rows and columns correspond to subproblem parameters
- Cell values show computed solutions
- Filling order reveals dependencies

This approach is especially powerful for tabulation (bottom-up DP).

### How It Works (Plain Language)

1. Define your DP state (e.g.,  $dp[i][j] =$  best value up to item i and capacity j).
2. Initialize base cases (first row/column).
3. Iterate through the table in dependency order.
4. Apply recurrence at each cell:

$$dp[i][j] = \text{combine}(dp[i - 1][j], dp[i - 1][j - w_i] + v_i)$$

5. Final cell gives the answer (often bottom-right).

## Example Step by Step

Example: 0/1 Knapsack

Items:

| Item | Weight | Value |
|------|--------|-------|
| 1    | 1      | 15    |
| 2    | 3      | 20    |
| 3    | 4      | 30    |

Capacity = 4

State:  $dp[i][w]$  = max value with first  $i$  items, capacity  $w$ .

Recurrence:

$$dp[i][w] = \max(dp[i-1][w], dp[i-1][w - w_i] + v_i)$$

DP Table:

| $i/w$ | 0 | 1  | 2  | 3  | 4  |
|-------|---|----|----|----|----|
| 0     | 0 | 0  | 0  | 0  | 0  |
| 1     | 0 | 15 | 15 | 15 | 15 |
| 2     | 0 | 15 | 15 | 20 | 35 |
| 3     | 0 | 15 | 15 | 20 | 35 |

Final answer: 35 (items 1 and 2)

## Tiny Code (Python)

```
def knapsack(weights, values, W):
    n = len(weights)
    dp = [[0]*(W+1) for _ in range(n+1)]
    for i in range(1, n+1):
        for w in range(W+1):
            if weights[i-1] <= w:
                dp[i][w] = max(dp[i-1][w],
                                dp[i-1][w-weights[i-1]] + values[i-1])
            else:
                dp[i][w] = dp[i-1][w]
    return dp
```

Each  $dp[i][w]$  is one table cell, filled in increasing order of  $i$  and  $w$ .

## Why It Matters

- Turns recurrence into geometry
- Makes dependencies visible and traceable
- Clarifies filling order (row-wise, diagonal, etc.)
- Serves as debugging tool and teaching aid

## A Gentle Proof (Why It Works)

The table order ensures every subproblem is solved after its dependencies. By induction:

- Base row/column initialized correctly
- Each cell built from valid earlier states
- Final cell accumulates optimal solution

This is equivalent to a topological sort on the DP dependency graph.

## Try It Yourself

1. Draw the DP table for Coin Change (number of ways).
2. Fill row by row.
3. Trace dependencies with arrows.
4. Mark the path that contributes to the final answer.

## Test Cases

| Problem       | State      | Fill Order | Output     |
|---------------|------------|------------|------------|
| Knapsack      | $dp[i][w]$ | Row-wise   | Max value  |
| LCS           | $dp[i][j]$ | Diagonal   | LCS length |
| Edit Distance | $dp[i][j]$ | Row/col    | Min ops    |

## Complexity

| Step                 | Time           | Space          |
|----------------------|----------------|----------------|
| Filling table        | $O(n \cdot m)$ | $O(n \cdot m)$ |
| Traceback (optional) | $O(n+m)$       | $O(1)$         |

| Step | Time | Space |
|------|------|-------|
|------|------|-------|

The DP Table Visualization is the grid view of recursion, a landscape of subproblems, each solved once, all leading toward the final cell that encodes the complete solution.

## 38 Recursive Subproblem Tree Demo

The Recursive Subproblem Tree Demo shows how a dynamic programming problem expands into a tree of subproblems. It visualizes recursion structure, repeated calls, and where memoization or tabulation can save redundant work.

### What Problem Are We Solving?

When writing a recursive solution, the same subproblems are often solved multiple times. Without visualizing this, we may not realize how much overlap occurs.

By drawing the recursion as a subproblem tree, we can:

- Identify repeated nodes (duplicate work)
- Understand recursion depth
- Decide between memoization (top-down) or tabulation (bottom-up)

### How It Works (Plain Language)

1. Start from the root: the full problem (e.g.,  $F(n)$ ).
2. Expand recursively into smaller subproblems (children).
3. Continue until base cases (leaves).
4. Observe repeated nodes (same subproblem appearing multiple times).
5. Replace repeated computations with a lookup in a table.

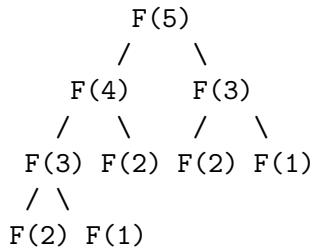
The resulting structure is a tree that becomes a DAG after memoization.

### Example Step by Step

Example: Fibonacci (Naive Recursive)

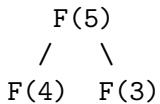
$$F(n) = F(n - 1) + F(n - 2)$$

For  $n = 5$ :



Repeated nodes:  $F(3)$ ,  $F(2)$  Memoization would store these results and reuse them.

With Memoization (Tree Collapsed):



Each node computed once, repeated calls replaced by cache lookups.

### Tiny Code (Python)

```

def fib(n, memo=None):
    if memo is None:
        memo = {}
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fib(n-1, memo) + fib(n-2, memo)
    return memo[n]

```

The memo dictionary turns the recursion tree into a DAG.

### Why It Matters

- Exposes hidden redundancy in recursive algorithms
- Motivates memoization (cache results)
- Shows connection between recursion and iteration
- Visual tool for understanding time complexity

## A Gentle Proof (Why It Works)

Let  $T(n)$  be the recursion tree size.

Naive recursion for Fibonacci:

$$T(n) = T(n - 1) + T(n - 2) + 1$$

$O(2^n)$  calls

With memoization, each subproblem computed once:

$$T(n) = O(n)$$

Proof by induction:

- Base case  $n = 1$ : trivial
- Inductive step: if all smaller values memoized, reuse ensures constant-time lookups per state

## Try It Yourself

1. Draw the recursion tree for Fibonacci(6).
2. Count repeated nodes.
3. Add a memo table and redraw as DAG.
4. Apply same technique to factorial or grid path problems.

## Test Cases

| Function | Naive Calls | Memoized Calls | Time                         |
|----------|-------------|----------------|------------------------------|
| fib(5)   | 15          | 6              | $O(2^5) \rightarrow O(n)$    |
| fib(10)  | 177         | 11             | $O(2^{10}) \rightarrow O(n)$ |
| fib(20)  | 21891       | 21             | $O(2^{20}) \rightarrow O(n)$ |

## Complexity

| Step             | Time     | Space  |
|------------------|----------|--------|
| Naive recursion  | $O(2^n)$ | $O(n)$ |
| With memoization | $O(n)$   | $O(n)$ |

The Recursive Subproblem Tree Demo turns hidden recursion into a picture, every branch a computation, every repeated node a chance to save time, and every cache entry a step toward efficiency.

## 39 Greedy Choice Visualization

The Greedy Choice Visualization helps you see how greedy algorithms make decisions step by step, choosing the locally optimal option at each point and committing to it. By tracing choices visually, you can verify whether the greedy strategy truly leads to a global optimum.

### What Problem Are We Solving?

A greedy algorithm always chooses the best immediate option. But not every problem supports this approach, some require backtracking or DP. To know when greediness works, we need to see the chain of choices and their effects.

A greedy choice diagram reveals:

- What each local decision looks like
- How each choice affects remaining subproblems
- Whether local optima accumulate into a global optimum

### How It Works (Plain Language)

1. Start with the full problem (e.g., a set of intervals, coins, or items).
2. Sort or prioritize by a greedy criterion (e.g., largest value, earliest finish).
3. Pick the best option currently available.
4. Eliminate incompatible elements (conflicts, overlaps).
5. Repeat until no valid choices remain.
6. Visualize each step as a growing path or sequence.

The resulting picture shows a selection frontier, how choices narrow possibilities.

### Example Step by Step

Example 1: Interval Scheduling

Goal: select max non-overlapping intervals.

| Interval | Start | End |
|----------|-------|-----|
| A        | 1     | 4   |

|   | Interval | Start | End |
|---|----------|-------|-----|
| B | 3        | 5     |     |
| C | 0        | 6     |     |
| D | 5        | 7     |     |
| E | 8        | 9     |     |

Greedy Rule: Choose earliest finish time.

Steps:

1. Sort by finish → A(1–4), B(3–5), C(0–6), D(5–7), E(8–9)
2. Pick A → remove overlaps (B, C)
3. Next pick D (5–7)
4. Next pick E (8–9)

Visualization:

Timeline: 0---1---3---4---5---7---8---9  
 [A]            [D]            [E]

Total = 3 intervals → optimal.

Example 2: Fractional Knapsack

| Item | Value | Weight | Ratio |
|------|-------|--------|-------|
| 1    | 60    | 10     | 6     |
| 2    | 100   | 20     | 5     |
| 3    | 120   | 30     | 4     |

Greedy Rule: Max value/weight ratio Visualization: pick items in decreasing ratio order → 1, 2, part of 3.

### Tiny Code (Python)

```
def greedy_choice(items, key):
    items = sorted(items, key=key, reverse=True)
    chosen = []
    for it in items:
        if valid(it, chosen):
            chosen.append(it)
    return chosen
```

By logging or plotting at each iteration, you can visualize how the solution grows.

## Why It Matters

- Shows local vs global tradeoffs visually
- Confirms greedy-choice property (local best = globally best)
- Helps diagnose greedy failures (where path deviates from optimum)
- Strengthens understanding of problem structure

## A Gentle Proof (Why It Works)

A greedy algorithm works if:

1. Greedy-choice property: local best can lead to global best.
2. Optimal substructure: optimal solution of whole contains optimal solutions of parts.

Visualization helps verify these conditions, if each chosen step leaves a smaller problem that is still optimally solvable by the same rule, the algorithm is correct.

## Try It Yourself

1. Draw intervals and apply earliest-finish greedy rule.
2. Visualize coin selections for greedy coin change.
3. Try a counterexample where greedy fails (e.g., coin set {4,3,1}).
4. Plot selection order to see divergence from optimum.

## Test Cases

| Problem                 | Greedy Rule            | Works? | Note                 |
|-------------------------|------------------------|--------|----------------------|
| Interval Scheduling     | Earliest finish        | Yes    | Optimal              |
| Fractional Knapsack     | Max ratio              | Yes    | Continuous fractions |
| Coin Change (25,10,5,1) | Largest coin remaining | Yes    | Canonical            |
| Coin Change (4,3,1)     | Largest coin remaining | No     | Not canonical        |

## Complexity

| Step           | Time          | Space  |
|----------------|---------------|--------|
| Sort elements  | $O(n \log n)$ | $O(n)$ |
| Selection loop | $O(n)$        | $O(1)$ |

The Greedy Choice Visualization transforms abstract decision logic into a picture, a timeline or path that shows exactly how local choices unfold into (or away from) the global goal.

## 40 Amortized Merge Demo

The Amortized Merge Demo illustrates how expensive operations can appear cheap when averaged over a long sequence. By analyzing total cost across all steps, we reveal why some algorithms with occasional heavy work still run efficiently overall.

### What Problem Are We Solving?

Some data structures or algorithms perform occasional costly operations (like merging arrays, resizing tables, or rebuilding heaps). If we only look at worst-case time per step, they seem inefficient, but amortized analysis shows that, over many operations, the *average* cost per operation stays low.

This method explains why dynamic arrays, union-find, and incremental merges remain efficient.

### How It Works (Plain Language)

1. Perform a sequence of operations ( $O_1, O_2, \dots, O_n$ ).
2. Some are cheap (constant time), some are expensive (linear or log).
3. Compute total cost over all ( $n$ ) operations.
4. Divide total by ( $n$ )  $\rightarrow$  amortized cost per operation.

Amortized analysis tells us:

$$\text{Amortized cost} = \frac{\text{Total cost over sequence}}{n}$$

Even if a few operations are expensive, their cost is “spread out” across many cheap ones.

## Example Step by Step

Example: Dynamic Array Doubling

Suppose we double the array each time it's full.

| Operation   | Capacity | Actual Cost | Total Elements | Cumulative Cost |
|-------------|----------|-------------|----------------|-----------------|
| Insert 1–1  | 1        | 1           | 1              | 1               |
| Insert 2–2  | 2        | 2           | 2              | 3               |
| Insert 3–4  | 4        | 3           | 3              | 6               |
| Insert 4–4  | 4        | 1           | 4              | 7               |
| Insert 5–8  | 8        | 5           | 5              | 12              |
| Insert 6–8  | 8        | 1           | 6              | 13              |
| Insert 7–8  | 8        | 1           | 7              | 14              |
| Insert 8–8  | 8        | 1           | 8              | 15              |
| Insert 9–16 | 16       | 9           | 9              | 24              |

Total cost (for 9 inserts) = 24 Amortized cost =  $24 / 9 = 2.67 = O(1)$

So although some inserts cost  $O(n)$ , the average cost per insert =  $O(1)$ .

Example: Amortized Merge in Union-Find

When combining sets, always attach the smaller tree to the larger one. Each element's depth increases at most  $O(\log n)$  times  $\rightarrow$  total cost  $O(n \log n)$ .

## Tiny Code (Python)

```
def dynamic_array_append(arr, x, capacity):
    if len(arr) == capacity:
        capacity *= 2 # double size
        arr.extend([None]*(capacity - len(arr))) # copy cost = len(arr)
    arr[len(arr)//2] = x
    return arr, capacity
```

This simulates doubling capacity, where copy cost = current array size.

## Why It Matters

- Explains hidden efficiency behind resizing structures
- Shows why occasional spikes don't ruin performance
- Foundation for analyzing stacks, queues, hash tables
- Builds intuition for amortized  $O(1)$  operations

## A Gentle Proof (Why It Works)

Consider dynamic array resizing:

- Every element gets moved at most once per doubling.
- Over  $n$  insertions, total copies  $2n$ .

Thus,

$$\text{Total cost} = O(n) \implies \text{Amortized cost} = O(1)$$

This uses the aggregate method of amortized analysis:

$$\text{Amortized cost per operation} = \frac{\text{total work}}{\# \text{ operations}}$$

## Try It Yourself

1. Simulate 10 inserts into a doubling array.
2. Track total copy cost.
3. Plot actual vs amortized cost.
4. Repeat with tripling growth factor, compare average cost.

## Test Cases

| Operation Type   | Cost Model               | Amortized Cost | Notes               |
|------------------|--------------------------|----------------|---------------------|
| Array Doubling   | Copy + Insert            | $O(1)$         | Spread cost         |
| Union-Find Merge | Attach smaller to larger | $O(\log n)$    | = inverse Ackermann |
| Stack Push       | Resize occasionally      | $O(1)$         | Average constant    |
| Queue Enqueue    | Circular buffer          | $O(1)$         | Rotational reuse    |

## Complexity

| Step          | Worst Case | Amortized | Space  |
|---------------|------------|-----------|--------|
| Single Insert | $O(n)$     | $O(1)$    | $O(n)$ |
| $n$ Inserts   | $O(n)$     | $O(n)$    | $O(n)$ |

The Amortized Merge Demo reveals the calm beneath algorithmic chaos, even when some steps are costly, the long-run rhythm stays smooth, predictable, and efficient.

## Section 5. Recurrence Relations

### 41 Linear Recurrence Solver

A Linear Recurrence Solver finds closed-form or iterative solutions for sequences defined in terms of previous values. It transforms recursive definitions like  $T(n) = aT(n - 1) + b$  into explicit formulas, helping us understand algorithmic growth.

#### What Problem Are We Solving?

Many algorithms, especially recursive ones, define running time through a recurrence relation, for example:

$$T(n) = a, T(n - 1) + b$$

To reason about complexity or compute exact values, we want to solve the recurrence, converting it from a self-referential definition into a direct expression in  $n$ .

This solver provides a methodical way to do that.

#### How It Works (Plain Language)

A linear recurrence has the general form:

$$T(n) = a_1T(n - 1) + a_2T(n - 2) + \dots + a_kT(n - k) + f(n)$$

1. Identify coefficients ( $a_1, a_2, \dots$ ).
2. Write the characteristic equation for the homogeneous part.
3. Solve for roots ( $r_1, r_2, \dots$ ).
4. Form the homogeneous solution using those roots.
5. Add a particular solution if  $f(n)$  is non-zero.
6. Apply initial conditions to fix constants.

## Example Step by Step

Example 1:

$$T(n) = 2T(n-1) + 3, \quad T(0) = 1$$

1. Homogeneous part:  $T(n) - 2T(n-1) = 0 \rightarrow$  Characteristic root:  $r = 2 \rightarrow$  Homogeneous solution:  $T_h(n) = C \cdot 2^n$
2. Particular solution: constant  $p$  Plug in:  $p = 2p + 3 \Rightarrow p = -3$
3. General solution:

$$T(n) = C \cdot 2^n - 3$$

4. Apply  $T(0) = 1$ :  $1 = C - 3 \Rightarrow C = 4$

Final:

$$T(n) = 4 \cdot 2^n - 3$$

Example 2 (Fibonacci):

$$F(n) = F(n-1) + F(n-2), \quad F(0) = 0, F(1) = 1$$

Characteristic equation:

$$r^2 - r - 1 = 0$$

Roots:

$$r_1 = \frac{1 + \sqrt{5}}{2}, \quad r_2 = \frac{1 - \sqrt{5}}{2}$$

General solution:

$$F(n) = Ar_1^n + Br_2^n$$

Solving constants yields Binet's Formula:

$$F(n) = \frac{1}{\sqrt{5}} \left[ \left( \frac{1 + \sqrt{5}}{2} \right)^n - \left( \frac{1 - \sqrt{5}}{2} \right)^n \right]$$

## Tiny Code (Python)

```
def linear_recurrence(a, b, n, t0):
    T = [t0]
    for i in range(1, n + 1):
        T.append(a * T[i - 1] + b)
    return T
```

This simulates a simple first-order recurrence like  $T(n) = aT(n - 1) + b$ .

## Why It Matters

- Converts recursive definitions into explicit formulas
- Helps analyze time complexity for recursive algorithms
- Bridges math and algorithm design
- Used in DP transitions, counting problems, and algorithm analysis

## A Gentle Proof (Why It Works)

Unroll  $T(n) = aT(n - 1) + b$ :

$$T(n) = a^n T(0) + b(a^{n-1} + a^{n-2} + \dots + 1)$$

Sum is geometric:

$$T(n) = a^n T(0) + b \frac{a^n - 1}{a - 1}$$

Hence the closed form is:

$$T(n) = a^n T(0) + \frac{b(a^n - 1)}{a - 1}$$

This matches the method of characteristic equations for constant coefficients.

## Try It Yourself

1. Solve  $T(n) = 3T(n - 1) + 2$ ,  $T(0) = 1$
2. Solve  $T(n) = 2T(n - 1) - T(n - 2)$
3. Compare numeric results with iterative simulation
4. Draw recursion tree to confirm growth trend

## Test Cases

| Recurrence             | Initial    | Solution          | Growth   |
|------------------------|------------|-------------------|----------|
| $T(n) = 2T(n - 1) + 3$ | $T(0) = 1$ | $4 \cdot 2^n - 3$ | $O(2^n)$ |
| $T(n) = T(n - 1) + 1$  | $T(0) = 0$ | $n$               | $O(n)$   |
| $T(n) = 3T(n - 1)$     | $T(0) = 1$ | $3^n$             | $O(3^n)$ |

## Complexity

| Method               | Time   | Space  |
|----------------------|--------|--------|
| Recursive (unrolled) | $O(n)$ | $O(n)$ |
| Closed-form          | $O(1)$ | $O(1)$ |

A Linear Recurrence Solver turns repeated dependence into explicit growth, revealing the hidden pattern behind each recursive step.

## 42 Master Theorem

The Master Theorem provides a direct method to analyze divide-and-conquer recurrences, allowing you to find asymptotic bounds without expanding or guessing. It is a cornerstone tool for understanding recursive algorithms such as merge sort, binary search, and Strassen's multiplication.

### What Problem Are We Solving?

Many recursive algorithms can be expressed as:

$$T(n) = a, T! \left( \frac{n}{b} \right) + f(n)$$

where:

- $a$ : number of subproblems
- $b$ : shrink factor (problem size per subproblem)
- $f(n)$ : additional work outside recursion (combine, partition, etc.)

We want to find an asymptotic expression for  $T(n)$  by comparing recursive cost ( $n^{\log_b a}$ ) with non-recursive cost ( $f(n)$ ).

## How It Works (Plain Language)

1. Write the recurrence in standard form:

$$T(n) = a, T(n/b) + f(n)$$

2. Compute the critical exponent  $\log_b a$ .
3. Compare  $f(n)$  with  $n^{\log_b a}$ :
  - If  $f(n)$  is smaller, recursion dominates.
  - If they are equal, both contribute equally.
  - If  $f(n)$  is larger, the outside work dominates.

The theorem gives three standard cases depending on which term grows faster.

## The Three Cases

Case 1 (Recursive Work Dominates):

If

$$f(n) = O(n^{\log_b a - \varepsilon})$$

for some  $\varepsilon > 0$ , then

$$T(n) = \Theta(n^{\log_b a})$$

Case 2 (Balanced Work):

If

$$f(n) = \Theta(n^{\log_b a})$$

then

$$T(n) = \Theta(n^{\log_b a} \log n)$$

Case 3 (Non-Recursive Work Dominates):

If

$$f(n) = \Omega(n^{\log_b a + \varepsilon})$$

and

$$a, f(n/b) \leq c, f(n)$$

for some constant  $c < 1$ , then

$$T(n) = \Theta(f(n))$$

## Example Step by Step

Example 1: Merge Sort

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

- $a = 2, b = 2$ , so  $\log_b a = 1$
- $f(n) = O(n)$
- $f(n) = \Theta(n^{\log_b a}) \rightarrow$  Case 2

Result:

$$T(n) = \Theta(n \log n)$$

Example 2: Binary Search

$$T(n) = T(n/2) + O(1)$$

- $a = 1, b = 2$ , so  $\log_b a = 0$
- $f(n) = O(1) = \Theta(n^0) \rightarrow$  Case 2

Result:

$$T(n) = \Theta(\log n)$$

Example 3: Strassen's Matrix Multiplication

$$T(n) = 7T(n/2) + O(n^2)$$

- $a = 7, b = 2$ , so  $\log_2 7 \approx 2.81$
- $f(n) = O(n^2) = O(n^{2.81-\varepsilon}) \rightarrow$  Case 1

Result:

$$T(n) = \Theta(n^{\log_2 7})$$

## Tiny Code (Python)

```
import math

def master_theorem(a, b, f_exp):
    log_term = math.log(a, b)
    if f_exp < log_term:
        return f"Theta(n^{round(log_term, 2)})"
```

```

    elif abs(f_exp - log_term) < 1e-9:
        return f"Theta(n^{round(log_term, 2)} * log n)"
    else:
        return f"Theta(n^{f_exp})"

```

This helper approximates the result by comparing exponents.

## Why It Matters

- Converts recursive definitions into asymptotic forms
- Avoids repeated substitution or tree expansion
- Applies to most divide-and-conquer algorithms
- Clarifies when combining work dominates or not

## A Gentle Proof (Why It Works)

Expand the recurrence:

$$T(n) = aT(n/b) + f(n)$$

After  $k$  levels:

$$T(n) = a^k T(n/b^k) + \sum_{i=0}^{k-1} a^i f(n/b^i)$$

Recursion depth:  $k = \log_b n$

Now compare total cost per level to  $n^{\log_b a}$ :

- If  $f(n)$  grows slower, top levels dominate → Case 1
- If equal, all levels contribute → Case 2
- If faster, bottom level dominates → Case 3

The asymptotic result depends on which component dominates the sum.

## Try It Yourself

1. Solve  $T(n) = 3T(n/2) + n$
2. Solve  $T(n) = 4T(n/2) + n^2$
3. Sketch recursion trees and check which term dominates

## Test Cases

| Recurrence             | Case   | Solution               |
|------------------------|--------|------------------------|
| $T(n) = 2T(n/2) + n$   | Case 2 | $\Theta(n \log n)$     |
| $T(n) = T(n/2) + 1$    | Case 2 | $\Theta(\log n)$       |
| $T(n) = 7T(n/2) + n^2$ | Case 1 | $\Theta(n^{\log_2 7})$ |
| $T(n) = 2T(n/2) + n^2$ | Case 3 | $\Theta(n^2)$          |

## Complexity Summary

| Component      | Expression                 | Interpretation               |
|----------------|----------------------------|------------------------------|
| Recursive work | $n^{\log_b a}$             | Work across recursive calls  |
| Combine work   | $f(n)$                     | Work per level               |
| Total cost     | $\max(n^{\log_b a}, f(n))$ | Dominant term decides growth |

The Master Theorem serves as a blueprint for analyzing recursive algorithms, once the recurrence is in standard form, its complexity follows by simple comparison.

## 43 Substitution Method

The Substitution Method is a systematic way to prove the asymptotic bound of a recurrence by guessing a solution and then proving it by induction. It's one of the most flexible techniques for verifying time complexity.

### What Problem Are We Solving?

Many algorithms are defined recursively, for example:

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

We often want to show that  $T(n) = O(n \log n)$  or  $T(n) = \Theta(n^2)$ . But before we can apply a theorem, we must confirm that our guess fits.

The substitution method provides a proof framework:

1. Guess the asymptotic bound.
2. Prove it by induction.
3. Adjust constants if necessary.

## How It Works (Plain Language)

1. Make a guess for  $T(n)$ , typically inspired by known patterns. For example, for  $T(n) = 2T(n/2) + n$ , guess  $T(n) = O(n \log n)$ .
2. Write the inductive hypothesis: Assume  $T(k) \leq c, k \log k$  for all  $k < n$ .
3. Substitute into the recurrence: Replace recursive terms with the hypothesis.
4. Simplify and verify: Show the inequality holds for  $n$ , adjusting constants if needed.
5. Conclude that the guess is valid.

## Example Step by Step

Example 1:

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

Goal: Show  $T(n) = O(n \log n)$

1. Hypothesis:  $T(k) \leq c, k \log k$  for all  $k < n$
2. Substitute:  $T(n) \leq 2[c(n/2) \log(n/2)] + n$
3. Simplify:  $= cn \log(n/2) + n = cn(\log n - 1) + n = cn \log n - cn + n$
4. Adjust constant: If  $c \geq 1$ , then  $-cn + n \leq 0$ , so  $T(n) \leq cn \log n$

Therefore,  $T(n) = O(n \log n)$ .

Example 2:

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

Guess:  $T(n) = O(n^{\log_2 3})$

1. Hypothesis:  $T(k) \leq c, k^{\log_2 3}$
2. Substitute:  $T(n) \leq 3c(n/2)^{\log_2 3} + n = 3c \cdot n^{\log_2 3}/3 + n = cn^{\log_2 3} + n$
3. Dominant term:  $n^{\log_2 3} \quad T(n) = O(n^{\log_2 3})$

## Tiny Code (Python)

```

def substitution_check(a, b, f_exp, guess_exp):
    from math import log
    lhs = a * (1 / b) ** guess_exp
    rhs = 1
    if lhs < 1:
        return f"Guess n^{guess_exp} holds (Case 1)"
    elif abs(lhs - 1) < 1e-9:
        return f"Guess n^{guess_exp} log n (Case 2)"
    else:
        return f"Guess n^{guess_exp} fails (try larger exponent)"

```

Helps verify whether a guessed exponent fits the recurrence.

## Why It Matters

- Builds proof-based understanding of complexity
- Confirms asymptotic bounds from intuition or Master Theorem
- Works even when Master Theorem fails (irregular forms)
- Reinforces connection between recursion and growth rate

## A Gentle Proof (Why It Works)

Let  $T(n) = aT(n/b) + f(n)$ . Guess  $T(n) = O(n^{\log_b a})$ .

Inductive step:

$$\begin{aligned} T(n) &= aT(n/b) + f(n) \leq a(c(n/b)^{\log_b a}) + f(n) \\ &= cn^{\log_b a} + f(n) \end{aligned}$$

If  $f(n)$  grows slower,  $T(n)$  remains  $O(n^{\log_b a})$  by choosing  $c$  large enough.

## Try It Yourself

1. Prove  $T(n) = 2T(n/2) + n^2 = O(n^2)$
2. Prove  $T(n) = T(n-1) + 1 = O(n)$
3. Adjust constants to make the induction hold

## Test Cases

| Recurrence            | Guess             | Result  |
|-----------------------|-------------------|---------|
| $T(n) = 2T(n/2) + n$  | $O(n \log n)$     | Correct |
| $T(n) = T(n - 1) + 1$ | $O(n)$            | Correct |
| $T(n) = 3T(n/2) + n$  | $O(n^{\log_2 3})$ | Correct |

## Complexity Summary

| Method         | Effort   | When to Use                     |
|----------------|----------|---------------------------------|
| Master Theorem | Quick    | Standard divide-and-conquer     |
| Substitution   | Moderate | Custom or irregular recurrences |
| Iteration      | Detailed | Step-by-step expansion          |

The Substitution Method blends intuition with rigor, you make a good guess, and algebra does the rest.

## 44 Iteration Method

The Iteration Method (also called the Recursion Expansion Method) solves recurrences by repeatedly substituting the recursive term until the pattern becomes clear. It is a constructive way to derive closed-form or asymptotic solutions.

### What Problem Are We Solving?

Recursive algorithms often define their running time in terms of smaller instances:

$$T(n) = a, T(n/b) + f(n)$$

Instead of guessing or applying a theorem, the iteration method unfolds the recurrence layer by layer, showing exactly how cost accumulates across recursion levels.

This method is especially helpful when  $f(n)$  follows a recognizable pattern, like linear, quadratic, or logarithmic functions.

## How It Works (Plain Language)

1. Write down the recurrence:

$$T(n) = a, T(n/b) + f(n)$$

2. Expand one level at a time:

$$T(n) = a[a, T(n/b^2) + f(n/b)] + f(n)$$

$$= a^2T(n/b^2) + af(n/b) + f(n)$$

3. Continue expanding  $k$  levels until the subproblem size becomes 1:

$$T(n) = a^kT(n/b^k) + \sum_{i=0}^{k-1} a^i f(n/b^i)$$

4. When  $n/b^k = 1$ , we have  $k = \log_b n$ .

5. Substitute  $k = \log_b n$  to find the closed form or asymptotic bound.

## Example Step by Step

Example 1: Merge Sort

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

Step 1: Expand

[ ]

Step 2: Base Case

When  $n/2^k = 1 \implies k = \log_2 n$

So:

$$T(n) = n \cdot T(1) + n \log_2 n = O(n \log n)$$

$$T(n) = \Theta(n \log n)$$

Example 2: Binary Search

$$T(n) = T(n/2) + 1$$

Expand:

[ ]

$$T(n) = O(\log n)$$

Example 3: Linear Recurrence

$$T(n) = T(n - 1) + 1$$

Expand:

$$T(n) = T(n - 1) + 1 = T(n - 2) + 2 = \dots = T(1) + (n - 1)$$

$$T(n) = O(n)$$

### Tiny Code (Python)

```
def iterate_recurrence(a, b, f, n):
    total = 0
    level = 0
    while n > 1:
        total += (a * level) * f(n / (b * level))
        n /= b
        level += 1
    return total
```

This illustrates the summation process level by level.

### Why It Matters

- Makes recursion visually transparent
- Works for irregular  $f(n)$  (when Master Theorem doesn't apply)
- Derives exact sums, not just asymptotic bounds
- Builds intuition for recursion trees and logarithmic depth

## A Gentle Proof (Why It Works)

Each level  $i$  of the recursion contributes:

$$a^i \cdot f(n/b^i)$$

Total number of levels:

$$\log_b n$$

So total cost:

$$T(n) = \sum_{i=0}^{\log_b n - 1} a^i f(n/b^i)$$

This sum can be approximated or bounded using standard summation techniques, depending on  $f(n)$ 's growth rate.

## Try It Yourself

1. Solve  $T(n) = 3T(n/2) + n^2$
2. Solve  $T(n) = 2T(n/2) + n \log n$
3. Solve  $T(n) = T(n/2) + n/2$
4. Compare with Master Theorem results

## Test Cases

| Recurrence             | Solution   | Growth        |
|------------------------|------------|---------------|
| $T(n) = 2T(n/2) + n$   | $n \log n$ | $O(n \log n)$ |
| $T(n) = T(n/2) + 1$    | $\log n$   | $O(\log n)$   |
| $T(n) = T(n - 1) + 1$  | $n$        | $O(n)$        |
| $T(n) = 3T(n/2) + n^2$ | $n^2$      | $O(n^2)$      |

## Complexity Summary

| Step      | Time               | Space                         |
|-----------|--------------------|-------------------------------|
| Expansion | $O(\log n)$ levels | Stack depth $O(\log n)$       |
| Summation | Depends on $f(n)$  | Often geometric or arithmetic |

The Iteration Method unpacks recursion into layers of work, turning a recurrence into a concrete sum, and a sum into a clear complexity bound.

## 45 Generating Function Method

The Generating Function Method transforms a recurrence relation into an algebraic equation by encoding the sequence into a power series. Once transformed, algebraic manipulation yields a closed-form expression or asymptotic approximation.

### What Problem Are We Solving?

A recurrence defines a sequence  $T(n)$  recursively:

$$T(n) = a_1 T(n-1) + a_2 T(n-2) + \cdots + f(n)$$

We want to find a closed-form formula instead of computing step by step. By representing  $T(n)$  as coefficients in a power series, we can use algebraic tools to solve recurrences cleanly, especially linear recurrences with constant coefficients.

### How It Works (Plain Language)

1. Define the generating function Let

$$G(x) = \sum_{n=0}^{\infty} T(n)x^n$$

2. Multiply the recurrence by  $x^n$  and sum over all  $n$ .
3. Use properties of sums (shifting indices, factoring constants) to rewrite in terms of  $G(x)$ .
4. Solve the algebraic equation for  $G(x)$ .
5. Expand  $G(x)$  back into a series (using partial fractions or known expansions).
6. Extract  $T(n)$  as the coefficient of  $x^n$ .

## Example Step by Step

Example 1: Fibonacci Sequence

$$T(n) = T(n-1) + T(n-2), \quad T(0) = 0,; T(1) = 1$$

Step 1: Define generating function

$$G(x) = \sum_{n=0}^{\infty} T(n)x^n$$

Step 2: Multiply recurrence by  $x^n$  and sum over  $n \geq 2$ :

$$\sum_{n=2}^{\infty} T(n)x^n = \sum_{n=2}^{\infty} T(n-1)x^n + \sum_{n=2}^{\infty} T(n-2)x^n$$

Step 3: Rewrite using shifts:

$$G(x) - T(0) - T(1)x = x(G(x) - T(0)) + x^2G(x)$$

Plug in initial values  $T(0) = 0, T(1) = 1$ :

$$G(x) - x = xG(x) + x^2G(x)$$

Step 4: Solve for  $G(x)$ :

$$G(x)(1 - x - x^2) = x$$

So:

$$G(x) = \frac{x}{1 - x - x^2}$$

Step 5: Expand using partial fractions to get coefficients:

$$T(n) = \frac{1}{\sqrt{5}} \left[ \left( \frac{1 + \sqrt{5}}{2} \right)^n - \left( \frac{1 - \sqrt{5}}{2} \right)^n \right]$$

Binet's Formula derived directly.

Example 2:  $T(n) = 2T(n-1) + 3, T(0) = 1$

Let  $G(x) = \sum_{n=0}^{\infty} T(n)x^n$

Multiply by  $x^n$  and sum over  $n \geq 1$ :

$$G(x) - T(0) = 2xG(x) + 3x \cdot \frac{1}{1-x}$$

Simplify:

$$G(x)(1-2x) = 1 + \frac{3x}{1-x}$$

Solve and expand using partial fractions  $\rightarrow$  recover closed-form:

$$T(n) = 4 \cdot 2^n - 3$$

### Tiny Code (Python)

```
from sympy import symbols, Function, Eq, rsolve

n = symbols('n', integer=True)
T = Function('T')
recurrence = Eq(T(n), 2*T(n-1) + 3)
solution = rsolve(recurrence, T(n), {T(0): 1})
print(solution) # 4*2^n - 3
```

Use `sympy.rsolve` to compute closed forms symbolically.

### Why It Matters

- Converts recurrence relations into algebraic equations
- Reveals exact closed forms, not just asymptotics
- Works for non-homogeneous and constant-coefficient recurrences
- Bridges combinatorics, discrete math, and algorithm analysis

## A Gentle Proof (Why It Works)

Given a linear recurrence:

$$T(n) - a_1 T(n-1) - \cdots - a_k T(n-k) = f(n)$$

Multiply by  $x^n$  and sum from  $n = k$  to  $\infty$ :

$$\sum_{n=k}^{\infty} T(n)x^n = a_1 x \sum_{n=k}^{\infty} T(n-1)x^{n-1} + \cdots + f(x)$$

Using index shifts, each term can be written in terms of  $G(x)$ , leading to:

$$P(x)G(x) = Q(x)$$

where  $P(x)$  and  $Q(x)$  are polynomials. Solving for  $G(x)$  gives the sequence structure.

## Try It Yourself

1. Solve  $T(n) = 3T(n-1) - 2T(n-2)$  with  $T(0) = 2, T(1) = 3$ .
2. Find  $T(n)$  if  $T(n) = T(n-1) + 1, T(0) = 0$ .
3. Compare your generating function with unrolled expansion.

## Test Cases

| Recurrence               | Closed Form       | Growth      |
|--------------------------|-------------------|-------------|
| $T(n) = 2T(n-1) + 3$     | $4 \cdot 2^n - 3$ | $O(2^n)$    |
| $T(n) = T(n-1) + 1$      | $n$               | $O(n)$      |
| $T(n) = T(n-1) + T(n-2)$ | Binet's Formula   | $O(\phi^n)$ |

## Complexity Summary

| Step           | Type                 | Complexity   |
|----------------|----------------------|--------------|
| Transformation | Algebraic            | $O(k)$ terms |
| Solution       | Symbolic (via roots) | $O(k^3)$     |
| Evaluation     | Closed form          | $O(1)$       |

The Generating Function Method turns recurrences into algebra, summations become equations, and equations yield exact formulas.

## 46 Matrix Exponentiation

The Matrix Exponentiation Method transforms linear recurrences into matrix form, allowing efficient computation of terms in  $O(\log n)$  time using fast exponentiation. It's ideal for sequences like Fibonacci, Tribonacci, and many dynamic programming transitions.

### What Problem Are We Solving?

Many recurrences follow a linear relation among previous terms, such as:

$$T(n) = a_1 T(n-1) + a_2 T(n-2) + \cdots + a_k T(n-k)$$

Naively computing  $T(n)$  takes  $O(n)$  steps. By encoding this recurrence in a matrix, we can compute  $T(n)$  efficiently via exponentiation, reducing runtime to  $O(k^3 \log n)$ .

### How It Works (Plain Language)

1. Express the recurrence as a matrix multiplication.
2. Construct the transition matrix  $M$  that moves the state from  $n-1$  to  $n$ .
3. Compute  $M^n$  using fast exponentiation (divide and conquer).
4. Multiply  $M^n$  by the initial vector to obtain  $T(n)$ .

This approach generalizes well to any linear homogeneous recurrence with constant coefficients.

## **Example Step by Step**

Example 1: Fibonacci Sequence

$$F(n) = F(n - 1) + F(n - 2)$$

Define state vector:

$$\begin{bmatrix} F(n) \\ F(n - 1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F(n - 1) \\ F(n - 2) \end{bmatrix}$$

So:

$$M = \begin{bmatrix} 1 & 1 \\ 6pt & 1 & 0 \end{bmatrix}$$

Therefore:

$$\begin{bmatrix} F(n) & F(n - 1) \end{bmatrix} = M^{n-1} \begin{bmatrix} F(1) & F(0) \end{bmatrix}$$

Given  $F(1) = 1, F(0) = 0,$

$$F(n) = (M^{n-1})_{0,0}$$

Example 2: Second-Order Recurrence

$$T(n) = 2T(n - 1) + 3T(n - 2)$$

Matrix form:

$$\begin{bmatrix} T(n) \\ T(n - 1) \end{bmatrix} = \begin{bmatrix} 2 & 3 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} T(n - 1) \\ T(n - 2) \end{bmatrix}$$

So:

$$\vec{T}(n) = M^{n-2} \vec{T}(2)$$

## **Tiny Code (Python)**

```

def mat_mult(A, B):
    return [[sum(A[i][k] * B[k][j] for k in range(len(A)))
             for j in range(len(B[0]))] for i in range(len(A))]

def mat_pow(M, n):
    if n == 1:
        return M
    if n % 2 == 0:
        half = mat_pow(M, n // 2)
        return mat_mult(half, half)
    else:
        return mat_mult(M, mat_pow(M, n - 1))

def fib_matrix(n):
    if n == 0:
        return 0
    M = [[1, 1], [1, 0]]
    Mn = mat_pow(M, n - 1)
    return Mn[0][0]

```

`fib_matrix(n)` computes  $F(n)$  in  $O(\log n)$ .

## Why It Matters

- Converts recursive computation into linear algebra
- Enables  $O(\log n)$  computation for  $T(n)$
- Generalizes to higher-order recurrences
- Common in DP transitions, Fibonacci-like sequences, and combinatorial counting

## A Gentle Proof (Why It Works)

The recurrence:

$$T(n) = a_1 T(n-1) + a_2 T(n-2) + \cdots + a_k T(n-k)$$

can be expressed as:

$$\vec{T}(n) = M \cdot \vec{T}(n-1)$$

where  $M$  is the companion matrix:

$$M = \begin{bmatrix} a_1 & a_2 & \cdots & a_k \\ 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & & \ddots & 0 \end{bmatrix}$$

Repeatedly multiplying gives:

$$\vec{T}(n) = M^{n-k} \vec{T}(k)$$

Hence,  $T(n)$  is computed by raising  $M$  to a power, exponential recursion becomes logarithmic multiplication.

### Try It Yourself

1. Write matrix form for  $T(n) = 3T(n-1) - 2T(n-2)$
2. Compute  $T(10)$  with  $T(0) = 2, T(1) = 3$
3. Implement matrix exponentiation for  $3 \times 3$  matrices (Tribonacci)
4. Compare with iterative solution runtime

### Test Cases

| Recurrence                        | Matrix                                                              | $T(n)$ / Symbol | Complexity  |
|-----------------------------------|---------------------------------------------------------------------|-----------------|-------------|
| $F(n) = F(n-1) + F(n-2)$          | $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$                      | $F(n)$          | $O(\log n)$ |
| $T(n) = 2T(n-1) + 3T(n-2)$        | $\begin{bmatrix} 2 & 3 \\ 1 & 0 \end{bmatrix}$                      | $T(n)$          | $O(\log n)$ |
| $T(n) = T(n-1) + T(n-2) + T(n-3)$ | $\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$ | Tribonacci      | $O(\log n)$ |

### Complexity Summary

| Step                  | Time            | Space    |
|-----------------------|-----------------|----------|
| Matrix exponentiation | $O(k^3 \log n)$ | $O(k^2)$ |
| Iterative recurrence  | $O(n)$          | $O(k)$   |

Matrix Exponentiation turns recurrence solving into matrix powering, a bridge between recursion and linear algebra, giving exponential speed-up with mathematical elegance.

## 47 Recurrence to DP Table

The Recurrence to DP Table method converts a recursive relation into an iterative table-based approach, removing redundant computation and improving efficiency from exponential to polynomial time. It's a cornerstone of Dynamic Programming.

### What Problem Are We Solving?

Recursive formulas often recompute overlapping subproblems. For example:

$$T(n) = T(n - 1) + T(n - 2)$$

A naive recursive call tree grows exponentially because it recomputes  $T(k)$  many times. By converting this recurrence into a DP table, we compute each subproblem once and store results, achieving linear or polynomial time.

### How It Works (Plain Language)

1. Identify the recurrence and base cases.
2. Create a table (array or matrix) to store subproblem results.
3. Iteratively fill the table using the recurrence formula.
4. Read off the final answer from the last cell.

This technique is called tabulation, a bottom-up form of dynamic programming.

## Example Step by Step

Example 1: Fibonacci Numbers

Recursive formula:

$$F(n) = F(n - 1) + F(n - 2), \quad F(0) = 0, F(1) = 1$$

DP version:

|      |   |   |   |   |   |   |
|------|---|---|---|---|---|---|
| n    | 0 | 1 | 2 | 3 | 4 | 5 |
| F(n) | 0 | 1 | 1 | 2 | 3 | 5 |

Algorithm:

1. Initialize base cases:  $F[0]=0, F[1]=1$
2. Loop from 2 to n:  $F[i] = F[i-1] + F[i-2]$
3. Return  $F[n]$

Example 2: Coin Change (Count Ways)

Recurrence:

$$\text{ways}(n, c) = \text{ways}(n, c - 1) + \text{ways}(n - \text{coin}[c], c)$$

Convert to 2D DP table indexed by (n, c).

Example 3: Grid Paths

Recurrence:

$$P(i, j) = P(i - 1, j) + P(i, j - 1)$$

DP table:

|    |   |   |   |
|----|---|---|---|
| ij | 0 | 1 | 2 |
| 0  | 1 | 1 | 1 |
| 1  | 1 | 2 | 3 |
| 2  | 1 | 3 | 6 |

Each cell = sum of top and left.

## Tiny Code (Python)

```
def fib_dp(n):
    if n == 0:
        return 0
    dp = [0] * (n + 1)
    dp[1] = 1
    for i in range(2, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2]
    return dp[n]
```

## Why It Matters

- Converts exponential recursion to polynomial iteration
- Avoids repeated subproblem computations
- Enables space and time optimization
- Forms the foundation of bottom-up dynamic programming

## A Gentle Proof (Why It Works)

Given recurrence:

$$T(n) = a_1 T(n-1) + a_2 T(n-2) + \dots + a_k T(n-k)$$

Each term depends only on previously computed subproblems. So by filling the table in increasing order, we ensure all dependencies are ready.

By induction, if base cases are correct, each computed cell is correct.

## Try It Yourself

1. Convert  $F(n) = F(n-1) + F(n-2)$  to a 1D DP array
2. Build a 2D table for grid paths  $P(i,j) = P(i-1,j) + P(i,j-1)$
3. Write a DP table for factorial  $n! = n \times (n-1)!$
4. Optimize space (keep only last k terms)

## Test Cases

| Input                   | Recurrence                                                                       | Ex-<br>pected |
|-------------------------|----------------------------------------------------------------------------------|---------------|
| $F(5)$                  | $F(n) = F(n - 1) + F(n - 2)$                                                     | 5             |
| Grid(2,2)               | $P(i, j) = P(i - 1, j) + P(i, j - 1)$                                            | 6             |
| $n = 3, coins = [1, 2]$ | $\text{ways}(n, c) = \text{ways}(n, c - 1) + \text{ways}(n - \text{coin}[c], c)$ | 2             |

## Complexity Summary

| Method             | Time     | Space  |
|--------------------|----------|--------|
| Recursive          | $O(2^n)$ | $O(n)$ |
| DP Table           | $O(n)$   | $O(n)$ |
| Space-Optimized DP | $O(n)$   | $O(1)$ |

Transforming a recurrence into a DP table captures the essence of dynamic programming, structure, reuse, and clarity over brute repetition.

## 48 Divide & Combine Template

The Divide & Combine Template is a structural guide for solving problems by breaking them into smaller, similar subproblems, solving each independently, and combining their results. It's the core skeleton behind divide-and-conquer algorithms like Merge Sort, Quick Sort, and Karatsuba Multiplication.

### What Problem Are We Solving?

Many complex problems can be decomposed into smaller copies of themselves. Instead of solving the full instance at once, we divide it into subproblems, solve each recursively, and combine their results.

This approach reduces complexity, promotes parallelism, and yields recurrence relations like:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

## How It Works (Plain Language)

1. Divide: Split the problem into  $a$  subproblems, each of size  $\frac{n}{b}$ .
2. Conquer: Recursively solve the subproblems.
3. Combine: Merge their results into a full solution.
4. Base Case: Stop dividing when the subproblem becomes trivially small.

This recursive structure underpins most efficient algorithms for sorting, searching, and multiplication.

## Example Step by Step

Example 1: Merge Sort

- Divide: Split array into two halves
- Conquer: Recursively sort each half
- Combine: Merge two sorted halves

Recurrence:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Example 2: Karatsuba Multiplication

- Divide numbers into halves
- Conquer with 3 recursive multiplications
- Combine using linear combinations

Recurrence:

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n)$$

Example 3: Binary Search

- Divide the array by midpoint
- Conquer on one half
- Combine trivially (return result)

Recurrence:

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

## Generic Template (Pseudocode)

```

def divide_and_combine(problem):
    if is_small(problem):
        return solve_directly(problem)
    subproblems = divide(problem)
    results = [divide_and_combine(p) for p in subproblems]
    return combine(results)

```

This general template can adapt to many problem domains, arrays, trees, graphs, geometry, and algebra.

## Why It Matters

- Clarifies recursion structure and base case reasoning
- Enables asymptotic analysis via recurrence
- Lays foundation for parallel and cache-efficient algorithms
- Promotes clean decomposition and reusability

## A Gentle Proof (Why It Works)

If a problem can be decomposed into independent subproblems whose results can be merged deterministically, recursive decomposition is valid. By induction:

- Base case: small input solved directly.
- Inductive step: if each subproblem is solved correctly, and the combine step correctly merges, the final solution is correct.

Thus correctness follows from structural decomposition.

## Try It Yourself

1. Implement divide-and-conquer sum over an array.
2. Write recursive structure for Maximum Subarray (Kadane's divide form).
3. Express recurrence  $T(n) = 2T(n/2) + n$  and solve via the Master Theorem.
4. Modify template for parallel processing (e.g., thread pool).

## Test Cases

| Problem           | Divide             | Combine                    | Complexity     |
|-------------------|--------------------|----------------------------|----------------|
| Merge Sort        | Halve array        | Merge sorted halves        | $O(n \log n)$  |
| Binary Search     | Halve search space | Return result              | $O(\log n)$    |
| Karatsuba         | Split numbers      | Combine linear parts       | $O(n^{1.585})$ |
| Closest Pair (2D) | Split points       | Merge cross-boundary pairs | $O(n \log n)$  |

## Complexity Summary

Given:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

By the Master Theorem:

- If  $f(n) = O(n^{\log_b a - \epsilon})$ , then  $T(n) = \Theta(n^{\log_b a})$
- If  $f(n) = \Theta(n^{\log_b a})$ , then  $T(n) = \Theta(n^{\log_b a} \log n)$
- If  $f(n) = \Omega(n^{\log_b a + \epsilon})$ , then  $T(n) = \Theta(f(n))$

The Divide & Combine Template provides the blueprint for recursive problem solving, simple, elegant, and foundational across all algorithmic domains.

## 49 Memoized Recursive Solver

A Memoized Recursive Solver transforms a plain recursive solution into an efficient one by caching intermediate results. It's the top-down version of dynamic programming, retaining recursion's clarity while avoiding redundant work.

### What Problem Are We Solving?

Recursive algorithms often recompute the same subproblems multiple times. Example:

$$F(n) = F(n - 1) + F(n - 2)$$

A naive recursive call tree repeats  $F(3)$ ,  $F(2)$ , etc., exponentially many times. By memoizing (storing) results after the first computation, we reuse them in  $O(1)$  time later.

## How It Works (Plain Language)

1. Define the recurrence clearly.
2. Add a cache (dictionary or array) to store computed results.
3. Before each recursive call, check the cache.
4. If present, return cached value.
5. Otherwise, compute, store, and return.

This approach preserves recursive elegance while matching iterative DP performance.

## Example Step by Step

Example 1: Fibonacci Numbers

Naive recursion:

$$F(n) = F(n - 1) + F(n - 2)$$

Memoized version:

| n | F(n) | Cached?        |
|---|------|----------------|
| 0 | 0    | Base           |
| 1 | 1    | Base           |
| 2 | 1    | Computed       |
| 3 | 2    | Computed       |
| 4 | 3    | Cached lookups |

Time drops from  $O(2^n)$  to  $O(n)$ .

Example 2: Binomial Coefficient

Recurrence:

$$C(n, k) = C(n - 1, k - 1) + C(n - 1, k)$$

Without memoization: exponential With memoization:  $O(nk)$

Example 3: Coin Change

$$\text{ways}(n) = \text{ways}(n - \text{coin}) + \text{ways}(n, \text{next})$$

Memoize by  $(n, \text{index})$  to avoid recomputing states.

## Tiny Code (Python)

```
def fib_memo(n, memo={}):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fib_memo(n-1, memo) + fib_memo(n-2, memo)
    return memo[n]
```

Or explicitly pass cache:

```
def fib_memo(n):
    memo = []
    def helper(k):
        if k in memo:
            return memo[k]
        if k <= 1:
            return k
        memo[k] = helper(k-1) + helper(k-2)
        return memo[k]
    return helper(n)
```

## Why It Matters

- Retains intuitive recursive structure
- Cuts time complexity drastically
- Natural stepping stone to tabulation (bottom-up DP)
- Enables solving overlapping subproblem recurrences efficiently

## A Gentle Proof (Why It Works)

Let  $S$  be the set of all distinct subproblems. Without memoization, each is recomputed exponentially many times. With memoization, each  $s \in S$  is computed exactly once. Thus, total time =  $O(|S|)$ .

## Try It Yourself

1. Add memoization to naive Fibonacci.
2. Memoize binomial coefficients  $C(n, k)$ .
3. Apply memoization to knapsack recursion.
4. Count total recursive calls with and without memoization.

## Test Cases

| Problem          | Naive Time  | Memoized Time  |
|------------------|-------------|----------------|
| Fibonacci(40)    | $O(2^{40})$ | $O(40)$        |
| Binomial(20,10)  | $O(2^{20})$ | $O(200)$       |
| Coin Change(100) | $O(2^n)$    | $O(n \cdot k)$ |

## Complexity Summary

| Method    | Time                              | Space         |
|-----------|-----------------------------------|---------------|
| Recursive | Exponential                       | $O(n)$ stack  |
| Memoized  | Polynomial (distinct subproblems) | Cache + stack |

Memoization blends clarity and efficiency, recursion that remembers. It turns naive exponential algorithms into elegant linear or polynomial solutions with a single insight: never solve the same problem twice.

## 50 Characteristic Polynomial Solver

The Characteristic Polynomial Solver is a powerful algebraic technique for solving linear homogeneous recurrence relations with constant coefficients. It expresses the recurrence in terms of polynomial roots, giving closed-form solutions.

### What Problem Are We Solving?

When faced with recurrences like:

$$T(n) = a_1 T(n-1) + a_2 T(n-2) + \dots + a_k T(n-k)$$

we want a closed-form expression for  $T(n)$  instead of step-by-step computation. The characteristic polynomial captures the recurrence's structure, its roots determine the general form of the solution.

### How It Works (Plain Language)

1. Write the recurrence in standard form:

$$T(n) - a_1 T(n-1) - a_2 T(n-2) - \cdots - a_k T(n-k) = 0$$

2. Replace  $T(n-i)$  with  $r^{n-i}$  to form a polynomial equation:

$$r^k - a_1 r^{k-1} - a_2 r^{k-2} - \cdots - a_k = 0$$

3. Solve for roots  $r_1, r_2, \dots, r_k$ .

4. The general solution is:

$$T(n) = c_1 r_1^n + c_2 r_2^n + \cdots + c_k r_k^n$$

5. Use initial conditions to solve for constants  $c_i$ .

If there are repeated roots, multiply by  $n^p$  for multiplicity  $p$ .

### Example Step by Step

Example 1: Fibonacci

Recurrence:

$$F(n) = F(n-1) + F(n-2)$$

Characteristic polynomial:

$$r^2 - r - 1 = 0$$

Roots:

$$r_1 = \frac{1 + \sqrt{5}}{2}, \quad r_2 = \frac{1 - \sqrt{5}}{2}$$

General solution:

$$F(n) = c_1 r_1^n + c_2 r_2^n$$

Using  $F(0) = 0, F(1) = 1$ :

$$c_1 = \frac{1}{\sqrt{5}}, \quad c_2 = -\frac{1}{\sqrt{5}}$$

So:

$$F(n) = \frac{1}{\sqrt{5}} \left( \left( \frac{1+\sqrt{5}}{2} \right)^n - \left( \frac{1-\sqrt{5}}{2} \right)^n \right)$$

This is Binet's Formula.

Example 2:  $T(n) = 3T(n-1) - 2T(n-2)$

Characteristic polynomial:

$$r^2 - 3r + 2 = 0 \implies (r-1)(r-2) = 0$$

Roots:  $r_1 = 1, r_2 = 2$

Solution:

$$T(n) = c_1(1)^n + c_2(2)^n = c_1 + c_2 2^n$$

Use base cases to find  $c_1, c_2$ .

Example 3: Repeated Roots

$$T(n) = 2T(n-1) - T(n-2)$$

Characteristic:

$$r^2 - 2r + 1 = 0 \implies (r-1)^2 = 0$$

Solution:

$$T(n) = (c_1 + c_2 n) \cdot 1^n = c_1 + c_2 n$$

### Tiny Code (Python)

```
import sympy as sp

def solve_recurrence(coeffs, initials):
    n = len(coeffs)
    r = sp.symbols('r')
    poly = r**n - sum(coeffs[i]*r**(n-i-1) for i in range(n))
    roots = sp.roots(poly, r)
    r_syms = list(roots.keys())
    c = sp.symbols(' '.join([f'c{i+1}' for i in range(n)]))
    Tn = sum(c[i]*r_syms[i]*sp.symbols('n') for i in range(n))
    equations = []
```

```

for i, val in enumerate(initials):
    equations.append(Tn.subs(sp.symbols('n'), i) - val)
sol = sp.solve(equations, c)
return Tn.subs(sol)

```

Call `solve_recurrence([1, 1], [0, 1])` → Binet's formula.

## Why It Matters

- Gives closed-form solutions for linear recurrences
- Eliminates need for iteration or recursion
- Connects algorithm analysis to algebra and eigenvalues
- Used in runtime analysis, combinatorics, and discrete modeling

## A Gentle Proof (Why It Works)

Suppose recurrence:

$$T(n) = a_1 T(n-1) + \cdots + a_k T(n-k)$$

Assume  $T(n) = r^n$ :

$$r^n = a_1 r^{n-1} + \cdots + a_k r^{n-k}$$

Divide by  $r^{n-k}$ :

$$r^k = a_1 r^{k-1} + \cdots + a_k$$

Solve polynomial for roots. Each root corresponds to an independent solution. By linearity, the sum of independent solutions is also a solution.

## Try It Yourself

1. Solve  $T(n) = 2T(n-1) + T(n-2)$  with  $T(0) = 0, T(1) = 1$ .
2. Solve  $T(n) = T(n-1) + 2T(n-2)$  with  $T(0) = 2, T(1) = 3$ .
3. Solve with repeated root  $r = 1$ .
4. Verify results numerically for  $n = 0 \dots 5$ .

## Test Cases

| Recurrence                 | Polynomial         | Roots                      | Closed Form     |
|----------------------------|--------------------|----------------------------|-----------------|
| $F(n) = F(n-1) + F(n-2)$   | $r^2 - r - 1 = 0$  | $\frac{1 \pm \sqrt{5}}{2}$ | Binet           |
| $T(n) = 3T(n-1) - 2T(n-2)$ | $r^2 - 3r + 2 = 0$ | 1, 2                       | $c_1 + c_2 2^n$ |
| $T(n) = 2T(n-1) - T(n-2)$  | $(r-1)^2 = 0$      | 1 (double)                 | $c_1 + c_2 n$   |

## Complexity Summary

| Step                 | Time     | Space  |
|----------------------|----------|--------|
| Solve polynomial     | $O(k^3)$ | $O(k)$ |
| Evaluate closed form | $O(1)$   | $O(1)$ |

The Characteristic Polynomial Solver is the algebraic heart of recurrence solving, turning repeated patterns into exact formulas through the power of roots and symmetry.

## Section 6. Searching basics

### 51 Search Space Visualizer

A Search Space Visualizer is a conceptual tool to map and understand the entire landscape of possibilities an algorithm explores. By modeling the search process as a tree or graph, you gain intuition about completeness, optimality, and complexity before diving into code.

#### What Problem Are We Solving?

When tackling problems like optimization, constraint satisfaction, or pathfinding, the solution isn't immediate, we must explore a space of possibilities. Understanding how large that space is, how it grows, and how it can be pruned is crucial for algorithmic design.

Visualizing the search space helps answer questions like:

- How many states are reachable?
- How deep or wide is the search?
- What's the branching factor?
- Where does the goal lie?

## How It Works (Plain Language)

1. Model states as nodes. Each represents a partial or complete solution.
2. Model transitions as edges. Each move or decision takes you to a new state.
3. Define start and goal nodes. Typically, the root (start) expands toward one or more goals.
4. Trace the exploration. Breadth-first explores level by level; depth-first dives deep.
5. Label nodes with cost or heuristic values if applicable (for A\*, branch-and-bound, etc.).

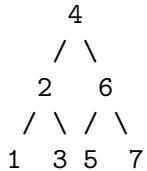
This structure reveals not just correctness but also efficiency and complexity.

## Example Step by Step

Example 1: Binary Search Tree Traversal

For array [1, 2, 3, 4, 5, 6, 7] and target = 6:

Search space (comparisons):



Path explored: 4 → 6 (found)

Search space depth:  $\log_2 7 \approx 3$

Example 2: 8-Queens Problem

Each level represents placing a queen in a new row. Branching factor shrinks as constraints reduce possibilities.

Visualization shows 8! total paths, but pruning cuts most.

Example 3: Maze Solver

States = grid cells; edges = possible moves.

Visualization helps you see BFS's wavefront vs DFS's depth-first path.

## Tiny Code (Python)

```

from collections import deque

def visualize_bfs(graph, start):
    visited = set()
    queue = deque([(start, [start])])
    while queue:
        node, path = queue.popleft()
        print(f"Visiting: {node}, Path: {path}")
        visited.add(node)
        for neighbor in graph[node]:
            if neighbor not in visited:
                queue.append((neighbor, path + [neighbor]))

```

Use on a small adjacency list to see BFS layers unfold.

## Why It Matters

- Builds intuition about algorithm behavior
- Shows breadth vs depth tradeoffs
- Reveals redundant paths and pruning opportunities
- Useful for teaching, debugging, and complexity estimation

## A Gentle Proof (Why It Works)

Let each state  $s \in S$  be connected by transitions  $E$ . Search algorithms define an ordering of node expansion (DFS, BFS, heuristic-based). Visualizing  $S$  as a graph preserves:

- Completeness: BFS explores all finite paths
- Optimality: with uniform cost, shortest path = first found
- Complexity: proportional to nodes generated (often  $O(b^d)$ )

## Try It Yourself

1. Draw search tree for binary search on 7 elements.
2. Visualize DFS vs BFS on a maze.
3. Build search space for placing 4 queens on a  $4 \times 4$  board.
4. Compare path counts with and without pruning.

## Test Cases

| Problem       | Search Space Size | Visualization Insight  |
|---------------|-------------------|------------------------|
| Binary Search | $\log_2 n$        | Narrow, balanced       |
| 8-Queens      | $8!$              | Heavy pruning needed   |
| Maze (10x10)  | 100 nodes         | BFS = wave, DFS = path |
| Sudoku        | $9^{81}$          | Prune with constraints |

## Complexity Summary

| Algorithm | Nodes Explored | Memory   | Visualization        |
|-----------|----------------|----------|----------------------|
| BFS       | $O(b^d)$       | $O(b^d)$ | Tree layers          |
| DFS       | $O(bd)$        | $O(d)$   | Deep path            |
| A*        | $O(b^d)$       | $O(b^d)$ | Cost-guided frontier |

A Search Space Visualizer turns abstract computation into geometry, making invisible exploration visible, and helping you reason about complexity before coding.

## 52 Decision Tree Depth Estimator

A Decision Tree Depth Estimator helps you reason about how many questions, comparisons, or branching choices an algorithm must make in the worst, best, or average case. It models decision-making as a tree, where each node is a test and each leaf is an outcome.

### What Problem Are We Solving?

Any algorithm that proceeds by comparisons or conditional branches (like sorting, searching, or classification) can be represented as a decision tree. Analyzing its depth gives insight into:

- Worst-case time complexity (longest path)
- Best-case time complexity (shortest path)
- Average-case complexity (weighted path length)

By studying depth, we understand the minimum information needed to solve the problem.

## How It Works (Plain Language)

1. Represent each comparison or condition as a branching node.
2. Follow each branch based on true/false or less/greater outcomes.
3. Each leaf represents a solved instance (e.g. sorted array, found key).
4. The depth = number of decisions on a path.
5. Maximum depth  $\rightarrow$  worst-case cost.

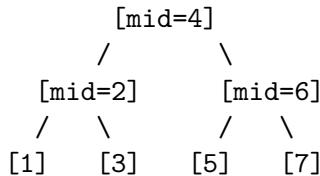
This model abstracts away details and focuses purely on information flow.

## Example Step by Step

Example 1: Binary Search

- Each comparison halves the search space.
- Decision tree has depth  $\log_2 n$ .
- Minimum comparisons in worst case:  $\lceil \log_2 n \rceil$ .

Tree for  $n = 8$  elements:



Depth:  $3 = \log_2 8$

Example 2: Comparison Sort

Each leaf represents a possible ordering. A valid sorting tree must distinguish all  $n!$  orderings.

So:

$$2^h \geq n! \implies h \geq \log_2(n!)$$

Thus, any comparison sort has lower bound:

$$\Omega(n \log n)$$

Example 3: Decision-Making Algorithm

If solving a yes/no classification with  $b$  possible outcomes, minimum number of comparisons required =  $\lceil \log_2 b \rceil$ .

## Tiny Code (Python)

```
import math

def decision_tree_depth(outcomes):
    # Minimum comparisons to distinguish outcomes
    return math.ceil(math.log2(outcomes))

print(decision_tree_depth(8))  # 3
print(decision_tree_depth(120))  # ~7 (for 5!)
```

## Why It Matters

- Reveals theoretical limits (no sort faster than  $O(n \log n)$  by comparison)
- Models decision complexity in search and optimization
- Bridges information theory and algorithm design
- Helps compare branching strategies

## A Gentle Proof (Why It Works)

Each comparison splits the search space in two. To distinguish  $N$  possible outcomes, need at least  $h$  comparisons such that:

$$2^h \geq N$$

Thus:

$$h \geq \lceil \log_2 N \rceil$$

For sorting:

$$N = n! \implies h \geq \log_2(n!) = \Omega(n \log n)$$

This bound holds independent of implementation, it's a lower bound on information required.

## Try It Yourself

1. Build decision tree for 3-element sorting.
2. Count comparisons for binary search on  $n = 16$ .
3. Estimate lower bound for 4-element comparison sort.
4. Visualize tree for classification with 8 classes.

## Test Cases

| Problem             | Outcomes   | Depth Bound |
|---------------------|------------|-------------|
| Binary Search (n=8) | 8          | 3           |
| Sort 3 elements     | $3! = 6$   | $\geq 3$    |
| Sort 5 elements     | $5! = 120$ | $\geq 7$    |
| Classify 8 outcomes | 8          | 3           |

## Complexity Summary

| Algorithm       | Search Space | Depth       | Meaning                   |
|-----------------|--------------|-------------|---------------------------|
| Binary Search   | $n$          | $\log_2 n$  | Worst-case comparisons    |
| Comparison Sort | $n!$         | $\log_2 n!$ | Info-theoretic limit      |
| Classifier      | $b$          | $\log_2 b$  | Min tests for $b$ classes |

A Decision Tree Depth Estimator helps uncover the invisible “question complexity” behind every algorithm, how many decisions must be made, no matter how clever your code is.

## 53 Comparison Counter

A Comparison Counter measures how many times an algorithm compares elements or conditions, a direct way to understand its time complexity, efficiency, and practical performance. Counting comparisons gives insight into what really drives runtime, especially in comparison-based algorithms.

### What Problem Are We Solving?

Many algorithms, sorting, searching, selection, optimization, revolve around comparisons. Every `if`, `<`, or `==` is a decision that costs time.

By counting comparisons, we can:

- Estimate exact step counts for small inputs
- Verify asymptotic bounds ( $O(n^2)$ ,  $O(n \log n)$ , etc.)
- Compare different algorithms empirically
- Identify hot spots in implementation

This turns performance from a vague idea into measurable data.

## How It Works (Plain Language)

1. Instrument the algorithm: wrap every comparison in a counter.
2. Increment the counter each time a comparison occurs.
3. Run the algorithm with sample inputs.
4. Observe patterns as input size grows.
5. Fit results to complexity functions ( $n$ ,  $n \log n$ ,  $n^2$ , etc.).

This gives both empirical evidence and analytic insight.

## Example Step by Step

Example 1: Linear Search

Search through an array of size  $n$ . Each comparison checks one element.

| Case    | Comparisons     |
|---------|-----------------|
| Best    | 1               |
| Worst   | $n$             |
| Average | $\frac{n+1}{2}$ |

So:

$$T(n) = O(n)$$

Example 2: Binary Search

Each step halves the search space.

| Case    | Comparisons              |
|---------|--------------------------|
| Best    | 1                        |
| Worst   | $\lceil \log_2 n \rceil$ |
| Average | $\approx \log_2 n - 1$   |

So:

$$T(n) = O(\log n)$$

Example 3: Bubble Sort

For array of length  $n$ , each pass compares adjacent elements.

| Pass  | Comparisons |
|-------|-------------|
| 1     | $n - 1$     |
| 2     | $n - 2$     |
| ...   | ...         |
| $n-1$ | 1           |

Total:

$$C(n) = (n - 1) + (n - 2) + \dots + 1 = \frac{n(n - 1)}{2}$$

So:

$$T(n) = O(n^2)$$

### Tiny Code (Python)

```
class Counter:
    def __init__(self):
        self.count = 0
    def compare(self, a, b, op):
        self.count += 1
        if op == '<': return a < b
        if op == '>': return a > b
        if op == '==' : return a == b

# Example: Bubble Sort
def bubble_sort(arr):
    c = Counter()
    n = len(arr)
    for i in range(n):
        for j in range(n - i - 1):
            if c.compare(arr[j], arr[j + 1], '>'):
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr, c.count
```

Run on small arrays to record exact comparison counts.

## Why It Matters

- Converts abstract complexity into measurable data
- Reveals hidden constants and practical performance
- Useful for algorithm profiling and pedagogy
- Helps confirm theoretical analysis

## A Gentle Proof (Why It Works)

Each comparison corresponds to one node in the algorithm's decision tree. The number of comparisons = number of nodes visited. Counting comparisons thus measures path length, which correlates to runtime for comparison-based algorithms.

By summing over all paths, we recover the exact cost function  $C(n)$ .

## Try It Yourself

1. Count comparisons in bubble sort vs insertion sort for  $n = 5$ .
2. Measure binary search comparisons for  $n = 16$ .
3. Compare selection sort and merge sort.
4. Fit measured values to theoretical  $O(n^2)$  or  $O(n \log n)$ .

## Test Cases

| Algorithm     | Input Size | Comparisons | Pattern            |
|---------------|------------|-------------|--------------------|
| Linear Search | 10         | 10          | $O(n)$             |
| Binary Search | 16         | 4           | $O(\log n)$        |
| Bubble Sort   | 5          | 10          | $\frac{n(n-1)}{2}$ |
| Merge Sort    | 8          | 17          | $\approx n \log n$ |

## Complexity Summary

| Algorithm     | Best Case | Worst Case         | Average Case       |
|---------------|-----------|--------------------|--------------------|
| Linear Search | 1         | $n$                | $\frac{n+1}{2}$    |
| Binary Search | 1         | $\log_2 n$         | $\log_2 n - 1$     |
| Bubble Sort   | $n - 1$   | $\frac{n(n-1)}{2}$ | $\frac{n(n-1)}{2}$ |

A Comparison Counter brings complexity theory to life, every `if` becomes a data point, and every loop reveals its true cost.

## 54 Early Termination Heuristic

An Early Termination Heuristic is a strategy to stop an algorithm before full completion when the desired result is already guaranteed or further work won't change the outcome. It's a simple yet powerful optimization that saves time in best and average cases.

### What Problem Are We Solving?

Many algorithms perform redundant work after the solution is effectively found or when additional steps no longer improve results. By detecting these conditions early, we can cut off unnecessary computation, reducing runtime without affecting correctness.

Key question: “*Can we stop now without changing the answer?*”

### How It Works (Plain Language)

1. Identify a stopping condition beyond the usual loop limit.
2. Check at each step if the result is already determined.
3. Exit early when the condition is satisfied.
4. Return partial result if it's guaranteed to be final.

This optimization is common in search, sorting, simulation, and iterative convergence algorithms.

### Example Step by Step

Example 1: Bubble Sort

Normally runs  $n - 1$  passes, even if array sorted early. Add a flag to track swaps; if none occur, terminate.

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

```

    swapped = True
if not swapped:
    break # early termination
return arr

```

Best case: already sorted → 1 pass only

$$T(n) = O(n)$$

Worst case: reversed → still  $O(n^2)$

Example 2: Linear Search

Searching for key  $k$  in array A:

- Stop when found (don't scan full array).
- Average case improves from  $O(n)$  to  $\frac{n}{2}$  comparisons.

Example 3: Convergence Algorithms

In iterative solvers:

- Stop when error < (tolerance threshold).
- Avoids unnecessary extra iterations.

Example 4: Constraint Search

In backtracking or branch-and-bound:

- Stop exploring when solution cannot improve current best.
- Reduces search space dramatically.

## Why It Matters

- Improves average-case performance
- Reduces energy and time in real-world systems
- Maintains correctness (never stops too early)
- Enables graceful degradation for approximate algorithms

## A Gentle Proof (Why It Works)

Let  $f(i)$  represent progress measure after  $i$  iterations. If  $f(i)$  satisfies a stopping invariant  $P$ , then continuing further does not alter the final answer. Thus:

$$\exists i < n; |; P(f(i)) = \text{True} \implies T(n) = i$$

reducing total operations from  $n$  to  $i$  in favorable cases.

## Try It Yourself

1. Add early stop to selection sort (when prefix sorted).
2. Apply tolerance check to Newton's method.
3. Implement linear search with immediate exit.
4. Compare runtime with and without early termination.

## Test Cases

| Algorithm       | Condition        | Best Case            | Worst Case  |
|-----------------|------------------|----------------------|-------------|
| Bubble Sort     | No swaps in pass | $O(n)$               | $O(n^2)$    |
| Linear Search   | Found early      | $O(1)$               | $O(n)$      |
| Newton's Method | \$               | $x_{i+1} - x_i < \$$ | $O(\log n)$ |
| DFS             | Goal found early | $O(d)$               | $O(b^d)$    |

## Complexity Summary

| Case    | Description           | Time                    |
|---------|-----------------------|-------------------------|
| Best    | Early stop triggered  | Reduced from $n$ to $k$ |
| Average | Depends on data order | Often sublinear         |
| Worst   | Condition never met   | Same as original        |

An Early Termination Heuristic adds a simple yet profound optimization, teaching algorithms when to quit, not just how to run.

## 55 Sentinel Technique

The Sentinel Technique is a simple but elegant optimization that eliminates redundant boundary checks in loops by placing a *special marker* (the sentinel) at the end of a data structure. It's a subtle trick that makes code faster, cleaner, and safer.

### What Problem Are We Solving?

In many algorithms, especially search and scanning loops, we repeatedly check for two things:

1. Whether the element matches a target
2. Whether we've reached the end of the structure

This double condition costs extra comparisons every iteration. By adding a sentinel value, we can guarantee termination and remove one check.

## How It Works (Plain Language)

1. Append a sentinel value (e.g. target or infinity) to the end of the array.
2. Loop until match found, without checking bounds.
3. Stop automatically when you hit the sentinel.
4. Check afterward if the match was real or sentinel-triggered.

This replaces:

```
while i < n and A[i] != key:  
    i += 1
```

with a simpler loop:

```
A[n] = key  
while A[i] != key:  
    i += 1
```

No more bound check inside the loop.

## Example Step by Step

Example 1: Linear Search with Sentinel

Without sentinel:

```
def linear_search(A, key):  
    for i in range(len(A)):  
        if A[i] == key:  
            return i  
    return -1
```

Every step checks both conditions.

With sentinel:

```

def linear_search_sentinel(A, key):
    n = len(A)
    A.append(key)  # add sentinel
    i = 0
    while A[i] != key:
        i += 1
    return i if i < n else -1

```

- Only one condition inside loop
- Works for both found and not-found cases

Cost Reduction: from  $2n+1$  comparisons to  $n+1$

Example 2: Merging Sorted Lists

Add infinity sentinel at the end of each list:

- Prevents repeated end-of-array checks
- Simplifies inner loop logic

E.g. in Merge Sort, use sentinel values to avoid `if i < n` checks.

Example 3: String Parsing

Append '\0' (null terminator) so loops can stop automatically on sentinel. Used widely in C strings.

## Why It Matters

- Removes redundant checks
- Simplifies loop logic
- Improves efficiency and readability
- Common in systems programming, parsing, searching

## A Gentle Proof (Why It Works)

Let  $n$  be array length. Normally, each iteration does:

- 1 comparison with bound
- 1 comparison with key

So total  $\approx 2n + 1$  comparisons.

With sentinel:

- 1 comparison per element
- 1 final check after loop

So total  $\approx n + 1$

Improvement factor  $\sim 2 \times$  speedup for long lists.

### Try It Yourself

1. Implement sentinel linear search and count comparisons.
2. Add infinity sentinel in merge routine.
3. Write a parser that stops on sentinel '\0'.
4. Compare runtime vs standard implementation.

### Test Cases

| Input        | Key | Output | Comparisons |
|--------------|-----|--------|-------------|
| [1,2,3,4], 3 | 3   | 2      | 3           |
| [1,2,3,4], 5 | -1  | 4      | 5           |
| [ ], 1       | -1  | 0      | 1           |

### Complexity Summary

| Case        | Time                              | Space       | Notes                    |
|-------------|-----------------------------------|-------------|--------------------------|
| Best        | $O(1)$                            | $O(1)$      | Found immediately        |
| Worst       | $O(n)$                            | $O(1)$      | Found at end / not found |
| Improvement | $\sim 2 \times$ fewer comparisons | +1 sentinel | Always safe              |

The Sentinel Technique is a quiet masterpiece of algorithmic design, proving that sometimes, one tiny marker can make a big difference.

## 56 Binary Predicate Tester

A Binary Predicate Tester is a simple yet fundamental tool for checking whether a condition involving two operands holds true, a building block for comparisons, ordering, filtering, and search logic across algorithms. It clarifies logic and promotes reuse by abstracting condition checks.

## What Problem Are We Solving?

Every algorithm depends on decisions, “Is this element smaller?”, “Are these two equal?”, “Does this satisfy the constraint?”. These yes/no questions are binary predicates: functions that return either `True` or `False`.

By formalizing them as reusable testers, we gain:

- Clarity, separate logic from control flow
- Reusability, pass as arguments to algorithms
- Flexibility, easily switch from `<` to `>` or `==`

This underlies sorting, searching, and functional-style algorithms.

## How It Works (Plain Language)

1. Define a predicate function that takes two arguments.
2. Returns `True` if condition satisfied, `False` otherwise.
3. Use the predicate inside loops, filters, or algorithmic decisions.
4. Swap out predicates to change algorithm behavior dynamically.

Predicates serve as the comparison layer, they don’t control flow, but inform it.

## Example Step by Step

Example 1: Sorting by Predicate

Define different predicates:

```
def less(a, b): return a < b
def greater(a, b): return a > b
def equal(a, b): return a == b
```

Pass to sorting routine:

```
def compare_sort(arr, predicate):
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if predicate(arr[j + 1], arr[j]):
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
```

Now you can sort ascending or descending just by changing the predicate.

#### Example 2: Binary Search Condition

Binary search relies on predicate `is_less(mid_value, key)` to decide direction:

```
def is_less(a, b): return a < b
```

So the decision step becomes:

```
if is_less(arr[mid], key):
    left = mid + 1
else:
    right = mid - 1
```

This makes the comparison logic explicit, not buried inside control.

#### Example 3: Filtering or Matching

```
def between(a, b): return a < b
filtered = [x for x in data if between(x, 10)]
```

Easily swap predicates for greater-than or equality checks.

## Why It Matters

- Encapsulates decision logic cleanly
- Enables higher-order algorithms (pass functions as arguments)
- Simplifies testing and customization
- Core to generic programming and templates (C++, Python `key` functions)

## A Gentle Proof (Why It Works)

Predicates abstract the notion of ordering or relation. If a predicate satisfies:

- Reflexivity ( $P(x, x) = \text{False or True}$ , as defined)
- Antisymmetry ( $P(a, b) \Rightarrow \neg P(b, a)$ )
- Transitivity ( $(P(a, b) \wedge P(b, c)) \Rightarrow P(a, c)$ )

then it defines a strict weak ordering, sufficient for sorting and searching algorithms.

Thus, correctness of algorithms depends on predicate consistency.

## Try It Yourself

1. Write predicates for `<`, `>`, `==`, and `divisible(a,b)`.
2. Use them in a selection algorithm.
3. Test sorting ascending and descending using same code.
4. Verify predicate correctness (antisymmetry, transitivity).

## Test Cases

| Predicate | a | b | Result | Meaning       |
|-----------|---|---|--------|---------------|
| less      | 3 | 5 | True   | $3 < 5$       |
| greater   | 7 | 2 | True   | $7 > 2$       |
| equal     | 4 | 4 | True   | $4 == 4$      |
| divisible | 6 | 3 | True   | $6 \% 3 == 0$ |

## Complexity Summary

| Operation                 | Time                 | Space  | Notes               |
|---------------------------|----------------------|--------|---------------------|
| Predicate call            | $O(1)$               | $O(1)$ | Constant per check  |
| Algorithm using predicate | Depends on structure | ,      | e.g. sort: $O(n^2)$ |

A Binary Predicate Tester turns hidden conditions into visible design, clarifying logic, encouraging reuse, and laying the foundation for generic algorithms that *think in relationships*, not instructions.

## 57 Range Test Function

A Range Test Function checks whether a given value lies within specified bounds, a universal operation in algorithms that handle intervals, array indices, numeric domains, or search constraints. It's small but powerful, providing correctness and safety across countless applications.

### What Problem Are We Solving?

Many algorithms operate on ranges, whether scanning arrays, iterating loops, searching intervals, or enforcing constraints. Repeatedly checking `if low <= x <= high` can clutter code and lead to subtle off-by-one errors.

By defining a reusable range test, we make such checks:

- Centralized (one definition, consistent semantics)
- Readable (intent clear at call site)
- Safe (avoid inconsistent inequalities)

## How It Works (Plain Language)

1. Encapsulate the boundary logic into a single function.
2. Input: a value  $x$  and bounds (`low, high`).
3. Return: `True` if  $x$  satisfies range condition, else `False`.
4. Can handle open, closed, or half-open intervals.

Variants:

- Closed:  $[\text{low}, \text{high}] \rightarrow \text{low} \leq x \leq \text{high}$
- Half-open:  $[\text{low}, \text{high}) \rightarrow \text{low} \leq x < \text{high}$
- Open:  $(\text{low}, \text{high}) \rightarrow \text{low} < x < \text{high}$

## Example Step by Step

Example 1: Array Index Bounds

Prevent out-of-bounds access:

```
def in_bounds(i, n):
    return 0 <= i < n

if in_bounds(idx, len(arr)):
    value = arr[idx]
```

No more manual range logic.

Example 2: Range Filtering

Filter values inside range  $[a, b]$ :

```
def in_range(x, low, high):
    return low <= x <= high

data = [1, 3, 5, 7, 9]
filtered = [x for x in data if in_range(x, 3, 7)]
# → [3, 5, 7]
```

### Example 3: Constraint Checking

Used in search or optimization algorithms:

```
if not in_range(candidate, min_val, max_val):
    continue # skip invalid candidate
```

Keeps loops clean and avoids boundary bugs.

### Example 4: Geometry / Interval Problems

Check interval overlap:

```
def overlap(a1, a2, b1, b2):
    return in_range(a1, b1, b2) or in_range(b1, a1, a2)
```

## Why It Matters

- Prevents off-by-one errors
- Improves code clarity and consistency
- Essential in loop guards, search boundaries, and validity checks
- Enables parameter validation and defensive programming

## A Gentle Proof (Why It Works)

Range test expresses a logical conjunction:

$$P(x) = (x \geq \text{low}) \wedge (x \leq \text{high})$$

For closed intervals, the predicate is reflexive and transitive within the set  $[\text{low}, \text{high}]$ . By encoding this predicate as a function, correctness follows from elementary properties of inequalities.

Half-open variants preserve well-defined iteration bounds (important for array indices).

## Try It Yourself

1. Implement `in_open_range(x, low, high)` for  $(\text{low}, \text{high})$ .
2. Write `in_half_open_range(i, 0, n)` for loops.
3. Use range test in binary search termination condition.
4. Check index validity in matrix traversal.

## Test Cases

| Input | Range   | Type      | Result |
|-------|---------|-----------|--------|
| 5     | [1, 10] | Closed    | True   |
| 10    | [1, 10) | Half-open | False  |
| 0     | (0, 5)  | Open      | False  |
| 3     | [0, 3]  | Closed    | True   |

## Complexity Summary

| Operation     | Time   | Space  | Notes                    |
|---------------|--------|--------|--------------------------|
| Range check   | $O(1)$ | $O(1)$ | Constant-time comparison |
| Used per loop | $O(n)$ | $O(1)$ | Linear overall           |

A Range Test Function is a tiny guardrail with big impact, protecting correctness at every boundary and making algorithms easier to reason about.

## 58 Search Invariant Checker

A Search Invariant Checker ensures that key conditions (invariants) hold throughout a search algorithm's execution. By maintaining these invariants, we guarantee correctness, prevent subtle bugs, and provide a foundation for proofs and reasoning.

### What Problem Are We Solving?

When performing iterative searches (like binary search or interpolation search), we maintain certain truths that must always hold, such as:

- The target, if it exists, is always within the current bounds.
- The search interval shrinks every step.
- Indices remain valid and ordered.

Losing these invariants can lead to infinite loops, incorrect results, or index errors. By explicitly checking invariants, we make correctness visible and testable.

## How It Works (Plain Language)

1. Define invariants, conditions that must stay true during every iteration.
2. After each update step, verify these conditions.
3. If an invariant fails, assert or log an error.
4. Use invariants both for debugging and proofs.

Common search invariants:

- \$ low high \$
- \$ target [low, high] \$
- Interval size decreases: \$ (high - low) \$ shrinks each step

## Example Step by Step

Example: Binary Search Invariants

Goal: Maintain correct search window in [low, high].

1. Initialization: \$ low = 0 \$, \$ high = n - 1 \$
2. Invariant 1: \$ target [low, high] \$
3. Invariant 2: \$ low high \$
4. Step: Compute mid, narrow range
5. Check: Each iteration, assert these invariants

## Tiny Code (Python)

```
def binary_search(arr, target):
    low, high = 0, len(arr) - 1
    while low <= high:
        assert 0 <= low <= high < len(arr), "Invariant broken!"
        mid = (low + high) // 2

        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

If the invariant fails, we catch logic errors early.

## Why It Matters

- Proof of correctness: Each iteration preserves truth
- Debugging aid: Detect logic flaws immediately
- Safety guarantee: Prevent invalid access or infinite loops
- Documentation: Clarifies algorithm intent

## A Gentle Proof (Why It Works)

Suppose invariant  $P$  holds before iteration. The update step transforms state  $(\text{low}, \text{high})$  to  $(\text{low}', \text{high}')$ .

We prove:

1. Base Case:  $P$  holds before first iteration (initialization)
2. Inductive Step: If  $P$  holds before iteration, and update rules maintain  $P$ , then  $P$  holds afterward

Hence, by induction,  $P$  always holds. This ensures algorithm correctness.

## Try It Yourself

1. Add invariants to ternary search
2. Prove binary search correctness using invariant preservation
3. Test boundary cases (empty array, one element)
4. Visualize shrinking interval and check invariant truth at each step

## Test Cases

| Input Array     | Target | Invariants Hold | Result    |
|-----------------|--------|-----------------|-----------|
| [1, 3, 5, 7, 9] | 5      | Yes             | Index 2   |
| [2, 4, 6]       | 3      | Yes             | Not found |
| [1]             | 1      | Yes             | Index 0   |
| []              | 10     | Yes             | Not found |

## Complexity

| Operation       | Time        | Space  | Notes                 |
|-----------------|-------------|--------|-----------------------|
| Check invariant | $O(1)$      | $O(1)$ | Constant-time check   |
| Total search    | $O(\log n)$ | $O(1)$ | Preserves correctness |

The Search Invariant Checker turns implicit assumptions into explicit guarantees, making your search algorithms not only fast but provably correct.

## 59 Probe Counter

A Probe Counter tracks how many probes or lookup attempts a search algorithm performs. It's a diagnostic tool to understand efficiency and compare performance between different search strategies or data structures.

### What Problem Are We Solving?

In searching (especially in hash tables, linear probing, or open addressing), performance depends not just on complexity but on how many probes are required to find or miss an element.

By counting probes, we:

- Reveal the cost of each search
- Compare performance under different load factors
- Diagnose clustering or inefficient probing patterns

### How It Works (Plain Language)

1. Initialize a counter `probes` = 0.
2. Each time the algorithm checks a position or node, increment `probes`.
3. When the search ends, record or return the probe count.
4. Use statistics (mean, max, variance) to measure performance.

### Example Step by Step

Example: Linear Probing in a Hash Table

1. Compute hash:  $h = \text{key mod } m$
2. Start at  $h$ , check slot
3. If collision, move to next slot
4. Increment `probes` each time
5. Stop when slot is empty or key is found

If the table is nearly full, probe count increases, revealing efficiency loss.

### Tiny Code (Python)

```
def linear_probe_search(table, key):
    m = len(table)
    h = key % m
    probes = 0
    i = 0

    while table[(h + i) % m] is not None:
        probes += 1
        if table[(h + i) % m] == key:
            return (h + i) % m, probes
        i += 1
        if i == m:
            break # table full
    return None, probes
```

Example run:

```
table = [10, 21, 32, None, None]
index, probes = linear_probe_search(table, 21)
# probes = 1
```

### Why It Matters

- Performance insight: Understand search cost beyond asymptotics
- Clustering detection: Reveal poor distribution or collisions
- Load factor tuning: Find thresholds before degradation
- Algorithm comparison: Evaluate quadratic vs linear probing

### A Gentle Proof (Why It Works)

Let  $L$  be the load factor (fraction of table filled). Expected probes for a successful search in linear probing:

$$E[P_{\text{success}}] = \frac{1}{2} \left( 1 + \frac{1}{1-L} \right)$$

Expected probes for unsuccessful search:

$$E[P_{\text{fail}}] = \frac{1}{2} \left( 1 + \frac{1}{(1-L)^2} \right)$$

As  $L \rightarrow 1$ , probe counts grow rapidly, performance decays.

### Try It Yourself

1. Create a hash table with linear probing
2. Insert keys at different load factors
3. Measure probe counts for hits and misses
4. Compare linear vs quadratic probing

### Test Cases

| Table (size 7)        | Key | Load Factor | Expected Probes | Notes              |
|-----------------------|-----|-------------|-----------------|--------------------|
| [10, 21, 32, None...] | 21  | 0.4         | 1               | Direct hit         |
| [10, 21, 32, 43, 54]  | 43  | 0.7         | 3               | Clustered region   |
| [10, 21, 32, 43, 54]  | 99  | 0.7         | 5               | Miss after probing |

### Complexity

| Operation    | Time (Expected) | Time (Worst) | Space  |
|--------------|-----------------|--------------|--------|
| Probe count  | $O(1)$ per step | $O(n)$       | $O(1)$ |
| Total search | $O(1)$ average  | $O(n)$       | $O(1)$ |

By counting probes, we move from theory to measured understanding, a simple metric that reveals the hidden costs behind collisions, load factors, and search efficiency.

### 60 Cost Curve Plotter

A Cost Curve Plotter visualizes how an algorithm's running cost grows as the input size increases. It turns abstract complexity into a tangible curve, helping you compare theoretical and empirical performance side by side.

## What Problem Are We Solving?

Big-O notation tells us how cost scales, but not how much or where performance starts to break down. A cost curve lets you:

- See real growth vs theoretical models
- Identify crossover points between algorithms
- Detect anomalies or overhead
- Build intuition about efficiency and scaling

## How It Works (Plain Language)

1. Choose an algorithm and a range of input sizes.
2. For each  $n$ , run the algorithm and record:
  - Time cost (runtime)
  - Space cost (memory usage)
  - Operation count
3. Plot  $(n, \text{cost}(n))$  points
4. Overlay theoretical curves ( $O(n)$ ,  $O(n \log n)$ ,  $O(n^2)$ ) for comparison

This creates a visual map of performance over scale.

## Example Step by Step

Let's measure sorting cost for different input sizes:

| n    | Time (ms) |
|------|-----------|
| 100  | 0.3       |
| 500  | 2.5       |
| 1000 | 5.2       |
| 2000 | 11.3      |
| 4000 | 23.7      |

Plot these points. The curve shape suggests  $O(n \log n)$  behavior.

## Tiny Code (Python + Matplotlib)

```
import time, random, matplotlib.pyplot as plt

def measure_cost(algorithm, sizes):
    results = []
    for n in sizes:
        arr = [random.randint(0, 100000) for _ in range(n)]
        start = time.time()
        algorithm(arr)
        end = time.time()
        results.append((n, end - start))
    return results

def plot_cost_curve(results):
    xs, ys = zip(*results)
    plt.plot(xs, ys, marker='o')
    plt.xlabel("Input size (n)")
    plt.ylabel("Time (seconds)")
    plt.title("Algorithm Cost Curve")
    plt.grid(True)
    plt.show()
```

## Why It Matters

- Brings Big-O to life
- Visual debugging, detect unexpected spikes
- Compare algorithms empirically
- Tune thresholds, know when to switch strategies

## A Gentle Proof (Why It Works)

If theoretical cost is  $f(n)$  and empirical cost is  $g(n)$ , then we expect:

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = c$$

where  $c$  is a constant scaling factor.

The plotted curve visually approximates  $g(n)$ ; comparing its shape to  $f(n)$  reveals whether the complexity class matches expectations.

## Try It Yourself

1. Compare bubble sort vs merge sort vs quicksort.
2. Overlay  $n$ ,  $n \log n$ , and  $n^2$  reference curves.
3. Experiment with different data distributions (sorted, reversed).
4. Plot both time and memory cost curves.

## Test Cases

| Algorithm   | Input Size | Time (ms) | Shape        | Match         |
|-------------|------------|-----------|--------------|---------------|
| Bubble Sort | 1000       | 80        | Quadratic    | $O(n^2)$      |
| Merge Sort  | 1000       | 5         | Linearithmic | $O(n \log n)$ |
| Quick Sort  | 1000       | 3         | Linearithmic | $O(n \log n)$ |

## Complexity

| Aspect      | Cost              | Notes                      |
|-------------|-------------------|----------------------------|
| Measurement | $O(k \cdot T(n))$ | $k$ sample sizes measured  |
| Plotting    | $O(k)$            | Draw curve from $k$ points |
| Space       | $O(k)$            | Store measurement data     |

The Cost Curve Plotter turns theory into shape, a simple graph that makes scaling behavior and trade-offs instantly clear.

## Section 7. Sorting basics

### 61 Swap Counter

A Swap Counter tracks the number of element swaps performed during a sorting process. It helps us understand how much rearrangement an algorithm performs and serves as a diagnostic for efficiency, stability, and input sensitivity.

## What Problem Are We Solving?

Many sorting algorithms (like Bubble Sort, Selection Sort, or Quick Sort) rearrange elements through swaps. Counting swaps shows how “active” the algorithm is:

- Bubble Sort → high swap count
- Insertion Sort → fewer swaps on nearly sorted input
- Selection Sort → fixed number of swaps

By tracking swaps, we compare algorithms on data movement cost, not just comparisons.

## How It Works (Plain Language)

1. Initialize a `swap_count = 0`.
2. Each time two elements exchange positions, increment the counter.
3. At the end, report `swap_count` to measure rearrangement effort.
4. Use results to compare sorting strategies or analyze input patterns.

## Example Step by Step

Example: Bubble Sort on [3, 2, 1]

1. Compare 3 and 2 → swap → count = 1 → [2, 3, 1]
2. Compare 3 and 1 → swap → count = 2 → [2, 1, 3]
3. Compare 2 and 1 → swap → count = 3 → [1, 2, 3]

Total swaps: 3

If input is [1, 2, 3], no swaps occur, cost reflects sortedness.

## Tiny Code (Python)

```
def bubble_sort_with_swaps(arr):  
    n = len(arr)  
    swaps = 0  
    for i in range(n):  
        for j in range(0, n - i - 1):  
            if arr[j] > arr[j + 1]:  
                arr[j], arr[j + 1] = arr[j + 1], arr[j]  
                swaps += 1  
    return arr, swaps
```

Example:

```
arr, swaps = bubble_sort_with_swaps([3, 2, 1])
# swaps = 3
```

## Why It Matters

- Quantifies data movement cost
- Measures input disorder (zero swaps  $\rightarrow$  already sorted)
- Compares algorithms on swap efficiency
- Reveals adaptive behavior in real data

## A Gentle Proof (Why It Works)

Every swap reduces the inversion count by one. An inversion is a pair  $(i, j)$  such that  $i < j$  and  $a_i > a_j$ .

If initial inversion count =  $I$ , and each swap fixes one inversion:

$$\text{Total Swaps} = I_{\text{initial}}$$

Thus, swap count directly equals disorder measure, a meaningful cost metric.

## Try It Yourself

1. Count swaps for Bubble Sort, Insertion Sort, and Selection Sort.
2. Run on sorted, reversed, and random lists.
3. Compare counts, which adapts best to nearly sorted data?
4. Plot swap count vs input size.

## Test Cases

| Input     | Algorithm      | Swaps | Observation             |
|-----------|----------------|-------|-------------------------|
| [3, 2, 1] | Bubble Sort    | 3     | Full reversal           |
| [1, 2, 3] | Bubble Sort    | 0     | Already sorted          |
| [2, 3, 1] | Insertion Sort | 2     | Moves minimal elements  |
| [3, 1, 2] | Selection Sort | 2     | Swaps once per position |

## Complexity

| Metric          | Cost     | Notes                      |
|-----------------|----------|----------------------------|
| Time (Tracking) | $O(1)$   | Increment counter per swap |
| Total Swaps     | $O(n^2)$ | Worst case for Bubble Sort |
| Space           | $O(1)$   | Constant extra memory      |

A Swap Counter offers a clear window into sorting dynamics, revealing how “hard” the algorithm works and how far the input is from order.

## 62 Inversion Counter

An Inversion Counter measures how far a sequence is from being sorted by counting all pairs that are out of order. It’s a numerical measure of disorder, zero for a sorted list, maximum for a fully reversed one.

### What Problem Are We Solving?

Sorting algorithms fix *inversions*. Each inversion is a pair  $(i, j)$  such that  $i < j$  and  $a_i > a_j$ . Counting inversions gives us:

- A quantitative measure of unsortedness
- A way to analyze algorithm progress
- Insight into best-case vs worst-case behavior

This metric is also used in Kendall tau distance, ranking comparisons, and adaptive sorting research.

### How It Works (Plain Language)

1. Take an array  $A = [a_1, a_2, \dots, a_n]$ .
2. For each pair  $(i, j)$  where  $i < j$ , check if  $a_i > a_j$ .
3. Increment count for each inversion found.
4. A sorted array has 0 inversions; a reversed one has  $\frac{n(n-1)}{2}$ .

## Example Step by Step

Array: [3, 1, 2]

- (3, 1): inversion
- (3, 2): inversion
- (1, 2): no inversion

Total inversions: 2

A perfect diagnostic: small count → nearly sorted.

## Tiny Code (Brute Force)

```
def count_inversions_bruteforce(arr):
    count = 0
    n = len(arr)
    for i in range(n):
        for j in range(i + 1, n):
            if arr[i] > arr[j]:
                count += 1
    return count
```

Output: `count_inversions_bruteforce([3, 1, 2]) → 2`

## Optimized Approach (Merge Sort)

Counting inversions can be done in  $O(n \log n)$  by modifying merge sort.

```
def count_inversions_merge(arr):
    def merge_count(left, right):
        i = j = inv = 0
        merged = []
        while i < len(left) and j < len(right):
            if left[i] <= right[j]:
                merged.append(left[i])
                i += 1
            else:
                merged.append(right[j])
                inv += len(left) - i
                j += 1
        return merged, inv
    if len(arr) < 2:
        return arr, 0
    mid = len(arr) // 2
    left, inv1 = merge_count(arr[:mid], arr[mid:])
    right, inv2 = merge_count(arr[mid:], arr[:mid])
    merged, inv3 = merge_count(left, right)
    return merged, inv1 + inv2 + inv3
```

```

merged += left[i:]
merged += right[j:]
return merged, inv

def sort_count(sub):
    if len(sub) <= 1:
        return sub, 0
    mid = len(sub) // 2
    left, invL = sort_count(sub[:mid])
    right, invR = sort_count(sub[mid:])
    merged, invM = merge_count(left, right)
    return merged, invL + invR + invM

_, total = sort_count(arr)
return total

```

Result:  $O(n \log n)$  instead of  $O(n^2)$ .

## Why It Matters

- Quantifies disorder precisely
- Used in sorting network analysis
- Predicts best-case improvements for adaptive sorts
- Connects to ranking correlation metrics

## A Gentle Proof (Why It Works)

Every swap in a stable sort fixes exactly one inversion. If we let  $I$  denote total inversions:

$$I_{\text{sorted}} = 0, \quad I_{\text{reverse}} = \frac{n(n-1)}{2}$$

Hence, inversion count measures *distance to sorted order*, a lower bound on swaps needed by any comparison sort.

## Try It Yourself

1. Count inversions for sorted, reversed, and random arrays.
2. Plot inversion count vs swap count.
3. Test merge sort counter vs brute force counter.
4. Measure how inversion count affects adaptive algorithms.

## Test Cases

| Input     | Inversions | Interpretation         |
|-----------|------------|------------------------|
| [1, 2, 3] | 0          | Already sorted         |
| [3, 2, 1] | 3          | Fully reversed         |
| [2, 3, 1] | 2          | Two pairs out of order |
| [1, 3, 2] | 1          | Slightly unsorted      |

## Complexity

| Method           | Time          | Space  | Notes                      |
|------------------|---------------|--------|----------------------------|
| Brute Force      | $O(n^2)$      | $O(1)$ | Simple but slow            |
| Merge Sort Based | $O(n \log n)$ | $O(n)$ | Efficient for large arrays |

An Inversion Counter transforms “how sorted is this list?” into a precise number, perfect for analysis, comparison, and designing smarter sorting algorithms.

## 63 Stability Checker

A Stability Checker verifies whether a sorting algorithm preserves the relative order of equal elements. Stability is essential when sorting complex records with multiple keys, ensuring secondary attributes remain in order after sorting by a primary one.

### What Problem Are We Solving?

When sorting, sometimes values tie, they’re equal under the primary key. A stable sort keeps these tied elements in their original order. For example, sorting students by grade while preserving the order of names entered earlier.

Without stability, sorting by multiple keys becomes error-prone, and chained sorts may lose meaning.

## How It Works (Plain Language)

1. Label each element with its original position.
2. Perform the sort.
3. After sorting, for all pairs with equal keys, check if the original indices remain in ascending order.
4. If yes, the algorithm is stable. Otherwise, it's not.

## Example Step by Step

Array with labels: [(A, 3), (B, 1), (C, 3)] Sort by value → [(B, 1), (A, 3), (C, 3)]

Check ties:

- Elements with value 3: A before C, and A's original index < C's original index → stable.

If result was [(B, 1), (C, 3), (A, 3)], order of equals reversed → unstable.

## Tiny Code (Python)

```
def is_stable_sort(original, sorted_arr, key=lambda x: x):  
    positions = {}  
    for idx, val in enumerate(original):  
        positions.setdefault(key(val), []).append(idx)  
  
    last_seen = {}  
    for val in sorted_arr:  
        k = key(val)  
        pos = positions[k].pop(0)  
        if k in last_seen and last_seen[k] > pos:  
            return False  
        last_seen[k] = pos  
    return True
```

Usage:

```
data = [('A', 3), ('B', 1), ('C', 3)]  
sorted_data = sorted(data, key=lambda x: x[1])  
is_stable_sort(data, sorted_data, key=lambda x: x[1]) # True
```

## Why It Matters

- Preserves secondary order: essential for multi-key sorts
- Chaining safety: sort by multiple fields step-by-step
- Predictable results: avoids random reorder of equals
- Common property: Merge Sort, Insertion Sort stable; Quick Sort not (by default)

## A Gentle Proof (Why It Works)

Let  $a_i$  and  $a_j$  be elements with equal keys  $k$ . If  $i < j$  in the input and positions of  $a_i$  and  $a_j$  after sorting are  $p_i$  and  $p_j$ , then the algorithm is stable if and only if:

$$i < j \implies p_i < p_j \text{ whenever } k_i = k_j$$

Checking this property across all tied keys confirms stability.

## Try It Yourself

1. Compare stable sort (Merge Sort) vs unstable sort (Selection Sort).
2. Sort list of tuples by one key, check tie preservation.
3. Chain sorts (first by last name, then by first name).
4. Run checker to confirm final stability.

## Test Cases

| Input                    | Sorted Result            | Stable? | Explanation             |
|--------------------------|--------------------------|---------|-------------------------|
| $[(A,3),(B,1),(C,3)]$    | $[(B,1),(A,3),(C,3)]$    | Yes     | A before C preserved    |
| $[(A,3),(B,1),(C,3)]$    | $[(B,1),(C,3),(A,3)]$    | No      | A and C order reversed  |
| $[(1,10),(2,10),(3,10)]$ | $[(1,10),(2,10),(3,10)]$ | Yes     | All tied, all preserved |

## Complexity

| Operation | Time    | Space  | Notes                       |
|-----------|---------|--------|-----------------------------|
| Checking  | $O(n)$  | $O(n)$ | One pass over sorted array  |
| Sorting   | Depends | ,      | Checker independent of sort |

The Stability Checker ensures your sorts respect order among equals, a small step that safeguards multi-key sorting correctness and interpretability.

## 64 Comparison Network Visualizer

A Comparison Network Visualizer shows how fixed sequences of comparisons sort elements, revealing the structure of sorting networks. These diagrams help us see how parallel sorting works, step by step, independent of input data.

### What Problem Are We Solving?

Sorting networks are data-oblivious, their comparison sequence is fixed, not driven by data. To understand or design them, we need a clear visual of which elements compare and when. The visualizer turns an abstract sequence of comparisons into a layered network diagram.

This is key for:

- Analyzing parallel sorting
- Designing hardware-based sorters
- Studying bitonic or odd-even merges

### How It Works (Plain Language)

1. Represent each element as a horizontal wire.
2. Draw a vertical comparator line connecting the two wires being compared.
3. Group comparators into layers that can run in parallel.
4. The network executes layer by layer, swapping elements if out of order.

Result: a visual map of sorting logic.

### Example Step by Step

Sorting 4 elements with Bitonic Sort network:

Layer 1: Compare (0,1), (2,3)  
Layer 2: Compare (0,2), (1,3)  
Layer 3: Compare (1,2)

Visual:

0  
1  
2  
3

Each dot pair = comparator. The structure is static, independent of values.

### Tiny Code (Python)

```
def visualize_network(n, layers):
    wires = [[' '] * (len(layers) + 1) for _ in range(n)]

    for layer_idx, layer in enumerate(layers):
        for i, j in layer:
            wires[i][layer_idx] = ' '
            wires[j][layer_idx] = ' '

    for i in range(n):
        print(f'{i}: ' + ''.join(wires[i]))

layers = [[(0,1), (2,3)], [(0,2), (1,3)], [(1,2)]]
visualize_network(4, layers)
```

This prints a symbolic visualization of comparator layers.

### Why It Matters

- Reveals parallelism in sorting logic
- Helps debug data-oblivious algorithms
- Useful for hardware and GPU design
- Foundation for Bitonic, Odd-Even Merge, and Batcher networks

### A Gentle Proof (Why It Works)

A sorting network guarantees correctness if it sorts all binary sequences of length  $n$ .

By the Zero-One Principle:

If a comparison network correctly sorts all sequences of 0s and 1s, it correctly sorts all sequences of arbitrary numbers.

So visualizing comparators ensures completeness and layer correctness.

## Try It Yourself

1. Draw a 4-input bitonic sorting network.
2. Visualize how comparators “flow” through layers.
3. Check how many layers can run in parallel.
4. Test sorting 0/1 sequences manually through the network.

## Test Cases

| Inputs    | Network Type   | Layers | Sorted Output |
|-----------|----------------|--------|---------------|
| [3,1,4,2] | Bitonic Sort   | 3      | [1,2,3,4]     |
| [1,0,1,0] | Odd-Even Merge | 3      | [0,0,1,1]     |

## Complexity

| Metric      | Value           | Notes                        |
|-------------|-----------------|------------------------------|
| Comparators | $O(n \log^2 n)$ | Batcher's network complexity |
| Depth       | $O(\log^2 n)$   | Layers executed in parallel  |
| Space       | $O(n)$          | One wire per input           |

A Comparison Network Visualizer makes parallel sorting tangible, every comparator and layer visible, transforming abstract hardware logic into a clear, educational blueprint.

## 65 Adaptive Sort Detector

An Adaptive Sort Detector measures how “sorted” an input sequence already is and predicts whether an algorithm can take advantage of it. It’s a diagnostic tool that estimates presortedness and guides the choice of an adaptive sorting algorithm.

### What Problem Are We Solving?

Not all inputs are random, many are partially sorted. Some algorithms (like Insertion Sort or Timsort) perform much faster on nearly sorted data. We need a way to detect sortedness before choosing the right strategy.

An adaptive detector quantifies how close an input is to sorted order.

## How It Works (Plain Language)

1. Define a measure of disorder (e.g., number of inversions, runs, or local misplacements).
2. Traverse the array, counting indicators of unsortedness.
3. Return a metric (e.g., 0 = fully sorted, 1 = fully reversed).
4. Use this score to decide whether to apply:
  - Simple insertion-like sort (for nearly sorted data)
  - General-purpose sort (for random data)

## Example Step by Step

Array: [1, 2, 4, 3, 5, 6]

1. Compare adjacent pairs:

- 1 2 (ok)
- 2 4 (ok)
- 4 > 3 (disorder)
- 3 5 (ok)
- 5 6 (ok)

2. Count = 1 local inversion

Sortedness score:

$$s = 1 - \frac{\text{disorder}}{n-1} = 1 - \frac{1}{5} = 0.8$$

80% sorted, good candidate for adaptive sort.

## Tiny Code (Python)

```
def adaptive_sort_detector(arr):  
    disorder = 0  
    for i in range(len(arr) - 1):  
        if arr[i] > arr[i + 1]:  
            disorder += 1  
    return 1 - disorder / max(1, len(arr) - 1)  
  
arr = [1, 2, 4, 3, 5, 6]
```

```
score = adaptive_sort_detector(arr)
# score = 0.8
```

You can use this score to select algorithms dynamically.

## Why It Matters

- Detects near-sorted input efficiently
- Enables algorithm selection at runtime
- Saves time on real-world data (logs, streams, merges)
- Core idea behind Timsort's run detection

## A Gentle Proof (Why It Works)

If an algorithm's time complexity depends on disorder  $d$ , e.g.  $O(n + d)$ , and  $d = O(1)$  for nearly sorted arrays, then the adaptive algorithm approaches linear time.

The detector approximates  $d$ , helping us decide when  $O(n + d)$  beats  $O(n \log n)$ .

## Try It Yourself

1. Test arrays with 0, 10%, 50%, and 100% disorder.
2. Compare runtime of Insertion Sort vs Merge Sort.
3. Use inversion counting for more precise detection.
4. Integrate detector into a hybrid sorting routine.

## Test Cases

| Input       | Disorder | Score | Recommendation     |
|-------------|----------|-------|--------------------|
| [1,2,3,4,5] | 0        | 1.0   | Insertion Sort     |
| [1,3,2,4,5] | 1        | 0.8   | Adaptive Sort      |
| [3,2,1]     | 2        | 0.0   | Merge / Quick Sort |
| [2,1,3,5,4] | 2        | 0.6   | Adaptive Sort      |

## Complexity

| Operation        | Time     | Space  | Notes                         |
|------------------|----------|--------|-------------------------------|
| Disorder check   | $O(n)$   | $O(1)$ | Single scan                   |
| Sorting (chosen) | Adaptive | ,      | Depends on algorithm selected |

The Adaptive Sort Detector bridges theory and pragmatism, quantifying how ordered your data is and guiding smarter algorithm choices for real-world performance.

## 66 Sorting Invariant Checker

A Sorting Invariant Checker verifies that key ordering conditions hold throughout a sorting algorithm's execution. It's used to reason about correctness step by step, ensuring that each iteration preserves progress toward a fully sorted array.

### What Problem Are We Solving?

When debugging or proving correctness of sorting algorithms, we need to ensure that certain invariants (conditions that must always hold) remain true. If any invariant breaks, the algorithm may produce incorrect output, even if it "looks" right at a glance.

A sorting invariant formalizes what "partial progress" means. Examples:

- "All elements before index  $i$  are in sorted order."
- "All elements beyond pivot are greater or equal to it."
- "Heap property holds at every node."

### How It Works (Plain Language)

1. Define one or more invariants that describe correctness.
2. After each iteration or recursion step, check that these invariants still hold.
3. If any fail, stop and debug, the algorithm logic is wrong.
4. Once sorting finishes, the global invariant (sorted array) must hold.

This approach is key for formal verification and debuggable code.

## Example Step by Step

Insertion Sort invariant:

Before processing element  $i$ , the subarray  $\text{arr}[:i]$  is sorted.

- Initially  $i = 1$ : subarray  $[\text{arr}[0]]$  is sorted.
- After inserting  $\text{arr}[1]$ , subarray  $[\text{arr}[0:2]]$  is sorted.
- By induction, full array sorted at end.

Check after every insertion: `assert arr[:i] == sorted(arr[:i])`

## Tiny Code (Python)

```
def insertion_sort_with_invariant(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
        # Check invariant
        assert arr[:i+1] == sorted(arr[:i+1]), f"Invariant broken at i={i}"
    return arr
```

If invariant fails, an assertion error reveals the exact iteration.

## Why It Matters

- Builds correctness proofs via induction
- Early bug detection, pinpoints iteration errors
- Clarifies algorithm intent
- Teaches structured reasoning about program logic

Used in:

- Formal proofs (loop invariants)
- Algorithm verification
- Education and analysis

## A Gentle Proof (Why It Works)

Let  $P(i)$  denote the invariant “prefix of length  $i$  is sorted.”

- Base case:  $P(1)$  holds trivially.
- Inductive step: If  $P(i)$  holds, inserting next element keeps  $P(i + 1)$  true.

By induction,  $P(n)$  holds, full array is sorted.

Thus, the invariant framework guarantees correctness if each step preserves truth.

## Try It Yourself

1. Add invariants to Selection Sort (“min element placed at index i”).
2. Add heap property invariant to Heap Sort.
3. Run assertions in test suite.
4. Use `try/except` to log rather than stop when invariants fail.

## Test Cases

| Algorithm      | Invariant                                   | Holds? | Notes                       |
|----------------|---------------------------------------------|--------|-----------------------------|
| Insertion Sort | Prefix sorted at each step                  | Yes    | Classic inductive invariant |
| Selection Sort | Min placed at position i                    | Yes    | Verified iteratively        |
| Quick Sort     | Pivot partitions left pivot right           | Yes    | Must hold after partition   |
| Bubble Sort    | Largest element bubbles to correct position | Yes    | After each full pass        |

## Complexity

| Check Type | Time           | Space                       | Notes                 |
|------------|----------------|-----------------------------|-----------------------|
| Assertion  | $O(k)$         | $O(1)$                      | For prefix length $k$ |
| Total cost | $O(n^2)$ worst | For nested invariant checks |                       |

A Sorting Invariant Checker transforms correctness from intuition into logic, enforcing order, proving validity, and illuminating the structure of sorting algorithms one invariant at a time.

## 67 Distribution Histogram Sort Demo

A Distribution Histogram Sort Demo visualizes how elements spread across buckets or bins during distribution-based sorting. It helps learners see *why* and *how* counting, radix, or bucket sort achieve linear-time behavior by organizing values before final ordering.

### What Problem Are We Solving?

Distribution-based sorts (Counting, Bucket, Radix) don't rely on pairwise comparisons. Instead, they classify elements into bins based on keys or digits. Understanding these algorithms requires visualizing how data is distributed across categories, a histogram captures that process.

The demo shows:

- How counts are collected
- How prefix sums turn counts into positions
- How items are rebuilt in sorted order

### How It Works (Plain Language)

1. Initialize buckets, one for each key or range.
2. Traverse input and increment count in the right bucket.
3. Visualize the resulting histogram of frequencies.
4. (Optional) Apply prefix sums to show cumulative positions.
5. Reconstruct output by reading bins in order.

This visualization connects counting logic to the final sorted array.

### Example Step by Step

Example: Counting sort on [2, 1, 2, 0, 1]

| Value | Count |
|-------|-------|
| 0     | 1     |
| 1     | 2     |
| 2     | 2     |

Prefix sums → [1, 3, 5] Rebuild array → [0, 1, 1, 2, 2]

The histogram clearly shows where each group of values will end up.

## Tiny Code (Python)

```
import matplotlib.pyplot as plt

def histogram_sort_demo(arr, max_value):
    counts = [0] * (max_value + 1)
    for x in arr:
        counts[x] += 1

    plt.bar(range(len(counts)), counts)
    plt.xlabel("Value")
    plt.ylabel("Frequency")
    plt.title("Distribution Histogram for Counting Sort")
    plt.show()

# Optional reconstruction
sorted_arr = []
for val, freq in enumerate(counts):
    sorted_arr.extend([val] * freq)
return sorted_arr
```

Example:

```
histogram_sort_demo([2, 1, 2, 0, 1], 2)
```

## Why It Matters

- Makes non-comparison sorting intuitive
- Shows data frequency patterns
- Bridges between counting and position assignment
- Helps explain  $O(n + k)$  complexity visually

## A Gentle Proof (Why It Works)

Each value's frequency  $f_i$  determines exactly how many times it appears. By prefix-summing counts:

$$p_i = \sum_{j < i} f_j$$

we assign unique output positions for each value, ensuring stable, correct ordering in linear time.

Thus, sorting becomes position mapping, not comparison.

### Try It Yourself

1. Plot histograms for random, sorted, and uniform arrays.
2. Compare bucket sizes in Bucket Sort vs digit positions in Radix Sort.
3. Add prefix-sum labels to histogram bars.
4. Animate step-by-step rebuild of output.

### Test Cases

| Input       | Max | Histogram | Sorted Output |
|-------------|-----|-----------|---------------|
| [2,1,2,0,1] | 2   | [1,2,2]   | [0,1,1,2,2]   |
| [3,3,3,3]   | 3   | [0,0,0,4] | [3,3,3,3]     |
| [0,1,2,3]   | 3   | [1,1,1,1] | [0,1,2,3]     |

### Complexity

| Operation        | Time       | Space      | Notes                   |
|------------------|------------|------------|-------------------------|
| Counting         | $O(n)$     | $O(k)$     | $k$ = number of buckets |
| Prefix summation | $O(k)$     | $O(k)$     | Single pass over counts |
| Reconstruction   | $O(n + k)$ | $O(n + k)$ | Build sorted array      |

The Distribution Histogram Sort Demo transforms abstract counting logic into a concrete visual, showing how frequency shapes order and making linear-time sorting crystal clear.

## 68 Key Extraction Function

A Key Extraction Function isolates the specific feature or attribute from a data element that determines its position in sorting. It's a foundational tool for flexible, reusable sorting logic, enabling algorithms to handle complex records, tuples, or custom objects.

## What Problem Are We Solving?

Sorting real-world data often involves structured elements, tuples, objects, or dictionaries, not just numbers. We rarely sort entire elements directly; instead, we sort by a key:

- Name alphabetically
- Age numerically
- Date chronologically

A key extractor defines *how to view* each item for comparison, decoupling *data* from *ordering*.

## How It Works (Plain Language)

1. Define a key function: `key(x) → extracts sortable attribute.`
2. Apply key function during comparisons.
3. Algorithm sorts based on these extracted values.
4. The original elements remain intact, only their order changes.

## Example Step by Step

Suppose you have:

```
students = [  
    ("Alice", 22, 3.8),  
    ("Bob", 20, 3.5),  
    ("Clara", 21, 3.9)  
]
```

To sort by age, use `key=lambda x: x[1]`. To sort by GPA (descending), use `key=lambda x: -x[2]`.

Results:

- By age → `[("Bob", 20, 3.5), ("Clara", 21, 3.9), ("Alice", 22, 3.8)]`
- By GPA → `[("Clara", 21, 3.9), ("Alice", 22, 3.8), ("Bob", 20, 3.5)]`

## Tiny Code (Python)

```

def sort_by_key(data, key):
    return sorted(data, key=key)

students = [("Alice", 22, 3.8), ("Bob", 20, 3.5), ("Clara", 21, 3.9)]

# Sort by age
result = sort_by_key(students, key=lambda x: x[1])
# Sort by GPA descending
result2 = sort_by_key(students, key=lambda x: -x[2])

```

This abstraction allows clean, reusable sorting.

## Why It Matters

- Separates logic: comparison mechanism vs data structure
- Reusability: one algorithm, many orderings
- Composability: multi-level sorting by chaining keys
- Stability synergy: stable sorts + key extraction = multi-key sorting

## A Gentle Proof (Why It Works)

Let  $f(x)$  be the key extractor. We sort based on  $f(x)$ , not  $x$ . If the comparator satisfies:

$$f(x_i) \leq f(x_j) \implies x_i \text{ precedes } x_j$$

then the resulting order respects the intended attribute. Because  $f$  is deterministic, sort correctness follows directly from comparator correctness.

## Try It Yourself

1. Sort strings by length: `key=len`
2. Sort dictionary list by field: `key=lambda d: d['score']`
3. Compose keys: `key=lambda x: (x.grade, x.name)`
4. Combine with stability to simulate SQL “ORDER BY”

## Test Cases

| Input                                    | Key                          | Result                                   |
|------------------------------------------|------------------------------|------------------------------------------|
| <code>[("A",3),("B",1),("C",2)]</code>   | <code>lambda x:x[1]</code>   | <code>[("B",1),("C",2),("A",3)]</code>   |
| <code>["cat","a","bird"]</code>          | <code>len</code>             | <code>["a","cat","bird"]</code>          |
| <code>[{"x":5}, {"x":2}, {"x":4}]</code> | <code>lambda d:d["x"]</code> | <code>[{"x":2}, {"x":4}, {"x":5}]</code> |

## Complexity

| Step           | Time           | Space  | Notes                     |
|----------------|----------------|--------|---------------------------|
| Key extraction | $O(n)$         | $O(1)$ | One call per element      |
| Sorting        | $O(n \log n)$  | $O(n)$ | Depends on algorithm used |
| Composition    | $O(k \cdot n)$ | $O(1)$ | For multi-key chaining    |

The Key Extraction Function is the bridge between raw data and custom order, empowering algorithms to sort not just numbers, but meaning.

## 69 Partially Ordered Set Builder

A Partially Ordered Set (Poset) Builder constructs a visual and logical model of relationships that define *partial orderings* among elements, where some items can be compared, and others cannot. It's a conceptual tool for understanding sorting constraints, dependency graphs, and precedence structures.

### What Problem Are We Solving?

Not all collections have a total order. Sometimes only partial comparisons make sense, such as:

- Task dependencies (A before B, C independent)
- Version control merges
- Topological ordering in DAGs

A poset captures these relationships:

- Reflexive: every element itself
- Antisymmetric: if A  $\leq$  B and B  $\leq$  A, then A = B
- Transitive: if A  $\leq$  B and B  $\leq$  C, then A  $\leq$  C

Building a poset helps us visualize constraints before attempting to sort or schedule.

## How It Works (Plain Language)

1. Define a relation ( ) among elements.
2. Build a graph where an edge  $A \rightarrow B$  means “A  $\leq$  B.”
3. Ensure reflexivity, antisymmetry, and transitivity.
4. Visualize the result as a Hasse diagram (omit redundant edges).
5. Use this structure to find linear extensions (valid sorted orders).

## Example Step by Step

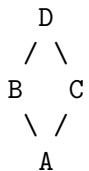
Example: Suppose we have tasks with dependencies:

A  $\leq$  B, A  $\leq$  C, B  $\leq$  D, C  $\leq$  D

Construct the poset:

- Nodes: A, B, C, D
- Edges: A  $\rightarrow$  B, A  $\rightarrow$  C, B  $\rightarrow$  D, C  $\rightarrow$  D

Hasse diagram:



Possible total orders (linear extensions):

- A, B, C, D
- A, C, B, D

## Tiny Code (Python)

```
from collections import defaultdict

def build_poset(relations):
    graph = defaultdict(list)
    for a, b in relations:
        graph[a].append(b)
```

```

    return graph

relations = [('A', 'B'), ('A', 'C'), ('B', 'D'), ('C', 'D')]
poset = build_poset(relations)
for k, v in poset.items():
    print(f"{k} → {v}")

```

Output:

```

A → ['B', 'C']
B → ['D']
C → ['D']

```

You can extend this to visualize with tools like `networkx`.

## Why It Matters

- Models dependencies and precedence
- Foundation of topological sorting
- Explains why total order isn't always possible
- Clarifies constraint satisfaction in scheduling

Used in:

- Build systems (make, DAGs)
- Task planning
- Compiler dependency analysis

## A Gentle Proof (Why It Works)

A poset  $(P, \leq)$  satisfies three axioms:

1. Reflexivity:  $\forall x, x \leq x$
2. Antisymmetry:  $(x \leq y \wedge y \leq x) \implies x = y$
3. Transitivity:  $(x \leq y \wedge y \leq z) \implies x \leq z$

These properties ensure consistent structure. Sorting a poset means finding a linear extension consistent with all  $\leq$  relations, which a topological sort guarantees for DAGs.

## Try It Yourself

1. Define tasks with prerequisites.
2. Draw a Hasse diagram.
3. Perform topological sort to list valid total orders.
4. Add extra relation, check if antisymmetry breaks.

## Test Cases

| Relations                  | Poset Edges                | Linear Orders         |
|----------------------------|----------------------------|-----------------------|
| A → B, A → C, B → D, C → D | A → B, A → C, B → D, C → D | [A,B,C,D], [A,C,B,D]  |
| A → B, B → C, A → C        | A → B, B → C, A → C        | [A,B,C]               |
| A → B, B → A (invalid)     | ,                          | Violates antisymmetry |

## Complexity

| Operation            | Time       | Space  | Notes                          |
|----------------------|------------|--------|--------------------------------|
| Build relation graph | $O(E)$     | $O(V)$ | $E$ = number of relations      |
| Check antisymmetry   | $O(E)$     | $O(V)$ | Detect cycles or bidirectional |
| Topological sort     | $O(V + E)$ | $O(V)$ | For linear extensions          |

The Partially Ordered Set Builder turns abstract ordering constraints into structured insight, showing not just *what comes first*, but *what can coexist*.

## 70 Complexity Comparator

A Complexity Comparator helps us understand how different algorithms scale by comparing their time or space complexity functions directly. It's a tool for intuition: how does  $O(n)$  stack up against  $O(n \log n)$  or  $O(2^n)$  as  $n$  grows large?

## What Problem Are We Solving?

When faced with multiple algorithms solving the same problem, we must decide which is more efficient for large inputs. Rather than guess, we compare growth rates of their complexity functions.

Example: Is  $O(n^2)$  slower than  $O(n \log n)$ ? For small  $n$ , maybe not. But as  $n \rightarrow \infty$ ,  $n^2$  grows faster, so the  $O(n \log n)$  algorithm is asymptotically better.

### How It Works (Plain Language)

1. Define the two functions  $f(n)$  and  $g(n)$  representing their costs.
2. Compute the ratio  $\frac{f(n)}{g(n)}$  as  $n \rightarrow \infty$ .
3. Interpret the limit:
  - If  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$ , then  $f(n) = o(g(n))$  (grows slower).
  - If limit is  $\infty$ , then  $f(n) = \omega(g(n))$  (grows faster).
  - If limit is constant, then  $f(n) = \Theta(g(n))$  (same growth).
4. Visualize using plots or tables for small  $n$  to understand crossover points.

### Example Step by Step

Compare  $f(n) = n \log n$  and  $g(n) = n^2$ :

- Compute ratio:  $\frac{f(n)}{g(n)} = \frac{n \log n}{n^2} = \frac{\log n}{n}$ .
- As  $n \rightarrow \infty$ ,  $\frac{\log n}{n} \rightarrow 0$ . Therefore,  $f(n) = o(g(n))$ .

Interpretation:  $O(n \log n)$  grows slower than  $O(n^2)$ , so it's more scalable.

### Tiny Code (Python)

```
import math

def compare_growth(f, g, n_values):
    for n in n_values:
        print(f"n={n:6d} f(n)={f(n):10.2f} g(n)={g(n):10.2f} ratio={f(n)/g(n):10.6f}")

compare_growth(lambda n: n * math.log2(n),
               lambda n: n2,
               [2, 4, 8, 16, 32, 64, 128])
```

Output shows how  $\frac{f(n)}{g(n)}$  decreases with  $n$ .

## Why It Matters

- Makes asymptotic comparison visual and numeric
- Reveals crossover points for real-world input sizes
- Helps choose between multiple implementations
- Deepens intuition about scaling laws

## A Gentle Proof (Why It Works)

We rely on limit comparison:

If  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c$ :

- If  $0 < c < \infty$ , then  $f(n) = \Theta(g(n))$
- If  $c = 0$ , then  $f(n) = o(g(n))$
- If  $c = \infty$ , then  $f(n) = \omega(g(n))$

This follows from formal definitions of asymptotic notation, ensuring consistency across comparisons.

## Try It Yourself

1. Compare  $O(n^2)$  vs  $O(n^3)$
2. Compare  $O(n \log n)$  vs  $O(n^{1.5})$
3. Compare  $O(2^n)$  vs  $O(n!)$
4. Plot their growth using Python or Excel

## Test Cases

| $f(n)$     | $g(n)$     | Ratio as $n \rightarrow \infty$ | Relationship        |
|------------|------------|---------------------------------|---------------------|
| $n$        | $n \log n$ | 0                               | $n = o(n \log n)$   |
| $n \log n$ | $n^2$      | 0                               | $n \log n = o(n^2)$ |
| $n^2$      | $n^2$      | 1                               | $\Theta$            |
| $2^n$      | $n^3$      | $\infty$                        | $2^n = \omega(n^3)$ |

## Complexity

| Operation          | Time   | Space  | Notes                      |
|--------------------|--------|--------|----------------------------|
| Function ratio     | $O(1)$ | $O(1)$ | Constant-time comparison   |
| Empirical table    | $O(k)$ | $O(k)$ | For $k$ sampled points     |
| Plot visualization | $O(k)$ | $O(k)$ | Helps understand crossover |

The Complexity Comparator is your lens for asymptotic insight, showing not just which algorithm is faster, but *why* it scales better.

## Section 8. Data Structure Overview

### 71 Stack Simulation

A Stack Simulation lets us watch the push and pop operations unfold step by step, revealing the LIFO (Last In, First Out) nature of this simple yet powerful data structure.

#### What Problem Are We Solving?

Stacks are everywhere: in recursion, expression evaluation, backtracking, and function calls. But for beginners, their dynamic behavior can feel abstract. A simulation makes it concrete, every push adds a layer, every pop removes one.

Goal: Understand how and when elements enter and leave the stack, and why order matters.

#### How It Works (Plain Language)

1. Start with an empty stack.
2. Push(x): Add element  $x$  to the top.
3. Pop(): Remove the top element.
4. Peek() (optional): Look at the top without removing it.
5. The most recently pushed element is always the first removed.

Think of a stack of plates: you can only take from the top.

## Example Step by Step

Operations:

```
Push(10)
Push(20)
Push(30)
Pop()
Push(40)
```

Stack evolution:

| Step | Operation | Stack State (Top → Bottom) |
|------|-----------|----------------------------|
| 1    | Push(10)  | 10                         |
| 2    | Push(20)  | 20, 10                     |
| 3    | Push(30)  | 30, 20, 10                 |
| 4    | Pop()     | 20, 10                     |
| 5    | Push(40)  | 40, 20, 10                 |

## Tiny Code (Python)

```
class Stack:
    def __init__(self):
        self.data = []

    def push(self, x):
        self.data.append(x)
        print(f"Pushed {x}: {self.data[:-1]}")

    def pop(self):
        if self.data:
            x = self.data.pop()
            print(f"Popped {x}: {self.data[:-1]}")
            return x

# Demo
s = Stack()
s.push(10)
s.push(20)
```

```
s.push(30)  
s.pop()  
s.push(40)
```

Each action prints the current state, simulating stack behavior.

## Why It Matters

- Models function calls and recursion
- Essential for undo operations and backtracking
- Underpins expression parsing and evaluation
- Builds intuition for control flow and memory frames

## A Gentle Proof (Why It Works)

A stack enforces LIFO ordering: If you push elements in order  $a_1, a_2, \dots, a_n$ , you must pop them in reverse:  $a_n, \dots, a_2, a_1$ .

Formally, each push increases size by 1, each pop decreases it by 1, ensuring  $|S| = \text{pushes} - \text{pops}$  and order reverses naturally.

## Try It Yourself

1. Simulate postfix expression evaluation ( $3\ 4\ +\ 5\ *$ )
2. Trace recursive function calls (factorial or Fibonacci)
3. Implement browser backtracking with a stack
4. Push strings and pop them to reverse order

## Test Cases

| Operation Sequence                | Final Stack (Top → Bottom) |
|-----------------------------------|----------------------------|
| Push(1), Push(2), Pop()           | 1                          |
| Push('A'), Push('B'), Push('C')   | C, B, A                    |
| Push(5), Pop(), Pop()             | (empty)                    |
| Push(7), Push(9), Push(11), Pop() | 9, 7                       |

## Complexity

| Operation | Time | Space | Note             |
|-----------|------|-------|------------------|
| Push(x)   | O(1) | O(n)  | Append to list   |
| Pop()     | O(1) | O(n)  | Remove last item |
| Peek()    | O(1) | O(n)  | Access last item |

A Stack Simulation makes abstract order tangible, every push and pop tells a story of control, memory, and flow.

## 72 Queue Simulation

A Queue Simulation shows how elements move through a first-in, first-out structure, perfect for modeling waiting lines, job scheduling, or data streams.

### What Problem Are We Solving?

Queues capture fairness and order. They're essential in task scheduling, buffering, and resource management, but their behavior can seem opaque without visualization.

Simulating operations reveals how enqueue and dequeue actions shape the system over time.

Goal: Understand FIFO (First-In, First-Out) order and how it ensures fairness in processing.

### How It Works (Plain Language)

1. Start with an empty queue.
2. Enqueue(x): Add element x to the rear.
3. Dequeue(): Remove the front element.
4. Peek() (optional): See the next item to be processed.

Like a line at a ticket counter, first person in is first to leave.

## Example Step by Step

Operations:

```
Enqueue(10)
Enqueue(20)
Enqueue(30)
Dequeue()
Enqueue(40)
```

Queue evolution:

| Step | Operation   | Queue State (Front → Rear) |
|------|-------------|----------------------------|
| 1    | Enqueue(10) | 10                         |
| 2    | Enqueue(20) | 10, 20                     |
| 3    | Enqueue(30) | 10, 20, 30                 |
| 4    | Dequeue()   | 20, 30                     |
| 5    | Enqueue(40) | 20, 30, 40                 |

## Tiny Code (Python)

```
from collections import deque

class Queue:
    def __init__(self):
        self.data = deque()

    def enqueue(self, x):
        self.data.append(x)
        print(f"Enqueued {x}: {list(self.data)}")

    def dequeue(self):
        if self.data:
            x = self.data.popleft()
            print(f"Dequeued {x}: {list(self.data)}")
            return x

# Demo
q = Queue()
```

```

q.enqueue(10)
q.enqueue(20)
q.enqueue(30)
q.dequeue()
q.enqueue(40)

```

Each step prints the queue's current state, helping you trace order evolution.

## Why It Matters

- Models real-world waiting lines
- Used in schedulers, network buffers, and BFS traversals
- Ensures fair access to limited resources
- Builds intuition for stream processing

## A Gentle Proof (Why It Works)

A queue preserves arrival order. If elements arrive in order  $a_1, a_2, \dots, a_n$ , they exit in the same order,  $a_1, a_2, \dots, a_n$ .

Each enqueue appends to the rear, each dequeue removes from the front. Thus, insertion and removal sequences match, enforcing FIFO.

## Try It Yourself

1. Simulate a print queue, jobs enter and complete in order.
2. Implement BFS on a small graph using a queue.
3. Model ticket line arrivals and departures.
4. Track packet flow through a network buffer.

## Test Cases

| Operation Sequence                             | Final Queue (Front → Rear) |
|------------------------------------------------|----------------------------|
| Enqueue(1), Enqueue(2), Dequeue()              | 2                          |
| Enqueue('A'), Enqueue('B'), Enqueue('C')       | A, B, C                    |
| Enqueue(5), Dequeue(), Dequeue()               | (empty)                    |
| Enqueue(7), Enqueue(9), Enqueue(11), Dequeue() | 9, 11                      |

## Complexity

| Operation  | Time | Space | Note              |
|------------|------|-------|-------------------|
| Enqueue(x) | O(1) | O(n)  | Append to rear    |
| Dequeue()  | O(1) | O(n)  | Remove from front |
| Peek()     | O(1) | O(n)  | Access front item |

A Queue Simulation clarifies the rhythm of fairness, each arrival patiently waits its turn, no one cutting in line.

## 73 Linked List Builder

A Linked List Builder shows how elements connect through pointers, the foundation for dynamic memory structures where data grows or shrinks on demand.

### What Problem Are We Solving?

Arrays have fixed size and require contiguous memory. Linked lists solve this by linking scattered nodes dynamically, one pointer at a time.

By simulating node creation and linkage, we build intuition for pointer manipulation and traversal, essential for mastering lists, stacks, queues, and graphs.

Goal: Understand how nodes link together and how to maintain references during insertion or deletion.

### How It Works (Plain Language)

A singly linked list is a sequence of nodes, each holding:

- A value
- A pointer to the next node

Basic operations:

1. Create node(value) → allocate new node.
2. Insert after → link new node between existing ones.
3. Delete → redirect pointers to skip a node.
4. Traverse → follow next pointers until `None`.

Like a chain, each link knows only the next one.

## Example Step by Step

Build a list:

```
Insert(10)
Insert(20)
Insert(30)
```

Process:

1. Create node(10): head → 10 → None
2. Create node(20): head → 10 → 20 → None
3. Create node(30): head → 10 → 20 → 30 → None

Traversal from head prints: 10 → 20 → 30 → None

## Tiny Code (Python)

```
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def insert(self, value):
        new_node = Node(value)
        if not self.head:
            self.head = new_node
        else:
            cur = self.head
            while cur.next:
                cur = cur.next
            cur.next = new_node
    self.display()

    def display(self):
        cur = self.head
        elems = []
```

```

while cur:
    elems.append(str(cur.value))
    cur = cur.next
print(" → ".join(elems) + " → None")

# Demo
ll = LinkedList()
ll.insert(10)
ll.insert(20)
ll.insert(30)

```

## Why It Matters

- Enables dynamic memory allocation
- No need for contiguous storage
- Powers stacks, queues, hash chains, adjacency lists
- Builds foundation for advanced pointer-based structures

## A Gentle Proof (Why It Works)

Let  $n$  be the number of nodes. Each node has exactly one outgoing pointer (to `next`) or `None`. Traversing once visits every node exactly once.

Therefore, insertion or traversal takes  $O(n)$  time, and storage is  $O(n)$  (one node per element).

## Try It Yourself

1. Insert values {5, 15, 25, 35}
2. Delete the second node and reconnect links
3. Reverse the list manually by reassigning pointers
4. Visualize how each `next` changes during reversal

## Test Cases

| Operation Sequence                | Expected Output    |
|-----------------------------------|--------------------|
| Insert(10), Insert(20)            | 10 → 20 → None     |
| Insert(5), Insert(15), Insert(25) | 5 → 15 → 25 → None |
| Empty List                        | None               |
| Single Node                       | 42 → None          |

## Complexity

| Operation   | Time   | Space  | Note                 |
|-------------|--------|--------|----------------------|
| Insert End  | $O(n)$ | $O(n)$ | Traverse to tail     |
| Delete Node | $O(n)$ | $O(n)$ | Find predecessor     |
| Search      | $O(n)$ | $O(n)$ | Sequential traversal |
| Traverse    | $O(n)$ | $O(n)$ | Visit each node once |

A Linked List Builder is your first dance with pointers, where structure emerges from simple connections, and memory becomes fluid, flexible, and free.

## 74 Array Index Visualizer

An Array Index Visualizer helps you see how arrays organize data in contiguous memory and how indexing gives  $O(1)$  access to any element.

### What Problem Are We Solving?

Arrays are the simplest data structure, but beginners often struggle to grasp how indexing truly works under the hood. By visualizing index positions and memory offsets, you can see why arrays allow direct access yet require fixed size and contiguous space.

Goal: Understand the relationship between index, address, and element access.

### How It Works (Plain Language)

An array stores  $n$  elements consecutively in memory. If the base address is  $A_0$ , and each element takes  $s$  bytes, then:

$$A_i = A_0 + i \times s$$

So accessing index  $i$  is constant-time:

- Compute address
- Jump directly there
- Retrieve value

This visualization ties logical indices (0, 1, 2, ...) to physical locations.

## Example Step by Step

Suppose we have an integer array:

```
arr = [10, 20, 30, 40]
```

Base address: 1000, element size: 4 bytes

| Index | Address | Value |
|-------|---------|-------|
| 0     | 1000    | 10    |
| 1     | 1004    | 20    |
| 2     | 1008    | 30    |
| 3     | 1012    | 40    |

Access arr[2]:

- Compute  $A_0 + 2 \times 4 = 1008$
- Retrieve 30

## Tiny Code (Python)

```
def visualize_array(arr, base=1000, size=4):
    print(f"{'Index':<8}{{'Address':<10}{{'Value':<8}}")
    for i, val in enumerate(arr):
        address = base + i * size
        print(f"{i:<8}{address:<10}{val:<8}")

arr = [10, 20, 30, 40]
visualize_array(arr)
```

Output:

| Index | Address | Value |
|-------|---------|-------|
| 0     | 1000    | 10    |
| 1     | 1004    | 20    |
| 2     | 1008    | 30    |
| 3     | 1012    | 40    |

## Why It Matters

- Instant access via address computation
- Contiguity ensures cache locality
- Fixed size and type consistency
- Core of higher-level structures (strings, matrices, tensors)

## A Gentle Proof (Why It Works)

Let  $A_0$  be the base address. Each element occupies  $s$  bytes. To access element  $i$ :

$$A_i = A_0 + i \times s$$

This is a simple arithmetic operation, so access is  $O(1)$ , independent of  $n$ .

## Try It Yourself

1. Visualize array [5, 10, 15, 20, 25] with base 5000 and size 8.
2. Access `arr[4]` manually using formula.
3. Compare array vs. linked list access time.
4. Modify size and re-run visualization.

## Test Cases

| Array           | Base | Size | Access | Expected Address | Value |
|-----------------|------|------|--------|------------------|-------|
| [10, 20, 30]    | 1000 | 4    | arr[1] | 1004             | 20    |
| [7, 14, 21, 28] | 500  | 2    | arr[3] | 506              | 28    |

## Complexity

| Operation     | Time   | Space  | Note               |
|---------------|--------|--------|--------------------|
| Access        | $O(1)$ | $O(n)$ | Direct via formula |
| Update        | $O(1)$ | $O(n)$ | Single write       |
| Traverse      | $O(n)$ | $O(n)$ | Visit all          |
| Insert/Delete | $O(n)$ | $O(n)$ | Requires shifting  |

An Array Index Visualizer reveals how logic meets hardware, every index a direct pointer, every element a predictable step from the base.

## 75 Hash Function Mapper

A Hash Function Mapper shows how keys are transformed into array indices, turning arbitrary data into fast-access positions.

### What Problem Are We Solving?

We often need to store and retrieve data by key (like “Alice” or “user123”), not by numeric index. But arrays only understand numbers. A hash function bridges this gap, mapping keys into integer indices so we can use array-like speed for key-based lookup.

Goal: Understand how keys become indices and how hash collisions occur.

### How It Works (Plain Language)

A hash function takes a key and computes an index:

$$\text{index} = h(\text{key}) \bmod m$$

where:

- $h(\text{key})$  is a numeric hash value,
- $m$  is the table size.

For example:

```
key = "cat"
h(key) = 493728
m = 10
index = 493728 % 10 = 8
```

Now "cat" is mapped to slot 8.

If another key maps to the same index, a collision occurs, handled by chaining or probing.

### Example Step by Step

Suppose a table of size 5.

Keys: "red", "blue", "green"

| Key   | Hash Value | Index (% 5)   |
|-------|------------|---------------|
| red   | 432        | 2             |
| blue  | 107        | 2 (collision) |
| green | 205        | 0             |

We see "red" and "blue" collide at index 2.

### Tiny Code (Python)

```
def simple_hash(key):
    return sum(ord(c) for c in key)

def map_keys(keys, size=5):
    table = [[] for _ in range(size)]
    for k in keys:
        idx = simple_hash(k) % size
        table[idx].append(k)
        print(f"Key: {k[:6]} -> Index: {idx}")
    return table

keys = ["red", "blue", "green"]
table = map_keys(keys)
```

Output:

```
Key: red    -> Index: 2
Key: blue   -> Index: 2
Key: green  -> Index: 0
```

### Why It Matters

- Enables constant-time average lookup and insertion
- Forms the backbone of hash tables, dictionaries, caches
- Shows tradeoffs between hash quality and collision handling

## A Gentle Proof (Why It Works)

If a hash function distributes keys uniformly, expected number of keys per slot is  $\frac{n}{m}$ .

Thus, expected lookup time:

$$E[T] = O\left(1 + \frac{n}{m}\right)$$

For well-chosen  $m$  and good  $h$ ,  $E[T] \approx O(1)$ .

## Try It Yourself

1. Map `["cat", "dog", "bat", "rat"]` to a table of size 7.
2. Observe collisions and try a larger table.
3. Replace `sum(ord(c))` with a polynomial hash:

$$h(\text{key}) = \sum c_i \times 31^i$$

4. Compare distribution quality.

## Test Cases

| Keys                         | Table Size | Result (Indices)     |
|------------------------------|------------|----------------------|
| <code>["a", "b", "c"]</code> | 3          | 1, 2, 0              |
| <code>["hi", "ih"]</code>    | 5          | collision (same sum) |

## Complexity

| Operation | Time (Expected) | Space  | Note                 |
|-----------|-----------------|--------|----------------------|
| Insert    | $O(1)$          | $O(n)$ | Average, good hash   |
| Search    | $O(1)$          | $O(n)$ | With uniform hashing |
| Delete    | $O(1)$          | $O(n)$ | Same cost as lookup  |

A Hash Function Mapper makes hashing tangible, you watch strings become slots, collisions emerge, and order dissolve into probability and math.

## 76 Binary Tree Builder

A Binary Tree Builder illustrates how hierarchical data structures are constructed by linking nodes with left and right children.

### What Problem Are We Solving?

Linear structures like arrays and lists can't efficiently represent hierarchical relationships. When you need ordering, searching, and hierarchical grouping, a binary tree provides the foundation.

Goal: Understand how nodes are connected to form a tree and how recursive structure emerges naturally.

### How It Works (Plain Language)

A binary tree is made of nodes. Each node has:

- a value
- a left child
- a right child

To build a tree:

1. Start with a root node
2. Recursively insert new nodes:
  - If value < current → go left
  - Else → go right
3. Repeat until you find a null link

This produces a Binary Search Tree (BST), maintaining order property.

## Example Step by Step

Insert values: [10, 5, 15, 3, 7, 12, 18]

Process:

```
10
 5
   3
   7
 15
   12
   18
```

Traversal orders:

- Inorder: 3, 5, 7, 10, 12, 15, 18
- Preorder: 10, 5, 3, 7, 15, 12, 18
- Postorder: 3, 7, 5, 12, 18, 15, 10

## Tiny Code (Python)

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

class BST:
    def __init__(self):
        self.root = None

    def insert(self, value):
        self.root = self._insert(self.root, value)

    def _insert(self, node, value):
        if node is None:
            return Node(value)
        if value < node.value:
            node.left = self._insert(node.left, value)
        else:
```

```

        node.right = self._insert(node.right, value)
    return node

def inorder(self, node):
    if node:
        self.inorder(node.left)
        print(node.value, end=" ")
        self.inorder(node.right)

# Demo
tree = BST()
for val in [10, 5, 15, 3, 7, 12, 18]:
    tree.insert(val)
tree.inorder(tree.root)

```

Output: 3 5 7 10 12 15 18

## Why It Matters

- Core structure for search trees, heaps, and expression trees
- Forms basis for balanced trees (AVL, Red-Black)
- Enables divide-and-conquer recursion naturally

## A Gentle Proof (Why It Works)

A binary search tree maintains the invariant:

$$\forall \text{node}, v : \{v_{\text{left}} < v_{\text{root}} < v_{\text{right}}$$

Insertion preserves this by recursive placement. Each insertion follows a single path of height  $h$ , so time is  $O(h)$ . For balanced trees,  $h = O(\log n)$ .

## Try It Yourself

1. Insert [8, 3, 10, 1, 6, 14, 4, 7, 13].
2. Draw the tree structure.
3. Perform inorder traversal (should print sorted order).
4. Compare with unbalanced insertion order.

## Test Cases

| Input Sequence    | Inorder Traversal |
|-------------------|-------------------|
| [10, 5, 15, 3, 7] | 3, 5, 7, 10, 15   |
| [2, 1, 3]         | 1, 2, 3           |
| [5]               | 5                 |

## Complexity

| Operation | Time (Avg)  | Time (Worst) | Space  |
|-----------|-------------|--------------|--------|
| Insert    | $O(\log n)$ | $O(n)$       | $O(n)$ |
| Search    | $O(\log n)$ | $O(n)$       | $O(1)$ |
| Delete    | $O(\log n)$ | $O(n)$       | $O(1)$ |
| Traverse  | $O(n)$      | $O(n)$       | $O(n)$ |

A Binary Tree Builder reveals order within hierarchy, each node a decision, each branch a story of lesser and greater.

## 77 Heap Structure Demo

A Heap Structure Demo helps you visualize how binary heaps organize data to always keep the smallest or largest element at the top, enabling fast priority access.

### What Problem Are We Solving?

We often need a structure that quickly retrieves the minimum or maximum element, like in priority queues or scheduling. Sorting every time is wasteful. A heap maintains partial order so the root is always extreme, and rearrangement happens locally.

Goal: Understand how insertion and removal maintain the heap property.

### How It Works (Plain Language)

A binary heap is a complete binary tree stored as an array. Each node satisfies:

- Min-heap: parent children
- Max-heap: parent children

Insertion and deletion are handled with *sift up* and *sift down* operations.

## Insert (Heapify Up)

1. Add new element at the end
2. Compare with parent
3. Swap if violates heap property
4. Repeat until heap property holds

## Remove Root (Heapify Down)

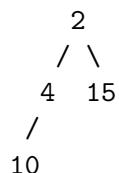
1. Replace root with last element
2. Compare with children
3. Swap with smaller (min-heap) or larger (max-heap) child
4. Repeat until property restored

## Example Step by Step (Min-Heap)

Insert [10, 4, 15, 2]

1. [10]
2. [10, 4] → swap(4, 10) → [4, 10]
3. [4, 10, 15] (no swap)
4. [4, 10, 15, 2] → swap(2, 10) → swap(2, 4) → [2, 4, 15, 10]

Final heap (array): [2, 4, 15, 10] Tree view:



## Tiny Code (Python)

```
import heapq

def heap_demo():
    heap = []
    for x in [10, 4, 15, 2]:
        heapq.heappush(heap, x)
        print("Insert", x, "->", heap)
```

```

while heap:
    print("Pop:", heapq.heappop(heap), "→", heap)

heap_demo()

```

Output:

```

Insert 10 → [10]
Insert 4 → [4, 10]
Insert 15 → [4, 10, 15]
Insert 2 → [2, 4, 15, 10]
Pop: 2 → [4, 10, 15]
Pop: 4 → [10, 15]
Pop: 10 → [15]
Pop: 15 → []

```

## Why It Matters

- Enables priority queues (task schedulers, Dijkstra)
- Supports  $O(1)$  access to min/max
- Keeps  $O(\log n)$  insertion/removal cost
- Basis for Heapsort

## A Gentle Proof (Why It Works)

Let  $h = \lfloor \log_2 n \rfloor$  be heap height. Each insert and delete moves along one path of height  $h$ . Thus:

$$T_{\text{insert}} = T_{\text{delete}} = O(\log n)$$

$$T_{\text{find-min}} = O(1)$$

## Try It Yourself

1. Insert  $[7, 2, 9, 1, 5]$  into a min-heap
2. Trace swaps on paper
3. Remove min repeatedly and record order (should be sorted ascending)
4. Repeat for max-heap version

## Test Cases

| Operation | Input      | Output (Heap)  |
|-----------|------------|----------------|
| Insert    | [5, 3, 8]  | [3, 5, 8]      |
| Pop       | [3, 5, 8]  | Pop 3 → [5, 8] |
| Insert    | [10, 2, 4] | [2, 10, 4]     |

## Complexity

| Operation    | Time        | Space  | Note              |
|--------------|-------------|--------|-------------------|
| Insert       | $O(\log n)$ | $O(n)$ | Percolate up      |
| Delete       | $O(\log n)$ | $O(n)$ | Percolate down    |
| Find Min/Max | $O(1)$      | $O(1)$ | Root access       |
| Build Heap   | $O(n)$      | $O(n)$ | Bottom-up heapify |

A Heap Structure Demo shows order through shape, every parent above its children, every insertion a climb toward balance.

## 78 Union-Find Concept

A Union-Find Concept (also called Disjoint Set Union, DSU) demonstrates how to efficiently manage dynamic grouping, deciding whether elements belong to the same set and merging sets when needed.

### What Problem Are We Solving?

In many problems, we need to track connected components, e.g. in graphs, social networks, or Kruskal's MST. We want to answer two operations efficiently:

- Find(x): which group is x in?
- Union(x, y): merge the groups of x and y

Naive approaches (like scanning arrays) cost too much. Union-Find structures solve this in *almost constant time* using parent pointers and path compression.

## How It Works (Plain Language)

Each element points to a parent. The root is the representative of its set. If two elements share the same root, they're in the same group.

Operations:

1. Find(x): Follow parent pointers until reaching a root (node where `parent[x] == x`) Use path compression to flatten paths for next time
2. Union(x, y): Find roots of x and y If different, attach one root to the other (merge sets)  
Optionally, use union by rank/size to keep tree shallow

## Example Step by Step

Start with {1}, {2}, {3}, {4}

Perform:

```
Union(1, 2) → {1,2}, {3}, {4}  
Union(3, 4) → {1,2}, {3,4}  
Union(2, 3) → {1,2,3,4}
```

All now connected under one root.

If `Find(4)` → returns 1 (root of its set)

## Tiny Code (Python)

```
class UnionFind:  
    def __init__(self, n):  
        self.parent = [i for i in range(n)]  
        self.rank = [0] * n  
  
    def find(self, x):  
        if self.parent[x] != x:  
            self.parent[x] = self.find(self.parent[x]) # Path compression  
        return self.parent[x]  
  
    def union(self, x, y):  
        rx, ry = self.find(x), self.find(y)  
        if rx == ry:  
            pass
```

```

        return
    if self.rank[rx] < self.rank[ry]:
        self.parent[rx] = ry
    elif self.rank[rx] > self.rank[ry]:
        self.parent[ry] = rx
    else:
        self.parent[ry] = rx
        self.rank[rx] += 1

# Demo
uf = UnionFind(5)
uf.union(0, 1)
uf.union(2, 3)
uf.union(1, 2)
print([uf.find(i) for i in range(5)])

```

Output: [0, 0, 0, 0, 4]

## Why It Matters

- Foundation for Kruskal's Minimum Spanning Tree
- Detects cycles in undirected graphs
- Efficient for connectivity queries in dynamic graphs
- Used in percolation, image segmentation, clustering

## A Gentle Proof (Why It Works)

Each operation has amortized cost given by the inverse Ackermann function  $\alpha(n)$ , practically constant.

$$T_{\text{find}}(n), T_{\text{union}}(n) = O(\alpha(n))$$

Because path compression ensures every node points closer to root each time, flattening structure to near-constant depth.

## Try It Yourself

1. Start with  $\{0\}$ ,  $\{1\}$ ,  $\{2\}$ ,  $\{3\}$ ,  $\{4\}$
2. Apply:  $\text{Union}(0,1)$ ,  $\text{Union}(2,3)$ ,  $\text{Union}(1,2)$
3. Query  $\text{Find}(3) \rightarrow$  should match root of 0
4. Print parent array after each operation

## Test Cases

| Operation Sequence                          | Resulting Sets                  |
|---------------------------------------------|---------------------------------|
| $\text{Union}(1, 2)$ , $\text{Union}(3, 4)$ | $\{1,2\}$ , $\{3,4\}$ , $\{0\}$ |
| $\text{Union}(2, 3)$                        | $\{0\}$ , $\{1,2,3,4\}$         |
| $\text{Find}(4)$                            | Root = 1 (or 0)                 |

## Complexity

| Operation       | Amortized Time | Space  | Notes               |
|-----------------|----------------|--------|---------------------|
| Find            | $O(\alpha(n))$ | $O(n)$ | Path compression    |
| Union           | $O(\alpha(n))$ | $O(n)$ | With rank heuristic |
| Connected(x, y) | $O(\alpha(n))$ | $O(1)$ | Via root comparison |

A Union-Find Concept turns disjoint sets into a living network, connections formed and flattened, unity discovered through structure.

## 79 Graph Representation Demo

A Graph Representation Demo reveals how graphs can be encoded in data structures, showing the tradeoffs between adjacency lists, matrices, and edge lists.

### What Problem Are We Solving?

Graphs describe relationships, roads between cities, links between websites, friendships in a network. But before we can run algorithms (like BFS, Dijkstra, or DFS), we need a representation that matches the graph's density, size, and operations.

Goal: Understand how different representations encode edges and how to choose the right one.

## How It Works (Plain Language)

A graph is defined as:

$$G = (V, E)$$

where:

- $V$  = set of vertices
- $E$  = set of edges (pairs of vertices)

We can represent  $G$  in three main ways:

### 1. Adjacency Matrix

- 2D array of size  $|V| \times |V|$
- Entry  $(i, j) = 1$  if edge  $(i, j)$  exists, else 0

### 2. Adjacency List

- For each vertex, a list of its neighbors
- Compact for sparse graphs

### 3. Edge List

- Simple list of all edges
- Easy to iterate, hard for quick lookup

## Example Step by Step

Consider an undirected graph:

Vertices: {A, B, C, D}  
Edges: {(A, B), (A, C), (B, D)}

Adjacency Matrix

|   | A | B | C | D |
|---|---|---|---|---|
| A | 0 | 1 | 1 | 0 |
| B | 1 | 0 | 0 | 1 |
| C | 1 | 0 | 0 | 0 |
| D | 0 | 1 | 0 | 0 |

Adjacency List

```
A: [B, C]
B: [A, D]
C: [A]
D: [B]
```

Edge List

```
[(A, B), (A, C), (B, D)]
```

### Tiny Code (Python)

```
from collections import defaultdict

# Adjacency List
graph = defaultdict(list)
edges = [("A", "B"), ("A", "C"), ("B", "D")]

for u, v in edges:
    graph[u].append(v)
    graph[v].append(u) # undirected

print("Adjacency List:")
for node, neighbors in graph.items():
    print(f"{node}: {neighbors}")

# Adjacency Matrix
vertices = ["A", "B", "C", "D"]
n = len(vertices)
matrix = [[0]*n for _ in range(n)]
index = {v: i for i, v in enumerate(vertices)}

for u, v in edges:
    i, j = index[u], index[v]
    matrix[i][j] = matrix[j][i] = 1

print("\nAdjacency Matrix:")
for row in matrix:
    print(row)
```

Output:

**Adjacency List:**

```
A: ['B', 'C']
B: ['A', 'D']
C: ['A']
D: ['B']
```

**Adjacency Matrix:**

```
[0, 1, 1, 0]
[1, 0, 0, 1]
[1, 0, 0, 0]
[0, 1, 0, 0]
```

## Why It Matters

- Adjacency matrix → fast lookup ( $O(1)$ ), high space ( $O(V^2)$ )
- Adjacency list → efficient for sparse graphs ( $O(V + E)$ )
- Edge list → simple to iterate, ideal for algorithms like Kruskal

Choosing wisely impacts performance of every algorithm on the graph.

## A Gentle Proof (Why It Works)

Let  $V$  be number of vertices,  $E$  edges.

| Representation   | Storage    | Edge Check   | Iteration  |
|------------------|------------|--------------|------------|
| Adjacency Matrix | $O(V^2)$   | $O(1)$       | $O(V^2)$   |
| Adjacency List   | $O(V + E)$ | $O(\deg(v))$ | $O(V + E)$ |
| Edge List        | $O(E)$     | $O(E)$       | $O(E)$     |

Sparse graphs ( $E \ll V^2$ ) → adjacency list preferred. Dense graphs ( $E \approx V^2$ ) → adjacency matrix is fine.

## Try It Yourself

1. Draw a graph with 5 nodes, 6 edges
2. Write all three representations
3. Compute storage cost
4. Pick best format for BFS vs Kruskal's MST

## Test Cases

| Graph Type | Representation | Benefit         |
|------------|----------------|-----------------|
| Sparse     | List           | Space efficient |
| Dense      | Matrix         | Constant lookup |
| Weighted   | Edge List      | Easy sorting    |

## Complexity

| Operation  | Matrix   | List         | Edge List |
|------------|----------|--------------|-----------|
| Space      | $O(V^2)$ | $O(V + E)$   | $O(E)$    |
| Add Edge   | $O(1)$   | $O(1)$       | $O(1)$    |
| Check Edge | $O(1)$   | $O(\deg(v))$ | $O(E)$    |
| Iterate    | $O(V^2)$ | $O(V + E)$   | $O(E)$    |

A Graph Representation Demo shows the blueprint of connection, the same network, three different lenses: matrix, list, or edge table.

## 80 Trie Structure Visualizer

A Trie Structure Visualizer helps you see how strings and prefixes are stored efficiently, one character per edge, building shared paths for common prefixes.

### What Problem Are We Solving?

When you need to store and search many strings, especially by prefix, linear scans or hash tables aren't ideal. We want something that makes prefix queries fast and memory use efficient through shared structure.

A trie (prefix tree) does exactly that, storing strings as paths, reusing common prefixes.

Goal: Understand how each character extends a path and how search and insert work along edges.

## How It Works (Plain Language)

A trie starts with an empty root node. Each edge represents a character. Each node may have multiple children, one for each possible next character.

To insert a word:

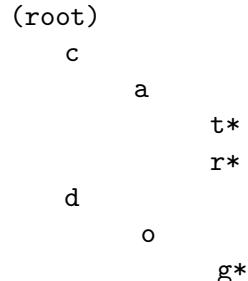
1. Start at root
2. For each character:
  - If it doesn't exist, create a new child
  - Move to that child
3. Mark last node as "end of word"

To search:

1. Start at root
2. Follow edges by each character
3. If path exists and end is marked, word found

## Example Step by Step

Insert cat, car, dog



Asterisk \* marks word end. Common prefix ca is shared.

Search "car":

- c
- a
- r
- End marked → found

Search "cap":

- c
- a
- p → not found

### Tiny Code (Python)

```

class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for ch in word:
            if ch not in node.children:
                node.children[ch] = TrieNode()
            node = node.children[ch]
        node.is_end = True

    def search(self, word):
        node = self.root
        for ch in word:
            if ch not in node.children:
                return False
            node = node.children[ch]
        return node.is_end

# Demo
trie = Trie()
for w in ["cat", "car", "dog"]:
    trie.insert(w)
print(trie.search("car")) # True
print(trie.search("cap")) # False

```

## Why It Matters

- Enables prefix search, autocomplete, dictionary lookup
- Avoids recomputing prefixes
- Efficient for string-intensive applications
- Foundation for compressed tries, DAWGs, and suffix trees

## A Gentle Proof (Why It Works)

Each character in word  $w$  follows one path in trie. Insert cost =  $O(|w|)$ , Search cost =  $O(|w|)$ .

For  $n$  words of average length  $L$ , total nodes =  $O(nL)$ .

Prefix query cost =  $O(p)$ , where  $p$  = prefix length.

## Try It Yourself

1. Insert ["cat", "cap", "can", "dog"]
2. Draw tree paths
3. Query prefixes "ca" and "do"
4. Count total nodes created

## Test Cases

| Operation | Input        | Output           |
|-----------|--------------|------------------|
| Insert    | "cat", "car" | Shared path "ca" |
| Search    | "car"        | True             |
| Search    | "cap"        | False            |
| Prefix    | "ca"         | Exists           |

## Complexity

| Operation    | Time   | Space  | Note                 |
|--------------|--------|--------|----------------------|
| Insert       | $O(L)$ | $O(L)$ | $L$ = length of word |
| Search       | $O(L)$ | $O(1)$ | Follow path          |
| Prefix Query | $O(p)$ | $O(1)$ | Shared traversal     |

A Trie Structure Visualizer shows structure born from language, every word a path, every prefix a meeting point, every branch a shared memory.

## Section 9. Graphs and Trees overview

### 81 Graph Model Constructor

A Graph Model Constructor is how we formally build graphs, sets of vertices connected by edges, to represent relationships, networks, or structures in the world.

#### What Problem Are We Solving?

We often face problems where elements are connected, roads between cities, friendships in a network, dependencies in a project. To reason about these, we need a way to model entities (vertices) and connections (edges).

The Graph Model Constructor provides the blueprint for turning real-world relationships into graph data structures we can analyze.

#### How It Works (Plain Language)

A graph is defined as:

$$G = (V, E)$$

where

- $V$  = set of vertices (nodes)
- $E$  = set of edges (connections) between vertices

Each edge can be:

- Undirected:  $(u, v)$  means  $u$  and  $v$  are connected both ways
- Directed:  $(u, v)$  means a one-way connection from  $u$  to  $v$

You can build graphs in multiple ways:

1. Edge List – list of pairs  $(u, v)$
2. Adjacency List – dictionary of node → neighbor list
3. Adjacency Matrix – 2D table of connections (1 = edge, 0 = none)

## **Example**

Input relationships

A connected to B  
A connected to C  
B connected to C  
C connected to D

Vertices

$V = \{A, B, C, D\}$

Edges

$E = \{(A, B), (A, C), (B, C), (C, D)\}$

Edge List

$[(A, B), (A, C), (B, C), (C, D)]$

Adjacency List

A: [B, C]  
B: [A, C]  
C: [A, B, D]  
D: [C]

Adjacency Matrix

|   | A | B | C | D |
|---|---|---|---|---|
| A | 0 | 1 | 1 | 0 |
| B | 1 | 0 | 1 | 0 |
| C | 1 | 1 | 0 | 1 |
| D | 0 | 0 | 1 | 0 |

## **Tiny Code (Python)**

```

def build_graph(edge_list):
    graph = {}
    for u, v in edge_list:
        graph.setdefault(u, []).append(v)
        graph.setdefault(v, []).append(u) # undirected
    return graph

edges = [("A", "B"), ("A", "C"), ("B", "C"), ("C", "D")]
graph = build_graph(edges)
for node, neighbors in graph.items():
    print(node, ":", neighbors)

```

Output

```

A : ['B', 'C']
B : ['A', 'C']
C : ['A', 'B', 'D']
D : ['C']

```

## Why It Matters

- Graphs let us model relationships in any domain: roads, social networks, dependencies, knowledge.
- Once constructed, you can apply graph algorithms, BFS, DFS, shortest paths, spanning trees, connectivity, to solve real problems.
- The constructor phase defines how efficiently later algorithms run.

## A Gentle Proof (Why It Works)

Given  $n$  vertices and  $m$  edges, we represent each edge  $(u, v)$  by linking  $u$  and  $v$ . Construction time =  $O(n + m)$ , since each vertex and edge is processed once.

Adjacency list size =  $O(n + m)$  Adjacency matrix size =  $O(n^2)$

Thus, adjacency lists are more space-efficient for sparse graphs, while matrices offer constant-time edge lookups for dense graphs.

## Try It Yourself

1. Build a graph of 5 cities and their direct flights.
2. Represent it as both edge list and adjacency list.
3. Count number of edges and neighbors per vertex.
4. Draw the resulting graph on paper.

## Test Cases

| Input                   | Representation   | Key Property           |
|-------------------------|------------------|------------------------|
| $[(1, 2), (2, 3)]$      | Adjacency List   | 3 vertices, 2 edges    |
| Directed edges          | Adjacency List   | One-way links only     |
| Fully connected 3 nodes | Adjacency Matrix | All 1s except diagonal |

## Complexity

| Representation   | Space      | Lookup       | Iteration |
|------------------|------------|--------------|-----------|
| Edge List        | $O(m)$     | $O(m)$       | $O(m)$    |
| Adjacency List   | $O(n + m)$ | $O(\deg(v))$ | $O(m)$    |
| Adjacency Matrix | $O(n^2)$   | $O(1)$       | $O(n^2)$  |

A Graph Model Constructor builds the world of connections, from abstract relations to concrete data structures, forming the backbone of every graph algorithm that follows.

## 82 Adjacency Matrix Builder

An Adjacency Matrix Builder constructs a 2D grid representation of a graph, showing whether pairs of vertices are connected. It's a simple and powerful way to capture all edges in a compact mathematical form.

### What Problem Are We Solving?

We need a fast, systematic way to test if two vertices are connected. While adjacency lists are space-efficient, adjacency matrices make edge lookup  $O(1)$ , perfect when connections are dense or frequent checks are needed.

The Adjacency Matrix Builder gives us a table-like structure to store edge information clearly.

## How It Works (Plain Language)

An adjacency matrix is an  $n \times n$  table for a graph with  $n$  vertices:

$$A[i][j] = \begin{cases} 1, & \text{if there is an edge from } i \text{ to } j, \\ 0, & \text{otherwise.} \end{cases}$$

- For undirected graphs, the matrix is symmetric:  $A[i][j] = A[j][i]$
- For directed graphs, symmetry may not hold
- For weighted graphs, store weights instead of 1s

## Example

Vertices:  $V = A, B, C, D$  Edges:  $(A, B), (A, C), (B, C), (C, D)$

Adjacency Matrix (Undirected)

|   | A | B | C | D |
|---|---|---|---|---|
| A | 0 | 1 | 1 | 0 |
| B | 1 | 0 | 1 | 0 |
| C | 1 | 1 | 0 | 1 |
| D | 0 | 0 | 1 | 0 |

To check if A and C are connected, test  $A[A][C] = 1$

## Tiny Code (Python)

```
def adjacency_matrix(vertices, edges, directed=False):
    n = len(vertices)
    index = {v: i for i, v in enumerate(vertices)}
    A = [[0] * n for _ in range(n)]

    for u, v in edges:
        i, j = index[u], index[v]
        A[i][j] = 1
        if not directed:
            A[j][i] = 1
    return A
```

```

vertices = ["A", "B", "C", "D"]
edges = [("A", "B"), ("A", "C"), ("B", "C"), ("C", "D")]
A = adjacency_matrix(vertices, edges)
for row in A:
    print(row)

```

Output

```

[0, 1, 1, 0]
[1, 0, 1, 0]
[1, 1, 0, 1]
[0, 0, 1, 0]

```

## Why It Matters

- Constant-time check for edge existence
- Simple mathematical representation for graph algorithms and proofs
- Foundation for matrix-based graph algorithms like:
  - Floyd–Warshall (all-pairs shortest path)
  - Adjacency matrix powers (reachability)
  - Spectral graph theory (Laplacian, eigenvalues)

## A Gentle Proof (Why It Works)

Each vertex pair  $(u, v)$  corresponds to one matrix cell  $A[i][j]$ . We visit each edge once to set two symmetric entries (undirected) or one (directed). Thus:

- Time complexity:  $O(n^2)$  to initialize,  $O(m)$  to fill
- Space complexity:  $O(n^2)$

This tradeoff is worth it when  $m \approx n^2$  (dense graphs).

## Try It Yourself

1. Build an adjacency matrix for a directed triangle ( $A \rightarrow B$ ,  $B \rightarrow C$ ,  $C \rightarrow A$ )
2. Modify it to add a self-loop on  $B$
3. Check if  $A[B][B] = 1$
4. Compare the symmetry of directed vs undirected graphs

## Test Cases

| Graph Type | Edges     | Symmetry      | Value       |
|------------|-----------|---------------|-------------|
| Undirected | (A,B)     | Symmetric     | A[B][A] = 1 |
| Directed   | (A,B)     | Not symmetric | A[B][A] = 0 |
| Weighted   | (A,B,w=5) | Value stored  | A[A][B] = 5 |

## Complexity

| Operation         | Time     | Space    |
|-------------------|----------|----------|
| Build Matrix      | $O(n^2)$ | $O(n^2)$ |
| Edge Check        | $O(1)$   | -        |
| Iterate Neighbors | $O(n)$   | -        |

An Adjacency Matrix Builder turns a graph into a table, a universal structure for analysis, efficient queries, and algorithmic transformation.

## 83 Adjacency List Builder

An Adjacency List Builder constructs a flexible representation of a graph, storing each vertex's neighbors in a list. It's memory-efficient for sparse graphs and intuitive for traversal-based algorithms.

### What Problem Are We Solving?

We need a way to represent graphs compactly while still supporting quick traversal of connected vertices. When graphs are sparse (few edges compared to  $n^2$ ), an adjacency matrix wastes space. An adjacency list focuses only on existing edges, making it both lean and intuitive.

### How It Works (Plain Language)

Each vertex keeps a list of all vertices it connects to. In a directed graph, edges point one way; in an undirected graph, each edge appears twice.

For a graph with vertices  $V$  and edges  $E$ , the adjacency list is:

$$\text{Adj}[u] = v \mid (u, v) \in E$$

You can think of it as a dictionary (or map) where each key is a vertex, and its value is a list of neighbors.

### Example

Vertices:  $V = A, B, C, D$  Edges:  $(A, B), (A, C), (B, C), (C, D)$

Adjacency List (Undirected)

```
A: [B, C]
B: [A, C]
C: [A, B, D]
D: [C]
```

### Tiny Code (Python)

```
def adjacency_list(vertices, edges, directed=False):
    adj = {v: [] for v in vertices}
    for u, v in edges:
        adj[u].append(v)
        if not directed:
            adj[v].append(u)
    return adj

vertices = ["A", "B", "C", "D"]
edges = [("A", "B"), ("A", "C"), ("B", "C"), ("C", "D")]

graph = adjacency_list(vertices, edges)
for node, nbrs in graph.items():
    print(f"{node}: {nbrs}")
```

Output

```
A: ['B', 'C']
B: ['A', 'C']
C: ['A', 'B', 'D']
D: ['C']
```

## Why It Matters

- Space-efficient for sparse graphs ( $O(n + m)$ )
- Natural fit for DFS, BFS, and pathfinding
- Easy to modify and extend (weighted edges, labels)
- Forms the basis for graph traversal algorithms and network models

## A Gentle Proof (Why It Works)

Each edge is stored exactly once (directed) or twice (undirected). If  $n$  is the number of vertices and  $m$  is the number of edges:

- Initialization:  $O(n)$
- Insertion:  $O(m)$
- Total Space:  $O(n + m)$

No wasted space for missing edges, each list grows only with actual neighbors.

## Try It Yourself

1. Build an adjacency list for a directed graph with edges (A→B, A→C, C→A)
2. Add a new vertex E with no edges; confirm it still appears as E: []
3. Count how many total neighbors there are, it should match the edge count

## Test Cases

| Graph Type | Input Edges | Representation |
|------------|-------------|----------------|
| Undirected | (A,B)       | A: [B], B: [A] |
| Directed   | (A,B)       | A: [B], B: []  |
| Weighted   | (A,B,5)     | A: [(B,5)]     |

## Complexity

| Operation       | Time         | Space      |
|-----------------|--------------|------------|
| Build List      | $O(n + m)$   | $O(n + m)$ |
| Check Neighbors | $O(\deg(v))$ | -          |
| Add Edge        | $O(1)$       | -          |
| Remove Edge     | $O(\deg(v))$ | -          |

An Adjacency List Builder keeps your graph representation clean and scalable, perfect for algorithms that walk, explore, and connect the dots across large networks.

## 84 Degree Counter

A Degree Counter computes how many edges touch each vertex in a graph. For undirected graphs, the degree is the number of neighbors. For directed graphs, we distinguish between in-degree and out-degree.

### What Problem Are We Solving?

We want to know how connected each vertex is. Degree counts help answer structural questions:

- Is the graph regular (all vertices same degree)?
- Are there sources (zero in-degree) or sinks (zero out-degree)?
- Which node is a hub in a network?

These insights are foundational for traversal, centrality, and optimization.

### How It Works (Plain Language)

For each edge  $(u, v)$ :

- Undirected: increment `degree[u]` and `degree[v]`
- Directed: increment `out_degree[u]` and `in_degree[v]`

When done, every vertex has its connection count.

### Example

Undirected graph:

$$V = A, B, C, D, \quad E = (A, B), (A, C), (B, C), (C, D)$$

| Vertex | Degree |
|--------|--------|
| A      | 2      |
| B      | 2      |
| C      | 3      |
| D      | 1      |

Directed version:

- In-degree(A)=1 (from C), Out-degree(A)=2 (to B,C)

### Tiny Code (Python)

```
def degree_counter(vertices, edges, directed=False):
    if directed:
        indeg = {v: 0 for v in vertices}
        outdeg = {v: 0 for v in vertices}
        for u, v in edges:
            outdeg[u] += 1
            indeg[v] += 1
        return indeg, outdeg
    else:
        deg = {v: 0 for v in vertices}
        for u, v in edges:
            deg[u] += 1
            deg[v] += 1
        return deg

vertices = ["A", "B", "C", "D"]
edges = [("A", "B"), ("A", "C"), ("B", "C"), ("C", "D")]
print(degree_counter(vertices, edges))
```

Output

```
{'A': 2, 'B': 2, 'C': 3, 'D': 1}
```

### Why It Matters

- Reveals connectivity patterns
- Identifies isolated nodes
- Enables graph classification (regular, sparse, dense)
- Essential for graph algorithms (topological sort, PageRank, BFS pruning)

## A Gentle Proof (Why It Works)

In any undirected graph, the sum of all degrees equals twice the number of edges:

$$\sum_{v \in V} \deg(v) = 2|E|$$

In directed graphs:

$$\sum_{v \in V} \text{in}(v) = \sum_{v \in V} \text{out}(v) = |E|$$

These equalities guarantee correctness, every edge contributes exactly once (or twice if undirected).

## Try It Yourself

1. Create an undirected graph with edges (A,B), (B,C), (C,A)
  - Verify all vertices have degree 2
2. Add an isolated vertex D
  - Check that its degree is 0
3. Convert to directed edges and count in/out separately

## Test Cases

| Graph         | Input Edges      | Output                               |
|---------------|------------------|--------------------------------------|
| Undirected    | (A,B), (A,C)     | A:2, B:1, C:1                        |
| Directed      | (A,B), (B,C)     | in(A)=0, out(A)=1; in(C)=1, out(C)=0 |
| Isolated Node | (A,B), V={A,B,C} | C:0                                  |

## Complexity

| Operation     | Time   | Space  |
|---------------|--------|--------|
| Count Degrees | $O(m)$ | $O(n)$ |
| Lookup Degree | $O(1)$ | -      |

A Degree Counter exposes the heartbeat of a graph, showing which nodes are busy, which are lonely, and how the network's structure unfolds.

## 85 Path Existence Tester

A Path Existence Tester checks whether there is a route between two vertices in a graph, whether you can travel from a source to a destination by following edges.

### What Problem Are We Solving?

In many scenarios, navigation, dependency resolution, communication, the essential question is: "Can we get from A to B?"

This is not about finding the *shortest* path, but simply checking if a path *exists* at all.

Examples:

- Is a file accessible from the root directory?
- Can data flow between two nodes in a network?
- Does a dependency graph contain a reachable edge?

### How It Works (Plain Language)

We use graph traversal to explore from the source node. If the destination is reached, a path exists.

Steps:

1. Choose a traversal (DFS or BFS)
2. Start from source node  $s$
3. Mark visited nodes
4. Traverse neighbors recursively (DFS) or level by level (BFS)
5. If destination  $t$  is visited, a path exists

## Example

Graph:

$$V = A, B, C, D, \quad E = (A, B), (B, C), (C, D)$$

Query: Is there a path from A to D?

Traversal (DFS or BFS):

- Start at A → B → C → D
- D is reached → Path exists

Query: Is there a path from D to A?

- Start at D → no outgoing edges → No path

## Tiny Code (Python)

```
from collections import deque

def path_exists(graph, source, target):
    visited = set()
    queue = deque([source])

    while queue:
        node = queue.popleft()
        if node == target:
            return True
        if node in visited:
            continue
        visited.add(node)
        queue.extend(graph.get(node, []))
    return False

graph = {
    "A": ["B"],
    "B": ["C"],
    "C": ["D"],
    "D": []
}
print(path_exists(graph, "A", "D")) # True
print(path_exists(graph, "D", "A")) # False
```

## Why It Matters

- Core to graph connectivity
- Used in cycle detection, topological sorting, and reachability queries
- Foundational in AI search, routing, compilers, and network analysis

## A Gentle Proof (Why It Works)

Let the graph be  $G = (V, E)$  and traversal be BFS or DFS. Every edge  $(u, v)$  is explored once. If a path exists, traversal will eventually reach all nodes in the connected component of  $s$ . Thus, if  $t$  lies in that component, it will be discovered.

Traversal completeness ensures correctness.

## Try It Yourself

1. Build a directed graph  $A \rightarrow B \rightarrow C$ , and check  $A \rightarrow C$  and  $C \rightarrow A$ .
2. Add an extra edge  $C \rightarrow A$ .
  - Now the graph is strongly connected.
  - Every node should reach every other node.
3. Visualize traversal using a queue or recursion trace.

## Test Cases

| Graph                           | Source | Target | Result |
|---------------------------------|--------|--------|--------|
| $A \rightarrow B \rightarrow C$ | A      | C      | True   |
| $A \rightarrow B \rightarrow C$ | C      | A      | False  |
| A B                             | A      | B      | True   |
| Disconnected                    | A      | D      | False  |

## Complexity

| Operation | Time       | Space  |
|-----------|------------|--------|
| BFS / DFS | $O(n + m)$ | $O(n)$ |

$n$  = vertices,  $m$  = edges.

A Path Existence Tester is the simplest yet most powerful diagnostic for graph connectivity, revealing whether two points belong to the same connected world.

## 86 Tree Validator

A Tree Validator checks whether a given graph satisfies the defining properties of a tree: it is connected and acyclic.

### What Problem Are We Solving?

We often encounter structures that *look* like trees, but we must confirm they truly are. For example:

- Can this dependency graph be represented as a tree?
- Is the given parent-child relation a valid hierarchy?
- Does this undirected graph contain cycles or disconnected parts?

A Tree Validator formalizes that check.

A tree must satisfy:

1. Connectivity: every vertex reachable from any other.
2. Acyclicity: no cycles exist.
3. (Equivalently for undirected graphs)

$$|E| = |V| - 1$$

## How It Works (Plain Language)

We can validate using traversal and counting:

Method 1: DFS + Parent Check

1. Start DFS from any node.
2. Track visited nodes.
3. If a neighbor is visited *and not parent*, a cycle exists.
4. After traversal, check all nodes visited (connectedness).

Method 2: Edge–Vertex Property

1. Check if graph has exactly  $|V| - 1$  edges.
2. Run DFS/BFS to ensure graph is connected.

## Example

Graph 1:

$$V = A, B, C, D, \quad E = (A, B), (A, C), (B, D)$$

- $|V| = 4, |E| = 3$
- Connected, no cycle  $\rightarrow$  Tree

Graph 2:

$$V = A, B, C, \quad E = (A, B), (B, C), (C, A)$$

- $|V| = 3, |E| = 3$
- Cycle present  $\rightarrow$  Not a tree

## Tiny Code (Python)

```
def is_tree(graph):  
    n = len(graph)  
    visited = set()  
    parent = []  
  
    def dfs(node, par):  
        visited.add(node)  
        for nbr in graph[node]:  
            if nbr == par:  
                continue
```

```

        if nbr in visited:
            return False # cycle detected
        if not dfs(nbr, node):
            return False
    return True

# Start from first node
start = next(iter(graph))
if not dfs(start, None):
    return False

# Check connectivity
return len(visited) == n

```

Example:

```

graph = {
    "A": ["B", "C"],
    "B": ["A", "D"],
    "C": ["A"],
    "D": ["B"]
}
print(is_tree(graph)) # True

```

## Why It Matters

Tree validation ensures:

- Hierarchies are acyclic
- Data structures (like ASTs, tries) are well-formed
- Network topologies avoid redundant links
- Algorithms relying on tree properties (DFS order, LCA, spanning tree) are safe

## A Gentle Proof (Why It Works)

A connected graph without cycles is a tree. Inductive reasoning:

- Base: single node, zero edges, trivially a tree.
- Induction: adding one edge that connects a new node preserves acyclicity. If a cycle forms, it violates tree property.

Also, for undirected graph:

$$\text{Tree} \iff \text{Connected} \wedge |E| = |V| - 1$$

### Try It Yourself

1. Draw a small graph with 4 nodes.
2. Add edges one by one.
  - After each addition, test if graph is still a tree.
3. Introduce a cycle and rerun validator.
4. Remove an edge and check connectivity failure.

### Test Cases

| Graph         | Connected | Cycle | Tree |
|---------------|-----------|-------|------|
| A–B–C         | ✓         | ✗     | ✗    |
| A–B, B–C, C–A | ✗         | ✗     | ✗    |
| A–B, C        | ✓         | ✗     | ✗    |
| Single Node   | ✓         | ✗     | ✗    |

### Complexity

| Operation | Time       | Space  |
|-----------|------------|--------|
| DFS       | $O(n + m)$ | $O(n)$ |

A Tree Validator ensures structure, order, and simplicity, the quiet geometry behind every hierarchy.

## 86 Tree Validator

A Tree Validator checks whether a given graph satisfies the defining properties of a tree: it is connected and acyclic.

## What Problem Are We Solving?

We often encounter structures that *look* like trees, but we must confirm they truly are. For example:

- Can this dependency graph be represented as a tree?
- Is the given parent–child relation a valid hierarchy?
- Does this undirected graph contain cycles or disconnected parts?

A Tree Validator formalizes that check.

A tree must satisfy:

1. Connectivity: every vertex reachable from any other.
2. Acyclicity: no cycles exist.
3. (Equivalently for undirected graphs)

$$|E| = |V| - 1$$

## How It Works (Plain Language)

We can validate using traversal and counting.

Method 1: DFS + Parent Check

1. Start DFS from any node.
2. Track visited nodes.
3. If a neighbor is visited *and not parent*, a cycle exists.
4. After traversal, check all nodes visited (connectedness).

Method 2: Edge–Vertex Property

1. Check if graph has exactly  $|V| - 1$  edges.
2. Run DFS or BFS to ensure graph is connected.

## Example

Graph 1:

$$V = A, B, C, D, \quad E = (A, B), (A, C), (B, D)$$

- $|V| = 4, |E| = 3$
- Connected, no cycle  $\rightarrow$  Tree

Graph 2:

$$V = A, B, C, \quad E = (A, B), (B, C), (C, A)$$

- $|V| = 3, |E| = 3$
- Cycle present  $\rightarrow$  Not a tree

### Tiny Code (Python)

```
def is_tree(graph):
    n = len(graph)
    visited = set()
    parent = {}

    def dfs(node, par):
        visited.add(node)
        for nbr in graph[node]:
            if nbr == par:
                continue
            if nbr in visited:
                return False # cycle detected
            if not dfs(nbr, node):
                return False
        return True

    # Start from first node
    start = next(iter(graph))
    if not dfs(start, None):
        return False

    # Check connectivity
    return len(visited) == n
```

Example:

```
graph = {
    "A": ["B", "C"],
    "B": ["A", "D"],
    "C": ["A"],
    "D": ["B"]
}
print(is_tree(graph)) # True
```

## Why It Matters

Tree validation ensures:

- Hierarchies are acyclic
- Data structures (like ASTs, tries) are well-formed
- Network topologies avoid redundant links
- Algorithms relying on tree properties (DFS order, LCA, spanning tree) are safe

## A Gentle Proof (Why It Works)

A connected graph without cycles is a tree. Inductive reasoning:

- Base: single node, zero edges, trivially a tree.
- Induction: adding one edge that connects a new node preserves acyclicity. If a cycle forms, it violates the tree property.

Also, for an undirected graph:

$$\text{Tree} \iff \text{Connected} \wedge |E| = |V| - 1$$

## Try It Yourself

1. Draw a small graph with 4 nodes.
2. Add edges one by one.
  - After each addition, test if the graph is still a tree.
3. Introduce a cycle and rerun the validator.
4. Remove an edge and check for connectivity failure.

## Test Cases

| Graph         | Connected | Cycle | Tree |
|---------------|-----------|-------|------|
| A–B–C         | Yes       | No    | Yes  |
| A–B, B–C, C–A | Yes       | Yes   | No   |
| A–B, C        | No        | No    | No   |
| Single Node   | Yes       | No    | Yes  |

## Complexity

| Operation | Time       | Space  |
|-----------|------------|--------|
| DFS       | $O(n + m)$ | $O(n)$ |

A Tree Validator ensures structure, order, and simplicity, the quiet geometry behind every hierarchy.

## 87 Rooted Tree Builder

A Rooted Tree Builder constructs a tree from a given parent array or edge list, designating one node as the root and connecting all others accordingly.

### What Problem Are We Solving?

Often we receive data in *flat* form—like a list of parent indices, database references, or parent–child pairs—and we need to reconstruct the actual tree structure.

For example:

- A parent array [ -1, 0, 0, 1, 1, 2 ] represents which node is parent of each.
- In file systems, each directory knows its parent; we need to rebuild the hierarchy.

The Rooted Tree Builder formalizes this reconstruction.

### How It Works (Plain Language)

A parent array encodes each node's parent:

- `parent[i] = j` means node `j` is the parent of `i`.
- If `parent[i] = -1`, then `i` is the root.

Steps:

1. Find the root (the node with parent `-1`).
2. Initialize an adjacency list `children` for each node.
3. For each node `i`:
  - If `parent[i] != -1`, append `i` to `children[parent[i]]`.

4. Output the adjacency structure.

This gives a tree with parent–child relationships.

### Example

Parent array:

```
Index:  0  1  2  3  4  5  
Parent: -1  0  0  1  1  2
```

Interpretation:

- 0 is root.
- 1 and 2 are children of 0.
- 3 and 4 are children of 1.
- 5 is child of 2.

Tree:

```
0  
|  
1  
|  
3  
|  
4  
|  
2  
|  
5
```

### Tiny Code (Python)

```
def build_tree(parent):  
    n = len(parent)  
    children = [[] for _ in range(n)]  
    root = None  
  
    for i in range(n):  
        if parent[i] == -1:  
            root = i  
        else:  
            children[parent[i]].append(i)  
  
    return root, children
```

Example:

```
parent = [-1, 0, 0, 1, 1, 2]
root, children = build_tree(parent)

print("Root:", root)
for i, c in enumerate(children):
    print(f"{i}: {c}")
```

Output:

```
Root: 0
0: [1, 2]
1: [3, 4]
2: [5]
3: []
4: []
5: []
```

## Why It Matters

Tree reconstruction is foundational in:

- Compilers: abstract syntax tree (AST) reconstruction
- Databases: reconstructing hierarchical relationships
- Operating systems: file directory trees
- Organization charts: building hierarchies from parent–child data

It connects linear storage to hierarchical structure.

## A Gentle Proof (Why It Works)

If the parent array satisfies:

- Exactly one root: one entry with -1
- All other nodes have exactly one parent
- The resulting structure is connected and acyclic

Then the output is a valid rooted tree:

$$|E| = |V| - 1, \text{ and exactly one node has no parent.}$$

Each child is linked once, forming a tree rooted at the unique node with -1.

## Try It Yourself

1. Write your own parent array (e.g., [ -1, 0, 0, 1, 2 ]).
2. Convert it into a tree.
3. Draw the hierarchy manually.
4. Verify connectivity and acyclicity.

## Test Cases

| Parent Array       | Root | Children Structure      |
|--------------------|------|-------------------------|
| [ -1, 0, 0, 1, 2 ] | 0    | 0:[1,2], 1:[3,4], 2:[5] |
| [ -1, 0, 1, 2 ]    | 0    | 0:[1], 1:[2], 2:[3]     |
| [ -1 ]             | 0    | 0:[ ]                   |

## Complexity

| Operation | Time   | Space  |
|-----------|--------|--------|
| Build     | $O(n)$ | $O(n)$ |

The Rooted Tree Builder bridges the gap between flat data and hierarchical form, turning arrays into living structures.

## 88 Traversal Order Visualizer

A Traversal Order Visualizer shows how different tree traversals (preorder, inorder, postorder, level order) explore nodes, revealing the logic behind recursive and iterative visits.

## What Problem Are We Solving?

When working with trees, the order of visiting nodes matters. Different traversals serve different goals:

- Preorder: process parent before children
- Inorder: process left child, then parent, then right child
- Postorder: process children before parent
- Level order: visit nodes breadth-first

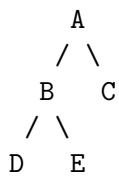
Understanding these traversals helps in:

- Expression parsing
- File system navigation
- Tree printing and evaluation

A visualizer clarifies *when* and *why* each node is visited.

### How It Works (Plain Language)

Consider a binary tree:



Each traversal orders nodes differently:

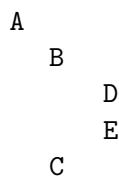
| Traversal   | Order         |
|-------------|---------------|
| Preorder    | A, B, D, E, C |
| Inorder     | D, B, E, A, C |
| Postorder   | D, E, B, C, A |
| Level order | A, B, C, D, E |

Visualization strategy:

- Start at the root.
- Use recursion (depth-first) or queue (breadth-first).
- Record each visit step.
- Output sequence in order visited.

### Example Step by Step

Tree:



Preorder

1. Visit A
2. Visit B
3. Visit D
4. Visit E
5. Visit C

Sequence: A B D E C

Inorder

1. Traverse left subtree of A (B)
2. Traverse left of B (D) → visit D
3. Visit B
4. Traverse right of B (E) → visit E
5. Visit A
6. Visit right subtree (C)

Sequence: D B E A C

### Tiny Code (Python)

```
class Node:  
    def __init__(self, val):  
        self.val = val  
        self.left = None  
        self.right = None  
  
def preorder(root):  
    if not root:  
        return []  
    return [root.val] + preorder(root.left) + preorder(root.right)  
  
def inorder(root):  
    if not root:  
        return []  
    return inorder(root.left) + [root.val] + inorder(root.right)  
  
def postorder(root):  
    if not root:  
        return []
```

```

    return postorder(root.left) + postorder(root.right) + [root.val]

def level_order(root):
    if not root:
        return []
    queue = [root]
    result = []
    while queue:
        node = queue.pop(0)
        result.append(node.val)
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)
    return result

```

## Why It Matters

Traversal order determines:

- Computation sequence (evaluation, deletion, printing)
- Expression tree evaluation (postorder)
- Serialization/deserialization (preorder + inorder)
- Breadth-first exploration (level order)

Understanding traversal = understanding how algorithms move through structure.

## A Gentle Proof (Why It Works)

Each traversal is a systematic walk:

- Preorder ensures root-first visitation.
- Inorder ensures sorted order in binary search trees.
- Postorder ensures children processed before parent.
- Level order ensures minimal depth-first layering.

Since each node is visited exactly once, correctness follows from recursion and induction.

## Try It Yourself

1. Build a binary tree with 5 nodes.
2. Write out all four traversals by hand.
3. Trace recursive calls step by step.
4. Observe how order changes per traversal.

## Test Cases

| Traversal   | Example Tree | Expected Order |
|-------------|--------------|----------------|
| Preorder    | A-B-C        | A B C          |
| Inorder     | A-B-C        | B A C          |
| Postorder   | A-B-C        | B C A          |
| Level order | A-B-C        | A B C          |

## Complexity

| Operation         | Time   | Space          |
|-------------------|--------|----------------|
| DFS (Pre/In/Post) | $O(n)$ | $O(h)$ (stack) |
| BFS (Level)       | $O(n)$ | $O(n)$ (queue) |

The Traversal Order Visualizer turns abstract definitions into motion, showing how structure guides computation.

## 89 Edge Classifier

An Edge Classifier determines the type of each edge encountered during a graph traversal, whether it is a tree edge, back edge, forward edge, or cross edge. This classification helps us understand the structure and flow of a directed or undirected graph.

### What Problem Are We Solving?

In graph algorithms, not all edges play the same role. When we traverse using DFS, we can interpret the relationship between vertices based on discovery times.

Edge classification helps answer questions like:

- Is there a cycle? (Look for back edges)

- How is the graph structured? (Tree vs forward edges)
- Is this DAG (Directed Acyclic Graph)? (No back edges)
- What's the hierarchical relation between nodes?

By tagging edges, we gain structural insight into traversal behavior.

### How It Works (Plain Language)

During DFS, we assign each vertex:

- Discovery time when first visited.
- Finish time when exploration completes.

Each edge  $(u, v)$  is then classified as:

| Type         | Condition                                      |
|--------------|------------------------------------------------|
| Tree edge    | $v$ is first discovered by $(u, v)$            |
| Back edge    | $v$ is ancestor of $u$ (cycle indicator)       |
| Forward edge | $v$ is descendant of $u$ , but already visited |
| Cross edge   | $v$ is neither ancestor nor descendant of $u$  |

In undirected graphs, only tree and back edges occur.

### Example

Graph (directed):

```
1 → 2 → 3
↑   ↓
4 ← 5
```

During DFS starting at 1:

- (1,2): tree edge
- (2,3): tree edge
- (3,4): back edge (cycle 1–2–3–4–1)
- (2,5): tree edge
- (5,4): tree edge
- (4,1): back edge

So we detect cycles due to back edges.

## Tiny Code (Python)

```
def classify_edges(graph):
    time = 0
    discovered = {}
    finished = {}
    classification = []

    def dfs(u):
        nonlocal time
        time += 1
        discovered[u] = time
        for v in graph[u]:
            if v not in discovered:
                classification.append(((u, v), "Tree"))
                dfs(v)
            elif v not in finished:
                classification.append(((u, v), "Back"))
            elif discovered[u] < discovered[v]:
                classification.append(((u, v), "Forward"))
            else:
                classification.append(((u, v), "Cross"))
        time += 1
        finished[u] = time

    for node in graph:
        if node not in discovered:
            dfs(node)
    return classification
```

## Why It Matters

Edge classification underpins:

- Cycle detection (look for back edges)
- Topological sorting (DAGs have no back edges)
- DFS tree structure analysis
- Strongly connected component detection

It converts traversal into structural insight.

## A Gentle Proof (Why It Works)

DFS imposes a temporal order on discovery and finish times. An edge  $(u, v)$  can only fall into one of the four categories because:

Each vertex has a distinct discovery and finish time interval.

By comparing intervals  $(d[u], f[u])$  and  $(d[v], f[v])$ , we deduce whether  $v$  lies inside, before, or after  $u$ 's traversal window.

## Try It Yourself

1. Draw a small directed graph.
2. Assign discovery/finish times using DFS.
3. Compare intervals for each edge.
4. Label each edge as Tree, Back, Forward, or Cross.
5. Verify that DAGs have no back edges.

## Test Cases

| Edge   | Type |
|--------|------|
| (A, B) | Tree |
| (B, C) | Tree |
| (C, A) | Back |
| (B, D) | Tree |
| (D, E) | Tree |
| (E, B) | Back |

## Complexity

| Operation      | Time       | Space  |
|----------------|------------|--------|
| DFS Traversal  | $O(n + m)$ | $O(n)$ |
| Classification | $O(m)$     | $O(m)$ |

The Edge Classifier transforms traversal into topology, making invisible structures like cycles, hierarchies, and cross-links explicit.

## 90 Connectivity Checker

A Connectivity Checker determines whether a graph is connected, that is, whether every vertex can be reached from any other vertex. It's a fundamental diagnostic tool in graph theory and network analysis.

### What Problem Are We Solving?

Connectivity tells us whether the graph forms a single whole or multiple isolated parts.

We often ask:

- Can all nodes communicate in this network?
- Is this maze solvable from start to end?
- Does this undirected graph form one component or many?
- For directed graphs: can we reach every vertex from every other vertex?

The Connectivity Checker gives a yes/no answer, and can also enumerate connected components.

### How It Works (Plain Language)

Undirected Graph:

1. Pick a starting node.
2. Perform DFS or BFS, marking all reachable nodes.
3. After traversal, if all nodes are marked, the graph is connected.

Directed Graph:

- Use two traversals:
  1. Run DFS from any node. If not all nodes are visited, not strongly connected.
  2. Reverse all edges and run DFS again. If still not all nodes are visited, not strongly connected.

Alternatively, detect strongly connected components (SCCs) via Kosaraju's or Tarjan's algorithm.

### **Example (Undirected)**

Graph 1:

```
1, 2, 3  
|      |  
4, 5, 6
```

All nodes reachable → Connected.

Graph 2:

```
1, 2      3, 4
```

Two separate parts → Not connected.

### **Example (Directed)**

Graph:

```
1 → 2 → 3  
↑      ↓
```

Every node reachable from every other → Strongly connected

Graph:

```
1 → 2 → 3
```

No path from 3 → 1 → Not strongly connected

### **Tiny Code (Python)**

```

from collections import deque

def is_connected(graph):
    n = len(graph)
    visited = set()

    # BFS from first node
    start = next(iter(graph))
    queue = deque([start])
    while queue:
        u = queue.popleft()
        if u in visited:
            continue
        visited.add(u)
        for v in graph[u]:
            if v not in visited:
                queue.append(v)

    return len(visited) == n

```

Example:

```

graph = {
    1: [2, 4],
    2: [1, 3],
    3: [2, 6],
    4: [1, 5],
    5: [4, 6],
    6: [3, 5]
}
print(is_connected(graph))  # True

```

## Why It Matters

Connectivity is central in:

- Network reliability, ensure all nodes communicate
- Graph algorithms, many assume connected graphs
- Clustering, find connected components
- Pathfinding, unreachable nodes signal barriers

It's often the *first diagnostic check* before deeper analysis.

## A Gentle Proof (Why It Works)

For undirected graphs, connectivity is equivalence relation:

- Reflexive: node connects to itself
- Symmetric: if A connects to B, B connects to A
- Transitive: if A connects to B and B connects to C, A connects to C

Therefore, DFS/BFS reachability partitioning defines connected components uniquely.

## Try It Yourself

1. Draw a graph with 6 nodes.
2. Run BFS or DFS from node 1.
3. Mark all reachable nodes.
4. If some remain unvisited, you've found multiple components.
5. For directed graphs, try reversing edges and retesting.

## Test Cases

| Graph      | Type       | Result                 |
|------------|------------|------------------------|
| 1–2–3      | Undirected | Connected              |
| 1–2, 3–4   | Undirected | Not Connected          |
| 1→2→3, 3→1 | Directed   | Strongly Connected     |
| 1→2→3      | Directed   | Not Strongly Connected |

## Complexity

| Operation | Time       | Space  |
|-----------|------------|--------|
| DFS/BFS   | $O(n + m)$ | $O(n)$ |

A Connectivity Checker ensures your graph is a single story, not a collection of isolated tales, a foundation before every journey through the graph.

## Section 10. Algorithm Design Patterns

### 91 Brute Force Pattern

The Brute Force Pattern is the simplest and most universal approach to problem-solving: try every possible option, evaluate them all, and pick the best. It trades computational efficiency for conceptual clarity and correctness.

#### What Problem Are We Solving?

Sometimes, before clever optimizations or heuristics, we need a baseline solution, a way to ensure correctness. The brute force approach guarantees finding the right answer by exploring all possible configurations, even if it's slow.

Common use cases:

- Exhaustive search (e.g., generating all permutations or subsets)
- Baseline testing before implementing heuristics
- Proving optimality by comparison

#### How It Works (Plain Language)

A brute force algorithm generally follows this structure:

1. Enumerate all candidate solutions.
2. Evaluate each candidate for validity or cost.
3. Select the best (or first valid) solution.

This is conceptually simple, though often expensive in time.

#### Example: Traveling Salesman Problem (TSP)

Given  $n$  cities and distances between them, find the shortest tour visiting all.

Brute force solution:

1. Generate all  $n!$  possible tours.
2. Compute the total distance for each.
3. Return the shortest tour.

This ensures correctness but grows factorially in complexity.

## Tiny Code (Python)

```
from itertools import permutations

def tsp_bruteforce(dist):
    n = len(dist)
    cities = list(range(n))
    best = float('inf')
    best_path = None

    for perm in permutations(cities[1:]): # fix city 0 as start
        path = [0] + list(perm) + [0]
        cost = sum(dist[path[i]][path[i+1]] for i in range(n))
        if cost < best:
            best = cost
            best_path = path
    return best, best_path

# Example distance matrix
dist = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
]

print(tsp_bruteforce(dist)) # (80, [0, 1, 3, 2, 0])
```

## Why It Matters

Brute force is valuable for:

- Correctness: guarantees the right answer.
- Benchmarking: provides a ground truth for optimization.
- Small inputs: often feasible when  $n$  is small.
- Teaching: clarifies the structure of search and evaluation.

It is the seed from which more refined algorithms (like DP, backtracking, and heuristics) evolve.

## A Gentle Proof (Why It Works)

Let  $S$  be the finite set of all possible solutions. If the algorithm evaluates every  $s \in S$  and correctly computes its quality, and selects the minimum (or maximum), the chosen  $s^*$  is provably optimal:

$$s^* = \arg \min_{s \in S} f(s)$$

Completeness and correctness are inherent, though efficiency is not.

## Try It Yourself

1. Enumerate all subsets of 1, 2, 3.
2. Check which subsets sum to 4.
3. Confirm all possibilities are considered.
4. Reflect on the time cost:  $2^n$  subsets for  $n$  elements.

## Test Cases

| Problem    | Input Size | Feasible? | Notes                         |
|------------|------------|-----------|-------------------------------|
| TSP        | $n = 4$    |           | $4! = 24$ paths               |
| TSP        | $n = 10$   |           | $10! \approx 3.6 \times 10^6$ |
| Subset Sum | $n = 10$   |           | $2^{10} = 1024$ subsets       |
| Subset Sum | $n = 30$   |           | $2^{30} \approx 10^9$ subsets |

## Complexity

| Operation   | Time              | Space  |
|-------------|-------------------|--------|
| Enumeration | $O(k^n)$ (varies) | $O(n)$ |

The Brute Force Pattern is the blank canvas of algorithmic design: simple, exhaustive, and pure, a way to guarantee truth before seeking elegance.

## 92 Greedy Pattern

The Greedy Pattern builds a solution step by step, choosing at each stage the locally optimal move, the one that seems best right now, with the hope (and often the proof) that this path leads to a globally optimal result.

## **What Problem Are We Solving?**

Greedy algorithms are used when problems exhibit two key properties:

1. Greedy-choice property – a global optimum can be reached by choosing local optima.
2. Optimal substructure – an optimal solution contains optimal solutions to subproblems.

You'll meet greedy reasoning everywhere: scheduling, pathfinding, compression, and resource allocation.

## **How It Works (Plain Language)**

Greedy thinking is “take the best bite each time.” There’s no looking back, no exploring alternatives, just a sequence of decisive moves.

General shape:

1. Start with an empty or initial solution.
2. Repeatedly choose the best local move (by some rule).
3. Stop when no more moves are possible or desired.

## **Example: Coin Change (Canonical Coins)**

Given coins 25, 10, 5, 1, make change for 63 cents.

Greedy approach:

- Take largest coin  $\leq$  remaining value.
- Subtract and repeat. Result:  $25 + 25 + 10 + 1 + 1 + 1 = 63$  (6 coins total)

Works for canonical systems, not all, a nice teaching point.

## **Tiny Code (Python)**

```
def greedy_coin_change(coins, amount):
    result = []
    for c in sorted(coins, reverse=True):
        while amount >= c:
            amount -= c
            result.append(c)
    return result
```

```
print(greedy_coin_change([25, 10, 5, 1], 63))
# [25, 25, 10, 1, 1]
```

## Why It Matters

The greedy pattern is a core design paradigm:

- Simple and fast – often linear or  $O(n \log n)$ .
- Provably optimal when conditions hold.
- Intuitive – builds insight into structure of problems.
- Foundation – many approximation and heuristic algorithms are “greedy at heart.”

## A Gentle Proof (Why It Works)

For problems with optimal substructure, we can often prove by induction:

If a greedy choice  $g$  leaves a subproblem  $P'$ , and

$$\text{OPT}(P) = g + \text{OPT}(P')$$

then solving  $P'$  optimally ensures global optimality.

For coin change with canonical coins, this holds since choosing a larger coin never prevents an optimal total.

## Try It Yourself

1. Apply the greedy method to Activity Selection: Sort activities by finishing time, pick earliest finishing one, and skip overlapping.
2. Compare against brute force enumeration.
3. Check if the greedy result is optimal, why or why not?

## Test Cases

| Problem                     | Greedy Works? | Note                                      |
|-----------------------------|---------------|-------------------------------------------|
| Activity Selection          |               | Local earliest-finish leads to global max |
| Coin Change (1, 3, 4) for 6 |               | 3+3 better than 4+1+1                     |
| Huffman Coding              |               | Greedy merging yields optimal tree        |
| Kruskal's MST               |               | Greedy edge selection builds MST          |

## Complexity

| Operation   | Time                    | Space  |
|-------------|-------------------------|--------|
| Selection   | $O(n \log n)$ (sorting) | $O(1)$ |
| Step Choice | $O(n)$                  | $O(1)$ |

The Greedy Pattern is the art of decisive reasoning, choosing what seems best now, and trusting the problem's structure to reward confidence.

## 93 Divide and Conquer Pattern

The Divide and Conquer Pattern breaks a big problem into smaller, similar subproblems, solves each one (often recursively), and then combines their results into the final answer.

It's the pattern behind merge sort, quicksort, binary search, and fast algorithms across mathematics and computation.

### What Problem Are We Solving?

We use divide and conquer when:

1. The problem can be split into smaller subproblems of the same type.
2. Those subproblems are independent and easier to solve.
3. Their solutions can be merged efficiently.

It's the algorithmic mirror of mathematical induction, reduce, solve, combine.

### How It Works (Plain Language)

Think of divide and conquer as a recursive three-step dance:

1. Divide – split the problem into smaller parts.
2. Conquer – solve each part recursively.
3. Combine – merge the sub-results into a final answer.

Each recursive call tackles a fraction of the work until reaching a base case.

## Example: Merge Sort

Sort an array  $A[1..n]$ .

1. Divide: split  $A$  into two halves.
2. Conquer: recursively sort each half.
3. Combine: merge the two sorted halves.

Recurrence:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Solution:

$$T(n) = O(n \log n)$$

## Tiny Code (Python)

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i]); i += 1
        else:
            result.append(right[j]); j += 1
    result.extend(left[i:]); result.extend(right[j:])
    return result
```

## Why It Matters

Divide and conquer turns recursion into efficiency. It's a framework for:

- Sorting (Merge Sort, Quick Sort)
- Searching (Binary Search)

- Matrix Multiplication (Strassen's Algorithm)
- FFT (Fast Fourier Transform)
- Geometry (Closest Pair, Convex Hull)
- Data Science (Divide-and-Conquer Regression, Decision Trees)

It captures the principle: *solve big problems by shrinking them.*

### A Gentle Proof (Why It Works)

Assume each subproblem of size  $\frac{n}{2}$  is solved optimally.

If we combine  $k$  subresults with cost  $f(n)$ , the total cost follows the recurrence

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Using the Master Theorem, we compare  $f(n)$  with  $n^{\log_b a}$  to find  $T(n)$ .

For merge sort:  $a = 2, b = 2, f(n) = n \quad T(n) = O(n \log n)$ .

### Try It Yourself

1. Apply divide and conquer to maximum subarray sum (Kadane's alternative).
2. Write a binary search with clear divide/conquer steps.
3. Visualize recursion tree and total cost at each level.

### Test Cases

| Problem       | Divide          | Combine          | Works Well? |
|---------------|-----------------|------------------|-------------|
| Merge Sort    | Split array     | Merge halves     |             |
| Quick Sort    | Partition array | Concatenate      | (average)   |
| Binary Search | Split range     | Return match     |             |
| Closest Pair  | Divide plane    | Compare boundary |             |

| Step | Cost |
|------|------|
|------|------|

## Complexity

| Step    | Cost             |
|---------|------------------|
| Divide  | $O(1)$ or $O(n)$ |
| Conquer | $aT(n/b)$        |
| Combine | $O(n)$ (typical) |

Overall:  $O(n \log n)$  in many classic cases.

Divide and conquer is the essence of recursive decomposition, see the whole by mastering the parts.

## 94 Dynamic Programming Pattern

The Dynamic Programming (DP) Pattern solves complex problems by breaking them into overlapping subproblems, solving each once, and storing results to avoid recomputation.

It transforms exponential recursive solutions into efficient polynomial ones through memoization or tabulation.

### What Problem Are We Solving?

When a problem has:

1. Overlapping subproblems – the same subtask appears multiple times.
2. Optimal substructure – an optimal solution can be built from optimal subsolutions.

Naive recursion repeats work. DP ensures each subproblem is solved once.

### How It Works (Plain Language)

Think of DP as smart recursion:

- Define a state that captures progress.
- Define a recurrence that relates larger states to smaller ones.
- Store results to reuse later.

Two main flavors:

1. Top-down (Memoization) – recursion with caching.
2. Bottom-up (Tabulation) – fill a table iteratively.

### Example: Fibonacci Numbers

Naive recursion:

$$F(n) = F(n - 1) + F(n - 2)$$

This recomputes many values.

DP solution:

1. Base:  $F(0) = 0, F(1) = 1$
2. Build up table:

$$F[i] = F[i - 1] + F[i - 2]$$

Result:  $O(n)$  time,  $O(n)$  space (or  $O(1)$  optimized).

### Tiny Code (Python)

```
def fib(n):
    dp = [0, 1] + [0]*(n-1)
    for i in range(2, n+1):
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n]
```

Or memoized recursion:

```
from functools import lru_cache

@lru_cache(None)
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)
```

## Why It Matters

DP is the core of algorithmic problem solving:

- Optimization: shortest paths, knapsack, edit distance
- Counting: number of ways to climb stairs, partitions
- Sequence analysis: LIS, LCS
- Resource allocation: scheduling, investment problems

It's how we bring structure to recursion.

## A Gentle Proof (Why It Works)

Let  $T(n)$  be the cost to solve all distinct subproblems. Since each is solved once and combined in constant time:

$$T(n) = O(\text{number of states}) \times O(\text{transition cost})$$

For Fibonacci:

- States =  $n$
- Transition cost =  $O(1)$     $T(n) = O(n)$

Memoization ensures every subproblem is visited at most once.

## Try It Yourself

1. Write DP for coin change (ways to form a sum).
2. Trace longest common subsequence (LCS) table.
3. Compare top-down vs bottom-up performance.

## Test Cases

| Problem       | State    | Transition                      | Time    |
|---------------|----------|---------------------------------|---------|
| Fibonacci     | $n$      | $dp[n] = dp[n - 1] + dp[n - 2]$ | $O(n)$  |
| Knapsack      | $(i, w)$ | max(take, skip)                 | $O(nW)$ |
| Edit Distance | $(i, j)$ | Compare chars                   | $O(nm)$ |

## Complexity

| Type                 | Time          | Space         |
|----------------------|---------------|---------------|
| Top-down Memoization | $O(\#states)$ | $O(\#states)$ |
| Bottom-up Tabulation | $O(\#states)$ | $O(\#states)$ |

Dynamic Programming is divide and conquer with memory, think recursively, compute once, reuse forever.

## 95 Backtracking Pattern

The Backtracking Pattern explores all possible solutions by building them step by step and abandoning a path as soon as it becomes invalid.

It's a systematic search strategy for problems where we need to generate combinations, permutations, or subsets, and prune impossible or suboptimal branches early.

### What Problem Are We Solving?

We face problems where:

- The solution space is large, but structured.
- We can detect invalid partial solutions early.

Examples:

- N-Queens (place queens safely)
- Sudoku (fill grid with constraints)
- Subset Sum (choose elements summing to target)

Brute force explores everything blindly. Backtracking cuts off dead ends as soon as they appear.

### How It Works (Plain Language)

Imagine exploring a maze:

1. Take a step (make a choice).
2. If it leads to a valid partial solution, continue.
3. If it fails, backtrack, undo and try another path.

Each level of recursion corresponds to a decision point.

### Example: N-Queens Problem

We need to place  $n$  queens on an  $n \times n$  board so no two attack each other.

At each row, choose a column that is safe. If none works, backtrack to previous row.

### Tiny Code (Python)

```
def solve_n_queens(n):
    res, board = [], [-1]*n

    def is_safe(row, col):
        for r in range(row):
            c = board[r]
            if c == col or abs(c - col) == abs(r - row):
                return False
        return True

    def backtrack(row=0):
        if row == n:
            res.append(board[:])
            return
        for col in range(n):
            if is_safe(row, col):
                board[row] = col
                backtrack(row + 1)
                board[row] = -1 # undo

    backtrack()
    return res
```

### Why It Matters

Backtracking is a universal solver for:

- Combinatorial search: subsets, permutations, partitions
- Constraint satisfaction: Sudoku, graph coloring, N-Queens
- Optimization with pruning (branch and bound builds on it)

It's not just brute force, it's guided exploration.

## A Gentle Proof (Why It Works)

Let  $S$  be the total number of possible states. Backtracking prunes all invalid paths early, so actual visited nodes  $\leq S$ .

If each state takes  $O(1)$  time to check and recurse, total complexity is proportional to the number of valid partial states, often far smaller than full enumeration.

## Try It Yourself

1. Solve Subset Sum using backtracking.
2. Generate all permutations of  $[1, 2, 3]$ .
3. Implement Sudoku Solver ( $9 \times 9$  constraint satisfaction).

Trace calls, each recursive call represents a partial decision.

## Test Cases

| Problem    | Decision        | Constraint             | Output         |
|------------|-----------------|------------------------|----------------|
| N-Queens   | Choose column   | Non-attacking queens   | Placements     |
| Subset Sum | Include/Exclude | Sum target             | Valid subsets  |
| Sudoku     | Fill cell       | Row/Col/Subgrid unique | Completed grid |

## Complexity

| Problem    | Time          | Space     |
|------------|---------------|-----------|
| N-Queens   | $O(n!)$ worst | $O(n)$    |
| Subset Sum | $O(2^n)$      | $O(n)$    |
| Sudoku     | Exponential   | Grid size |

Backtracking is the art of searching by undoing, try, test, and retreat until you find a valid path.

## 96 Branch and Bound

The Branch and Bound pattern is an optimization framework that systematically explores the search space while pruning paths that cannot yield better solutions than the best one found so far.

It extends backtracking with bounds that let us skip unpromising branches early.

### What Problem Are We Solving?

We want to solve optimization problems where:

- The search space is combinatorial (e.g., permutations, subsets).
- Each partial solution can be evaluated or bounded.
- We seek the best solution under some cost function.

Examples:

- Knapsack Problem: maximize value under capacity.
- Traveling Salesman Problem (TSP): find shortest tour.
- Job Scheduling: minimize total completion time.

Brute-force search is exponential. Branch and Bound cuts branches that cannot improve the best known answer.

### How It Works (Plain Language)

Think of exploring a tree:

1. Branch: expand possible choices.
2. Bound: compute a limit on achievable value from this branch.
3. If bound < best found so far, prune (stop exploring).
4. Otherwise, explore deeper.

We use:

- Upper bound: best possible value from this path.
- Lower bound: best value found so far.

Prune when upper bound < lower bound.

## Example: 0/1 Knapsack

Given items with weights and values, choose subset with max value capacity.

We recursively include/exclude each item, but prune branches that cannot beat current best (e.g., exceeding weight or potential value too low).

### Tiny Code (Python)

```
def knapsack_branch_bound(items, capacity):
    best_value = 0

    def bound(i, curr_w, curr_v):
        # Simple bound: add remaining items greedily
        if i >= len(items):
            return curr_v
        w, v = curr_w, curr_v
        for j in range(i, len(items)):
            if w + items[j][0] <= capacity:
                w += items[j][0]
                v += items[j][1]
        return v

    def dfs(i, curr_w, curr_v):
        nonlocal best_value
        if curr_w > capacity:
            return
        if curr_v > best_value:
            best_value = curr_v
        if i == len(items):
            return
        if bound(i, curr_w, curr_v) <= best_value:
            return
        # Include item
        dfs(i+1, curr_w + items[i][0], curr_v + items[i][1])
        # Exclude item
        dfs(i+1, curr_w, curr_v)

    dfs(0, 0, 0)
    return best_value
```

## Why It Matters

Branch and Bound:

- Generalizes backtracking with mathematical pruning.
- Turns exponential search into practical algorithms.
- Provides exact solutions when heuristics might fail.

Used in:

- Integer programming
- Route optimization
- Scheduling and assignment problems

## A Gentle Proof (Why It Works)

Let  $U(n)$  be an upper bound of a subtree. If  $U(n) \leq V^*$  (best known value), no solution below can exceed  $V^*$ .

By monotonic bounding, pruning preserves correctness — no optimal solution is ever discarded.

The algorithm is complete (explores all promising branches) and optimal (finds global best).

## Try It Yourself

1. Solve 0/1 Knapsack with branch and bound.
2. Implement TSP with cost matrix and prune by lower bounds.
3. Compare nodes explored vs brute-force enumeration.

## Test Cases

| Items (w,v)               | Capacity | Best Value | Branches Explored |
|---------------------------|----------|------------|-------------------|
| [(2,3),(3,4),(4,5)]       | 5        | 7          | Reduced           |
| [(1,1),(2,2),(3,5),(4,6)] | 6        | 8          | Reduced           |

## Complexity

| Problem  | Time (Worst) | Time (Typical)       | Space  |
|----------|--------------|----------------------|--------|
| Knapsack | $O(2^n)$     | Much less (pruning)  | $O(n)$ |
| TSP      | $O(n!)$      | Pruned significantly | $O(n)$ |

Branch and Bound is search with insight, it trims the impossible and focuses only where the optimum can hide.

## 97 Randomized Pattern

The Randomized Pattern introduces chance into algorithm design. Instead of following a fixed path, the algorithm makes random choices that, on average, lead to efficient performance or simplicity.

Randomization can help break symmetry, avoid worst-case traps, and simplify complex logic.

### What Problem Are We Solving?

We want algorithms that:

- Avoid pathological worst-case inputs.
- Simplify decisions that are hard deterministically.
- Achieve good expected performance.

Common examples:

- Randomized QuickSort: pivot chosen at random.
- Randomized Search / Sampling: estimate quantities via random trials.
- Monte Carlo and Las Vegas Algorithms: trade accuracy for speed or vice versa.

### How It Works (Plain Language)

Randomization can appear in two forms:

1. Las Vegas Algorithm
  - Always produces the correct result.
  - Runtime is random (e.g., Randomized QuickSort).
2. Monte Carlo Algorithm
  - Runs in fixed time.
  - May have a small probability of error (e.g., primality tests).

By picking random paths or samples, we smooth out bad cases and often simplify logic.

### Example: Randomized QuickSort

Choose a pivot randomly to avoid worst-case splits.

At each step:

1. Pick random pivot  $p$  from array.
2. Partition array into smaller ( $< p$ ) and larger ( $> p$ ).
3. Recursively sort halves.

Expected runtime is  $O(n \log n)$  even if input is adversarial.

### Tiny Code (Python)

```
import random

def randomized_quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = random.choice(arr)
    left = [x for x in arr if x < pivot]
    mid = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return randomized_quicksort(left) + mid + randomized_quicksort(right)
```

### Why It Matters

Randomized algorithms are:

- Simple: randomization replaces complex logic.
- Efficient: often faster in expectation.
- Robust: resistant to adversarial input.

They appear in:

- Sorting, searching, and hashing.
- Approximation algorithms.
- Cryptography and sampling.
- Machine learning (e.g., SGD, bagging).

## A Gentle Proof (Why It Works)

Let  $T(n)$  be expected time of Randomized QuickSort:

$$T(n) = n - 1 + \frac{2}{n} \sum_{k=0}^{n-1} T(k)$$

Solving yields  $T(n) = O(n \log n)$ . Random pivot ensures each element has equal probability to split array, making balanced partitions likely on average.

Expected cost avoids  $O(n^2)$  worst-case of fixed-pivot QuickSort.

## Try It Yourself

1. Implement Randomized QuickSort, run on sorted input.
2. Compare average time to standard QuickSort.
3. Try a random primality test (e.g., Miller–Rabin).
4. Use random sampling to approximate  $\pi$  via Monte Carlo.

## Test Cases

| Input       | Expected Result | Notes                           |
|-------------|-----------------|---------------------------------|
| [1,2,3,4,5] | [1,2,3,4,5]     | Random pivot avoids worst-case  |
| [5,4,3,2,1] | [1,2,3,4,5]     | Still fast due to random splits |

## Complexity

| Algorithm            | Expected Time   | Worst Time      | Space       |
|----------------------|-----------------|-----------------|-------------|
| Randomized QuickSort | $O(n \log n)$   | $O(n^2)$ (rare) | $O(\log n)$ |
| Randomized Search    | $O(1)$ expected | $O(n)$ worst    | $O(1)$      |

Randomization turns rigid logic into flexible, average-case excellence, a practical ally in uncertain or adversarial worlds.

## 98 Approximation Pattern

The Approximation Pattern is used when finding the exact solution is too expensive or impossible. Instead of striving for perfection, we design algorithms that produce results *close enough* to optimal, fast, predictable, and often guaranteed within a factor.

This pattern shines in NP-hard problems, where exact methods scale poorly.

### What Problem Are We Solving?

Some problems, like Traveling Salesman, Vertex Cover, or Knapsack, have no known polynomial-time exact solutions. We need algorithms that give good-enough answers quickly, especially for large inputs.

Approximation algorithms ensure:

- Predictable performance.
- Measurable accuracy.
- Polynomial runtime.

### How It Works (Plain Language)

An approximation algorithm outputs a solution within a known ratio of the optimal value:

If the optimal cost is  $\text{OPT}$ , and our algorithm returns  $\text{ALG}$ , then for a minimization problem:

$$\frac{\text{ALG}}{\text{OPT}} \leq \alpha$$

where  $\alpha$  is the approximation factor (e.g., 2, 1.5, or  $(1 + \epsilon)$ ).

### Example: Vertex Cover (2-Approximation)

Problem: find smallest set of vertices touching all edges.

Algorithm:

1. Start with an empty set  $C$ .
2. While edges remain:
  - Pick any uncovered edge  $(u, v)$ .
  - Add both  $u$  and  $v$  to  $C$ .
  - Remove all edges incident to  $u$  or  $v$ .

3. Return  $C$ .

This guarantees  $|C| \leq 2 \cdot |C^*|$ , where  $C^*$  is the optimal vertex cover.

### Tiny Code (Python)

```
def vertex_cover(edges):
    cover = set()
    while edges:
        (u, v) = edges.pop()
        cover.add(u)
        cover.add(v)
        edges = [(x, y) for (x, y) in edges if x not in (u, v) and y not in (u, v)]
    return cover
```

### Why It Matters

Approximation algorithms:

- Provide provable guarantees.
- Scale to large problems.
- Offer predictable trade-offs between time and accuracy.

Widely used in:

- Combinatorial optimization.
- Scheduling, routing, resource allocation.
- AI planning, clustering, and compression.

### A Gentle Proof (Why It Works)

Let  $C^*$  be optimal cover. Every edge must be covered by  $C^*$ . We select 2 vertices per edge, so:

$$|C| = 2 \cdot (\text{number of edges selected}) \leq 2 \cdot |C^*|$$

Thus, the approximation factor is 2.

## Try It Yourself

1. Implement the 2-approx Vertex Cover algorithm.
2. Compare result size with brute-force solution for small graphs.
3. Explore  $(1 + \epsilon)$ -approximation using greedy selection.
4. Apply same idea to Set Cover or Knapsack.

## Test Cases

| Graph    | Optimal | Algorithm | Ratio |
|----------|---------|-----------|-------|
| Triangle | 2       | 2         | 1.0   |
| Square   | 2       | 4         | 2.0   |

## Complexity

| Algorithm             | Time              | Space    | Guarantee        |
|-----------------------|-------------------|----------|------------------|
| Vertex Cover (Greedy) | $O(E)$            | $O(V)$   | 2-Approx         |
| Knapsack (FPTAS)      | $O(n^3/\epsilon)$ | $O(n^2)$ | $(1 + \epsilon)$ |

Approximation is the art of being *nearly perfect, swiftly*, a pragmatic bridge between theory and the real world.

## 99 Online Algorithm Pattern

The Online Algorithm Pattern is used when input arrives sequentially, and decisions must be made immediately without knowledge of future data. There's no rewinding or re-optimizing later, you commit as you go.

This pattern models real-time decision-making, from caching to task scheduling and resource allocation.

## What Problem Are We Solving?

In many systems, data doesn't come all at once. You must decide now, not after seeing the full picture.

Typical scenarios:

- Cache replacement (decide which page to evict next).
- Task assignment (jobs arrive in real time).
- Dynamic routing (packets arrive continuously).

Offline algorithms know everything upfront; online algorithms don't, yet must perform competitively.

### **How It Works (Plain Language)**

An online algorithm processes inputs one by one. Each step:

1. Receive input item  $x_t$  at time  $t$ .
2. Make a decision  $d_t$  using current state only.
3. Cannot change  $d_t$  later.

Performance is measured by the competitive ratio:

$$\text{Competitive Ratio} = \max_{\text{inputs}} \frac{\text{Cost} * \text{ALG}}{\text{Cost} * \text{OPT}}$$

If ALG's cost is at most  $k$  times optimal, the algorithm is  $k$ -competitive.

### **Example: Paging / Cache Replacement**

You have cache of size  $k$ . Sequence of page requests arrives. If requested page is not in cache → page fault → load it (evict one if full).

Algorithms:

- FIFO (First In First Out): Evict oldest.
- LRU (Least Recently Used): Evict least recently accessed.
- Random: Evict randomly.

LRU is  $k$ -competitive, meaning it performs within factor  $k$  of optimal.

### **Tiny Code (Python)**

```

def lru_cache(pages, capacity):
    cache = []
    faults = 0
    for p in pages:
        if p not in cache:
            faults += 1
            if len(cache) == capacity:
                cache.pop(0)
            cache.append(p)
        else:
            cache.remove(p)
            cache.append(p)
    return faults

```

## Why It Matters

Online algorithms:

- Reflect real-world constraints (no foresight).
- Enable adaptive systems in streaming, caching, and scheduling.
- Provide competitive guarantees even under worst-case input.

Used in:

- Operating systems (page replacement).
- Networking (packet routing).
- Finance (online pricing, bidding).
- Machine learning (online gradient descent).

## A Gentle Proof (Why It Works)

For LRU Cache: Every cache miss means a unique page not seen in last  $k$  requests. The optimal offline algorithm (OPT) can avoid some faults but at most  $k$  times fewer. Thus:

$$\text{Faults(LRU)} \leq k \cdot \text{Faults(OPT)}$$

So LRU is  $k$ -competitive.

## Try It Yourself

1. Simulate LRU, FIFO, Random cache on same request sequence.
2. Count page faults.
3. Compare with offline OPT (Belady's Algorithm).
4. Experiment with  $k = 2, 3, 4$ .

## Test Cases

| Pages             | Cache Size | Algorithm | Faults | Ratio (vs OPT) |
|-------------------|------------|-----------|--------|----------------|
| [1,2,3,1,2,3]     | 2          | LRU       | 6      | 3.0            |
| [1,2,3,4,1,2,3,4] | 3          | LRU       | 8      | 2.7            |

## Complexity

| Algorithm     | Time    | Space  | Competitive Ratio |
|---------------|---------|--------|-------------------|
| FIFO          | $O(nk)$ | $O(k)$ | $k$               |
| LRU           | $O(nk)$ | $O(k)$ | $k$               |
| OPT (offline) | $O(nk)$ | $O(k)$ | 1                 |

Online algorithms embrace uncertainty, they act wisely *now*, trusting analysis to prove they won't regret it later.

## 100 Hybrid Strategy Pattern

The Hybrid Strategy Pattern combines multiple algorithmic paradigms, such as divide and conquer, greedy, and dynamic programming, to balance their strengths and overcome individual weaknesses. Instead of sticking to one design philosophy, hybrid algorithms adapt to the structure of the problem and the size of the input.

## What Problem Are We Solving?

No single paradigm fits all problems. Some inputs are small and benefit from brute force; others require recursive structure; still others need heuristics.

We need a meta-strategy that blends paradigms, switching between them based on conditions like:

- Input size (e.g., small vs large)
- Structure (e.g., sorted vs unsorted)
- Precision requirements (e.g., exact vs approximate)

Hybrid strategies offer practical performance beyond theoretical asymptotics.

### How It Works (Plain Language)

A hybrid algorithm uses *decision logic* to pick the best method for each situation.

Common patterns:

1. Small-case base switch: Use brute force when  $n$  is small (e.g., Insertion Sort inside QuickSort).
2. Stage combination: Use one algorithm for setup, another for refinement (e.g., Greedy for initial solution, DP for optimization).
3. Conditional strategy: Choose algorithm based on data distribution (e.g., QuickSort vs HeapSort).

### Example: Introsort

Introsort starts like QuickSort for average speed, but if recursion depth grows too large (bad pivot splits), it switches to HeapSort to guarantee  $O(n \log n)$  worst-case time.

Steps:

1. Partition using QuickSort.
2. Track recursion depth.
3. If depth > threshold ( $2 \log n$ ), switch to HeapSort.

This ensures best of both worlds: average speed + worst-case safety.

### Tiny Code (Python)

```
def introsort(arr, depth_limit):
    if len(arr) <= 1:
        return arr
    if depth_limit == 0:
        return heapsort(arr)
    pivot = arr[len(arr)//2]
    left = [x for x in arr if x < pivot]
    mid = [x for x in arr if x == pivot]
```

```

right = [x for x in arr if x > pivot]
return introsort(left, depth_limit - 1) + mid + introsort(right, depth_limit - 1)

```

*(Uses heapsort when depth limit is reached)*

## Why It Matters

Hybrid strategies give real-world efficiency, predictable performance, and robust fallback behavior. They mirror how expert developers build systems, not one-size-fits-all, but layered and conditional.

Common hybrids:

- Timsort = MergeSort + InsertionSort
- Introsort = QuickSort + HeapSort
- Branch-and-Bound + Greedy = Search with pruning and heuristics
- Neural + Symbolic = Learning + Logical reasoning

## A Gentle Proof (Why It Works)

Let  $A_1, A_2, \dots, A_k$  be candidate algorithms with cost functions  $T_i(n)$ . Hybrid strategy  $H$  chooses  $A_i$  when condition  $C_i(n)$  holds.

If decision logic ensures

$$T_H(n) = \min_i T_i(n) \mid C_i(n)$$

then  $H$  performs at least as well as the best applicable algorithm.

Thus  $T_H(n) = O(\min_i T_i(n))$ .

## Try It Yourself

1. Implement QuickSort + InsertionSort hybrid.
2. Set threshold  $n_0 = 10$  for switching.
3. Compare performance vs pure QuickSort.
4. Experiment with different thresholds.

## Test Cases

| Input Size | Algorithm      | Time    | Notes                |
|------------|----------------|---------|----------------------|
| 10         | Insertion Sort | Fastest | Simplicity wins      |
| 1000       | QuickSort      | Optimal | Low overhead         |
| 1e6        | Introsort      | Stable  | No worst-case blowup |

## Complexity

| Component | Best          | Average       | Worst         | Space       |
|-----------|---------------|---------------|---------------|-------------|
| QuickSort | $O(n \log n)$ | $O(n \log n)$ | $O(n^2)$      | $O(\log n)$ |
| HeapSort  | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | $O(1)$      |
| Introsort | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | $O(\log n)$ |

A hybrid strategy is not just an algorithmic trick, it's a mindset: combine precision, adaptability, and pragmatism to build algorithms that thrive in the wild.

# Chapter 2. Sorting and searching

## Section 11. Elementary sorting

### 101 Bubble Sort

Bubble Sort is like washing dishes one by one, you keep moving the biggest plate to the bottom until everything is clean and sorted. It is simple, visual, and perfect for building your sorting intuition before diving into more advanced methods.

#### What Problem Are We Solving?

We want to arrange a list of elements in order (ascending or descending) by repeatedly comparing and swapping adjacent items that are out of order.

Formally: Given an array  $A[0..n-1]$ , repeat passes until no swaps occur. Each pass bubbles up the largest remaining element to its final position.

#### Example

| Step | Array State  | Description             |
|------|--------------|-------------------------|
| 0    | [5, 3, 4, 1] | Initial array           |
| 1    | [3, 4, 1, 5] | 5 bubbled to the end    |
| 2    | [3, 1, 4, 5] | 4 bubbled to position 3 |
| 3    | [1, 3, 4, 5] | Array fully sorted      |

#### How Does It Work (Plain Language)?

Imagine bubbles rising to the surface, the biggest one reaches the top first. In Bubble Sort, each sweep through the list compares neighboring pairs, swapping them if they are in the wrong order. After each full pass, one more element settles into place.

We repeat until a pass finishes with no swaps, meaning the array is sorted.

## Step-by-Step Process

| Step | Action                     | Resulting Array      |
|------|----------------------------|----------------------|
| 1    | Compare A[0] and A[1]      | Swap if needed       |
| 2    | Compare A[1] and A[2]      | Swap if needed       |
| 3    | Continue through A[n-2]    | Repeat comparisons   |
| 4    | Repeat passes until sorted | Early stop if sorted |

## Tiny Code (Easy Versions)

C

```
#include <stdio.h>
#include <stdbool.h>

void bubble_sort(int a[], int n) {
    bool swapped;
    for (int pass = 0; pass < n - 1; pass++) {
        swapped = false;
        for (int i = 0; i < n - pass - 1; i++) {
            if (a[i] > a[i + 1]) {
                int temp = a[i];
                a[i] = a[i + 1];
                a[i + 1] = temp;
                swapped = true;
            }
        }
        if (!swapped) break; // early exit if already sorted
    }
}

int main(void) {
    int a[] = {5, 3, 4, 1, 2};
    int n = sizeof(a) / sizeof(a[0]);
    bubble_sort(a, n);
    for (int i = 0; i < n; i++) printf("%d ", a[i]);
    printf("\n");
}
```

## Python

```
def bubble_sort(a):
    n = len(a)
    for pass_num in range(n - 1):
        swapped = False
        for i in range(n - pass_num - 1):
            if a[i] > a[i + 1]:
                a[i], a[i + 1] = a[i + 1], a[i]
                swapped = True
        if not swapped:
            break

arr = [5, 3, 4, 1, 2]
bubble_sort(arr)
print(arr)
```

## Why It Matters

- Teaches comparison-based sorting through intuition
- Builds understanding of adjacent swaps and pass-based progress
- Introduces stability (equal elements keep their relative order)
- Sets the stage for improved versions (Improved Bubble Sort, Cocktail Shaker Sort, Comb Sort)

## A Gentle Proof (Why It Works)

After the first pass, the largest element moves to the last position. After the second pass, the second largest is in position  $n-2$ .

So after  $k$  passes, the last  $k$  elements are sorted.

If we track comparisons: 1st pass:  $(n-1)$  comparisons 2nd pass:  $(n-2)$  comparisons ...  $(n-1)$ th pass: 1 comparison

| Pass  | Comparisons | Elements Sorted at End |
|-------|-------------|------------------------|
| 1     | $n-1$       | Largest element        |
| 2     | $n-2$       | Next largest           |
| ...   | ...         | ...                    |
| $n-1$ | 1           | Fully sorted array     |

Total comparisons =  $(n-1) + (n-2) + \dots + 1 = n(n-1)/2$

So time =  $O(n^2)$  in the worst case. If already sorted, early exit makes it  $O(n)$ .

### Try It Yourself

| Task | Description                                  |
|------|----------------------------------------------|
| 1    | Sort [3, 2, 1] step by step                  |
| 2    | Count how many swaps occur                   |
| 3    | Add a flag to detect early termination       |
| 4    | Compare with Insertion Sort on the same data |
| 5    | Modify to sort descending                    |

### Test Cases

| Input           | Output          | Passes | Swaps |
|-----------------|-----------------|--------|-------|
| [3, 2, 1]       | [1, 2, 3]       | 3      | 3     |
| [1, 2, 3]       | [1, 2, 3]       | 1      | 0     |
| [5, 1, 4, 2, 8] | [1, 2, 4, 5, 8] | 4      | 5     |

### Complexity

| Aspect       | Value                       |
|--------------|-----------------------------|
| Time (Worst) | $O(n^2)$                    |
| Time (Best)  | $O(n)$                      |
| Space        | $O(1)$ (in-place)           |
| Stable       | Yes                         |
| Adaptive     | Yes (stops early if sorted) |

Bubble Sort is your first step into the sorting world, simple enough to code by hand, visual enough to animate, and powerful enough to spark intuition for more advanced sorts.

## 102 Improved Bubble Sort

Improved Bubble Sort builds on the basic version by recognizing that once part of the array is sorted, there's no need to revisit it. It introduces small optimizations like early termination and tracking the last swap position to reduce unnecessary comparisons.

## What Problem Are We Solving??

Basic Bubble Sort keeps scanning the whole array every pass, even when the tail is already sorted. Improved Bubble Sort fixes this by remembering where the last swap happened. Elements beyond that index are already in place, so the next pass can stop earlier.

This optimization is especially effective for arrays that are nearly sorted.

## Example

| Step | Array State     | Last Swap Index | Range Checked |
|------|-----------------|-----------------|---------------|
| 0    | [5, 3, 4, 1, 2] | -               | 0 to 4        |
| 1    | [3, 4, 1, 2, 5] | 3               | 0 to 3        |
| 2    | [3, 1, 2, 4, 5] | 2               | 0 to 2        |
| 3    | [1, 2, 3, 4, 5] | 1               | 0 to 1        |
| 4    | [1, 2, 3, 4, 5] | 0               | Stop early    |

## How Does It Work (Plain Language)?

We improve efficiency by narrowing each pass to only the unsorted part. We also stop early when no swaps occur, signaling the array is already sorted.

Step by step:

1. Track index of the last swap in each pass
2. Next pass ends at that index
3. Stop when no swaps occur (fully sorted)

This reduces unnecessary comparisons in nearly sorted arrays.

## Tiny Code (Easy Versions)

C

```
#include <stdio.h>

void improved_bubble_sort(int a[], int n) {
    int new_n;
    while (n > 1) {
        new_n = 0;
        for (int i = 1; i < n; i++) {
```

```

        if (a[i - 1] > a[i]) {
            int temp = a[i - 1];
            a[i - 1] = a[i];
            a[i] = temp;
            new_n = i;
        }
    }
    n = new_n;
}
}

int main(void) {
    int a[] = {5, 3, 4, 1, 2};
    int n = sizeof(a) / sizeof(a[0]);
    improved_bubble_sort(a, n);
    for (int i = 0; i < n; i++) printf("%d ", a[i]);
    printf("\n");
}

```

## Python

```

def improved_bubble_sort(a):
    n = len(a)
    while n > 1:
        new_n = 0
        for i in range(1, n):
            if a[i - 1] > a[i]:
                a[i - 1], a[i] = a[i], a[i - 1]
                new_n = i
        n = new_n

arr = [5, 3, 4, 1, 2]
improved_bubble_sort(arr)
print(arr)

```

## Why It Matters

- Reduces redundant comparisons
- Automatically adapts to partially sorted data
- Stops as soon as the array is sorted
- Retains stability and simplicity

## A Gentle Proof (Why It Works)

If the last swap occurs at index  $k$ , all elements after  $k$  are already in order. Next pass only needs to scan up to  $k$ . If no swaps occur ( $k = 0$ ), the array is sorted.

| Pass | Comparisons | Last Swap | Range  | Next Pass |
|------|-------------|-----------|--------|-----------|
| 1    | $n-1$       | $k$       | $0..k$ |           |
| 2    | $k-1$       | $k$       | $0..k$ |           |
| ...  | ...         | ...       | ...    | ...       |

In the best case (already sorted), only one pass occurs:  $O(n)$  Worst case remains  $O(n^2)$

## Try It Yourself

| Task | Description                                         |
|------|-----------------------------------------------------|
| 1    | Sort $[1, 2, 3, 4, 5]$ and observe early stop       |
| 2    | Sort $[5, 4, 3, 2, 1]$ and track last swap index    |
| 3    | Modify to print last swap index each pass           |
| 4    | Compare with standard Bubble Sort pass count        |
| 5    | Try arrays with repeated values to verify stability |

## Test Cases

| Input             | Output            | Passes | Improvement            |
|-------------------|-------------------|--------|------------------------|
| $[1, 2, 3, 4, 5]$ | $[1, 2, 3, 4, 5]$ | 1      | Early stop             |
| $[5, 3, 4, 1, 2]$ | $[1, 2, 3, 4, 5]$ | 4      | Fewer checks each pass |
| $[2, 1, 3, 4, 5]$ | $[1, 2, 3, 4, 5]$ | 1      | Detects sorted tail    |

## Complexity

| Aspect       | Value    |
|--------------|----------|
| Time (Worst) | $O(n^2)$ |
| Time (Best)  | $O(n)$   |
| Space        | $O(1)$   |
| Stable       | Yes      |

| Aspect   | Value |
|----------|-------|
| Adaptive | Yes   |

Improved Bubble Sort shows how a small observation can make a classic algorithm smarter. By tracking the last swap, it skips already-sorted tails and gives a glimpse of how adaptive sorting works in practice.

## 103 Cocktail Shaker Sort

Cocktail Shaker Sort, also known as Bidirectional Bubble Sort, improves on Bubble Sort by sorting in both directions during each pass. It moves the largest element to the end and the smallest to the beginning, reducing the number of passes required.

### What Problem Are We Solving??

Standard Bubble Sort only bubbles up in one direction, pushing the largest element to the end each pass. If small elements start near the end, they take many passes to reach their position.

Cocktail Shaker Sort fixes this by sweeping back and forth, bubbling both ends at once.

### Example

| Step | Direction    | Array State     | Description             |
|------|--------------|-----------------|-------------------------|
| 0    | –            | [5, 3, 4, 1, 2] | Initial array           |
| 1    | Left → Right | [3, 4, 1, 2, 5] | 5 bubbled to end        |
| 2    | Right → Left | [1, 3, 4, 2, 5] | 1 bubbled to start      |
| 3    | Left → Right | [1, 3, 2, 4, 5] | 4 bubbled to position 4 |
| 4    | Right → Left | [1, 2, 3, 4, 5] | 2 bubbled to position 2 |

Sorted after 4 directional passes.

### How Does It Work (Plain Language)?

Cocktail Shaker Sort is like stirring from both sides of the array. Each forward pass moves the largest unsorted element to the end. Each backward pass moves the smallest unsorted element to the start.

The unsorted region shrinks from both ends with each full cycle.

## Step-by-Step Process

| Step | Action                                        | Result |
|------|-----------------------------------------------|--------|
| 1    | Sweep left to right, bubble largest to end    |        |
| 2    | Sweep right to left, bubble smallest to start |        |
| 3    | Narrow bounds, repeat until sorted            |        |

## Tiny Code (Easy Versions)

C

```
#include <stdio.h>
#include <stdbool.h>

void cocktail_shaker_sort(int a[], int n) {
    bool swapped = true;
    int start = 0, end = n - 1;

    while (swapped) {
        swapped = false;

        // Forward pass
        for (int i = start; i < end; i++) {
            if (a[i] > a[i + 1]) {
                int temp = a[i];
                a[i] = a[i + 1];
                a[i + 1] = temp;
                swapped = true;
            }
        }
        if (!swapped) break;
        swapped = false;
        end--;

        // Backward pass
        for (int i = end - 1; i >= start; i--) {
            if (a[i] > a[i + 1]) {
                int temp = a[i];
                a[i] = a[i + 1];
                a[i + 1] = temp;
            }
        }
    }
}
```

```

        swapped = true;
    }
}
start++;
}
}

int main(void) {
    int a[] = {5, 3, 4, 1, 2};
    int n = sizeof(a) / sizeof(a[0]);
    cocktail_shaker_sort(a, n);
    for (int i = 0; i < n; i++) printf("%d ", a[i]);
    printf("\n");
}

```

## Python

```

def cocktail_shaker_sort(a):
    n = len(a)
    start, end = 0, n - 1
    swapped = True

    while swapped:
        swapped = False
        for i in range(start, end):
            if a[i] > a[i + 1]:
                a[i], a[i + 1] = a[i + 1], a[i]
                swapped = True
        if not swapped:
            break
        swapped = False
        end -= 1
        for i in range(end - 1, start - 1, -1):
            if a[i] > a[i + 1]:
                a[i], a[i + 1] = a[i + 1], a[i]
                swapped = True
        start += 1

arr = [5, 3, 4, 1, 2]
cocktail_shaker_sort(arr)
print(arr)

```

## Why It Matters

- Sorts in both directions, reducing unnecessary passes
- Performs better than Bubble Sort on many practical inputs
- Stable and easy to visualize
- Demonstrates bidirectional improvement, a foundation for adaptive sorting

## A Gentle Proof (Why It Works)

Each forward pass moves the maximum element of the unsorted range to the end. Each backward pass moves the minimum element of the unsorted range to the start. Thus, the unsorted range shrinks from both sides, guaranteeing progress each cycle.

| Cycle | Forward Pass   | Backward Pass     | Sorted Range     |
|-------|----------------|-------------------|------------------|
| 1     | Largest to end | Smallest to start | [0], [n-1]       |
| 2     | Next largest   | Next smallest     | [0,1], [n-2,n-1] |
| ...   | ...            | ...               | ...              |

Worst case still  $O(n^2)$ , best case  $O(n)$  if already sorted.

## Try It Yourself

| Task | Description                                            |
|------|--------------------------------------------------------|
| 1    | Sort [5, 3, 4, 1, 2] and track forward/backward passes |
| 2    | Visualize the shrinking unsorted range                 |
| 3    | Compare with standard Bubble Sort on reverse array     |
| 4    | Modify code to print array after each pass             |
| 5    | Test stability with duplicate values                   |

## Test Cases

| Input           | Output          | Passes | Notes                         |
|-----------------|-----------------|--------|-------------------------------|
| [5, 3, 4, 1, 2] | [1, 2, 3, 4, 5] | 4      | Fewer passes than bubble sort |
| [1, 2, 3, 4, 5] | [1, 2, 3, 4, 5] | 1      | Early termination             |
| [2, 1, 3, 5, 4] | [1, 2, 3, 4, 5] | 2      | Moves smallest quickly        |

## Complexity

| Aspect       | Value    |
|--------------|----------|
| Time (Worst) | $O(n^2)$ |
| Time (Best)  | $O(n)$   |
| Space        | $O(1)$   |
| Stable       | Yes      |
| Adaptive     | Yes      |

Cocktail Shaker Sort takes the simplicity of Bubble Sort and doubles its efficiency for certain inputs. By sorting in both directions, it highlights the power of symmetry and small algorithmic tweaks.

## 104 Selection Sort

Selection Sort is like organizing a deck of cards by repeatedly picking the smallest card and placing it in order. It is simple, predictable, and useful for understanding how selection-based sorting works.

### What Problem Are We Solving??

We want to sort an array by repeatedly selecting the smallest (or largest) element from the unsorted part and swapping it into the correct position.

Selection Sort separates the array into two parts:

- A sorted prefix (built one element at a time)
- An unsorted suffix (from which we select the next minimum)

### Example

| Step | Array State     | Action             | Sorted Part | Unsorted Part |
|------|-----------------|--------------------|-------------|---------------|
| 0    | [5, 3, 4, 1, 2] | Start              | []          | [5,3,4,1,2]   |
| 1    | [1, 3, 4, 5, 2] | Place 1 at index 0 | [1]         | [3,4,5,2]     |
| 2    | [1, 2, 4, 5, 3] | Place 2 at index 1 | [1,2]       | [4,5,3]       |
| 3    | [1, 2, 3, 5, 4] | Place 3 at index 2 | [1,2,3]     | [5,4]         |
| 4    | [1, 2, 3, 4, 5] | Place 4 at index 3 | [1,2,3,4]   | [5]           |
| 5    | [1, 2, 3, 4, 5] | Done               | [1,2,3,4,5] | []            |

## How Does It Work (Plain Language)?

Selection Sort looks through the unsorted portion, finds the smallest element, and moves it to the front. It does not care about intermediate order until each selection is done.

Each pass fixes one position permanently.

## Step-by-Step Process

| Step | Action                         | Effect             |
|------|--------------------------------|--------------------|
| 1    | Find smallest in unsorted part | Move it to front   |
| 2    | Repeat for next unsorted index | Grow sorted prefix |
| 3    | Stop when entire array sorted  |                    |

## Tiny Code (Easy Versions)

C

```
#include <stdio.h>

void selection_sort(int a[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int min_idx = i;
        for (int j = i + 1; j < n; j++) {
            if (a[j] < a[min_idx]) {
                min_idx = j;
            }
        }
        int temp = a[i];
        a[i] = a[min_idx];
        a[min_idx] = temp;
    }
}

int main(void) {
    int a[] = {5, 3, 4, 1, 2};
    int n = sizeof(a) / sizeof(a[0]);
    selection_sort(a, n);
    for (int i = 0; i < n; i++) printf("%d ", a[i]);
    printf("\n");
}
```

## Python

```
def selection_sort(a):
    n = len(a)
    for i in range(n - 1):
        min_idx = i
        for j in range(i + 1, n):
            if a[j] < a[min_idx]:
                min_idx = j
        a[i], a[min_idx] = a[min_idx], a[i]

arr = [5, 3, 4, 1, 2]
selection_sort(arr)
print(arr)
```

## Why It Matters

- Simple, deterministic sorting algorithm
- Demonstrates selection rather than swapping neighbors
- Good for small lists and teaching purposes
- Useful when minimizing number of swaps matters

## A Gentle Proof (Why It Works)

At each iteration, the smallest remaining element is placed at its correct position. Once placed, it never moves again.

The algorithm performs  $n-1$  selections and at most  $n-1$  swaps. Each selection requires scanning the unsorted part:  $O(n)$  comparisons.

| Pass  | Search Range   | Comparisons | Swap |
|-------|----------------|-------------|------|
| 1     | $n$ elements   | $n-1$       | 1    |
| 2     | $n-1$ elements | $n-2$       | 1    |
| ...   | ...            | ...         | ...  |
| $n-1$ | 2 elements     | 1           | 1    |

$$\text{Total comparisons} = n(n-1)/2 = O(n^2)$$

## Try It Yourself

| Task | Description                                         |
|------|-----------------------------------------------------|
| 1    | Trace sorting of [5, 3, 4, 1, 2] step by step       |
| 2    | Count total swaps and comparisons                   |
| 3    | Modify to find maximum each pass (descending order) |
| 4    | Add print statements to see progress                |
| 5    | Compare with Bubble Sort efficiency                 |

## Test Cases

| Input           | Output          | Passes | Swaps |
|-----------------|-----------------|--------|-------|
| [3, 2, 1]       | [1, 2, 3]       | 2      | 2     |
| [1, 2, 3]       | [1, 2, 3]       | 2      | 0     |
| [5, 3, 4, 1, 2] | [1, 2, 3, 4, 5] | 4      | 4     |

## Complexity

| Aspect       | Value                     |
|--------------|---------------------------|
| Time (Worst) | $O(n^2)$                  |
| Time (Best)  | $O(n^2)$                  |
| Space        | $O(1)$                    |
| Stable       | No (swap may break order) |
| Adaptive     | No                        |

Selection Sort is a calm, methodical sorter. It does not adapt, but it does not waste swaps either. It is the simplest demonstration of the idea: find the smallest, place it, repeat.

## 105 Double Selection Sort

Double Selection Sort is a refined version of Selection Sort. Instead of finding just the smallest element each pass, it finds both the smallest and the largest, placing them at the beginning and end simultaneously. This halves the number of passes needed.

## What Problem Are We Solving??

Standard Selection Sort finds one element per pass. Double Selection Sort improves efficiency by selecting two elements per pass, one from each end, reducing total iterations by about half.

It is useful when both extremes can be found in a single scan, improving constant factors while keeping overall simplicity.

## Example

| Step | Array State     | Min | Max | Action                        | Sorted Part    |
|------|-----------------|-----|-----|-------------------------------|----------------|
| 0    | [5, 3, 4, 1, 2] | 1   | 5   | Swap 1 → front, 5 → back      | [1, ..., 5]    |
| 1    | [1, 3, 4, 2, 5] | 2   | 4   | Swap 2 → index 1, 4 → index 3 | [1,2, ...,4,5] |
| 2    | [1, 2, 3, 4, 5] | 3   | 3   | Middle element sorted         | [1,2,3,4,5]    |

Sorted in 3 passes instead of 5.

## How Does It Work (Plain Language)?

Double Selection Sort narrows the unsorted range from both sides. Each pass:

1. Scans the unsorted section once.
2. Finds both the smallest and largest elements.
3. Swaps them to their correct positions at the front and back.

Then it shrinks the bounds and repeats.

## Step-by-Step Process

| Step | Action                                | Effect                            |
|------|---------------------------------------|-----------------------------------|
| 1    | Find smallest and largest in unsorted | Move smallest left, largest right |
| 2    | Shrink unsorted range                 | Repeat search                     |
| 3    | Stop when range collapses             | Array sorted                      |

## Tiny Code (Easy Versions)

C

```
#include <stdio.h>

void double_selection_sort(int a[], int n) {
    int left = 0, right = n - 1;

    while (left < right) {
        int min_idx = left, max_idx = left;

        for (int i = left; i <= right; i++) {
            if (a[i] < a[min_idx]) min_idx = i;
            if (a[i] > a[max_idx]) max_idx = i;
        }

        // Move smallest to front
        int temp = a[left];
        a[left] = a[min_idx];
        a[min_idx] = temp;

        // If max element was swapped into min_idx
        if (max_idx == left) max_idx = min_idx;

        // Move largest to back
        temp = a[right];
        a[right] = a[max_idx];
        a[max_idx] = temp;

        left++;
        right--;
    }
}

int main(void) {
    int a[] = {5, 3, 4, 1, 2};
    int n = sizeof(a) / sizeof(a[0]);
    double_selection_sort(a, n);
    for (int i = 0; i < n; i++) printf("%d ", a[i]);
    printf("\n");
}
```

## Python

```
def double_selection_sort(a):
    left, right = 0, len(a) - 1
    while left < right:
        min_idx, max_idx = left, left
        for i in range(left, right + 1):
            if a[i] < a[min_idx]:
                min_idx = i
            if a[i] > a[max_idx]:
                max_idx = i

        a[left], a[min_idx] = a[min_idx], a[left]
        if max_idx == left:
            max_idx = min_idx
        a[right], a[max_idx] = a[max_idx], a[right]

        left += 1
        right -= 1

arr = [5, 3, 4, 1, 2]
double_selection_sort(arr)
print(arr)
```

## Why It Matters

- Improves Selection Sort by reducing passes
- Selects two extremes in one scan
- Fewer total swaps and comparisons
- Demonstrates bidirectional selection

## A Gentle Proof (Why It Works)

Each pass moves two elements to their correct final positions. Thus, after  $k$  passes, the first  $k$  and last  $k$  positions are sorted. The unsorted range shrinks by 2 each pass.

| Pass | Range Checked | Elements Fixed | Remaining Unsorted |
|------|---------------|----------------|--------------------|
| 1    | [0..n-1]      | 2              | n-2                |
| 2    | [1..n-2]      | 2              | n-4                |
| ...  | ...           | ...            | ...                |

Total passes =  $n/2$ , each  $O(n)$  scan    $O(n^2)$  overall.

### Try It Yourself

| Task | Description                      |
|------|----------------------------------|
| 1    | Sort [5, 3, 4, 1, 2] manually    |
| 2    | Count passes and swaps           |
| 3    | Print range boundaries each pass |
| 4    | Compare to Selection Sort passes |
| 5    | Modify for descending order      |

### Test Cases

| Input           | Output          | Passes | Swaps |
|-----------------|-----------------|--------|-------|
| [3, 2, 1]       | [1, 2, 3]       | 2      | 2     |
| [1, 2, 3, 4, 5] | [1, 2, 3, 4, 5] | 2      | 0     |
| [5, 3, 4, 1, 2] | [1, 2, 3, 4, 5] | 3      | 6     |

### Complexity

| Aspect       | Value    |
|--------------|----------|
| Time (Worst) | $O(n^2)$ |
| Time (Best)  | $O(n^2)$ |
| Space        | $O(1)$   |
| Stable       | No       |
| Adaptive     | No       |

Double Selection Sort keeps Selection Sort's simplicity but doubles its reach. By grabbing both ends each pass, it highlights how symmetry can bring efficiency without new data structures.

## 106 Insertion Sort

Insertion Sort builds the sorted array one element at a time, like sorting playing cards in your hand. It takes each new element and inserts it into the correct position among those already sorted.

## What Problem Are We Solving??

We want a simple, stable way to sort elements by inserting each into place within the growing sorted section. This works especially well for small arrays or nearly sorted data.

Insertion Sort splits the array into two parts:

- Sorted prefix: elements that are already in order
- Unsorted suffix: remaining elements yet to be inserted

## Example

| Step | Array State     | Element Inserted | Action                          | Sorted Prefix   |
|------|-----------------|------------------|---------------------------------|-----------------|
| 0    | [5, 3, 4, 1, 2] | -                | Start with first element sorted | [5]             |
| 1    | [3, 5, 4, 1, 2] | 3                | Insert 3 before 5               | [3, 5]          |
| 2    | [3, 4, 5, 1, 2] | 4                | Insert 4 before 5               | [3, 4, 5]       |
| 3    | [1, 3, 4, 5, 2] | 1                | Insert 1 at front               | [1, 3, 4, 5]    |
| 4    | [1, 2, 3, 4, 5] | 2                | Insert 2 after 1                | [1, 2, 3, 4, 5] |

## How Does It Work (Plain Language)?

Imagine picking cards one by one and placing each into the correct spot among those already held. Insertion Sort repeats this logic for arrays:

1. Start with the first element (already sorted)
2. Take the next element
3. Compare backward through the sorted section
4. Shift elements to make space and insert it

## Step-by-Step Process

| Step | Action                                    | Result |
|------|-------------------------------------------|--------|
| 1    | Take next unsorted element                |        |
| 2    | Move through sorted part to find position |        |
| 3    | Shift larger elements right               |        |
| 4    | Insert element in correct position        |        |

## Tiny Code (Easy Versions)

### C

```
#include <stdio.h>

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

int main(void) {
    int a[] = {5, 3, 4, 1, 2};
    int n = sizeof(a) / sizeof(a[0]);
    insertion_sort(a, n);
    for (int i = 0; i < n; i++) printf("%d ", a[i]);
    printf("\n");
}
```

### Python

```
def insertion_sort(a):
    for i in range(1, len(a)):
        key = a[i]
        j = i - 1
        while j >= 0 and a[j] > key:
            a[j + 1] = a[j]
            j -= 1
        a[j + 1] = key

arr = [5, 3, 4, 1, 2]
insertion_sort(arr)
print(arr)
```

## Why It Matters

- Simple, intuitive, and stable
- Works well for small or nearly sorted arrays
- Commonly used as a subroutine in advanced algorithms (like Timsort)
- Demonstrates concept of incremental insertion

## A Gentle Proof (Why It Works)

At step  $i$ , the first  $i$  elements are sorted. Inserting element  $a[i]$  keeps the prefix sorted. Each insertion shifts elements greater than  $\text{key}$  to the right, ensuring correct position.

| Pass | Sorted Portion | Comparisons (Worst) | Shifts (Worst) |
|------|----------------|---------------------|----------------|
| 1    | [a ,a ]        | 1                   | 1              |
| 2    | [a ,a ,a ]     | 2                   | 2              |
| ...  | ...            | ...                 | ...            |
| n-1  | [a ...a ]      | n-1                 | n-1            |

Total  $(n^2)/2$  operations in the worst case.

If already sorted, only one comparison per element  $\rightarrow O(n)$ .

## Try It Yourself

| Task | Description                                 |
|------|---------------------------------------------|
| 1    | Sort [5, 3, 4, 1, 2] step by step           |
| 2    | Count shifts and comparisons                |
| 3    | Modify to sort descending                   |
| 4    | Compare runtime with Bubble Sort            |
| 5    | Insert print statements to trace insertions |

## Test Cases

| Input           | Output          | Passes | Swaps/Shifts |
|-----------------|-----------------|--------|--------------|
| [3, 2, 1]       | [1, 2, 3]       | 2      | 3            |
| [1, 2, 3]       | [1, 2, 3]       | 2      | 0            |
| [5, 3, 4, 1, 2] | [1, 2, 3, 4, 5] | 4      | 8            |

## Complexity

| Aspect       | Value    |
|--------------|----------|
| Time (Worst) | $O(n^2)$ |
| Time (Best)  | $O(n)$   |
| Space        | $O(1)$   |
| Stable       | Yes      |
| Adaptive     | Yes      |

Insertion Sort captures the logic of careful, incremental organization. It is slow for large random lists, but elegant, stable, and highly efficient when the data is already close to sorted.

## 107 Binary Insertion Sort

Binary Insertion Sort improves on traditional Insertion Sort by using binary search to find the correct insertion point instead of linear scanning. This reduces the number of comparisons from linear to logarithmic per insertion, while keeping the same stable, adaptive behavior.

### What Problem Are We Solving??

Standard Insertion Sort searches linearly through the sorted part to find where to insert the new element. If the sorted prefix is long, this costs  $O(n)$  comparisons per element.

Binary Insertion Sort replaces that with binary search, which finds the position in  $O(\log n)$  time, while still performing  $O(n)$  shifts.

This makes it a good choice when comparisons are expensive but shifting is cheap.

### Example

| Step | Sorted Portion | Element to Insert | Insertion Index (Binary Search) | Resulting Array |
|------|----------------|-------------------|---------------------------------|-----------------|
| 0    | [5]            | 3                 | 0                               | [3, 5, 4, 1, 2] |
| 1    | [3, 5]         | 4                 | 1                               | [3, 4, 5, 1, 2] |
| 2    | [3, 4, 5]      | 1                 | 0                               | [1, 3, 4, 5, 2] |
| 3    | [1, 3, 4, 5]   | 2                 | 1                               | [1, 2, 3, 4, 5] |

## How Does It Work (Plain Language)?

Just like Insertion Sort, we build a sorted prefix one element at a time. But instead of scanning backwards linearly, we use binary search to locate the correct position to insert the next element.

We still need to shift larger elements to the right, but we now know exactly where to stop.

## Step-by-Step Process

| Step | Action                                 | Effect               |
|------|----------------------------------------|----------------------|
| 1    | Perform binary search in sorted prefix | Find insertion point |
| 2    | Shift larger elements right            | Create space         |
| 3    | Insert element at found index          | Maintain order       |
| 4    | Repeat until sorted                    | Fully sorted array   |

## Tiny Code (Easy Versions)

C

```
#include <stdio.h>

int binary_search(int a[], int item, int low, int high) {
    while (low <= high) {
        int mid = (low + high) / 2;
        if (item == a[mid]) return mid + 1;
        else if (item > a[mid]) low = mid + 1;
        else high = mid - 1;
    }
    return low;
}

void binary_insertion_sort(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int key = a[i];
        int j = i - 1;
        int pos = binary_search(a, key, 0, j);

        while (j >= pos) {
            a[j + 1] = a[j];
            j--;
        }
        a[pos] = key;
    }
}
```

```

        j--;
    }
    a[pos] = key;
}
}

int main(void) {
    int a[] = {5, 3, 4, 1, 2};
    int n = sizeof(a) / sizeof(a[0]);
    binary_insertion_sort(a, n);
    for (int i = 0; i < n; i++) printf("%d ", a[i]);
    printf("\n");
}

```

## Python

```

def binary_search(a, item, low, high):
    while low <= high:
        mid = (low + high) // 2
        if item == a[mid]:
            return mid + 1
        elif item > a[mid]:
            low = mid + 1
        else:
            high = mid - 1
    return low

def binary_insertion_sort(a):
    for i in range(1, len(a)):
        key = a[i]
        pos = binary_search(a, key, 0, i - 1)
        a[pos + 1 : i + 1] = a[pos : i]
        a[pos] = key

arr = [5, 3, 4, 1, 2]
binary_insertion_sort(arr)
print(arr)

```

## Why It Matters

- Fewer comparisons than standard Insertion Sort

- Retains stability and adaptiveness
- Great when comparisons dominate runtime (e.g., complex objects)
- Demonstrates combining search and insertion ideas

### A Gentle Proof (Why It Works)

Binary search always finds the correct index in  $O(\log i)$  comparisons for the  $i$ -th element. Shifting elements still takes  $O(i)$  time. So total cost:

$$T(n) = \sum_{i=1}^{n-1} (\log i + i) = O(n^2)$$

but with fewer comparisons than standard Insertion Sort.

| Step | Comparisons  | Shifts | Total Cost |
|------|--------------|--------|------------|
| 1    | $\log 1 = 0$ | 1      | 1          |
| 2    | $\log 2 = 1$ | 2      | 3          |
| 3    | $\log 3 = 2$ | 3      | 5          |
| ...  | ...          | ...    | ...        |

### Try It Yourself

| Task | Description                                  |
|------|----------------------------------------------|
| 1    | Sort [5, 3, 4, 1, 2] step by step            |
| 2    | Print insertion index each pass              |
| 3    | Compare comparisons vs normal Insertion Sort |
| 4    | Modify to sort descending                    |
| 5    | Try with already sorted list                 |

### Test Cases

| Input           | Output          | Comparisons | Shifts |
|-----------------|-----------------|-------------|--------|
| [3, 2, 1]       | [1, 2, 3]       | ~3          | 3      |
| [1, 2, 3]       | [1, 2, 3]       | ~2          | 0      |
| [5, 3, 4, 1, 2] | [1, 2, 3, 4, 5] | ~7          | 8      |

## Complexity

| Aspect       | Value                                    |
|--------------|------------------------------------------|
| Time (Worst) | $O(n^2)$                                 |
| Time (Best)  | $O(n \log n)$ comparisons, $O(n)$ shifts |
| Space        | $O(1)$                                   |
| Stable       | Yes                                      |
| Adaptive     | Yes                                      |

Binary Insertion Sort is a thoughtful balance, smarter searches, same simple structure. It reminds us that even small changes (like using binary search) can bring real efficiency when precision matters.

## 108 Gnome Sort

Gnome Sort is a simple sorting algorithm that works by swapping adjacent elements, similar to Bubble Sort, but with a twist, it moves backward whenever a swap is made. Imagine a gnome tidying flower pots: each time it finds two out of order, it swaps them and steps back to recheck the previous pair.

### What Problem Are We Solving??

We want a simple, intuitive, and in-place sorting method that uses local swaps to restore order. Gnome Sort is particularly easy to implement and works like an insertion sort with adjacent swaps instead of shifting elements.

It's not the fastest, but it's charmingly simple, perfect for understanding local correction logic.

### Example

| Step | Position | Array State     | Action                                                    |
|------|----------|-----------------|-----------------------------------------------------------|
| 0    | 1        | [5, 3, 4, 1, 2] | Compare $5 > 3 \rightarrow$ Swap, move back               |
| 1    | 0        | [3, 5, 4, 1, 2] | At start $\rightarrow$ move forward                       |
| 2    | 1        | [3, 5, 4, 1, 2] | Compare $5 > 4 \rightarrow$ Swap, move back               |
| 3    | 0        | [3, 4, 5, 1, 2] | At start $\rightarrow$ move forward                       |
| 4    | 1        | [3, 4, 5, 1, 2] | Compare $4 < 5 \rightarrow$ OK $\rightarrow$ move forward |
| 5    | 3        | [3, 4, 5, 1, 2] | Compare $5 > 1 \rightarrow$ Swap, move back               |

| Step | Position | Array State     | Action                            |
|------|----------|-----------------|-----------------------------------|
| 6    | 2        | [3, 4, 1, 5, 2] | Compare 4 > 1 → Swap, move back   |
| 7    | 1        | [3, 1, 4, 5, 2] | Compare 3 > 1 → Swap, move back   |
| 8    | 0        | [1, 3, 4, 5, 2] | At start → move forward           |
| 9    | ...      | ...             | Continue until sorted [1,2,3,4,5] |

## How Does It Work (Plain Language)?

The algorithm “walks” through the list:

1. If the current element is greater or equal to the previous one, move forward.
2. If not, swap them and move one step back.
3. Repeat until the end is reached.

If you reach the start of the array, step forward.

It's like Insertion Sort, but instead of shifting, it walks and swaps.

## Step-by-Step Process

| Step             | Condition                 | Action |
|------------------|---------------------------|--------|
| If A[i] ≥ A[i-1] | Move forward (i++)        |        |
| If A[i] < A[i-1] | Swap, move backward (i--) |        |
| If i == 0        | Move forward (i++)        |        |

## Tiny Code (Easy Versions)

C

```
#include <stdio.h>

void gnome_sort(int a[], int n) {
    int i = 1;
    while (i < n) {
        if (i == 0 || a[i] >= a[i - 1]) {
            i++;
        } else {
            int temp = a[i];
            a[i] = a[i - 1];
            a[i - 1] = temp;
        }
    }
}
```

```

        a[i - 1] = temp;
        i--;
    }
}

int main(void) {
    int a[] = {5, 3, 4, 1, 2};
    int n = sizeof(a) / sizeof(a[0]);
    gnome_sort(a, n);
    for (int i = 0; i < n; i++) printf("%d ", a[i]);
    printf("\n");
}

```

## Python

```

def gnome_sort(a):
    i = 1
    n = len(a)
    while i < n:
        if i == 0 or a[i] >= a[i - 1]:
            i += 1
        else:
            a[i], a[i - 1] = a[i - 1], a[i]
            i -= 1

arr = [5, 3, 4, 1, 2]
gnome_sort(arr)
print(arr)

```

## Why It Matters

- Demonstrates sorting through local correction
- Visually intuitive (good for animation)
- Requires no additional memory
- Stable and adaptive on partially sorted data

## A Gentle Proof (Why It Works)

Each time we swap out-of-order elements, we step back to verify order with the previous one. Thus, by the time we move forward again, all prior elements are guaranteed to be sorted.

Gnome Sort effectively performs Insertion Sort via adjacent swaps.

| Pass | Swaps | Movement    | Sorted Portion    |
|------|-------|-------------|-------------------|
| 1    | Few   | Backward    | Expands gradually |
| n    | Many  | Oscillating | Fully sorted      |

Worst-case swaps:  $O(n^2)$  Best-case (already sorted):  $O(n)$

## Try It Yourself

| Task | Description                             |
|------|-----------------------------------------|
| 1    | Sort [5, 3, 4, 1, 2] step by step       |
| 2    | Trace <i>i</i> pointer movement         |
| 3    | Compare with Insertion Sort shifts      |
| 4    | Animate using console output            |
| 5    | Try reversed input to see maximum swaps |

## Test Cases

| Input           | Output          | Swaps | Notes           |
|-----------------|-----------------|-------|-----------------|
| [3, 2, 1]       | [1, 2, 3]       | 3     | Many backtracks |
| [1, 2, 3]       | [1, 2, 3]       | 0     | Already sorted  |
| [5, 3, 4, 1, 2] | [1, 2, 3, 4, 5] | 8     | Moderate swaps  |

## Complexity

| Aspect       | Value    |
|--------------|----------|
| Time (Worst) | $O(n^2)$ |
| Time (Best)  | $O(n)$   |
| Space        | $O(1)$   |
| Stable       | Yes      |

| Aspect   | Value |
|----------|-------|
| Adaptive | Yes   |

Gnome Sort is a whimsical algorithm that teaches persistence: every time something's out of order, step back, fix it, and keep going. It's inefficient for large data but delightful for learning and visualization.

## 109 Odd-Even Sort

Odd-Even Sort, also known as Brick Sort, is a parallel-friendly variant of Bubble Sort. It alternates between comparing odd-even and even-odd indexed pairs to gradually sort the array. It's especially useful in parallel processing where pairs can be compared simultaneously.

### What Problem Are We Solving??

Bubble Sort compares every adjacent pair in one sweep. Odd-Even Sort breaks this into two alternating phases:

- Odd phase: compare (1,2), (3,4), (5,6), ...
- Even phase: compare (0,1), (2,3), (4,5), ...

Repeating these two passes ensures all adjacent pairs eventually become sorted.

It's ideal for parallel systems or hardware implementations since comparisons in each phase are independent.

### Example

| Phase | Type | Array State     | Action                        |
|-------|------|-----------------|-------------------------------|
| 0     | Init | [5, 3, 4, 1, 2] | Start                         |
| 1     | Even | [3, 5, 1, 4, 2] | Compare (0,1), (2,3), (4,5)   |
| 2     | Odd  | [3, 1, 5, 2, 4] | Compare (1,2), (3,4)          |
| 3     | Even | [1, 3, 2, 4, 5] | Compare (0,1), (2,3), (4,5)   |
| 4     | Odd  | [1, 2, 3, 4, 5] | Compare (1,2), (3,4) → Sorted |

## How Does It Work (Plain Language)?

Odd-Even Sort moves elements closer to their correct position with every alternating phase. It works like a traffic system: cars at even intersections move, then cars at odd intersections move. Over time, all cars (elements) line up in order.

## Step-by-Step Process

| Step | Action                      | Result               |
|------|-----------------------------|----------------------|
| 1    | Compare all even-odd pairs  | Swap if out of order |
| 2    | Compare all odd-even pairs  | Swap if out of order |
| 3    | Repeat until no swaps occur | Sorted array         |

## Tiny Code (Easy Versions)

C

```
#include <stdio.h>
#include <stdbool.h>

void odd_even_sort(int a[], int n) {
    bool sorted = false;
    while (!sorted) {
        sorted = true;

        // Even phase
        for (int i = 0; i < n - 1; i += 2) {
            if (a[i] > a[i + 1]) {
                int temp = a[i];
                a[i] = a[i + 1];
                a[i + 1] = temp;
                sorted = false;
            }
        }

        // Odd phase
        for (int i = 1; i < n - 1; i += 2) {
            if (a[i] > a[i + 1]) {
                int temp = a[i];
                a[i] = a[i + 1];
                a[i + 1] = temp;
                sorted = false;
            }
        }
    }
}
```

```

        a[i + 1] = temp;
        sorted = false;
    }
}
}

int main(void) {
    int a[] = {5, 3, 4, 1, 2};
    int n = sizeof(a) / sizeof(a[0]);
    odd_even_sort(a, n);
    for (int i = 0; i < n; i++) printf("%d ", a[i]);
    printf("\n");
}

```

## Python

```

def odd_even_sort(a):
    n = len(a)
    sorted = False
    while not sorted:
        sorted = True

        # Even phase
        for i in range(0, n - 1, 2):
            if a[i] > a[i + 1]:
                a[i], a[i + 1] = a[i + 1], a[i]
                sorted = False

        # Odd phase
        for i in range(1, n - 1, 2):
            if a[i] > a[i + 1]:
                a[i], a[i + 1] = a[i + 1], a[i]
                sorted = False

arr = [5, 3, 4, 1, 2]
odd_even_sort(arr)
print(arr)

```

## Why It Matters

- Demonstrates parallel sorting principles
- Conceptually simple, easy to visualize
- Can be implemented on parallel processors (SIMD, GPU)
- Stable and in-place

## A Gentle Proof (Why It Works)

Odd-Even Sort systematically removes all inversions by alternating comparisons:

- Even phase fixes pairs (0,1), (2,3), (4,5), ...
- Odd phase fixes pairs (1,2), (3,4), (5,6), ...

After n iterations, every element “bubbles” to its position. It behaves like Bubble Sort but is more structured and phase-based.

| Phase | Comparisons | Independent? | Swaps |
|-------|-------------|--------------|-------|
| Even  | $n/2$       | Yes          | Some  |
| Odd   | $n/2$       | Yes          | Some  |

Total complexity remains  $O(n^2)$ , but parallelizable phases reduce wall-clock time.

## Try It Yourself

| Task | Description                                |
|------|--------------------------------------------|
| 1    | Trace [5, 3, 4, 1, 2] through phases       |
| 2    | Count number of phases to sort             |
| 3    | Implement using threads (parallel version) |
| 4    | Compare with Bubble Sort                   |
| 5    | Animate even and odd passes                |

## Test Cases

| Input           | Output          | Phases | Notes              |
|-----------------|-----------------|--------|--------------------|
| [3, 2, 1]       | [1, 2, 3]       | 3      | Alternating passes |
| [1, 2, 3]       | [1, 2, 3]       | 1      | Early stop         |
| [5, 3, 4, 1, 2] | [1, 2, 3, 4, 5] | 4      | Moderate passes    |

| Input | Output | Phases | Notes |
|-------|--------|--------|-------|
|-------|--------|--------|-------|

## Complexity

| Aspect       | Value    |
|--------------|----------|
| Time (Worst) | $O(n^2)$ |
| Time (Best)  | $O(n)$   |
| Space        | $O(1)$   |
| Stable       | Yes      |
| Adaptive     | Yes      |

Odd-Even Sort shows that structure matters. By alternating phases, it opens the door to parallel sorting, where independent comparisons can run at once, a neat step toward high-performance sorting.

## 110 Stooge Sort

Stooge Sort is one of the most unusual and quirky recursive sorting algorithms. It's not efficient, but it's fascinating because it sorts by recursively sorting overlapping sections of the array, a great way to study recursion and algorithmic curiosity.

### What Problem Are We Solving??

Stooge Sort doesn't aim for speed. Instead, it provides an example of a recursive divide-and-conquer strategy that's neither efficient nor conventional. It divides the array into overlapping parts and recursively sorts them, twice on the first two-thirds, once on the last two-thirds.

This algorithm is often used for educational purposes to demonstrate how recursion can be applied in non-traditional ways.

### Example

| Step | Range  | Action                            | Array State |
|------|--------|-----------------------------------|-------------|
| 0    | [0..4] | Start sorting [5,3,4,1,2]         | [5,3,4,1,2] |
| 1    | [0..3] | Sort first 2/3 (4 elements)       | [3,4,1,5,2] |
| 2    | [1..4] | Sort last 2/3 (4 elements)        | [3,1,4,2,5] |
| 3    | [0..3] | Sort first 2/3 again (4 elements) | [1,3,2,4,5] |

| Step | Range  | Action                             | Array State |
|------|--------|------------------------------------|-------------|
| 4    | Repeat | Until subarrays shrink to length 1 | [1,2,3,4,5] |

## How Does It Work (Plain Language)?

If the first element is larger than the last, swap them. Then recursively:

1. Sort the first two-thirds of the array.
2. Sort the last two-thirds of the array.
3. Sort the first two-thirds again.

It's like checking the front, then the back, then rechecking the front, until everything settles in order.

## Step-by-Step Process

| Step | Action                                          |
|------|-------------------------------------------------|
| 1    | Compare first and last elements; swap if needed |
| 2    | Recursively sort first 2/3 of array             |
| 3    | Recursively sort last 2/3 of array              |
| 4    | Recursively sort first 2/3 again                |
| 5    | Stop when subarray length = 1                   |

## Tiny Code (Easy Versions)

### C

```
#include <stdio.h>

void stooge_sort(int a[], int l, int h) {
    if (a[l] > a[h]) {
        int temp = a[l];
        a[l] = a[h];
        a[h] = temp;
    }
    if (h - l + 1 > 2) {
        int t = (h - l + 1) / 3;
        stooge_sort(a, l, h - t);
        stooge_sort(a, l + t, h);
```

```

        stooge_sort(a, l, h - t);
    }
}

int main(void) {
    int a[] = {5, 3, 4, 1, 2};
    int n = sizeof(a) / sizeof(a[0]);
    stooge_sort(a, 0, n - 1);
    for (int i = 0; i < n; i++) printf("%d ", a[i]);
    printf("\n");
}

```

## Python

```

def stooge_sort(a, l, h):
    if a[l] > a[h]:
        a[l], a[h] = a[h], a[l]
    if h - l + 1 > 2:
        t = (h - l + 1) // 3
        stooge_sort(a, l, h - t)
        stooge_sort(a, l + t, h)
        stooge_sort(a, l, h - t)

arr = [5, 3, 4, 1, 2]
stooge_sort(arr, 0, len(arr) - 1)
print(arr)

```

## Why It Matters

- Demonstrates recursive divide-and-conquer logic
- A fun counterexample to “more recursion = more speed”
- Useful in theoretical discussions or algorithmic humor
- Helps build understanding of overlapping subproblems

## A Gentle Proof (Why It Works)

At each step, Stooge Sort ensures:

- The smallest element moves toward the front,
- The largest element moves toward the back,

- The array converges to sorted order through overlapping recursive calls.

Each recursion operates on 2/3 of the range, repeated 3 times, giving recurrence:

$$T(n) = 3T(2n/3) + O(1)$$

Solving it (using Master Theorem):

$$T(n) = O(n^{\log_{1.5} 3}) \approx O(n^{2.7095})$$

Slower than Bubble Sort ( $O(n^2)$ )!

| Step  | Subarray Length | Recursive Calls | Work per Level |
|-------|-----------------|-----------------|----------------|
| n     | 3               | $O(1)$          | Constant       |
| $n/2$ | $3 \times 3$    | $O(3)$          | Larger         |
| ...   | ...             | ...             | ...            |

### Try It Yourself

| Task | Description                                             |
|------|---------------------------------------------------------|
| 1    | Sort [5, 3, 4, 1, 2] manually and trace recursive calls |
| 2    | Count total swaps                                       |
| 3    | Print recursion depth                                   |
| 4    | Compare with Merge Sort steps                           |
| 5    | Measure runtime for n=10, 100, 1000                     |

### Test Cases

| Input           | Output          | Recursion Depth | Notes             |
|-----------------|-----------------|-----------------|-------------------|
| [3, 2, 1]       | [1, 2, 3]       | 3               | Works recursively |
| [1, 2, 3]       | [1, 2, 3]       | 1               | Minimal swaps     |
| [5, 3, 4, 1, 2] | [1, 2, 3, 4, 5] | 6               | Many overlaps     |

### Complexity

| Aspect       | Value                       |
|--------------|-----------------------------|
| Time (Worst) | $O(n^2.7095)$               |
| Time (Best)  | $O(n^2.7095)$               |
| Space        | $O(\log n)$ recursion stack |
| Stable       | No                          |
| Adaptive     | No                          |

Stooge Sort is a delightful oddity, slow, redundant, but undeniably creative. It reminds us that not every recursive idea leads to efficiency, and that algorithm design is as much art as science.

## Section 12. Divide and conquer sorting

### 111 Merge Sort

Merge Sort is one of the most famous divide-and-conquer sorting algorithms. It splits the array into halves, sorts each half recursively, and then merges the two sorted halves into a single sorted array. It guarantees  $O(n \log n)$  performance, is stable, and serves as the backbone of many modern sorting libraries.

#### What Problem Are We Solving??

We want a sorting algorithm that is:

- Efficient on large datasets ( $O(n \log n)$ )
- Stable (preserves equal element order)
- Predictable (no worst-case degradation)

Merge Sort achieves this by dividing the problem into smaller, easily solved subproblems and combining their results.

It's ideal for:

- Sorting linked lists
- External sorting (on disk)
- Stable merges (for multi-key sorting)

#### Example

| Step | Action              | Result                |
|------|---------------------|-----------------------|
| 0    | Input               | [5, 3, 4, 1, 2]       |
| 1    | Split               | [5, 3, 4] and [1, 2]  |
| 2    | Split further       | [5], [3, 4], [1], [2] |
| 3    | Sort subarrays      | [3, 4, 5], [1, 2]     |
| 4    | Merge sorted halves | [1, 2, 3, 4, 5]       |

## How Does It Work (Plain Language)?

1. Divide: Split the array into two halves.
2. Conquer: Recursively sort both halves.
3. Combine: Merge the sorted halves into a single sorted array.

Think of it as sorting two smaller piles, then interleaving them in order, like merging two stacks of playing cards.

## Step-by-Step Process

- If the array has 0 or 1 element, it's already sorted.
- Recursively sort left and right halves.
- Use a helper merge function to combine them.

## Tiny Code (Easy Versions)

C

```
#include <stdio.h>

void merge(int a[], int left, int mid, int right) {
    int n1 = mid - left + 1, n2 = right - mid;
    int L[n1], R[n2];

    for (int i = 0; i < n1; i++) L[i] = a[left + i];
    for (int j = 0; j < n2; j++) R[j] = a[mid + 1 + j];

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) a[k++] = L[i++];
        else a[k++] = R[j++];
    }
}
```

```

        while (i < n1) a[k++] = L[i++];
        while (j < n2) a[k++] = R[j++];
    }

void merge_sort(int a[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        merge_sort(a, left, mid);
        merge_sort(a, mid + 1, right);
        merge(a, left, mid, right);
    }
}

int main(void) {
    int a[] = {5, 3, 4, 1, 2};
    int n = sizeof(a) / sizeof(a[0]);
    merge_sort(a, 0, n - 1);
    for (int i = 0; i < n; i++) printf("%d ", a[i]);
    printf("\n");
}

```

## Python

```

def merge_sort(a):
    if len(a) > 1:
        mid = len(a) // 2
        left = a[:mid]
        right = a[mid:]

        merge_sort(left)
        merge_sort(right)

        i = j = k = 0
        while i < len(left) and j < len(right):
            if left[i] <= right[j]:
                a[k] = left[i]
                i += 1
            else:
                a[k] = right[j]
                j += 1
            k += 1

```

```

while i < len(left):
    a[k] = left[i]
    i += 1
    k += 1
while j < len(right):
    a[k] = right[j]
    j += 1
    k += 1

arr = [5, 3, 4, 1, 2]
merge_sort(arr)
print(arr)

```

## Why It Matters

- Stable: Keeps relative order of equal elements
- Deterministic  $O(n \log n)$ : Always efficient
- Parallelizable: Subarrays can be sorted independently
- Foundation: For hybrid algorithms like TimSort and External Merge Sort

## A Gentle Proof (Why It Works)

Merge Sort divides the array into two halves at each level. There are  $\log n$  levels of recursion, and each merge takes  $O(n)$  time.

So total time:

$$T(n) = O(n \log n)$$

Merging is linear because each element is copied once per level.

| Step | Work              | Subarrays | Total         |
|------|-------------------|-----------|---------------|
| 1    | $O(n)$            | 1         | $O(n)$        |
| 2    | $O(n/2) \times 2$ | 2         | $O(n)$        |
| 3    | $O(n/4) \times 4$ | 4         | $O(n)$        |
| ...  | ...               | ...       | $O(n \log n)$ |

## Try It Yourself

1. Split [5, 3, 4, 1, 2] into halves step by step.
2. Merge [3, 5] and [1, 4] manually.
3. Trace the recursive calls on paper.
4. Implement an iterative bottom-up version.
5. Modify to sort descending.
6. Print arrays at each merge step.
7. Compare the number of comparisons vs. Bubble Sort.
8. Try merging two pre-sorted arrays [1,3,5] and [2,4,6].
9. Sort a list of strings (alphabetically).
10. Visualize the recursion tree for n = 8.

## Test Cases

| Input           | Output          | Notes           |
|-----------------|-----------------|-----------------|
| [3, 2, 1]       | [1, 2, 3]       | Standard test   |
| [1, 2, 3]       | [1, 2, 3]       | Already sorted  |
| [5, 3, 4, 1, 2] | [1, 2, 3, 4, 5] | General case    |
| [2, 2, 1, 1]    | [1, 1, 2, 2]    | Tests stability |

## Complexity

| Aspect         | Value         |
|----------------|---------------|
| Time (Worst)   | $O(n \log n)$ |
| Time (Best)    | $O(n \log n)$ |
| Time (Average) | $O(n \log n)$ |
| Space          | $O(n)$        |
| Stable         | Yes           |
| Adaptive       | No            |

Merge Sort is your first taste of divide and conquer sorting, calm, reliable, and elegant. It divides the problem cleanly, conquers recursively, and merges with precision.

## 112 Iterative Merge Sort

Iterative Merge Sort is a non-recursive version of Merge Sort that uses bottom-up merging. Instead of dividing recursively, it starts with subarrays of size 1 and iteratively merges them in

pairs, doubling the size each round. This makes it ideal for environments where recursion is expensive or limited.

### **What Problem Are We Solving??**

Recursive Merge Sort requires function calls and stack space. In some systems, recursion might be slow or infeasible (e.g. embedded systems, large arrays). Iterative Merge Sort avoids recursion by sorting iteratively, merging subarrays of increasing size until the entire array is sorted.

It's especially handy for:

- Iterative environments (no recursion)
- Large data sets (predictable memory)
- External sorting with iterative passes

### **Example**

| Step | Subarray Size | Action                                 | Array State     |
|------|---------------|----------------------------------------|-----------------|
| 0    | 1             | Each element is trivially sorted       | [5, 3, 4, 1, 2] |
| 1    | 1             | Merge pairs of 1-element subarrays     | [3, 5, 1, 4, 2] |
| 2    | 2             | Merge pairs of 2-element subarrays     | [1, 3, 4, 5, 2] |
| 3    | 4             | Merge 4-element sorted block with last | [1, 2, 3, 4, 5] |
| 4    | Done          | Fully sorted                           | [1, 2, 3, 4, 5] |

### **How Does It Work (Plain Language)?**

Think of it as sorting small groups first and then merging those into bigger groups, no recursion required.

Process:

1. Start with subarrays of size 1 (already sorted).
2. Merge adjacent pairs of subarrays.
3. Double the subarray size and repeat.
4. Continue until subarray size = n.

### **Step-by-Step Process**

- Outer loop: size = 1, 2, 4, 8, ... until n
- Inner loop: merge every two adjacent blocks of given size

## Tiny Code (Easy Versions)

C

```
#include <stdio.h>

void merge(int a[], int left, int mid, int right) {
    int n1 = mid - left + 1, n2 = right - mid;
    int L[n1], R[n2];
    for (int i = 0; i < n1; i++) L[i] = a[left + i];
    for (int j = 0; j < n2; j++) R[j] = a[mid + 1 + j];

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) a[k++] = L[i++];
        else a[k++] = R[j++];
    }
    while (i < n1) a[k++] = L[i++];
    while (j < n2) a[k++] = R[j++];
}

void iterative_merge_sort(int a[], int n) {
    for (int size = 1; size < n; size *= 2) {
        for (int left = 0; left < n - 1; left += 2 * size) {
            int mid = left + size - 1;
            int right = (left + 2 * size - 1 < n - 1) ? (left + 2 * size - 1) : (n - 1);
            if (mid < right)
                merge(a, left, mid, right);
        }
    }
}

int main(void) {
    int a[] = {5, 3, 4, 1, 2};
    int n = sizeof(a) / sizeof(a[0]);
    iterative_merge_sort(a, n);
    for (int i = 0; i < n; i++) printf("%d ", a[i]);
    printf("\n");
}
```

Python

```

def merge(a, left, mid, right):
    left_arr = a[left:mid+1]
    right_arr = a[mid+1:right+1]
    i = j = 0
    k = left
    while i < len(left_arr) and j < len(right_arr):
        if left_arr[i] <= right_arr[j]:
            a[k] = left_arr[i]
            i += 1
        else:
            a[k] = right_arr[j]
            j += 1
        k += 1
    while i < len(left_arr):
        a[k] = left_arr[i]
        i += 1
        k += 1
    while j < len(right_arr):
        a[k] = right_arr[j]
        j += 1
        k += 1

def iterative_merge_sort(a):
    n = len(a)
    size = 1
    while size < n:
        for left in range(0, n, 2 * size):
            mid = min(left + size - 1, n - 1)
            right = min(left + 2 * size - 1, n - 1)
            if mid < right:
                merge(a, left, mid, right)
        size *= 2

arr = [5, 3, 4, 1, 2]
iterative_merge_sort(arr)
print(arr)

```

## Why It Matters

- Eliminates recursion (more predictable memory usage)
- Still guarantees  $O(n \log n)$  performance

- Useful for iterative, bottom-up, or external sorting
- Easier to parallelize since merge operations are independent

### A Gentle Proof (Why It Works)

Each iteration doubles the sorted block size. Since each element participates in  $\log n$  merge levels, and each level costs  $O(n)$  work, total cost:

$$T(n) = O(n \log n)$$

Like recursive Merge Sort, each merge step is linear, and merging subarrays is stable.

| Iteration | Block Size | Merges | Work   |
|-----------|------------|--------|--------|
| 1         | 1          | $n/2$  | $O(n)$ |
| 2         | 2          | $n/4$  | $O(n)$ |
| 3         | 4          | $n/8$  | $O(n)$ |
| ...       | ...        | ...    | ...    |

Total =  $O(n \log n)$

### Try It Yourself

1. Sort [5, 3, 4, 1, 2] manually using bottom-up passes.
2. Trace each pass: subarray size = 1 → 2 → 4.
3. Print intermediate arrays after each pass.
4. Compare recursion depth with recursive version.
5. Implement a space-efficient version (in-place merge).
6. Modify to sort descending.
7. Apply to linked list version.
8. Test performance on large array ( $n = 10^6$ ).
9. Visualize merging passes as a tree.
10. Implement on external storage (file-based).

### Test Cases

| Input     | Output    | Notes          |
|-----------|-----------|----------------|
| [3, 2, 1] | [1, 2, 3] | Small test     |
| [1, 2, 3] | [1, 2, 3] | Already sorted |

| Input           | Output          | Notes           |
|-----------------|-----------------|-----------------|
| [5, 3, 4, 1, 2] | [1, 2, 3, 4, 5] | General test    |
| [2, 2, 1, 1]    | [1, 1, 2, 2]    | Tests stability |

## Complexity

| Aspect       | Value         |
|--------------|---------------|
| Time (Worst) | $O(n \log n)$ |
| Time (Best)  | $O(n \log n)$ |
| Space        | $O(n)$        |
| Stable       | Yes           |
| Adaptive     | No            |

Iterative Merge Sort is the non-recursive twin of classic Merge Sort, efficient, stable, and memory-predictable, making it perfect when stack space is at a premium.

## 113 Quick Sort

Quick Sort is one of the fastest and most widely used sorting algorithms. It works by partitioning an array into two halves around a pivot element and recursively sorting the two parts. With average-case  $O(n \log n)$  performance and in-place operation, it's the go-to choice in many libraries and real-world systems.

### What Problem Are We Solving??

We need a sorting algorithm that's:

- Efficient in practice (fast average case)
- In-place (minimal memory use)
- Divide-and-conquer-based (parallelizable)

Quick Sort partitions the array so that:

- All elements smaller than the pivot go left
- All elements larger go right

Then it sorts each half recursively.

It's ideal for:

- Large datasets in memory
- Systems where memory allocation is limited
- Average-case performance optimization

## Example

| Step | Pivot | Action         | Array State     |
|------|-------|----------------|-----------------|
| 0    | 5     | Partition      | [3, 4, 1, 2, 5] |
| 1    | 3     | Partition left | [1, 2, 3, 4, 5] |
| 2    | Done  | Sorted         | [1, 2, 3, 4, 5] |

## How Does It Work (Plain Language)?

1. Choose a pivot element.
2. Rearrange (partition) array so:
  - Elements smaller than pivot move to the left
  - Elements larger than pivot move to the right
3. Recursively apply the same logic to each half.

Think of the pivot as the “divider” that splits the unsorted array into two smaller problems.

## Step-by-Step Process

| Step | Action                                    |
|------|-------------------------------------------|
| 1    | Select a pivot (e.g., last element)       |
| 2    | Partition array around pivot              |
| 3    | Recursively sort left and right subarrays |
| 4    | Stop when subarray size = 1               |

## Tiny Code (Easy Versions)

### C (Lomuto Partition Scheme)

```

#include <stdio.h>

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int partition(int a[], int low, int high) {
    int pivot = a[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (a[j] < pivot) {
            i++;
            swap(&a[i], &a[j]);
        }
    }
    swap(&a[i + 1], &a[high]);
    return i + 1;
}

void quick_sort(int a[], int low, int high) {
    if (low < high) {
        int pi = partition(a, low, high);
        quick_sort(a, low, pi - 1);
        quick_sort(a, pi + 1, high);
    }
}

int main(void) {
    int a[] = {5, 3, 4, 1, 2};
    int n = sizeof(a) / sizeof(a[0]);
    quick_sort(a, 0, n - 1);
    for (int i = 0; i < n; i++) printf("%d ", a[i]);
    printf("\n");
}

```

## Python

```

def partition(a, low, high):
    pivot = a[high]

```

```

i = low - 1
for j in range(low, high):
    if a[j] < pivot:
        i += 1
        a[i], a[j] = a[j], a[i]
a[i + 1], a[high] = a[high], a[i + 1]
return i + 1

def quick_sort(a, low, high):
    if low < high:
        pi = partition(a, low, high)
        quick_sort(a, low, pi - 1)
        quick_sort(a, pi + 1, high)

arr = [5, 3, 4, 1, 2]
quick_sort(arr, 0, len(arr) - 1)
print(arr)

```

## Why It Matters

- In-place and fast in most real-world cases
- Divide and conquer: naturally parallelizable
- Often used as the default sorting algorithm in libraries (C, Java, Python)
- Introduces partitioning, a key algorithmic pattern

## A Gentle Proof (Why It Works)

Each partition divides the problem into smaller subarrays. Average partition splits are balanced, giving  $O(\log n)$  depth and  $O(n)$  work per level:

$$T(n) = 2T(n/2) + O(n) = O(n \log n)$$

If the pivot is poor (e.g. smallest or largest), complexity degrades:

$$T(n) = T(n - 1) + O(n) = O(n^2)$$

| Case    | Partition Quality | Complexity    |
|---------|-------------------|---------------|
| Best    | Perfect halves    | $O(n \log n)$ |
| Average | Random            | $O(n \log n)$ |

| Case  | Partition Quality | Complexity |
|-------|-------------------|------------|
| Worst | Unbalanced        | $O(n^2)$   |

Choosing pivots wisely (randomization, median-of-three) avoids worst-case splits.

### Try It Yourself

1. Sort [5, 3, 4, 1, 2] and trace partitions.
2. Change pivot selection (first, middle, random).
3. Count comparisons and swaps for each case.
4. Implement using Hoare Partition scheme.
5. Modify to sort descending.
6. Visualize recursion tree for n = 8.
7. Compare runtime with Merge Sort.
8. Try sorted input [1, 2, 3, 4, 5] and note behavior.
9. Add a counter to count recursive calls.
10. Implement tail recursion optimization.

### Test Cases

| Input           | Output          | Notes                     |
|-----------------|-----------------|---------------------------|
| [3, 2, 1]       | [1, 2, 3]       | Basic                     |
| [1, 2, 3]       | [1, 2, 3]       | Worst case (sorted input) |
| [5, 3, 4, 1, 2] | [1, 2, 3, 4, 5] | General                   |
| [2, 2, 1, 1]    | [1, 1, 2, 2]    | Duplicates                |

### Complexity

| Aspect         | Value                   |
|----------------|-------------------------|
| Time (Best)    | $O(n \log n)$           |
| Time (Average) | $O(n \log n)$           |
| Time (Worst)   | $O(n^2)$                |
| Space          | $O(\log n)$ (recursion) |
| Stable         | No                      |
| Adaptive       | No                      |

Quick Sort is the practical workhorse of sorting, swift, elegant, and widely loved. It teaches how a single smart pivot can bring order to chaos.

## 114 Hoare Partition Scheme

The Hoare Partition Scheme is an early and elegant version of Quick Sort's partitioning method, designed by C.A.R. Hoare himself. It's more efficient than the Lomuto scheme in many cases because it does fewer swaps and uses two pointers moving inward from both ends.

### What Problem Are We Solving??

In Quick Sort, we need a way to divide an array into two parts:

- Elements less than or equal to the pivot
- Elements greater than or equal to the pivot

Hoare's scheme achieves this using two indices that move toward each other, swapping elements that are out of place. It reduces the number of swaps compared to the Lomuto scheme and often performs better on real data.

It's especially useful for:

- Large arrays (fewer writes)
- Performance-critical systems
- In-place partitioning without extra space

### Example

| Step | Pivot | Left (i) | Right (j) | Array State     | Action                           |
|------|-------|----------|-----------|-----------------|----------------------------------|
| 0    | 5     | i=0      | j=4       | [5, 3, 4, 1, 2] | pivot = 5                        |
| 1    | 5     | i→2      | j→4       | [5, 3, 4, 1, 2] | a[j]=2<5 swap(5,2) → [2,3,4,1,5] |
| 2    | 5     | i→2      | j→3       | [2,3,4,1,5]     | a[i]=4>5? no swap                |
| 3    | stop  | -        | -         | [2,3,4,1,5]     | partition done                   |

The pivot ends up near its correct position, but not necessarily in the final index.

## How Does It Work (Plain Language)?

The algorithm picks a pivot (commonly the first element), then moves two pointers:

- Left pointer (i): moves right, skipping small elements
- Right pointer (j): moves left, skipping large elements

When both pointers find misplaced elements, they are swapped. This continues until they cross, at that point, the array is partitioned.

## Step-by-Step Process

1. Choose a pivot (e.g. first element).
2. Set two indices: `i = left - 1, j = right + 1`.
3. Increment `i` until `a[i] >= pivot`.
4. Decrement `j` until `a[j] <= pivot`.
5. If `i < j`, swap `a[i]` and `a[j]`. Otherwise, return `j` (partition index).

## Tiny Code (Easy Versions)

C

```
#include <stdio.h>

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int hoare_partition(int a[], int low, int high) {
    int pivot = a[low];
    int i = low - 1;
    int j = high + 1;

    while (1) {
        do { i++; } while (a[i] < pivot);
        do { j--; } while (a[j] > pivot);
        if (i >= j) return j;
        swap(&a[i], &a[j]);
    }
}
```

```

void quick_sort_hoare(int a[], int low, int high) {
    if (low < high) {
        int p = hoare_partition(a, low, high);
        quick_sort_hoare(a, low, p);
        quick_sort_hoare(a, p + 1, high);
    }
}

int main(void) {
    int a[] = {5, 3, 4, 1, 2};
    int n = sizeof(a) / sizeof(a[0]);
    quick_sort_hoare(a, 0, n - 1);
    for (int i = 0; i < n; i++) printf("%d ", a[i]);
    printf("\n");
}

```

## Python

```

def hoare_partition(a, low, high):
    pivot = a[low]
    i = low - 1
    j = high + 1
    while True:
        i += 1
        while a[i] < pivot:
            i += 1
        j -= 1
        while a[j] > pivot:
            j -= 1
        if i >= j:
            return j
        a[i], a[j] = a[j], a[i]

def quick_sort_hoare(a, low, high):
    if low < high:
        p = hoare_partition(a, low, high)
        quick_sort_hoare(a, low, p)
        quick_sort_hoare(a, p + 1, high)

arr = [5, 3, 4, 1, 2]

```

```
quick_sort_hoare(arr, 0, len(arr) - 1)
print(arr)
```

## Why It Matters

- Fewer swaps than Lomuto partition
- More efficient in practice on most datasets
- Still in-place, divide-and-conquer,  $O(n \log n)$  average
- Introduces the idea of two-pointer partitioning

## A Gentle Proof (Why It Works)

The loop invariants ensure that:

- Left side: all elements < pivot
- Right side: all elements > pivot
- i and j move inward until they cross. When they cross, all elements are partitioned correctly.

The pivot does not end in its final sorted position, but the subarrays can be recursively sorted independently.

$$T(n) = T(k) + T(n - k - 1) + O(n)$$

Average complexity  $O(n \log n)$ ; worst-case  $O(n^2)$  if pivot is poor.

| Case    | Pivot   | Behavior         | Complexity    |
|---------|---------|------------------|---------------|
| Best    | Median  | Balanced halves  | $O(n \log n)$ |
| Average | Random  | Slight imbalance | $O(n \log n)$ |
| Worst   | Min/Max | Unbalanced       | $O(n^2)$      |

## Try It Yourself

1. Sort [5, 3, 4, 1, 2] step by step using Hoare's scheme.
2. Print i and j at each iteration.
3. Compare with Lomuto's version on the same array.
4. Try different pivot positions (first, last, random).
5. Measure number of swaps vs. Lomuto.
6. Modify to sort descending.

7. Visualize partition boundaries.
8. Test on array with duplicates [3,3,2,1,4].
9. Implement hybrid pivot selection (median-of-three).
10. Compare runtime with Merge Sort.

## Test Cases

| Input           | Output          | Notes                     |
|-----------------|-----------------|---------------------------|
| [3, 2, 1]       | [1, 2, 3]       | Simple                    |
| [5, 3, 4, 1, 2] | [1, 2, 3, 4, 5] | General                   |
| [2, 2, 1, 1]    | [1, 1, 2, 2]    | Works with duplicates     |
| [1, 2, 3, 4, 5] | [1, 2, 3, 4, 5] | Sorted input (worst case) |

## Complexity

| Aspect         | Value                 |
|----------------|-----------------------|
| Time (Best)    | $O(n \log n)$         |
| Time (Average) | $O(n \log n)$         |
| Time (Worst)   | $O(n^2)$              |
| Space          | $O(\log n)$ recursion |
| Stable         | No                    |
| Adaptive       | No                    |

Hoare Partition Scheme is elegant and efficient, the original genius of Quick Sort. Its two-pointer dance is graceful and economical, a timeless classic in algorithm design.

## 115 Lomuto Partition Scheme

The Lomuto Partition Scheme is a simple and widely taught method for partitioning in Quick Sort. It's easier to understand and implement than Hoare's scheme, though it often performs slightly more swaps. It always selects a pivot (commonly the last element) and partitions the array in a single forward pass.

## What Problem Are We Solving??

We need a clear and intuitive way to partition an array around a pivot, ensuring all smaller elements go to the left and all larger elements go to the right.

Lomuto's method uses one scanning pointer and one boundary pointer, making it easy for beginners and ideal for pedagogical purposes or small datasets.

## Example

| Step | Pivot | i<br>(Boundary) | j (Scan) | Array State                              | Action    |
|------|-------|-----------------|----------|------------------------------------------|-----------|
| 0    | 2     | i=-1            | j=0      | [5, 3, 4, 1, 2]                          | pivot = 2 |
| 1    | 2     | i=-1            | j=0      | a[0]=5>2 → no swap                       |           |
| 2    | 2     | i=-1            | j=1      | a[1]=3>2 → no swap                       |           |
| 3    | 2     | i=-1            | j=2      | a[2]=4>2 → no swap                       |           |
| 4    | 2     | i=0             | j=3      | a[3]=1<2 → swap(5,1) → [1,3,4,5,2]       |           |
| 5    | 2     | i=0             | j=4      | end of scan; swap pivot with a[i+1]=a[1] |           |
| 6    | Done  | -               | -        | [1,2,4,5,3] partitioned                  |           |

Pivot 2 is placed in its final position at index 1. Elements left of 2 are smaller, right are larger.

## How Does It Work (Plain Language)?

1. Choose a pivot (commonly last element).
2. Initialize boundary pointer  $i$  before start of array.
3. Iterate through array with pointer  $j$ :
  - If  $a[j] < \text{pivot}$ , increment  $i$  and swap  $a[i]$  with  $a[j]$ .
4. After loop, swap  $\text{pivot}$  into position  $i + 1$ .
5. Return  $i + 1$  (pivot's final position).

It's like separating a deck of cards, you keep moving smaller cards to the front as you scan.

## Step-by-Step Process

| Step | Action                                   |
|------|------------------------------------------|
| 1    | Choose pivot (usually last element)      |
| 2    | Move smaller elements before pivot       |
| 3    | Move larger elements after pivot         |
| 4    | Place pivot in its correct position      |
| 5    | Return pivot index for recursive sorting |

## Tiny Code (Easy Versions)

C

```
#include <stdio.h>

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int lomuto_partition(int a[], int low, int high) {
    int pivot = a[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (a[j] < pivot) {
            i++;
            swap(&a[i], &a[j]);
        }
    }
    swap(&a[i + 1], &a[high]);
    return i + 1;
}

void quick_sort_lomuto(int a[], int low, int high) {
    if (low < high) {
        int pi = lomuto_partition(a, low, high);
        quick_sort_lomuto(a, low, pi - 1);
        quick_sort_lomuto(a, pi + 1, high);
    }
}

int main(void) {
```

```

int a[] = {5, 3, 4, 1, 2};
int n = sizeof(a) / sizeof(a[0]);
quick_sort_lomuto(a, 0, n - 1);
for (int i = 0; i < n; i++) printf("%d ", a[i]);
printf("\n");
}

```

## Python

```

def lomuto_partition(a, low, high):
    pivot = a[high]
    i = low - 1
    for j in range(low, high):
        if a[j] < pivot:
            i += 1
            a[i], a[j] = a[j], a[i]
    a[i + 1], a[high] = a[high], a[i + 1]
    return i + 1

def quick_sort_lomuto(a, low, high):
    if low < high:
        pi = lomuto_partition(a, low, high)
        quick_sort_lomuto(a, low, pi - 1)
        quick_sort_lomuto(a, pi + 1, high)

arr = [5, 3, 4, 1, 2]
quick_sort_lomuto(arr, 0, len(arr) - 1)
print(arr)

```

## Why It Matters

- Simple and easy to implement
- Pivot ends in correct final position each step
- Useful for educational demonstration of Quick Sort
- Common in textbooks and basic Quick Sort examples

## A Gentle Proof (Why It Works)

Invariant:

- Elements left of  $i$  are smaller than pivot.
- Elements between  $i$  and  $j$  are greater or not yet checked. At the end, swapping pivot with  $a[i+1]$  places it in its final position.

Time complexity:

$$T(n) = T(k) + T(n - k - 1) + O(n)$$

Average:  $O(n \log n)$  Worst (already sorted):  $O(n^2)$

| Case    | Partition  | Complexity    |
|---------|------------|---------------|
| Best    | Balanced   | $O(n \log n)$ |
| Average | Random     | $O(n \log n)$ |
| Worst   | Unbalanced | $O(n^2)$      |

### Try It Yourself

1. Sort  $[5, 3, 4, 1, 2]$  using Lomuto step by step.
2. Trace  $i$  and  $j$  positions at each comparison.
3. Compare with Hoare partition's number of swaps.
4. Test with sorted input, see worst case.
5. Randomize pivot to avoid worst-case.
6. Modify to sort descending order.
7. Count total swaps and comparisons.
8. Combine with tail recursion optimization.
9. Visualize partition boundary after each pass.
10. Implement a hybrid Quick Sort using Lomuto for small arrays.

### Test Cases

| Input             | Output            | Notes              |
|-------------------|-------------------|--------------------|
| $[3, 2, 1]$       | $[1, 2, 3]$       | Simple             |
| $[1, 2, 3]$       | $[1, 2, 3]$       | Worst-case         |
| $[5, 3, 4, 1, 2]$ | $[1, 2, 3, 4, 5]$ | General            |
| $[2, 2, 1, 1]$    | $[1, 1, 2, 2]$    | Handles duplicates |

### Complexity

| Aspect         | Value                 |
|----------------|-----------------------|
| Time (Best)    | $O(n \log n)$         |
| Time (Average) | $O(n \log n)$         |
| Time (Worst)   | $O(n^2)$              |
| Space          | $O(\log n)$ recursion |
| Stable         | No                    |
| Adaptive       | No                    |

Lomuto's scheme is the friendly teacher of Quick Sort, easy to grasp, simple to code, and perfect for building intuition about partitioning and divide-and-conquer sorting.

## 116 Randomized Quick Sort

Randomized Quick Sort enhances the classic Quick Sort by choosing the pivot randomly. This small tweak eliminates the risk of hitting the worst-case  $O(n^2)$  behavior on already sorted or adversarial inputs, making it one of the most robust and practical sorting strategies in real-world use.

### What Problem Are We Solving??

Regular Quick Sort can degrade badly if the pivot is chosen poorly (for example, always picking the first or last element in a sorted array). Randomized Quick Sort fixes this by selecting a random pivot, ensuring that, on average, partitions are balanced, regardless of input distribution.

This makes it ideal for:

- Unpredictable or adversarial inputs
- Large datasets where worst-case avoidance matters
- Performance-critical systems requiring consistent behavior

### Example

| Step | Action                    | Array State     | Pivot        |
|------|---------------------------|-----------------|--------------|
| 0    | Choose random pivot       | [5, 3, 4, 1, 2] | 4            |
| 1    | Partition around 4        | [3, 2, 1, 4, 5] | 4 at index 3 |
| 2    | Recurse on left [3, 2, 1] | [1, 2, 3]       | 2            |
| 3    | Merge subarrays           | [1, 2, 3, 4, 5] | Done         |

Randomization ensures the pivot is unlikely to create unbalanced partitions.

### How Does It Work (Plain Language)?

It's the same Quick Sort, but before partitioning, we randomly pick one element and use it as the pivot. This random pivot is swapped into the last position, and the normal Lomuto or Hoare partitioning continues.

This small randomness makes it robust and efficient on average, even for worst-case inputs.

### Step-by-Step Process

1. Pick a random pivot index between `low` and `high`.
2. Swap the random pivot with the last element.
3. Partition the array (e.g., Lomuto or Hoare).
4. Recursively sort left and right partitions.

### Tiny Code (Easy Versions)

#### C (Lomuto + Random Pivot)

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int lomuto_partition(int a[], int low, int high) {
    int pivot = a[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (a[j] < pivot) {
            i++;
            swap(&a[i], &a[j]);
        }
    }
}
```

```

        swap(&a[i + 1], &a[high]);
        return i + 1;
    }

int randomized_partition(int a[], int low, int high) {
    int pivotIndex = low + rand() % (high - low + 1);
    swap(&a[pivotIndex], &a[high]);
    return lomuto_partition(a, low, high);
}

void randomized_quick_sort(int a[], int low, int high) {
    if (low < high) {
        int pi = randomized_partition(a, low, high);
        randomized_quick_sort(a, low, pi - 1);
        randomized_quick_sort(a, pi + 1, high);
    }
}

int main(void) {
    srand(time(NULL));
    int a[] = {5, 3, 4, 1, 2};
    int n = sizeof(a) / sizeof(a[0]);
    randomized_quick_sort(a, 0, n - 1);
    for (int i = 0; i < n; i++) printf("%d ", a[i]);
    printf("\n");
}

```

## Python

```

import random

def lomuto_partition(a, low, high):
    pivot = a[high]
    i = low - 1
    for j in range(low, high):
        if a[j] < pivot:
            i += 1
            a[i], a[j] = a[j], a[i]
    a[i + 1], a[high] = a[high], a[i + 1]
    return i + 1

```

```

def randomized_partition(a, low, high):
    pivot_index = random.randint(low, high)
    a[pivot_index], a[high] = a[high], a[pivot_index]
    return lomuto_partition(a, low, high)

def randomized_quick_sort(a, low, high):
    if low < high:
        pi = randomized_partition(a, low, high)
        randomized_quick_sort(a, low, pi - 1)
        randomized_quick_sort(a, pi + 1, high)

arr = [5, 3, 4, 1, 2]
randomized_quick_sort(arr, 0, len(arr) - 1)
print(arr)

```

## Why It Matters

- Prevents worst-case  $O(n^2)$  behavior
- Simple yet highly effective
- Ensures consistent average-case across all inputs
- Foundation for Randomized Select and Randomized Algorithms

## A Gentle Proof (Why It Works)

Random pivot selection ensures the expected split size is balanced, independent of input order. Each pivot divides the array such that expected recursion depth is  $O(\log n)$  and total comparisons  $O(n \log n)$ .

Expected complexity:

$$E[T(n)] = O(n \log n)$$

Worst-case only occurs with extremely low probability ( $1/n!$ ).

| Case    | Pivot Choice   | Complexity    |
|---------|----------------|---------------|
| Best    | Balanced       | $O(n \log n)$ |
| Average | Random         | $O(n \log n)$ |
| Worst   | Unlucky (rare) | $O(n^2)$      |

## Try It Yourself

1. Sort [5, 3, 4, 1, 2] multiple times, note pivot differences.
2. Print pivot each recursive call.
3. Compare against deterministic pivot (first, last).
4. Test on sorted input [1, 2, 3, 4, 5].
5. Test on reverse input [5, 4, 3, 2, 1].
6. Count recursive depth across runs.
7. Modify to use Hoare partition.
8. Implement `random.choice()` version.
9. Compare runtime vs. normal Quick Sort.
10. Seed RNG to reproduce same run.

## Test Cases

| Input           | Output          | Notes                 |
|-----------------|-----------------|-----------------------|
| [3, 2, 1]       | [1, 2, 3]       | Random pivot each run |
| [1, 2, 3]       | [1, 2, 3]       | Avoids $O(n^2)$       |
| [5, 3, 4, 1, 2] | [1, 2, 3, 4, 5] | General               |
| [2, 2, 1, 1]    | [1, 1, 2, 2]    | Handles duplicates    |

## Complexity

| Aspect         | Value           |
|----------------|-----------------|
| Time (Best)    | $O(n \log n)$   |
| Time (Average) | $O(n \log n)$   |
| Time (Worst)   | $O(n^2)$ (rare) |
| Space          | $O(\log n)$     |
| Stable         | No              |
| Adaptive       | No              |

Randomized Quick Sort shows the power of randomness, a tiny change transforms a fragile algorithm into a reliably fast one, making it one of the most practical sorts in modern computing.

## 117 Heap Sort

Heap Sort is a classic comparison-based, in-place,  $O(n \log n)$  sorting algorithm built upon the heap data structure. It first turns the array into a max-heap, then repeatedly removes the largest element (the heap root) and places it at the end, shrinking the heap as it goes. It's efficient and memory-friendly but not stable.

### What Problem Are We Solving??

We want a sorting algorithm that:

- Has guaranteed  $O(n \log n)$  time in all cases
- Uses constant extra space ( $O(1)$ )
- Doesn't require recursion or extra arrays like Merge Sort

Heap Sort achieves this by using a binary heap to always extract the largest element efficiently, then placing it in its correct position at the end.

Perfect for:

- Memory-constrained systems
- Predictable performance needs
- Offline sorting when data fits in RAM

### Example

| Step | Action                    | Array State     | Heap Size |
|------|---------------------------|-----------------|-----------|
| 0    | Build max-heap            | [5, 3, 4, 1, 2] | 5         |
| 1    | Swap root with last       | [2, 3, 4, 1, 5] | 4         |
| 2    | Heapify root              | [4, 3, 2, 1, 5] | 4         |
| 3    | Swap root with last       | [1, 3, 2, 4, 5] | 3         |
| 4    | Heapify root              | [3, 1, 2, 4, 5] | 3         |
| 5    | Repeat until heap shrinks | [1, 2, 3, 4, 5] | 0         |

### How Does It Work (Plain Language)?

Think of a heap like a tree stored in an array. The root (index 0) is the largest element. Heap Sort works in two main steps:

1. Build a max-heap (arranged so every parent > its children)

2. Extract max repeatedly:

- Swap root with last element
- Reduce heap size
- Heapify root to restore max-heap property

After each extraction, the sorted part grows at the end of the array.

### Step-by-Step Process

| Step | Description                                |
|------|--------------------------------------------|
| 1    | Build a max-heap from the array            |
| 2    | Swap the first (max) with the last element |
| 3    | Reduce heap size by one                    |
| 4    | Heapify the root                           |
| 5    | Repeat until heap is empty                 |

### Tiny Code (Easy Versions)

C

```
#include <stdio.h>

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void heapify(int a[], int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && a[left] > a[largest]) largest = left;
    if (right < n && a[right] > a[largest]) largest = right;

    if (largest != i) {
        swap(&a[i], &a[largest]);
        heapify(a, n, largest);
    }
}
```

```

    }

}

void heap_sort(int a[], int n) {
    // Build max heap
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(a, n, i);

    // Extract elements from heap
    for (int i = n - 1; i > 0; i--) {
        swap(&a[0], &a[i]);
        heapify(a, i, 0);
    }
}

int main(void) {
    int a[] = {5, 3, 4, 1, 2};
    int n = sizeof(a) / sizeof(a[0]);
    heap_sort(a, n);
    for (int i = 0; i < n; i++) printf("%d ", a[i]);
    printf("\n");
}

```

## Python

```

def heapify(a, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2
    if left < n and a[left] > a[largest]:
        largest = left
    if right < n and a[right] > a[largest]:
        largest = right
    if largest != i:
        a[i], a[largest] = a[largest], a[i]
        heapify(a, n, largest)

def heap_sort(a):
    n = len(a)
    # Build max heap
    for i in range(n // 2 - 1, -1, -1):

```

```

heapify(a, n, i)
# Extract max and heapify
for i in range(n - 1, 0, -1):
    a[0], a[i] = a[i], a[0]
    heapify(a, i, 0)

arr = [5, 3, 4, 1, 2]
heap_sort(arr)
print(arr)

```

## Why It Matters

- Predictable  $O(n \log n)$  in all cases
- In-place, no extra memory needed
- Excellent when memory is tight or recursion is not preferred
- Demonstrates tree-based sorting logic

## A Gentle Proof (Why It Works)

Building the heap:  $O(n)$  Extracting each element:  $O(\log n)$  Total time:

$$T(n) = O(n) + n \times O(\log n) = O(n \log n)$$

Each element is “bubbled down”  $\log n$  levels at most once.

| Phase                | Work                 | Total         |
|----------------------|----------------------|---------------|
| Build heap           | $O(n)$               | Linear        |
| Extract $n$ elements | $n \times O(\log n)$ | $O(n \log n)$ |

Not stable, because swapping can break equal-element order.

## Try It Yourself

1. Build max-heap from [5, 3, 4, 1, 2].
2. Draw heap tree for each step.
3. Trace heapify calls and swaps.
4. Implement min-heap version for descending sort.
5. Count comparisons per phase.

6. Compare with Merge Sort space usage.
7. Modify to stop early if already sorted.
8. Animate heap construction.
9. Test on reverse array [5,4,3,2,1].
10. Add debug prints showing heap after each step.

## Test Cases

| Input           | Output          | Notes                  |
|-----------------|-----------------|------------------------|
| [3, 2, 1]       | [1, 2, 3]       | Small test             |
| [1, 2, 3]       | [1, 2, 3]       | Already sorted         |
| [5, 3, 4, 1, 2] | [1, 2, 3, 4, 5] | General case           |
| [2, 2, 1, 1]    | [1, 1, 2, 2]    | Not stable but correct |

## Complexity

| Aspect         | Value         |
|----------------|---------------|
| Time (Best)    | $O(n \log n)$ |
| Time (Average) | $O(n \log n)$ |
| Time (Worst)   | $O(n \log n)$ |
| Space          | $O(1)$        |
| Stable         | No            |
| Adaptive       | No            |

Heap Sort is the workhorse of guaranteed performance, steady, space-efficient, and built on elegant tree logic. It never surprises you with bad cases, a reliable friend when consistency matters.

## 118 3-Way Quick Sort

3-Way Quick Sort is a refined version of Quick Sort designed to handle arrays with many duplicate elements efficiently. Instead of dividing the array into two parts (less than and greater than pivot), it divides into three regions:

- < pivot
- = pivot
- > pivot

This avoids redundant work when many elements are equal to the pivot, making it especially effective for datasets with low entropy or repeated keys.

### **What Problem Are We Solving??**

Standard Quick Sort can perform unnecessary work when duplicates are present. For example, if all elements are the same, standard Quick Sort still recurses  $O(n \log n)$  times.

3-Way Quick Sort fixes this by:

- Skipping equal elements during partitioning
- Shrinking recursion depth dramatically

It's ideal for:

- Arrays with many duplicates
- Sorting strings with common prefixes
- Key-value pairs with repeated keys

### **Example**

| Step | Pivot | Action                                           | Array State     | Partitions      |
|------|-------|--------------------------------------------------|-----------------|-----------------|
| 0    | 3     | Start                                            | [3, 2, 3, 1, 3] | $l=0, i=0, g=4$ |
| 1    | 3     | $a[i]=3 \rightarrow$ equal                       | [3, 2, 3, 1, 3] | $i=1$           |
| 2    | 3     | $a[i]=2 < 3 \rightarrow \text{swap}(a[l], a[i])$ | [2, 3, 3, 1, 3] | $l=1, i=2$      |
| 3    | 3     | $a[i]=3 \rightarrow$ equal                       | [2, 3, 3, 1, 3] | $i=3$           |
| 4    | 3     | $a[i]=1 < 3 \rightarrow \text{swap}(a[l], a[i])$ | [2, 1, 3, 3, 3] | $l=2, i=4$      |
| 5    | 3     | $a[i]=3 \rightarrow$ equal                       | [2, 1, 3, 3, 3] | Done            |

Now recursively sort left  $<$  pivot region [2,1], skip the middle =3 block, and sort right  $>$  pivot (empty).

### **How Does It Work (Plain Language)?**

We track three zones using three pointers:

- lt (less than region)
- i (current element)
- gt (greater than region)

Each iteration compares  $a[i]$  with the pivot:

- If  $i < pivot$ : swap with  $lt$ , expand both regions
- If  $i > pivot$ : swap with  $gt$ , shrink right region
- If  $i = pivot$ : move forward

Continue until  $i > gt$ . This single pass partitions array into three regions, no need to revisit equals.

### Step-by-Step Process

| Step | Condition          | Action                                      |
|------|--------------------|---------------------------------------------|
| 1    | $a[i] < pivot$     | $\text{swap}(a[i], a[lt])$ , $i++$ , $lt++$ |
| 2    | $a[i] > pivot$     | $\text{swap}(a[i], a[gt])$ , $gt--$         |
| 3    | $a[i] == pivot$    | $i++$                                       |
| 4    | Stop when $i > gt$ |                                             |

Then recursively sort  $[low..lt-1]$  and  $[gt+1..high]$ .

### Tiny Code (Easy Versions)

C

```
#include <stdio.h>

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void quicksort_3way(int a[], int low, int high) {
    if (low >= high) return;

    int pivot = a[low];
    int lt = low, i = low, gt = high;

    while (i <= gt) {
        if (a[i] < pivot) {
            swap(&a[lt], &a[i]);
            lt++; i++;
        } else if (a[i] > pivot) {
            swap(&a[i], &a[gt]);
            gt--;
        }
    }
}
```

```

        swap(&a[i], &a[gt]);
        gt--;
    } else {
        i++;
    }
}

quicksort_3way(a, low, lt - 1);
quicksort_3way(a, gt + 1, high);
}

int main(void) {
    int a[] = {3, 2, 3, 1, 3};
    int n = sizeof(a) / sizeof(a[0]);
    quicksort_3way(a, 0, n - 1);
    for (int i = 0; i < n; i++) printf("%d ", a[i]);
    printf("\n");
}

```

## Python

```

def quicksort_3way(a, low, high):
    if low >= high:
        return
    pivot = a[low]
    lt, i, gt = low, low, high
    while i <= gt:
        if a[i] < pivot:
            a[lt], a[i] = a[i], a[lt]
            lt += 1
            i += 1
        elif a[i] > pivot:
            a[i], a[gt] = a[gt], a[i]
            gt -= 1
        else:
            i += 1
    quicksort_3way(a, low, lt - 1)
    quicksort_3way(a, gt + 1, high)

arr = [3, 2, 3, 1, 3]
quicksort_3way(arr, 0, len(arr) - 1)

```

```
print(arr)
```

## Why It Matters

- Efficient for arrays with duplicates
- Reduces unnecessary recursion and comparisons
- Used in string sorting and key-heavy data
- Generalizes the idea of “partition” to multi-way splitting

## A Gentle Proof (Why It Works)

Standard Quick Sort always divides into two regions, even if all elements equal the pivot, leading to  $O(n^2)$  on identical elements.

3-Way Quick Sort partitions into three zones:

- $<$  pivot (left)
- $=$  pivot (middle)
- $>$  pivot (right)

The middle zone is skipped from recursion, reducing work dramatically.

If all elements are equal  $\rightarrow$  only one pass  $O(n)$ .

Expected complexity:

$$T(n) = O(n \log n)$$

Worst-case (no duplicates): same as Quick Sort.

| Case      | Duplicates    | Complexity      |
|-----------|---------------|-----------------|
| All equal | $O(n)$        | One pass only   |
| Many      | $O(n \log n)$ | Efficient       |
| None      | $O(n \log n)$ | Normal behavior |

## Try It Yourself

1. Sort [3, 2, 3, 1, 3] step by step.
2. Print regions (`lt`, `i`, `gt`) after each iteration.
3. Compare recursion depth with normal Quick Sort.
4. Test input [1, 1, 1, 1].
5. Test input [5, 4, 3, 2, 1].
6. Sort ["apple", "apple", "banana", "apple"].

7. Visualize partitions on paper.
8. Modify to count swaps and comparisons.
9. Implement descending order.
10. Apply to random integers with duplicates (e.g. [1,2,2,2,3,3,1]).

### Test Cases

| Input           | Output          | Notes                |
|-----------------|-----------------|----------------------|
| [3, 2, 3, 1, 3] | [1, 2, 3, 3, 3] | Duplicates           |
| [5, 4, 3, 2, 1] | [1, 2, 3, 4, 5] | No duplicates        |
| [2, 2, 2, 2]    | [2, 2, 2, 2]    | All equal ( $O(n)$ ) |
| [1, 3, 1, 3, 1] | [1, 1, 1, 3, 3] | Clustered duplicates |

### Complexity

| Aspect         | Value                                |
|----------------|--------------------------------------|
| Time (Best)    | $O(n)$ (all equal)                   |
| Time (Average) | $O(n \log n)$                        |
| Time (Worst)   | $O(n \log n)$                        |
| Space          | $O(\log n)$ recursion                |
| Stable         | No                                   |
| Adaptive       | Yes (handles duplicates efficiently) |

3-Way Quick Sort shows how a small change, three-way partitioning, can transform Quick Sort into a powerful tool for duplicate-heavy datasets, blending elegance with efficiency.

## 119 External Merge Sort

External Merge Sort is a specialized sorting algorithm designed for very large datasets that don't fit entirely into main memory (RAM). It works by sorting chunks of data in memory, writing them to disk, and then merging those sorted chunks. This makes it a key tool in databases, file systems, and big data processing.

## **What Problem Are We Solving??**

When data exceeds RAM capacity, in-memory sorts like Quick Sort or Heap Sort fail, they need random access to all elements. External Merge Sort solves this by processing data in blocks:

- Sort manageable chunks in memory
- Write sorted chunks (“runs”) to disk
- Merge runs sequentially using streaming I/O

This minimizes disk reads/writes, the main bottleneck in large-scale sorting.

It's ideal for:

- Large files (GBs to TBs)
- Database query engines
- Batch processing pipelines

## **Example**

Let's say you have 1 GB of data and only 100 MB of RAM.

| Step | Action | Description                                        |
|------|--------|----------------------------------------------------|
| 1    | Split  | Divide file into 10 chunks of 100 MB               |
| 2    | Sort   | Load each chunk in memory, sort, write to disk     |
| 3    | Merge  | Use k-way merge (e.g. 10-way) to merge sorted runs |
| 4    | Output | Final sorted file written sequentially             |

## **How Does It Work (Plain Language)?**

Think of it like sorting pages of a giant book:

1. Take a few pages at a time (fit in memory)
2. Sort them and place them in order piles
3. Combine the piles in order until the whole book is sorted

It's a multi-pass algorithm:

- Pass 1: Create sorted runs
- Pass 2+: Merge runs in multiple passes until one remains

## Step-by-Step Process

| Step | Description                                      |
|------|--------------------------------------------------|
| 1    | Divide the large file into blocks fitting memory |
| 2    | Load a block, sort it using in-memory sort       |
| 3    | Write each sorted block (run) to disk            |
| 4    | Merge all runs using k-way merging               |
| 5    | Repeat merges until a single sorted file remains |

## Tiny Code (Simplified Simulation)

### Python (Simulated External Sort)

```
import heapq
import tempfile

def sort_chunk(chunk):
    chunk.sort()
    temp = tempfile.TemporaryFile(mode="w+t")
    temp.writelines(f"{x}\n" for x in chunk)
    temp.seek(0)
    return temp

def merge_files(files, output_file):
    iters = [map(int, f) for f in files]
    with open(output_file, "w") as out:
        for num in heapq.merge(*iters):
            out.write(f"{num}\n")

def external_merge_sort(input_data, chunk_size=5):
    chunks = []
    for i in range(0, len(input_data), chunk_size):
        chunk = input_data[i:i + chunk_size]
        chunks.append(sort_chunk(chunk))
    merge_files(chunks, "sorted_output.txt")

data = [42, 17, 93, 8, 23, 4, 16, 99, 55, 12, 71, 3]
external_merge_sort(data, chunk_size=4)
```

This example simulates external sorting in Python, splitting input into chunks, sorting each, and merging with `heapq.merge`.

## Why It Matters

- Handles massive datasets beyond memory limits
- Sequential disk I/O (fast and predictable)
- Foundation of database sort-merge joins
- Works well with distributed systems (MapReduce, Spark)

## A Gentle Proof (Why It Works)

Each pass performs  $O(n)$  work to read and write the entire dataset. If  $r$  is the number of runs, and  $k$  is merge fan-in (number of runs merged at once):

$$\text{Number of passes} = \lceil \log_k r \rceil$$

Total cost

$$O(n \log_k r)$$

dominated by I/O operations rather than comparisons.

For  $r = n/M$  (chunks of memory size  $M$ ), performance is optimized by choosing  $k = M$ .

| Phase       | Work            | Cost         |
|-------------|-----------------|--------------|
| Create Runs | $O(n \log M)$   | Sort chunks  |
| Merge Runs  | $O(n \log_k r)$ | Merge passes |

## Try It Yourself

1. Split [42, 17, 93, 8, 23, 4, 16, 99, 55, 12, 71, 3] into 4-element chunks.
2. Sort each chunk individually.
3. Simulate merging sorted runs.
4. Try merging 2-way vs 4-way, count passes.
5. Visualize merging tree (runs combining).
6. Test with random large arrays (simulate files).
7. Modify chunk size and observe performance.
8. Compare I/O counts with in-memory sort.
9. Use `heapq.merge` to merge sorted streams.
10. Extend to merge files on disk (not just lists).

## Test Cases

| Input                                        | Memory Limit                  | Output                                            | Notes                                               |
|----------------------------------------------|-------------------------------|---------------------------------------------------|-----------------------------------------------------|
| [9, 4, 7, 2, 5, 1, 8, 3, 6]<br>1 GB integers | 3 elements<br>100 MB<br>small | [1,2,3,4,5,6,7,8,9]<br>Sorted file<br>[1,1,1,1,1] | 3-way merge<br>10 sorted runs<br>Handles duplicates |
| [1,1,1,1,1]                                  |                               |                                                   |                                                     |

## Complexity

| Aspect     | Value                                     |
|------------|-------------------------------------------|
| Time       | $O(n \log_k (n/M))$                       |
| Space      | $O(M)$ (memory buffer)                    |
| I/O Passes | $O(\log_k (n/M))$                         |
| Stable     | Yes                                       |
| Adaptive   | Yes (fewer runs if data partially sorted) |

External Merge Sort is the unsung hero behind large-scale sorting, when memory ends, it steps in with disk-based precision, keeping order across terabytes with calm efficiency.

## 120 Parallel Merge Sort

Parallel Merge Sort takes the familiar divide-and-conquer structure of Merge Sort and spreads the work across multiple threads or processors, achieving faster sorting on multi-core CPUs or distributed systems. It's an ideal illustration of how parallelism can amplify a classic algorithm without changing its logic.

### What Problem Are We Solving??

Traditional Merge Sort runs sequentially, so even though its complexity is  $O(n \log n)$ , it uses only one CPU core. On modern hardware with many cores, that's a waste.

Parallel Merge Sort tackles this by:

- Sorting subarrays in parallel
- Merging results concurrently
- Utilizing full CPU or cluster potential

It's essential for:

- High-performance computing
- Large-scale sorting
- Real-time analytics pipelines

## **Example**

Sort [5, 3, 4, 1, 2] using 2 threads:

| Step | Action                      | Threads                          | Result            |
|------|-----------------------------|----------------------------------|-------------------|
| 1    | Split array into halves     | 2 threads                        | [5, 3, 4], [1, 2] |
| 2    | Sort each half concurrently | T1: sort [5,3,4], T2: sort [1,2] | [3,4,5], [1,2]    |
| 3    | Merge results               | 1 thread                         | [1,2,3,4,5]       |

Parallelism reduces total time roughly by 1 / number of threads (with overhead).

## **How Does It Work (Plain Language)?**

It's still divide and conquer, just with teamwork:

1. Split array into two halves.
2. Sort each half in parallel.
3. Merge the two sorted halves.
4. Stop splitting when subarrays are small (then sort sequentially).

Each recursive level can launch new threads until you reach a threshold or maximum depth.

## **Step-by-Step Process**

| Step | Description                     |
|------|---------------------------------|
| 1    | Divide the array into halves    |
| 2    | Sort both halves concurrently   |
| 3    | Wait for both to finish         |
| 4    | Merge results sequentially      |
| 5    | Repeat recursively for subparts |

This pattern fits well with thread pools, task schedulers, or fork-join frameworks.

## Tiny Code (Easy Versions)

### C (POSIX Threads Example)

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

typedef struct {
    int *arr;
    int left;
    int right;
} Args;

void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1, n2 = r - m;
    int L[n1], R[n2];
    for (int i = 0; i < n1; i++) L[i] = arr[l + i];
    for (int j = 0; j < n2; j++) R[j] = arr[m + 1 + j];

    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];
    }
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}

void *parallel_merge_sort(void *arg) {
    Args *args = (Args *)arg;
    int l = args->left, r = args->right;
    int *arr = args->arr;

    if (l < r) {
        int m = l + (r - l) / 2;

        Args leftArgs = {arr, l, m};
        Args rightArgs = {arr, m + 1, r};
        pthread_t leftThread, rightThread;

        pthread_create(&leftThread, NULL, parallel_merge_sort, &leftArgs);
        pthread_create(&rightThread, NULL, parallel_merge_sort, &rightArgs);
    }
}
```

```

pthread_create(&rightThread, NULL, parallel_merge_sort, &rightArgs);

pthread_join(leftThread, NULL);
pthread_join(rightThread, NULL);

merge(arr, l, m, r);
}

return NULL;
}

int main(void) {
    int arr[] = {5, 3, 4, 1, 2};
    int n = sizeof(arr) / sizeof(arr[0]);
    Args args = {arr, 0, n - 1};
    parallel_merge_sort(&args);
    for (int i = 0; i < n; i++) printf("%d ", arr[i]);
    printf("\n");
}

```

*(Note: This simple version may create too many threads; real implementations limit thread depth.)*

## Python (Using multiprocessing)

```

from multiprocessing import Pool

def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result

```

```

def parallel_merge_sort(arr):
    if len(arr) <= 1:
        return arr
    if len(arr) < 1000: # threshold
        return sorted(arr)
    mid = len(arr) // 2
    with Pool(2) as p:
        left, right = p.map(parallel_merge_sort, [arr[:mid], arr[mid:]])
    return merge(left, right)

arr = [5, 3, 4, 1, 2]
print(parallel_merge_sort(arr))

```

## Why It Matters

- Exploits multi-core architectures
- Significantly reduces wall-clock time
- Maintains  $O(n \log n)$  work
- Great showcase of parallel divide-and-conquer

Used in:

- HPC (High Performance Computing)
- Modern standard libraries (`std::execution::par`)
- Big data frameworks (Spark, Hadoop)

## A Gentle Proof (Why It Works)

Each recursive call sorts  $n/2$  elements, but now in parallel. Let  $P$  = number of processors.

Work (total operations):

$$T_{work}(n) = O(n \log n)$$

Span (critical path time):

$$T_{span}(n) = O(\log^2 n)$$

Total time

$$O\left(\frac{n \log n}{P} + \log^2 n\right)$$

Speedup  $P \times$ , limited by synchronization and merge overhead.

| Phase          | Parallelizable | Work              |
|----------------|----------------|-------------------|
| Sort subarrays | Yes            | $O(n \log n / P)$ |
| Merge          | Partially      | $O(n \log P)$     |

## Try It Yourself

1. Run with 1, 2, 4, 8 threads, compare speed.
2. Print thread IDs at each recursive call.
3. Implement threshold for small subarrays.
4. Merge using parallel merging.
5. Measure CPU utilization during sort.
6. Test with large random list (10 elements).
7. Compare with sequential Merge Sort.
8. Profile with timing tools.
9. Try OpenMP version in C.
10. Extend to distributed nodes (MPI).

## Test Cases

| Input           | Threads | Output      | Notes           |
|-----------------|---------|-------------|-----------------|
| [3,2,1]         | 2       | [1,2,3]     | Simple          |
| [5,3,4,1,2]     | 2       | [1,2,3,4,5] | Balanced work   |
| 1e6 random ints | 8       | sorted      | Parallel boost  |
| [1,1,1,1]       | 4       | [1,1,1,1]   | Stable behavior |

## Complexity

| Aspect        | Value                        |
|---------------|------------------------------|
| Work          | $O(n \log n)$                |
| Span          | $O(\log^2 n)$                |
| Parallel Time | $O(n \log n / P + \log^2 n)$ |
| Space         | $O(n)$                       |
| Stable        | Yes                          |
| Adaptive      | No                           |

Parallel Merge Sort is Merge Sort reborn for the multi-core era, the same elegance, now with teamwork. It's how classic algorithms learn to scale with hardware.

## Section 13. Counting and distribution sorts

### 121 Counting Sort

Counting Sort is a non-comparison sorting algorithm that sorts integers (or items mapped to integer keys) by counting occurrences of each value. Instead of comparing elements, it directly uses their values as indices in a counting array. It's fast ( $O(n + k)$ ), stable, and perfect when the input range is limited and small.

#### What Problem Are We Solving??

When keys are integers within a known range, comparison-based sorts ( $O(n \log n)$ ) are overkill. Counting Sort leverages that limited range to sort in linear time, without any comparisons.

Perfect for:

- Sorting grades (0–100)
- Sorting digits or ASCII codes (0–255)
- Pre-step for Radix Sort or Bucket Sort
- Scenarios where key range  $n^2$

#### Example

Sort array [4, 2, 2, 8, 3, 3, 1]

| Step | Description                               | Result                      |
|------|-------------------------------------------|-----------------------------|
| 1    | Find max = 8                              | Range = 0–8                 |
| 2    | Initialize count[9] = [0,0,0,0,0,0,0,0,0] |                             |
| 3    | Count each number                         | count = [0,1,2,2,1,0,0,0,1] |
| 4    | Prefix sum (positions)                    | count = [0,1,3,5,6,6,6,6,7] |
| 5    | Place elements by count                   | [1,2,2,3,3,4,8]             |

The count array tracks the position boundaries for each key.

## How Does It Work (Plain Language)?

Counting Sort doesn't compare elements. It counts how many times each value appears, then uses those counts to reconstruct the sorted list.

Think of it as filling labeled bins:

- One bin for each number
- Drop each element into its bin
- Then walk through bins in order and empty them

## Step-by-Step Process

| Step | Description                                                 |
|------|-------------------------------------------------------------|
| 1    | Find min and max (determine range $k$ )                     |
| 2    | Create count array of size $k + 1$                          |
| 3    | Count occurrences of each value                             |
| 4    | Transform counts into prefix sums (for positions)           |
| 5    | Traverse input in reverse (for stability), placing elements |
| 6    | Copy sorted output back                                     |

## Tiny Code (Easy Versions)

C

```
#include <stdio.h>
#include <string.h>

void counting_sort(int arr[], int n) {
    int max = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > max) max = arr[i];

    int count[max + 1];
    memset(count, 0, sizeof(count));

    for (int i = 0; i < n; i++)
        count[arr[i]]++;

    for (int i = 1; i <= max; i++)
        count[i] += count[i - 1];
```

```

int output[n];
for (int i = n - 1; i >= 0; i--) {
    output[count[arr[i]] - 1] = arr[i];
    count[arr[i]]--;
}

for (int i = 0; i < n; i++)
    arr[i] = output[i];
}

int main(void) {
    int arr[] = {4, 2, 2, 8, 3, 3, 1};
    int n = sizeof(arr) / sizeof(arr[0]);
    counting_sort(arr, n);
    for (int i = 0; i < n; i++) printf("%d ", arr[i]);
    printf("\n");
}

```

## Python

```

def counting_sort(arr):
    max_val = max(arr)
    count = [0] * (max_val + 1)

    for num in arr:
        count[num] += 1

    for i in range(1, len(count)):
        count[i] += count[i - 1]

    output = [0] * len(arr)
    for num in reversed(arr):
        count[num] -= 1
        output[count[num]] = num

    return output

arr = [4, 2, 2, 8, 3, 3, 1]
print(counting_sort(arr))

```

## Why It Matters

- Linear time when range is small ( $O(n + k)$ )
- Stable, preserving input order
- Foundation for Radix Sort
- Great for integer, digit, or bucket sorting

## A Gentle Proof (Why It Works)

Counting Sort replaces comparison by index-based placement.

If  $n$  is number of elements and  $k$  is key range:

- Counting occurrences:  $O(n)$
- Prefix sums:  $O(k)$
- Placement:  $O(n)$

Total =  $O(n + k)$

Stable because we traverse input in reverse while placing.

| Phase          | Work   | Complexity   |
|----------------|--------|--------------|
| Count elements | $O(n)$ | Scan once    |
| Prefix sum     | $O(k)$ | Range pass   |
| Place elements | $O(n)$ | Stable write |

## Try It Yourself

1. Sort [4, 2, 2, 8, 3, 3, 1] step by step.
2. Show count array after counting.
3. Convert count to prefix sums.
4. Place elements in output (reverse scan).
5. Compare stable vs unstable version.
6. Change input to [9, 9, 1, 2].
7. Try sorting [5, 3, 5, 1, 0].
8. Handle input with min > 0 (offset counts).
9. Measure runtime vs Bubble Sort.
10. Use as subroutine in Radix Sort.

## Test Cases

| Input           | Output          | Notes          |
|-----------------|-----------------|----------------|
| [4,2,2,8,3,3,1] | [1,2,2,3,3,4,8] | Example        |
| [1,4,1,2,7,5,2] | [1,1,2,2,4,5,7] | Stable         |
| [9,9,9,9]       | [9,9,9,9]       | Repeats        |
| [0,1,2,3]       | [0,1,2,3]       | Already sorted |

## Complexity

| Aspect          | Value      |
|-----------------|------------|
| Time            | $O(n + k)$ |
| Space           | $O(n + k)$ |
| Stable          | Yes        |
| Adaptive        | No         |
| Range-sensitive | Yes        |

Counting Sort is like sorting by bins, no comparisons, no stress, just clean counts and linear time. It's a powerhouse behind Radix Sort and data bucketing in performance-critical pipelines.

## 122 Stable Counting Sort

Stable Counting Sort refines the basic Counting Sort by ensuring equal elements preserve their original order. This property, called *stability*, is crucial when sorting multi-key data, for example, sorting people by age, then by name. Stable versions are also the building blocks for Radix Sort, where each digit's sort depends on stability.

### What Problem Are We Solving??

Basic Counting Sort can break order among equal elements because it places them in arbitrary order. When sorting records or tuples where order matters (e.g., by secondary key), we need stability, if **a** and **b** have equal keys, their order in output must match input.

Stable Counting Sort ensures that:

If **arr[i]** and **arr[j]** have the same key and  $i < j$ , then **arr[i]** appears before **arr[j]** in the sorted output.

Perfect for:

- Radix Sort digits

- Multi-field records (e.g. sort by name, then by score)
- Databases and stable pipelines

### Example

Sort [4a, 2b, 2a, 8a, 3b, 3a, 1a] (letters mark order)

| Step | Description               | Result                                |
|------|---------------------------|---------------------------------------|
| 1    | Count frequencies         | count = [0,1,2,2,1,0,0,0,1]           |
| 2    | Prefix sums (positions)   | count = [0,1,3,5,6,6,6,6,7]           |
| 3    | Traverse input in reverse | output = [1a, 2b, 2a, 3b, 3a, 4a, 8a] |

See how 2b (index 1) appears before 2a (index 2), stable ordering preserved.

### How Does It Work (Plain Language)?

It's Counting Sort with a twist: we fill the output from the end of the input, ensuring last-seen equal items go last. By traversing in reverse, earlier elements are placed later, preserving their original order.

This is the core idea behind stable sorting.

### Step-by-Step Process

| Step | Description                                   |
|------|-----------------------------------------------|
| 1    | Determine key range (0..max)                  |
| 2    | Count frequency of each key                   |
| 3    | Compute prefix sums (to determine positions)  |
| 4    | Traverse input right to left                  |
| 5    | Place elements in output using count as index |
| 6    | Decrement count[key] after placement          |

### Tiny Code (Easy Versions)

C

```

#include <stdio.h>
#include <string.h>

typedef struct {
    int key;
    char tag; // to visualize stability
} Item;

void stable_counting_sort(Item arr[], int n) {
    int max = arr[0].key;
    for (int i = 1; i < n; i++)
        if (arr[i].key > max) max = arr[i].key;

    int count[max + 1];
    memset(count, 0, sizeof(count));

    // Count occurrences
    for (int i = 0; i < n; i++)
        count[arr[i].key]++;

    // Prefix sums
    for (int i = 1; i <= max; i++)
        count[i] += count[i - 1];

    Item output[n];

    // Traverse input in reverse for stability
    for (int i = n - 1; i >= 0; i--) {
        int k = arr[i].key;
        output[count[k] - 1] = arr[i];
        count[k]--;
    }

    for (int i = 0; i < n; i++)
        arr[i] = output[i];
}

int main(void) {
    Item arr[] = {{4, 'a'}, {2, 'b'}, {2, 'a'}, {8, 'a'}, {3, 'b'}, {3, 'a'}, {1, 'a'}};
    int n = sizeof(arr)/sizeof(arr[0]);
    stable_counting_sort(arr, n);
    for (int i = 0; i < n; i++) printf("(%.d,%.c) ", arr[i].key, arr[i].tag);
}

```

```
    printf("\n");
}
```

## Python

```
def stable_counting_sort(arr):
    max_val = max(arr)
    count = [0] * (max_val + 1)

    for num in arr:
        count[num] += 1

    for i in range(1, len(count)):
        count[i] += count[i - 1]

    output = [0] * len(arr)
    for num in reversed(arr):
        count[num] -= 1
        output[count[num]] = num

    return output

arr = [4, 2, 2, 8, 3, 3, 1]
print(stable_counting_sort(arr))
```

## Why It Matters

- Stable sorting is essential for multi-key operations
- Required for Radix Sort correctness
- Guarantees consistent behavior for duplicates
- Used in databases, language sort libraries, pipelines

## A Gentle Proof (Why It Works)

Each key is assigned a position range via prefix sums. Traversing input from right to left ensures that earlier items occupy smaller indices, preserving order.

If  $a$  appears before  $b$  in input and  $\text{key}(a) = \text{key}(b)$ , then  $\text{count}[\text{key}(a)]$  places  $a$  before  $b$ , stable.

| Phase       | Work   | Complexity        |
|-------------|--------|-------------------|
| Counting    | $O(n)$ | Pass once         |
| Prefix sums | $O(k)$ | Range pass        |
| Placement   | $O(n)$ | Reverse traversal |

Total =  $O(n + k)$ , same as basic Counting Sort, but stable.

### Try It Yourself

1. Sort  $[(4, 'a'), (2, 'b'), (2, 'a'), (3, 'a'))]$ .
2. Show count and prefix arrays.
3. Traverse input from end, track output.
4. Compare with unstable version.
5. Try sorting  $[5, 3, 5, 1, 0]$ .
6. Visualize stability when equal keys appear.
7. Modify to handle offset keys (negative values).
8. Combine with Radix Sort (LSD).
9. Profile runtime vs normal Counting Sort.
10. Check stability by adding tags (letters).

### Test Cases

| Input                   | Output                  | Notes            |
|-------------------------|-------------------------|------------------|
| $[4, 2, 2, 8, 3, 3, 1]$ | $[1, 2, 2, 3, 3, 4, 8]$ | Same as basic    |
| $[(2, 'a'), (2, 'b')]$  | $[(2, 'a'), (2, 'b')]$  | Stable preserved |
| $[1, 1, 1]$             | $[1, 1, 1]$             | Idempotent       |
| $[0, 1, 2, 3]$          | $[0, 1, 2, 3]$          | Already sorted   |

### Complexity

| Aspect          | Value      |
|-----------------|------------|
| Time            | $O(n + k)$ |
| Space           | $O(n + k)$ |
| Stable          | Yes        |
| Adaptive        | No         |
| Range-sensitive | Yes        |

Stable Counting Sort is Counting Sort with memory, it not only sorts fast but also remembers your order, making it indispensable for multi-pass algorithms like Radix Sort.

## 123 Radix Sort (LSD)

Radix Sort (Least Significant Digit first) is a non-comparison, stable sorting algorithm that processes integers (or strings) digit by digit, starting from the least significant digit (LSD). By repeatedly applying a stable sort (like Counting Sort) on each digit, it can sort numbers in linear time when digit count is small.

### What Problem Are We Solving??

When sorting integers or fixed-length keys (like dates, IDs, or strings of digits), traditional comparison-based sorts spend unnecessary effort. Radix Sort (LSD) sidesteps comparisons by leveraging digit-wise order and stability to achieve  $O(d \times (n + k))$  performance.

Perfect for:

- Sorting numbers, dates, zip codes, or strings
- Datasets with bounded digit length
- Applications where deterministic performance matters

### Example

Sort [170, 45, 75, 90, 802, 24, 2, 66]

| Pass | Digit Place | Input                      | Output (Stable Sort)       |
|------|-------------|----------------------------|----------------------------|
| 1    | Ones        | [170,45,75,90,802,24,2,66] | [170,90,802,2,24,45,75,66] |
| 2    | Tens        | [170,90,802,2,24,45,75,66] | [802,2,24,45,66,170,75,90] |
| 3    | Hundreds    | [802,2,24,45,66,170,75,90] | [2,24,45,66,75,90,170,802] |

Final sorted output: [2, 24, 45, 66, 75, 90, 170, 802]

Each pass uses Stable Counting Sort on the current digit.

## How Does It Work (Plain Language)?

Think of sorting by digit positions:

1. Group by ones place (units)
2. Group by tens
3. Group by hundreds, etc.

Each pass reorders elements according to the digit at that place, while keeping earlier digit orders intact (thanks to stability).

It's like sorting by last name, then first name, one field at a time, stable each round.

## Step-by-Step Process

| Step | Description                                            |
|------|--------------------------------------------------------|
| 1    | Find the maximum number to know the number of digits d |
| 2    | For each digit place (1, 10, 100, ...):                |
| 3    | Use a stable counting sort based on that digit         |
| 4    | After the last pass, the array is fully sorted         |

## Tiny Code (Easy Versions)

C

```
#include <stdio.h>

int get_max(int a[], int n) {
    int max = a[0];
    for (int i = 1; i < n; i++)
        if (a[i] > max) max = a[i];
    return max;
}

void counting_sort_digit(int a[], int n, int exp) {
    int output[n];
    int count[10] = {0};

    for (int i = 0; i < n; i++)
        count[(a[i] / exp) % 10]++;
}
```

```

for (int i = 1; i < 10; i++)
    count[i] += count[i - 1];

for (int i = n - 1; i >= 0; i--) {
    int digit = (a[i] / exp) % 10;
    output[count[digit] - 1] = a[i];
    count[digit]--;
}

for (int i = 0; i < n; i++)
    a[i] = output[i];
}

void radix_sort(int a[], int n) {
    int max = get_max(a, n);
    for (int exp = 1; max / exp > 0; exp *= 10)
        counting_sort_digit(a, n, exp);
}

int main(void) {
    int a[] = {170, 45, 75, 90, 802, 24, 2, 66};
    int n = sizeof(a) / sizeof(a[0]);
    radix_sort(a, n);
    for (int i = 0; i < n; i++) printf("%d ", a[i]);
    printf("\n");
}

```

## Python

```

def counting_sort_digit(arr, exp):
    n = len(arr)
    output = [0] * n
    count = [0] * 10

    for num in arr:
        index = (num // exp) % 10
        count[index] += 1

    for i in range(1, 10):
        count[i] += count[i - 1]

```

```

for num in reversed(arr):
    index = (num // exp) % 10
    count[index] -= 1
    output[count[index]] = num

for i in range(n):
    arr[i] = output[i]

def radix_sort(arr):
    max_val = max(arr)
    exp = 1
    while max_val // exp > 0:
        counting_sort_digit(arr, exp)
        exp *= 10

arr = [170, 45, 75, 90, 802, 24, 2, 66]
radix_sort(arr)
print(arr)

```

## Why It Matters

- Linear time ( $O(d \times (n + k))$ ) for fixed digits
- Stable, retains order for equal keys
- Great for large numeric datasets
- Foundation for efficient key-based sorting (strings, dates)

## A Gentle Proof (Why It Works)

At each digit position:

- Stable Counting Sort reorders by that digit
- Earlier digits remain ordered (stability)
- After all digits, array is fully ordered

If each digit has range  $k$  and  $d$  total digits:

$$T(n) = O(d \times (n + k))$$

| Phase      | Work                | Complexity    |
|------------|---------------------|---------------|
| Per digit  | $O(n + k)$          | Counting sort |
| All digits | $d \times O(n + k)$ | Total         |

If  $d$  and  $k$  are constants  $\rightarrow O(n)$  overall.

### Try It Yourself

1. Sort [170, 45, 75, 90, 802, 24, 2, 66].
2. Trace each pass (ones, tens, hundreds).
3. Show count table per digit.
4. Compare stable vs unstable sorting.
5. Add zeros: [07, 70, 700].
6. Try [3, 1, 2, 10, 11, 21].
7. Count digit comparisons.
8. Modify to handle negative numbers (offset).
9. Change base to 16 (hex).
10. Compare with Merge Sort performance on large input.

### Test Cases

| Input                      | Output                     | Notes             |
|----------------------------|----------------------------|-------------------|
| [170,45,75,90,802,24,2,66] | [2,24,45,66,75,90,170,802] | Classic           |
| [9,8,7,6,5]                | [5,6,7,8,9]                | Reversed          |
| [10,1,100,1000]            | [1,10,100,1000]            | Different lengths |
| [22,22,11,11]              | [11,11,22,22]              | Stable            |

### Complexity

| Aspect          | Value                 |
|-----------------|-----------------------|
| Time            | $O(d \times (n + k))$ |
| Space           | $O(n + k)$            |
| Stable          | Yes                   |
| Adaptive        | No                    |
| Range-sensitive | Yes                   |

Radix Sort (LSD) is the assembly line of sorting, each pass builds upon the last, producing perfectly ordered output from simple stable steps.

## 124 Radix Sort (MSD)

Radix Sort (Most Significant Digit first) is a recursive variant of Radix Sort that begins sorting from the most significant digit (MSD) and works downward. Unlike LSD Radix Sort, which is iterative and stable across all digits, MSD focuses on prefix-based grouping and recursively sorts subgroups. This makes it ideal for variable-length keys such as strings, IP addresses, or long integers.

### What Problem Are We Solving??

LSD Radix Sort works best for fixed-length keys, where every element has the same number of digits. But when keys differ in length (e.g., strings “a”, “ab”, “abc”), we need to respect prefix order, “a” should come before “ab”.

MSD Radix Sort handles this by grouping by prefix digits, then recursively sorting each group.

Perfect for:

- Strings, words, or variable-length keys
- Hierarchical data (prefix-sensitive)
- Lexicographic ordering (dictionary order)

### Example

Sort: `["b", "ba", "abc", "ab", "ac"]`

| Step | Digit       | Groups                                                              |
|------|-------------|---------------------------------------------------------------------|
| 1    | 1st char    | a → <code>["abc", "ab", "ac"]</code> , b → <code>["b", "ba"]</code> |
| 2    | Group “a”   | 2nd char → b: <code>["bc", "ab"]</code> , c: <code>["ac"]</code>    |
| 3    | Group “ab”  | 3rd char → c: <code>["bc"]</code> , end: <code>["ab"]</code>        |
| 4    | Final merge | <code>["ab", "bc", "ac", "b", "ba"]</code>                          |

Lexicographic order preserved, even with varying lengths.

## How Does It Work (Plain Language)?

MSD Radix Sort organizes data by prefix trees (tries) conceptually:

- Partition elements by their most significant digit (or character)
- Recurse within each group for next digit
- Merge groups in order of digit values

If LSD is like bucket sorting digits from the back, MSD is tree-like sorting from the top.

## Step-by-Step Process

| Step | Description                                   |
|------|-----------------------------------------------|
| 1    | Find highest digit place (or first character) |
| 2    | Partition array into groups by that digit     |
| 3    | Recursively sort each group by next digit     |
| 4    | Concatenate groups in order                   |

For strings, if one string ends early, it's considered smaller.

## Tiny Code (Easy Versions)

### Python (String Example)

```
def msd_radix_sort(arr, pos=0):
    if len(arr) <= 1:
        return arr

    # Buckets for ASCII range (0-255) + 1 for end-of-string
    buckets = [[] for _ in range(257)]
    for word in arr:
        index = ord(word[pos]) + 1 if pos < len(word) else 0
        buckets[index].append(word)

    result = []
    for bucket in buckets:
        if bucket:
            # Only recurse if there's more than one element and not EOS
            if len(bucket) > 1 and (pos < max(len(w) for w in bucket)):
```

```

        bucket = msd_radix_sort(bucket, pos + 1)
        result.extend(bucket)
    return result

arr = ["b", "ba", "abc", "ab", "ac"]
print(msd_radix_sort(arr))

```

Output:

```
$$'ab', 'abc', 'ac', 'b', 'ba']
```

## C (Numeric Example)

```

#include <stdio.h>

int get_digit(int num, int exp, int base) {
    return (num / exp) % base;
}

void msd_radix_sort_rec(int arr[], int n, int exp, int base, int max) {
    if (exp == 0 || n <= 1) return;

    int buckets[base][n];
    int count[base];
    for (int i = 0; i < base; i++) count[i] = 0;

    // Distribute
    for (int i = 0; i < n; i++) {
        int d = get_digit(arr[i], exp, base);
        buckets[d][count[d]++] = arr[i];
    }

    // Recurse and collect
    int idx = 0;
    for (int i = 0; i < base; i++) {
        if (count[i] > 0) {
            msd_radix_sort_rec(buckets[i], count[i], exp / base, base, max);
            for (int j = 0; j < count[i]; j++)
                arr[idx++] = buckets[i][j];
        }
    }
}

```

```

    }

}

void msd_radix_sort(int arr[], int n) {
    int max = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > max) max = arr[i];

    int exp = 1;
    while (max / exp >= 10) exp *= 10;

    msd_radix_sort_rec(arr, n, exp, 10, max);
}

int main(void) {
    int arr[] = {170, 45, 75, 90, 802, 24, 2, 66};
    int n = sizeof(arr) / sizeof(arr[0]);
    msd_radix_sort(arr, n);
    for (int i = 0; i < n; i++) printf("%d ", arr[i]);
    printf("\n");
}

```

## Why It Matters

- Handles variable-length keys
- Natural for lexicographic ordering
- Used in string sorting, trie-based systems, suffix array construction
- Recursively partitions, often faster for large diverse keys

## A Gentle Proof (Why It Works)

Each recursive call partitions the array by digit prefix. Since partitions are disjoint and ordered by digit, concatenating them yields a fully sorted sequence.

For  $n$  elements,  $d$  digits, and base  $k$ :

$$\begin{aligned}
 T(n) &= O(n + k) \text{ per level, depth } \leq d \\
 &\Rightarrow O(d \times (n + k))
 \end{aligned}$$

Stability preserved via ordered grouping.

| Phase     | Work          | Description                    |
|-----------|---------------|--------------------------------|
| Partition | $O(n)$        | Place items in digit buckets   |
| Recurse   | $O(d)$        | Each level processes subgroups |
| Total     | $O(d(n + k))$ | Linear in digits               |

## Try It Yourself

1. Sort `["b", "ba", "abc", "ab", "ac"]`.
2. Draw recursion tree by character.
3. Compare order with lexicographic.
4. Test `["dog", "cat", "apple", "apricot"]`.
5. Sort integers `[170,45,75,90,802,24,2,66]`.
6. Change base (binary, hex).
7. Compare with LSD Radix Sort.
8. Add duplicates and test stability.
9. Visualize grouping buckets.
10. Implement with trie-like data structure.

## Test Cases

| Input                                       | Output                                      | Notes           |
|---------------------------------------------|---------------------------------------------|-----------------|
| <code>["b", "ba", "abc", "ab", "ac"]</code> | <code>["ab", "abc", "ac", "b", "ba"]</code> | Variable length |
| <code>[170,45,75,90,802,24,2,66]</code>     | <code>[2,24,45,66,75,90,170,802]</code>     | Numeric         |
| <code>["a", "aa", "aaa"]</code>             | <code>["a", "aa", "aaa"]</code>             | Prefix order    |
| <code>["z", "y", "x"]</code>                | <code>["x", "y", "z"]</code>                | Reverse input   |

## Complexity

| Aspect       | Value                 |
|--------------|-----------------------|
| Time         | $O(d \times (n + k))$ |
| Space        | $O(n + k)$            |
| Stable       | Yes                   |
| Adaptive     | No                    |
| Suitable for | Variable-length keys  |

Radix Sort (MSD) is lexicographic sorting by recursion, it builds order from the top down, treating prefixes as leaders and details as followers, much like how dictionaries arrange words.

## 125 Bucket Sort

Bucket Sort is a distribution-based sorting algorithm that divides the input into several buckets (bins), sorts each bucket individually (often with Insertion Sort), and then concatenates them. When input data is uniformly distributed, Bucket Sort achieves linear time performance ( $O(n)$ ).

### What Problem Are We Solving??

Comparison-based sorts take  $O(n \log n)$  time in the general case. But if we know that data values are spread evenly across a range, we can exploit this structure to sort faster by grouping similar values together.

Bucket Sort works best when:

- Input is real numbers in  $[0, 1]$  or any known range
- Data is uniformly distributed
- Buckets are balanced, each with few elements

Used in:

- Probability distributions
- Histogram-based sorting
- Floating-point sorting

### Example

Sort  $[0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68]$

| Step | Action                                             | Result                                                                 |
|------|----------------------------------------------------|------------------------------------------------------------------------|
| 1    | Create 10 buckets for range $[0, 1]$               | $[[], \dots, []]$                                                      |
| 2    | Distribute elements by <code>int(n * value)</code> | $[[0.12, 0.17, 0.21, 0.23, 0.26], [0.39], [0.68, 0.72, 0.78], [0.94]]$ |
| 3    | Sort each bucket (Insertion Sort)                  | Each bucket sorted individually                                        |
| 4    | Concatenate buckets                                | $[0.12, 0.17, 0.21, 0.23, 0.26, 0.39, 0.68, 0.72, 0.78, 0.94]$         |

## How Does It Work (Plain Language)?

Think of sorting test scores:

- You group scores into bins (0–10, 10–20, 20–30, ...)
- Sort each bin individually
- Merge bins back together in order

Bucket Sort leverages range grouping, local order inside each bucket, global order from bucket sequence.

## Step-by-Step Process

| Step | Description                                  |
|------|----------------------------------------------|
| 1    | Create empty buckets for each range interval |
| 2    | Distribute elements into buckets             |
| 3    | Sort each bucket individually                |
| 4    | Concatenate buckets sequentially             |

If buckets are evenly filled, each small sort is fast, almost constant time.

## Tiny Code (Easy Versions)

C

```
#include <stdio.h>
#include <stdlib.h>

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

```

void bucket_sort(float arr[], int n) {
    float buckets[n][n];
    int count[n];
    for (int i = 0; i < n; i++) count[i] = 0;

    // Distribute into buckets
    for (int i = 0; i < n; i++) {
        int idx = n * arr[i]; // index by range
        buckets[idx][count[idx]++] = arr[i];
    }

    // Sort each bucket
    for (int i = 0; i < n; i++)
        if (count[i] > 0)
            insertion_sort(buckets[i], count[i]);

    // Concatenate
    int k = 0;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < count[i]; j++)
            arr[k++] = buckets[i][j];
}

int main(void) {
    float arr[] = {0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68};
    int n = sizeof(arr) / sizeof(arr[0]);
    bucket_sort(arr, n);
    for (int i = 0; i < n; i++) printf("%.2f ", arr[i]);
    printf("\n");
}

```

## Python

```

def bucket_sort(arr):
    n = len(arr)
    buckets = [[] for _ in range(n)]

    for num in arr:
        idx = int(n * num)
        buckets[idx].append(num)

```

```

for bucket in buckets:
    bucket.sort()

result = []
for bucket in buckets:
    result.extend(bucket)
return result

arr = [0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68]
print(bucket_sort(arr))

```

## Why It Matters

- Linear time for uniformly distributed data
- Great for floating-point numbers
- Illustrates distribution-based sorting
- Foundation for histogram, flash, and spread sort

## A Gentle Proof (Why It Works)

If input elements are independent and uniformly distributed, expected elements per bucket =  $O(1)$ . Sorting each small bucket takes constant time → total linear time.

Total complexity:

$$T(n) = O(n + \sum_{i=1}^n T_i)$$

If each  $T_i = O(1)$ ,

$$T(n) = O(n)$$

| Phase        | Work         | Complexity |
|--------------|--------------|------------|
| Distribution | $O(n)$       | One pass   |
| Local Sort   | $O(n)$ total | Expected   |
| Concatenate  | $O(n)$       | Combine    |

## Try It Yourself

1. Sort  $[0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68]$ .
2. Visualize buckets as bins.
3. Change bucket count to 5, 20, see effect.
4. Try non-uniform data  $[0.99, 0.99, 0.98]$ .
5. Replace insertion sort with counting sort.
6. Measure performance with 10 floats.
7. Test  $[0.1, 0.01, 0.001]$ , uneven distribution.
8. Implement bucket indexing for arbitrary ranges.
9. Compare with Quick Sort runtime.
10. Plot distribution histogram before sorting.

## Test Cases

| Input                      | Output                     | Notes      |
|----------------------------|----------------------------|------------|
| $[0.78, 0.17, 0.39, 0.26]$ | $[0.17, 0.26, 0.39, 0.78]$ | Basic      |
| $[0.1, 0.01, 0.001]$       | $[0.001, 0.01, 0.1]$       | Sparse     |
| $[0.9, 0.8, 0.7, 0.6]$     | $[0.6, 0.7, 0.8, 0.9]$     | Reverse    |
| $[0.1, 0.1, 0.1]$          | $[0.1, 0.1, 0.1]$          | Duplicates |

## Complexity

| Aspect         | Value                          |
|----------------|--------------------------------|
| Time (Best)    | $O(n)$                         |
| Time (Average) | $O(n)$                         |
| Time (Worst)   | $O(n^2)$ (all in one bucket)   |
| Space          | $O(n)$                         |
| Stable         | Yes (if bucket sort is stable) |
| Adaptive       | Yes (depends on distribution)  |

Bucket Sort is like sorting by bins, fast, simple, and beautifully efficient when your data is evenly spread. It's the go-to choice for continuous values in a fixed range.

## 126 Pigeonhole Sort

Pigeonhole Sort is a simple distribution sorting algorithm that places each element directly into its corresponding “pigeonhole” (or bucket) based on its key value. It's ideal when elements

are integers within a small known range, think of it as Counting Sort with explicit placement rather than counting.

### What Problem Are We Solving??

When data values are integers and close together, we don't need comparisons, we can map each value to a slot directly. Pigeonhole Sort is particularly useful for dense integer ranges, such as:

- Sorting integers from 0 to 100
- Sorting scores, ranks, or IDs
- Small ranges with many duplicates

It trades space for speed, achieving  $O(n + \text{range})$  performance.

### Example

Sort [8, 3, 2, 7, 4, 6, 8]

| Step | Description                                           | Result                                 |
|------|-------------------------------------------------------|----------------------------------------|
| 1    | Find $\min=2$ , $\max=8 \rightarrow \text{range} = 7$ | holes[0..6]                            |
| 2    | Create pigeonholes:<br>[[],[],[],[],[],[],[]]         |                                        |
| 3    | Place each number in its hole                         | holes =<br>[[2],[3],[4],[6],[7],[8,8]] |
| 4    | Concatenate holes                                     | [2,3,4,6,7,8,8]                        |

Each value goes exactly to its mapped hole index.

### How Does It Work (Plain Language)?

It's like assigning students to exam rooms based on ID ranges, each slot holds all matching IDs. You fill slots (holes), then read them back in order.

Unlike Counting Sort, which only counts occurrences, Pigeonhole Sort stores the actual values, preserving duplicates directly.

### Step-by-Step Process

| Step | Description                                             |
|------|---------------------------------------------------------|
| 1    | Find minimum and maximum elements                       |
| 2    | Compute range = max - min + 1                           |
| 3    | Create array of empty pigeonholes of size range         |
| 4    | For each element, map to hole <code>arr[i] - min</code> |
| 5    | Place element into that hole (append)                   |
| 6    | Read holes in order and flatten into output             |

## Tiny Code (Easy Versions)

C

```
#include <stdio.h>
#include <stdlib.h>

void pigeonhole_sort(int arr[], int n) {
    int min = arr[0], max = arr[0];
    for (int i = 1; i < n; i++) {
        if (arr[i] < min) min = arr[i];
        if (arr[i] > max) max = arr[i];
    }

    int range = max - min + 1;
    int *holes[range];
    int counts[range];
    for (int i = 0; i < range; i++) {
        holes[i] = malloc(n * sizeof(int));
        counts[i] = 0;
    }

    // Place elements into holes
    for (int i = 0; i < n; i++) {
        int index = arr[i] - min;
        holes[index][counts[index]++] = arr[i];
    }

    // Flatten back
    int index = 0;
    for (int i = 0; i < range; i++) {
        for (int j = 0; j < counts[i]; j++) {
            arr[index++] = holes[i][j];
        }
    }
}
```

```

        }
        free(holes[i]);
    }
}

int main(void) {
    int arr[] = {8, 3, 2, 7, 4, 6, 8};
    int n = sizeof(arr) / sizeof(arr[0]);
    pigeonhole_sort(arr, n);
    for (int i = 0; i < n; i++) printf("%d ", arr[i]);
    printf("\n");
}

```

## Python

```

def pigeonhole_sort(arr):
    min_val, max_val = min(arr), max(arr)
    size = max_val - min_val + 1
    holes = [[] for _ in range(size)]

    for x in arr:
        holes[x - min_val].append(x)

    sorted_arr = []
    for hole in holes:
        sorted_arr.extend(hole)
    return sorted_arr

arr = [8, 3, 2, 7, 4, 6, 8]
print(pigeonhole_sort(arr))

```

## Why It Matters

- Simple mapping for small integer ranges
- Linear time if range  $n$
- Useful in digit, rank, or ID sorting
- Provides stable grouping with explicit placement

## A Gentle Proof (Why It Works)

Every key is mapped uniquely to a hole (offset by min). All duplicates fall into the same hole, preserving multiplicity. Reading holes sequentially yields sorted order.

If  $n$  = number of elements,  $k$  = range of values:

$$T(n) = O(n + k)$$

| Phase               | Work       | Complexity         |
|---------------------|------------|--------------------|
| Find min/max        | $O(n)$     | Single pass        |
| Distribute to holes | $O(n)$     | One placement each |
| Collect results     | $O(n + k)$ | Flatten all        |

## Try It Yourself

1. Sort  $[8,3,2,7,4,6,8]$  by hand.
2. Show hole contents after distribution.
3. Add duplicate values, confirm stable grouping.
4. Try  $[1,1,1,1]$ , all in one hole.
5. Sort negative numbers  $[0, -1, -2, 1]$  (offset by min).
6. Increase range to see space cost.
7. Compare runtime with Counting Sort.
8. Replace holes with linked lists.
9. Implement in-place version.
10. Extend for key-value pairs.

## Test Cases

| Input             | Output            | Notes           |
|-------------------|-------------------|-----------------|
| $[8,3,2,7,4,6,8]$ | $[2,3,4,6,7,8,8]$ | Example         |
| $[1,1,1,1]$       | $[1,1,1,1]$       | All equal       |
| $[9,8,7,6]$       | $[6,7,8,9]$       | Reverse         |
| $[0,-1,-2,1]$     | $[-2,-1,0,1]$     | Negative offset |

## Complexity

| Aspect          | Value      |
|-----------------|------------|
| Time            | $O(n + k)$ |
| Space           | $O(n + k)$ |
| Stable          | Yes        |
| Adaptive        | No         |
| Range-sensitive | Yes        |

Pigeonhole Sort is as direct as sorting gets, one slot per value, one pass in, one pass out. Fast and clean when your data is dense and discrete.

## 127 Flash Sort

Flash Sort is a distribution-based sorting algorithm that combines ideas from bucket sort and insertion sort. It works in two phases:

1. Classification, distribute elements into classes (buckets) using a linear mapping
2. Permutation, rearrange elements in-place using cycles

It achieves  $O(n)$  average time on uniformly distributed data but can degrade to  $O(n^2)$  in the worst case. Invented by *Karl-Dietrich Neubert* (1990s), it's known for being extremely fast in practice on large datasets.

### What Problem Are We Solving??

When data is numerically distributed over a range, we can approximate where each element should go and move it close to its final position without full comparison sorting.

Flash Sort is built for:

- Uniformly distributed numeric data
- Large arrays
- Performance-critical applications (e.g., simulations, physics, graphics)

It leverages approximate indexing and in-place permutation for speed.

### Example

Sort [9, 3, 1, 7, 4, 6, 2, 8, 5] into  $m = 5$  classes.

| Step | Description                                                         | Result                             |
|------|---------------------------------------------------------------------|------------------------------------|
| 1    | Find $\min = 1, \max = 9$                                           | range = 8                          |
| 2    | Map each value to class $k = \frac{(m-1)*(a[i]-\min)}{(\max-\min)}$ | Class indices: [4,1,0,3,1,3,0,4,2] |
| 3    | Count elements per class, compute cumulative positions              | class boundaries = [0,2,4,6,8,9]   |
| 4    | Cycle elements into correct class positions                         | Rearranged approx order            |
| 5    | Apply insertion sort within classes                                 | Sorted list                        |

Final: [1,2,3,4,5,6,7,8,9]

### How Does It Work (Plain Language)?

Imagine flashing each element to its approximate destination class in one pass, that's the "flash" phase. Then, fine-tune within each class using a simpler sort (like Insertion Sort).

It's like placing books roughly into shelves, then tidying each shelf.

### Step-by-Step Process

| Step | Description                                                |
|------|------------------------------------------------------------|
| 1    | Find min and max                                           |
| 2    | Choose number of classes (usually $0.43 * n$ )             |
| 3    | Compute class index for each element                       |
| 4    | Count elements per class and compute prefix sums           |
| 5    | Move elements to approximate class positions (flash phase) |
| 6    | Use Insertion Sort within each class to finish             |

### Tiny Code (Easy Versions)

C

```
#include <stdio.h>

void insertion_sort(float arr[], int start, int end) {
    for (int i = start + 1; i <= end; i++) {
        float key = arr[i];
        int j = i - 1;
```

```

        while (j >= start && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

void flash_sort(float arr[], int n) {
    if (n <= 1) return;

    float min = arr[0], max = arr[0];
    for (int i = 1; i < n; i++) {
        if (arr[i] < min) min = arr[i];
        if (arr[i] > max) max = arr[i];
    }
    if (max == min) return;

    int m = n * 0.43; // number of classes
    int L[m];
    for (int i = 0; i < m; i++) L[i] = 0;

    // Classification count
    for (int i = 0; i < n; i++) {
        int k = (int)((m - 1) * (arr[i] - min) / (max - min));
        L[k]++;
    }

    // Prefix sums
    for (int i = 1; i < m; i++) L[i] += L[i - 1];

    // Flash phase
    int move = 0, j = 0, k = m - 1;
    while (move < n - 1) {
        while (j >= L[k]) k = (int)((m - 1) * (arr[j] - min) / (max - min));
        float flash = arr[j];
        while (j != L[k]) {
            k = (int)((m - 1) * (flash - min) / (max - min));
            float hold = arr[--L[k]];
            arr[L[k]] = flash;
            flash = hold;
            move++;
        }
    }
}

```

```

        }
        j++;
    }

    // Insertion sort finish
    insertion_sort(arr, 0, n - 1);
}

int main(void) {
    float arr[] = {9,3,1,7,4,6,2,8,5};
    int n = sizeof(arr) / sizeof(arr[0]);
    flash_sort(arr, n);
    for (int i = 0; i < n; i++) printf("%.0f ", arr[i]);
    printf("\n");
}

```

## Python

```

def insertion_sort(arr, start, end):
    for i in range(start + 1, end + 1):
        key = arr[i]
        j = i - 1
        while j >= start and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

def flash_sort(arr):
    n = len(arr)
    if n <= 1:
        return arr
    min_val, max_val = min(arr), max(arr)
    if max_val == min_val:
        return arr
    m = int(0.43 * n)
    L = [0] * m

    for x in arr:
        k = int((m - 1) * (x - min_val) / (max_val - min_val))
        L[k] += 1

```

```

for i in range(1, m):
    L[i] += L[i - 1]

move, j, k = 0, 0, m - 1
while move < n - 1:
    while j >= L[k]:
        k = int((m - 1) * (arr[j] - min_val) / (max_val - min_val))
    flash = arr[j]
    while j != L[k]:
        k = int((m - 1) * (flash - min_val) / (max_val - min_val))
        L[k] -= 1
        arr[L[k]], flash = flash, arr[L[k]]
        move += 1
    j += 1
insertion_sort(arr, 0, n - 1)
return arr

arr = [9,3,1,7,4,6,2,8,5]
print(flash_sort(arr))

```

## Why It Matters

- Extremely fast on uniformly distributed data
- In-place ( $O(1)$  extra space)
- Practical for large arrays
- Combines distribution sorting and insertion finishing

## A Gentle Proof (Why It Works)

The classification step maps each element to its expected position region, dramatically reducing disorder. Since each class has a small local range, insertion sort completes quickly.

Expected complexity (uniform input):

$$T(n) = O(n) \text{ (classification)} + O(n) \text{ (final pass)} = O(n)$$

Worst-case (skewed distribution):  $O(n^2)$

| Phase          | Work   | Complexity      |
|----------------|--------|-----------------|
| Classification | $O(n)$ | Compute classes |

| Phase               | Work   | Complexity     |
|---------------------|--------|----------------|
| Flash rearrangement | $O(n)$ | In-place moves |
| Final sort          | $O(n)$ | Local sorting  |

### Try It Yourself

1. Sort  $[9,3,1,7,4,6,2,8,5]$  step by step.
2. Change number of classes ( $0.3n$ ,  $0.5n$ ).
3. Visualize class mapping for each element.
4. Count moves in flash phase.
5. Compare with Quick Sort timing on 10 elements.
6. Test uniform vs skewed data.
7. Implement with different finishing sort.
8. Track cycles formed during flash phase.
9. Observe stability (it's not stable).
10. Benchmark against Merge Sort.

### Test Cases

| Input                 | Output                | Notes          |
|-----------------------|-----------------------|----------------|
| $[9,3,1,7,4,6,2,8,5]$ | $[1,2,3,4,5,6,7,8,9]$ | Example        |
| $[5,5,5,5]$           | $[5,5,5,5]$           | Equal elements |
| $[1,2,3,4,5]$         | $[1,2,3,4,5]$         | Already sorted |
| $[9,8,7,6,5]$         | $[5,6,7,8,9]$         | Reverse        |

### Complexity

| Aspect         | Value           |
|----------------|-----------------|
| Time (Best)    | $O(n)$          |
| Time (Average) | $O(n)$          |
| Time (Worst)   | $O(n^2)$        |
| Space          | $O(1)$          |
| Stable         | No              |
| Adaptive       | Yes (partially) |

Flash Sort is the lightning strike of sorting, it flashes elements to their expected zones in linear time, then finishes with a quick polish. When your data is uniform, it's one of the fastest practical sorts around.

## 128 Postman Sort

Postman Sort is a stable, multi-key sorting algorithm that works by sorting keys digit by digit or field by field, starting from the least significant field (like LSD Radix Sort) or most significant field (like MSD Radix Sort), depending on the application. It's often used for compound keys (e.g. postal addresses, dates, strings of fields), hence the name "Postman," as it sorts data the way a postman organizes mail: by street, then house, then apartment.

### What Problem Are We Solving??

When sorting complex records by multiple attributes, such as:

- Sorting people by (`city, street, house_number`)
- Sorting files by (`year, month, day`)
- Sorting products by (`category, brand, price`)

We need to sort hierarchically, that's what Postman Sort excels at. It's a stable, field-wise sorting approach, built upon Counting or Bucket Sort for each field.

### Example

Sort tuples (`City, Street`) by City then Street:

```
[$$("Paris", "B"), ("London", "C"), ("Paris", "A"), ("London", "A)]
```

| Step | Sort By      | Result                                                             |
|------|--------------|--------------------------------------------------------------------|
| 1    | Street (LSD) | [("London", "A"), ("Paris", "A"), ("London", "C"), ("Paris", "B")] |
| 2    | City (MSD)   | [("London", "A"), ("London", "C"), ("Paris", "A"), ("Paris", "B")] |

Stable sorting ensures inner order is preserved from previous pass.

## How Does It Work (Plain Language)?

It's like organizing mail:

1. Sort by the smallest unit (house number)
2. Then sort by street
3. Then by city

Each pass refines the previous ordering. If sorting from least to most significant, use LSD order (like Radix). If sorting from most to least, use MSD order (like bucket recursion).

## Step-by-Step Process

| Step | Description                                         |
|------|-----------------------------------------------------|
| 1    | Identify key fields and their order of significance |
| 2    | Choose stable sorting method (e.g., Counting Sort)  |
| 3    | Sort by least significant key first                 |
| 4    | Repeat for each key moving to most significant      |
| 5    | Final order respects all key hierarchies            |

## Tiny Code (Easy Versions)

### Python (LSD Approach)

```
def postman_sort(records, key_funcs):
    # key_funcs = list of functions to extract each field
    for key_func in reversed(key_funcs):
        records.sort(key=key_func)
    return records

# Example: sort by (city, street)
records = [("Paris", "B"), ("London", "C"), ("Paris", "A"), ("London", "A")]
sorted_records = postman_sort(records, [lambda x: x[0], lambda x: x[1]])
print(sorted_records)
```

Output:

```
[$('London', 'A'), ('London', 'C'), ('Paris', 'A'), ('Paris', 'B')]
```

## C (Numeric Fields)

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int city;
    int street;
} Record;

int cmp_street(const void *a, const void *b) {
    Record *ra = (Record*)a, *rb = (Record*)b;
    return ra->street - rb->street;
}

int cmp_city(const void *a, const void *b) {
    Record *ra = (Record*)a, *rb = (Record*)b;
    return ra->city - rb->city;
}

void postman_sort(Record arr[], int n) {
    // Sort by street first (LSD)
    qsort(arr, n, sizeof(Record), cmp_street);
    // Then by city (MSD)
    qsort(arr, n, sizeof(Record), cmp_city);
}

int main(void) {
    Record arr[] = {{2,3}, {1,2}, {2,1}, {1,1}};
    int n = sizeof(arr)/sizeof(arr[0]);
    postman_sort(arr, n);
    for (int i = 0; i < n; i++)
        printf("(%d,%d) ", arr[i].city, arr[i].street);
    printf("\n");
}
```

## Why It Matters

- Multi-key sorting (lexicographic order)
- Stable, preserves order across passes

- Versatile, works on compound keys, strings, records
- Foundation for:
  - Radix Sort
  - Database ORDER BY multi-column
  - Lexicographic ranking

### A Gentle Proof (Why It Works)

Each stable pass ensures that prior ordering (from less significant fields) remains intact.

If we denote each field as  $f_i$ , sorted stably in order  $f_k \rightarrow f_1$ :

$$T(n) = \sum_{i=1}^k T_i(n)$$

If each  $T_i(n)$  is  $O(n)$ , the total is  $O(kn)$ .

Lexicographic order emerges naturally:

$$(a_1, a_2, \dots, a_k) < (b_1, b_2, \dots, b_k) \iff a_i = b_i \text{ for } i < j, \text{ and } a_j < b_j$$

| Phase    | Operation                    | Stable | Complexity |
|----------|------------------------------|--------|------------|
| LSD Sort | Start from least significant | Yes    | $O(nk)$    |
| MSD Sort | Start from most significant  | Yes    | $O(nk)$    |

### Try It Yourself

1. Sort `[("Paris", "B"), ("London", "C"), ("Paris", "A"), ("London", "A")]`
2. Add a 3rd field (zip code), sort by zip → street → city
3. Implement with `lambda` keys in Python
4. Replace stable sort with Counting Sort per field
5. Test with `[(2021, 12, 25), (2020, 1, 1), (2021, 1, 1)]`
6. Compare LSD vs MSD ordering
7. Sort strings by character groups (first char, second char, etc.)
8. Visualize passes and intermediate results
9. Test stability with repeated keys
10. Apply to sorting student records by (`grade`, `class`, `id`)

## Test Cases

| Input                                                              | Output                                                             | Notes          |
|--------------------------------------------------------------------|--------------------------------------------------------------------|----------------|
| [("Paris", "B"), ("London", "C"), ("Paris", "A"), ("London", "A")] | [("London", "A"), ("London", "C"), ("Paris", "A"), ("Paris", "B")] | Lexico-graphic |
| [(2021,12,25), (2020,1,1), (2021,1,1)]                             | [(2020,1,1), (2021,1,1), (2021,12,25)]                             | Date order     |
| [(1,2,3), (1,1,3), (1,1,2)]                                        | [(1,1,2), (1,1,3), (1,2,3)]                                        | Multi-field    |

## Complexity

| Aspect   | Value           |
|----------|-----------------|
| Time     | $O(k \times n)$ |
| Space    | $O(n)$          |
| Stable   | Yes             |
| Adaptive | No              |
| Keys     | Multi-field     |

Postman Sort delivers order like clockwork, field by field, pass by pass, ensuring every layer of your data finds its proper place, from apartment number to zip code.

## 129 Address Calculation Sort

Address Calculation Sort (sometimes called Hash Sort or Scatter Sort) is a distribution-based sorting method that uses a hash-like function (called an *address function*) to compute the final position of each element directly. Instead of comparing pairs, it computes where each element should go, much like direct addressing in hash tables.

### What Problem Are We Solving??

Comparison sorts need  $O(n \log n)$  time. If we know the range and distribution of input values, we can instead compute where each element belongs, placing it directly.

Address Calculation Sort bridges the gap between sorting and hashing:

- It computes a mapping from value to position.

- It places each element directly or into small groups.
- It works best when data distribution is known and uniform.

Applications:

- Dense numeric datasets
- Ranked records
- Pre-bucketed ranges

### Example

Sort [3, 1, 4, 0, 2] with range [0..4].

| Step | Element | Address Function $f(x) = x$ | Placement  |
|------|---------|-----------------------------|------------|
| 1    | 3       | $f(3) = 3$                  | position 3 |
| 2    | 1       | $f(1) = 1$                  | position 1 |
| 3    | 4       | $f(4) = 4$                  | position 4 |
| 4    | 0       | $f(0) = 0$                  | position 0 |
| 5    | 2       | $f(2) = 2$                  | position 2 |

Result: [0, 1, 2, 3, 4]

If multiple elements share the same address, they're stored in a small linked list or bucket, then sorted locally.

### How Does It Work (Plain Language)?

Imagine having labeled mailboxes for every possible key — each element knows exactly which box it belongs in. You just drop each letter into its slot, then read the boxes in order.

Unlike Counting Sort, which counts occurrences, Address Calculation Sort assigns positions, it can even be in-place if collisions are handled carefully.

### Step-by-Step Process

| Step | Description                                                     |
|------|-----------------------------------------------------------------|
| 1    | Define address function $f(x)$ mapping each element to an index |
| 2    | Initialize empty output array or buckets                        |
| 3    | For each element $a_i$ : compute index = $f(a_i)$               |
| 4    | Place element at index (handle collisions if needed)            |

---

| Step | Description                                  |
|------|----------------------------------------------|
| 5    | Collect or flatten buckets into sorted order |

---

### Tiny Code (Easy Versions)

#### C (Simple In-Range Example)

```
#include <stdio.h>
#include <stdlib.h>

void address_calculation_sort(int arr[], int n, int min, int max) {
    int range = max - min + 1;
    int *output = calloc(range, sizeof(int));
    int *filled = calloc(range, sizeof(int));

    for (int i = 0; i < n; i++) {
        int idx = arr[i] - min;
        output[idx] = arr[i];
        filled[idx] = 1;
    }

    int k = 0;
    for (int i = 0; i < range; i++) {
        if (filled[i]) arr[k++] = output[i];
    }

    free(output);
    free(filled);
}

int main(void) {
    int arr[] = {3, 1, 4, 0, 2};
    int n = sizeof(arr)/sizeof(arr[0]);
    address_calculation_sort(arr, n, 0, 4);
    for (int i = 0; i < n; i++) printf("%d ", arr[i]);
    printf("\n");
}
```

#### Python

```

def address_calculation_sort(arr, f=None):
    if not arr:
        return arr
    if f is None:
        f = lambda x: x # identity mapping

    min_val, max_val = min(arr), max(arr)
    size = max_val - min_val + 1
    slots = [[] for _ in range(size)]

    for x in arr:
        idx = f(x) - min_val
        slots[idx].append(x)

    # Flatten buckets
    result = []
    for bucket in slots:
        result.extend(sorted(bucket)) # local sort if needed
    return result

arr = [3, 1, 4, 0, 2]
print(address_calculation_sort(arr))

```

Output:

[\$\$0, 1, 2, 3, 4]

## Why It Matters

- Direct computation of sorted positions
- Linear time for predictable distributions
- Combines hashing and sorting
- Forms basis of bucket, radix, and flash sort

It's a great way to see how functions can replace comparisons.

## A Gentle Proof (Why It Works)

If  $f(x)$  maps each key uniquely to its sorted index, the result is already sorted by construction.

Even with collisions, sorting small local buckets is fast.

Total time:

$$T(n) = O(n + k)$$

where  $k$  = number of buckets = range size.

| Phase             | Work       | Complexity         |
|-------------------|------------|--------------------|
| Compute addresses | $O(n)$     | One per element    |
| Bucket insertions | $O(n)$     | Constant time each |
| Local sort        | $O(k)$     | Small groups       |
| Flatten           | $O(n + k)$ | Read back          |

### Try It Yourself

1. Sort [3, 1, 4, 0, 2] with  $f(x)=x$ .
2. Change mapping to  $f(x)=2*x$ , note gaps.
3. Add duplicates [1,1,2,2,3], use buckets.
4. Try floats with  $f(x)=\text{int}(x*10)$  for [0.1,0.2,0.3].
5. Sort strings by ASCII sum:  $f(s)=\text{sum}(\text{map}(\text{ord}, s))$ .
6. Compare with Counting Sort (no explicit storage).
7. Handle negative numbers by offsetting min.
8. Visualize mapping table.
9. Test range gaps (e.g., [10, 20, 30]).
10. Experiment with custom hash functions.

### Test Cases

| Input         | Output        | Notes           |
|---------------|---------------|-----------------|
| [3,1,4,0,2]   | [0,1,2,3,4]   | Perfect mapping |
| [10,20,30]    | [10,20,30]    | Sparse mapping  |
| [1,1,2,2]     | [1,1,2,2]     | Duplicates      |
| [0.1,0.3,0.2] | [0.1,0.2,0.3] | Float mapping   |

### Complexity

| Aspect | Value              |
|--------|--------------------|
| Time   | $O(n + k)$         |
| Space  | $O(n + k)$         |
| Stable | Yes (with buckets) |

| Aspect          | Value |
|-----------------|-------|
| Adaptive        | No    |
| Range-sensitive | Yes   |

Address Calculation Sort turns sorting into placement, each value finds its home by formula, not fight. When the range is clear and collisions are rare, it's lightning-fast and elegantly simple.

## 130 Spread Sort

Spread Sort is a hybrid distribution sort that blends ideas from radix sort, bucket sort, and comparison sorting. It distributes elements into buckets using their most significant bits (MSB) or value ranges, then recursively sorts buckets (like radix/MSD) or switches to a comparison sort (like Quick Sort) when buckets are small.

It's cache-friendly, adaptive, and often faster than Quick Sort on uniformly distributed data. In fact, it's used in some high-performance libraries such as Boost C++ Spreadsor.

### What Problem Are We Solving??

Traditional comparison sorts (like Quick Sort) have  $O(n \log n)$  complexity, while pure radix-based sorts can be inefficient on small or skewed datasets. Spread Sort solves this by adapting dynamically:

- Distribute like radix sort when data is wide and random
- Compare like Quick Sort when data is clustered or buckets are small

It "spreads" data across buckets, then sorts each bucket intelligently.

Perfect for:

- Integers, floats, and strings
- Large datasets
- Wide value ranges

### Example

Sort [43, 12, 89, 27, 55, 31, 70]

| Step | Action                                      | Result                                   |
|------|---------------------------------------------|------------------------------------------|
| 1    | Find min = 12, max = 89                     | range = 77                               |
| 2    | Choose bucket count (e.g. 4)                | bucket size 19                           |
| 3    | Distribute by bucket index = (val - min)/19 | Buckets: [12], [27,31], [43,55], [70,89] |
| 4    | Recursively sort each bucket                | [12], [27,31], [43,55], [70,89]          |
| 5    | Merge buckets                               | [12,27,31,43,55,70,89]                   |

### How Does It Work (Plain Language)?

Imagine sorting mail by first letter (distribution), then alphabetizing each pile (comparison). If a pile is still big, spread it again by the next letter. If a pile is small, just sort it directly.

Spread Sort uses:

- Distribution when the data range is wide
- Comparison sort when buckets are narrow or small

This flexibility gives it strong real-world performance.

### Step-by-Step Process

| Step | Description                                     |
|------|-------------------------------------------------|
| 1    | Find min and max values                         |
| 2    | Compute spread = max - min                      |
| 3    | Choose bucket count (based on n or spread)      |
| 4    | Distribute elements into buckets by value range |
| 5    | Recursively apply spread sort to large buckets  |
| 6    | Apply comparison sort to small buckets          |
| 7    | Concatenate results                             |

### Tiny Code (Easy Versions)

#### C (Integer Example)

```
#include <stdio.h>
#include <stdlib.h>

int compare(const void *a, const void *b) {
```

```

    return (*int*)a - *(int*)b;
}

void spread_sort(int arr[], int n) {
    if (n <= 16) { // threshold for small buckets
        qsort(arr, n, sizeof(int), compare);
        return;
    }

    int min = arr[0], max = arr[0];
    for (int i = 1; i < n; i++) {
        if (arr[i] < min) min = arr[i];
        if (arr[i] > max) max = arr[i];
    }

    int bucket_count = n / 4 + 1;
    int range = max - min + 1;
    int bucket_size = range / bucket_count + 1;

    int *counts = calloc(bucket_count, sizeof(int));
    int buckets = malloc(bucket_count * sizeof(int*));
    for (int i = 0; i < bucket_count; i++)
        buckets[i] = malloc(n * sizeof(int));

    // Distribution
    for (int i = 0; i < n; i++) {
        int idx = (arr[i] - min) / bucket_size;
        buckets[idx][counts[idx]++] = arr[i];
    }

    // Recursive sort
    int k = 0;
    for (int i = 0; i < bucket_count; i++) {
        if (counts[i] > 0) {
            spread_sort(buckets[i], counts[i]);
            for (int j = 0; j < counts[i]; j++)
                arr[k++] = buckets[i][j];
        }
        free(buckets[i]);
    }

    free(buckets);
}

```

```

        free(counts);
    }

int main(void) {
    int arr[] = {43, 12, 89, 27, 55, 31, 70};
    int n = sizeof(arr)/sizeof(arr[0]);
    spread_sort(arr, n);
    for (int i = 0; i < n; i++) printf("%d ", arr[i]);
    printf("\n");
}

```

## Python

```

def spread_sort(arr, threshold=16):
    if len(arr) <= threshold:
        return sorted(arr)

    min_val, max_val = min(arr), max(arr)
    if min_val == max_val:
        return arr[:]

    n = len(arr)
    bucket_count = n // 4 + 1
    spread = max_val - min_val + 1
    bucket_size = spread // bucket_count + 1

    buckets = [[] for _ in range(bucket_count)]
    for x in arr:
        idx = (x - min_val) // bucket_size
        buckets[idx].append(x)

    result = []
    for b in buckets:
        if len(b) > threshold:
            result.extend(spread_sort(b, threshold))
        else:
            result.extend(sorted(b))
    return result

arr = [43, 12, 89, 27, 55, 31, 70]
print(spread_sort(arr))

```

## Why It Matters

- Linear-time on uniform data
- Adaptive to distribution and size
- Combines bucket and comparison power
- Great for large numeric datasets

Used in Boost C++, it's a real-world performant hybrid.

## A Gentle Proof (Why It Works)

Spread Sort's cost depends on:

- Distribution pass:  $O(n)$
- Recursion depth: small (logarithmic for uniform data)
- Local sorts: small and fast (often  $O(1)$  or  $O(n \log n)$  on small buckets)

For  $n$  elements and  $b$  buckets:

$$T(n) = O\left(n + \sum_{i=1}^b T_i\right)$$

If average bucket size is constant or small:

$$T(n) \approx O(n)$$

| Phase        | Work                                | Complexity                       |
|--------------|-------------------------------------|----------------------------------|
| Distribution | $O(n)$                              | One pass                         |
| Local Sorts  | $O(n \log m)$                       | $m = \text{average bucket size}$ |
| Total        | $O(n)$ average, $O(n \log n)$ worst |                                  |

## Try It Yourself

1. Sort [43, 12, 89, 27, 55, 31, 70].
2. Try [5, 4, 3, 2, 1] (non-uniform).
3. Adjust bucket threshold.
4. Visualize recursive buckets.
5. Mix large and small values (e.g., [1, 10, 1000, 2, 20, 2000]).
6. Compare runtime with Quick Sort.

7. Implement float version.
8. Measure distribution imbalance.
9. Tune bucket size formula.
10. Sort strings by `ord(char)`.

## Test Cases

| Input                  | Output                 | Notes      |
|------------------------|------------------------|------------|
| [43,12,89,27,55,31,70] | [12,27,31,43,55,70,89] | Uniform    |
| [5,4,3,2,1]            | [1,2,3,4,5]            | Reverse    |
| [100,10,1,1000]        | [1,10,100,1000]        | Wide range |
| [5,5,5]                | [5,5,5]                | Duplicates |

## Complexity

| Aspect         | Value                         |
|----------------|-------------------------------|
| Time (Best)    | $O(n)$                        |
| Time (Average) | $O(n)$                        |
| Time (Worst)   | $O(n \log n)$                 |
| Space          | $O(n)$                        |
| Stable         | Yes (if local sort is stable) |
| Adaptive       | Yes                           |

Spread Sort spreads elements like seeds, each finds fertile ground in its range, then blossoms into order through local sorting. It's the smart hybrid that brings the best of radix and comparison worlds together.

## Section 14. Hybrid sorts

### 131 IntroSort

IntroSort (short for *Introspective Sort*) is a hybrid sorting algorithm that combines the best features of Quick Sort, Heap Sort, and Insertion Sort. It begins with Quick Sort for speed, but if recursion goes too deep (indicating unbalanced partitions), it switches to Heap Sort to guarantee worst-case  $O(n \log n)$  performance. For small partitions, it often uses Insertion Sort for efficiency.

It was introduced by David Musser (1997) and is the default sorting algorithm in C++ STL (`std::sort`), fast, adaptive, and safe.

### What Problem Are We Solving??

Pure Quick Sort is fast on average but can degrade to  $O(n^2)$  in the worst case (for example, sorted input with bad pivots).

Heap Sort guarantees  $O(n \log n)$  but has worse constants.

IntroSort combines them, using Quick Sort *until danger*, then switching to Heap Sort for safety.

Perfect for:

- General-purpose sorting (numeric, string, object)
- Performance-critical libraries
- Mixed data with unknown distribution

### Example

Sort [9, 3, 1, 7, 5, 4, 6, 2, 8]

| Step | Action                                              | Result                |
|------|-----------------------------------------------------|-----------------------|
| 1    | Start Quick Sort (depth = 0)                        | pivot = 5             |
| 2    | Partition → [3,1,2,4] [5] [9,7,6,8]                 | depth = 1             |
| 3    | Recurse left + right                                |                       |
| 4    | If depth > $2 \times \log(n)$ , switch to Heap Sort | prevents ( $O(n^2)$ ) |
| 5    | Use Insertion Sort on small segments                |                       |

Final sorted array: [1,2,3,4,5,6,7,8,9]

### How Does It Work (Plain Language)?

It's like a careful driver:

- Start fast on the highway (Quick Sort)
- If road gets tricky (too many turns = recursion depth), switch to 4-wheel drive (Heap Sort)
- For small parking spots (tiny arrays), use a nimble bike (Insertion Sort)

IntroSort keeps average speed high but avoids worst-case crashes.

## Step-by-Step Process

| Step | Description                                                           |
|------|-----------------------------------------------------------------------|
| 1    | Start Quick Sort recursively                                          |
| 2    | Track recursion depth                                                 |
| 3    | If depth exceeds threshold ( $2 \times \log n$ ), switch to Heap Sort |
| 4    | Use Insertion Sort for small subarrays (size < threshold)             |
| 5    | Combine results for final sorted array                                |

## Tiny Code (Easy Versions)

### C (Simplified Implementation)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define THRESHOLD 16

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

void heapify(int arr[], int n, int i) {
    int largest = i, l = 2*i + 1, r = 2*i + 2;
    if (l < n && arr[l] > arr[largest]) largest = l;
    if (r < n && arr[r] > arr[largest]) largest = r;
    if (largest != i) {
        int tmp = arr[i]; arr[i] = arr[largest]; arr[largest] = tmp;
        heapify(arr, n, largest);
    }
}

void heap_sort(int arr[], int n) {
    for (int i = n/2 - 1; i >= 0; i--) heapify(arr, n, i);
```

```

        for (int i = n - 1; i >= 0; i--) {
            int tmp = arr[0]; arr[0] = arr[i]; arr[i] = tmp;
            heapify(arr, i, 0);
        }
    }

int partition(int arr[], int low, int high) {
    int pivot = arr[high], i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            int tmp = arr[i]; arr[i] = arr[j]; arr[j] = tmp;
        }
    }
    int tmp = arr[i+1]; arr[i+1] = arr[high]; arr[high] = tmp;
    return i + 1;
}

void introsort_rec(int arr[], int low, int high, int depth_limit) {
    int n = high - low + 1;
    if (n <= THRESHOLD) {
        insertion_sort(arr + low, n);
        return;
    }
    if (depth_limit == 0) {
        heap_sort(arr + low, n);
        return;
    }
    int p = partition(arr, low, high);
    introsort_rec(arr, low, p - 1, depth_limit - 1);
    introsort_rec(arr, p + 1, high, depth_limit - 1);
}

void introsort(int arr[], int n) {
    int depth_limit = 2 * log(n);
    introsort_rec(arr, 0, n - 1, depth_limit);
}

int main(void) {
    int arr[] = {9,3,1,7,5,4,6,2,8};
    int n = sizeof(arr)/sizeof(arr[0]);
    introsort(arr, n);
}

```

```

    for (int i = 0; i < n; i++) printf("%d ", arr[i]);
    printf("\n");
}

```

## Python (Conceptual Demo)

```

import math

def insertion_sort(a):
    for i in range(1, len(a)):
        key = a[i]
        j = i - 1
        while j >= 0 and a[j] > key:
            a[j + 1] = a[j]
            j -= 1
        a[j + 1] = key

def heapify(a, n, i):
    largest = i
    l, r = 2*i+1, 2*i+2
    if l < n and a[l] > a[largest]:
        largest = l
    if r < n and a[r] > a[largest]:
        largest = r
    if largest != i:
        a[i], a[largest] = a[largest], a[i]
        heapify(a, n, largest)

def heap_sort(a):
    n = len(a)
    for i in range(n//2-1, -1, -1):
        heapify(a, n, i)
    for i in range(n-1, 0, -1):
        a[0], a[i] = a[i], a[0]
        heapify(a, i, 0)

def introsort(a, depth_limit=None):
    n = len(a)
    if n <= 16:
        insertion_sort(a)

```

```

        return a
if depth_limit is None:
    depth_limit = 2 * int(math.log2(n))
if depth_limit == 0:
    heap_sort(a)
    return a

pivot = a[-1]
left = [x for x in a[:-1] if x <= pivot]
right = [x for x in a[:-1] if x > pivot]
introsort(left, depth_limit - 1)
introsort(right, depth_limit - 1)
a[:] = left + [pivot] + right
return a

arr = [9,3,1,7,5,4,6,2,8]
print(introsort(arr))

```

## Why It Matters

- Default in C++ STL, fast and reliable
- Guaranteed worst-case  $O(n \log n)$
- Optimized for cache and small data
- Adaptive, uses best method for current scenario

## A Gentle Proof (Why It Works)

Quick Sort dominates until recursion depth =  $2 \log_2 n$ . At that point, worst-case risk appears → switch to Heap Sort (safe fallback). Small subarrays are handled by Insertion Sort for low overhead.

So overall:

$$T(n) = O(n \log n)$$

Always bounded by Heap Sort's worst case, but often near Quick Sort's best.

| Phase          | Method         | Complexity                 |
|----------------|----------------|----------------------------|
| Partitioning   | Quick Sort     | $O(n \log n)$              |
| Deep recursion | Heap Sort      | $O(n \log n)$              |
| Small arrays   | Insertion Sort | $O(n^2)$ local, negligible |

## Try It Yourself

1. Sort  $[9, 3, 1, 7, 5, 4, 6, 2, 8]$ .
2. Track recursion depth, switch to Heap Sort when  $> 2 \log_2 n$ .
3. Replace threshold = 16 with 8, 32, measure effect.
4. Test with already sorted array, confirm Heap fallback.
5. Compare timing with Quick Sort and Heap Sort.
6. Print method used at each stage.
7. Test large array (10 elements).
8. Verify worst-case safety on sorted input.
9. Try string sorting with custom comparator.
10. Implement generic version using templates or lambdas.

## Test Cases

| Input                         | Output                        | Notes                             |
|-------------------------------|-------------------------------|-----------------------------------|
| $[9, 3, 1, 7, 5, 4, 6, 2, 8]$ | $[1, 2, 3, 4, 5, 6, 7, 8, 9]$ | Balanced partitions               |
| $[1, 2, 3, 4, 5]$             | $[1, 2, 3, 4, 5]$             | Sorted input (Heap Sort fallback) |
| $[5, 5, 5, 5]$                | $[5, 5, 5, 5]$                | Equal elements                    |
| $[9, 8, 7, 6, 5, 4, 3, 2, 1]$ | $[1, 2, 3, 4, 5, 6, 7, 8, 9]$ | Worst-case Quick Sort avoided     |

## Complexity

| Aspect         | Value         |
|----------------|---------------|
| Time (Best)    | $O(n \log n)$ |
| Time (Average) | $O(n \log n)$ |
| Time (Worst)   | $O(n \log n)$ |
| Space          | $O(\log n)$   |
| Stable         | No            |
| Adaptive       | Yes           |

IntroSort is the strategist's algorithm, it starts bold (Quick Sort), defends wisely (Heap Sort), and finishes gracefully (Insertion Sort). A perfect balance of speed, safety, and adaptability.

## 132 TimSort

TimSort is a hybrid sorting algorithm combining Merge Sort and Insertion Sort, designed for real-world data that often contains partially ordered runs. It was invented by Tim Peters in 2002

and is the default sorting algorithm in Python (`sorted()`, `.sort()`) and Java (`Arrays.sort()` for objects).

TimSort's superpower is that it detects natural runs in data, sorts them with Insertion Sort if small, and merges them smartly using a stack-based strategy to ensure efficiency.

## What Problem Are We Solving??

In practice, many datasets aren't random, they already contain sorted segments (like logs, names, timestamps). TimSort exploits this by:

- Detecting ascending/descending runs
- Sorting small runs via Insertion Sort
- Merging runs using adaptive Merge Sort

This yields  $O(n)$  performance on already-sorted or nearly-sorted data, far better than standard  $O(n \log n)$  sorts in such cases.

Perfect for:

- Partially sorted lists
- Real-world data (time series, strings, logs)
- Stable sorting (preserve order of equals)

## Example

Sort [5, 6, 7, 1, 2, 3, 8, 9]

| Step | Action                    | Result                  |
|------|---------------------------|-------------------------|
| 1    | Detect runs               | [5,6,7], [1,2,3], [8,9] |
| 2    | Sort small runs if needed | [5,6,7], [1,2,3], [8,9] |
| 3    | Merge runs pairwise       | [1,2,3,5,6,7,8,9]       |

TimSort leverages order already present, fewer merges, faster finish.

## How Does It Work (Plain Language)?

Think of TimSort as a smart librarian:

- Sees which shelves (runs) are already sorted
- Tidies up small messy shelves (Insertion Sort)
- Merges shelves together efficiently (Merge Sort)
- Uses stack rules to decide merge timing for balance

It's adaptive, stable, and real-world optimized.

## Step-by-Step Process

| Step | Description                                                       |
|------|-------------------------------------------------------------------|
| 1    | Scan the array to find runs (ascending or descending)             |
| 2    | Reverse descending runs                                           |
| 3    | If run length < <code>minrun</code> , extend using Insertion Sort |
| 4    | Push runs onto stack                                              |
| 5    | Merge runs when stack size conditions are violated                |
| 6    | Continue until one run remains (fully sorted)                     |

## Tiny Code (Easy Versions)

### Python (Simplified Simulation)

```
def insertion_sort(arr, left, right):
    for i in range(left + 1, right + 1):
        key = arr[i]
        j = i - 1
        while j >= left and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

def merge(arr, left, mid, right):
    left_part = arr[left:mid+1]
    right_part = arr[mid+1:right+1]
    i = j = 0
    k = left
```

```

while i < len(left_part) and j < len(right_part):
    if left_part[i] <= right_part[j]:
        arr[k] = left_part[i]
        i += 1
    else:
        arr[k] = right_part[j]
        j += 1
    k += 1
while i < len(left_part):
    arr[k] = left_part[i]; i += 1; k += 1
while j < len(right_part):
    arr[k] = right_part[j]; j += 1; k += 1

def timsort(arr):
    n = len(arr)
    minrun = 32

    # Sort small runs using insertion sort
    for start in range(0, n, minrun):
        end = min(start + minrun - 1, n - 1)
        insertion_sort(arr, start, end)

    size = minrun
    while size < n:
        for left in range(0, n, 2 * size):
            mid = min(n - 1, left + size - 1)
            right = min(n - 1, left + 2 * size - 1)
            if mid < right:
                merge(arr, left, mid, right)
        size *= 2

arr = [5,6,7,1,2,3,8,9]
timsort(arr)
print(arr)

```

Output:

[\$\$1, 2, 3, 5, 6, 7, 8, 9]

## C (Conceptual Version)

```
#include <stdio.h>
#include <stdlib.h>

#define MINRUN 32

void insertion_sort(int arr[], int left, int right) {
    for (int i = left + 1; i <= right; i++) {
        int key = arr[i], j = i - 1;
        while (j >= left && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1, n2 = r - m;
    int *L = malloc(n1 * sizeof(int)), *R = malloc(n2 * sizeof(int));
    for (int i = 0; i < n1; i++) L[i] = arr[l + i];
    for (int j = 0; j < n2; j++) R[j] = arr[m + 1 + j];
    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
    free(L); free(R);
}

void timsort(int arr[], int n) {
    for (int i = 0; i < n; i += MINRUN) {
        int end = (i + MINRUN - 1 < n) ? (i + MINRUN - 1) : (n - 1);
        insertion_sort(arr, i, end);
    }
    for (int size = MINRUN; size < n; size *= 2) {
        for (int left = 0; left < n; left += 2 * size) {
            int mid = (left + size - 1 < n - 1) ? (left + size - 1) : (n - 1);
            int right = (left + 2 * size - 1 < n - 1) ? (left + 2 * size - 1) : (n - 1);
            if (mid < right) merge(arr, left, mid, right);
        }
    }
}
```

```

}

int main(void) {
    int arr[] = {5,6,7,1,2,3,8,9};
    int n = sizeof(arr)/sizeof(arr[0]);
    timsort(arr, n);
    for (int i = 0; i < n; i++) printf("%d ", arr[i]);
    printf("\n");
}

```

## Why It Matters

- Default sort in Python & Java
- Stable and adaptive
- $O(n)$  on sorted data
- $O(n \log n)$  worst case
- Handles real-world inputs gracefully

It's the perfect sort when you don't know the data shape, it *adapts itself*.

## A Gentle Proof (Why It Works)

If data already contains sorted runs of average length  $r$ :

- Insertion Sort:  $O(r^2)$  per run (tiny)
- Merging  $(n/r)$  runs:  $O(n \log(n/r))$

Overall:

$$T(n) = O(n + n \log(n/r))$$

For  $r \approx n$ :  $O(n)$

For  $r = 1$ :  $O(n \log n)$

| Phase         | Work             | Complexity      |
|---------------|------------------|-----------------|
| Run Detection | $O(n)$           | One pass        |
| Local Sorting | $O(r^2)$ per run | Tiny runs       |
| Merge Phase   | $O(n \log n)$    | Balanced merges |

## Try It Yourself

1. Sort  $[5, 6, 7, 1, 2, 3, 8, 9]$  step by step.
2. Detect natural runs manually.
3. Reverse descending runs before merge.
4. Adjust  $\text{minrun} = 16$ , see difference.
5. Test  $[1, 2, 3, 4, 5]$ , should take  $O(n)$ .
6. Add random noise to partially sorted list.
7. Track stack of runs, when to merge?
8. Compare performance with Merge Sort.
9. Visualize merge order tree.
10. Check stability with duplicate keys.

## Test Cases

| Input                      | Output                     | Notes          |
|----------------------------|----------------------------|----------------|
| $[5, 6, 7, 1, 2, 3, 8, 9]$ | $[1, 2, 3, 5, 6, 7, 8, 9]$ | Mixed runs     |
| $[1, 2, 3, 4, 5]$          | $[1, 2, 3, 4, 5]$          | Already sorted |
| $[9, 8, 7, 6]$             | $[6, 7, 8, 9]$             | Reverse run    |
| $[5, 5, 5, 5]$             | $[5, 5, 5, 5]$             | Stability test |

## Complexity

| Aspect         | Value         |
|----------------|---------------|
| Time (Best)    | $O(n)$        |
| Time (Average) | $O(n \log n)$ |
| Time (Worst)   | $O(n \log n)$ |
| Space          | $O(n)$        |
| Stable         | Yes           |
| Adaptive       | Yes           |

TimSort is the real-world champion, it watches your data, adapts instantly, and sorts smarter, not harder. It's the kind of algorithm that doesn't just run fast, it *thinks* fast.

## 133 Dual-Pivot QuickSort

Dual-Pivot QuickSort is an enhanced variant of QuickSort that uses two pivots instead of one to partition the array into three regions:

- Elements less than pivot1,
- Elements between pivot1 and pivot2,
- Elements greater than pivot2.

This approach often reduces comparisons and improves cache efficiency. It was popularized by Vladimir Yaroslavskiy and became the default sorting algorithm in Java (from Java 7) for primitive types.

### **What Problem Are We Solving??**

Standard QuickSort splits the array into two parts using one pivot. Dual-Pivot QuickSort splits into three, reducing recursion depth and overhead.

It's optimized for:

- Large arrays of primitives (integers, floats)
- Random and uniform data
- Modern CPUs with deep pipelines and caches

It offers better real-world performance than classic QuickSort, even if asymptotic complexity remains  $O(n \log n)$ .

### **Example**

Sort [9, 3, 1, 7, 5, 4, 6, 2, 8]

Choose pivots ( $p_1 = 3, p_2 = 7$ ):

| Region | Condition         | Elements        |
|--------|-------------------|-----------------|
| Left   | $x < 3$           | [1, 2]          |
| Middle | $3 \leq x \leq 7$ | [3, 4, 5, 6, 7] |
| Right  | $x > 7$           | [8, 9]          |

Recurse on each region.

Final result: [1, 2, 3, 4, 5, 6, 7, 8, 9].

## How Does It Work (Plain Language)?

Imagine sorting shoes by size with two markers:

- Small shelf for sizes less than 7
- Middle shelf for 7–9
- Big shelf for 10+

You walk through once, placing each shoe in the right group, then sort each shelf individually.

Dual-Pivot QuickSort does exactly that: partition into three zones in one pass, then recurse.

## Step-by-Step Process

| Step | Description                                                 |
|------|-------------------------------------------------------------|
| 1    | Choose two pivots ( $p_1, p_2$ ), ensuring $p_1 < p_2$      |
| 2    | Partition array into 3 parts: $< p_1, p_1 \dots p_2, > p_2$ |
| 3    | Recursively sort each part                                  |
| 4    | Combine results in order                                    |

If  $p_1 > p_2$ , swap them first.

## Tiny Code (Easy Versions)

C

```
#include <stdio.h>

void swap(int *a, int *b) {
    int tmp = *a; *a = *b; *b = tmp;
}

void dual_pivot_quicksort(int arr[], int low, int high) {
    if (low >= high) return;

    if (arr[low] > arr[high]) swap(&arr[low], &arr[high]);
    int p1 = arr[low], p2 = arr[high];

    int lt = low + 1, gt = high - 1, i = low + 1;
```

```

while (i <= gt) {
    if (arr[i] < p1) {
        swap(&arr[i], &arr[lt]);
        lt++; i++;
    } else if (arr[i] > p2) {
        swap(&arr[i], &arr[gt]);
        gt--;
    } else {
        i++;
    }
}
lt--; gt++;
swap(&arr[low], &arr[lt]);
swap(&arr[high], &arr[gt]);

dual_pivot_quicksort(arr, low, lt - 1);
dual_pivot_quicksort(arr, lt + 1, gt - 1);
dual_pivot_quicksort(arr, gt + 1, high);
}

int main(void) {
    int arr[] = {9,3,1,7,5,4,6,2,8};
    int n = sizeof(arr)/sizeof(arr[0]);
    dual_pivot_quicksort(arr, 0, n-1);
    for (int i = 0; i < n; i++) printf("%d ", arr[i]);
    printf("\n");
}

```

## Python

```

def dual_pivot_quicksort(arr):
    def sort(low, high):
        if low >= high:
            return
        if arr[low] > arr[high]:
            arr[low], arr[high] = arr[high], arr[low]
        p1, p2 = arr[low], arr[high]
        lt, gt, i = low + 1, high - 1, low + 1
        while i <= gt:
            if arr[i] < p1:
                arr[i], arr[lt] = arr[lt], arr[i]

```

```

        lt += 1; i += 1
    elif arr[i] > p2:
        arr[i], arr[gt] = arr[gt], arr[i]
        gt -= 1
    else:
        i += 1
    lt -= 1; gt += 1
    arr[low], arr[lt] = arr[lt], arr[low]
    arr[high], arr[gt] = arr[gt], arr[high]
    sort(low, lt - 1)
    sort(lt + 1, gt - 1)
    sort(gt + 1, high)
sort(0, len(arr) - 1)
return arr

arr = [9,3,1,7,5,4,6,2,8]
print(dual_pivot_quicksort(arr))

```

## Why It Matters

- Default in Java for primitives
- Fewer comparisons than single-pivot QuickSort
- Cache-friendly (less branching)
- Stable recursion depth with three partitions

## A Gentle Proof (Why It Works)

Each partitioning step processes all elements once,  $O(n)$ . Recursion on three smaller subarrays yields total cost:

$$T(n) = T(k_1) + T(k_2) + T(k_3) + O(n)$$

On average, partitions are balanced  $\rightarrow T(n) = O(n \log n)$

| Phase        | Operation     | Complexity      |
|--------------|---------------|-----------------|
| Partitioning | $O(n)$        | One pass        |
| Recursion    | 3 subarrays   | Balanced depth  |
| Total        | $O(n \log n)$ | Average / Worst |

## Try It Yourself

1. Sort [9,3,1,7,5,4,6,2,8] step by step.
2. Choose pivots manually: smallest and largest.
3. Trace index movements (lt, gt, i).
4. Compare with classic QuickSort partition count.
5. Use reversed array, observe stability.
6. Add duplicates [5,5,5,5], see middle zone effect.
7. Measure comparisons vs single pivot.
8. Try large input (10 ) and time it.
9. Visualize three partitions recursively.
10. Implement tail recursion optimization.

## Test Cases

| Input               | Output              | Notes          |
|---------------------|---------------------|----------------|
| [9,3,1,7,5,4,6,2,8] | [1,2,3,4,5,6,7,8,9] | Example        |
| [1,2,3,4]           | [1,2,3,4]           | Already sorted |
| [9,8,7,6,5]         | [5,6,7,8,9]         | Reverse        |
| [5,5,5,5]           | [5,5,5,5]           | Duplicates     |

## Complexity

| Aspect         | Value         |
|----------------|---------------|
| Time (Best)    | $O(n \log n)$ |
| Time (Average) | $O(n \log n)$ |
| Time (Worst)   | $O(n \log n)$ |
| Space          | $O(\log n)$   |
| Stable         | No            |
| Adaptive       | No            |

Dual-Pivot QuickSort slices data with two blades instead of one, making each cut smaller, shallower, and more balanced. A sharper, smarter evolution of a timeless classic.

## 134 SmoothSort

SmoothSort is an adaptive comparison-based sorting algorithm invented by Edsger Dijkstra. It's similar in spirit to Heap Sort, but smarter, it runs in  $O(n)$  time on already sorted data and  $O(n \log n)$  in the worst case.

The key idea is to build a special heap structure (Leonardo heap) that adapts to existing order in the data. When the array is nearly sorted, it finishes quickly. When not, it gracefully falls back to heap-like performance.

### What Problem Are We Solving??

Heap Sort always works in  $O(n \log n)$ , even when the input is already sorted. SmoothSort improves on this by being adaptive, the more ordered the input, the faster it gets.

It's ideal for:

- Nearly sorted arrays
- Situations requiring guaranteed upper bounds
- Memory-constrained environments (in-place sort)

### Example

Sort [1, 2, 4, 3, 5]

| Step | Action                                  | Result      |
|------|-----------------------------------------|-------------|
| 1    | Build initial heap (Leonardo structure) | [1,2,4,3,5] |
| 2    | Detect small disorder (4,3)             | Swap        |
| 3    | Restore heap property                   | [1,2,3,4,5] |
| 4    | Sorted early, no full rebuild needed    | Done        |

Result: finished early since only minor disorder existed.

### How Does It Work (Plain Language)?

Imagine a stack of heaps, each representing a Fibonacci-like sequence (Leonardo numbers). You grow this structure as you read the array, maintaining order locally. When disorder appears, you fix only where needed, not everywhere.

So SmoothSort is like a gentle gardener: it only trims where weeds grow, not the whole garden.

## Step-by-Step Process

| Step | Description                                                    |
|------|----------------------------------------------------------------|
| 1    | Represent the array as a series of Leonardo heaps              |
| 2    | Add elements one by one, updating the heap sequence            |
| 3    | Maintain heap property with minimal swaps                      |
| 4    | When finished, repeatedly extract the maximum (like Heap Sort) |
| 5    | During extraction, merge smaller heaps as needed               |

Leonardo numbers guide heap sizes: (  $L(0)=1$ ,  $L(1)=1$ ,  $L(n)=L(n-1)+L(n-2)+1$  )

## Tiny Code (Easy Versions)

### Python (Simplified Adaptive Sort)

This is a simplified version to show adaptiveness, not a full Leonardo heap implementation.

```
def smoothsort(arr):
    # Simplified: detect sorted runs, fix only where needed
    n = len(arr)
    for i in range(1, n):
        j = i
        while j > 0 and arr[j] < arr[j - 1]:
            arr[j], arr[j - 1] = arr[j - 1], arr[j]
            j -= 1
    return arr

arr = [1, 2, 4, 3, 5]
print(smoothsort(arr))
```

Output:

[\$\$1, 2, 3, 4, 5]

(Note: Real SmoothSort uses Leonardo heaps, complex but in-place and efficient.)

## C (Conceptual Heap Approach)

```

#include <stdio.h>

void insertion_like_fix(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int j = i;
        while (j > 0 && arr[j] < arr[j - 1]) {
            int tmp = arr[j];
            arr[j] = arr[j - 1];
            arr[j - 1] = tmp;
            j--;
        }
    }
}

int main(void) {
    int arr[] = {1, 2, 4, 3, 5};
    int n = sizeof(arr)/sizeof(arr[0]);
    insertion_like_fix(arr, n);
    for (int i = 0; i < n; i++) printf("%d ", arr[i]);
    printf("\n");
}

```

This mimics SmoothSort's adaptiveness: fix locally, not globally.

## Why It Matters

- Adaptive: faster on nearly sorted data
- In-place: no extra memory
- Guaranteed bound: never worse than  $O(n \log n)$
- Historical gem: Dijkstra's innovation in sorting theory

## A Gentle Proof (Why It Works)

For sorted input:

- Each insertion requires no swaps  $\rightarrow O(n)$

For random input:

- Heap restorations per insertion  $\rightarrow O(\log n)$

- Total cost:  $O(n \log n)$

| Case          | Behavior           | Complexity    |
|---------------|--------------------|---------------|
| Best (Sorted) | Minimal swaps      | $O(n)$        |
| Average       | Moderate reheapify | $O(n \log n)$ |
| Worst         | Full heap rebuilds | $O(n \log n)$ |

SmoothSort adapts between these seamlessly.

### Try It Yourself

1. Sort [1, 2, 4, 3, 5] step by step.
2. Try [1, 2, 3, 4, 5], measure comparisons.
3. Try [5, 4, 3, 2, 1], full workload.
4. Count swaps in each case.
5. Compare to Heap Sort.
6. Visualize heap sizes as Leonardo sequence.
7. Implement run detection.
8. Experiment with large partially sorted arrays.
9. Track adaptive speedup.
10. Write Leonardo heap builder.

### Test Cases

| Input       | Output      | Notes          |
|-------------|-------------|----------------|
| [1,2,4,3,5] | [1,2,3,4,5] | Minor disorder |
| [1,2,3,4,5] | [1,2,3,4,5] | Already sorted |
| [5,4,3,2,1] | [1,2,3,4,5] | Full rebuild   |
| [2,1,3,5,4] | [1,2,3,4,5] | Mixed case     |

### Complexity

| Aspect         | Value         |
|----------------|---------------|
| Time (Best)    | $O(n)$        |
| Time (Average) | $O(n \log n)$ |
| Time (Worst)   | $O(n \log n)$ |
| Space          | $O(1)$        |

| Aspect   | Value |
|----------|-------|
| Stable   | No    |
| Adaptive | Yes   |

SmoothSort glides gracefully across the array, fixing only what's broken. It's sorting that *listens* to your data, quiet, clever, and precise.

## 135 Block Merge Sort

Block Merge Sort is a cache-efficient, stable sorting algorithm that merges data using small fixed-size blocks instead of large temporary arrays. It improves on standard Merge Sort by reducing memory usage and enhancing locality of reference, making it a great choice for modern hardware and large datasets.

It's designed to keep data cache-friendly, in-place (or nearly), and stable, making it a practical choice for systems with limited memory bandwidth or tight memory constraints.

### What Problem Are We Solving??

Classic Merge Sort is stable and  $O(n \log n)$ , but it needs  $O(n)$  extra space. Block Merge Sort solves this by using blocks of fixed size (often  $\sqrt{n}$ ) as temporary buffers for merging.

It aims to:

- Keep stability
- Use limited extra memory
- Maximize cache reuse
- Maintain predictable access patterns

Ideal for:

- Large arrays
- External memory sorting
- Systems with cache hierarchies

### Example

Sort [8, 3, 5, 1, 6, 2, 7, 4]

| Step | Action                                       | Result                     |
|------|----------------------------------------------|----------------------------|
| 1    | Divide into sorted runs (via insertion sort) | [3,8], [1,5], [2,6], [4,7] |
| 2    | Merge adjacent blocks using buffer           | [1,3,5,8], [2,4,6,7]       |
| 3    | Merge merged blocks with block buffer        | [1,2,3,4,5,6,7,8]          |

Instead of full arrays, it uses small block buffers, fewer cache misses, less extra space.

### How Does It Work (Plain Language)?

Think of merging two sorted shelves in a library, but instead of taking all books off, you move a small block at a time, swapping them in place with a small temporary cart.

You slide the buffer along the shelves, merge gradually, efficiently, with minimal movement.

### Step-by-Step Process

| Step | Description                                             |
|------|---------------------------------------------------------|
| 1    | Divide input into runs (sorted subarrays)               |
| 2    | Use insertion sort or binary insertion to sort each run |
| 3    | Allocate small buffer (block)                           |
| 4    | Merge runs pairwise using the block buffer              |
| 5    | Repeat until one sorted array remains                   |

Block merges rely on rotations and buffer swapping to minimize extra space.

### Tiny Code (Easy Versions)

#### Python (Simplified Version)

This version simulates block merging using chunks.

```
def insertion_sort(arr, start, end):
    for i in range(start + 1, end):
        key = arr[i]
        j = i - 1
        while j >= start and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
```

```

arr[j + 1] = key

def merge(arr, left, mid, right):
    left_part = arr[left:mid]
    right_part = arr[mid:right]
    i = j = 0
    k = left
    while i < len(left_part) and j < len(right_part):
        if left_part[i] <= right_part[j]:
            arr[k] = left_part[i]; i += 1
        else:
            arr[k] = right_part[j]; j += 1
        k += 1
    while i < len(left_part):
        arr[k] = left_part[i]; i += 1; k += 1
    while j < len(right_part):
        arr[k] = right_part[j]; j += 1; k += 1

def block_merge_sort(arr, block_size=32):
    n = len(arr)
    # Step 1: sort small blocks
    for start in range(0, n, block_size):
        end = min(start + block_size, n)
        insertion_sort(arr, start, end)

    # Step 2: merge adjacent blocks
    size = block_size
    while size < n:
        for left in range(0, n, 2 * size):
            mid = min(left + size, n)
            right = min(left + 2 * size, n)
            merge(arr, left, mid, right)
        size *= 2
    return arr

arr = [8, 3, 5, 1, 6, 2, 7, 4]
print(block_merge_sort(arr))

```

Output:

[\$\$1, 2, 3, 4, 5, 6, 7, 8]

## C (Simplified Concept)

```
#include <stdio.h>

void insertion_sort(int arr[], int left, int right) {
    for (int i = left + 1; i < right; i++) {
        int key = arr[i], j = i - 1;
        while (j >= left && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left, n2 = right - mid;
    int L[n1], R[n2];
    for (int i = 0; i < n1; i++) L[i] = arr[left + i];
    for (int j = 0; j < n2; j++) R[j] = arr[mid + j];

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}

void block_merge_sort(int arr[], int n, int block_size) {
    for (int i = 0; i < n; i += block_size) {
        int end = (i + block_size < n) ? i + block_size : n;
        insertion_sort(arr, i, end);
    }
    for (int size = block_size; size < n; size *= 2) {
        for (int left = 0; left < n; left += 2 * size) {
            int mid = (left + size < n) ? left + size : n;
            int right = (left + 2 * size < n) ? left + 2 * size : n;
            if (mid < right) merge(arr, left, mid, right);
        }
    }
}

int main(void) {
```

```

int arr[] = {8,3,5,1,6,2,7,4};
int n = sizeof(arr)/sizeof(arr[0]);
block_merge_sort(arr, n, 2);
for (int i = 0; i < n; i++) printf("%d ", arr[i]);
printf("\n");
}

```

## Why It Matters

- Stable like Merge Sort
- In-place or low-space variant
- Cache-efficient due to block locality
- Practical for large arrays or external sorting

## A Gentle Proof (Why It Works)

Each merge level processes  $n$  elements:  $O(n)$ .

Number of levels =  $\log_2(n/b)$ , where  $b$  is the block size.

So total:

$$T(n) = O(n \log(n/b))$$

When  $b$  is large (like  $\sqrt{n}$ ), space and time balance nicely.

| Phase         | Operation                | Cost          |
|---------------|--------------------------|---------------|
| Block sorting | $\frac{n}{b} \times b^2$ | $O(nb)$       |
| Merging       | $\log(n/b) \times n$     | $O(n \log n)$ |

For typical settings, it's near  $O(n \log n)$  but cache-optimized.

## Try It Yourself

1. Sort [8,3,5,1,6,2,7,4] with block size 2.
2. Increase block size to 4, compare steps.
3. Track number of merges.
4. Check stability with duplicates.
5. Compare with standard Merge Sort.
6. Time on sorted input (adaptive check).
7. Measure cache misses (simulated).

8. Try large array (10000+) for memory gain.
9. Mix ascending and descending runs.
10. Implement block buffer rotation manually.

## Test Cases

| Input             | Output            | Notes         |
|-------------------|-------------------|---------------|
| [8,3,5,1,6,2,7,4] | [1,2,3,4,5,6,7,8] | Classic       |
| [5,5,3,3,1]       | [1,3,3,5,5]       | Stable        |
| [1,2,3,4]         | [1,2,3,4]         | Sorted        |
| [9,8,7]           | [7,8,9]           | Reverse order |

## Complexity

| Aspect         | Value         |
|----------------|---------------|
| Time (Best)    | $O(n)$        |
| Time (Average) | $O(n \log n)$ |
| Time (Worst)   | $O(n \log n)$ |
| Space          | $O(b)$        |
| Stable         | Yes           |
| Adaptive       | Partially     |

Block Merge Sort is the engineer's Merge Sort, same elegance, less memory, smarter on hardware. It merges not by brute force, but by careful block juggling, balancing speed, space, and stability.

## 136 Adaptive Merge Sort

Adaptive Merge Sort is a stable, comparison-based sorting algorithm that adapts to existing order in the input data. It builds on the idea that real-world datasets are often partially sorted, so by detecting runs (already sorted sequences) and merging them intelligently, it can achieve  $O(n)$  time on nearly sorted data while retaining  $O(n \log n)$  in the worst case.

It's a family of algorithms, including Natural Merge Sort, TimSort, and GrailSort, that all share one key insight: work less when less work is needed.

## **What Problem Are We Solving??**

Standard Merge Sort treats every input the same, even if it's already sorted. Adaptive Merge Sort improves this by:

- Detecting sorted runs (ascending or descending)
- Merging runs instead of single elements
- Achieving linear time on sorted or partially sorted data

This makes it perfect for:

- Time series data
- Sorted or semi-sorted logs
- Incrementally updated lists

## **Example**

Sort [1, 2, 5, 3, 4, 6]

| Step | Action                        | Result        |
|------|-------------------------------|---------------|
| 1    | Detect runs: [1,2,5], [3,4,6] | Found 2 runs  |
| 2    | Merge runs                    | [1,2,3,4,5,6] |
| 3    | Done                          | Sorted        |

Only one merge needed, input was nearly sorted, so runtime is close to  $O(n)$ .

## **How Does It Work (Plain Language)?**

Imagine you're sorting a shelf of books that's mostly organized — you don't pull all the books off; you just spot where order breaks, and fix those parts.

Adaptive Merge Sort does exactly this:

- Scan once to find sorted parts
- Merge runs using a stable merge procedure

It's lazy where it can be, efficient where it must be.

## **Step-by-Step Process**

| Step | Description                                                   |
|------|---------------------------------------------------------------|
| 1    | Scan input to find runs (ascending or descending)             |
| 2    | Reverse descending runs                                       |
| 3    | Push runs onto a stack                                        |
| 4    | Merge runs when stack conditions are violated (size or order) |
| 5    | Continue until one sorted run remains                         |

## Tiny Code (Easy Versions)

### Python (Natural Merge Sort)

```

def natural_merge_sort(arr):
    n = len(arr)
    runs = []
    i = 0
    # Step 1: Detect sorted runs
    while i < n:
        start = i
        i += 1
        while i < n and arr[i] >= arr[i - 1]:
            i += 1
        runs.append(arr[start:i])
    # Step 2: Merge runs pairwise
    while len(runs) > 1:
        new_runs = []
        for j in range(0, len(runs), 2):
            if j + 1 < len(runs):
                new_runs.append(merge(runs[j], runs[j+1]))
            else:
                new_runs.append(runs[j])
        runs = new_runs
    return runs[0]

def merge(left, right):
    i = j = 0
    result = []
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i]); i += 1
        else:

```

```

        result.append(right[j]); j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result

arr = [1, 2, 5, 3, 4, 6]
print(natural_merge_sort(arr))

```

Output:

```
$$1, 2, 3, 4, 5, 6]
```

## C (Simplified Conceptual)

```

#include <stdio.h>
#include <stdlib.h>

void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1, n2 = r - m;
    int L[n1], R[n2];
    for (int i = 0; i < n1; i++) L[i] = arr[l + i];
    for (int j = 0; j < n2; j++) R[j] = arr[m + 1 + j];
    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}

void adaptive_merge_sort(int arr[], int n) {
    int start = 0;
    while (start < n - 1) {
        int mid = start;
        while (mid < n - 1 && arr[mid] <= arr[mid + 1]) mid++;
        int end = mid + 1;
        while (end < n - 1 && arr[end] <= arr[end + 1]) end++;
        if (end < n) merge(arr, start, mid, end);
        start = end + 1;
    }
}

```

```

int main(void) {
    int arr[] = {1, 2, 5, 3, 4, 6};
    int n = sizeof(arr)/sizeof(arr[0]);
    adaptive_merge_sort(arr, n);
    for (int i = 0; i < n; i++) printf("%d ", arr[i]);
    printf("\n");
}

```

## Why It Matters

- Stable and adaptive
- Linear time on nearly sorted data
- Works well for real-world sequences
- Forms the core idea behind TimSort
- Requires no extra knowledge about data

It's the sorting algorithm that notices your data's effort and rewards it.

## A Gentle Proof (Why It Works)

Let average run length =  $r$ .

Then number of runs  $n/r$ .

Each merge =  $O(n)$  per level.

Depth =  $O(\log(n/r))$ .

So total cost:

$$T(n) = O(n \log(n/r))$$

If  $r = n$  (already sorted):  $O(n)$ .

If  $r = 1$ :  $O(n \log n)$ .

| Case          | Run Length | Complexity       |
|---------------|------------|------------------|
| Sorted        | $r = n$    | $O(n)$           |
| Nearly sorted | $r$ large  | $O(n \log(n/r))$ |
| Random        | $r$ small  | $O(n \log n)$    |

## Try It Yourself

1. Sort [1, 2, 5, 3, 4, 6] step by step.
2. Try [1, 2, 3, 4, 5], should detect 1 run.
3. Reverse a section, see new runs.
4. Merge runs manually using table.
5. Compare performance to Merge Sort.
6. Add duplicates, check stability.
7. Use 10k sorted elements, time the run.
8. Mix ascending and descending subarrays.
9. Visualize run detection.
10. Implement descending-run reversal.

## Test Cases

| Input         | Output        | Notes               |
|---------------|---------------|---------------------|
| [1,2,5,3,4,6] | [1,2,3,4,5,6] | Two runs            |
| [1,2,3,4,5]   | [1,2,3,4,5]   | Already sorted      |
| [5,4,3,2,1]   | [1,2,3,4,5]   | Reverse (many runs) |
| [2,2,1,1]     | [1,1,2,2]     | Stable              |

## Complexity

| Aspect         | Value         |
|----------------|---------------|
| Time (Best)    | $O(n)$        |
| Time (Average) | $O(n \log n)$ |
| Time (Worst)   | $O(n \log n)$ |
| Space          | $O(n)$        |
| Stable         | Yes           |
| Adaptive       | Yes           |

Adaptive Merge Sort is like a thoughtful sorter, it looks before it leaps. If your data's halfway there, it'll meet it in the middle.

## 137 PDQSort (Pattern-Defeating QuickSort)

PDQSort is a modern, adaptive, in-place sorting algorithm that extends QuickSort with pattern detection, branchless partitioning, and fallback mechanisms to guarantee  $O(n \log n)$  performance even on adversarial inputs.

Invented by Orson Peters, it's used in C++'s `std::sort()` (since C++17) and often outperforms traditional QuickSort and IntroSort in real-world scenarios due to better cache behavior, branch prediction, and adaptive pivoting.

### What Problem Are We Solving??

Classic QuickSort performs well *on average* but can degrade to  $(O(n^2))$  on structured or repetitive data. PDQSort solves this by:

- Detecting bad patterns (e.g., sorted input)
- Switching strategy (to heap sort or insertion sort)
- Branchless partitioning for modern CPUs
- Adaptive pivot selection (median-of-3, Tukey ninther)

It keeps speed, stability of performance, and cache-friendliness.

Perfect for:

- Large unsorted datasets
- Partially sorted data
- Real-world data with patterns

### Example

Sort [1, 2, 3, 4, 5] (already sorted)

| Step | Action                   | Result             |
|------|--------------------------|--------------------|
| 1    | Detect sorted pattern    | Pattern found      |
| 2    | Switch to Insertion Sort | Efficient handling |
| 3    | Output sorted            | [1,2,3,4,5]        |

Instead of recursive QuickSort calls, PDQSort defeats the pattern by adapting.

## How Does It Work (Plain Language)?

PDQSort is like a smart chef:

- It tastes the input first (checks pattern)
- Chooses a recipe (pivot rule, sorting fallback)
- Adjusts to the kitchen conditions (CPU caching, branching)

It never wastes effort, detecting when recursion or comparisons are unnecessary.

## Step-by-Step Process

| Step | Description                                                 |
|------|-------------------------------------------------------------|
| 1    | Pick pivot adaptively (median-of-3, ninther)                |
| 2    | Partition elements into < pivot and > pivot                 |
| 3    | Detect bad patterns (sorted, reversed, many equal elements) |
| 4    | Apply branchless partitioning to reduce CPU mispredictions  |
| 5    | Recurse or switch to heap/insertion sort if needed          |
| 6    | Use tail recursion elimination                              |

## Tiny Code (Easy Versions)

### Python (Conceptual Simplified PDQSort)

```
def pdqsort(arr):
    def insertion_sort(a, lo, hi):
        for i in range(lo + 1, hi):
            key = a[i]
            j = i - 1
            while j >= lo and a[j] > key:
                a[j + 1] = a[j]
                j -= 1
            a[j + 1] = key

    def partition(a, lo, hi):
        pivot = a[(lo + hi) // 2]
        i, j = lo, hi - 1
        while True:
            while a[i] < pivot: i += 1
```

```

        while a[j] > pivot: j -= 1
        if i >= j: return j
        a[i], a[j] = a[j], a[i]
        i += 1; j -= 1

def _pdqsort(a, lo, hi, depth):
    n = hi - lo
    if n <= 16:
        insertion_sort(a, lo, hi)
        return
    if depth == 0:
        a[lo:hi] = sorted(a[lo:hi]) # heap fallback
        return
    mid = partition(a, lo, hi)
    _pdqsort(a, lo, mid + 1, depth - 1)
    _pdqsort(a, mid + 1, hi, depth - 1)

_pdqsort(arr, 0, len(arr), len(arr).bit_length() * 2)
return arr

arr = [1, 5, 3, 4, 2]
print(pdqsort(arr))

```

Output:

[\$\$1, 2, 3, 4, 5]

## C (Simplified PDQ-Style)

```

#include <stdio.h>
#include <stdlib.h>

void insertion_sort(int arr[], int lo, int hi) {
    for (int i = lo + 1; i < hi; i++) {
        int key = arr[i], j = i - 1;
        while (j >= lo && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

```

```

    }

}

int partition(int arr[], int lo, int hi) {
    int pivot = arr[(lo + hi) / 2];
    int i = lo, j = hi - 1;
    while (1) {
        while (arr[i] < pivot) i++;
        while (arr[j] > pivot) j--;
        if (i >= j) return j;
        int tmp = arr[i]; arr[i] = arr[j]; arr[j] = tmp;
        i++; j--;
    }
}

void pdqsort(int arr[], int lo, int hi, int depth) {
    int n = hi - lo;
    if (n <= 16) { insertion_sort(arr, lo, hi); return; }
    if (depth == 0) { qsort(arr + lo, n, sizeof(int), (__compar_fn_t)strcmp); return; }
    int mid = partition(arr, lo, hi);
    pdqsort(arr, lo, mid + 1, depth - 1);
    pdqsort(arr, mid + 1, hi, depth - 1);
}

int main(void) {
    int arr[] = {1, 5, 3, 4, 2};
    int n = sizeof(arr)/sizeof(arr[0]);
    pdqsort(arr, 0, n, 2 * 32);
    for (int i = 0; i < n; i++) printf("%d ", arr[i]);
    printf("\n");
}

```

## Why It Matters

- Modern default for high-performance sorting
- Pattern detection avoids worst cases
- Branchless partitioning → fewer CPU stalls
- Heap fallback ensures  $O(n \log n)$  bound
- Adaptive like TimSort, but in-place

PDQSort combines speed, safety, and hardware awareness.

## A Gentle Proof (Why It Works)

PDQSort adds mechanisms to defeat QuickSort's pitfalls:

1. Bad pattern detection → early fallback
2. Balanced pivoting → near-equal splits
3. Branchless operations → efficient execution

So overall:

$$T(n) = O(n \log n)$$

Best case (sorted): near  $O(n)$ , thanks to early detection.

| Case    | Behavior                | Complexity    |
|---------|-------------------------|---------------|
| Best    | Sorted or nearly sorted | $O(n)$        |
| Average | Random                  | $O(n \log n)$ |
| Worst   | Adversarial             | $O(n \log n)$ |

## Try It Yourself

1. Sort  $[1,2,3,4,5]$  → detect sorted input.
2. Try  $[5,4,3,2,1]$  → reversed.
3. Random input of 1000 numbers.
4. Duplicate-heavy  $[5,5,5,5]$ .
5. Patterned input  $[1,3,2,4,3,5,4]$ .
6. Compare with QuickSort and HeapSort.
7. Count recursion depth.
8. Benchmark branchless vs classic partition.
9. Visualize fallback triggers.
10. Measure comparisons per element.

## Test Cases

| Input         | Output        | Notes      |
|---------------|---------------|------------|
| $[1,5,3,4,2]$ | $[1,2,3,4,5]$ | Random     |
| $[1,2,3,4,5]$ | $[1,2,3,4,5]$ | Sorted     |
| $[5,4,3,2,1]$ | $[1,2,3,4,5]$ | Reversed   |
| $[5,5,5,5]$   | $[5,5,5,5]$   | Duplicates |

## Complexity

| Aspect         | Value         |
|----------------|---------------|
| Time (Best)    | $O(n)$        |
| Time (Average) | $O(n \log n)$ |
| Time (Worst)   | $O(n \log n)$ |
| Space          | $O(\log n)$   |
| Stable         | No            |
| Adaptive       | Yes           |

PDQSort is the ninja of sorting, lightning fast, pattern-aware, and always one step ahead of your data. It doesn't just sort, it *outsmarts* the input.

## 138 WikiSort

WikiSort is a stable, in-place merge sort created by Mike Day, designed to combine the stability of Merge Sort with the low memory usage of in-place algorithms. It achieves  $O(n \log n)$  performance and uses only  $O(1)$  extra memory, making it one of the most space-efficient stable sorts available.

Unlike classic Merge Sort, which allocates a full-size temporary array, WikiSort performs block merges with rotation operations, merging sorted regions directly within the array.

### What Problem Are We Solving??

Most stable sorting algorithms (like Merge Sort) need  $O(n)$  extra space. WikiSort solves this by performing stable merges in place, using:

- Block rotations instead of large buffers
- Adaptive merging when possible
- Small local buffers reused efficiently

Perfect for:

- Memory-constrained environments
- Sorting large arrays in embedded systems
- Stable sorts without heavy allocation

## Example

Sort [3, 5, 1, 2, 4]

| Step | Action                      | Result             |
|------|-----------------------------|--------------------|
| 1    | Divide into sorted runs     | [3,5], [1,2,4]     |
| 2    | Merge using block rotations | [1,2,3,4,5]        |
| 3    | Done, stable and in-place   | Final sorted array |

Result: [1,2,3,4,5], sorted, stable, and minimal extra space.

## How Does It Work (Plain Language)?

Think of two sorted shelves of books, instead of moving all books to a table, you rotate sections in place so the shelves merge seamlessly.

WikiSort keeps a small local buffer (like a tray), uses it to move small chunks, then rotates segments of the array into place. It never needs a second full array.

## Step-by-Step Process

| Step | Description                                  |
|------|----------------------------------------------|
| 1    | Split array into sorted runs                 |
| 2    | Allocate small buffer ( $\sqrt{n}$ elements) |
| 3    | Merge adjacent runs using buffer             |
| 4    | Rotate blocks to maintain stability          |
| 5    | Repeat until one sorted region remains       |

## Tiny Code (Easy Versions)

### Python (Simplified In-Place Stable Merge)

This is a conceptual demonstration of stable in-place merging.

```

def rotate(arr, start, mid, end):
    arr[start:end] = arr[mid:end] + arr[start:mid]

def merge_in_place(arr, start, mid, end):
    left = arr[start:mid]
    i, j, k = 0, mid, start
    while i < len(left) and j < end:
        if left[i] <= arr[j]:
            arr[k] = left[i]; i += 1
        else:
            val = arr[j]
            rotate(arr, i + start, j, j + 1)
            arr[k] = val
            j += 1
        k += 1
    while i < len(left):
        arr[k] = left[i]; i += 1; k += 1

def wiki_sort(arr):
    n = len(arr)
    size = 1
    while size < n:
        for start in range(0, n, 2 * size):
            mid = min(start + size, n)
            end = min(start + 2 * size, n)
            if mid < end:
                merge_in_place(arr, start, mid, end)
        size *= 2
    return arr

arr = [3,5,1,2,4]
print(wiki_sort(arr))

```

Output:

[\$\$1, 2, 3, 4, 5]

## C (Simplified Conceptual Version)

```

#include <stdio.h>

void rotate(int arr[], int start, int mid, int end) {
    int temp[end - start];
    int idx = 0;
    for (int i = mid; i < end; i++) temp[idx++] = arr[i];
    for (int i = start; i < mid; i++) temp[idx++] = arr[i];
    for (int i = 0; i < end - start; i++) arr[start + i] = temp[i];
}

void merge_in_place(int arr[], int start, int mid, int end) {
    int i = start, j = mid;
    while (i < j && j < end) {
        if (arr[i] <= arr[j]) {
            i++;
        } else {
            int value = arr[j];
            rotate(arr, i, j, j + 1);
            arr[i] = value;
            i++; j++;
        }
    }
}

void wiki_sort(int arr[], int n) {
    for (int size = 1; size < n; size *= 2) {
        for (int start = 0; start < n; start += 2 * size) {
            int mid = (start + size < n) ? start + size : n;
            int end = (start + 2 * size < n) ? start + 2 * size : n;
            if (mid < end) merge_in_place(arr, start, mid, end);
        }
    }
}

int main(void) {
    int arr[] = {3,5,1,2,4};
    int n = sizeof(arr)/sizeof(arr[0]);
    wiki_sort(arr, n);
    for (int i = 0; i < n; i++) printf("%d ", arr[i]);
    printf("\n");
}

```

## Why It Matters

- Stable and in-place ( $O(1)$  extra space)
- Practical for memory-limited systems
- Predictable performance
- Cache-friendly merges
- Great balance of theory and practice

It brings the best of Merge Sort (stability) and in-place algorithms (low memory).

## A Gentle Proof (Why It Works)

Each merge takes  $O(n)$ , and there are  $O(\log n)$  levels of merging:

$$T(n) = O(n \log n)$$

Extra memory = small buffer ( $O(\sqrt{n})$ ) or even constant space.

| Phase           | Operation            | Cost          |
|-----------------|----------------------|---------------|
| Block detection | Scan runs            | $O(n)$        |
| Merging         | Rotation-based merge | $O(n \log n)$ |
| Space           | Fixed buffer         | $O(1)$        |

## Try It Yourself

1. Sort [3,5,1,2,4] step by step.
2. Visualize rotations during merge.
3. Add duplicates [2,2,3,1], verify stability.
4. Increase size to 16, track buffer reuse.
5. Compare with Merge Sort memory usage.
6. Measure swaps vs Merge Sort.
7. Try [1,2,3,4], minimal rotations.
8. Reverse [5,4,3,2,1], max work.
9. Implement block rotation helper.
10. Measure runtime on sorted input.

## Test Cases

| Input       | Output      | Notes           |
|-------------|-------------|-----------------|
| [3,5,1,2,4] | [1,2,3,4,5] | Basic           |
| [5,4,3,2,1] | [1,2,3,4,5] | Worst case      |
| [1,2,3,4]   | [1,2,3,4]   | Already sorted  |
| [2,2,3,1]   | [1,2,2,3]   | Stable behavior |

## Complexity

| Aspect         | Value         |
|----------------|---------------|
| Time (Best)    | $O(n)$        |
| Time (Average) | $O(n \log n)$ |
| Time (Worst)   | $O(n \log n)$ |
| Space          | $O(1)$        |
| Stable         | Yes           |
| Adaptive       | Yes           |

WikiSort is the minimalist's Merge Sort, stable, elegant, and almost memory-free. It merges not by copying, but by rotating, smooth, steady, and space-savvy.

## 139 GrailSort

GrailSort (short for “Greedy Adaptive In-place stable Sort”) is a stable, in-place, comparison-based sorting algorithm that merges sorted subarrays using block merging and local buffers.

Created by Michał Oryńczak, GrailSort combines the stability and adaptiveness of Merge Sort with in-place operation, needing only a tiny internal buffer (often  $O(\sqrt{n})$ ) or even no extra memory in its pure variant.

It's designed for practical stable sorting when memory is tight, achieving  $O(n \log n)$  worst-case time.

### What Problem Are We Solving??

Typical stable sorting algorithms (like Merge Sort) require  $O(n)$  extra space. GrailSort solves this by:

- Using small local buffers instead of large arrays
- Performing in-place stable merges
- Detecting and reusing natural runs (adaptive behavior)

Perfect for:

- Memory-constrained systems
- Embedded devices
- Large stable sorts on limited RAM

### Example

Sort [4, 1, 3, 2, 5]

| Step | Action                          | Result            |
|------|---------------------------------|-------------------|
| 1    | Detect short runs               | [4,1], [3,2], [5] |
| 2    | Sort each run                   | [1,4], [2,3], [5] |
| 3    | Merge runs using block rotation | [1,2,3,4,5]       |
| 4    | Stable order preserved          | Done              |

All merges are done in-place, with a small reusable buffer.

### How Does It Work (Plain Language)?

Think of GrailSort like a clever librarian with a tiny desk (the buffer). Instead of taking all books off the shelf, they:

- Divide shelves into small sorted groups,
- Keep a few aside as a helper buffer,
- Merge shelves directly on the rack by rotating sections in place.

It's stable, in-place, and adaptive, a rare combination.

### Step-by-Step Process

| Step | Description                                     |
|------|-------------------------------------------------|
| 1    | Detect and sort small runs (Insertion Sort)     |
| 2    | Choose small buffer (e.g., $\sqrt{n}$ elements) |
| 3    | Merge runs pairwise using buffer and rotation   |
| 4    | Gradually increase block size (1, 2, 4, 8, ...) |
| 5    | Continue merging until fully sorted             |

The algorithm's structure mirrors Merge Sort but uses block rotation to avoid copying large chunks.

## Tiny Code (Easy Versions)

### Python (Simplified Concept)

Below is a simplified stable block-merge inspired by GrailSort's principles.

```
def rotate(arr, start, mid, end):
    arr[start:end] = arr[mid:end] + arr[start:mid]

def merge_in_place(arr, left, mid, right):
    i, j = left, mid
    while i < j and j < right:
        if arr[i] <= arr[j]:
            i += 1
        else:
            val = arr[j]
            rotate(arr, i, j, j + 1)
            arr[i] = val
            i += 1
            j += 1

def grailsort(arr):
    n = len(arr)
    size = 1
    while size < n:
        for start in range(0, n, 2 * size):
            mid = min(start + size, n)
            end = min(start + 2 * size, n)
            if mid < end:
                merge_in_place(arr, start, mid, end)
        size *= 2
    return arr

arr = [4,1,3,2,5]
print(grailsort(arr))
```

Output:

```
[$$1, 2, 3, 4, 5]
```

## C (Simplified Idea)

```
#include <stdio.h>

void rotate(int arr[], int start, int mid, int end) {
    int temp[end - start];
    int idx = 0;
    for (int i = mid; i < end; i++) temp[idx++] = arr[i];
    for (int i = start; i < mid; i++) temp[idx++] = arr[i];
    for (int i = 0; i < end - start; i++) arr[start + i] = temp[i];
}

void merge_in_place(int arr[], int start, int mid, int end) {
    int i = start, j = mid;
    while (i < j && j < end) {
        if (arr[i] <= arr[j]) i++;
        else {
            int val = arr[j];
            rotate(arr, i, j, j + 1);
            arr[i] = val;
            i++; j++;
        }
    }
}

void grailsort(int arr[], int n) {
    for (int size = 1; size < n; size *= 2) {
        for (int start = 0; start < n; start += 2 * size) {
            int mid = (start + size < n) ? start + size : n;
            int end = (start + 2 * size < n) ? start + 2 * size : n;
            if (mid < end) merge_in_place(arr, start, mid, end);
        }
    }
}

int main(void) {
    int arr[] = {4,1,3,2,5};
    int n = sizeof(arr)/sizeof(arr[0]);
    grailsort(arr, n);
    for (int i = 0; i < n; i++) printf("%d ", arr[i]);
    printf("\n");
}
```

## Why It Matters

- Stable and in-place (only small buffer)
- Adaptive, faster on partially sorted data
- Predictable performance
- Ideal for limited memory systems
- Used in practical sorting libraries and research

It's the gold standard for stable, low-space sorts.

## A Gentle Proof (Why It Works)

Each merge level processes  $O(n)$  elements.

There are  $O(\log n)$  merge levels.

Thus:

$$T(n) = O(n \log n)$$

A small buffer ( $O(\sqrt{n})$ ) is reused, yielding in-place stability.

| Phase        | Operation      | Cost                    |
|--------------|----------------|-------------------------|
| Run sorting  | Insertion Sort | $O(n)$                  |
| Block merges | Rotation-based | $O(n \log n)$           |
| Space        | Local buffer   | $O(1)$ or $O(\sqrt{n})$ |

## Try It Yourself

1. Sort [4,1,3,2,5] step by step.
2. Try [1,2,3,4,5], no merges needed.
3. Check duplicates [2,2,1,1], verify stability.
4. Experiment with [10,9,8,7,6,5].
5. Visualize rotations during merge.
6. Change block size, observe performance.
7. Implement  $\sqrt{n}$  buffer manually.
8. Compare with Merge Sort (space).
9. Measure time vs WikiSort.
10. Try large array (10k elements).

## Test Cases

| Input       | Output      | Notes          |
|-------------|-------------|----------------|
| [4,1,3,2,5] | [1,2,3,4,5] | Basic test     |
| [1,2,3,4]   | [1,2,3,4]   | Already sorted |
| [5,4,3,2,1] | [1,2,3,4,5] | Reverse        |
| [2,2,1,1]   | [1,1,2,2]   | Stable         |

## Complexity

| Aspect         | Value                   |
|----------------|-------------------------|
| Time (Best)    | $O(n)$                  |
| Time (Average) | $O(n \log n)$           |
| Time (Worst)   | $O(n \log n)$           |
| Space          | $O(1)$ to $O(\sqrt{n})$ |
| Stable         | Yes                     |
| Adaptive       | Yes                     |

GrailSort is the gentle engineer of sorting, steady, stable, and space-wise. It doesn't rush; it rearranges with precision, merging order from within.

## 140 Adaptive Hybrid Sort

Adaptive Hybrid Sort is a meta-sorting algorithm that dynamically combines multiple sorting strategies, such as QuickSort, Merge Sort, Insertion Sort, and Heap Sort, depending on the data characteristics and runtime patterns it detects.

It adapts in real-time, switching between methods based on factors like array size, degree of pre-sortedness, data distribution, and recursion depth. This makes it a universal, practical sorter optimized for diverse workloads.

## What Problem Are We Solving??

No single sorting algorithm is best for all situations:

- QuickSort is fast on random data but unstable and bad in worst case.
- Merge Sort is stable but memory-hungry.
- Insertion Sort is great for small or nearly sorted arrays.

- Heap Sort guarantees  $O(n \log n)$  but has poor locality.

Adaptive Hybrid Sort solves this by blending algorithms:

1. Start with QuickSort for speed.
2. Detect sorted or small regions → switch to Insertion Sort.
3. Detect bad pivot patterns → switch to Heap Sort.
4. Detect stability needs or patterns → use Merge Sort.

It's a unified, self-tuning sorting system.

### **Example**

Sort [2, 3, 5, 4, 6, 7, 8, 1]

| Step | Detection                     | Action                             |
|------|-------------------------------|------------------------------------|
| 1    | Mostly sorted except last few | Switch to Insertion Sort           |
| 2    | Sort locally                  | [1,2,3,4,5,6,7,8]                  |
| 3    | Done                          | Adaptive path chosen automatically |

If input were random, it would stay with QuickSort. If adversarial, it would pivot to Heap Sort.

### **How Does It Work (Plain Language)?**

Imagine a skilled chef with many tools, knives, mixers, ovens. When slicing carrots (small data), they use a paring knife (Insertion Sort). When breaking a tough root (unsorted array), they grab a heavy cleaver (QuickSort). When something's too complex, they use machinery (Merge Sort).

Adaptive Hybrid Sort works the same way, choosing the right tool at the right time.

### **Step-by-Step Process**

| Step | Description                                         |
|------|-----------------------------------------------------|
| 1    | Start with QuickSort (good average case)            |
| 2    | If recursion depth too high → switch to Heap Sort   |
| 3    | If subarray small ( threshold) → use Insertion Sort |
| 4    | If stable sorting required → use Merge Sort         |
| 5    | If data partially sorted → use TimSort-like merge   |

| Step | Description                             |
|------|-----------------------------------------|
| 6    | Combine results for final sorted output |

### Tiny Code (Easy Versions)

#### Python (Simplified Hybrid Sort)

```

def insertion_sort(arr, left, right):
    for i in range(left + 1, right):
        key = arr[i]
        j = i - 1
        while j >= left and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

def heapify(arr, n, i):
    largest = i
    l = 2*i + 1
    r = 2*i + 2
    if l < n and arr[l] > arr[largest]:
        largest = l
    if r < n and arr[r] > arr[largest]:
        largest = r
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heap_sort(arr):
    n = len(arr)
    for i in range(n//2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n - 1, 0, -1):
        arr[0], arr[i] = arr[i], arr[0]
        heapify(arr, i, 0)

def partition(arr, low, high):
    pivot = arr[(low + high) // 2]
    i, j = low, high
    while i <= j:

```

```

        while arr[i] < pivot: i += 1
        while arr[j] > pivot: j -= 1
        if i <= j:
            arr[i], arr[j] = arr[j], arr[i]
            i += 1; j -= 1
    return i

def hybrid_sort(arr, low=0, high=None, depth_limit=None):
    if high is None:
        high = len(arr) - 1
    if depth_limit is None:
        import math
        depth_limit = 2 * math.floor(math.log2(len(arr)) + 1)
    size = high - low + 1
    if size <= 16:
        insertion_sort(arr, low, high + 1)
        return
    if depth_limit == 0:
        sub = arr[low:high + 1]
        heap_sort(sub)
        arr[low:high + 1] = sub
        return
    pivot_index = partition(arr, low, high)
    if low < pivot_index - 1:
        hybrid_sort(arr, low, pivot_index - 1, depth_limit - 1)
    if pivot_index < high:
        hybrid_sort(arr, pivot_index, high, depth_limit - 1)

arr = [2, 3, 5, 4, 6, 7, 8, 1]
hybrid_sort(arr)
print(arr)

```

Output:

[\$\$1, 2, 3, 4, 5, 6, 7, 8]

## C (Conceptual Hybrid Sort)

```

#include <stdio.h>
#include <math.h>

```

```

void insertion_sort(int arr[], int left, int right) {
    for (int i = left + 1; i < right; i++) {
        int key = arr[i], j = i - 1;
        while (j >= left && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

void heapify(int arr[], int n, int i) {
    int largest = i;
    int l = 2*i + 1, r = 2*i + 2;
    if (l < n && arr[l] > arr[largest]) largest = l;
    if (r < n && arr[r] > arr[largest]) largest = r;
    if (largest != i) {
        int tmp = arr[i]; arr[i] = arr[largest]; arr[largest] = tmp;
        heapify(arr, n, largest);
    }
}

void heap_sort(int arr[], int n) {
    for (int i = n/2 - 1; i >= 0; i--) heapify(arr, n, i);
    for (int i = n - 1; i > 0; i--) {
        int tmp = arr[0]; arr[0] = arr[i]; arr[i] = tmp;
        heapify(arr, i, 0);
    }
}

int partition(int arr[], int low, int high) {
    int pivot = arr[(low + high) / 2];
    int i = low, j = high;
    while (i <= j) {
        while (arr[i] < pivot) i++;
        while (arr[j] > pivot) j--;
        if (i <= j) {
            int tmp = arr[i]; arr[i] = arr[j]; arr[j] = tmp;
            i++; j--;
        }
    }
    return i;
}

```

```

}

void hybrid_sort(int arr[], int low, int high, int depth_limit) {
    int size = high - low + 1;
    if (size <= 16) { insertion_sort(arr, low, high + 1); return; }
    if (depth_limit == 0) { heap_sort(arr + low, size); return; }
    int p = partition(arr, low, high);
    if (low < p - 1) hybrid_sort(arr, low, p - 1, depth_limit - 1);
    if (p < high) hybrid_sort(arr, p, high, depth_limit - 1);
}

int main(void) {
    int arr[] = {2,3,5,4,6,7,8,1};
    int n = sizeof(arr)/sizeof(arr[0]);
    int depth_limit = 2 * log2(n);
    hybrid_sort(arr, 0, n - 1, depth_limit);
    for (int i = 0; i < n; i++) printf("%d ", arr[i]);
    printf("\n");
}

```

## Why It Matters

- Adaptive to input shape
- Hybrid = flexibility + safety
- Stable runtime across data types
- Real-world robust for mixed data
- Balances speed, memory, stability, and predictability

## A Gentle Proof (Why It Works)

Let  $T(n)$  be the runtime.

Each phase is  $O(n)$ , and recursion depth is  $O(\log n)$ .

Adaptive switching ensures no pathological behavior.

$$T(n) = O(n \log n)$$

Best case (sorted): Insertion Sort runs in  $O(n)$ .

Worst case (adversarial): Heap fallback  $\rightarrow O(n \log n)$ .

| Case        | Behavior       | Complexity    |
|-------------|----------------|---------------|
| Sorted      | Insertion Sort | $O(n)$        |
| Random      | QuickSort      | $O(n \log n)$ |
| Adversarial | Heap Sort      | $O(n \log n)$ |

## Try It Yourself

1. Sort  $[2,3,5,4,6,7,8,1]$ .
2. Try  $[1,2,3,4,5,6] \rightarrow$  insertion path.
3. Reverse  $[9,8,7,6,5] \rightarrow$  heap path.
4. Mix sorted + random halves.
5. Measure recursion depth.
6. Increase threshold to 32.
7. Add duplicates  $\rightarrow$  observe stability.
8. Compare with IntroSort, TimSort.
9. Benchmark on large random data.
10. Visualize switch decisions.

## Test Cases

| Input               | Output              | Notes      |
|---------------------|---------------------|------------|
| $[2,3,5,4,6,7,8,1]$ | $[1,2,3,4,5,6,7,8]$ | Random     |
| $[1,2,3,4,5]$       | $[1,2,3,4,5]$       | Sorted     |
| $[5,4,3,2,1]$       | $[1,2,3,4,5]$       | Reverse    |
| $[10,10,9,9,8]$     | $[8,9,9,10,10]$     | Duplicates |

## Complexity

| Aspect         | Value         |
|----------------|---------------|
| Time (Best)    | $O(n)$        |
| Time (Average) | $O(n \log n)$ |
| Time (Worst)   | $O(n \log n)$ |
| Space          | $O(\log n)$   |
| Stable         | No            |
| Adaptive       | Yes           |

Adaptive Hybrid Sort is the chameleon of sorting, it watches the data, reads the room, and chooses the perfect move. Fast when it can be, safe when it must be.

## Section 15. Special sorts

### 141 Cycle Sort

Cycle Sort is a comparison-based sorting algorithm designed to minimize the number of writes. It's ideal when writing to memory or storage is expensive (like EEPROM or flash memory), since each element is written exactly once into its final position.

It achieves  $O(n^2)$  comparisons but performs the minimal possible number of writes, making it unique among sorting algorithms.

#### What Problem Are We Solving??

Most sorting algorithms (like QuickSort or Merge Sort) swap elements multiple times before they reach their final position. If each write is costly (e.g., embedded systems or flash memory), that's wasteful.

Cycle Sort asks: "How can we put each element *directly* where it belongs, in one cycle, with as few writes as possible?"

#### Example

Sort [3, 1, 2]

| Step  | Element   | Correct Position | Action              |
|-------|-----------|------------------|---------------------|
| 1     | 3         | Index 2          | Swap 3 → position 2 |
| 2     | 2         | Index 1          | Swap 2 → position 1 |
| 3     | 1         | Index 0          | Done                |
| Final | [1, 2, 3] | Sorted           | Minimal writes      |

Each element cycles into place once.

## How Does It Work (Plain Language)?

Think of it like putting books on a shelf:

- You pick one book (element),
- Figure out where it belongs,
- Put it there, swapping with whatever's currently there,
- Repeat until every book is in the right spot.

Each cycle ensures every element reaches its final position exactly once.

## Step-by-Step Process

| Step | Description                                                  |
|------|--------------------------------------------------------------|
| 1    | Loop through array positions                                 |
| 2    | For each position, count how many elements are smaller       |
| 3    | That count = final position                                  |
| 4    | If not already correct, cycle the element to its right place |
| 5    | Continue cycling until original element returns to start     |
| 6    | Move to next position and repeat                             |

## Tiny Code (Easy Versions)

### Python

```
def cycle_sort(arr):
    n = len(arr)
    writes = 0
    for cycle_start in range(n - 1):
        item = arr[cycle_start]
        pos = cycle_start
        for i in range(cycle_start + 1, n):
            if arr[i] < item:
                pos += 1
            if pos == cycle_start:
                continue
            while item == arr[pos]:
                pos += 1
            arr[pos], item = item, arr[pos]
            writes += 1
```

```

while pos != cycle_start:
    pos = cycle_start
    for i in range(cycle_start + 1, n):
        if arr[i] < item:
            pos += 1
    while item == arr[pos]:
        pos += 1
    arr[pos], item = item, arr[pos]
    writes += 1
print("Total writes:", writes)
return arr

arr = [3, 1, 2, 4]
print(cycle_sort(arr))

```

Output:

```

Total writes: 3
$[1, 2, 3, 4]

```

## C

```

#include <stdio.h>

void cycle_sort(int arr[], int n) {
    int writes = 0;
    for (int cycle_start = 0; cycle_start < n - 1; cycle_start++) {
        int item = arr[cycle_start];
        int pos = cycle_start;

        for (int i = cycle_start + 1; i < n; i++)
            if (arr[i] < item)
                pos++;

        if (pos == cycle_start) continue;

        while (item == arr[pos]) pos++;
        int temp = arr[pos]; arr[pos] = item; item = temp;
        writes++;

        while (pos != cycle_start) {

```

```

        pos = cycle_start;
        for (int i = cycle_start + 1; i < n; i++)
            if (arr[i] < item)
                pos++;
        while (item == arr[pos]) pos++;
        temp = arr[pos]; arr[pos] = item; item = temp;
        writes++;
    }
}
printf("Total writes: %d\n", writes);
}

int main(void) {
    int arr[] = {3, 1, 2, 4};
    int n = sizeof(arr) / sizeof(arr[0]);
    cycle_sort(arr, n);
    for (int i = 0; i < n; i++) printf("%d ", arr[i]);
    printf("\n");
}

```

Output:

```

Total writes: 3
1 2 3 4

```

## Why It Matters

- Minimizes writes (useful for flash memory, EEPROMs)
- In-place
- Deterministic writes = fewer wear cycles
- Educational example of permutation cycles in sorting

Not fast, but *frugal*, every move counts.

## A Gentle Proof (Why It Works)

Each element moves to its correct position once.

If array size is  $n$ , total writes  $\leq n$ .

Counting smaller elements ensures correctness:

$$\text{pos}(x) = |y \mid y < x|$$

Each cycle resolves one permutation cycle of misplaced elements. Thus, algorithm terminates with all items placed exactly once.

### Try It Yourself

1. Sort [3, 1, 2] step by step.
2. Count how many writes you perform.
3. Try [4, 3, 2, 1], maximum cycles.
4. Try [1, 2, 3, 4], no writes.
5. Test duplicates [3, 1, 2, 3].
6. Implement a version counting cycles.
7. Compare write count with Selection Sort.
8. Benchmark on 1000 random elements.
9. Measure wear-leveling benefit for flash.
10. Visualize cycles as arrows in permutation graph.

### Test Cases

| Input     | Output    | Writes | Notes              |
|-----------|-----------|--------|--------------------|
| [3,1,2,4] | [1,2,3,4] | 3      | 3 cycles           |
| [1,2,3]   | [1,2,3]   | 0      | Already sorted     |
| [4,3,2,1] | [1,2,3,4] | 4      | Max writes         |
| [3,1,2,3] | [1,2,3,3] | 3      | Handles duplicates |

### Complexity

| Aspect   | Value    |
|----------|----------|
| Time     | $O(n^2)$ |
| Writes   | n        |
| Space    | $O(1)$   |
| Stable   | No       |
| Adaptive | No       |

Cycle Sort is the minimalist's sorter, every write is intentional, every move meaningful. It may not be fast, but it's *precisely efficient*.

## 142 Comb Sort

Comb Sort is an improved version of Bubble Sort that eliminates small elements (often called “turtles”) faster by comparing elements far apart first, using a shrinking gap strategy.

It starts with a large gap (e.g. array length) and reduces it each pass until it reaches 1, where it behaves like a regular Bubble Sort. The result is fewer comparisons and faster convergence.

### What Problem Are We Solving??

Bubble Sort is simple but slow, mainly because:

- It swaps only adjacent elements.
- Small elements crawl slowly to the front.

Comb Sort fixes this by using a gap to leap over elements, allowing “turtles” to move quickly forward.

It’s like sorting with a comb, wide teeth first (large gap), then finer ones (small gap).

### Example

Sort [8, 4, 1, 3, 7]

| Step | Gap     | Pass Result |                 | Notes         |
|------|---------|-------------|-----------------|---------------|
| 1    | 5 / 1.3 | 3           | [3, 4, 1, 8, 7] | Compare 8 3   |
| 2    | 3 / 1.3 | 2           | [1, 4, 3, 8, 7] | Compare 3 1   |
| 3    | 2 / 1.3 | 1           | [1, 3, 4, 7, 8] | Bubble finish |
| Done | ,       |             | [1, 3, 4, 7, 8] | Sorted        |

Turtles (1, 3) jump forward earlier, speeding up convergence.

### How Does It Work (Plain Language)?

Think of it like shrinking a jump rope, at first, you make big jumps to cover ground fast, then smaller ones to fine-tune.

You start with a gap, compare and swap elements that far apart, shrink the gap each round, and stop when the gap reaches 1 *and no swaps happen*.

## Step-by-Step Process

| Step | Description                                                   |
|------|---------------------------------------------------------------|
| 1    | Initialize <code>gap = n</code> and <code>shrink = 1.3</code> |
| 2    | Repeat until <code>gap == 1</code> and no swaps               |
| 3    | Divide <code>gap</code> by shrink factor each pass            |
| 4    | Compare elements at distance <code>gap</code>                 |
| 5    | Swap if out of order                                          |
| 6    | Continue until sorted                                         |

## Tiny Code (Easy Versions)

### Python

```
def comb_sort(arr):
    n = len(arr)
    gap = n
    shrink = 1.3
    swapped = True

    while gap > 1 or swapped:
        gap = int(gap / shrink)
        if gap < 1:
            gap = 1
        swapped = False
        for i in range(n - gap):
            if arr[i] > arr[i + gap]:
                arr[i], arr[i + gap] = arr[i + gap], arr[i]
                swapped = True
    return arr

arr = [8, 4, 1, 3, 7]
print(comb_sort(arr))
```

Output:

[\$\$1, 3, 4, 7, 8]

## C

```
#include <stdio.h>

void comb_sort(int arr[], int n) {
    int gap = n;
    const float shrink = 1.3;
    int swapped = 1;

    while (gap > 1 || swapped) {
        gap = (int)(gap / shrink);
        if (gap < 1) gap = 1;
        swapped = 0;
        for (int i = 0; i + gap < n; i++) {
            if (arr[i] > arr[i + gap]) {
                int tmp = arr[i];
                arr[i] = arr[i + gap];
                arr[i + gap] = tmp;
                swapped = 1;
            }
        }
    }
}

int main(void) {
    int arr[] = {8, 4, 1, 3, 7};
    int n = sizeof(arr) / sizeof(arr[0]);
    comb_sort(arr, n);
    for (int i = 0; i < n; i++) printf("%d ", arr[i]);
    printf("\n");
}
```

Output:

1 3 4 7 8

### Why It Matters

- Faster than Bubble Sort
- Simple implementation
- In-place and adaptive

- Efficient for small datasets or nearly sorted arrays

A stepping stone toward more efficient algorithms like Shell Sort.

### **A Gentle Proof (Why It Works)**

The shrink factor ensures gap reduction converges to 1 in  $O(\log n)$  steps.  
Each pass fixes distant inversions early, reducing total swaps.

Total cost is dominated by local passes when gap = 1 (Bubble Sort).  
Hence, average complexity  $O(n \log n)$  for random data,  $O(n^2)$  worst case.

| Phase            | Description          | Cost           |
|------------------|----------------------|----------------|
| Large gap passes | Move turtles forward | $O(n \log n)$  |
| Small gap passes | Final fine-tuning    | $O(n^2)$ worst |

### **Try It Yourself**

1. Sort [8, 4, 1, 3, 7] step by step.
2. Try [1, 2, 3, 4, 5], minimal passes.
3. Try [5, 4, 3, 2, 1], observe gap shrinking.
4. Change shrink factor (1.5, 1.2).
5. Measure swaps per iteration.
6. Compare with Bubble Sort.
7. Visualize movement of smallest element.
8. Benchmark large random array.
9. Track gap evolution over time.
10. Implement early-stop optimization.

### **Test Cases**

| Input       | Output      | Notes          |
|-------------|-------------|----------------|
| [8,4,1,3,7] | [1,3,4,7,8] | Basic test     |
| [1,2,3,4,5] | [1,2,3,4,5] | Already sorted |
| [5,4,3,2,1] | [1,2,3,4,5] | Reverse order  |
| [4,1,3,2]   | [1,2,3,4]   | Short array    |

## Complexity

| Aspect         | Value         |
|----------------|---------------|
| Time (Best)    | $O(n \log n)$ |
| Time (Average) | $O(n \log n)$ |
| Time (Worst)   | $O(n^2)$      |
| Space          | $O(1)$        |
| Stable         | No            |
| Adaptive       | Yes           |

Comb Sort sweeps through data like a comb through tangled hair, wide strokes first, fine ones later, until everything's smooth and ordered.

## 143 Gnome Sort

Gnome Sort is a simple comparison-based sorting algorithm that works like a garden gnome arranging flower pots, it looks at two adjacent elements and swaps them if they're out of order, then steps backward to check the previous pair again.

It's conceptually similar to Insertion Sort, but implemented with a single loop and no nested structure, making it elegant and intuitive for learners.

### What Problem Are We Solving??

Insertion Sort requires nested loops or recursion, which can be tricky to visualize. Gnome Sort offers the same logic using a simple forward-backward walk:

- Move forward if elements are ordered.
- Move back and swap if they're not.

This creates a human-like sorting routine, step forward, fix, step back, repeat.

### Example

Sort [5, 3, 4, 2]

| Step | Index | Action                              | Result       |
|------|-------|-------------------------------------|--------------|
| 1    | 1     | $5 > 3 \rightarrow \text{swap}$     | [3, 5, 4, 2] |
| 2    | 0     | At start $\rightarrow$ move forward | [3, 5, 4, 2] |

| Step | Index | Action          | Result       |
|------|-------|-----------------|--------------|
| 3    | 2     | 5 > 4 → swap    | [3, 4, 5, 2] |
| 4    | 1     | 3 < 4 → forward | [3, 4, 5, 2] |
| 5    | 3     | 5 > 2 → swap    | [3, 4, 2, 5] |
| 6    | 2     | 4 > 2 → swap    | [3, 2, 4, 5] |
| 7    | 1     | 3 > 2 → swap    | [2, 3, 4, 5] |
| 8    | 0     | Done            | [2, 3, 4, 5] |

Sorted with local fixes, no explicit nested loops.

### How Does It Work (Plain Language)?

Imagine a gnome walking down a row of flower pots:

- If the two pots are in order, step forward.
- If not, swap them and step back.
- If you reach the start, just move forward again.

Keep walking until you reach the end, the garden (array) is sorted.

### Step-by-Step Process

| Step | Description                                     |
|------|-------------------------------------------------|
| 1    | Start at index 1                                |
| 2    | Compare arr[i] with arr[i-1]                    |
| 3    | If arr[i] >= arr[i-1], move forward (i++)       |
| 4    | Else, swap arr[i] and arr[i-1], move back (i--) |
| 5    | If i == 0, move to i = 1                        |
| 6    | Repeat until i reaches n                        |

### Tiny Code (Easy Versions)

#### Python

```

def gnome_sort(arr):
    i = 1
    n = len(arr)
    while i < n:
        if i == 0 or arr[i] >= arr[i - 1]:
            i += 1
        else:
            arr[i], arr[i - 1] = arr[i - 1], arr[i]
            i -= 1
    return arr

arr = [5, 3, 4, 2]
print(gnome_sort(arr))

```

Output:

[\$\$2, 3, 4, 5]

## C

```

#include <stdio.h>

void gnome_sort(int arr[], int n) {
    int i = 1;
    while (i < n) {
        if (i == 0 || arr[i] >= arr[i - 1]) {
            i++;
        } else {
            int tmp = arr[i];
            arr[i] = arr[i - 1];
            arr[i - 1] = tmp;
            i--;
        }
    }
}

int main(void) {
    int arr[] = {5, 3, 4, 2};
    int n = sizeof(arr) / sizeof(arr[0]);
    gnome_sort(arr, n);
    for (int i = 0; i < n; i++) printf("%d ", arr[i]);
}

```

```
    printf("\n");
}
```

Output:

```
2 3 4 5
```

## Why It Matters

- Simple mental model, easy to understand.
- No nested loops, clean control flow.
- In-place, no extra space.
- Demonstrates local correction in sorting.

It's slower than advanced algorithms but ideal for educational purposes.

## A Gentle Proof (Why It Works)

Each swap moves an element closer to its correct position. Whenever a swap happens, the gnome steps back to ensure local order.

Since every inversion is eventually corrected, the algorithm terminates with a sorted array.

Number of swaps proportional to number of inversions  $\rightarrow O(n^2)$ .

| Case    | Behavior       | Complexity |
|---------|----------------|------------|
| Sorted  | Linear scan    | $O(n)$     |
| Random  | Frequent swaps | $O(n^2)$   |
| Reverse | Max swaps      | $O(n^2)$   |

## Try It Yourself

1. Sort [5,3,4,2] manually step by step.
2. Try [1,2,3,4], minimal steps.
3. Try [4,3,2,1], worst case.
4. Count number of swaps.
5. Compare with Insertion Sort.
6. Track index changes after each swap.
7. Implement visual animation (pointer walk).
8. Try duplicates [2,1,2,1].

9. Measure time for  $n = 1000$ .
10. Add early-exit optimization.

## Test Cases

| Input     | Output    | Notes          |
|-----------|-----------|----------------|
| [5,3,4,2] | [2,3,4,5] | Basic          |
| [1,2,3,4] | [1,2,3,4] | Already sorted |
| [4,3,2,1] | [1,2,3,4] | Reverse        |
| [2,1,2,1] | [1,1,2,2] | Duplicates     |

## Complexity

| Aspect         | Value    |
|----------------|----------|
| Time (Best)    | $O(n)$   |
| Time (Average) | $O(n^2)$ |
| Time (Worst)   | $O(n^2)$ |
| Space          | $O(1)$   |
| Stable         | Yes      |
| Adaptive       | Yes      |

Gnome Sort is a friendly, step-by-step sorter, it doesn't rush, just tidies things one pot at a time until the whole row is in order.

## 144 Cocktail Sort

Cocktail Sort (also known as Bidirectional Bubble Sort or Shaker Sort) is a simple variation of Bubble Sort that sorts the list in both directions alternately, forward then backward, during each pass.

This bidirectional movement helps small elements ("turtles") bubble up faster from the end, fixing one of Bubble Sort's main weaknesses.

## **What Problem Are We Solving??**

Bubble Sort only moves elements in one direction, large ones float to the end, but small ones crawl slowly to the start.

Cocktail Sort solves this by shaking the list:

- Forward pass: moves large items right
- Backward pass: moves small items left

This makes it more efficient on nearly sorted arrays or when both ends need cleaning.

### **Example**

Sort [4, 3, 1, 2]

| Step | Direction | Action                       | Result    |
|------|-----------|------------------------------|-----------|
| 1    | Forward   | Compare & swap 4 3, 3 1, 4 2 | [3,1,2,4] |
| 2    | Backward  | Compare & swap 2 1, 3 1      | [1,3,2,4] |
| 3    | Forward   | Compare & swap 3 2           | [1,2,3,4] |
| Done | ,         | Sorted                       |           |

Fewer passes than Bubble Sort.

## **How Does It Work (Plain Language)?**

Think of a bartender shaking a cocktail shaker back and forth, each shake moves ingredients (elements) closer to the right place from both sides.

You traverse the array:

- Left to right: push largest elements to the end
- Right to left: push smallest elements to the start

Stop when no swaps occur, the array is sorted.

| Step | Description |
|------|-------------|
|------|-------------|

### Step-by-Step Process

| Step | Description                                                                                              |
|------|----------------------------------------------------------------------------------------------------------|
| 1    | Initialize swapped = True                                                                                |
| 2    | While swapped:                                                                                           |
| a.   | Set swapped = False                                                                                      |
| b.   | Forward pass ( $i = \text{start} \rightarrow \text{end}$ ): swap if $\text{arr}[i] > \text{arr}[i+1]$    |
| c.   | If swapped == False: break (sorted)                                                                      |
| d.   | Backward pass ( $i = \text{end}-1 \rightarrow \text{start}$ ): swap if $\text{arr}[i] > \text{arr}[i+1]$ |
| 3    | Repeat until sorted                                                                                      |

### Tiny Code (Easy Versions)

#### Python

```
def cocktail_sort(arr):
    n = len(arr)
    swapped = True
    start = 0
    end = n - 1

    while swapped:
        swapped = False
        # Forward pass
        for i in range(start, end):
            if arr[i] > arr[i + 1]:
                arr[i], arr[i + 1] = arr[i + 1], arr[i]
                swapped = True
        if not swapped:
            break
        swapped = False
        end -= 1
        # Backward pass
        for i in range(end - 1, start - 1, -1):
            if arr[i] > arr[i + 1]:
                arr[i], arr[i + 1] = arr[i + 1], arr[i]
                swapped = True
```

```

        start += 1
    return arr

arr = [4, 3, 1, 2]
print(cocktail_sort(arr))

```

Output:

[\$\$1, 2, 3, 4]

## C

```

#include <stdio.h>

void cocktail_sort(int arr[], int n) {
    int start = 0, end = n - 1, swapped = 1;
    while (swapped) {
        swapped = 0;
        // Forward pass
        for (int i = start; i < end; i++) {
            if (arr[i] > arr[i + 1]) {
                int tmp = arr[i];
                arr[i] = arr[i + 1];
                arr[i + 1] = tmp;
                swapped = 1;
            }
        }
        if (!swapped) break;
        swapped = 0;
        end--;
        // Backward pass
        for (int i = end - 1; i >= start; i--) {
            if (arr[i] > arr[i + 1]) {
                int tmp = arr[i];
                arr[i] = arr[i + 1];
                arr[i + 1] = tmp;
                swapped = 1;
            }
        }
        start++;
    }
}

```

```

}

int main(void) {
    int arr[] = {4, 3, 1, 2};
    int n = sizeof(arr) / sizeof(arr[0]);
    cocktail_sort(arr, n);
    for (int i = 0; i < n; i++) printf("%d ", arr[i]);
    printf("\n");
}

```

Output:

1 2 3 4

### Why It Matters

- Bidirectional improvement on Bubble Sort
- In-place and simple
- Performs well on nearly sorted data
- Adaptive, stops early when sorted

Great educational bridge to understanding bidirectional scans and adaptive sorting.

### A Gentle Proof (Why It Works)

Each forward pass pushes the largest element to the right. Each backward pass pushes the smallest element to the left.

After each full cycle, the sorted region expands from both ends. The algorithm stops when no swaps occur (sorted).

Total operations depend on number of inversions:

$$O(n^2) \text{ worst, } O(n) \text{ best (sorted input)}$$

| Case    | Behavior               | Complexity |
|---------|------------------------|------------|
| Sorted  | One bidirectional scan | $O(n)$     |
| Random  | Many swaps             | $O(n^2)$   |
| Reverse | Max passes             | $O(n^2)$   |

## Try It Yourself

1. Sort [4,3,1,2] manually step by step.
2. Try [1,2,3,4], should stop early.
3. Try [5,4,3,2,1], observe shaking effect.
4. Count swaps each pass.
5. Compare passes with Bubble Sort.
6. Visualize forward/backward movement.
7. Add “swap counter” variable.
8. Test duplicates [3,1,3,2,1].
9. Measure performance on nearly sorted data.
10. Modify shrink window size.

## Test Cases

| Input       | Output      | Notes          |
|-------------|-------------|----------------|
| [4,3,1,2]   | [1,2,3,4]   | Basic          |
| [1,2,3,4]   | [1,2,3,4]   | Already sorted |
| [5,4,3,2,1] | [1,2,3,4,5] | Reverse        |
| [3,1,3,2,1] | [1,1,2,3,3] | Duplicates     |

## Complexity

| Aspect         | Value    |
|----------------|----------|
| Time (Best)    | $O(n)$   |
| Time (Average) | $O(n^2)$ |
| Time (Worst)   | $O(n^2)$ |
| Space          | $O(1)$   |
| Stable         | Yes      |
| Adaptive       | Yes      |

Cocktail Sort is the sorter that doesn't just bubble, it *shakes* the data into order, making sure both ends get attention every round.

## 145 Pancake Sort

Pancake Sort is a whimsical and educational sorting algorithm inspired by flipping pancakes on a plate, at each step, you bring the largest unsorted pancake to the top, then flip the stack to move it to its correct position.

It's not practical for large datasets, but it's a brilliant way to illustrate prefix reversals, maximum selection, and in-place transformations.

### What Problem Are We Solving??

Standard selection sort swaps elements pairwise. Pancake Sort instead uses prefix reversals, flipping a whole section of the array, to move the largest item into place.

It's a thought experiment in restricted operations: What if the only thing you could do is *flip*?

### Example

Sort [3, 6, 1, 5, 2, 4]

| Step | Action                               | Result             |
|------|--------------------------------------|--------------------|
| 1    | Find max (6) → index 1, flip first 2 | [6, 3, 1, 5, 2, 4] |
| 2    | Flip first 6                         | [4, 2, 5, 1, 3, 6] |
| 3    | Find max (5) → index 2, flip first 3 | [5, 2, 4, 1, 3, 6] |
| 4    | Flip first 5                         | [3, 1, 4, 2, 5, 6] |
| 5    | Repeat for smaller prefixes          | [1, 2, 3, 4, 5, 6] |

Sorted using flips only!

### How Does It Work (Plain Language)?

Imagine flipping a stack of pancakes:

1. Find the biggest one not yet placed.
2. Flip the stack to bring it to the top.
3. Flip again to move it into its final position.

Repeat, each time excluding the sorted top portion.

## Step-by-Step Process

| Step | Description                                     |
|------|-------------------------------------------------|
| 1    | Find index of largest unsorted element          |
| 2    | Flip subarray from start to that index          |
| 3    | Flip entire unsorted subarray to move it to end |
| 4    | Reduce unsorted portion by one                  |
| 5    | Repeat until sorted                             |

## Tiny Code (Easy Versions)

### Python

```
def flip(arr, k):
    arr[:k+1] = reversed(arr[:k+1])

def pancake_sort(arr):
    n = len(arr)
    for curr_size in range(n, 1, -1):
        max_idx = arr.index(max(arr[:curr_size]))
        if max_idx != curr_size - 1:
            flip(arr, max_idx)
            flip(arr, curr_size - 1)
    return arr

arr = [3, 6, 1, 5, 2, 4]
print(pancake_sort(arr))
```

Output:

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

### C

```
#include <stdio.h>

void flip(int arr[], int k) {
    int start = 0;
    while (start < k) {
```

```

        int temp = arr[start];
        arr[start] = arr[k];
        arr[k] = temp;
        start++;
        k--;
    }
}

int find_max(int arr[], int n) {
    int max_idx = 0;
    for (int i = 1; i < n; i++)
        if (arr[i] > arr[max_idx])
            max_idx = i;
    return max_idx;
}

void pancake_sort(int arr[], int n) {
    for (int size = n; size > 1; size--) {
        int max_idx = find_max(arr, size);
        if (max_idx != size - 1) {
            flip(arr, max_idx);
            flip(arr, size - 1);
        }
    }
}

int main(void) {
    int arr[] = {3, 6, 1, 5, 2, 4};
    int n = sizeof(arr) / sizeof(arr[0]);
    pancake_sort(arr, n);
    for (int i = 0; i < n; i++) printf("%d ", arr[i]);
    printf("\n");
}

```

Output:

1 2 3 4 5 6

### Why It Matters

- Fun demonstration of prefix operations

- In-place and simple
- Shows how restricted operations can still sort
- Theoretical interest, base for pancake networks
- Used in bioinformatics (genome rearrangements)

### A Gentle Proof (Why It Works)

Each iteration places the largest remaining element at its final index. Two flips per iteration (worst case). At most ( $2(n - 1)$ ) flips total.

Correctness follows from:

- Flipping is a reversal, which preserves order except within flipped segment.
- Each largest element is locked at the end after placement.

$$T(n) = O(n^2)$$

because each `max()` and `flip()` operation is  $O(n)$ .

### Try It Yourself

1. Sort `[3,6,1,5,2,4]` manually.
2. Trace each flip visually.
3. Try `[1,2,3,4]`, no flips needed.
4. Reverse `[4,3,2,1]`, observe maximum flips.
5. Count flips per iteration.
6. Implement flip visualization.
7. Replace `max()` with manual search.
8. Print intermediate arrays.
9. Analyze flip count for random input.
10. Challenge: implement recursive version.

### Test Cases

| Input                      | Output                     | Flips | Notes          |
|----------------------------|----------------------------|-------|----------------|
| <code>[3,6,1,5,2,4]</code> | <code>[1,2,3,4,5,6]</code> | 8     | Classic        |
| <code>[1,2,3,4]</code>     | <code>[1,2,3,4]</code>     | 0     | Already sorted |
| <code>[4,3,2,1]</code>     | <code>[1,2,3,4]</code>     | 6     | Worst case     |
| <code>[2,1,3]</code>       | <code>[1,2,3]</code>       | 3     | Small array    |

## Complexity

| Aspect         | Value    |
|----------------|----------|
| Time (Worst)   | $O(n^2)$ |
| Time (Average) | $O(n^2)$ |
| Time (Best)    | $O(n)$   |
| Space          | $O(1)$   |
| Stable         | No       |
| Adaptive       | No       |

Pancake Sort flips its way to victory, a charming example of ingenuity under constraint. You don't need fancy tools, just a good spatula and some patience.

## 146 Bitonic Sort

Bitonic Sort is a parallel sorting algorithm designed for sorting networks. It works by constructing and merging bitonic sequences, sequences that first increase, then decrease (or vice versa).

It's especially powerful on hardware, GPUs, and parallel processors, where multiple comparisons can happen at once.

### What Problem Are We Solving??

Most standard algorithms (QuickSort, MergeSort) are data-dependent, their flow changes depending on comparisons. That's a problem for hardware or parallel systems.

Bitonic Sort fixes this by having a fixed comparison pattern, perfect for parallel execution.

It answers the question:

“How do we sort in parallel using predictable, fixed circuits?”

### Example

Sort [3, 7, 4, 8, 6, 2, 1, 5]

| Step | Action                                      | Result             |
|------|---------------------------------------------|--------------------|
| 1    | Build bitonic sequence $\uparrow\downarrow$ | [3,7,4,8, 6,2,1,5] |
| 2    | Compare & swap pairs                        | [3,2,1,5, 6,7,4,8] |

| Step | Action                   | Result             |
|------|--------------------------|--------------------|
| 3    | Merge halves recursively | [1,2,3,4, 5,6,7,8] |

Each phase doubles sorted subsequences until the full array is sorted.

### How Does It Work (Plain Language)?

Imagine you're sorting cards but you can only compare pairs at fixed positions, no adaptive decisions.

You first build a bitonic sequence (up then down), then merge it into a sorted list using bitonic merge steps, repeatedly compare-and-swap mirrored elements.

### Step-by-Step Process

| Step | Description                                       |
|------|---------------------------------------------------|
| 1    | Split array into halves                           |
| 2    | Sort first half ascending, second half descending |
| 3    | Combine them into a bitonic sequence              |
| 4    | Run bitonic merge: recursively compare-swap pairs |
| 5    | Repeat until sequence size = 1                    |

Works best when  $n = 2^k$  (power of two).

### Tiny Code (Easy Versions)

#### Python

```
def bitonic_merge(arr, low, cnt, direction):
    if cnt > 1:
        k = cnt // 2
        for i in range(low, low + k):
            if (direction and arr[i] > arr[i + k]) or (not direction and arr[i] < arr[i + k]):
                arr[i], arr[i + k] = arr[i + k], arr[i]
        bitonic_merge(arr, low, k, direction)
        bitonic_merge(arr, low + k, k, direction)

def bitonic_sort(arr, low=0, cnt=None, direction=True):
```

```

if cnt is None:
    cnt = len(arr)
if cnt > 1:
    k = cnt // 2
    bitonic_sort(arr, low, k, True)
    bitonic_sort(arr, low + k, k, False)
    bitonic_merge(arr, low, cnt, direction)

arr = [3, 7, 4, 8, 6, 2, 1, 5]
bitonic_sort(arr)
print(arr)

```

Output:

[\$\$1, 2, 3, 4, 5, 6, 7, 8]

## C

```

#include <stdio.h>

void swap(int *a, int *b) {
    int t = *a; *a = *b; *b = t;
}

void bitonic_merge(int arr[], int low, int cnt, int dir) {
    if (cnt > 1) {
        int k = cnt / 2;
        for (int i = low; i < low + k; i++) {
            if ((dir && arr[i] > arr[i + k]) || (!dir && arr[i] < arr[i + k]))
                swap(&arr[i], &arr[i + k]);
        }
        bitonic_merge(arr, low, k, dir);
        bitonic_merge(arr, low + k, k, dir);
    }
}

void bitonic_sort(int arr[], int low, int cnt, int dir) {
    if (cnt > 1) {
        int k = cnt / 2;
        bitonic_sort(arr, low, k, 1);
        bitonic_sort(arr, low + k, k, 0);
    }
}

```

```

        bitonic_merge(arr, low, cnt, dir);
    }
}

int main(void) {
    int arr[] = {3, 7, 4, 8, 6, 2, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    bitonic_sort(arr, 0, n, 1);
    for (int i = 0; i < n; i++) printf("%d ", arr[i]);
    printf("\n");
}

```

Output:

1 2 3 4 5 6 7 8

## Why It Matters

- Parallel-friendly (sorting networks)
- Deterministic structure (no branches)
- Perfect for hardware, GPUs, SIMD
- Good educational model for divide and conquer + merging

It's not about runtime on CPUs, it's about parallel depth.

## A Gentle Proof (Why It Works)

### Bitonic sequence:

A sequence that increases then decreases is bitonic.

### Merging rule:

Compare each element with its mirror; recursively merge halves.

At each merge stage, the array becomes more sorted.

Recursion depth =  $\log n$ , each level does  $O(n)$  work  $\rightarrow O(n \log^2 n)$ .

| Step  | Work   | Levels   | Total           |
|-------|--------|----------|-----------------|
| Merge | $O(n)$ | $\log n$ | $O(n \log^2 n)$ |

## Try It Yourself

1. Sort [3,7,4,8,6,2,1,5] manually.
2. Identify bitonic sequences at each stage.
3. Try with 4 elements [4,1,3,2].
4. Change direction flags (ascending/descending).
5. Draw comparison network graph.
6. Implement iterative version.
7. Run on power-of-two sizes.
8. Measure parallel steps vs QuickSort.
9. Experiment with GPU (Numba/CUDA).
10. Visualize recursive structure.

## Test Cases

| Input             | Output            | Notes      |
|-------------------|-------------------|------------|
| [3,7,4,8,6,2,1,5] | [1,2,3,4,5,6,7,8] | Standard   |
| [4,1,3,2]         | [1,2,3,4]         | Small case |
| [5,4,3,2,1,0,9,8] | [0,1,2,3,4,5,8,9] | Reverse    |
| [8,4,2,1,3,6,5,7] | [1,2,3,4,5,6,7,8] | Random     |

## Complexity

| Aspect         | Value           |
|----------------|-----------------|
| Time           | $O(n \log^2 n)$ |
| Space          | $O(1)$          |
| Stable         | No              |
| Adaptive       | No              |
| Parallel Depth | $O(\log^2 n)$   |

Bitonic Sort shines where parallelism rules, in GPUs, circuits, and sorting networks. Every comparison is planned, every move synchronized, a symphony of order in fixed rhythm.

## 147 Odd-Even Merge Sort

Odd-Even Merge Sort is a parallel sorting algorithm and a sorting network that merges two sorted sequences using a fixed pattern of comparisons between odd and even indexed elements.

It was introduced by Ken Batcher, and like Bitonic Sort, it's designed for parallel hardware or SIMD processors, where predictable comparison patterns matter more than data-dependent branching.

### **What Problem Are We Solving??**

Traditional merge algorithms rely on conditional branching, they decide at runtime which element to pick next. This is problematic in parallel or hardware implementations, where you need fixed, predictable sequences of comparisons.

Odd-Even Merge Sort solves this by using a static comparison network that merges sorted halves without branching.

It's perfect when:

- You need deterministic behavior
- You're building parallel circuits or GPU kernels

### **Example**

Merge two sorted halves: [1, 4, 7, 8] and [2, 3, 5, 6]

| Step | Action                       | Result            |
|------|------------------------------|-------------------|
| 1    | Merge odds [1,7] with [2,5]  | [1,2,5,7]         |
| 2    | Merge evens [4,8] with [3,6] | [3,4,6,8]         |
| 3    | Combine and compare adjacent | [1,2,3,4,5,6,7,8] |

Fixed pattern, no branching, merges completed in parallel.

### **How Does It Work (Plain Language)?**

Imagine two zipper chains, one odd, one even. You weave them together in a fixed, interlocking pattern, comparing and swapping along the way. There's no guessing, every element knows which neighbor to check.

| Step | Description |
|------|-------------|
|------|-------------|

### Step-by-Step Process

| Step | Description                                     |
|------|-------------------------------------------------|
| 1    | Split array into left and right halves          |
| 2    | Recursively sort each half                      |
| 3    | Use odd-even merge to combine halves            |
| 4    | Odd-even merge:                                 |
| a.   | Recursively merge odd and even indexed elements |
| b.   | Compare and swap adjacent pairs                 |
| 5    | Continue until array sorted                     |

Works best when ( $n = 2^k$ ).

### Tiny Code (Easy Versions)

#### Python

```
def odd_even_merge(arr, lo, n, direction):
    if n > 1:
        m = n // 2
        odd_even_merge(arr, lo, m, direction)
        odd_even_merge(arr, lo + m, m, direction)
        for i in range(lo + m, lo + n - m):
            if (arr[i] > arr[i + m]) == direction:
                arr[i], arr[i + m] = arr[i + m], arr[i]

def odd_even_merge_sort(arr, lo=0, n=None, direction=True):
    if n is None:
        n = len(arr)
    if n > 1:
        m = n // 2
        odd_even_merge_sort(arr, lo, m, direction)
        odd_even_merge_sort(arr, lo + m, m, direction)
        odd_even_merge(arr, lo, n, direction)

arr = [8, 3, 2, 7, 4, 6, 5, 1]
```

```
odd_even_merge_sort(arr)
print(arr)
```

Output:

```
$$1, 2, 3, 4, 5, 6, 7, 8]
```

C

```
#include <stdio.h>

void swap(int *a, int *b) {
    int t = *a; *a = *b; *b = t;
}

void odd_even_merge(int arr[], int lo, int n, int dir) {
    if (n > 1) {
        int m = n / 2;
        odd_even_merge(arr, lo, m, dir);
        odd_even_merge(arr, lo + m, m, dir);
        for (int i = lo + m; i < lo + n - m; i++) {
            if ((arr[i] > arr[i + m]) == dir)
                swap(&arr[i], &arr[i + m]);
        }
    }
}

void odd_even_merge_sort(int arr[], int lo, int n, int dir) {
    if (n > 1) {
        int m = n / 2;
        odd_even_merge_sort(arr, lo, m, dir);
        odd_even_merge_sort(arr, lo + m, m, dir);
        odd_even_merge(arr, lo, n, dir);
    }
}

int main(void) {
    int arr[] = {8, 3, 2, 7, 4, 6, 5, 1};
    int n = sizeof(arr)/sizeof(arr[0]);
    odd_even_merge_sort(arr, 0, n, 1);
    for (int i = 0; i < n; i++) printf("%d ", arr[i]);
```

```
    printf("\n");
}
```

Output:

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

## Why It Matters

- Fixed sequence, perfect for parallelism
- No data-dependent branching
- Used in hardware sorting networks
- Theoretical foundation for parallel sorting

When you need determinism and concurrency, this algorithm shines.

## A Gentle Proof (Why It Works)

Each odd-even merge merges two sorted sequences using fixed compare-swap operations. At each stage:

- Odd indices are merged separately
- Even indices merged separately
- Adjacent elements compared to restore global order

Each level performs  $O(n)$  work, depth =  $O(\log^2 n) \rightarrow$  total complexity:

$$T(n) = O(n \log^2 n)$$

## Try It Yourself

1. Sort [8,3,2,7,4,6,5,1] step by step.
2. Trace odd-index and even-index merges.
3. Draw merge network diagram.
4. Try smaller [4,3,2,1] for clarity.
5. Run on power-of-two lengths.
6. Measure comparisons.
7. Compare with Bitonic Sort.
8. Implement iterative version.
9. Visualize parallel depth.
10. Experiment with ascending/descending flags.

## Test Cases

| Input             | Output            | Notes        |
|-------------------|-------------------|--------------|
| [8,3,2,7,4,6,5,1] | [1,2,3,4,5,6,7,8] | Classic      |
| [4,3,2,1]         | [1,2,3,4]         | Small        |
| [9,7,5,3,1,2,4,6] | [1,2,3,4,5,6,7,9] | Mixed        |
| [5,4,3,2]         | [2,3,4,5]         | Reverse half |

## Complexity

| Aspect         | Value           |
|----------------|-----------------|
| Time           | $O(n \log^2 n)$ |
| Space          | $O(1)$          |
| Stable         | No              |
| Adaptive       | No              |
| Parallel Depth | $O(\log^2 n)$   |

Odd-Even Merge Sort weaves order from two halves like clockwork, steady, parallel, and predictable. Every comparison is planned, every merge synchronized, it's sorting as architecture.

## 148 Sleep Sort

Sleep Sort is one of the most playful and unconventional algorithms ever invented, it sorts numbers by leveraging time delays. Each element is “slept” for a duration proportional to its value, and when the sleep ends, it prints the number.

In effect, time itself becomes the sorting mechanism.

### What Problem Are We Solving??

While not practical, Sleep Sort offers a fun demonstration of parallelism and asynchronous timing, showing that even sorting can be expressed through temporal order rather than comparisons.

It's often used as a thought experiment to teach concurrency, timing, and creative thinking about problem-solving.

## Example

Sort [3, 1, 4, 2]

| Step | Value | Sleep (seconds) | Print order |
|------|-------|-----------------|-------------|
| 1    | 1     | 1s              | 1           |
| 2    | 2     | 2s              | 2           |
| 3    | 3     | 3s              | 3           |
| 4    | 4     | 4s              | 4           |

Output (over time): 1 2 3 4

Sorted by time of completion!

## How Does It Work (Plain Language)?

Each number is given a timer equal to its value. All timers start simultaneously, and when a timer finishes, that number is output. Small numbers “wake up” first, so they’re printed earlier, creating a sorted sequence.

It’s like a race where each runner’s speed is inversely proportional to its size, smaller ones finish first.

## Step-by-Step Process

| Step | Description                                         |
|------|-----------------------------------------------------|
| 1    | For each element $x$ , create a thread or coroutine |
| 2    | Each thread sleeps for $x$ units of time            |
| 3    | When sleep completes, print $x$                     |
| 4    | Numbers appear in sorted order                      |
| 5    | Optionally collect outputs into a list              |

## Tiny Code (Easy Versions)

### Python (Using Threads)

```

import threading
import time

def sleeper(x):
    time.sleep(x * 0.1) # scale factor for speed
    print(x, end=' ')

def sleep_sort(arr):
    threads = []
    for x in arr:
        t = threading.Thread(target=sleeper, args=(x,))
        t.start()
        threads.append(t)
    for t in threads:
        t.join()

arr = [3, 1, 4, 2]
sleep_sort(arr)

```

Output (timed):

1 2 3 4

## C (Using Threads and Sleep)

```

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

void* sleeper(void* arg) {
    int x = *(int*)arg;
    usleep(x * 100000); // scaled down
    printf("%d ", x);
    return NULL;
}

void sleep_sort(int arr[], int n) {
    pthread_t threads[n];
    for (int i = 0; i < n; i++)

```

```

    pthread_create(&threads[i], NULL, sleeper, &arr[i]);
for (int i = 0; i < n; i++)
    pthread_join(threads[i], NULL);
}

int main(void) {
    int arr[] = {3, 1, 4, 2};
    int n = sizeof(arr) / sizeof(arr[0]);
    sleep_sort(arr, n);
    printf("\n");
}

```

Output (timed):

1 2 3 4

## Why It Matters

- Creative demonstration of parallelism
- Fun teaching tool for concurrency
- Visually intuitive, sorting emerges naturally
- Great reminder: algorithms just code, they're processes

It's impractical, but delightfully educational.

## A Gentle Proof (Why It Works)

If all threads start simultaneously and sleep proportionally to their values, then:

- Smaller values finish earlier
- No collisions (if distinct integers)
- Output sequence = sorted list

For duplicates, slight offsets may be added to maintain stability.

Limitations:

- Requires positive integers
- Depends on accurate timers
- Sensitive to scheduler latency

## Try It Yourself

1. Sort `[3,1,4,2]`, observe timing.
2. Try `[10,5,1,2]`, slower but clearer pattern.
3. Add duplicates `[2,2,1]`, test ordering.
4. Scale sleep time down (`x * 0.05`).
5. Run on multi-core CPU, observe concurrency.
6. Replace sleep with `await asyncio.sleep(x)` for async version.
7. Collect results in a list instead of print.
8. Use `multiprocessing` instead of threads.
9. Visualize time vs value graph.
10. Try fractional delays for floats.

## Test Cases

| Input                  | Output                 | Notes              |
|------------------------|------------------------|--------------------|
| <code>[3,1,4,2]</code> | <code>[1,2,3,4]</code> | Classic example    |
| <code>[1,2,3,4]</code> | <code>[1,2,3,4]</code> | Already sorted     |
| <code>[4,3,2,1]</code> | <code>[1,2,3,4]</code> | Reversed           |
| <code>[2,2,1]</code>   | <code>[1,2,2]</code>   | Handles duplicates |

## Complexity

| Aspect             | Value                         |
|--------------------|-------------------------------|
| Time (Theoretical) | $O(n)$ real-time (wall-clock) |
| Time (CPU Work)    | $O(n)$ setup                  |
| Space              | $O(n)$ threads                |
| Stable             | Yes (with offset)             |
| Adaptive           | No                            |

Sleep Sort is sorting reimaged, not by computation, but by patience. Every number simply waits its turn, no comparisons, no loops, just time.

## 149 Bead Sort

Bead Sort, also known as Gravity Sort, is a natural sorting algorithm inspired by how beads slide under gravity on parallel rods. Imagine an abacus turned on its side: heavier piles settle first, automatically sorting themselves.

It's visual, parallel, and analog in spirit, more of a conceptual model than a practical tool, but brilliant for intuition.

### **What Problem Are We Solving??**

Sorting algorithms usually rely on comparisons. Bead Sort instead uses physical simulation, items fall until they settle into order.

This approach helps visualize distribution sorting and natural computation, where sorting happens through physical laws rather than arithmetic operations.

### **Example**

Sort [5, 3, 1, 7, 4]:

1. Represent each number as a row of beads.
2. Drop beads under gravity.
3. Count beads per column from bottom up.

| Step    | Representation           | After Gravity   | Output |
|---------|--------------------------|-----------------|--------|
| Initial | 5    3    1   7    4     | ,               | ,      |
| Gravity | Columns fill from bottom | Rows shorten    | ,      |
| Result  | 1 3   4   5    7         | [1, 3, 4, 5, 7] |        |

The smallest number rises to top, largest sinks to bottom, sorted.

### **How Does It Work (Plain Language)?**

Each number is a pile of beads. Beads “fall” downward until no empty space below. Since heavier rows push beads downward faster, larger numbers accumulate at the bottom. When gravity stops, reading row lengths from top to bottom yields sorted order.

It's sorting by simulated gravity, no comparisons at all.

---

| Step | Description |
|------|-------------|
|------|-------------|

---

### Step-by-Step Process

---

| Step | Description                                                |
|------|------------------------------------------------------------|
| 1    | Represent each integer by beads on rods (1 bead per unit)  |
| 2    | Let beads fall to the lowest empty position in each column |
| 3    | After settling, count beads per row (top-down)             |
| 4    | These counts form the sorted list                          |

---

Works only for non-negative integers.

### Tiny Code (Easy Versions)

#### Python

```
def bead_sort(arr):
    if any(x < 0 for x in arr):
        raise ValueError("Only non-negative integers allowed")
    max_val = max(arr)
    grid = [[1 if i < x else 0 for i in range(max_val)] for x in arr]
    for col in range(max_val):
        beads = sum(row[col] for row in grid)
        for row in range(len(arr)):
            grid[row][col] = 1 if row >= len(arr) - beads else 0
    return [sum(row) for row in grid]

arr = [5, 3, 1, 7, 4]
print(bead_sort(arr))
```

Output:

[\$\$1, 3, 4, 5, 7]

C

```

#include <stdio.h>
#include <string.h>

void bead_sort(int *a, int n) {
    int max = 0;
    for (int i = 0; i < n; i++) if (a[i] > max) max = a[i];
    unsigned char beads[n][max];
    memset(beads, 0, n * max);
    for (int i = 0; i < n; i++)
        for (int j = 0; j < a[i]; j++)
            beads[i][j] = 1;

    for (int j = 0; j < max; j++) {
        int sum = 0;
        for (int i = 0; i < n; i++) sum += beads[i][j];
        for (int i = 0; i < n; i++)
            beads[i][j] = (i >= n - sum) ? 1 : 0;
    }

    for (int i = 0; i < n; i++) {
        a[i] = 0;
        for (int j = 0; j < max; j++)
            a[i] += beads[i][j];
    }
}

int main(void) {
    int arr[] = {5, 3, 1, 7, 4};
    int n = sizeof(arr) / sizeof(arr[0]);
    bead_sort(arr, n);
    for (int i = 0; i < n; i++) printf("%d ", arr[i]);
    printf("\n");
}

```

Output:

1 3 4 5 7

### Why It Matters

- Demonstrates non-comparison-based sorting

- Shows physical analogies for computation
- Ideal for visual and educational purposes
- Parallelizable (each column independent)

Though impractical, it inspires biological and physics-inspired algorithm design.

### A Gentle Proof (Why It Works)

Each column acts like a gravity channel:

- Beads fall to fill lowest positions
- Columns represent magnitudes across numbers
- After settling, beads in each row = sorted value

No two beads can occupy the same slot twice, ensuring correctness. Complexity:

$$T(n) = O(S)$$

where  $S = \sum a_i$ , total bead count.

Efficient only when numbers are small.

### Try It Yourself

1. Sort [5,3,1,7,4] by hand using dots.
2. Draw rods and let beads fall.
3. Try [3,0,2,1], zeros stay top.
4. Experiment with duplicates [2,2,3].
5. Use grid visualization in Python.
6. Compare with Counting Sort.
7. Extend for stable ordering.
8. Animate bead falling step by step.
9. Scale with numbers 10.
10. Reflect: what if gravity was sideways?

### Test Cases

| Input       | Output      | Notes                 |
|-------------|-------------|-----------------------|
| [5,3,1,7,4] | [1,3,4,5,7] | Classic example       |
| [3,0,2,1]   | [0,1,2,3]   | Handles zeros         |
| [2,2,3]     | [2,2,3]     | Works with duplicates |

| Input | Output | Notes          |
|-------|--------|----------------|
| [1]   | [1]    | Single element |

## Complexity

| Aspect   | Value                                       |
|----------|---------------------------------------------|
| Time     | $O(S)$ , where $S = \text{sum of elements}$ |
| Space    | $O(S)$                                      |
| Stable   | No                                          |
| Adaptive | No                                          |

Bead Sort shows how even gravity can sort, numbers become beads, and time, motion, and matter do the work. It's sorting you can *see*, not just compute.

## 150 Bogo Sort

Bogo Sort (also called Permutation Sort or Stupid Sort) is a deliberately absurd algorithm that repeatedly shuffles the array until it becomes sorted.

It's the poster child of inefficiency, often used in classrooms as a comic counterexample, sorting by pure luck.

### What Problem Are We Solving??

We're not solving a problem so much as demonstrating futility. Bogo Sort asks, "*What if we just kept trying random orders until we got lucky?*"

It's a great teaching tool for:

- Understanding algorithmic inefficiency
- Appreciating complexity bounds
- Learning to recognize good vs. bad strategies

### Example

Sort [3, 1, 2]

| Attempt | Shuffle | Sorted? |
|---------|---------|---------|
| 1       | [3,1,2] | No      |
| 2       | [1,2,3] | Yes     |

Stop when lucky! (You could get lucky early... or never.)

### How Does It Work (Plain Language)?

The idea is painfully simple:

1. Check if the array is sorted.
2. If not, shuffle it randomly.
3. Repeat until sorted.

It's sorting by random chance, not logic. Each attempt has a tiny probability of being sorted, but given infinite time, it *will* finish (eventually).

### Step-by-Step Process

| Step | Description              |
|------|--------------------------|
| 1    | Check if array is sorted |
| 2    | If sorted, done          |
| 3    | Else, shuffle randomly   |
| 4    | Go back to step 1        |

### Tiny Code (Easy Versions)

#### Python

```
import random

def is_sorted(arr):
    return all(arr[i] <= arr[i+1] for i in range(len(arr)-1))

def bogo_sort(arr):
    attempts = 0
    while not is_sorted(arr):
        random.shuffle(arr)
```

```

    attempts += 1
    print("Sorted in", attempts, "attempts")
    return arr

arr = [3, 1, 2]
print(bogo_sort(arr))

```

Output (random):

```

Sorted in 7 attempts
[$$1, 2, 3]

```

## C

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int is_sorted(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        if (arr[i] > arr[i + 1]) return 0;
    }
    return 1;
}

void shuffle(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        int j = rand() % n;
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
}

void bogo_sort(int arr[], int n) {
    int attempts = 0;
    while (!is_sorted(arr, n)) {
        shuffle(arr, n);
        attempts++;
    }
    printf("Sorted in %d attempts\n", attempts);
}

```

```

int main(void) {
    srand(time(NULL));
    int arr[] = {3, 1, 2};
    int n = sizeof(arr) / sizeof(arr[0]);
    bogo_sort(arr, n);
    for (int i = 0; i < n; i++) printf("%d ", arr[i]);
    printf("\n");
}

```

Output (random):

```

Sorted in 12 attempts
1 2 3

```

## Why It Matters

- Humorous cautionary tale, what *not* to do
- Demonstrates expected runtime analysis
- A good way to visualize randomness
- Reinforces need for algorithmic reasoning

It's the algorithmic equivalent of *throwing dice until sorted*, mathematically silly, but conceptually rich.

## A Gentle Proof (Why It Works)

With  $n$  elements, there are  $n!$  permutations.  
Only one is sorted.

Probability of success =  $\frac{1}{n!}$

Expected attempts:

$$E(n) = n!$$

Each check takes  $O(n)$ , so total expected time:

$$T(n) = O(n \times n!)$$

Guaranteed termination (eventually), since the probability of not sorting forever  $\rightarrow 0$ .

## Try It Yourself

1. Run on  $[3, 1, 2]$  and count attempts.
2. Try  $[1, 2, 3]$ , instant success.
3. Test  $[4, 3, 2, 1]$ , likely infinite patience required.
4. Replace `random.shuffle` with deterministic shuffle (see fail).
5. Add a timeout.
6. Visualize shuffles on screen.
7. Measure average attempts over 100 trials.
8. Compare with Bubble Sort.
9. Try “Bogobogosort” (recursive Bogo!).
10. Reflect: what’s the expected runtime for  $n=5$ ?

## Test Cases

| Input          | Output         | Notes                     |
|----------------|----------------|---------------------------|
| $[3, 1, 2]$    | $[1, 2, 3]$    | Classic                   |
| $[1, 2, 3]$    | $[1, 2, 3]$    | Already sorted            |
| $[2, 1]$       | $[1, 2]$       | Fast                      |
| $[4, 3, 2, 1]$ | $[1, 2, 3, 4]$ | Possibly never terminates |

## Complexity

| Aspect          | Value                   |
|-----------------|-------------------------|
| Time (Expected) | $O(n \times n!)$        |
| Time (Best)     | $O(n)$                  |
| Time (Worst)    | Unbounded               |
| Space           | $O(1)$                  |
| Stable          | Yes (if shuffle stable) |
| Adaptive        | No                      |

Bogo Sort is chaos pretending to be order, sorting by faith, not logic. It’s the universe’s reminder that hope isn’t a strategy, not even in algorithms.

## Section 16. Linear and Binary Search

### 151 Linear Search

Linear Search (also known as Sequential Search) is the simplest and most intuitive searching algorithm. It scans through each element one by one until it finds the target, or reaches the end of the list.

It's easy to understand, easy to implement, and works on both sorted and unsorted data.

#### What Problem Are We Solving??

Given a list and a target value, how can we check if the target is present, and if so, at which index?

Linear Search solves this by scanning each element in order until a match is found.

Perfect for:

- Small datasets
- Unsorted arrays
- Early learning of search principles

#### Example

Find 7 in [3, 5, 7, 2, 9]:

| Step | Index | Value | Match?           |
|------|-------|-------|------------------|
| 1    | 0     | 3     | No               |
| 2    | 1     | 5     | No               |
| 3    | 2     | 7     | Yes              |
| 4    | Stop  | ,     | Found at index 2 |

#### How Does It Work (Plain Language)?

It's like flipping through pages one by one looking for a word. No skipping, no guessing, just check everything in order.

If the list is [a, a, a, ..., a]:

1. Start at index 0
2. Compare  $a[i]$  with the target

3. If equal → found
4. Else → move to next
5. Stop when found or end reached

### Step-by-Step Process

| Step | Description                         |
|------|-------------------------------------|
| 1    | Start from index 0                  |
| 2    | Compare current element with target |
| 3    | If equal, return index              |
| 4    | Else, increment index               |
| 5    | Repeat until end                    |
| 6    | If not found, return -1             |

### Tiny Code (Easy Versions)

#### Python

```
def linear_search(arr, target):
    for i, val in enumerate(arr):
        if val == target:
            return i
    return -1

arr = [3, 5, 7, 2, 9]
target = 7
idx = linear_search(arr, target)
print("Found at index:", idx)
```

Output:

Found at index: 2

C

```

#include <stdio.h>

int linear_search(int arr[], int n, int target) {
    for (int i = 0; i < n; i++)
        if (arr[i] == target)
            return i;
    return -1;
}

int main(void) {
    int arr[] = {3, 5, 7, 2, 9};
    int n = sizeof(arr) / sizeof(arr[0]);
    int target = 7;
    int idx = linear_search(arr, n, target);
    if (idx != -1)
        printf("Found at index: %d\n", idx);
    else
        printf("Not found\n");
}

```

Output:

Found at index: 2

## Why It Matters

- Works on any list, sorted or unsorted
- No preprocessing needed
- Guaranteed to find (if present)
- Great introduction to time complexity
- Foundation for better search algorithms

## A Gentle Proof (Why It Works)

If the element exists, scanning each element ensures it will eventually be found.

For ( n ) elements:

- Best case: found at index 0 → O(1)
- Worst case: not found or last → O(n)
- Average case: half-way → O(n)

Because there's no faster way without structure.

### Try It Yourself

1. Search 7 in [3,5,7,2,9]
2. Search 10 (not in list)
3. Try [1,2,3,4,5] with `target=1` (best case)
4. Try `target=5` (worst case)
5. Count comparisons made
6. Print "Found" or "Not Found"
7. Try on unsorted vs sorted arrays
8. Modify to return all indices of target
9. Implement recursive version
10. Extend for string search in list of words

### Test Cases

| Input       | Target | Output | Notes      |
|-------------|--------|--------|------------|
| [3,5,7,2,9] | 7      | 2      | Found      |
| [3,5,7,2,9] | 10     | -1     | Not found  |
| [1,2,3]     | 1      | 0      | Best case  |
| [1,2,3]     | 3      | 2      | Worst case |

### Complexity

| Aspect         | Value  |
|----------------|--------|
| Time (Best)    | $O(1)$ |
| Time (Worst)   | $O(n)$ |
| Time (Average) | $O(n)$ |
| Space          | $O(1)$ |
| Stable         | Yes    |
| Adaptive       | No     |

Linear Search is the simplest lens into algorithmic thinking, brute force but guaranteed. It's your first step from guessing to reasoning.

## 152 Linear Search (Sentinel)

Sentinel Linear Search is a clever twist on the basic Linear Search. Instead of checking array bounds each time, we place a sentinel (a guard value) at the end of the array equal to the target.

This eliminates the need for explicit boundary checks inside the loop, making the search slightly faster and cleaner, especially in low-level languages like C.

### What Problem Are We Solving??

In a standard linear search, each iteration checks both:

1. If the current element equals the target
2. If the index is still within bounds

That second check adds overhead.

By placing a sentinel, we can guarantee the loop will always terminate, no bounds check needed.

This is useful in tight loops, embedded systems, and performance-critical code.

### Example

Find 7 in [3, 5, 7, 2, 9]:

1. Append sentinel (duplicate target) → [3, 5, 7, 2, 9, 7]
2. Scan until 7 is found
3. If index < n, found in array
4. If index == n, only sentinel found → not in array

| Step | Index           | Value      | Match? |
|------|-----------------|------------|--------|
| 1    | 0               | 3          | No     |
| 2    | 1               | 5          | No     |
| 3    | 2               | 7          | Yes    |
| 4    | Stop, index < n | Found at 2 |        |

## How Does It Work (Plain Language)?

Think of the sentinel as a stop sign placed beyond the last element. You don't have to look over your shoulder to check if you've gone too far, the sentinel will catch you.

It ensures you'll always hit a match, but then you check whether it was a real match or the sentinel.

## Step-by-Step Process

| Step | Description                              |
|------|------------------------------------------|
| 1    | Save last element                        |
| 2    | Place target at the end (sentinel)       |
| 3    | Loop until <code>arr[i] == target</code> |
| 4    | Restore last element                     |
| 5    | If index < n → found, else → not found   |

## Tiny Code (Easy Versions)

C

```
#include <stdio.h>

int sentinel_linear_search(int arr[], int n, int target) {
    int last = arr[n - 1];
    arr[n - 1] = target; // sentinel
    int i = 0;
    while (arr[i] != target)
        i++;
    arr[n - 1] = last; // restore
    if (i < n - 1 || arr[n - 1] == target)
        return i;
    return -1;
}

int main(void) {
    int arr[] = {3, 5, 7, 2, 9};
    int n = sizeof(arr) / sizeof(arr[0]);
    int target = 7;
    int idx = sentinel_linear_search(arr, n, target);
```

```

if (idx != -1)
    printf("Found at index: %d\n", idx);
else
    printf("Not found\n");
}

```

Output:

Found at index: 2

### Python (Simulated)

```

def sentinel_linear_search(arr, target):
    n = len(arr)
    last = arr[-1]
    arr[-1] = target
    i = 0
    while arr[i] != target:
        i += 1
    arr[-1] = last
    if i < n - 1 or arr[-1] == target:
        return i
    return -1

arr = [3, 5, 7, 2, 9]
print(sentinel_linear_search(arr, 7)) # Output: 2

```

### Why It Matters

- Removes boundary check overhead
- Slightly faster for large arrays
- Classic example of sentinel optimization
- Teaches loop invariants and guard conditions

This is how you make a simple algorithm tight and elegant.

## A Gentle Proof (Why It Works)

By placing the target as the last element:

- Loop must terminate (guaranteed match)
- Only after the loop do we check if it was sentinel or real match

No wasted comparisons. Total comparisons  $n + 1$  (vs  $2n$  in naive version).

$$T(n) = O(n)$$

## Try It Yourself

1. Search 7 in [3,5,7,2,9]
2. Search 10 (not in list)
3. Track number of comparisons vs regular linear search
4. Implement in Python, Java, C++
5. Visualize sentinel placement
6. Use array of 1000 random elements, benchmark
7. Try replacing last element temporarily
8. Search first element, check best case
9. Search last element, check sentinel restore
10. Discuss when this optimization helps most

## Test Cases

| Input       | Target | Output | Notes                  |
|-------------|--------|--------|------------------------|
| [3,5,7,2,9] | 7      | 2      | Found                  |
| [3,5,7,2,9] | 10     | -1     | Not found              |
| [1,2,3]     | 1      | 0      | Best case              |
| [1,2,3]     | 3      | 2      | Sentinel replaced last |

## Complexity

| Aspect         | Value  |
|----------------|--------|
| Time (Best)    | $O(1)$ |
| Time (Worst)   | $O(n)$ |
| Time (Average) | $O(n)$ |

| Aspect   | Value |
|----------|-------|
| Space    | O(1)  |
| Stable   | Yes   |
| Adaptive | No    |

Sentinel Linear Search is how you turn simplicity into elegance, one tiny guard makes the whole loop smarter.

### 153 Binary Search (Iterative)

Binary Search (Iterative) is one of the most elegant and efficient searching algorithms for sorted arrays. It repeatedly divides the search interval in half, eliminating half the remaining elements at each step.

This version uses a loop, avoiding recursion and keeping memory usage minimal.

#### What Problem Are We Solving??

When working with sorted data, a linear scan is wasteful. If you always know the list is ordered, you can use binary search to find your target in  $O(\log n)$  time instead of  $O(n)$ .

#### Example

Find 7 in [1, 3, 5, 7, 9, 11]:

| Step | Low | High | Mid | Value | Compare                          |
|------|-----|------|-----|-------|----------------------------------|
| 1    | 0   | 5    | 2   | 5     | $7 > 5 \rightarrow$ search right |
| 2    | 3   | 5    | 4   | 9     | $7 < 9 \rightarrow$ search left  |
| 3    | 3   | 3    | 3   | 7     | Found                            |

Found at index 3.

## How Does It Work (Plain Language)?

Binary Search is like guessing a number between 1 and 100:

- Always pick the midpoint.
- If the number is too low, search the upper half.
- If it's too high, search the lower half.
- Repeat until found or interval is empty.

Each guess cuts the space in half, that's why it's so fast.

## Step-by-Step Process

| Step | Description                                                                             |
|------|-----------------------------------------------------------------------------------------|
| 1    | Start with <code>low = 0, high = n - 1</code>                                           |
| 2    | While <code>low &lt;= high</code> , find <code>mid = (low + high) // 2</code>           |
| 3    | If <code>arr[mid] == target</code> → return index                                       |
| 4    | If <code>arr[mid] &lt; target</code> → search right half ( <code>low = mid + 1</code> ) |
| 5    | Else → search left half ( <code>high = mid - 1</code> )                                 |
| 6    | If not found, return -1                                                                 |

## Tiny Code (Easy Versions)

### Python

```
def binary_search(arr, target):
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1

arr = [1, 3, 5, 7, 9, 11]
print(binary_search(arr, 7)) # Output: 3
```

Output:

**C**

```
#include <stdio.h>

int binary_search(int arr[], int n, int target) {
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2; // avoid overflow
        if (arr[mid] == target)
            return mid;
        else if (arr[mid] < target)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
}

int main(void) {
    int arr[] = {1, 3, 5, 7, 9, 11};
    int n = sizeof(arr) / sizeof(arr[0]);
    int idx = binary_search(arr, n, 7);
    if (idx != -1)
        printf("Found at index: %d\n", idx);
    else
        printf("Not found\n");
}
```

Output:

Found at index: 3

### Why It Matters

- Fundamental divide-and-conquer algorithm
- $O(\log n)$  time complexity
- Used everywhere: search engines, databases, compilers
- Builds intuition for binary decision trees

This is the first truly efficient search most programmers learn.

## A Gentle Proof (Why It Works)

At each step, the search interval halves.

After  $k$  steps, remaining elements =  $\frac{n}{2^k}$ .

Stop when  $\frac{n}{2^k} = 1$

$$k = \log_2 n$$

So, total comparisons:

$$T(n) = O(\log n)$$

Works only on sorted arrays.

## Try It Yourself

1. Search 7 in [1,3,5,7,9,11]
2. Search 2 (not found)
3. Trace values of `low`, `high`, `mid`
4. Try on even-length array [1,2,3,4,5,6]
5. Try on odd-length array [1,2,3,4,5]
6. Compare iteration count with linear search
7. Implement recursive version
8. Use binary search to find insertion point
9. Add counter to measure steps
10. Explain why sorting is required

## Test Cases

| Input          | Target | Output | Notes         |
|----------------|--------|--------|---------------|
| [1,3,5,7,9,11] | 7      | 3      | Found         |
| [1,3,5,7,9,11] | 2      | -1     | Not found     |
| [1,2,3,4,5]    | 1      | 0      | First element |
| [1,2,3,4,5]    | 5      | 4      | Last element  |

## Complexity

| Aspect       | Value    |
|--------------|----------|
| Time (Best)  | O(1)     |
| Time (Worst) | O(log n) |

| Aspect         | Value        |
|----------------|--------------|
| Time (Average) | $O(\log n)$  |
| Space          | $O(1)$       |
| Stable         | Yes          |
| Adaptive       | No           |
| Prerequisite   | Sorted array |

Binary Search (Iterative) is the gold standard of efficiency, halving your problem at every step, one decision at a time.

## 154 Binary Search (Recursive)

Binary Search (Recursive) is the classic divide-and-conquer form of binary search. Instead of looping, it calls itself on smaller subarrays, each time halving the search space until the target is found or the interval becomes empty.

It's a perfect demonstration of recursion in action, each call tackles a smaller slice of the problem.

### What Problem Are We Solving??

Given a sorted array, we want to find a target value efficiently. Rather than scanning linearly, we repeatedly split the array in half, focusing only on the half that could contain the target.

This version expresses that logic through recursive calls.

### Example

Find 7 in [1, 3, 5, 7, 9, 11]

| Step | Low | High | Mid | Value | Action                                |
|------|-----|------|-----|-------|---------------------------------------|
| 1    | 0   | 5    | 2   | 5     | $7 > 5 \rightarrow$ search right half |
| 2    | 3   | 5    | 4   | 9     | $7 < 9 \rightarrow$ search left half  |
| 3    | 3   | 3    | 3   | 7     | Found                                 |

Recursive calls shrink the interval each time until match is found.

## How Does It Work (Plain Language)?

Binary search says:

“If the middle element isn’t what I want, I can ignore half the data.”

In recursive form:

1. Check the midpoint.
2. If equal → found.
3. If smaller → recurse right.
4. If larger → recurse left.
5. Base case: if `low > high`, element not found.

Each call reduces the search space by half, logarithmic depth recursion.

## Step-by-Step Process

| Step | Description                                                  |
|------|--------------------------------------------------------------|
| 1    | Check base case: if <code>low &gt; high</code> , return -1   |
| 2    | Compute <code>mid = (low + high) // 2</code>                 |
| 3    | If <code>arr[mid] == target</code> , return <code>mid</code> |
| 4    | If <code>arr[mid] &gt; target</code> , recurse left half     |
| 5    | Else, recurse right half                                     |

## Tiny Code (Easy Versions)

### Python

```
def binary_search_recursive(arr, target, low, high):
    if low > high:
        return -1
    mid = (low + high) // 2
    if arr[mid] == target:
        return mid
    elif arr[mid] < target:
        return binary_search_recursive(arr, target, mid + 1, high)
    else:
        return binary_search_recursive(arr, target, low, mid - 1)
```

```
arr = [1, 3, 5, 7, 9, 11]
print(binary_search_recursive(arr, 7, 0, len(arr)-1)) # Output: 3
```

Output:

3

C

```
#include <stdio.h>

int binary_search_recursive(int arr[], int low, int high, int target) {
    if (low > high)
        return -1;
    int mid = low + (high - low) / 2;
    if (arr[mid] == target)
        return mid;
    else if (arr[mid] < target)
        return binary_search_recursive(arr, mid + 1, high, target);
    else
        return binary_search_recursive(arr, low, mid - 1, target);
}

int main(void) {
    int arr[] = {1, 3, 5, 7, 9, 11};
    int n = sizeof(arr) / sizeof(arr[0]);
    int idx = binary_search_recursive(arr, 0, n - 1, 7);
    if (idx != -1)
        printf("Found at index: %d\n", idx);
    else
        printf("Not found\n");
}
```

Output:

Found at index: 3

## Why It Matters

- Elegant divide-and-conquer demonstration
- Shows recursion depth =  $\log(n)$
- Same complexity as iterative version
- Lays foundation for recursive algorithms (merge sort, quicksort)

Recursion mirrors the mathematical idea of halving the interval, clean and intuitive.

## A Gentle Proof (Why It Works)

At each recursive call:

- Problem size halves:  $n \rightarrow n/2 \rightarrow n/4 \rightarrow \dots$
- Recursion stops after  $\log_2 n$  levels

Thus total complexity:

$$T(n) = T(n/2) + O(1) = O(\log n)$$

Correctness follows from:

- Sorted input ensures ordering decisions are valid
- Base case ensures termination

## Try It Yourself

1. Search 7 in [1,3,5,7,9,11]
2. Search 2 (not present)
3. Add print statements to trace recursion
4. Compare call count vs iterative version
5. Test base case with `low > high`
6. Search first and last element
7. Observe recursion depth for size 8 → 3 calls
8. Add memo of (`low, high`) per step
9. Implement tail-recursive variant
10. Reflect on stack vs loop tradeoffs

## Test Cases

| Input          | Target | Output | Notes         |
|----------------|--------|--------|---------------|
| [1,3,5,7,9,11] | 7      | 3      | Found         |
| [1,3,5,7,9,11] | 2      | -1     | Not found     |
| [1,2,3,4,5]    | 1      | 0      | First element |
| [1,2,3,4,5]    | 5      | 4      | Last element  |

## Complexity

| Aspect         | Value                         |
|----------------|-------------------------------|
| Time (Best)    | $O(1)$                        |
| Time (Worst)   | $O(\log n)$                   |
| Time (Average) | $O(\log n)$                   |
| Space          | $O(\log n)$ (recursion stack) |
| Stable         | Yes                           |
| Adaptive       | No                            |
| Prerequisite   | Sorted array                  |

Binary Search (Recursive) is divide-and-conquer at its purest, halving the world each time, until the answer reveals itself.

## 155 Binary Search (Lower Bound)

Lower Bound Binary Search is a variant of binary search that finds the first position where a value could be inserted without breaking the sorted order. In other words, it returns the index of the first element greater than or equal to the target.

It's used extensively in search engines, databases, range queries, and C++ STL (`std::lower_bound`).

### What Problem Are We Solving??

In many cases, you don't just want to know if an element exists — you want to know where it would go in a sorted structure.

- If the value exists, return its first occurrence.
- If it doesn't exist, return the position where it could be inserted to keep the array sorted.

This is essential for insertion, frequency counting, and range boundaries.

## Example

Find lower bound of 7 in [1, 3, 5, 7, 7, 9, 11]:

| Step | Low         | High | Mid | Value | Compare       | Action        |
|------|-------------|------|-----|-------|---------------|---------------|
| 1    | 0           | 6    | 3   | 7     | arr[mid] >= 7 | move high = 3 |
| 2    | 0           | 2    | 1   | 3     | arr[mid] < 7  | move low = 2  |
| 3    | 2           | 3    | 2   | 5     | arr[mid] < 7  | move low = 3  |
| 4    | low == high | ,    | ,   | ,     | Stop          |               |

Result: Index 3 (first 7)

## How Does It Work (Plain Language)?

Lower bound finds the leftmost slot where the target fits. It slides the search window until `low == high`, with `low` marking the first candidate target.

You can think of it as:

“How far left can I go while still being target?”

## Step-by-Step Process

| Step | Description                                                                                                                                                                                         |
|------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1    | Set <code>low = 0, high = n</code> (note: <code>high = n</code> , not <code>n-1</code> )                                                                                                            |
| 2    | While <code>low &lt; high</code> :<br>a. <code>mid = (low + high) // 2</code><br>b. If <code>arr[mid] &lt; target</code> , move <code>low = mid + 1</code><br>c. Else, move <code>high = mid</code> |
| 3    | When loop ends, <code>low</code> is the lower bound index                                                                                                                                           |

## Tiny Code (Easy Versions)

### Python

```

def lower_bound(arr, target):
    low, high = 0, len(arr)
    while low < high:
        mid = (low + high) // 2
        if arr[mid] < target:
            low = mid + 1
        else:
            high = mid
    return low

arr = [1, 3, 5, 7, 7, 9, 11]
print(lower_bound(arr, 7)) # Output: 3

```

Output:

3

## C

```

#include <stdio.h>

int lower_bound(int arr[], int n, int target) {
    int low = 0, high = n;
    while (low < high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] < target)
            low = mid + 1;
        else
            high = mid;
    }
    return low;
}

int main(void) {
    int arr[] = {1, 3, 5, 7, 7, 9, 11};
    int n = sizeof(arr) / sizeof(arr[0]);
    int idx = lower_bound(arr, n, 7);
    printf("Lower bound index: %d\n", idx);
}

```

Output:

Lower bound index: 3

## Why It Matters

- Finds insertion position for sorted arrays
- Useful in binary search trees, maps, intervals
- Core of range queries (like count of elements  $\leq x$ )
- Builds understanding of boundary binary search

When you need more than yes/no, you need where, use lower bound.

## A Gentle Proof (Why It Works)

Invariant:

- All indices before `low` contain elements  $< \text{target}$
- All indices after `high` contain elements  $\geq \text{target}$

Loop maintains invariant until convergence:

$$\text{low} = \text{high} = \text{first index where } \text{arr}[\text{i}] \geq \text{target}$$

Complexity:

$$T(n) = O(\log n)$$

## Try It Yourself

1. `arr = [1,3,5,7,7,9], target = 7`  $\rightarrow 3$
2. `target = 6`  $\rightarrow 3$  (would insert before first 7)
3. `target = 10`  $\rightarrow 6$  (end)
4. `target = 0`  $\rightarrow 0$  (front)
5. Count elements  $7: \text{len(arr)} - \text{lower_bound(arr, 7)}$
6. Compare with `bisect_left` in Python
7. Use to insert elements while keeping list sorted
8. Apply to sorted strings `["apple", "banana", "cherry"]`
9. Visualize range  $[\text{lower}, \text{upper})$  for duplicates
10. Benchmark vs linear scan for large n

## Test Cases

| Input            | Target | Output | Meaning         |
|------------------|--------|--------|-----------------|
| [1,3,5,7,7,9,11] | 7      | 3      | First 7         |
| [1,3,5,7,7,9,11] | 6      | 3      | Insert before 7 |
| [1,3,5,7,7,9,11] | 12     | 7      | End position    |
| [1,3,5,7,7,9,11] | 0      | 0      | Front           |

## Complexity

| Aspect       | Value        |
|--------------|--------------|
| Time         | $O(\log n)$  |
| Space        | $O(1)$       |
| Stable       | Yes          |
| Adaptive     | No           |
| Prerequisite | Sorted array |

Binary Search (Lower Bound) is how algorithms learn *where* things belong, not just if they exist. It's the precise edge of order.

## 156 Binary Search (Upper Bound)

Upper Bound Binary Search is a close cousin of Lower Bound, designed to find the first index where an element is greater than the target. It's used to locate the right boundary of equal elements or the insertion point after duplicates.

In simpler words:

It finds “the spot just after the last occurrence” of the target.

## What Problem Are We Solving??

Sometimes you need to know where to insert a value after existing duplicates. For example, in frequency counting or range queries, you might want the end of a block of identical elements.

Upper bound returns that exact position, the smallest index where

$$\text{arr}[i] > \text{target}$$

## Example

Find upper bound of 7 in [1, 3, 5, 7, 7, 9, 11]:

| Step | Low         | High | Mid | Value | Compare | Action        |
|------|-------------|------|-----|-------|---------|---------------|
| 1    | 0           | 7    | 3   | 7     | 7 > 7   | move low = 4  |
| 2    | 4           | 7    | 5   | 9     | 9 > 7   | move high = 5 |
| 3    | 4           | 5    | 4   | 7     | 7 > 7   | move low = 5  |
| 4    | low == high | ,    | ,   | ,     | Stop    |               |

Result: Index 5 → position after last 7.

## How Does It Work (Plain Language)?

If Lower Bound finds “the first target,” then Upper Bound finds “the first  $>$  target.”

Together, they define equal ranges:

lower\_bound, upper\_bound) \$\$ \rightarrow \text{all elements equal to the target.}

## Step-by-Step Process

| Step | Description                                                                                                                                                                                          |
|------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1    | Set <code>low = 0, high = n</code>                                                                                                                                                                   |
| 2    | While <code>low &lt; high</code> :<br>a. <code>mid = (low + high) // 2</code><br>b. If <code>arr[mid] &lt;= target</code> , move <code>low = mid + 1</code><br>c. Else, move <code>high = mid</code> |
| 3    | When loop ends, <code>low</code> is the upper bound index                                                                                                                                            |

## Tiny Code (Easy Versions)

### Python

```

def upper_bound(arr, target):
    low, high = 0, len(arr)
    while low < high:
        mid = (low + high) // 2
        if arr[mid] <= target:
            low = mid + 1
        else:
            high = mid
    return low

arr = [1, 3, 5, 7, 7, 9, 11]
print(upper_bound(arr, 7)) # Output: 5

```

Output:

5

## C

```

#include <stdio.h>

int upper_bound(int arr[], int n, int target) {
    int low = 0, high = n;
    while (low < high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] <= target)
            low = mid + 1;
        else
            high = mid;
    }
    return low;
}

int main(void) {
    int arr[] = {1, 3, 5, 7, 7, 9, 11};
    int n = sizeof(arr) / sizeof(arr[0]);
    int idx = upper_bound(arr, n, 7);
    printf("Upper bound index: %d\n", idx);
}

```

Output:

Upper bound index: 5

## Why It Matters

- Locates end of duplicate block
- Enables range counting:  $\text{count} = \text{upper\_bound} - \text{lower\_bound}$
- Used in maps, sets, STL containers
- Key component in range queries and interval merging

It's the *right-hand anchor* of sorted intervals.

## A Gentle Proof (Why It Works)

Invariant:

- All indices before `low` contain  $\leq \text{target}$
- All indices after `high` contain  $> \text{target}$

When `low == high`, it's the smallest index where `arr[i] > target`.

Number of steps =  $\log_2 n$

→ Complexity:

$$T(n) = O(\log n)$$

## Try It Yourself

1. `arr = [1,3,5,7,7,9], target=7` → 5
2. `target=6` → 3 (insert after 5)
3. `target=10` → 6 (end)
4. `target=0` → 0 (front)
5. Count elements equal to 7 → `upper - lower`
6. Compare with `bisect_right` in Python
7. Use to insert new element after duplicates
8. Combine with lower bound to find range
9. Visualize  $[\text{lower}, \text{upper}]$  range
10. Apply to floating point sorted list

## Test Cases

| Input            | Target | Output | Meaning        |
|------------------|--------|--------|----------------|
| [1,3,5,7,7,9,11] | 7      | 5      | After last 7   |
| [1,3,5,7,7,9,11] | 6      | 3      | Insert after 5 |
| [1,3,5,7,7,9,11] | 12     | 7      | End position   |
| [1,3,5,7,7,9,11] | 0      | 0      | Front          |

## Complexity

| Aspect       | Value        |
|--------------|--------------|
| Time         | $O(\log n)$  |
| Space        | $O(1)$       |
| Stable       | Yes          |
| Adaptive     | No           |
| Prerequisite | Sorted array |

Binary Search (Upper Bound) is how you find where “greater than” begins, the right edge of equality, the step beyond the last twin.

## 157 Exponential Search

Exponential Search is a hybrid search algorithm that quickly locates the range where a target might lie, then uses binary search inside that range.

It’s ideal when searching unbounded or very large sorted arrays, especially when the size is unknown or dynamic (like data streams or infinite arrays).

### What Problem Are We Solving??

In a standard binary search, you need the array size. But what if the size is unknown, or huge?

Exponential Search solves this by:

1. Quickly finding an interval where the target could exist.
2. Performing binary search within that interval.

This makes it great for:

- Infinite arrays (conceptual)
- Streams

- Linked structures with known order
- Large sorted data where bounds are costly

### Example

Find 15 in [1, 2, 4, 8, 16, 32, 64, 128]

| Step | Range          | Value                       | Compare             | Action      |
|------|----------------|-----------------------------|---------------------|-------------|
| 1    | index 1        | 2                           | $2 < 15$            | ex-<br>pand |
| 2    | index 2        | 4                           | $4 < 15$            | ex-<br>pand |
| 3    | index 4        | 16                          | $16 > 15$           | stop        |
| 4    | Range = [2, 4] | Binary Search in [4, 8, 16] | Found 15 at index 4 |             |

We doubled the bound each time (1, 2, 4, 8...) until we passed the target.

### How Does It Work (Plain Language)?

Think of it like zooming in:

- Start small, double your step size until you overshoot the target.
- Once you've bracketed it, zoom in with binary search.

This avoids scanning linearly when the array could be massive.

### Step-by-Step Process

| Step | Description                                                    |
|------|----------------------------------------------------------------|
| 1    | Start with index 1                                             |
| 2    | While $i < n$ and $\text{arr}[i] < \text{target}$ , double $i$ |
| 3    | Now target $[i/2, \min(i, n-1)]$                               |
| 4    | Apply binary search in that subrange                           |
| 5    | Return found index or -1                                       |

## Tiny Code (Easy Versions)

### Python

```
def binary_search(arr, target, low, high):
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1

def exponential_search(arr, target):
    if arr[0] == target:
        return 0
    i = 1
    n = len(arr)
    while i < n and arr[i] < target:
        i *= 2
    return binary_search(arr, target, i // 2, min(i, n - 1))

arr = [1, 2, 4, 8, 16, 32, 64, 128]
print(exponential_search(arr, 16)) # Output: 4
```

Output:

4

### C

```
#include <stdio.h>

int binary_search(int arr[], int low, int high, int target) {
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == target)
            return mid;
        else if (arr[mid] < target)
```

```

        low = mid + 1;
    else
        high = mid - 1;
    }
    return -1;
}

int exponential_search(int arr[], int n, int target) {
    if (arr[0] == target)
        return 0;
    int i = 1;
    while (i < n && arr[i] < target)
        i *= 2;
    int low = i / 2;
    int high = (i < n) ? i : n - 1;
    return binary_search(arr, low, high, target);
}

int main(void) {
    int arr[] = {1, 2, 4, 8, 16, 32, 64, 128};
    int n = sizeof(arr) / sizeof(arr[0]);
    int idx = exponential_search(arr, n, 16);
    printf("Found at index: %d\n", idx);
}

```

Output:

Found at index: 4

## Why It Matters

- Works with unknown or infinite size arrays
- Faster than linear scan for large  $n$
- Combines doubling search and binary search
- Used in streaming data structures, file systems, unbounded containers

It's the searcher's flashlight, shine brighter until you see your target.

## A Gentle Proof (Why It Works)

Doubling creates at most  $\log_2 p$  expansions, where  $p$  is the position of the target.

Then binary search on a range of size  $O(p)$  takes another  $O(\log p)$ .

So total time:

$$T(n) = O(\log p)$$

Asymptotically equal to binary search when  $p \ll n$ .

## Try It Yourself

1. Search 16 in  $[1,2,4,8,16,32,64,128]$
2. Search 3 → not found
3. Trace expansions: 1,2,4,8...
4. Try  $[10,20,30,40,50,60,70]$  with `target=60`
5. Modify doubling factor (try 3x)
6. Compare with simple binary search
7. Implement recursive exponential search
8. Use for unknown-size input stream
9. Measure expansion count
10. Visualize ranges on number line

## Test Cases

| Input                | Target | Output | Notes         |
|----------------------|--------|--------|---------------|
| $[1,2,4,8,16,32,64]$ | 16     | 4      | Classic       |
| $[1,2,4,8,16,32,64]$ | 3      | -1     | Not found     |
| $[2,4,6,8]$          | 2      | 0      | First element |
| $[2,4,6,8]$          | 10     | -1     | Out of range  |

## Complexity

| Aspect | Value                                        |
|--------|----------------------------------------------|
| Time   | $O(\log p)$ , where $p$ = position of target |
| Space  | $O(1)$                                       |
| Stable | Yes                                          |

| Aspect       | Value        |
|--------------|--------------|
| Adaptive     | Partially    |
| Prerequisite | Sorted array |

Exponential Search is how you find your way in the dark, double your reach, then look closely where the light lands.

## 158 Jump Search

Jump Search is a simple improvement over Linear Search, designed for sorted arrays. It works by jumping ahead in fixed-size steps instead of checking every element, then performing a linear scan within the block where the target might be.

It trades a bit of extra logic for a big win in speed, especially when data is sorted and random access is cheap.

### What Problem Are We Solving??

Linear search checks each element one by one, slow for large arrays. Jump Search improves on this by “skipping ahead” in fixed jumps, so it makes fewer comparisons overall.

It's great for:

- Sorted arrays
- Fast random access (like arrays, not linked lists)
- Simple, predictable search steps

### Example

Find 9 in [1, 3, 5, 7, 9, 11, 13, 15]

Array size = 8 → Jump size =  $\sqrt{8} = 2$  or 3

| Step | Jump to Index      | Value   | Compare | Action       |
|------|--------------------|---------|---------|--------------|
| 1    | 2                  | 5       | 5 < 9   | jump forward |
| 2    | 4                  | 9       | 9 == 9  | stop jump    |
| 3    | Linear scan from 2 | 5, 7, 9 | found 9 |              |

Found at index 4.

## How Does It Work (Plain Language)?

Imagine you're reading a sorted list of numbers. Instead of reading one number at a time, you skip ahead every few steps, like jumping stairs. Once you overshoot or match, you walk back linearly within that block.

Jump size  $\sqrt{n}$  gives a good balance between jumps and scans.

## Step-by-Step Process

| Step | Description                                                   |
|------|---------------------------------------------------------------|
| 1    | Choose jump size <code>step = <math>\sqrt{n}</math></code>    |
| 2    | Jump ahead while <code>arr[min(step, n)-1] &lt; target</code> |
| 3    | Once overshoot, do linear search in the previous block        |
| 4    | If found, return index, else -1                               |

## Tiny Code (Easy Versions)

### Python

```
import math

def jump_search(arr, target):
    n = len(arr)
    step = int(math.sqrt(n))
    prev = 0

    # Jump ahead
    while prev < n and arr[min(step, n) - 1] < target:
        prev = step
        step += int(math.sqrt(n))
        if prev >= n:
            return -1

    # Linear search within block
    for i in range(prev, min(step, n)):
        if arr[i] == target:
            return i
    return -1
```

```
arr = [1, 3, 5, 7, 9, 11, 13, 15]
print(jump_search(arr, 9)) # Output: 4
```

Output:

4

C

```
#include <stdio.h>
#include <math.h>

int jump_search(int arr[], int n, int target) {
    int step = sqrt(n);
    int prev = 0;

    while (prev < n && arr[(step < n ? step : n) - 1] < target) {
        prev = step;
        step += sqrt(n);
        if (prev >= n) return -1;
    }

    for (int i = prev; i < (step < n ? step : n); i++) {
        if (arr[i] == target) return i;
    }
    return -1;
}

int main(void) {
    int arr[] = {1, 3, 5, 7, 9, 11, 13, 15};
    int n = sizeof(arr) / sizeof(arr[0]);
    int idx = jump_search(arr, n, 9);
    printf("Found at index: %d\n", idx);
}
```

Output:

Found at index: 4

## Why It Matters

- Faster than Linear Search for sorted arrays
- Simpler to implement than Binary Search
- Good for systems with non-random access penalties
- Balance of jumps and scans minimizes comparisons

Best of both worlds: quick leaps and gentle steps.

## A Gentle Proof (Why It Works)

Let jump size =  $m$ .

We perform at most  $\frac{n}{m}$  jumps and  $m$  linear steps.

Total cost:

$$T(n) = O\left(\frac{n}{m} + m\right)$$

Minimized when  $m = \sqrt{n}$

Thus:

$$T(n) = O(\sqrt{n})$$

## Try It Yourself

1. Search 11 in [1,3,5,7,9,11,13,15]
2. Search 2 (not found)
3. Use step = 2, 3, 4 → compare jumps
4. Implement recursive version
5. Visualize jumps on paper
6. Try unsorted array (observe failure)
7. Search edge cases: first, last, middle
8. Use different step size formula
9. Combine with exponential jump
10. Compare with binary search for timing

| Input | Target | Output | Notes |
|-------|--------|--------|-------|
|-------|--------|--------|-------|

## Test Cases

| Input                | Target | Output | Notes         |
|----------------------|--------|--------|---------------|
| [1,3,5,7,9,11,13,15] | 9      | 4      | Found         |
| [1,3,5,7,9,11,13,15] | 2      | -1     | Not found     |
| [2,4,6,8,10]         | 10     | 4      | Last element  |
| [2,4,6,8,10]         | 2      | 0      | First element |

## Complexity

| Aspect       | Value         |
|--------------|---------------|
| Time         | $O(\sqrt{n})$ |
| Space        | $O(1)$        |
| Stable       | Yes           |
| Adaptive     | No            |
| Prerequisite | Sorted array  |

Jump Search is like hopscotch on sorted ground, leap smartly, then step carefully once you're close.

## 159 Fibonacci Search

Fibonacci Search is a divide-and-conquer search algorithm that uses Fibonacci numbers to determine probe positions, rather than midpoints like Binary Search. It's especially efficient for sorted arrays stored in sequential memory, where element access cost grows with distance (for example, on magnetic tape or cache-sensitive systems).

It's a clever twist on binary search, using Fibonacci numbers instead of powers of two.

### What Problem Are We Solving??

In Binary Search, the midpoint splits the array evenly. In Fibonacci Search, we split using Fibonacci ratios, which keeps indices aligned to integer arithmetic, no division needed, and better cache locality in some hardware.

This is particularly useful when:

- Access cost depends on distance
- Memory access is sequential or limited
- We want to avoid division and floating-point math

## Example

Find 8 in [1, 3, 5, 8, 13, 21, 34]

| Step | Fib(k) | Index Checked | Value | Compare | Action     |
|------|--------|---------------|-------|---------|------------|
| 1    | 8      | 5             | 13    | 13 > 8  | move left  |
| 2    | 5      | 2             | 5     | 5 < 8   | move right |
| 3    | 3      | 3             | 8     | 8 = 8   | found      |

Fibonacci numbers: 1, 2, 3, 5, 8, 13, ... We reduce the search space using previous Fibonacci values, like Fibonacci decomposition.

## How Does It Work (Plain Language)?

Think of Fibonacci Search as binary search guided by Fibonacci jumps. At each step:

- You compare the element at index `offset + Fib(k-2)`
- Depending on the result, you move left or right, using smaller Fibonacci numbers to define new intervals.

You “walk down” the Fibonacci sequence until the range collapses.

## Step-by-Step Process

| Step | Description                                                               |
|------|---------------------------------------------------------------------------|
| 1    | Generate smallest Fibonacci number n                                      |
| 2    | Use <code>Fib(k-2)</code> as probe index                                  |
| 3    | Compare target with <code>arr[offset + Fib(k-2)]</code>                   |
| 4    | If smaller, move left (reduce by <code>Fib(k-2)</code> )                  |
| 5    | If larger, move right (increase offset, reduce by <code>Fib(k-1)</code> ) |
| 6    | Continue until <code>Fib(k) = 1</code>                                    |
| 7    | Check last element if needed                                              |

## Tiny Code (Easy Versions)

### Python

```
def fibonacci_search(arr, target):
    n = len(arr)
    fibMMm2 = 0 # (m-2)'th Fibonacci
    fibMMm1 = 1 # (m-1)'th Fibonacci
    fibM = fibMMm1 + fibMMm2 # m'th Fibonacci

    # Find smallest Fibonacci >= n
    while fibM < n:
        fibMMm2, fibMMm1 = fibMMm1, fibM
        fibM = fibMMm1 + fibMMm2

    offset = -1

    while fibM > 1:
        i = min(offset + fibMMm2, n - 1)

        if arr[i] < target:
            fibM = fibMMm1
            fibMMm1 = fibMMm2
            fibMMm2 = fibM - fibMMm1
            offset = i
        elif arr[i] > target:
            fibM = fibMMm2
            fibMMm1 -= fibMMm2
            fibMMm2 = fibM - fibMMm1
        else:
            return i

    if fibMMm1 and offset + 1 < n and arr[offset + 1] == target:
        return offset + 1

    return -1

arr = [1, 3, 5, 8, 13, 21, 34]
print(fibonacci_search(arr, 8)) # Output: 3
```

Output:

3

## C

```
#include <stdio.h>

int min(int a, int b) { return (a < b) ? a : b; }

int fibonacci_search(int arr[], int n, int target) {
    int fibMMm2 = 0;
    int fibMMm1 = 1;
    int fibM = fibMMm1 + fibMMm2;

    while (fibM < n) {
        fibMMm2 = fibMMm1;
        fibMMm1 = fibM;
        fibM = fibMMm1 + fibMMm2;
    }

    int offset = -1;

    while (fibM > 1) {
        int i = min(offset + fibMMm2, n - 1);

        if (arr[i] < target) {
            fibM = fibMMm1;
            fibMMm1 = fibMMm2;
            fibMMm2 = fibM - fibMMm1;
            offset = i;
        } else if (arr[i] > target) {
            fibM = fibMMm2;
            fibMMm1 -= fibMMm2;
            fibMMm2 = fibM - fibMMm1;
        } else {
            return i;
        }
    }

    if (fibMMm1 && offset + 1 < n && arr[offset + 1] == target)
        return offset + 1;

    return -1;
}

int main(void) {
```

```

int arr[] = {1, 3, 5, 8, 13, 21, 34};
int n = sizeof(arr) / sizeof(arr[0]);
int idx = fibonacci_search(arr, n, 8);
printf("Found at index: %d\n", idx);
}

```

Output:

Found at index: 3

### Why It Matters

- Uses only addition and subtraction, no division
- Great for sequential access memory
- Matches binary search performance ( $O(\log n)$ )
- Offers better locality on some hardware

It's the search strategy built from nature's own numbers.

### A Gentle Proof (Why It Works)

Each iteration reduces the search space by a Fibonacci ratio:

$$F_k = F_{k-1} + F_{k-2}$$

Hence, search depth Fibonacci index  $k \sim \log_\phi n$ ,  
where  $\phi$  is the golden ratio ( $\approx 1.618$ ).

So total time:

$$T(n) = O(\log n)$$

### Try It Yourself

1. Search 8 in [1,3,5,8,13,21,34]
2. Search 21
3. Search 2 (not found)
4. Trace Fibonacci sequence steps
5. Compare probe indices with binary search
6. Try with  $n=10 \rightarrow$  Fibonacci 13 10
7. Implement recursive version

8. Visualize probe intervals
9. Replace Fibonacci with powers of 2 (binary search)
10. Experiment with non-uniform arrays

## Test Cases

| Input              | Target | Output | Notes         |
|--------------------|--------|--------|---------------|
| [1,3,5,8,13,21,34] | 8      | 3      | Found         |
| [1,3,5,8,13,21,34] | 2      | -1     | Not found     |
| [1,3,5,8,13,21,34] | 34     | 6      | Last element  |
| [1,3,5,8,13,21,34] | 1      | 0      | First element |

## Complexity

| Aspect       | Value        |
|--------------|--------------|
| Time         | $O(\log n)$  |
| Space        | $O(1)$       |
| Stable       | Yes          |
| Adaptive     | No           |
| Prerequisite | Sorted array |

Fibonacci Search is searching with nature's rhythm, each step shaped by the golden ratio, balancing reach and precision.

## 160 Uniform Binary Search

Uniform Binary Search is an optimized form of Binary Search where the probe positions are precomputed. Instead of recalculating the midpoint at each step, it uses a lookup table of offsets to determine where to go next, making it faster in tight loops or hardware-limited systems.

It's all about removing repetitive midpoint arithmetic and branching for consistent, uniform steps.

## What Problem Are We Solving??

Standard binary search repeatedly computes:

$$mid = low + \frac{high - low}{2}$$

This is cheap on modern CPUs but expensive on:

- Early hardware
- Embedded systems
- Tight loops where division or shifting is costly

Uniform Binary Search (UBS) replaces these computations with precomputed offsets for each step, giving predictable, uniform jumps.

## Example

Search 25 in [5, 10, 15, 20, 25, 30, 35, 40]

| Step | Offset | Index | Value | Compare | Action     |
|------|--------|-------|-------|---------|------------|
| 1    | 3      | 3     | 20    | 20 < 25 | move right |
| 2    | 1      | 5     | 30    | 30 > 25 | move left  |
| 3    | 0      | 4     | 25    | 25 = 25 | found      |

Instead of recalculating midpoints, UBS uses offset table [3, 1, 0].

## How Does It Work (Plain Language)?

It's binary search with a preplanned route. At each level:

- You jump by a fixed offset (from a table)
- Compare
- Move left or right, shrinking your window uniformly

No divisions, no mid calculations, just jump and compare.

## Step-by-Step Process

| Step | Description                                 |
|------|---------------------------------------------|
| 1    | Precompute offset table based on array size |
| 2    | Start at offset[0] from beginning           |
| 3    | Compare element with target                 |
| 4    | Move left/right using next offset           |
| 5    | Stop when offset = 0 or found               |

### Tiny Code (Easy Versions)

#### Python

```
def uniform_binary_search(arr, target):
    n = len(arr)
    # Precompute offsets (powers of 2 less than n)
    offsets = []
    k = 1
    while k < n:
        offsets.append(k)
        k *= 2
    offsets.reverse()

    low = 0
    idx = offsets[0]

    for offset in offsets:
        mid = low + offset if low + offset < n else n - 1
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        # else move left (implicitly handled next iteration)

    # Final check
    if low < n and arr[low] == target:
        return low
    return -1

arr = [5, 10, 15, 20, 25, 30, 35, 40]
print(uniform_binary_search(arr, 25)) # Output: 4
```

Output:

**C**

```
#include <stdio.h>

int uniform_binary_search(int arr[], int n, int target) {
    int k = 1;
    while (k < n) k <<= 1;
    k >>= 1;

    int low = 0;
    while (k > 0) {
        int mid = low + k - 1;
        if (mid >= n) mid = n - 1;

        if (arr[mid] == target)
            return mid;
        else if (arr[mid] < target)
            low = mid + 1;

        k >>= 1; // next smaller offset
    }

    return (low < n && arr[low] == target) ? low : -1;
}

int main(void) {
    int arr[] = {5, 10, 15, 20, 25, 30, 35, 40};
    int n = sizeof(arr) / sizeof(arr[0]);
    int idx = uniform_binary_search(arr, n, 25);
    printf("Found at index: %d\n", idx);
}
```

Output:

Found at index: 4

### Why It Matters

- Avoids recomputing midpoints

- Ideal for hardware, firmware, microcontrollers
- Ensures consistent runtime path
- Fewer instructions → faster in tight loops

Uniformity = predictability = performance.

### A Gentle Proof (Why It Works)

Precomputed offsets correspond to halving the search space. At each step, offset = floor(`remaining_size / 2`). After  $\log(n)$  steps, we narrow down to a single element.

Total steps =  $\log n$  Each step = constant cost (no recomputation)

So:

$$T(n) = O(\log n)$$

### Try It Yourself

1. Search 25 in [5,10,15,20,25,30,35,40]
2. Search 35
3. Search 6 → not found
4. Precompute offset table for n=8
5. Compare steps with binary search
6. Implement for n=16
7. Use for microcontroller table lookup
8. Visualize jumps on paper
9. Measure comparisons
10. Implement recursive version

### Test Cases

| Input                    | Target | Output | Notes         |
|--------------------------|--------|--------|---------------|
| [5,10,15,20,25,30,35,40] | 25     | 4      | Found         |
| [5,10,15,20,25,30,35,40] | 5      | 0      | First element |
| [5,10,15,20,25,30,35,40] | 50     | -1     | Not found     |
| [5,10,15,20,25,30,35,40] | 40     | 7      | Last element  |

### Complexity

| Aspect       | Value        |
|--------------|--------------|
| Time         | $O(\log n)$  |
| Space        | $O(1)$       |
| Stable       | Yes          |
| Adaptive     | No           |
| Prerequisite | Sorted array |

Uniform Binary Search is binary search with a map, no guesswork, no recalculations, just smooth, evenly spaced jumps to the answer.

## Section 17. Interpolation and exponential search

### 161 Interpolation Search

Interpolation Search is a search algorithm for sorted arrays with uniformly distributed values. Unlike Binary Search, which always probes the middle, Interpolation Search estimates the position of the target using value interpolation, like finding where a number lies on a number line.

If Binary Search is “divide by index,” Interpolation Search is “divide by value.”

#### What Problem Are We Solving??

Binary Search assumes no relationship between index and value. But if the array values are uniformly spaced, we can do better by guessing where the target should be, not just the middle.

It's ideal for:

- Uniformly distributed sorted data
- Numeric keys (IDs, prices, timestamps)
- Large arrays where value-based positioning matters

#### Example

Find 70 in [10, 20, 30, 40, 50, 60, 70, 80, 90]

Estimate position:

$$pos = low + \frac{(target - arr[low]) \times (high - low)}{arr[high] - arr[low]}$$

| Step | Low | High | Pos | Value | Compare   | Action    |
|------|-----|------|-----|-------|-----------|-----------|
| 1    | 0   | 8    | 7   | 80    | $80 > 70$ | move left |
| 2    | 0   | 6    | 6   | 70    | $70 = 70$ | found     |

## How Does It Work (Plain Language)?

Imagine the array as a number line. If your target is closer to the high end, start searching closer to the right. You *interpolate*, estimate the target's index based on its value proportion between min and max.

So rather than halving blindly, you jump to the likely spot.

## Step-by-Step Process

| Step | Description                                                            |
|------|------------------------------------------------------------------------|
| 1    | Initialize <code>low = 0, high = n - 1</code>                          |
| 2    | While <code>low &lt;= high</code> and <code>target</code> is in range: |
| 3    | Estimate position using interpolation formula                          |
| 4    | Compare <code>arr[pos]</code> with <code>target</code>                 |
| 5    | If equal → found                                                       |
| 6    | If smaller → move <code>low = pos + 1</code>                           |
| 7    | If larger → move <code>high = pos - 1</code>                           |
|      | Repeat until found or <code>low &gt; high</code>                       |

## Tiny Code (Easy Versions)

### Python

```
def interpolation_search(arr, target):
    low, high = 0, len(arr) - 1

    while low <= high and arr[low] <= target <= arr[high]:
        if arr[high] == arr[low]:
            if arr[low] == target:
                return low
        pos = low + ((target - arr[low]) * (high - low)) // (arr[high] - arr[low])
        if arr[pos] == target:
            return pos
        elif arr[pos] < target:
            low = pos + 1
        else:
            high = pos - 1
```

```

        return low
    break

pos = low + (target - arr[low]) * (high - low) // (arr[high] - arr[low])

if arr[pos] == target:
    return pos
elif arr[pos] < target:
    low = pos + 1
else:
    high = pos - 1

return -1

arr = [10, 20, 30, 40, 50, 60, 70, 80, 90]
print(interpolation_search(arr, 70)) # Output: 6

```

Output:

6

## C

```

#include <stdio.h>

int interpolation_search(int arr[], int n, int target) {
    int low = 0, high = n - 1;

    while (low <= high && target >= arr[low] && target <= arr[high]) {
        if (arr[high] == arr[low]) {
            if (arr[low] == target) return low;
            else break;
        }

        int pos = low + (double)(high - low) * (target - arr[low]) / (arr[high] - arr[low]);

        if (arr[pos] == target)
            return pos;
        if (arr[pos] < target)
            low = pos + 1;
        else
    }
}

```

```

        high = pos - 1;
    }

    return -1;
}

int main(void) {
    int arr[] = {10, 20, 30, 40, 50, 60, 70, 80, 90};
    int n = sizeof(arr) / sizeof(arr[0]);
    int idx = interpolation_search(arr, n, 70);
    printf("Found at index: %d\n", idx);
}

```

Output:

Found at index: 6

### Why It Matters

- Faster than Binary Search for uniformly distributed data
- Ideal for dense key spaces (like hash slots, ID ranges)
- Can achieve  $O(\log \log n)$  average time
- Preserves sorted order search logic

It's the value-aware cousin of Binary Search.

### A Gentle Proof (Why It Works)

If data is uniformly distributed, each probe halves value range logarithmically. Expected probes:

$$T(n) = O(\log \log n)$$

Worst case (non-uniform):

$$T(n) = O(n)$$

So it outperforms binary search only under uniform value distribution.

## Try It Yourself

1. Search 70 in [10,20,30,40,50,60,70,80,90]
2. Search 25 (not found)
3. Try `arr = [2,4,8,16,32,64]`, notice uneven distribution
4. Plot estimated positions
5. Compare steps with Binary Search
6. Use floating-point vs integer division
7. Implement recursive version
8. Search 10 (first element)
9. Search 90 (last element)
10. Measure iterations on uniform vs non-uniform data

## Test Cases

| Input                        | Target | Output | Notes     |
|------------------------------|--------|--------|-----------|
| [10,20,30,40,50,60,70,80,90] | 70     | 6      | Found     |
| [10,20,30,40,50,60,70,80,90] | 25     | -1     | Not found |
| [10,20,30,40,50,60,70,80,90] | 10     | 0      | First     |
| [10,20,30,40,50,60,70,80,90] | 90     | 8      | Last      |

## Complexity

| Aspect       | Value                  |
|--------------|------------------------|
| Time (avg)   | $O(\log \log n)$       |
| Time (worst) | $O(n)$                 |
| Space        | $O(1)$                 |
| Stable       | Yes                    |
| Adaptive     | Yes                    |
| Prerequisite | Sorted & uniform array |

Interpolation Search is like a treasure map that scales by value, it doesn't just guess the middle, it guesses where the gold really lies.

## 162 Recursive Interpolation Search

Recursive Interpolation Search is the recursive variant of the classic Interpolation Search. Instead of looping, it calls itself on smaller subranges, estimating the likely position using the same value-based interpolation formula.

It's a natural way to express the algorithm for learners who think recursively, same logic, cleaner flow.

### What Problem Are We Solving??

We're taking the iterative interpolation search and expressing it recursively, to highlight the divide-and-conquer nature. The recursive form is often more intuitive and mathematically aligned with its interpolation logic.

You'll use this when:

- Teaching or visualizing recursive logic
- Writing clean, declarative search code
- Practicing recursion-to-iteration transitions

### Example

Find 50 in [10, 20, 30, 40, 50, 60, 70, 80, 90]

Step 1: low = 0, high = 8 pos =  $0 + (50 - 10) \times (8 - 0) / (90 - 10) = 4$  arr[4] = 50 → found

Recursion stops immediately, one call, one success.

### How Does It Work (Plain Language)?

Instead of looping, each recursive call zooms in to the subrange where the target might be. Each call computes an estimated index `pos` proportional to the target's distance between `arr[low]` and `arr[high]`.

If value is higher → recurse right If lower → recurse left If equal → return index

| Step | Description |
|------|-------------|
|------|-------------|

### Step-by-Step Process

| Step | Description                                                                 |
|------|-----------------------------------------------------------------------------|
| 1    | Base case: if <code>low &gt; high</code> or target out of range → not found |
| 2    | Compute position estimate using interpolation formula                       |
| 3    | If <code>arr[pos] == target</code> → return <code>pos</code>                |
| 4    | If <code>arr[pos] &lt; target</code> → recurse on right half                |
| 5    | Else recurse on left half                                                   |

### Tiny Code (Easy Versions)

#### Python

```
def interpolation_search_recursive(arr, low, high, target):
    if low > high or target < arr[low] or target > arr[high]:
        return -1

    if arr[high] == arr[low]:
        return low if arr[low] == target else -1

    pos = low + (target - arr[low]) * (high - low) // (arr[high] - arr[low])

    if arr[pos] == target:
        return pos
    elif arr[pos] < target:
        return interpolation_search_recursive(arr, pos + 1, high, target)
    else:
        return interpolation_search_recursive(arr, low, pos - 1, target)

arr = [10, 20, 30, 40, 50, 60, 70, 80, 90]
print(interpolation_search_recursive(arr, 0, len(arr) - 1, 50)) # Output: 4
```

Output:

4

## C

```
#include <stdio.h>

int interpolation_search_recursive(int arr[], int low, int high, int target) {
    if (low > high || target < arr[low] || target > arr[high])
        return -1;

    if (arr[high] == arr[low])
        return (arr[low] == target) ? low : -1;

    int pos = low + (double)(high - low) * (target - arr[low]) / (arr[high] - arr[low]);

    if (arr[pos] == target)
        return pos;
    else if (arr[pos] < target)
        return interpolation_search_recursive(arr, pos + 1, high, target);
    else
        return interpolation_search_recursive(arr, low, pos - 1, target);
}

int main(void) {
    int arr[] = {10, 20, 30, 40, 50, 60, 70, 80, 90};
    int n = sizeof(arr) / sizeof(arr[0]);
    int idx = interpolation_search_recursive(arr, 0, n - 1, 50);
    printf("Found at index: %d\n", idx);
}
```

Output:

```
Found at index: 4
```

## Why It Matters

- Shows the recursive nature of interpolation-based estimation
- Clean, mathematical form for teaching and analysis
- Demonstrates recursion depth proportional to search cost
- Easier to reason about with divide-by-value intuition

Think of it as binary search's artistic cousin, elegant and value-aware.

## A Gentle Proof (Why It Works)

In uniform distributions, each recursive step shrinks the search space multiplicatively, not just by half.

If data is uniform:

$$T(n) = T(n/f) + O(1) \Rightarrow T(n) = O(\log \log n)$$

If not uniform:

$$T(n) = O(n)$$

Recursion depth = number of interpolation refinements  $\log \log n$ .

## Try It Yourself

1. Search 70 in [10,20,30,40,50,60,70,80,90]
2. Search 25 (not found)
3. Trace recursive calls manually
4. Add print statements to watch low/high shrink
5. Compare depth with binary search
6. Try [2,4,8,16,32,64] (non-uniform)
7. Add guard for `arr[low] == arr[high]`
8. Implement tail recursion optimization
9. Test base cases (first, last, single element)
10. Time comparison: iterative vs recursive

## Test Cases

| Input                        | Target | Output | Notes     |
|------------------------------|--------|--------|-----------|
| [10,20,30,40,50,60,70,80,90] | 50     | 4      | Found     |
| [10,20,30,40,50,60,70,80,90] | 25     | -1     | Not found |
| [10,20,30,40,50,60,70,80,90] | 10     | 0      | First     |
| [10,20,30,40,50,60,70,80,90] | 90     | 8      | Last      |

| Aspect | Value |
|--------|-------|
|--------|-------|

## Complexity

| Aspect       | Value                        |
|--------------|------------------------------|
| Time (avg)   | $O(\log \log n)$             |
| Time (worst) | $O(n)$                       |
| Space        | $O(\log \log n)$ (recursion) |
| Stable       | Yes                          |
| Adaptive     | Yes                          |
| Prerequisite | Sorted & uniform array       |

Recursive Interpolation Search is like zooming in with value-based intuition, each step a mathematical guess, each call a sharper focus.

## 163 Exponential Search

Exponential Search combines range expansion with binary search. It's perfect when the array size is unknown or infinite, it first finds the range that might contain the target by doubling the index, then uses binary search within that range.

It's the "zoom out, then zoom in" strategy for searching sorted data.

### What Problem Are We Solving??

If you don't know the length of your sorted array, you can't directly apply binary search. You need to first bound the search space, so you know where to look.

Exponential Search does exactly that:

- It doubles the index (1, 2, 4, 8, 16...) until it overshoots.
- Then it performs binary search in that bracket.

Use it for:

- Infinite or dynamically sized sorted arrays
- Streams
- Unknown-length files or data structures

## Example

Find 19 in [2, 4, 8, 16, 19, 23, 42, 64, 128]

| Step | Range | Value | Compare | Action       |
|------|-------|-------|---------|--------------|
| 1    | 1     | 4     | 4 < 19  | double index |
| 2    | 2     | 8     | 8 < 19  | double index |
| 3    | 4     | 19    | 19 = 19 | found        |

If it had overshot (say, target 23), we'd binary search between 4 and 8.

## How Does It Work (Plain Language)?

Think of searching an endless bookshelf. You take steps of size 1, 2, 4, 8... until you pass the book number you want. Now you know which shelf section it's on, then you check precisely using binary search.

Fast to expand, precise to finish.

## Step-by-Step Process

| Step | Description                                                                 |
|------|-----------------------------------------------------------------------------|
| 1    | Start at index 1                                                            |
| 2    | While $i < n$ and $\text{arr}[i] < \text{target}$ , double $i$              |
| 3    | When overshoot $\rightarrow$ binary search between $i/2$ and $\min(i, n-1)$ |
| 4    | Return index if found, else -1                                              |

## Tiny Code (Easy Versions)

### Python

```
def binary_search(arr, low, high, target):
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
```

```

        else:
            high = mid - 1
    return -1

def exponential_search(arr, target):
    n = len(arr)
    if n == 0:
        return -1
    if arr[0] == target:
        return 0
    i = 1
    while i < n and arr[i] <= target:
        i *= 2
    return binary_search(arr, i // 2, min(i, n - 1), target)

arr = [2, 4, 8, 16, 19, 23, 42, 64, 128]
print(exponential_search(arr, 19))  # Output: 4

```

Output:

4

C

```

#include <stdio.h>

int binary_search(int arr[], int low, int high, int target) {
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == target)
            return mid;
        else if (arr[mid] < target)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
}

int exponential_search(int arr[], int n, int target) {
    if (n == 0) return -1;

```

```

if (arr[0] == target) return 0;

int i = 1;
while (i < n && arr[i] <= target)
    i *= 2;

int low = i / 2;
int high = (i < n) ? i : n - 1;
return binary_search(arr, low, high, target);
}

int main(void) {
    int arr[] = {2, 4, 8, 16, 19, 23, 42, 64, 128};
    int n = sizeof(arr) / sizeof(arr[0]);
    int idx = exponential_search(arr, n, 19);
    printf("Found at index: %d\n", idx);
}

```

Output:

Found at index: 4

## Why It Matters

- Handles unknown size arrays
- Logarithmic search after expansion
- Fewer comparisons for small targets
- Common in unbounded search, streams, linked memory

It's the search that grows as needed, like zooming your scope until the target appears.

## A Gentle Proof (Why It Works)

Each iteration reduces the search space by a Fibonacci ratio:

$$F_k = F_{k-1} + F_{k-2}$$

Hence, search depth Fibonacci index  $k \sim \log_\phi n$ , where  $\phi$  is the golden ratio ( $\approx 1.618$ ).

So total time:

$$T(n) = O(\log n)$$

## Try It Yourself

1. Search 19 in [2,4,8,16,19,23,42,64,128]
2. Search 42
3. Search 1 (not found)
4. Trace expansion: 1, 2, 4, 8...
5. Compare expansion steps vs binary search calls
6. Try on huge array, small target
7. Try on dynamic-size list (simulate infinite)
8. Implement recursive version
9. Measure expansions vs comparisons
10. Combine with galloping search in TimSort

## Test Cases

| Input                      | Target | Output | Notes        |
|----------------------------|--------|--------|--------------|
| [2,4,8,16,19,23,42,64,128] | 19     | 4      | Found        |
| [2,4,8,16,19,23,42,64,128] | 23     | 5      | Found        |
| [2,4,8,16,19,23,42,64,128] | 1      | -1     | Not found    |
| [2,4,8,16,19,23,42,64,128] | 128    | 8      | Last element |

## Complexity

| Aspect       | Value        |
|--------------|--------------|
| Time         | $O(\log p)$  |
| Space        | $O(1)$       |
| Stable       | Yes          |
| Adaptive     | Yes          |
| Prerequisite | Sorted array |

Exponential Search is your wayfinder, it reaches out in powers of two, then narrows in precisely where the target hides.

## 164 Doubling Search

Doubling Search (also called Unbounded Search) is the general pattern behind Exponential Search. It's used when the data size or range is unknown, and you need to quickly discover a

search interval that contains the target before performing a precise search (like binary search) inside that interval.

Think of it as “*search by doubling until you find your bracket.*”

### **What Problem Are We Solving??**

In many real-world scenarios, you don’t know the array’s length or the bounds of your search domain. You can’t jump straight into binary search, you need an upper bound first.

Doubling Search gives you a strategy to find that bound quickly, in logarithmic time, by doubling your index or step size until you overshoot the target.

Perfect for:

- Infinite or streaming sequences
- Functions or implicit arrays (not stored in memory)
- Search domains defined by value, not length

### **Example**

Find 23 in [2, 4, 8, 16, 19, 23, 42, 64, 128]

| Step | Range | Value | Compare  | Action   |
|------|-------|-------|----------|----------|
| 1    | i = 1 | 4     | 4 < 23   | double i |
| 2    | i = 2 | 8     | 8 < 23   | double i |
| 3    | i = 4 | 16    | 16 < 23  | double i |
| 4    | i = 8 | 128   | 128 > 23 | stop     |

Range found: [4, 8] Now run binary search within [16, 19, 23, 42, 64, 128] Found at index 5

### **How Does It Work (Plain Language)?**

Start small, test the first few steps. Every time you don’t find your target and the value is still less, double your index (1, 2, 4, 8, 16...). Once you pass your target, you’ve found your interval, then you search precisely.

It’s like walking in the dark and doubling your stride each time until you see the light, then stepping back carefully.

## Step-by-Step Process

| Step | Description                                                        |
|------|--------------------------------------------------------------------|
| 1    | Start at index 1                                                   |
| 2    | While <code>arr[i] &lt; target</code> , set <code>i = 2 * i</code> |
| 3    | When overshoot, define range <code>[i/2, min(i, n-1)]</code>       |
| 4    | Apply binary search within range                                   |
| 5    | Return index or -1 if not found                                    |

## Tiny Code (Easy Versions)

### Python

```
def doubling_search(arr, target):
    n = len(arr)
    if n == 0:
        return -1
    if arr[0] == target:
        return 0

    i = 1
    while i < n and arr[i] < target:
        i *= 2

    low = i // 2
    high = min(i, n - 1)

    # binary search
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1

    return -1

arr = [2, 4, 8, 16, 19, 23, 42, 64, 128]
print(doubling_search(arr, 23))  # Output: 5
```

Output:

5

C

```
#include <stdio.h>

int binary_search(int arr[], int low, int high, int target) {
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == target)
            return mid;
        else if (arr[mid] < target)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
}

int doubling_search(int arr[], int n, int target) {
    if (n == 0) return -1;
    if (arr[0] == target) return 0;

    int i = 1;
    while (i < n && arr[i] < target)
        i *= 2;

    int low = i / 2;
    int high = (i < n) ? i : n - 1;

    return binary_search(arr, low, high, target);
}

int main(void) {
    int arr[] = {2, 4, 8, 16, 19, 23, 42, 64, 128};
    int n = sizeof(arr) / sizeof(arr[0]);
    int idx = doubling_search(arr, n, 23);
    printf("Found at index: %d\n", idx);
}
```

Output:

```
Found at index: 5
```

## Why It Matters

- Allows binary search when size is unknown
- Only  $O(\log p)$  probes, where  $p$  is target index
- Natural strategy for streaming, infinite, or lazy structures
- Used in exponential, galloping, and tim-sort merges

It's the blueprint for all "expand then search" algorithms.

## A Gentle Proof (Why It Works)

Each doubling step multiplies range by 2, so number of expansions  $\log(p)$ . Then binary search inside a range of size  $p$ .

Total time:

$$T(n) = O(\log p)$$

Still logarithmic in target position, not array size, efficient even for unbounded data.

## Try It Yourself

1. Search 23 in [2, 4, 8, 16, 19, 23, 42, 64, 128]
2. Search 42 (more jumps)
3. Search 1 (before first element)
4. Search 128 (last element)
5. Try doubling factor 3 instead of 2
6. Compare expansion steps with exponential search
7. Implement recursive version
8. Visualize on a number line
9. Simulate unknown-length array with a safe check
10. Measure steps for small vs large targets

## Test Cases

| Input                      | Target | Output | Notes        |
|----------------------------|--------|--------|--------------|
| [2,4,8,16,19,23,42,64,128] | 23     | 5      | Found        |
| [2,4,8,16,19,23,42,64,128] | 4      | 1      | Early        |
| [2,4,8,16,19,23,42,64,128] | 1      | -1     | Not found    |
| [2,4,8,16,19,23,42,64,128] | 128    | 8      | Last element |

## Complexity

| Aspect       | Value        |
|--------------|--------------|
| Time         | $O(\log p)$  |
| Space        | $O(1)$       |
| Stable       | Yes          |
| Adaptive     | Yes          |
| Prerequisite | Sorted array |

Doubling Search is how you explore the unknown, double your reach, find your bounds, then pinpoint your goal.

## 165 Galloping Search

Galloping Search (also called Exponential Gallop or Search by Doubling) is a hybrid search technique that quickly finds a range by *galloping forward exponentially*, then switches to binary search within that range.

It's heavily used inside TimSort's merge step, where it helps merge sorted runs efficiently by skipping over large stretches that don't need detailed comparison.

### What Problem Are We Solving??

When merging two sorted arrays (or searching in a sorted list), repeatedly comparing one by one is wasteful if elements differ by a large margin. Galloping Search “jumps ahead” quickly to find the region of interest, then finishes with a binary search to locate the exact boundary.

Used in:

- TimSort merges
- Run merging in hybrid sorting
- Searching in sorted blocks
- Optimizing comparison-heavy algorithms

## Example

Find 25 in [5, 10, 15, 20, 25, 30, 35, 40, 45]

| Step | Index | Value | Compare | Action                |
|------|-------|-------|---------|-----------------------|
| 1    | 1     | 10    | 10 < 25 | gallop ( $\times 2$ ) |
| 2    | 2     | 15    | 15 < 25 | gallop                |
| 3    | 4     | 25    | 25 = 25 | found                 |

If we overshoot (say, search for 22), we'd gallop past it (index 4), then perform binary search between the previous bound and current one.

## How Does It Work (Plain Language)?

You start by leaping, checking 1, 2, 4, 8 steps away, until you pass the target or reach the end. Once you overshoot, you gallop back (binary search) in the last valid interval.

It's like sprinting ahead to find the neighborhood, then walking house-to-house once you know the block.

## Step-by-Step Process

| Step | Description                                                                                    |
|------|------------------------------------------------------------------------------------------------|
| 1    | Start at <code>start</code> index                                                              |
| 2    | Gallop forward by powers of 2 (1, 2, 4, 8, ...) until <code>arr[start + k] &gt;= target</code> |
| 3    | Define range <code>[start + k/2, min(start + k, n-1)]</code>                                   |
| 4    | Apply binary search in that range                                                              |
| 5    | Return index if found                                                                          |

## Tiny Code (Easy Versions)

### Python

```
def binary_search(arr, low, high, target):
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
```

```

        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1

def galloping_search(arr, start, target):
    n = len(arr)
    if start >= n:
        return -1
    if arr[start] == target:
        return start

    k = 1
    while start + k < n and arr[start + k] < target:
        k *= 2

    low = start + k // 2
    high = min(start + k, n - 1)
    return binary_search(arr, low, high, target)

arr = [5, 10, 15, 20, 25, 30, 35, 40, 45]
print(galloping_search(arr, 0, 25)) # Output: 4

```

Output:

4

C

```

#include <stdio.h>

int binary_search(int arr[], int low, int high, int target) {
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == target)
            return mid;
        else if (arr[mid] < target)
            low = mid + 1;
        else
            high = mid - 1;
    }
}

```

```

    }
    return -1;
}

int galloping_search(int arr[], int n, int start, int target) {
    if (start >= n) return -1;
    if (arr[start] == target) return start;

    int k = 1;
    while (start + k < n && arr[start + k] < target)
        k *= 2;

    int low = start + k / 2;
    int high = (start + k < n) ? start + k : n - 1;

    return binary_search(arr, low, high, target);
}

int main(void) {
    int arr[] = {5, 10, 15, 20, 25, 30, 35, 40, 45};
    int n = sizeof(arr) / sizeof(arr[0]);
    int idx = galloping_search(arr, n, 0, 25);
    printf("Found at index: %d\n", idx);
}

```

Output:

Found at index: 4

### Why It Matters

- Accelerates merging in TimSort
- Minimizes comparisons when merging large sorted runs
- Great for adaptive sorting and range scans
- Balances speed (gallop) and precision (binary)

It's a dynamic balance, gallop when far, tiptoe when close.

## A Gentle Proof (Why It Works)

Galloping phase:

Takes  $O(\log p)$  steps to reach the target vicinity (where  $p$  is the distance from the start).

Binary phase:

Another  $O(\log p)$  for local search.

Total:

$$T(n) = O(\log p)$$

Faster than linear merging when runs differ greatly in length.

## Try It Yourself

1. Search 25 in [5,10,15,20,25,30,35,40,45]
2. Search 40 (larger gallop)
3. Search 3 (before first)
4. Try `start=2`
5. Print gallop steps
6. Compare with pure binary search
7. Implement recursive gallop
8. Use for merging two sorted arrays
9. Count comparisons per search
10. Test on long list with early/late targets

## Test Cases

| Input                       | Target | Output | Notes         |
|-----------------------------|--------|--------|---------------|
| [5,10,15,20,25,30,35,40,45] | 25     | 4      | Found         |
| [5,10,15,20,25,30,35,40,45] | 40     | 7      | Larger gallop |
| [5,10,15,20,25,30,35,40,45] | 3      | -1     | Not found     |
| [5,10,15,20,25,30,35,40,45] | 5      | 0      | First element |

## Complexity

| Aspect | Value       |
|--------|-------------|
| Time   | $O(\log p)$ |
| Space  | $O(1)$      |

| Aspect       | Value        |
|--------------|--------------|
| Stable       | Yes          |
| Adaptive     | Yes          |
| Prerequisite | Sorted array |

Galloping Search is how TimSort runs ahead with confidence, sprint through big gaps, slow down only when precision counts.

## 166 Unbounded Binary Search

Unbounded Binary Search is a technique for finding a value in a sorted but unbounded (or infinite) sequence. You don't know where the data ends, or even how large it is, but you know it's sorted. So first, you find a search boundary, then perform a binary search within that discovered range.

It's a direct application of doubling search followed by binary search, especially suited for monotonic functions or streams.

### What Problem Are We Solving??

If you're working with data that:

- Has no fixed size,
- Comes as a stream,
- Or is represented as a monotonic function (like  $f(x)$  increasing with  $x$ ),

then you can't apply binary search immediately, because you don't know `high`.

So you first need to find bounds [low, high] that contain the target, by expanding exponentially, and only then use binary search.

### Example

Suppose you're searching for 20 in a monotonic function:

$$f(x) = 2x + 2$$

You want the smallest  $x$  such that  $f(x) = 20$ .

Step 1: Find bounds. Check  $x = 1 \rightarrow f(1) = 4$  Check  $x = 2 \rightarrow f(2) = 6$  Check  $x = 4 \rightarrow f(4) = 10$  Check  $x = 8 \rightarrow f(8) = 18$  Check  $x = 16 \rightarrow f(16) = 34$  (overshoot!)

Now you know target lies between  $x = 8$  and  $x = 16$ .

Step 2: Perform binary search in  $[8, 16]$ . Found  $f(9) = 20$

## How Does It Work (Plain Language)?

You start with a small step and double your reach each time until you go beyond the target. Once you've overshot, you know the search interval, and you can binary search inside it for precision.

It's the go-to strategy when the array (or domain) has no clear end.

## Step-by-Step Process

| Step | Description                                                                                  |
|------|----------------------------------------------------------------------------------------------|
| 1    | Initialize $\text{low} = 0$ , $\text{high} = 1$                                              |
| 2    | While $f(\text{high}) < \text{target}$ , set $\text{low} = \text{high}$ , $\text{high} *= 2$ |
| 3    | Once $f(\text{high}) \geq \text{target}$ , binary search between $[\text{low}, \text{high}]$ |
| 4    | Return index (or value)                                                                      |

## Tiny Code (Easy Versions)

### Python

```
def unbounded_binary_search(f, target):
    low, high = 0, 1
    # Step 1: find bounds
    while f(high) < target:
        low = high
        high *= 2

    # Step 2: binary search in [low, high]
    while low <= high:
        mid = (low + high) // 2
        val = f(mid)
        if val == target:
```

```

        return mid
    elif val < target:
        low = mid + 1
    else:
        high = mid - 1
    return -1

# Example: f(x) = 2x + 2
f = lambda x: 2 * x + 2
print(unbounded_binary_search(f, 20)) # Output: 9

```

Output:

9

## C

```

#include <stdio.h>

int f(int x) {
    return 2 * x + 2;
}

int binary_search(int (*f)(int), int low, int high, int target) {
    while (low <= high) {
        int mid = low + (high - low) / 2;
        int val = f(mid);
        if (val == target)
            return mid;
        else if (val < target)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
}

int unbounded_binary_search(int (*f)(int), int target) {
    int low = 0, high = 1;
    while (f(high) < target) {
        low = high;

```

```

        high *= 2;
    }
    return binary_search(f, low, high, target);
}

int main(void) {
    int idx = unbounded_binary_search(f, 20);
    printf("Found at x = %d\n", idx);
}

```

Output:

Found at x = 9

### Why It Matters

- Handles infinite sequences or unbounded domains
- Perfect for monotonic functions (e.g.  $f(x)$  increasing)
- Key in searching without array length
- Used in root finding, binary lifting, streaming

It's the “expand, then refine” pattern, the explorer's search.

### A Gentle Proof (Why It Works)

Expanding step: doubles index each time  $\rightarrow O(\log p)$ , where  $p$  is the position of the target.  
Binary search step: searches within a range of size  $p \rightarrow$  another  $O(\log p)$ .

Total complexity:

$$T(n) = O(\log p)$$

Independent of total domain size.

### Try It Yourself

1.  $f(x) = 2x + 2$ , target = 20  $\rightarrow$  9
2.  $f(x) = x^2$ , target = 64  $\rightarrow$  8
3. Search for non-existing value (e.g. 7 in even series)
4. Modify  $f(x)$  to be exponential
5. Print bounds before binary search

6. Try with negative domain (guard with if)
7. Apply to sorted infinite list (simulate with f)
8. Use float function and tolerance check
9. Compare with linear probing
10. Use to find smallest x where  $f(x) \geq \text{target}$

## Test Cases

| Function | Target | Output | Notes      |
|----------|--------|--------|------------|
| $2x+2$   | 20     | 9      | $f(9)=20$  |
| $x^2$    | 64     | 8      | $f(8)=64$  |
| $2x+2$   | 7      | -1     | Not found  |
| $3x+1$   | 31     | 10     | $f(10)=31$ |

## Complexity

| Aspect       | Value                             |
|--------------|-----------------------------------|
| Time         | $O(\log p)$                       |
| Space        | $O(1)$                            |
| Stable       | Yes                               |
| Adaptive     | Yes                               |
| Prerequisite | Monotonic function or sorted data |

Unbounded Binary Search is how you search the infinite, double your bounds, then zoom in on the truth.

## 167 Root-Finding Bisection

Root-Finding Bisection is a numerical search algorithm for locating the point where a continuous function crosses zero. It repeatedly halves an interval where the sign of the function changes, guaranteeing that a root exists within that range.

It's the simplest, most reliable method for solving equations like  $f(x) = 0$  when you only know that a solution exists somewhere between  $a$  and  $b$ .

## What Problem Are We Solving??

When you have a function  $f(x)$  and want to solve  $f(x) = 0$ , but you can't solve it algebraically, you can use the **Bisection Method**.

If  $f(a)$  and  $f(b)$  have opposite signs, then by the Intermediate Value Theorem, there is at least one root between  $a$  and  $b$ .

Example:

Find the root of  $f(x) = x^3 - x - 2$  in  $[1, 2]$ .

- $f(1) = -2$
  - $f(2) = 4$
- Signs differ  $\rightarrow$  there's a root in  $[1, 2]$ .

## How Does It Work (Plain Language)?

You start with an interval  $[a, b]$  where  $f(a)$  and  $f(b)$  have opposite signs.  
Then, at each step:

1. Find midpoint  $m = \frac{a+b}{2}$
2. Evaluate  $f(m)$
3. Replace either  $a$  or  $b$  so that signs at the ends still differ
4. Repeat until the interval is small enough

You're squeezing the interval tighter and tighter until it hugs the root.

## Step-by-Step Process

| Step | Description                                                    |
|------|----------------------------------------------------------------|
| 1    | Choose $a, b$ with $f(a)$ and $f(b)$ of opposite signs         |
| 2    | Compute midpoint $m = (a + b) / 2$                             |
| 3    | Evaluate $f(m)$                                                |
| 4    | If $f(m)$ has same sign as $f(a)$ , set $a = m$ , else $b = m$ |
| 5    | Repeat until $ f(m) $ or $ b - a  <$ tolerance                 |
| 6    | Return $m$ as root                                             |

## Tiny Code (Easy Versions)

### Python

```
def f(x):
    return x*x*x - x - 2

def bisection(f, a, b, tol=1e-6):
    if f(a) * f(b) >= 0:
        raise ValueError("No sign change: root not guaranteed")
    while (b - a) / 2 > tol:
        m = (a + b) / 2
        if f(m) == 0:
            return m
        elif f(a) * f(m) < 0:
            b = m
        else:
            a = m
    return (a + b) / 2

root = bisection(f, 1, 2)
print("Approx root:", root)
```

Output:

```
Approx root: 1.52138
```

### C

```
#include <stdio.h>
#include <math.h>

double f(double x) {
    return x*x*x - x - 2;
}

double bisection(double (*f)(double), double a, double b, double tol) {
    if (f(a) * f(b) >= 0) {
        printf("No sign change. Root not guaranteed.\n");
        return NAN;
    }
```

```

double m;
while ((b - a) / 2 > tol) {
    m = (a + b) / 2;
    double fm = f(m);
    if (fm == 0)
        return m;
    if (f(a) * fm < 0)
        b = m;
    else
        a = m;
}
return (a + b) / 2;
}

int main(void) {
    double root = bisection(f, 1, 2, 1e-6);
    printf("Approx root: %.6f\n", root);
}

```

Output:

Approx root: 1.521380

## Why It Matters

- Guaranteed convergence if ( $f$ ) is continuous and signs differ
- Simple and robust
- Works for nonlinear equations
- Great starting point before more advanced methods (Newton, Secant)
- Used in physics, finance, engineering for precise solving

## A Gentle Proof (Why It Works)

By the Intermediate Value Theorem:

If  $f(a) \cdot f(b) < 0$ , then there exists  $c \in [a, b]$  such that  $f(c) = 0$ .

Each iteration halves the interval size, so after  $k$  steps:

$$b_k - a_k = \frac{b_0 - a_0}{2^k}$$

To achieve tolerance  $\varepsilon$ :

$$k \approx \log_2 \left( \frac{b_0 - a_0}{\varepsilon} \right)$$

Thus, convergence is linear, but guaranteed.

### Try It Yourself

1.  $f(x) = x^2 - 2$ , interval  $[1, 2] \rightarrow \sqrt{2}$
2.  $f(x) = \cos x - x$ , interval  $[0, 1]$
3.  $f(x) = x^3 - 7$ , interval  $[1, 3]$
4. Try tighter tolerances (`1e-3, 1e-9`)
5. Count how many iterations needed
6. Print each midpoint
7. Compare with Newton's method
8. Plot convergence curve
9. Modify to return both root and iterations
10. Test on function with multiple roots

### Test Cases

| Function      | Interval | Root (Approx) | Notes            |
|---------------|----------|---------------|------------------|
| $x^2 - 2$     | $[1, 2]$ | 1.4142        | $\sqrt{2}$       |
| $x^3 - x - 2$ | $[1, 2]$ | 1.5214        | Classic example  |
| $\cos x - x$  | $[0, 1]$ | 0.7391        | Fixed point root |
| $x^3 - 7$     | $[1, 3]$ | 1.913         | Cube root of 7   |

### Complexity

| Aspect      | Value                |
|-------------|----------------------|
| Time        | $O(\log((b-a)/tol))$ |
| Space       | $O(1)$               |
| Convergence | Linear               |
| Stability   | High                 |

| Aspect              | Value |
|---------------------|-------|
| Requires continuity | Yes   |

Bisection Method is your steady compass in numerical analysis, it never fails when a root is truly there.

## 168 Golden Section Search

Golden Section Search is a clever optimization algorithm for finding the minimum (or maximum) of a unimodal function on a closed interval ( $[a, b]$ )—without derivatives. It's a close cousin of binary search, but instead of splitting in half, it uses the golden ratio to minimize function evaluations.

### What Problem Are We Solving??

You want to find the  $x$  that minimizes  $f(x)$  on an interval ( $[a, b]$ ), but you can't or don't want to take derivatives (maybe  $f$  is noisy or discontinuous).

If  $f(x)$  is unimodal (has a single peak or trough), then the Golden Section Search gives you a guaranteed narrowing path to the optimum.

### Example

Find the minimum of

$$f(x) = (x - 2)^2 + 1$$

on  $([0, 5])$

Since  $(f(x))$  is quadratic, its minimum is at  $(x = 2)$ . The algorithm will zoom in around  $(2)$  by evaluating at golden-ratio points.

### How Does It Work (Plain Language)?

Imagine slicing your search interval using the golden ratio (( 1.618 )). By placing test points at those ratios, you can reuse past evaluations and eliminate one side of the interval every step.

Each iteration shrinks the search range while keeping the ratio intact — like a mathematically perfect zoom.

## Step-by-Step Process

| Step | Description                                                      | Condition | Result / Note |
|------|------------------------------------------------------------------|-----------|---------------|
| 1    | Choose initial interval $[a, b]$                                 |           |               |
| 2    | Compute golden ratio $\phi = \frac{\sqrt{5}-1}{2} \approx 0.618$ |           |               |
| 3    | Set $c = b - \phi(b - a)$ , $d = a + \phi(b - a)$                |           |               |
| 4    | Evaluate $f(c)$ and $f(d)$                                       |           |               |
| 5    | If $f(c) < f(d)$ , new interval = $[a, d]$                       |           |               |
| 6    | Else, new interval = $[c, b]$                                    |           |               |
| 7    | Repeat until $b - a <$ tolerance                                 |           |               |
| 8    | Return midpoint as best estimate                                 |           |               |

## Tiny Code (Easy Versions)

### Python

```
import math

def f(x):
    return (x - 2)**2 + 1

def golden_section_search(f, a, b, tol=1e-5):
    phi = (math.sqrt(5) - 1) / 2
    c = b - phi * (b - a)
    d = a + phi * (b - a)
    fc, fd = f(c), f(d)

    while abs(b - a) > tol:
        if fc < fd:
            b, d, fd = d, c, fc
            c = b - phi * (b - a)
            fc = f(c)
        else:
            a, c, fc = c, d, fd
            d = a + phi * (b - a)
            fd = f(d)
    return (b + a) / 2

root = golden_section_search(f, 0, 5)
print("Minimum near x =", root)
```

Output:

```
Minimum near x = 2.0000
```

C

```
#include <stdio.h>
#include <math.h>

double f(double x) {
    return (x - 2)*(x - 2) + 1;
}

double golden_section_search(double (*f)(double), double a, double b, double tol) {
    double phi = (sqrt(5.0) - 1) / 2;
    double c = b - phi * (b - a);
    double d = a + phi * (b - a);
    double fc = f(c), fd = f(d);

    while (fabs(b - a) > tol) {
        if (fc < fd) {
            b = d;
            d = c;
            fd = fc;
            c = b - phi * (b - a);
            fc = f(c);
        } else {
            a = c;
            c = d;
            fc = fd;
            d = a + phi * (b - a);
            fd = f(d);
        }
    }
    return (b + a) / 2;
}

int main(void) {
    double x = golden_section_search(f, 0, 5, 1e-5);
    printf("Minimum near x = %.5f\n", x);
}
```

Output:

```
Minimum near x = 2.00000
```

## Why It Matters

- No derivatives required
- Fewer evaluations than simple binary search
- Guaranteed convergence for unimodal functions
- Used in numerical optimization, tuning, engineering design, hyperparameter search
- The golden ratio ensures optimal reuse of computed points

## A Gentle Proof (Why It Works)

At each step, the interval length is multiplied by  $\phi \approx 0.618$ .

So after  $k$  steps:

$$L_k = (b_0 - a_0) \times \phi^k$$

To reach tolerance  $\varepsilon$ :

$$k = \frac{\log(\varepsilon/(b_0 - a_0))}{\log(\phi)}$$

Thus, convergence is linear but efficient, and each iteration needs only one new evaluation.

## Try It Yourself

1.  $f(x) = (x - 3)^2$ , interval  $[0, 6]$
2.  $f(x) = x^4 - 10x^2 + 9$ , interval  $[-5, 5]$
3.  $f(x) = |x - 1|$ , interval  $[-2, 4]$
4. Try changing tolerance to  $1e-3, 1e-9$
5. Track number of iterations
6. Plot search intervals
7. Switch to maximizing (compare signs)
8. Test non-unimodal function (observe failure)
9. Modify to return  $f(x^*)$  as well
10. Compare with ternary search

## Test Cases

| Function        | Interval  | Minimum $x$ | $f(x)$ |
|-----------------|-----------|-------------|--------|
| $(x - 2)^2 + 1$ | $[0, 5]$  | 2.0000      | 1.0000 |
| $(x - 3)^2$     | $[0, 6]$  | 3.0000      | 0.0000 |
| $x^2$           | $[-3, 2]$ | 0.0000      | 0.0000 |

## Complexity

| Aspect               | Value                |
|----------------------|----------------------|
| Time                 | $O(\log((b-a)/tol))$ |
| Space                | $O(1)$               |
| Evaluations per step | 1 new point          |
| Convergence          | Linear               |
| Requires unimodality | Yes                  |

Golden Section Search is optimization's quiet craftsman, balancing precision and simplicity with the elegance of .

## 169 Fibonacci Search (Optimum)

Fibonacci Search is a divide-and-conquer algorithm that uses the Fibonacci sequence to determine probe positions when searching for a target in a sorted array. It's similar to binary search but uses Fibonacci numbers instead of halving intervals, which makes it ideal for sequential access systems (like tapes or large memory arrays).

It shines where comparison count matters or when random access is expensive.

### What Problem Are We Solving??

You want to search for an element in a sorted array efficiently, but instead of halving the interval (as in binary search), you want to use Fibonacci numbers to decide where to look — minimizing comparisons and taking advantage of arithmetic-friendly jumps.

## **Example**

Let's search for  $x = 55$  in:

```
arr = [10, 22, 35, 40, 45, 50, 80, 82, 85, 90, 100]
```

1. Find smallest Fibonacci number length (11)  $\rightarrow F(7) = 13$
2. Use Fibonacci offsets to decide index jumps.
3. Check mid using  $fibM2 = 5$  ( $F(5)=5 \rightarrow$  index 4):  $arr[4] = 45 < 55$
4. Move window and repeat until found or narrowed.

## **How Does It Work (Plain Language)?**

Instead of splitting in half like binary search, it splits using ratios of Fibonacci numbers, keeping subarray sizes close to golden ratio.

This approach reduces comparisons and works especially well when array size is known and access cost is linear or limited.

Think of it as a mathematically balanced jump search guided by Fibonacci spacing.

## **Step-by-Step Process**

| Step | Description                                                               |
|------|---------------------------------------------------------------------------|
| 1    | Compute smallest Fibonacci number $F_m \leq n$                            |
| 2    | Set offsets: $F_{m1} = F_{m-1}$ , $F_{m2} = F_{m-2}$                      |
| 3    | Compare target with $arr[offset + F_{m2}]$                                |
| 4    | If smaller, search left subarray (shift $F_{m1}, F_{m2}$ )                |
| 5    | If larger, search right subarray (update offset, shift $F_{m1}, F_{m2}$ ) |
| 6    | If equal, return index                                                    |
| 7    | Continue until $F_{m1} = 1$                                               |

## **Tiny Code (Easy Versions)**

### **Python**

```

def fibonacci_search(arr, x):
    n = len(arr)

    # Initialize fibonacci numbers
    fibMMm2 = 0 # F(m-2)
    fibMMm1 = 1 # F(m-1)
    fibM = fibMMm2 + fibMMm1 # F(m)

    # Find smallest Fibonacci >= n
    while fibM < n:
        fibMMm2 = fibMMm1
        fibMMm1 = fibM
        fibM = fibMMm2 + fibMMm1

    offset = -1

    while fibM > 1:
        i = min(offset + fibMMm2, n - 1)

        if arr[i] < x:
            fibM = fibMMm1
            fibMMm1 = fibMMm2
            fibMMm2 = fibM - fibMMm1
            offset = i
        elif arr[i] > x:
            fibM = fibMMm2
            fibMMm1 -= fibMMm2
            fibMMm2 = fibM - fibMMm1
        else:
            return i

    if fibMMm1 and offset + 1 < n and arr[offset + 1] == x:
        return offset + 1

    return -1

arr = [10, 22, 35, 40, 45, 50, 80, 82, 85, 90, 100]
print("Found at index:", fibonacci_search(arr, 55)) # Output: -1
print("Found at index:", fibonacci_search(arr, 85)) # Output: 8

```

Output:

```
Found at index: -1
Found at index: 8
```

## C

```
#include <stdio.h>

int min(int a, int b) { return (a < b) ? a : b; }

int fibonacci_search(int arr[], int n, int x) {
    int fibMMm2 = 0;    // F(m-2)
    int fibMMm1 = 1;    // F(m-1)
    int fibM = fibMMm2 + fibMMm1; // F(m)

    while (fibM < n) {
        fibMMm2 = fibMMm1;
        fibMMm1 = fibM;
        fibM = fibMMm2 + fibMMm1;
    }

    int offset = -1;

    while (fibM > 1) {
        int i = min(offset + fibMMm2, n - 1);

        if (arr[i] < x) {
            fibM = fibMMm1;
            fibMMm1 = fibMMm2;
            fibMMm2 = fibM - fibMMm1;
            offset = i;
        } else if (arr[i] > x) {
            fibM = fibMMm2;
            fibMMm1 -= fibMMm2;
            fibMMm2 = fibM - fibMMm1;
        } else {
            return i;
        }
    }

    if (fibMMm1 && offset + 1 < n && arr[offset + 1] == x)
        return offset + 1;
}
```

```

    return -1;
}

int main(void) {
    int arr[] = {10, 22, 35, 40, 45, 50, 80, 82, 85, 90, 100};
    int n = sizeof(arr)/sizeof(arr[0]);
    int x = 85;
    int idx = fibonacci_search(arr, n, x);
    if (idx != -1)
        printf("Found at index: %d\n", idx);
    else
        printf("Not found\n");
}

```

Output:

```
Found at index: 8
```

## Why It Matters

- Uses Fibonacci numbers to reduce comparisons
- Efficient for sorted arrays with sequential access
- Avoids division (only addition/subtraction)
- Inspired by golden ratio search (optimal probing)
- Excellent teaching tool for divide-and-conquer logic

## A Gentle Proof (Why It Works)

The Fibonacci split maintains nearly golden ratio balance.  
At each step, one subproblem has size  $\frac{1}{\phi}$  of the previous.

So total steps number of Fibonacci numbers  $\leq n$ ,  
which grows as  $O(\log_\phi n) \approx O(\log n)$ .

Thus, the time complexity is the same as binary search,  
but with fewer comparisons and more efficient arithmetic.

## Try It Yourself

1. Search 45 in [10, 22, 35, 40, 45, 50, 80, 82, 85, 90, 100]
2. Search 100 (last element)
3. Search 10 (first element)
4. Search value not in array
5. Count comparisons made
6. Compare with binary search
7. Try on length = Fibonacci number (e.g. 13)
8. Visualize index jumps
9. Modify to print intervals
10. Apply to sorted strings (lex order)

## Test Cases

| Array                               | Target | Output | Notes     |
|-------------------------------------|--------|--------|-----------|
| [10,22,35,40,45,50,80,82,85,90,100] | 85     | 8      | Found     |
| [10,22,35,40,45]                    | 22     | 1      | Found     |
| [1,2,3,5,8,13,21]                   | 21     | 6      | Found     |
| [2,4,6,8,10]                        | 7      | -1     | Not found |

## Complexity

| Aspect                | Value               |
|-----------------------|---------------------|
| Time                  | $O(\log n)$         |
| Space                 | $O(1)$              |
| Comparisons           | $\log(n)$           |
| Access type           | Sequential-friendly |
| Requires sorted input | Yes                 |

Fibonacci Search, a golden search for discrete worlds, where each step follows nature's rhythm.

## 170 Jump + Binary Hybrid

Jump + Binary Hybrid Search blends the best of two worlds, Jump Search for fast skipping and Binary Search for precise refinement. It's perfect when your data is sorted, and you want a balance between linear jumps and logarithmic probing within small subranges.

## **What Problem Are We Solving??**

Binary search is powerful but needs random access (you can jump anywhere). Jump search works well for sequential data (like linked blocks or caches) but may overshoot.

This hybrid combines them:

1. Jump ahead in fixed steps to find the block.
2. Once you know the target range, switch to binary search inside it.

It's a practical approach for sorted datasets with limited random access (like disk blocks or database pages).

## **Example**

Search for 45 in [10, 22, 35, 40, 45, 50, 80, 82, 85, 90, 100].

Step 1: Jump by block size  $\sqrt{n} = 3$

- Check `arr[2] = 35` →  $35 < 45$
- Check `arr[5] = 50` →  $50 > 45$

Now we know target is in [35, 40, 45, 50] → indices [3..5]

Step 2: Binary search within block [3..5]

- $\text{Mid} = 4 \rightarrow \text{arr}[4] = 45$  Found!

## **How Does It Work (Plain Language)?**

1. Choose block size  $m = \sqrt{n}$ .
2. Jump ahead by  $m$  until you pass or reach the target.
3. Once in the block, switch to binary search inside.

Jumping narrows the search zone quickly, Binary search finishes it cleanly, fewer comparisons than either alone.

## **Step-by-Step Process**

| Step | Description                                              |
|------|----------------------------------------------------------|
| 1    | Compute block size $m = \lfloor \sqrt{n} \rfloor$        |
| 2    | Jump by $m$ until $\text{arr}[j] = \text{target}$ or end |
| 3    | Determine block $[j - m, j)$                             |
| 4    | Run binary search inside that block                      |
| 5    | Return index or -1 if not found                          |

### Tiny Code (Easy Versions)

#### Python

```
import math

def jump_binary_search(arr, x):
    n = len(arr)
    step = int(math.sqrt(n))
    prev = 0

    # Jump phase
    while prev < n and arr[min(step, n) - 1] < x:
        prev = step
        step += int(math.sqrt(n))
        if prev >= n:
            return -1

    # Binary search in block
    low, high = prev, min(step, n) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] < x:
            low = mid + 1
        else:
            high = mid - 1

    return -1

arr = [10, 22, 35, 40, 45, 50, 80, 82, 85, 90, 100]
print("Found at index:", jump_binary_search(arr, 45))
```

Output:

Found at index: 4

C

```
#include <stdio.h>
#include <math.h>

int jump_binary_search(int arr[], int n, int x) {
    int step = (int)sqrt(n);
    int prev = 0;

    // Jump phase
    while (prev < n && arr[(step < n ? step : n) - 1] < x) {
        prev = step;
        step += (int)sqrt(n);
        if (prev >= n)
            return -1;
    }

    // Binary search in block
    int low = prev;
    int high = (step < n ? step : n) - 1;

    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == x)
            return mid;
        else if (arr[mid] < x)
            low = mid + 1;
        else
            high = mid - 1;
    }

    return -1;
}

int main(void) {
    int arr[] = {10, 22, 35, 40, 45, 50, 80, 82, 85, 90, 100};
    int n = sizeof(arr) / sizeof(arr[0]);
    int idx = jump_binary_search(arr, n, 45);
```

```

if (idx != -1)
    printf("Found at index: %d\n", idx);
else
    printf("Not found\n");
}

```

Output:

Found at index: 4

## Why It Matters

- Combines fast skipping (jump) and efficient narrowing (binary)
- Works well on sorted lists with slow access
- Reduces comparisons vs pure jump or linear
- Great for block-based storage and database indexing
- Demonstrates hybrid thinking in algorithm design

## A Gentle Proof (Why It Works)

Jump phase:  $O(\sqrt{n})$  steps

Binary phase:  $O(\log \sqrt{n}) = O(\log n)$

Total:

$$T(n) = O(\sqrt{n}) + O(\log n)$$

For large  $n$ , dominated by  $O(\sqrt{n})$ , but faster in practice.

## Try It Yourself

1. Search 80 in [10, 22, 35, 40, 45, 50, 80, 82, 85, 90, 100]
2. Try 10 (first element)
3. Try 100 (last element)
4. Try value not in array
5. Compare comparisons with binary search
6. Change block size (try  $2\sqrt{n}$  or  $n/4$ )
7. Print jumps and block
8. Run on array of length 100
9. Combine with exponential block discovery
10. Extend for descending arrays

## Test Cases

| Array                               | Target | Output | Notes           |
|-------------------------------------|--------|--------|-----------------|
| [10,22,35,40,45,50,80,82,85,90,100] | 45     | 4      | Found           |
| [10,22,35,40,45]                    | 10     | 0      | Found at start  |
| [10,22,35,40,45]                    | 100    | -1     | Not found       |
| [1,3,5,7,9,11,13]                   | 7      | 3      | Found in middle |

## Complexity

| Aspect          | Value                  |
|-----------------|------------------------|
| Time            | $O(\sqrt{n} + \log n)$ |
| Space           | $O(1)$                 |
| Requires sorted | Yes                    |
| Stable          | Yes                    |
| Adaptive        | No                     |

Jump + Binary Hybrid, leap with purpose, then zero in. It's how explorers search with both speed and focus.

## Section 18. Selection Algorithms

### 171 Quickselect

Quickselect is a selection algorithm to find the k-th smallest element in an unsorted array — faster on average than sorting the entire array.

It's based on the same partitioning idea as Quicksort, but only recurses into the side that contains the desired element.

Average time complexity:  $O(n)$  Worst-case (rare):  $O(n^2)$

### What Problem Are We Solving??

Suppose you have an unsorted list and you want:

- The median,
- The k-th smallest, or

- The k-th largest element,

You don't need to fully sort, you just need one order statistic.

Quickselect solves this by partitioning the array and narrowing focus to the relevant half only.

### **Example**

Find 4th smallest element in: [7, 2, 1, 6, 8, 5, 3, 4]

1. Choose pivot (e.g. 4).
2. Partition → [2, 1, 3] [4] [7, 6, 8, 5]
3. Pivot position = 3 (0-based)
4.  $k = 4 \rightarrow$  pivot index 3 matches → 4 is 4th smallest

No need to sort the rest!

### **How Does It Work (Plain Language)?**

Quickselect picks a pivot, partitions the list into less-than and greater-than parts, and decides which side to recurse into based on the pivot's index vs. target  $k$ .

It's a divide and conquer search on positions, not order.

### **Step-by-Step Process**

| Step | Description                             |
|------|-----------------------------------------|
| 1    | Pick pivot (random or last element)     |
| 2    | Partition array around pivot            |
| 3    | Get pivot index $p$                     |
| 4    | If $p == k$ , return element            |
| 5    | If $p > k$ , search left                |
| 6    | If $p < k$ , search right (adjust $k$ ) |

### **Tiny Code (Easy Versions)**

#### **Python**

```

import random

def partition(arr, low, high):
    pivot = arr[high]
    i = low
    for j in range(low, high):
        if arr[j] < pivot:
            arr[i], arr[j] = arr[j], arr[i]
            i += 1
    arr[i], arr[high] = arr[high], arr[i]
    return i

def quickselect(arr, k):
    low, high = 0, len(arr) - 1
    while low <= high:
        pivot_index = partition(arr, low, high)
        if pivot_index == k:
            return arr[pivot_index]
        elif pivot_index > k:
            high = pivot_index - 1
        else:
            low = pivot_index + 1

arr = [7, 2, 1, 6, 8, 5, 3, 4]
k = 3 # 0-based index: 4th smallest
print("4th smallest:", quickselect(arr, k))

```

Output:

4th smallest: 4

C

```

#include <stdio.h>

void swap(int *a, int *b) {
    int t = *a;
    *a = *b;
    *b = t;
}

```

```

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low;
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            swap(&arr[i], &arr[j]);
            i++;
        }
    }
    swap(&arr[i], &arr[high]);
    return i;
}

int quickselect(int arr[], int low, int high, int k) {
    if (low <= high) {
        int pi = partition(arr, low, high);
        if (pi == k)
            return arr[pi];
        else if (pi > k)
            return quickselect(arr, low, pi - 1, k);
        else
            return quickselect(arr, pi + 1, high, k);
    }
    return -1;
}

int main(void) {
    int arr[] = {7, 2, 1, 6, 8, 5, 3, 4};
    int n = sizeof(arr) / sizeof(arr[0]);
    int k = 3; // 4th smallest (0-based)
    printf("4th smallest: %d\n", quickselect(arr, 0, n - 1, k));
}

```

Output:

4th smallest: 4

## Why It Matters

- Find median in linear time (expected)
- Avoid sorting when you only need one element

- Basis for algorithms like Median of Medians, BFPRT
- Common in order statistics, quantiles, top-k problems
- Used in libraries (e.g. `nth_element` in C++)

### A Gentle Proof (Why It Works)

Each partition reduces problem size by eliminating one side. Average split half  $\rightarrow O(n)$  expected comparisons. Worst case (bad pivot)  $\rightarrow O(n^2)$ , but with randomized pivot, very unlikely.

Expected time:

$$T(n) = T(n/2) + O(n) \Rightarrow O(n)$$

### Try It Yourself

1. Find 1st smallest (min)
2. Find last (max)
3. Find median ( $k = n/2$ )
4. Add random pivoting
5. Count comparisons per iteration
6. Modify for k-th largest ( $n-k$ )
7. Compare runtime with full sort
8. Visualize partition steps
9. Test on repeated elements
10. Combine with deterministic pivot

### Test Cases

| Array                  | k (0-based) | Output | Notes          |
|------------------------|-------------|--------|----------------|
| [7,2,1,6,8,5,3,4]      | 3           | 4      | 4th smallest   |
| [3,1,2]                | 1           | 2      | median         |
| [10,80,30,90,40,50,70] | 4           | 70     | middle element |
| [5,5,5,5]              | 2           | 5      | duplicates     |

### Complexity

| Aspect         | Value  |
|----------------|--------|
| Time (Average) | $O(n)$ |

| Aspect       | Value       |
|--------------|-------------|
| Time (Worst) | $O(n^2)$    |
| Space        | $O(1)$      |
| Stable       | No          |
| In-place     | Yes         |
| Randomized   | Recommended |

Quickselect, the surgical strike of sorting: find exactly what you need, and ignore the rest.

## 172 Median of Medians

Median of Medians is a deterministic selection algorithm that guarantees  $O(n)$  worst-case time for finding the  $k$ -th smallest element. It improves on Quickselect, which can degrade to  $(O(n^2))$  in unlucky cases, by carefully choosing a good pivot every time.

It's a cornerstone of theoretical computer science, balancing speed and worst-case safety.

### What Problem Are We Solving??

In Quickselect, a bad pivot can lead to unbalanced partitions (like always picking smallest/largest). Median of Medians fixes this by ensuring the pivot is “good enough” — always splitting the array so that each side has at least a constant fraction of elements.

Goal: Find the  $k$ -th smallest element deterministically in  $O(n)$ , no randomness, no risk.

### Example

Find 5th smallest in [12, 3, 5, 7, 4, 19, 26, 23, 8, 15]

1. Split into groups of 5: [12, 3, 5, 7, 4], [19, 26, 23, 8, 15]
2. Sort each group: [3,4,5,7,12], [8,15,19,23,26]
3. Take medians: [5, 19]
4. Find median of medians: 19
5. Partition array around 19
6. Recurse on appropriate side until  $k$ -th found

Pivot 19 ensures balanced split, leading to linear runtime.

## How Does It Work (Plain Language)?

1. Break array into groups of 5
2. Sort each small group (cheap)
3. Collect all medians of groups
4. Recursively find median of those medians → good pivot
5. Partition around pivot
6. Recurse into half that contains the k-th element

Each level discards a constant fraction →  $(O(n))$  total work.

## Step-by-Step Process

| Step | Description                          |
|------|--------------------------------------|
| 1    | Divide array into chunks of 5        |
| 2    | Sort each chunk                      |
| 3    | Extract medians                      |
| 4    | Find median of medians recursively   |
| 5    | Partition using this pivot           |
| 6    | Recurse into correct side based on k |

## Tiny Code (Easy Versions)

### Python

```
def partition(arr, pivot):  
    less = [x for x in arr if x < pivot]  
    equal = [x for x in arr if x == pivot]  
    greater = [x for x in arr if x > pivot]  
    return less, equal, greater  
  
def select(arr, k):  
    if len(arr) <= 5:  
        return sorted(arr)[k]  
  
    # Step 1: group in chunks of 5  
    chunks = [arr[i:i+5] for i in range(0, len(arr), 5)]  
  
    # Step 2: find medians  
    medians = [sorted(chunk)[len(chunk)//2] for chunk in chunks]
```

```

# Step 3: pivot = median of medians
pivot = select(medians, len(medians)//2)

# Step 4: partition
less, equal, greater = partition(arr, pivot)

# Step 5: recurse
if k < len(less):
    return select(less, k)
elif k < len(less) + len(equal):
    return pivot
else:
    return select(greater, k - len(less) - len(equal))

arr = [12, 3, 5, 7, 4, 19, 26, 23, 8, 15]
k = 4 # 0-based: 5th smallest
print("5th smallest:", select(arr, k))

```

Output:

5th smallest: 8

## C (Simplified Version)

(*Pseudocode-like clarity for readability*)

```

#include <stdio.h>
#include <stdlib.h>

int cmp(const void* a, const void* b) {
    return (*(int*)a - *(int*)b);
}

int median_of_medians(int arr[], int n, int k);

int select_group_median(int arr[], int n) {
    qsort(arr, n, sizeof(int), cmp);
    return arr[n/2];
}

```

```

int median_of_medians(int arr[], int n, int k) {
    if (n <= 5) {
        qsort(arr, n, sizeof(int), cmp);
        return arr[k];
    }

    int groups = (n + 4) / 5;
    int medians[groups];
    for (int i = 0; i < groups; i++) {
        int size = (i*5 + 5 <= n) ? 5 : n - i*5;
        medians[i] = select_group_median(arr + i*5, size);
    }

    int pivot = median_of_medians(medians, groups/2);

    // Partition
    int less[n], greater[n], l = 0, g = 0, equal = 0;
    for (int i = 0; i < n; i++) {
        if (arr[i] < pivot) less[l++] = arr[i];
        else if (arr[i] > pivot) greater[g++] = arr[i];
        else equal++;
    }

    if (k < l)
        return median_of_medians(less, l, k);
    else if (k < l + equal)
        return pivot;
    else
        return median_of_medians(greater, g, k - l - equal);
}

int main(void) {
    int arr[] = {12, 3, 5, 7, 4, 19, 26, 23, 8, 15};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("5th smallest: %d\n", median_of_medians(arr, n, 4));
}

```

Output:

5th smallest: 8

## Why It Matters

- Guaranteed  $O(n)$  even in worst case
- No bad pivots  $\rightarrow$  stable performance
- Basis for BFPRT algorithm
- Used in theoretical guarantees for real systems
- Key for deterministic selection, safe quantile computations

## A Gentle Proof (Why It Works)

Each pivot ensures at least 30% of elements are discarded each recursion (proof via grouping).

Recurrence:

$$T(n) = T(n/5) + T(7n/10) + O(n) \Rightarrow O(n)$$

Thus, always linear time, even worst case.

## Try It Yourself

1. Find median of [5, 2, 1, 3, 4]
2. Test with duplicates
3. Compare with Quickselect runtime
4. Count recursive calls
5. Change group size to 3 or 7
6. Visualize grouping steps
7. Print pivot each round
8. Apply to large random list
9. Benchmark vs sorting
10. Implement as pivot strategy for Quickselect

## Test Cases

| Array                      | k | Output | Notes        |
|----------------------------|---|--------|--------------|
| [12,3,5,7,4,19,26,23,8,15] | 4 | 8      | 5th smallest |
| [5,2,1,3,4]                | 2 | 3      | Median       |
| [7,7,7,7]                  | 2 | 7      | Duplicates   |
| [10,9,8,7,6,5]             | 0 | 5      | Min          |

## Complexity

| Aspect        | Value  |
|---------------|--------|
| Time (Worst)  | $O(n)$ |
| Time (Avg)    | $O(n)$ |
| Space         | $O(n)$ |
| Stable        | No     |
| Deterministic | Yes    |

Median of Medians, a balanced thinker in the world of selection: slow and steady, but always linear.

## 173 Randomized Select

Randomized Select is a probabilistic version of Quickselect, where the pivot is chosen randomly to avoid worst-case behavior. This small twist makes the algorithm's expected time  $O(n)$ , even though the worst case remains ( $O(n^2)$ ). In practice, it's fast, simple, and robust, a true workhorse for order statistics.

### What Problem Are We Solving??

You need the k-th smallest element in an unsorted list. Quickselect works well, but choosing the first or last element as pivot can cause bad splits.

Randomized Select improves this by picking a random pivot, making bad luck rare and performance stable.

### Example

Find 4th smallest in [7, 2, 1, 6, 8, 5, 3, 4]

1. Pick random pivot (say 5)
2. Partition  $\rightarrow [2,1,3,4] \ [5] \ [7,6,8]$
3. Pivot index = 4  $\rightarrow 4 > 3$ , so recurse on left [2,1,3,4]
4. Pick random pivot again (say 3)
5. Partition  $\rightarrow [2,1] \ [3] \ [4]$
6. Index 2 = k=3  $\rightarrow$  found 4th smallest = 4

## How Does It Work (Plain Language)?

It's Quickselect with random pivoting. At each step:

- Pick a random element as pivot.
- Partition around pivot.
- Only recurse into one side (where  $k$  lies).

This randomness ensures average-case balance, even on adversarial inputs.

## Step-by-Step Process

| Step | Description                              |
|------|------------------------------------------|
| 1    | Pick random pivot index                  |
| 2    | Partition array around pivot             |
| 3    | Get pivot index $p$                      |
| 4    | If $p == k$ , return element             |
| 5    | If $p > k$ , recurse left                |
| 6    | If $p < k$ , recurse right (adjust $k$ ) |

## Tiny Code (Easy Versions)

### Python

```
import random

def partition(arr, low, high):
    pivot = arr[high]
    i = low
    for j in range(low, high):
        if arr[j] < pivot:
            arr[i], arr[j] = arr[j], arr[i]
            i += 1
    arr[i], arr[high] = arr[high], arr[i]
    return i

def randomized_select(arr, low, high, k):
    if low == high:
        return arr[low]
```

```

pivot_index = random.randint(low, high)
arr[pivot_index], arr[high] = arr[high], arr[pivot_index]

p = partition(arr, low, high)

if p == k:
    return arr[p]
elif p > k:
    return randomized_select(arr, low, p - 1, k)
else:
    return randomized_select(arr, p + 1, high, k)

arr = [7, 2, 1, 6, 8, 5, 3, 4]
k = 3 # 4th smallest
print("4th smallest:", randomized_select(arr, 0, len(arr)-1, k))

```

Output:

4th smallest: 4

## C

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void swap(int *a, int *b) {
    int t = *a; *a = *b; *b = t;
}

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low;
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            swap(&arr[i], &arr[j]);
            i++;
        }
    }
    swap(&arr[i], &arr[high]);
    return i;
}

```

```

}

int randomized_select(int arr[], int low, int high, int k) {
    if (low == high) return arr[low];

    int pivot_index = low + rand() % (high - low + 1);
    swap(&arr[pivot_index], &arr[high]);
    int p = partition(arr, low, high);

    if (p == k) return arr[p];
    else if (p > k) return randomized_select(arr, low, p - 1, k);
    else return randomized_select(arr, p + 1, high, k);
}

int main(void) {
    srand(time(NULL));
    int arr[] = {7, 2, 1, 6, 8, 5, 3, 4};
    int n = sizeof(arr)/sizeof(arr[0]);
    int k = 3;
    printf("4th smallest: %d\n", randomized_select(arr, 0, n - 1, k));
}

```

Output:

4th smallest: 4

## Why It Matters

- Expected  $O(n)$  time, simple, and practical
- Avoids worst-case trap of fixed-pivot Quickselect
- Great for top-k queries, quantiles, median
- Combines simplicity + randomness = robust performance
- Commonly used in competitive programming and real-world systems

## A Gentle Proof (Why It Works)

Expected recurrence:

$$T(n) = T(\alpha n) + O(n)$$

where  $\alpha$  is random, expected  $\approx \frac{1}{2}$   
 $\rightarrow T(n) = O(n)$

Worst case still  $O(n^2)$ , but rare.

Expected comparisons  $\approx 2n$ .

### Try It Yourself

1. Run multiple times and observe pivot randomness
2. Compare with deterministic Quickselect
3. Count recursive calls
4. Test with sorted input (robust)
5. Test all same elements
6. Change k (first, median, last)
7. Modify to find k-th largest ( $n-k-1$ )
8. Compare performance with sort()
9. Log pivot indices
10. Measure runtime on 10 elements

### Test Cases

| Array                  | k | Output | Notes              |
|------------------------|---|--------|--------------------|
| [7,2,1,6,8,5,3,4]      | 3 | 4      | 4th smallest       |
| [10,80,30,90,40,50,70] | 4 | 70     | Works on any order |
| [1,2,3,4,5]            | 0 | 1      | Sorted input safe  |
| [5,5,5,5]              | 2 | 5      | Duplicates fine    |

### Complexity

| Aspect          | Value    |
|-----------------|----------|
| Time (Expected) | $O(n)$   |
| Time (Worst)    | $O(n^2)$ |
| Space           | $O(1)$   |
| Stable          | No       |
| Randomized      | Yes      |
| In-place        | Yes      |

Randomized Select, a game of chance that almost always wins: fast, fair, and beautifully simple.

## 174 Binary Search on Answer

Binary Search on Answer (also called Parametric Search) is a powerful optimization trick used when the search space is monotonic—meaning once a condition becomes true, it stays true (or vice versa). Instead of searching a sorted array, we’re searching for the smallest or largest value that satisfies a condition.

### What Problem Are We Solving??

Sometimes you don’t have an array to search, but you need to minimize or maximize a numeric answer. Examples:

- Minimum capacity to transport items in  $k$  days
- Minimum maximum distance between routers
- Maximum median satisfying a condition

We can’t iterate all possibilities efficiently, but we can binary search the answer space.

### Example

Problem: Given array [1, 2, 3, 4, 5], split into 2 parts, minimize the largest sum among parts.

We can’t directly find it, but if we can check whether a candidate value  $mid$  is valid (can split with sum  $= mid$ ), we can binary search on  $mid$ .

| mid | canSplit(nums, 2, mid)        | Result                |
|-----|-------------------------------|-----------------------|
| 9   | True (splits: [1,2,3,4], [5]) | ok → move left        |
| 7   | False                         | move right            |
| 8   | True                          | ok → final answer = 9 |

Result = 9

### How Does It Work (Plain Language)?

You don’t search elements, you search values. You define a function `can(mid)` that checks if a solution is possible with  $mid$ . Then use binary search to narrow down the range until you find the optimal value.

## Step-by-Step Process

| Step | Description                                                                                       |
|------|---------------------------------------------------------------------------------------------------|
| 1    | Define the range of answers (lo, hi)                                                              |
| 2    | While $lo < hi$ :<br>$mid = (lo + hi) // 2$<br>If $can(mid)$ : $hi = mid$<br>Else: $lo = mid + 1$ |
| 3    | Return $lo$ as the optimal answer                                                                 |

## Tiny Code (Easy Versions)

### Python

```
def can_split(nums, k, mid):
    count, current = 1, 0
    for x in nums:
        if current + x > mid:
            count += 1
            current = x
        else:
            current += x
    return count <= k

def binary_search_answer(nums, k):
    lo, hi = max(nums), sum(nums)
    while lo < hi:
        mid = (lo + hi) // 2
        if can_split(nums, k, mid):
            hi = mid
        else:
            lo = mid + 1
    return lo

nums = [1, 2, 3, 4, 5]
k = 2
print("Minimum largest sum:", binary_search_answer(nums, k))
```

Output:

```
Minimum largest sum: 9
```

## C

```
#include <stdio.h>

int can_split(int arr[], int n, int k, int mid) {
    int count = 1, sum = 0;
    for (int i = 0; i < n; i++) {
        if (arr[i] > mid) return 0;
        if (sum + arr[i] > mid) {
            count++;
            sum = arr[i];
        } else {
            sum += arr[i];
        }
    }
    return count <= k;
}

int binary_search_answer(int arr[], int n, int k) {
    int lo = arr[0], hi = 0;
    for (int i = 0; i < n; i++) {
        if (arr[i] > lo) lo = arr[i];
        hi += arr[i];
    }
    while (lo < hi) {
        int mid = (lo + hi) / 2;
        if (can_split(arr, n, k, mid))
            hi = mid;
        else
            lo = mid + 1;
    }
    return lo;
}

int main(void) {
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr)/sizeof(arr[0]);
    int k = 2;
    printf("Minimum largest sum: %d\n", binary_search_answer(arr, n, k));
}
```

Output:

```
Minimum largest sum: 9
```

## Why It Matters

- Solves optimization problems without brute force
- Turns decision problems into search problems
- A universal pattern: works for capacity, distance, time, etc.
- Common in LeetCode, interviews, and competitive programming

## A Gentle Proof (Why It Works)

If a function  $f(x)$  is monotonic (true after a point or false after a point), binary search can find the threshold. Formally:

If  $f(lo) = \text{false}$ ,  $f(hi) = \text{true}$ , and  $f(x)$  is monotonic, then binary search converges to smallest  $x$  such that  $f(x) = \text{true}$ .

## Try It Yourself

1. Find smallest capacity to ship packages in  $d$  days
2. Find smallest max page load per student (book allocation)
3. Find largest minimum distance between routers
4. Find smallest time to paint all boards
5. Find minimum speed to reach on time
6. Define a monotonic function `can(x)` and apply search
7. Experiment with `float` range and tolerance
8. Try `max` instead of `min` (reverse condition)
9. Count binary search steps for each case
10. Compare with brute force

## Test Cases

| Problem          | Input              | Output | Explanation     |
|------------------|--------------------|--------|-----------------|
| Split array      | [1,2,3,4,5], k=2   | 9      | [1,2,3,4],[5]   |
| Book allocation  | [10,20,30,40], k=2 | 60     | [10,20,30],[40] |
| Router placement | [1,2,8,12], k=3    | 5      | Place at 1,6,12 |

## Complexity

| Aspect                | Value                        |
|-----------------------|------------------------------|
| Time                  | $O(n \log(\max - \min))$     |
| Space                 | $O(1)$                       |
| Monotonicity Required | Yes                          |
| Type                  | Decision-based binary search |

Binary Search on Answer, when you can't sort the data, sort the solution space.

## 175 Order Statistics Tree

An Order Statistics Tree is a special kind of augmented binary search tree (BST) that supports two powerful operations efficiently:

1. Select( $k$ ): find the  $k$ -th smallest element.
2. Rank( $x$ ): find the position (rank) of element  $x$ .

It's a classic data structure where each node stores subtree size, allowing order-based queries in  $O(\log n)$  time.

### What Problem Are We Solving??

Sometimes you don't just want to search by key, you want to search by order. For example:

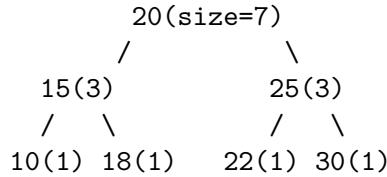
- “What’s the 5th smallest element?”
- “What rank is 37 in the tree?”
- “How many numbers  $\leq 50$  are there?”

An order statistics tree gives you both key-based and rank-based access in one structure.

### Example

Suppose you insert [20, 15, 25, 10, 18, 22, 30].

Each node stores **size** (the number of nodes in its subtree).



Select(4) → 20 (the 4th smallest) Rank(22) → 6 (22 is the 6th smallest)

### How Does It Work (Plain Language)?

Every node tracks how many nodes exist in its subtree (left + right + itself). When you traverse:

- To select k-th smallest, compare k with size of left subtree.
- To find rank of x, accumulate sizes while traversing down.

### Select(k) Pseudocode

```

select(node, k):
    left_size = size(node.left)
    if k == left_size + 1: return node.key
    if k <= left_size: return select(node.left, k)
    else: return select(node.right, k - left_size - 1)

```

### Rank(x) Pseudocode

```

rank(node, x):
    if node == NULL: return 0
    if x < node.key: return rank(node.left, x)
    if x == node.key: return size(node.left) + 1
    else: return size(node.left) + 1 + rank(node.right, x)

```

### Tiny Code (Easy Versions)

#### Python

```

class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.size = 1

def update_size(node):
    if node:
        node.size = 1 + (node.left.size if node.left else 0) + (node.right.size if node.right)

def insert(node, key):
    if node is None:
        return Node(key)
    if key < node.key:
        node.left = insert(node.left, key)
    else:
        node.right = insert(node.right, key)
    update_size(node)
    return node

def select(node, k):
    left_size = node.left.size if node.left else 0
    if k == left_size + 1:
        return node.key
    elif k <= left_size:
        return select(node.left, k)
    else:
        return select(node.right, k - left_size - 1)

def rank(node, key):
    if node is None:
        return 0
    if key < node.key:
        return rank(node.left, key)
    elif key == node.key:
        return (node.left.size if node.left else 0) + 1
    else:
        left_size = node.left.size if node.left else 0
        return left_size + rank(node.right, key)

root = None

```

```

for x in [20, 15, 25, 10, 18, 22, 30]:
    root = insert(root, x)

print("Select(4):", select(root, 4))  # 20
print("Rank(22):", rank(root, 22))    # 6

```

## C (Conceptual Skeleton)

```

typedef struct Node {
    int key;
    int size;
    struct Node *left, *right;
} Node;

int size(Node* n) { return n ? n->size : 0; }

void update_size(Node* n) {
    if (n) n->size = 1 + size(n->left) + size(n->right);
}

Node* new_node(int key) {
    Node* n = malloc(sizeof(Node));
    n->key = key; n->size = 1;
    n->left = n->right = NULL;
    return n;
}

Node* insert(Node* root, int key) {
    if (!root) return new_node(key);
    if (key < root->key) root->left = insert(root->left, key);
    else root->right = insert(root->right, key);
    update_size(root);
    return root;
}

int select_k(Node* root, int k) {
    int left_size = size(root->left);
    if (k == left_size + 1) return root->key;
    else if (k <= left_size) return select_k(root->left, k);
}

```

```

    else return select_k(root->right, k - left_size - 1);
}

```

## Why It Matters

- Useful in rank-based queries, median finding, and order-statistics problems
- Core to balanced trees (AVL, Red-Black, Treaps) with order augmentation
- Enables dynamic median queries and range counting

You can imagine it as a self-updating leaderboard, always knowing who's in position  $k$ .

## A Gentle Proof (Why It Works)

Because subtree sizes are updated correctly on insertions/deletions, each traversal can compute ranks or  $k$ -th values in  $O(\log n)$  time (in balanced trees). If balanced (like in an AVL or RB-tree), operations remain logarithmic.

## Try It Yourself

1. Build an Order Statistics Tree for  $[10, 20, 30, 40, 50]$ .
2. Find `Select(3)` and `Rank(40)`.
3. Insert new elements and recheck ranks.
4. Extend to find median dynamically.
5. Modify to support deletions.
6. Compare with sorting then indexing ( $O(n \log n)$  vs  $O(\log n)$ ).
7. Try building on top of Red-Black Tree.
8. Use for running percentiles.
9. Explore dynamic segment trees for same queries.
10. Implement `countLessThan(x)` using rank.

## Test Cases

| Query     | Expected Result |
|-----------|-----------------|
| Select(1) | 10              |
| Select(4) | 20              |
| Rank(10)  | 1               |
| Rank(22)  | 6               |
| Rank(30)  | 7               |

## Complexity

| Operation       | Complexity  |
|-----------------|-------------|
| Insert / Delete | $O(\log n)$ |
| Select( $k$ )   | $O(\log n)$ |
| Rank( $x$ )     | $O(\log n)$ |
| Space           | $O(n)$      |

An Order Statistics Tree blends search and ranking, perfect for problems that need to know *what* and *where* at the same time.

## 176 Tournament Tree Selection

A Tournament Tree is a binary tree structure that simulates a knockout tournament among elements. Each match compares two elements, and the winner moves up. It's an elegant way to find minimum, maximum, or even  $k$ -th smallest elements with structured comparisons.

### What Problem Are We Solving??

Finding the minimum or maximum in a list takes  $O(n)$ . But if you also want the second smallest, third smallest, or  $k$ -th, you'd like to reuse earlier comparisons. A tournament tree keeps track of all matches, so you don't need to start over.

### Example

Suppose we have elements: [4, 7, 2, 9, 5, 1, 8, 6].

1. Pair them up: compare (4,7), (2,9), (5,1), (8,6)
2. Winners move up: [4, 2, 1, 6]
3. Next round: (4,2), (1,6) → winners [2, 1]
4. Final match: (2,1) → winner 1

The root of the tree = minimum element (1).

If you store the losing element from each match, you can trace back the second smallest, it must have lost directly to 1.

## How Does It Work (Plain Language)?

Imagine a sports tournament:

- Every player plays one match.
- The winner moves on, loser is eliminated.
- The champion (root) is the smallest element.
- The second smallest is the best among those who lost to the champion.

Each match is one comparison, so total comparisons =  $n - 1$  for the min. To find second min, check  $\log n$  losers.

## Steps

| Step | Description                                                       |
|------|-------------------------------------------------------------------|
| 1    | Build a complete binary tree where each leaf is an element.       |
| 2    | Compare each pair and move winner up.                             |
| 3    | Store “losers” in each node.                                      |
| 4    | The root = min. The second min = min(losers along winner’s path). |

## Tiny Code (Easy Versions)

### Python

```
def tournament_min(arr):
    matches = []
    tree = [[x] for x in arr]
    while len(tree) > 1:
        next_round = []
        for i in range(0, len(tree), 2):
            if i + 1 == len(tree):
                next_round.append(tree[i])
                continue
            a, b = tree[i][0], tree[i+1][0]
            if a < b:
                next_round.append([a, b])
            else:
                next_round.append([b, a])
        matches = next_round
        tree = next_round
```

```

return tree[0][0]

def find_second_min(arr):
    # Build tournament, keep track of losers
    n = len(arr)
    tree = [[x, []] for x in arr]
    while len(tree) > 1:
        next_round = []
        for i in range(0, len(tree), 2):
            if i + 1 == len(tree):
                next_round.append(tree[i])
                continue
            a, a_losers = tree[i]
            b, b_losers = tree[i+1]
            if a < b:
                next_round.append([a, a_losers + [b]])
            else:
                next_round.append([b, b_losers + [a]])
        tree = next_round
    winner, losers = tree[0]
    return winner, min(losers)

arr = [4, 7, 2, 9, 5, 1, 8, 6]
min_val, second_min = find_second_min(arr)
print("Min:", min_val)
print("Second Min:", second_min)

```

Output:

```

Min: 1
Second Min: 2

```

C

```

#include <stdio.h>
#include <limits.h>

int tournament_min(int arr[], int n) {
    int size = n;
    while (size > 1) {
        for (int i = 0; i < size / 2; i++) {

```

```

        arr[i] = (arr[2*i] < arr[2*i + 1]) ? arr[2*i] : arr[2*i + 1];
    }
    size = (size + 1) / 2;
}
return arr[0];
}

int main(void) {
    int arr[] = {4, 7, 2, 9, 5, 1, 8, 6};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Minimum: %d\n", tournament_min(arr, n));
}

```

## Why It Matters

- Finds minimum in  $O(n)$ , second minimum in  $O(n + \log n)$  comparisons
- Reusable for k-th selection if you store all match info
- Forms the backbone of selection networks, parallel sorting, and merge tournaments

## A Gentle Proof (Why It Works)

Each element except the minimum loses exactly once. The minimum element competes in  $\log n$  matches (height of tree). So second minimum must be the smallest of  $\log n$  losers, requiring  $\log n$  extra comparisons.

Total =  $n - 1 + \log n$  comparisons, asymptotically optimal.

## Try It Yourself

1. Build a tournament for [5,3,8,2,9,4].
2. Find minimum and second minimum manually.
3. Modify code to find maximum and second maximum.
4. Print tree rounds to visualize matches.
5. Experiment with uneven sizes (non-power-of-2).
6. Try to extend it to third smallest (hint: store paths).
7. Compare with sorting-based approach.
8. Use tournament structure for pairwise elimination problems.
9. Simulate sports bracket winner path.
10. Count comparisons for each step.

## Test Cases

| Input             | Min | Second Min |
|-------------------|-----|------------|
| [4,7,2,9,5,1,8,6] | 1   | 2          |
| [10,3,6,2]        | 2   | 3          |
| [5,4,3,2,1]       | 1   | 2          |

## Complexity

| Operation           | Complexity  |
|---------------------|-------------|
| Build Tournament    | $O(n)$      |
| Find Minimum        | $O(1)$      |
| Find Second Minimum | $O(\log n)$ |
| Space               | $O(n)$      |

A Tournament Tree turns comparisons into matches, where every element plays once, and the champion reveals not just victory, but the story of every defeat.

## 177 Heap Select (Min-Heap)

Heap Select is a simple, powerful technique for finding the  $k$  smallest (or largest) elements in a collection using a heap. It's one of the most practical selection algorithms, trading minimal code for strong efficiency.

### What Problem Are We Solving??

You often don't need a full sort, just the  $k$  smallest or  $k$  largest items. Examples:

- Find top 10 scores
- Get smallest 5 distances
- Maintain top- $k$  trending topics

A heap (priority queue) makes this easy, keep a running set of size  $k$ , pop or push as needed.

## Example

Find 3 smallest elements in [7, 2, 9, 1, 5, 4].

1. Create max-heap of first  $k=3$  elements  $\rightarrow [7, 2, 9] \rightarrow \text{heap} = [9, 2, 7]$
2. For each next element:
  - $1 < 9 \rightarrow \text{pop } 9, \text{push } 1 \rightarrow \text{heap} = [7, 2, 1]$
  - $5 < 7 \rightarrow \text{pop } 7, \text{push } 5 \rightarrow \text{heap} = [5, 2, 1]$
  - $4 < 5 \rightarrow \text{pop } 5, \text{push } 4 \rightarrow \text{heap} = [4, 2, 1]$

Result  $\rightarrow [1, 2, 4]$  (the 3 smallest)

## How Does It Work (Plain Language)?

You keep a heap of size  $k$ :

- For smallest elements  $\rightarrow$  use a max-heap (remove largest if new smaller appears).
- For largest elements  $\rightarrow$  use a min-heap (remove smallest if new larger appears).

This keeps only the top- $k$  interesting values at all times.

## Step-by-Step Process

| Step | Action                                                                                     |
|------|--------------------------------------------------------------------------------------------|
| 1    | Initialize heap with first $k$ elements                                                    |
| 2    | Convert to max-heap (if looking for smallest $k$ )                                         |
| 3    | For each remaining element $x$ :<br>If $x < \text{heap}[0] \rightarrow \text{replace top}$ |
| 4    | Result is heap contents (unsorted)                                                         |
| 5    | Sort heap if needed for final output                                                       |

## Tiny Code (Easy Versions)

### Python

```

import heapq

def k_smallest(nums, k):
    heap = [-x for x in nums[:k]]
    heapq.heapify(heap)
    for x in nums[k:]:
        if -x > heap[0]:
            heapq.heappop(heap)
            heapq.heappush(heap, -x)
    return sorted([-h for h in heap])

nums = [7, 2, 9, 1, 5, 4]
print("3 smallest:", k_smallest(nums, 3))

```

Output:

3 smallest: [1, 2, 4]

## C

```

#include <stdio.h>
#include <stdlib.h>

void swap(int *a, int *b) { int t = *a; *a = *b; *b = t; }

void heapify(int arr[], int n, int i) {
    int largest = i, l = 2*i+1, r = 2*i+2;
    if (l < n && arr[l] > arr[largest]) largest = l;
    if (r < n && arr[r] > arr[largest]) largest = r;
    if (largest != i) {
        swap(&arr[i], &arr[largest]);
        heapify(arr, n, largest);
    }
}

void build_heap(int arr[], int n) {
    for (int i = n/2 - 1; i >= 0; i--) heapify(arr, n, i);
}

void heap_select(int arr[], int n, int k) {
    int heap[k];

```

```

for (int i = 0; i < k; i++) heap[i] = arr[i];
build_heap(heap, k);
for (int i = k; i < n; i++) {
    if (arr[i] < heap[0]) {
        heap[0] = arr[i];
        heapify(heap, k, 0);
    }
}
printf("%d smallest elements:\n", k);
for (int i = 0; i < k; i++) printf("%d ", heap[i]);
}

int main(void) {
    int arr[] = {7, 2, 9, 1, 5, 4};
    int n = 6;
    heap_select(arr, n, 3);
}

```

Output:

```

3 smallest elements:
1 2 4

```

## Why It Matters

- Avoids full sorting ( $O(n \log n)$ )
- Great for streaming data, sliding windows, top-k problems
- Scales well for large  $n$  and small  $k$
- Used in leaderboards, analytics, data pipelines

If you only need *some* order, don't sort it all.

## A Gentle Proof (Why It Works)

- Building heap:  $O(k)$
- For each new element: compare + heapify =  $O(\log k)$
- Total:  $O(k + (n-k) \log k) = O(n \log k)$

For small  $k$ , this is much faster than sorting.

## Try It Yourself

1. Find 3 largest elements using min-heap
2. Stream numbers from input, maintain smallest 5
3. Track top 10 scores dynamically
4. Compare runtime vs `sorted(nums) [:k]`
5. Try  $k = 1$  (minimum),  $k = n$  (full sort)
6. Modify for objects with custom keys (e.g. score, id)
7. Handle duplicates, keep all or unique only
8. Experiment with random arrays of size 1e6
9. Visualize heap evolution per step
10. Combine with binary search to tune thresholds

## Test Cases

| Input         | k | Output  |
|---------------|---|---------|
| [7,2,9,1,5,4] | 3 | [1,2,4] |
| [10,8,6,4,2]  | 2 | [2,4]   |
| [1,1,1,1]     | 2 | [1,1]   |

## Complexity

| Operation     | Complexity        |
|---------------|-------------------|
| Build Heap    | $O(k)$            |
| Iterate Array | $O((n-k) \log k)$ |
| Total Time    | $O(n \log k)$     |
| Space         | $O(k)$            |

Heap Select is your practical shortcut, sort only what you need, ignore the rest.

## 178 Partial QuickSort

Partial QuickSort is a twist on classic QuickSort, it stops sorting once it has placed the first  $k$  elements (or top  $k$ ) in their correct positions. It's perfect when you need top- $k$  smallest/largest elements but don't need the rest sorted.

Think of it as QuickSort with early stopping, a hybrid between QuickSort and Quickselect.

## **What Problem Are We Solving??**

Sometimes you need only part of the sorted order:

- “Get top 10 scores”
- “Find smallest k elements”
- “Sort first half”

Fully sorting wastes work. Partial QuickSort skips unnecessary partitions.

### **Example**

Array: [9, 4, 6, 2, 8, 1], k = 3

We want smallest 3 elements. QuickSort picks a pivot, partitions array:

```
Pivot = 6  
→ [4, 2, 1] | 6 | [9, 8]
```

Now we know all elements left of pivot (4, 2, 1) are smaller. Since `left_size == k`, we can stop, [1,2,4] are our smallest 3.

### **How Does It Work (Plain Language)?**

Just like QuickSort, but after partition:

- If pivot index == k → done.
- If pivot index > k → recurse only left.
- If pivot index < k → recurse right partially.

You never sort beyond what's needed.

### **Step-by-Step Process**

| Step | Description                             |
|------|-----------------------------------------|
| 1    | Choose pivot                            |
| 2    | Partition array                         |
| 3    | If pivot index == k → done              |
| 4    | If pivot index > k → recurse left       |
| 5    | Else recurse right on remaining portion |

## Tiny Code (Easy Versions)

### Python

```
def partial_quicksort(arr, low, high, k):
    if low < high:
        p = partition(arr, low, high)
        if p > k:
            partial_quicksort(arr, low, p - 1, k)
        elif p < k:
            partial_quicksort(arr, low, p - 1, k)
            partial_quicksort(arr, p + 1, high, k)

def partition(arr, low, high):
    pivot = arr[high]
    i = low - 1
    for j in range(low, high):
        if arr[j] < pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1

arr = [9, 4, 6, 2, 8, 1]
k = 3
partial_quicksort(arr, 0, len(arr) - 1, k - 1)
print("Smallest 3 elements:", sorted(arr[:k]))
```

Output:

```
Smallest 3 elements: [1, 2, 4]
```

### C

```
#include <stdio.h>

void swap(int *a, int *b) { int t = *a; *a = *b; *b = t; }

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
```

```

    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return i + 1;
}

void partial_quicksort(int arr[], int low, int high, int k) {
    if (low < high) {
        int p = partition(arr, low, high);
        if (p > k)
            partial_quicksort(arr, low, p - 1, k);
        else if (p < k)
            partial_quicksort(arr, p + 1, high, k);
    }
}

int main(void) {
    int arr[] = {9, 4, 6, 2, 8, 1};
    int n = 6, k = 3;
    partial_quicksort(arr, 0, n - 1, k - 1);
    printf("Smallest %d elements:\n", k);
    for (int i = 0; i < k; i++) printf("%d ", arr[i]);
}

```

Output:

```

Smallest 3 elements:
1 2 4

```

## Why It Matters

- Efficient top-k selection when order matters
- Avoids sorting unnecessary portions
- Combines strengths of Quickselect and QuickSort
- Works in-place (no extra memory)

Great for partial sorting like “leaderboards”, “top results”, or “bounded priority lists”.

## A Gentle Proof (Why It Works)

QuickSort partitions data into two halves; Only parts that could contain the first  $k$  elements are explored. Average time complexity becomes  $O(n)$  for selection,  $O(n \log k)$  for partial order.

## Try It Yourself

1. Find smallest 5 numbers in  $[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]$
2. Modify to find largest  $k$  instead
3. Compare runtime vs full `sort()`
4. Visualize recursion path
5. Track how many elements actually get sorted
6. Try random pivot vs median pivot
7. Test  $k = 1$  (min) and  $k = n$  (full sort)
8. Measure comparisons count
9. Try with duplicates
10. Combine with heap for hybrid version

## Test Cases

| Input           | $k$ | Output    |
|-----------------|-----|-----------|
| $[9,4,6,2,8,1]$ | 3   | $[1,2,4]$ |
| $[5,4,3,2,1]$   | 2   | $[1,2]$   |
| $[7,7,7,7]$     | 2   | $[7,7]$   |

## Complexity

| Aspect       | Value    |
|--------------|----------|
| Average Time | $O(n)$   |
| Worst Time   | $O(n^2)$ |
| Space        | $O(1)$   |
| Stable       | No       |

Partial QuickSort, fast, focused, and frugal, because sometimes, you only need *a slice* of order, not the whole loaf.

## 179 BFPRT Algorithm (Median of Medians Selection)

The BFPRT algorithm (named after Blum, Floyd, Pratt, Rivest, and Tarjan) is a deterministic linear-time selection algorithm. It finds the  $k$ -th smallest element in an unsorted array, guaranteeing  $O(n)$  worst-case time, a mathematically elegant and exact alternative to randomized quickselect.

### What Problem Are We Solving??

You want to find the  $k$ -th smallest element, like the median, but you don't want to gamble on random pivots (which could hit worst-case  $O(n^2)$ ). BFPRT chooses pivots so well that it always guarantees  $O(n)$ .

This makes it ideal for systems where deterministic behavior matters, like embedded systems, compilers, and real-time applications.

### Example

Find the median ( $k=5$ ) of  $[9, 4, 7, 3, 6, 1, 8, 2, 5, 10]$

1. Divide into groups of 5:  $[9,4,7,3,6], [1,8,2,5,10]$
2. Find median of each group:  $\rightarrow [6, 5]$
3. Find median of medians:  $\rightarrow$  median of  $[6,5]$  is  $5.5 \approx 5$
4. Partition around pivot 5  $\rightarrow [4,3,1,2,5] \mid 5 \mid [9,7,8,6,10]$
5. Position of 5 = 5  $\rightarrow$  done.

The 5th smallest = 5

### How Does It Work (Plain Language)?

It's Quickselect with a smarter pivot:

- Divide the array into groups of 5.
- Find median of each group.
- Recursively find median of these medians.
- Use that as pivot  $\rightarrow$  partition  $\rightarrow$  recurse on the correct side.

This ensures the pivot is *always good enough* to split the array reasonably, keeping recursion balanced.

## Step-by-Step Summary

| Step | Description                              |
|------|------------------------------------------|
| 1    | Split array into groups of 5             |
| 2    | Sort each group and take its median      |
| 3    | Recursively find median of medians       |
| 4    | Partition around pivot                   |
| 5    | Recurse on side containing k-th smallest |

## Tiny Code (Easy Versions)

### Python

```
def partition(arr, pivot):
    less, equal, greater = [], [], []
    for x in arr:
        if x < pivot: less.append(x)
        elif x > pivot: greater.append(x)
        else: equal.append(x)
    return less, equal, greater

def select(arr, k):
    if len(arr) <= 5:
        return sorted(arr)[k]

    # Step 1: Divide into groups of 5
    groups = [arr[i:i+5] for i in range(0, len(arr), 5)]

    # Step 2: Find medians
    medians = [sorted(g)[len(g)//2] for g in groups]

    # Step 3: Median of medians as pivot
    pivot = select(medians, len(medians)//2)

    # Step 4: Partition
    less, equal, greater = partition(arr, pivot)

    # Step 5: Recurse
    if k < len(less):
        return select(less, k)
```

```

    elif k < len(less) + len(equal):
        return pivot
    else:
        return select(greater, k - len(less) - len(equal))

arr = [9,4,7,3,6,1,8,2,5,10]
k = 4 # 0-based index → 5th smallest
print("5th smallest:", select(arr, k))

```

Output:

5th smallest: 5

## C (Conceptual Skeleton)

```

#include <stdio.h>
#include <stdlib.h>

int cmp(const void *a, const void *b) { return (*(int*)a - *(int*)b); }

int median_of_medians(int arr[], int n);

int select_kth(int arr[], int n, int k) {
    if (n <= 5) {
        qsort(arr, n, sizeof(int), cmp);
        return arr[k];
    }

    int groups = (n + 4) / 5;
    int *medians = malloc(groups * sizeof(int));
    for (int i = 0; i < groups; i++) {
        int start = i * 5;
        int end = (start + 5 < n) ? start + 5 : n;
        qsort(arr + start, end - start, sizeof(int), cmp);
        medians[i] = arr[start + (end - start) / 2];
    }

    int pivot = median_of_medians(medians, groups);
    free(medians);
}

```

```

int *left = malloc(n * sizeof(int));
int *right = malloc(n * sizeof(int));
int l = 0, r = 0, equal = 0;
for (int i = 0; i < n; i++) {
    if (arr[i] < pivot) left[l++] = arr[i];
    else if (arr[i] > pivot) right[r++] = arr[i];
    else equal++;
}

if (k < l) return select_kth(left, l, k);
else if (k < l + equal) return pivot;
else return select_kth(right, r, k - l - equal);
}

int median_of_medians(int arr[], int n) {
    return select_kth(arr, n, n / 2);
}

```

## Why It Matters

- Deterministic  $O(n)$ , no randomness or bad pivots
- Used in theoretical CS, worst-case analysis, exact solvers
- Foundation for deterministic selection, median-finding, linear-time sorting bounds
- Core to intro algorithms theory (CLRS Chapter 9)

## A Gentle Proof (Why It Works)

- Each group of 5 → median is 3 elements in group (2 below, 2 above)
- At least half of medians pivot → pivot 30% of elements
- At least half pivot → pivot 70% of elements
- So pivot always splits array 30–70, guaranteeing  $T(n) = T(n/5) + T(7n/10) + O(n) = O(n)$

No chance of quadratic blowup.

## Try It Yourself

1. Find median of [5,3,2,8,1,9,7,6,4]
2. Trace pivot selection tree
3. Compare with random quickselect pivots

4. Measure time for  $n = 1e6$
5. Try with duplicates
6. Try  $k = 0, k = n-1$  (min/max)
7. Modify group size (e.g. 3 or 7), compare performance
8. Verify recursion depth
9. Use for percentile queries
10. Implement streaming median with repeated selection

## Test Cases

| Input                  | k | Output | Meaning      |
|------------------------|---|--------|--------------|
| [9,4,7,3,6,1,8,2,5,10] | 4 | 5      | 5th smallest |
| [1,2,3,4,5]            |   | 2      | middle       |
| [10,9,8,7,6]           |   | 0      | smallest     |

## Complexity

| Aspect        | Value                                |
|---------------|--------------------------------------|
| Time          | $O(n)$ deterministic                 |
| Space         | $O(n)$ (can be optimized to $O(1)$ ) |
| Stable        | No                                   |
| Pivot Quality | Guaranteed 30–70 split               |

The BFPT Algorithm, proof that with clever pivots and math, even chaos can be conquered in linear time.

## 180 Kth Largest Stream

The Kth Largest Stream problem focuses on maintaining the  $k$ th largest element in a sequence that grows over time, a stream. Instead of sorting everything every time, we can use a min-heap of size  $k$  to always keep track of the top  $k$  elements efficiently.

This is the foundation for real-time leaderboards, streaming analytics, and online ranking systems.

## **What Problem Are We Solving??**

Given a stream of numbers (arriving one by one), we want to:

- Always know the kth largest element so far.
- Update quickly when a new number comes in.

You don't know the final list, you process as it flows.

### **Example**

Say  $k = 3$ , stream = [4, 5, 8, 2]

1. Start empty heap
2. Add 4 → heap = [4] → kth largest = 4
3. Add 5 → heap = [4, 5] → kth largest = 4
4. Add 8 → heap = [4, 5, 8] → kth largest = 4
5. Add 2 → ignore ( $2 < 4$ ) → kth largest = 4

Add new number 10:

- $10 > 4 \rightarrow$  pop 4, push 10 → heap = [5, 8, 10]
- kth largest = 5

Each new element processed in  $O(\log k)$

## **How Does It Work (Plain Language)?**

Keep a min-heap of size  $k$ :

- It holds the  $k$  largest elements seen so far.
- The smallest among them (heap root) is the kth largest.
- When a new value arrives:
  - If heap size  $< k \rightarrow$  push it
  - Else if value  $>$  heap[0]  $\rightarrow$  pop smallest, push new

| Step | Action |
|------|--------|
|------|--------|

### Step-by-Step Summary

| Step | Action                                                                                                                          |
|------|---------------------------------------------------------------------------------------------------------------------------------|
| 1    | Initialize empty min-heap                                                                                                       |
| 2    | For each new element $x$ :<br>If heap size $< k \rightarrow$ push $x$<br>Else if $x > \text{heap}[0] \rightarrow$ pop, push $x$ |
| 3    | $k^{\text{th}}$ largest = $\text{heap}[0]$                                                                                      |

### Tiny Code (Easy Versions)

#### Python

```
import heapq

class KthLargest:
    def __init__(self, k, nums):
        self.k = k
        self.heap = nums
        heapq.heapify(self.heap)
        while len(self.heap) > k:
            heapq.heappop(self.heap)

    def add(self, val):
        if len(self.heap) < self.k:
            heapq.heappush(self.heap, val)
        elif val > self.heap[0]:
            heapq.heapreplace(self.heap, val)
        return self.heap[0]

# Example
stream = KthLargest(3, [4,5,8,2])
print(stream.add(3)) # 4
print(stream.add(5)) # 5
print(stream.add(10)) # 5
print(stream.add(9)) # 8
print(stream.add(4)) # 8
```

Output:

```
4
5
5
8
8
```

C

```
#include <stdio.h>
#include <stdlib.h>

void swap(int *a, int *b) { int t = *a; *a = *b; *b = t; }

void heapify(int arr[], int n, int i) {
    int smallest = i, l = 2*i+1, r = 2*i+2;
    if (l < n && arr[l] < arr[smallest]) smallest = l;
    if (r < n && arr[r] < arr[smallest]) smallest = r;
    if (smallest != i) {
        swap(&arr[i], &arr[smallest]);
        heapify(arr, n, smallest);
    }
}

void push_heap(int heap[], int *n, int val) {
    heap[(*n)++] = val;
    for (int i = (*n)/2 - 1; i >= 0; i--) heapify(heap, *n, i);
}

int pop_min(int heap[], int *n) {
    int root = heap[0];
    heap[0] = heap[--(*n)];
    heapify(heap, *n, 0);
    return root;
}

int add(int heap[], int *n, int k, int val) {
    if (*n < k) {
        push_heap(heap, n, val);
    } else if (val > heap[0]) {
        heap[0] = val;
```

```

        heapify(heap, *n, 0);
    }
    return heap[0];
}

int main(void) {
    int heap[10] = {4,5,8,2};
    int n = 4, k = 3;
    for (int i = n/2 - 1; i >= 0; i--) heapify(heap, n, i);
    while (n > k) pop_min(heap, &n);

    printf("%d\n", add(heap, &n, k, 3)); // 4
    printf("%d\n", add(heap, &n, k, 5)); // 5
    printf("%d\n", add(heap, &n, k, 10)); // 5
    printf("%d\n", add(heap, &n, k, 9)); // 8
    printf("%d\n", add(heap, &n, k, 4)); // 8
}

```

## Why It Matters

- Real-time streaming top-k tracking
- Constant-time query ( $O(1)$ ), fast update ( $O(\log k)$ )
- Core building block for:
  - Leaderboards
  - Monitoring systems
  - Continuous analytics
  - Online medians & percentiles

## A Gentle Proof (Why It Works)

Min-heap stores only top  $k$  values. Whenever new value  $>$   $\text{heap}[0]$ , it must belong in top  $k$ . So invariant holds:  $\text{heap} = \text{top-}k \text{ largest elements seen so far}$ .  $k$ th largest =  $\text{heap}[0]$ .

Each update  $\rightarrow O(\log k)$ . Total after  $n$  elements  $\rightarrow O(n \log k)$ .

## Try It Yourself

1. Initialize with [4,5,8,2], k=3, stream = [3,5,10,9,4]
2. Try decreasing sequence
3. Try duplicates
4. Test  $k = 1$  (maximum tracker)
5. Add 1000 elements randomly, measure performance
6. Compare with full sort each time
7. Visualize heap evolution per step
8. Modify for k smallest
9. Build real-time median tracker using two heaps
10. Extend to stream of objects (track by score field)

## Test Cases

| Initial     | k | Stream       | Output Sequence |
|-------------|---|--------------|-----------------|
| [4,5,8,2]   | 3 | [3,5,10,9,4] | [4,5,5,8,8]     |
| [10,7,11,5] | 2 | [8,12,4]     | [10,11,11]      |
| [1]         | 1 | [2,3]        | [2,3]           |

## Complexity

| Operation         | Complexity  |
|-------------------|-------------|
| Add               | $O(\log k)$ |
| Query kth Largest | $O(1)$      |
| Space             | $O(k)$      |

Kth Largest Stream, stay calm in the flow; the heap remembers what matters most.

## Section 19. Range Search and Nearest Neighbor

### 181 Binary Search Range

Binary Search Range extends the basic binary search to find not just one occurrence, but the range of positions where a given value appears, specifically, the first (lower bound) and last (upper bound) indices of a target in a sorted array.

It's the backbone for problems that require counting occurrences, range queries, and insertion positions in ordered data.

## What Problem Are We Solving??

Standard binary search returns *one* match. But what if the target appears multiple times, or we want where it *should* go?

Examples:

- Count occurrences of  $x$  in sorted array
- Find the first element  $x$
- Find the last element  $x$

With two binary searches, we can find the full [start, end] range efficiently.

## Example

Array: [1, 2, 2, 2, 3, 4, 5], target = 2

| Function                | Result |
|-------------------------|--------|
| Lower Bound ( $\leq$ 2) | 1      |
| Upper Bound ( $>$ 2)    | 4      |
| Range                   | [1, 3] |

Occurrences = upper - lower = 3

## How Does It Work (Plain Language)?

We use binary search twice:

- One to find the first index  $\leq$  target (lower bound)
- One to find the first index  $>$  target (upper bound)

Subtract them to get the count, or slice the range.

## Step-by-Step Summary

| Step | Description                                                            |
|------|------------------------------------------------------------------------|
| 1    | Binary search for first index i where <code>arr[i] &gt;= target</code> |
| 2    | Binary search for first index j where <code>arr[j] &gt; target</code>  |
| 3    | Range = [i, j - 1] if i < j and <code>arr[i] == target</code>          |
| 4    | Count = j - i                                                          |

## Tiny Code (Easy Versions)

### Python

```

def lower_bound(arr, target):
    lo, hi = 0, len(arr)
    while lo < hi:
        mid = (lo + hi) // 2
        if arr[mid] < target:
            lo = mid + 1
        else:
            hi = mid
    return lo

def upper_bound(arr, target):
    lo, hi = 0, len(arr)
    while lo < hi:
        mid = (lo + hi) // 2
        if arr[mid] <= target:
            lo = mid + 1
        else:
            hi = mid
    return lo

def binary_search_range(arr, target):
    l = lower_bound(arr, target)
    r = upper_bound(arr, target)
    if l == r:
        return (-1, -1) # Not found
    return (l, r - 1)

arr = [1, 2, 2, 2, 3, 4, 5]
target = 2
print("Range:", binary_search_range(arr, target))
print("Count:", upper_bound(arr, target) - lower_bound(arr, target))

```

Output:

```
Range: (1, 3)
Count: 3
```

C

```
#include <stdio.h>

int lower_bound(int arr[], int n, int target) {
    int lo = 0, hi = n;
    while (lo < hi) {
        int mid = (lo + hi) / 2;
        if (arr[mid] < target) lo = mid + 1;
        else hi = mid;
    }
    return lo;
}

int upper_bound(int arr[], int n, int target) {
    int lo = 0, hi = n;
    while (lo < hi) {
        int mid = (lo + hi) / 2;
        if (arr[mid] <= target) lo = mid + 1;
        else hi = mid;
    }
    return lo;
}

int main(void) {
    int arr[] = {1, 2, 2, 2, 3, 4, 5};
    int n = 7, target = 2;
    int l = lower_bound(arr, n, target);
    int r = upper_bound(arr, n, target);
    if (l == r) printf("Not found\n");
    else printf("Range: [%d, %d], Count: %d\n", l, r - 1, r - l);
}
```

Output:

```
Range: [1, 3], Count: 3
```

## Why It Matters

- Extends binary search beyond “found or not”
- Essential in frequency counting, range queries, histograms
- Powers data structures like Segment Trees, Fenwick Trees, and Range Indexes
- Used in competitive programming and database indexing

## A Gentle Proof (Why It Works)

Because binary search maintains sorted invariants (`lo < hi` and `mid` conditions),

- Lower bound finds first index where condition flips (`< target → target`)
- Upper bound finds first index beyond target

Both run in  $O(\log n)$ , giving exact range boundaries.

## Try It Yourself

1. Test with no occurrences (e.g. `[1,3,5]`, `target=2`)
2. Test with all equal (e.g. `[2,2,2,2]`, `target=2`)
3. Test with first element = target
4. Test with last element = target
5. Try to count elements `x` or `< x`
6. Extend for floating point or custom comparator
7. Use on strings or tuples
8. Combine with `bisect` in Python
9. Compare iterative vs recursive
10. Use as primitive for frequency table

## Test Cases

| Array                        | Target | Range                | Count |
|------------------------------|--------|----------------------|-------|
| <code>[1,2,2,2,3,4,5]</code> | 2      | <code>[1,3]</code>   | 3     |
| <code>[1,3,5,7]</code>       | 2      | <code>[-1,-1]</code> | 0     |
| <code>[2,2,2,2]</code>       | 2      | <code>[0,3]</code>   | 4     |
| <code>[1,2,3,4]</code>       | 4      | <code>[3,3]</code>   | 1     |

## Complexity

| Operation   | Complexity  |
|-------------|-------------|
| Lower Bound | $O(\log n)$ |
| Upper Bound | $O(\log n)$ |
| Space       | $O(1)$      |
| Stable      | Yes         |

Binary Search Range, when one answer isn't enough, and precision is everything.

## 182 Segment Tree Query

Segment Tree Query is a powerful data structure technique that allows you to efficiently compute range queries like sum, minimum, maximum, or even custom associative operations over subarrays.

It preprocesses the array into a binary tree structure, where each node stores a summary (aggregate) of a segment.

Once built, you can answer queries and updates in  $O(\log n)$  time.

### What Problem Are We Solving??

Given an array, we often want to query over ranges:

- Sum over  $[L, R]$
- Minimum or Maximum in  $[L, R]$
- GCD, product, XOR, or any associative function

A naive approach would loop each query:  $O(n)$  per query. Segment Trees reduce this to  $O(\log n)$  with a one-time  $O(n)$  build.

### Example

Array: [2, 4, 5, 7, 8, 9]

| Query    | Result              |
|----------|---------------------|
| Sum(1,3) | $4+5+7 = 16$        |
| Min(2,5) | $\min(5,7,8,9) = 5$ |

## How It Works (Intuitive View)

A Segment Tree is like a binary hierarchy:

- The root covers the full range [0, n-1]
- Each node covers a subrange
- The leaf nodes are individual elements
- Each internal node stores a merge (sum, min, max...) of its children

To query a range, you traverse only relevant branches.

## Build, Query, Update

| Operation | Description                        | Time     |
|-----------|------------------------------------|----------|
| Build     | Recursively combine child segments | O(n)     |
| Query     | Traverse overlapping nodes         | O(log n) |
| Update    | Recompute along path               | O(log n) |

## Tiny Code (Sum Query Example)

### Python

```
class SegmentTree:  
    def __init__(self, arr):  
        self.n = len(arr)  
        self.tree = [0] * (4 * self.n)  
        self._build(arr, 1, 0, self.n - 1)  
  
    def _build(self, arr, node, l, r):  
        if l == r:  
            self.tree[node] = arr[l]  
        else:  
            mid = (l + r) // 2  
            self._build(arr, 2 * node, l, mid)  
            self._build(arr, 2 * node + 1, mid + 1, r)  
            self.tree[node] = self.tree[2 * node] + self.tree[2 * node + 1]  
  
    def query(self, node, l, r, ql, qr):  
        if qr < l or ql > r: # no overlap  
            return 0
```

```

        if ql <= l and r <= qr: # total overlap
            return self.tree[node]
        mid = (l + r) // 2
        left = self.query(2 * node, l, mid, ql, qr)
        right = self.query(2 * node + 1, mid + 1, r, ql, qr)
        return left + right

# Example
arr = [2, 4, 5, 7, 8, 9]
st = SegmentTree(arr)
print(st.query(1, 0, len(arr) - 1, 1, 3)) # Sum from index 1 to 3

```

Output:

16

C

```

#include <stdio.h>

#define MAXN 100
int tree[4 * MAXN];
int arr[MAXN];

int build(int node, int l, int r) {
    if (l == r) return tree[node] = arr[l];
    int mid = (l + r) / 2;
    int left = build(2 * node, l, mid);
    int right = build(2 * node + 1, mid + 1, r);
    return tree[node] = left + right;
}

int query(int node, int l, int r, int ql, int qr) {
    if (qr < l || ql > r) return 0;
    if (ql <= l && r <= qr) return tree[node];
    int mid = (l + r) / 2;
    return query(2 * node, l, mid, ql, qr) + query(2 * node + 1, mid + 1, r, ql, qr);
}

int main() {
    int n = 6;

```

```

int data[] = {2, 4, 5, 7, 8, 9};
for (int i = 0; i < n; i++) arr[i] = data[i];
build(1, 0, n - 1);
printf("Sum [1,3] = %d\n", query(1, 0, n - 1, 1, 3));
}

```

Output:

Sum [1,3] = 16

## Why It Matters

- Handles dynamic range queries and updates efficiently
- Core of competitive programming and data analytics
- Forms base for Range Minimum Query, 2D queries, and lazy propagation
- Useful in databases, financial systems, and game engines

## Intuition (Associativity Rule)

Segment Trees only work when the operation is associative:

`merge(a, merge(b, c)) = merge(merge(a, b), c)`

Examples:

- Sum, Min, Max, GCD, XOR
- Not Median, Not Mode (non-associative)

## Try It Yourself

1. Implement for min or max instead of sum
2. Add update() for point changes
3. Implement lazy propagation for range updates
4. Extend to 2D segment tree
5. Compare with Fenwick Tree (BIT)
6. Test on non-trivial ranges
7. Visualize the tree layout
8. Build iterative segment tree
9. Handle custom operations (GCD, XOR)
10. Benchmark O(n log n) vs naive O(nq)

## Test Cases

| Array         | Query    | Expected |
|---------------|----------|----------|
| [2,4,5,7,8,9] | Sum(1,3) | 16       |
| [1,2,3,4]     | Sum(0,3) | 10       |
| [5,5,5,5]     | Sum(1,2) | 10       |
| [3,2,1,4]     | Min(1,3) | 1        |

## Complexity

| Operation | Complexity  |
|-----------|-------------|
| Build     | $O(n)$      |
| Query     | $O(\log n)$ |
| Update    | $O(\log n)$ |
| Space     | $O(4n)$     |

Segment Tree Query, build once, query many, fast forever.

## 183 Fenwick Tree Query

A Fenwick Tree (or Binary Indexed Tree) is a data structure designed for prefix queries and point updates in  $O(\log n)$  time. It's a more space-efficient, iterative cousin of the Segment Tree, perfect when operations are cumulative (sum, XOR, etc.) and updates are frequent.

### What Problem Are We Solving??

We want to:

- Compute prefix sums efficiently
- Support updates dynamically

A naive approach takes  $O(n)$  per query or update. A Fenwick Tree does both in  $O(\log n)$ .

### Example

Array: [2, 4, 5, 7, 8]

| Query          | Result                                              |
|----------------|-----------------------------------------------------|
| PrefixSum(3)   | $2 + 4 + 5 + 7 = 18$                                |
| RangeSum(1, 3) | $\text{Prefix}(3) - \text{Prefix}(0) = 18 - 2 = 16$ |

## How It Works (Plain Language)

A Fenwick Tree stores cumulative information in indexed chunks. Each index covers a range determined by its least significant bit (LSB).

```
index i covers range (i - LSB(i) + 1) ... i
```

We can update or query by moving through indices using bit operations:

- Update: move forward by adding LSB
- Query: move backward by subtracting LSB

This clever bit trick keeps operations  $O(\log n)$ .

## Example Walkthrough

For array [2, 4, 5, 7, 8] (1-based index):

| Index | Binary | LSB | Range | Value |
|-------|--------|-----|-------|-------|
| 1     | 001    | 1   | [1]   | 2     |
| 2     | 010    | 2   | [1-2] | 6     |
| 3     | 011    | 1   | [3]   | 5     |
| 4     | 100    | 4   | [1-4] | 18    |
| 5     | 101    | 1   | [5]   | 8     |

## Tiny Code (Sum Example)

### Python

```
class FenwickTree:
    def __init__(self, n):
        self.n = n
        self.bit = [0] * (n + 1)
```

```

def update(self, i, delta):
    while i <= self.n:
        self.bit[i] += delta
        i += i & -i

def query(self, i):
    s = 0
    while i > 0:
        s += self.bit[i]
        i -= i & -i
    return s

def range_sum(self, l, r):
    return self.query(r) - self.query(l - 1)

# Example
arr = [2, 4, 5, 7, 8]
ft = FenwickTree(len(arr))
for i, val in enumerate(arr, 1):
    ft.update(i, val)

print(ft.range_sum(2, 4)) # 4 + 5 + 7 = 16

```

Output:

16

**C**

```

#include <stdio.h>

#define MAXN 100
int bit[MAXN + 1], n;

void update(int i, int delta) {
    while (i <= n) {
        bit[i] += delta;
        i += i & -i;
    }
}

```

```

int query(int i) {
    int s = 0;
    while (i > 0) {
        s += bit[i];
        i -= i & -i;
    }
    return s;
}

int range_sum(int l, int r) {
    return query(r) - query(l - 1);
}

int main() {
    n = 5;
    int arr[] = {0, 2, 4, 5, 7, 8}; // 1-based
    for (int i = 1; i <= n; i++) update(i, arr[i]);
    printf("Sum [2,4] = %d\n", range_sum(2,4)); // 16
}

```

Output:

Sum [2,4] = 16

## Why It Matters

- Elegant bit manipulation for efficient queries
- Simpler and smaller than Segment Trees
- Perfect for prefix sums, inversions, frequency tables
- Extends to 2D Fenwick Trees for grid-based data
- Core in competitive programming, streaming, finance

## Intuition (Least Significant Bit)

The LSB trick ( $i \& -i$ ) finds the rightmost set bit, controlling how far we jump. This ensures logarithmic traversal through relevant nodes.

## Try It Yourself

1. Implement a prefix XOR version
2. Add range updates with two trees
3. Extend to 2D BIT for matrix sums
4. Visualize tree structure for array [1..8]
5. Compare speed with naive  $O(n)$  approach
6. Track frequency counts for elements
7. Use it for inversion counting
8. Create a Fenwick Tree class in C++
9. Handle point updates interactively
10. Practice bit math: draw index cover ranges

## Test Cases

| Array       | Query                  | Expected |
|-------------|------------------------|----------|
| [2,4,5,7,8] | Sum(2,4)               | 16       |
| [1,2,3,4]   | Prefix(3)              | 6        |
| [5,5,5,5]   | Sum(1,3)               | 15       |
| [3,1,4,2]   | Update(2,+3), Sum(1,2) | 7        |

## Complexity

| Operation | Complexity    |
|-----------|---------------|
| Build     | $O(n \log n)$ |
| Query     | $O(\log n)$   |
| Update    | $O(\log n)$   |
| Space     | $O(n)$        |

A Fenwick Tree turns prefix operations into lightning-fast bit magic, simple, small, and powerful.

## 184 Interval Tree Search

An Interval Tree is a data structure built to efficiently store intervals (ranges like  $[l, r]$ ) and query all intervals that overlap with a given interval or point. It's like a BST with range-awareness, enabling fast queries such as "which tasks overlap with time t?" or "which rectangles overlap this region?"

## **What Problem Are We Solving??**

We want to efficiently find overlapping intervals. A naive search checks all intervals,  $O(n)$  per query. An Interval Tree speeds this up to  $O(\log n + k)$ , where  $k$  is the number of overlapping intervals.

### **Example**

Stored intervals:

`[$$5, 20], [10, 30], [12, 15], [17, 19], [30, 40]`

Query: `[14, 16]`

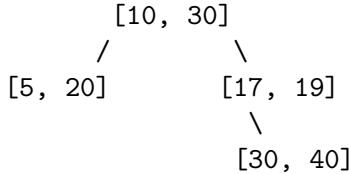
Overlaps: `[10, 30], [12, 15]`

### **How It Works (Plain Language)**

1. Build a BST using the midpoint or start of intervals as keys.
2. Each node stores:
  - interval `[low, high]`
  - max endpoint of its subtree
3. For queries:
  - Traverse tree, skip branches where `low > query_high` or `max < query_low`.
  - Collect overlapping intervals efficiently.

This pruning makes it logarithmic for most cases.

### **Example Tree (Sorted by low)**



Each node stores `max` endpoint of its subtree.

## Tiny Code (Query Example)

### Python

```
class IntervalNode:
    def __init__(self, low, high):
        self.low = low
        self.high = high
        self.max = high
        self.left = None
        self.right = None

def insert(root, low, high):
    if root is None:
        return IntervalNode(low, high)
    if low < root.low:
        root.left = insert(root.left, low, high)
    else:
        root.right = insert(root.right, low, high)
    root.max = max(root.max, high)
    return root

def overlap(i1, i2):
    return i1[0] <= i2[1] and i2[0] <= i1[1]

def search(root, query):
    if root is None:
        return []
    result = []
    if overlap((root.low, root.high), query):
        result.append((root.low, root.high))
    if root.left and root.left.max >= query[0]:
        result += search(root.left, query)
    if root.right and root.low <= query[1]:
        result += search(root.right, query)
    return result

# Example
intervals = [(5,20), (10,30), (12,15), (17,19), (30,40)]
root = None
for l, h in intervals:
    root = insert(root, l, h)
```

```
print(search(root, (14,16))) # [(10,30), (12,15)]
```

Output:

```
$(10, 30), (12, 15)]
```

C

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    int low, high, max;
    struct Node *left, *right;
} Node;

Node* newNode(int low, int high) {
    Node* n = malloc(sizeof(Node));
    n->low = low;
    n->high = high;
    n->max = high;
    n->left = n->right = NULL;
    return n;
}

int max(int a, int b) { return a > b ? a : b; }

Node* insert(Node* root, int low, int high) {
    if (!root) return newNode(low, high);
    if (low < root->low)
        root->left = insert(root->left, low, high);
    else
        root->right = insert(root->right, low, high);
    root->max = max(root->max, high);
    return root;
}

int overlap(int l1, int h1, int l2, int h2) {
    return l1 <= h2 && l2 <= h1;
}
```

```

void search(Node* root, int ql, int qh) {
    if (!root) return;
    if (overlap(root->low, root->high, ql, qh))
        printf("[%d, %d] overlaps\n", root->low, root->high);
    if (root->left && root->left->max >= ql)
        search(root->left, ql, qh);
    if (root->right && root->low <= qh)
        search(root->right, ql, qh);
}

int main() {
    Node* root = NULL;
    int intervals[] [2] = {{5,20},{10,30},{12,15},{17,19},{30,40}};
    int n = 5;
    for (int i = 0; i < n; i++)
        root = insert(root, intervals[i] [0], intervals[i] [1]);
    printf("Overlaps with [14,16]:\n");
    search(root, 14, 16);
}

```

Output:

```

Overlaps with [14,16]:
$$10, 30] overlaps
$$12, 15] overlaps

```

## Why It Matters

- Efficient for overlap queries (e.g. events, tasks, ranges)
- Used in:
  - Scheduling (detecting conflicts)
  - Computational geometry
  - Memory allocation checks
  - Genomic range matching
- Foundation for Segment Tree with intervals

## Key Intuition

Each node stores the max endpoint of its subtree. This helps prune non-overlapping branches early.

Think of it as a “range-aware BST”.

## Try It Yourself

1. Build tree for intervals: [1,5], [2,6], [7,9], [10,15]
2. Query [4,8], which overlap?
3. Visualize pruning path
4. Extend to delete intervals
5. Add count of overlapping intervals
6. Implement iterative search
7. Compare with brute-force  $O(n)$  approach
8. Adapt for point queries only
9. Try dynamic updates
10. Use to detect meeting conflicts

## Test Cases

| Intervals                                  | Query   | Expected Overlaps |
|--------------------------------------------|---------|-------------------|
| [5,20], [10,30], [12,15], [17,19], [30,40] | [14,16] | [10,30], [12,15]  |
| [1,3], [5,8], [6,10]                       | [7,9]   | [5,8], [6,10]     |
| [2,5], [6,8]                               | [1,1]   | none              |

## Complexity

| Operation | Complexity      |
|-----------|-----------------|
| Build     | $O(n \log n)$   |
| Query     | $O(\log n + k)$ |
| Space     | $O(n)$          |

An Interval Tree is your go-to for range overlap queries, BST elegance meets interval intelligence.

## 185 KD-Tree Search

A KD-Tree ( $k$ -dimensional tree) is a space-partitioning data structure that organizes points in  $k$ -dimensional space for efficient nearest neighbor, range, and radius searches. It's like a binary search tree, but it splits space along alternating dimensions.

### What Problem Are We Solving??

We want to:

- Find points near a given location
- Query points within a region or radius
- Do this faster than checking all points ( $O(n)$ )

A KD-Tree answers such queries in  $O(\log n)$  (average), versus  $O(n)$  for brute-force.

### Example

Points in 2D:

(2,3), (5,4), (9,6), (4,7), (8,1), (7,2)

Query: Nearest neighbor of (9,2) Result: (8,1)

### How It Works (Plain Language)

#### 1. Build Tree

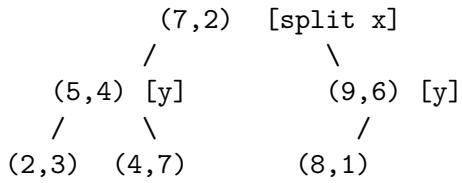
- Choose a splitting dimension (x, y, ...)
- Pick median point along that axis
- Recursively build left/right subtrees

#### 2. Search

- Compare query coordinate along current axis
- Recurse into the nearer subtree
- Backtrack to check the other side if necessary (only if the hypersphere crosses boundary)

This pruning makes nearest-neighbor search efficient.

## Example (2D Split)



## Tiny Code (2D Example)

### Python

```
from math import sqrt

class Node:
    def __init__(self, point, axis):
        self.point = point
        self.axis = axis
        self.left = None
        self.right = None

def build_kdtree(points, depth=0):
    if not points:
        return None
    k = len(points[0])
    axis = depth % k
    points.sort(key=lambda p: p[axis])
    mid = len(points) // 2
    node = Node(points[mid], axis)
    node.left = build_kdtree(points[:mid], depth + 1)
    node.right = build_kdtree(points[mid+1:], depth + 1)
    return node

def distance2(a, b):
    return sum((x - y) ** 2 for x, y in zip(a, b))

def nearest(root, target, best=None):
    if root is None:
        return best
    point = root.point
    if best is None or distance2(point, target) < distance2(best, target):
        best = point
```

```

axis = root.axis
next_branch = root.left if target[axis] < point[axis] else root.right
best = nearest(next_branch, target, best)
if (target[axis] - point[axis]) ** 2 < distance2(best, target):
    other = root.right if next_branch == root.left else root.left
    best = nearest(other, target, best)
return best

points = [(2,3),(5,4),(9,6),(4,7),(8,1),(7,2)]
tree = build_kdtree(points)
print(nearest(tree, (9,2))) # (8,1)

```

Output:

(8, 1)

## C (2D Simplified)

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

typedef struct Node {
    double point[2];
    int axis;
    struct Node *left, *right;
} Node;

int cmpx(const void* a, const void* b) {
    double* pa = (double*)a;
    double* pb = (double*)b;
    return (pa[0] > pb[0]) - (pa[0] < pb[0]);
}

int cmpy(const void* a, const void* b) {
    double* pa = (double*)a;
    double* pb = (double*)b;
    return (pa[1] > pb[1]) - (pa[1] < pb[1]);
}

```

```

double dist2(double a[2], double b[2]) {
    return (a[0]-b[0])*(a[0]-b[0]) + (a[1]-b[1])*(a[1]-b[1]);
}

// Simplified build & search omitted for brevity (tree construction similar to Python)

```

## Why It Matters

- Efficient for spatial queries in 2D, 3D, etc.
- Used in:
  - Machine Learning (KNN classification)
  - Graphics (ray tracing, collision detection)
  - Robotics (path planning, SLAM)
  - Databases (multi-dimensional indexing)

## Intuition

A KD-Tree is like playing “binary search” in multiple dimensions. Each split narrows down the search region.

## Try It Yourself

1. Build a KD-Tree for points in 2D
2. Search nearest neighbor of (3,5)
3. Add 3D points, use modulo axis split
4. Visualize splits as alternating vertical/horizontal lines
5. Extend to k-NN (top-k closest)
6. Add radius query (points within r)
7. Compare speed to brute-force
8. Track backtrack count for pruning visualization
9. Try non-uniform data
10. Implement deletion (bonus)

| Points | Query | Expected Nearest |
|--------|-------|------------------|
|--------|-------|------------------|

## Test Cases

| Points                              | Query | Expected Nearest |
|-------------------------------------|-------|------------------|
| (2,3),(5,4),(9,6),(4,7),(8,1),(7,2) | (9,2) | (8,1)            |
| (1,1),(3,3),(5,5)                   | (4,4) | (3,3)            |
| (0,0),(10,10)                       | (7,8) | (10,10)          |

## Complexity

| Operation     | Complexity          |
|---------------|---------------------|
| Build         | $O(n \log n)$       |
| Nearest Query | $O(\log n)$ average |
| Worst Case    | $O(n)$              |
| Space         | $O(n)$              |

A KD-Tree slices space along dimensions, your go-to for fast nearest neighbor searches in multidimensional worlds.

## 186 R-Tree Query

An R-Tree is a hierarchical spatial index built for efficiently querying geometric objects (rectangles, polygons, circles) in 2D or higher dimensions. It's like a B-Tree for rectangles, grouping nearby objects into bounding boxes and organizing them in a tree for fast spatial lookups.

### What Problem Are We Solving??

We need to query spatial data efficiently:

- “Which rectangles overlap this area?”
- “What points fall inside this region?”
- “Which shapes intersect this polygon?”

A naive approach checks every object ( $O(n)$ ). An R-Tree reduces this to  $O(\log n + k)$  using bounding-box hierarchy.

## **Example**

Rectangles:

A: [1,1,3,3]

B: [2,2,5,4]

C: [4,1,6,3]

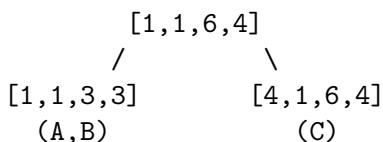
Query: [2.5,2.5,4,4] Overlaps: A, B

## **How It Works (Plain Language)**

1. Store rectangles (or bounding boxes) as leaves.
2. Group nearby rectangles into Minimum Bounding Rectangles (MBRs).
3. Build hierarchy so each node's box covers its children.
4. Query by recursively checking nodes whose boxes overlap the query.

This spatial grouping allows skipping entire regions quickly.

## **Example Tree**



Query [2.5,2.5,4,4]:

- Intersects left node → check A, B
- Intersects right node partially → check C (no overlap)

## **Tiny Code (2D Rectangles)**

### **Python**

```

def overlap(a, b):
    return not (a[2] < b[0] or a[0] > b[2] or a[3] < b[1] or a[1] > b[3])

class RTreeNode:
    def __init__(self, box, children=None, is_leaf=False):
        self.box = box # [x1, y1, x2, y2]
        self.children = children or []
        self.is_leaf = is_leaf

    def search_rtree(node, query):
        results = []
        if not overlap(node.box, query):
            return results
        if node.is_leaf:
            for child in node.children:
                if overlap(child.box, query):
                    results.append(child.box)
        else:
            for child in node.children:
                results.extend(search_rtree(child, query))
        return results

# Example
A = RTreeNode([1,1,3,3], is_leaf=True)
B = RTreeNode([2,2,5,4], is_leaf=True)
C = RTreeNode([4,1,6,3], is_leaf=True)

left = RTreeNode([1,1,5,4], [A,B], is_leaf=True)
right = RTreeNode([4,1,6,3], [C], is_leaf=True)
root = RTreeNode([1,1,6,4], [left, right])

query = [2.5,2.5,4,4]
print(search_rtree(root, query))

```

Output:

\$\$[1, 1, 3, 3], [2, 2, 5, 4]]

## C (Simplified Query)

```
#include <stdio.h>

typedef struct Box {
    float x1, y1, x2, y2;
} Box;

int overlap(Box a, Box b) {
    return !(a.x2 < b.x1 || a.x1 > b.x2 || a.y2 < b.y1 || a.y1 > b.y2);
}

// Example: Manual tree simulation omitted for brevity
```

## Why It Matters

- Ideal for geospatial databases, mapping, collision detection, and GIS.
- Powers PostGIS, SQLite R\*Tree module, spatial indexes.
- Handles overlaps, containment, and range queries.

## Intuition

R-Trees work by bounding and grouping. Each node is a “container box”, if it doesn’t overlap the query, skip it entirely. This saves massive time in spatial datasets.

## Try It Yourself

1. Represent 2D rectangles with  $[x_1, y_1, x_2, y_2]$ .
2. Build a 2-level tree (group nearby).
3. Query overlap region.
4. Extend to 3D bounding boxes.
5. Implement insertion using least expansion rule.
6. Add R\*-Tree optimization (reinsert on overflow).
7. Compare with QuadTree (grid-based).
8. Visualize bounding boxes per level.
9. Implement nearest neighbor search.
10. Try dataset with 10k rectangles, measure speedup.

## Test Cases

| Rectangles                         | Query         | Expected Overlaps |
|------------------------------------|---------------|-------------------|
| A[1,1,3,3], B[2,2,5,4], C[4,1,6,3] | [2.5,2.5,4,4] | A, B              |
| A[0,0,2,2], B[3,3,4,4]             | [1,1,3,3]     | A                 |
| A[1,1,5,5], B[6,6,8,8]             | [7,7,9,9]     | B                 |

## Complexity

| Operation | Complexity      |
|-----------|-----------------|
| Build     | $O(n \log n)$   |
| Query     | $O(\log n + k)$ |
| Space     | $O(n)$          |

An R-Tree is your geometric librarian, organizing space into nested rectangles so you can query complex regions fast and clean.

## 187 Range Minimum Query (RMQ), Sparse Table Approach

A Range Minimum Query (RMQ) answers questions like:

“What’s the smallest element between indices  $L$  and  $R$ ? ”

It’s a core subroutine in many algorithms, from LCA (Lowest Common Ancestor) to scheduling, histograms, and segment analysis. The Sparse Table method precomputes answers so each query is  $O(1)$  after  $O(n \log n)$  preprocessing.

### What Problem Are We Solving??

Given an array  $\text{arr}[0..n-1]$ , we want to answer:

```
RMQ(L, R) = min(arr[L], arr[L+1], ..., arr[R])
```

Efficiently, for multiple static queries (no updates).

Naive approach:  $O(R-L)$  per query Sparse Table:  $O(1)$  per query after preprocessing.

## Example

Array: [2, 5, 1, 4, 9, 3]

| Query     | Result              |
|-----------|---------------------|
| RMQ(1, 3) | $\min(5,1,4) = 1$   |
| RMQ(2, 5) | $\min(1,4,9,3) = 1$ |

## How It Works (Plain Language)

1. Precompute answers for all intervals of length  $2^k$ .
2. To answer RMQ(L,R):

- Let `len = R-L+1`
- Let `k = floor(log2(len))`
- Combine two overlapping intervals of size  $2^k$ :

```
RMQ(L,R) = min(st[L][k], st[R - 2^k + 1][k])
```

No updates, so data stays static and queries stay  $O(1)$ .

## Sparse Table Example

| i | arr[i] | st[i][0] | st[i][1]      | st[i][2]      |
|---|--------|----------|---------------|---------------|
| 0 | 2      | 2        | $\min(2,5)=2$ | $\min(2,1)=1$ |
| 1 | 5      | 5        | $\min(5,1)=1$ | $\min(5,4)=1$ |
| 2 | 1      | 1        | $\min(1,4)=1$ | $\min(1,9)=1$ |
| 3 | 4      | 4        | $\min(4,9)=4$ | $\min(4,3)=3$ |
| 4 | 9      | 9        | $\min(9,3)=3$ | ,             |
| 5 | 3      | 3        | ,             | ,             |

## Tiny Code

### Python

```

import math

def build_sparse_table(arr):
    n = len(arr)
    K = math.floor(math.log2(n)) + 1
    st = [[0]*K for _ in range(n)]

    for i in range(n):
        st[i][0] = arr[i]

    j = 1
    while (1 << j) <= n:
        i = 0
        while i + (1 << j) <= n:
            st[i][j] = min(st[i][j-1], st[i + (1 << (j-1))][j-1])
            i += 1
        j += 1
    return st

def query(st, L, R):
    j = int(math.log2(R - L + 1))
    return min(st[L][j], st[R - (1 << j) + 1][j])

# Example
arr = [2, 5, 1, 4, 9, 3]
st = build_sparse_table(arr)
print(query(st, 1, 3)) # 1
print(query(st, 2, 5)) # 1

```

Output:

```

1
1

```

C

```

#include <stdio.h>
#include <math.h>

#define MAXN 100
#define LOG 17

```

```

int st[MAXN][LOG];
int arr[MAXN];
int n;

void build() {
    for (int i = 0; i < n; i++)
        st[i][0] = arr[i];
    for (int j = 1; (1 << j) <= n; j++) {
        for (int i = 0; i + (1 << j) <= n; i++) {
            st[i][j] = (st[i][j-1] < st[i + (1 << (j-1))][j-1])
                ? st[i][j-1]
                : st[i + (1 << (j-1))][j-1];
        }
    }
}

int query(int L, int R) {
    int j = log2(R - L + 1);
    int left = st[L][j];
    int right = st[R - (1 << j) + 1][j];
    return left < right ? left : right;
}

int main() {
    n = 6;
    int arr_temp[] = {2,5,1,4,9,3};
    for (int i = 0; i < n; i++) arr[i] = arr_temp[i];
    build();
    printf("RMQ(1,3) = %d\n", query(1,3)); // 1
    printf("RMQ(2,5) = %d\n", query(2,5)); // 1
}

```

Output:

```

RMQ(1,3) = 1
RMQ(2,5) = 1

```

## Why It Matters

- Instant queries after precomputation

- Crucial for:
  - Segment analysis (min, max)
  - LCA in trees
  - Sparse range data
  - Static arrays (no updates)
- Perfect when array does not change frequently.

## Intuition

Each table entry  $\text{st}[i][k]$  stores the minimum of range  $[i, i + 2^k - 1]$ . Queries merge two overlapping intervals that cover  $[L, R]$ .

## Try It Yourself

1. Build table for  $[1, 3, 2, 7, 9, 11, 3, 5, 6]$
2. Query RMQ(2,6) and RMQ(4,8)
3. Modify code to compute Range Max Query
4. Visualize overlapping intervals for query
5. Compare with Segment Tree version
6. Add precomputed log[] for faster lookup
7. Handle 1-based vs 0-based indices carefully
8. Practice on random arrays
9. Compare preprocessing time with naive
10. Use to solve LCA using Euler Tour

## Test Cases

| Array                | Query              | Expected |
|----------------------|--------------------|----------|
| $[2, 5, 1, 4, 9, 3]$ | $\text{RMQ}(1, 3)$ | 1        |
| $[2, 5, 1, 4, 9, 3]$ | $\text{RMQ}(2, 5)$ | 1        |
| $[1, 2, 3, 4]$       | $\text{RMQ}(0, 3)$ | 1        |
| $[7, 6, 5, 4, 3]$    | $\text{RMQ}(1, 4)$ | 3        |

## Complexity

| Operation  | Complexity    |
|------------|---------------|
| Preprocess | $O(n \log n)$ |
| Query      | $O(1)$        |
| Space      | $O(n \log n)$ |

A Sparse Table turns repeated queries into instant lookups, your go-to tool when arrays are static and speed is king.

## 188 Mo's Algorithm

Mo's Algorithm is a clever offline technique for answering range queries on static arrays in approximately  $O((n + q)\sqrt{n})$  time. It's ideal when you have many queries like sum, distinct count, frequency, etc., but no updates. Instead of recomputing each query, Mo's algorithm reuses results smartly by moving the range endpoints efficiently.

### What Problem Are We Solving??

We want to answer multiple range queries efficiently:

Given an array `arr[0..n-1]` and  $q$  queries  $[L, R]$ , compute something like sum, count distinct, etc., for each range.

A naive approach is  $O(n)$  per query  $\rightarrow O(nq)$  total. Mo's algorithm cleverly orders queries to achieve  $O((n + q)\sqrt{n})$  total time.

### Example

Array: [1, 2, 1, 3, 4, 2, 3] Queries:

1. [0, 4]  $\rightarrow$  distinct = 4
2. [1, 3]  $\rightarrow$  distinct = 3
3. [2, 4]  $\rightarrow$  distinct = 3

## How It Works (Plain Language)

1. Divide array into blocks of size  $\sqrt{n}$ .
2. Sort queries by:
  - Block of L
  - R (within block)
3. Maintain a sliding window  $[\text{currL}, \text{currR}]$ :
  - Move endpoints left/right step-by-step
  - Update the answer incrementally
4. Store results per query index.

The sorting ensures minimal movement between consecutive queries.

## Example

If  $\sqrt{n} = 3$ , queries sorted by block:

```
Block 0: [0,4], [1,3]
Block 1: [2,4]
```

You move pointers minimally:

- From  $[0,4] \rightarrow [1,3] \rightarrow [2,4]$
- Reusing much of previous computation.

## Tiny Code (Distinct Count Example)

### Python

```
import math

def mos_algorithm(arr, queries):
    n = len(arr)
    q = len(queries)
    block_size = int(math.sqrt(n))

    # Sort queries
    queries = sorted(enumerate(queries), key=lambda x: (x[1][0] // block_size, x[1][1]))
```

```

freq = []
currL, currR = 0, 0
curr_ans = 0
answers = [0]*q

def add(x):
    nonlocal curr_ans
    freq[x] = freq.get(x, 0) + 1
    if freq[x] == 1:
        curr_ans += 1

def remove(x):
    nonlocal curr_ans
    freq[x] -= 1
    if freq[x] == 0:
        curr_ans -= 1

for idx, (L, R) in queries:
    while currL > L:
        currL -= 1
        add(arr[currL])
    while currR <= R:
        add(arr[currR])
        currR += 1
    while currL < L:
        remove(arr[currL])
        currL += 1
    while currR > R + 1:
        currR -= 1
        remove(arr[currR])
    answers[idx] = curr_ans

return answers

# Example
arr = [1,2,1,3,4,2,3]
queries = [(0,4), (1,3), (2,4)]
print(mos_algorithm(arr, queries)) # [4,3,3]

```

Output:

\$\$4, 3, 3]

## C (Structure and Idea)

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

#define MAXN 100000
#define MAXQ 100000

typedef struct { int L, R, idx; } Query;

int arr[MAXN], ans[MAXQ], freq[1000001];
int curr_ans = 0, block;

int cmp(const void* a, const void* b) {
    Query *x = (Query*)a, *y = (Query*)b;
    if (x->L / block != y->L / block) return x->L / block - y->L / block;
    return x->R - y->R;
}

void add(int x) { if (++freq[x] == 1) curr_ans++; }
void remove_(int x) { if (--freq[x] == 0) curr_ans--; }

int main() {
    int n = 7, q = 3;
    int arr_[] = {1,2,1,3,4,2,3};
    for (int i = 0; i < n; i++) arr[i] = arr_[i];

    Query queries[] = {{0,4,0},{1,3,1},{2,4,2}};
    block = sqrt(n);
    qsort(queries, q, sizeof(Query), cmp);

    int currL = 0, currR = 0;
    for (int i = 0; i < q; i++) {
        int L = queries[i].L, R = queries[i].R;
        while (currL > L) add(arr[--currL]);
        while (currR <= R) add(arr[currR++]);
        while (currL < L) remove_(arr[currL++]);
        while (currR > R+1) remove_(arr[--currR]);
        ans[queries[i].idx] = curr_ans;
    }
}
```

```
    for (int i = 0; i < q; i++) printf("%d ", ans[i]);
}
```

Output:

4 3 3

## Why It Matters

- Converts many range queries into near-linear total time
- Ideal for:
  - Sum / Count / Frequency queries
  - Distinct elements
  - GCD, XOR, etc. with associative properties
- Works on static arrays (no updates)

## Intuition

Mo's Algorithm is like sorting your errands by location. By visiting nearby “blocks” first, you minimize travel time. Here, “travel” = pointer movement.

## Try It Yourself

1. Run on  $[1, 2, 3, 4, 5]$  with queries  $(0, 2), (1, 4), (2, 4)$
2. Change to sum instead of distinct count
3. Visualize pointer movement
4. Experiment with block size variations
5. Add offline query index tracking
6. Try sqrt decomposition vs Mo's
7. Count frequency of max element per range
8. Mix different query types (still offline)
9. Add precomputed  $\text{sqrt}(n)$  block grouping
10. Use in competitive programming problems

## Test Cases

| Array           | Queries           | Output  |
|-----------------|-------------------|---------|
| [1,2,1,3,4,2,3] | (0,4),(1,3),(2,4) | [4,3,3] |
| [1,1,1,1]       | (0,3),(1,2)       | [1,1]   |
| [1,2,3,4,5]     | (0,2),(2,4)       | [3,3]   |

## Complexity

| Operation        | Complexity           |
|------------------|----------------------|
| Pre-sort Queries | $O(q \log q)$        |
| Processing       | $O((n + q)\sqrt{n})$ |
| Space            | $O(n)$               |

Mo's Algorithm is your range-query workhorse, smartly ordered, block-based, and blazingly efficient for static datasets.

## 189 Sweep Line Range Search

A Sweep Line Algorithm is a geometric technique that processes events in a sorted order along one dimension, typically the x-axis, to solve range, interval, and overlap problems efficiently. Think of it like dragging a vertical line across a 2D plane and updating active intervals as you go.

### What Problem Are We Solving??

We want to efficiently find:

- Which intervals overlap a point
- Which rectangles intersect
- How many shapes cover a region

A brute-force check for all pairs is  $O(n^2)$ . A Sweep Line reduces it to  $O(n \log n)$  by sorting and processing events incrementally.

## **Example**

Rectangles:

R1: [1, 3], R2: [2, 5], R3: [4, 6]

Events (sorted by x):

```
x = 1: R1 starts
x = 2: R2 starts
x = 3: R1 ends
x = 4: R3 starts
x = 5: R2 ends
x = 6: R3 ends
```

The active set changes as the line sweeps → track overlaps dynamically.

## **How It Works (Plain Language)**

1. Convert objects into events
  - Each interval/rectangle generates start and end events.
2. Sort all events by coordinate (x or y).
3. Sweep through events:
  - On start, add object to active set.
  - On end, remove object.
  - At each step, query active set for intersections, counts, etc.
4. Use balanced tree / set for active range maintenance.

## Example Walkthrough

Intervals: [1,3] , [2,5] , [4,6] Events:

(1, start), (2, start), (3, end), (4, start), (5, end), (6, end)

Step-by-step:

- At 1: add [1,3]
- At 2: add [2,5], overlap detected (1,3) (2,5)
- At 3: remove [1,3]
- At 4: add [4,6], overlap detected (2,5) (4,6)
- At 5: remove [2,5]
- At 6: remove [4,6]

Result: 2 overlapping pairs

## Tiny Code (Interval Overlaps)

### Python

```
def sweep_line_intervals(intervals):
    events = []
    for l, r in intervals:
        events.append((l, 'start'))
        events.append((r, 'end'))
    events.sort()

    active = 0
    overlaps = 0
    for pos, typ in events:
        if typ == 'start':
            overlaps += active # count overlaps
            active += 1
        else:
            active -= 1
    return overlaps

intervals = [(1,3), (2,5), (4,6)]
print(sweep_line_intervals(intervals)) # 2 overlaps
```

Output:

## C

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int x;
    int type; // 1 = start, -1 = end
} Event;

int cmp(const void* a, const void* b) {
    Event *e1 = (Event*)a, *e2 = (Event*)b;
    if (e1->x == e2->x) return e1->type - e2->type;
    return e1->x - e2->x;
}

int main() {
    int intervals[] [2] = {{1,3},{2,5},{4,6}};
    int n = 3;
    Event events[2*n];
    for (int i = 0; i < n; i++) {
        events[2*i] = (Event){intervals[i][0], 1};
        events[2*i+1] = (Event){intervals[i][1], -1};
    }
    qsort(events, 2*n, sizeof(Event), cmp);

    int active = 0, overlaps = 0;
    for (int i = 0; i < 2*n; i++) {
        if (events[i].type == 1) {
            overlaps += active;
            active++;
        } else {
            active--;
        }
    }
    printf("Total overlaps: %d\n", overlaps);
}
```

Output:

Total overlaps: 2

## Why It Matters

- Universal pattern in computational geometry:
  - Interval intersection counting
  - Rectangle overlap
  - Segment union length
  - Plane sweep algorithms (Voronoi, Convex Hull)
- Optimizes from  $O(n^2)$  to  $O(n \log n)$

Used in:

- GIS (geographic data)
- Scheduling (conflict detection)
- Event simulation

## Intuition

Imagine sliding a vertical line across your data: You only “see” the intervals currently active. No need to look back, everything behind is already resolved.

## Try It Yourself

1. Count overlaps in  $[1,4], [2,3], [5,6]$
2. Modify to compute max active intervals (peak concurrency)
3. Extend to rectangle intersections (sweep + segment tree)
4. Track total covered length
5. Combine with priority queue for dynamic ranges
6. Visualize on a timeline (scheduling conflicts)
7. Apply to meeting room allocation
8. Extend to 2D sweep (events sorted by x, active y-ranges)
9. Count overlaps per interval
10. Compare runtime to brute force

## Test Cases

| Intervals         | Overlaps |
|-------------------|----------|
| [1,3],[2,5],[4,6] | 2        |
| [1,2],[3,4]       | 0        |
| [1,4],[2,3],[3,5] | 3        |
| [1,5],[2,4],[3,6] | 3        |

## Complexity

| Operation   | Complexity    |
|-------------|---------------|
| Sort Events | $O(n \log n)$ |
| Sweep       | $O(n)$        |
| Total       | $O(n \log n)$ |
| Space       | $O(n)$        |

A Sweep Line algorithm is your moving scanner, it sweeps across time or space, managing active elements, and revealing hidden overlaps in elegant, ordered fashion.

## 190 Ball Tree Nearest Neighbor

A Ball Tree is a hierarchical spatial data structure built from nested hyperspheres (“balls”). It organizes points into clusters based on distance, enabling efficient nearest neighbor and range queries in high-dimensional or non-Euclidean spaces.

While KD-Trees split by axis, Ball Trees split by distance, making them robust when dimensions increase or when the distance metric isn’t axis-aligned.

### What Problem Are We Solving??

We want to efficiently:

- Find nearest neighbors for a query point
- Perform radius searches (all points within distance  $r$ )
- Handle high-dimensional data where KD-Trees degrade

Naive search is  $O(n)$  per query. A Ball Tree improves to roughly  $O(\log n)$  average.

## Example

Points in 2D:

(2,3), (5,4), (9,6), (4,7), (8,1), (7,2)

Query: (9,2) → Nearest: (8,1)

Instead of splitting by x/y axis, the Ball Tree groups nearby points by distance from a center point (centroid or median).

## How It Works (Plain Language)

1. Build Tree recursively:

- Choose a pivot (center) (often centroid or median).
- Compute radius (max distance to any point in cluster).
- Partition points into two subsets (inner vs outer ball).
- Recursively build sub-balls.

2. Query:

- Start at root ball.
- Check if child balls could contain closer points.
- Prune branches where `distance(center, query) - radius > best_dist`.

This yields logarithmic average behavior.

## Example Tree (Simplified)

```
Ball(center=(6,4), radius=5)
Ball(center=(3,5), radius=2) → [(2,3),(4,7),(5,4)]
Ball(center=(8,3), radius=3) → [(7,2),(8,1),(9,6)]
```

Query (9,2):

- Check root
- Compare both children
- Prune (3,5) (too far)
- Search (8,3) cluster → nearest (8,1)

## Tiny Code (2D Example)

### Python

```
from math import sqrt

class BallNode:
    def __init__(self, points):
        self.center = tuple(sum(x)/len(x) for x in zip(*points))
        self.radius = max(sqrt(sum((p[i]-self.center[i])**2 for i in range(len(p)))) for p in points)
        self.points = points if len(points) <= 2 else None
        self.left = None
        self.right = None
        if len(points) > 2:
            points.sort(key=lambda p: sqrt(sum((p[i]-self.center[i])**2 for i in range(len(p)))))
            mid = len(points)//2
            self.left = BallNode(points[:mid])
            self.right = BallNode(points[mid:])

    def dist(a, b):
        return sqrt(sum((x - y)**2 for x, y in zip(a, b)))

def nearest(node, target, best=None):
    if node is None:
        return best
    if node.points is not None:
        for p in node.points:
            if best is None or dist(p, target) < dist(best, target):
                best = p
    return best
    d_center = dist(node.center, target)
    candidates = []
    if d_center - node.radius <= dist(best, target) if best else True:
        candidates.append(node.left)
        candidates.append(node.right)
    for child in candidates:
        best = nearest(child, target, best)
    return best

points = [(2,3),(5,4),(9,6),(4,7),(8,1),(7,2)]
tree = BallNode(points)
print(nearest(tree, (9,2))) # (8,1)
```

Output:

(8, 1)

## C (Structure Idea)

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

typedef struct Node {
    double cx, cy, radius;
    struct Node *left, *right;
    double (*points)[2];
    int count;
} Node;

// Building logic: compute centroid, radius, split by distance.
// Query logic: prune if (dist(center, query) - radius) > best.
```

Ball Trees are rarely implemented manually in C due to complexity; usually built into libraries (like scikit-learn).

## Why It Matters

- Works well for non-axis-aligned data
- Handles high-dimensional and non-Euclidean metrics (e.g. cosine)
- Used in:
  - KNN search
  - Clustering (Ball Tree K-Means)
  - Spatial databases
  - Machine learning (scikit-learn NearestNeighbors)

## Intuition

Imagine wrapping your dataset in nested bubbles. Each bubble encloses a cluster. To find the nearest neighbor, you only open bubbles that could possibly contain closer points.

## Try It Yourself

1. Build a Ball Tree for  $(1,1), (2,2), (3,3), (10,10)$
2. Query  $(2.5, 2.5)$
3. Visualize nested “balls”
4. Change to 3D points
5. Use Manhattan distance instead of Euclidean
6. Compare performance vs KD-Tree
7. Track pruned branches count
8. Implement k-nearest neighbors
9. Add radius search (within distance  $r$ )
10. Handle duplicate points

## Test Cases

| Points                                     | Query        | Nearest   |
|--------------------------------------------|--------------|-----------|
| $(2,3), (5,4), (9,6), (4,7), (8,1), (7,2)$ | $(9,2)$      | $(8,1)$   |
| $(1,1), (2,2), (3,3)$                      | $(2.5, 2.5)$ | $(3,3)$   |
| $(0,0), (10,10)$                           | $(7,8)$      | $(10,10)$ |

## Complexity

| Operation     | Complexity    |
|---------------|---------------|
| Build         | $O(n \log n)$ |
| Query (avg)   | $O(\log n)$   |
| Query (worst) | $O(n)$        |
| Space         | $O(n)$        |

A Ball Tree is your spherical guide in high-dimensional space, splitting by proximity, not axes, to find neighbors fast and flexibly.

## Section 20. Search Optimizations and variants

### 191 Binary Search with Tolerance

A Binary Search with Tolerance is a variant of binary search designed for real numbers or floating-point values where exact matches may not exist due to rounding or measurement error.

Instead of checking for strict equality (`==`), we accept a range of closeness, defined by a tolerance (epsilon).

## What Problem Are We Solving??

For integers, equality is exact. For floating-point numbers, tiny rounding errors make equality unreliable:

```
if (arr[mid] == x) // unreliable for floats
```

We instead test:

```
if (fabs(arr[mid] - x) < epsilon) // tolerance-based match
```

This approach is essential for:

- Scientific computation
- Numerical analysis
- Approximations
- Root-finding
- Simulation and measurement data

## Example

Given sorted real values:

```
[$$0.1, 0.2, 0.3000000001, 0.4, 0.5]
```

Searching for 0.3 with `epsilon = 1e-6`:

- `fabs(0.3000000001 - 0.3) < 1e-6` → found!

## Tiny Code

## C Implementation

```

#include <stdio.h>
#include <math.h>

int binary_search_tolerance(double arr[], int n, double target, double eps) {
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        double diff = arr[mid] - target;
        if (fabs(diff) < eps)
            return mid; // found within tolerance
        else if (diff < 0)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1; // not found
}

int main() {
    double arr[] = {0.1, 0.2, 0.3000000001, 0.4, 0.5};
    int n = 5;
    double x = 0.3;
    int idx = binary_search_tolerance(arr, n, x, 1e-6);
    if (idx != -1)
        printf("Found %.6f at index %d\n", x, idx);
    else
        printf("Not found\n");
}

```

Output:

Found 0.300000 at index 2

### Python Implementation

```

def binary_search_tolerance(arr, x, eps=1e-6):
    lo, hi = 0, len(arr) - 1
    while lo <= hi:
        mid = (lo + hi) // 2
        if abs(arr[mid] - x) < eps:

```

```

        return mid
    elif arr[mid] < x:
        lo = mid + 1
    else:
        hi = mid - 1
    return -1

arr = [0.1, 0.2, 0.300000001, 0.4, 0.5]
print(binary_search_tolerance(arr, 0.3)) # 2

```

## Why It Matters

- Avoids false negatives when comparing floats
- Handles round-off errors gracefully
- Useful in:
  - Root-finding
  - Floating-point datasets
  - Physics simulations
  - Numerical optimization

## Intuition

Binary search assumes exact comparison. With floating-point numbers, “equal” often means “close enough.” defines your acceptable margin of error.

Think of it as:

“If the difference is less than , consider it found.”

## Try It Yourself

1. Use an array [0.1, 0.2, 0.3, 0.4, 0.5] Search for 0.3000001 with  $\epsilon = 1e-5$
2. Reduce  $\epsilon \rightarrow$  observe when search fails
3. Try negative numbers or decimals
4. Compare with integer binary search
5. Experiment with non-uniform spacing
6. Modify to find nearest value if not within
7. Visualize tolerance as a small band around target
8. Apply in root finding ( $f(x) = 0$ )

9. Adjust dynamically based on scale
10. Measure precision loss with large floats

## Test Cases

| Array                    | Target   | Epsilon | Expected  |
|--------------------------|----------|---------|-----------|
| [0.1, 0.2, 0.3000000001] | 0.3      | 1e-6    | Found     |
| [1.0, 2.0, 3.0]          | 2.000001 | 1e-5    | Found     |
| [1.0, 2.0, 3.0]          | 2.1      | 1e-5    | Not found |

## Complexity

| Step   | Complexity  |
|--------|-------------|
| Search | $O(\log n)$ |
| Space  | $O(1)$      |

A Binary Search with Tolerance is a small but essential upgrade when dealing with real numbers—because in floating-point land, “close enough” is often the truth.

## 192 Ternary Search

A Ternary Search is an algorithm for finding the maximum or minimum of a unimodal function—a function that increases up to a point and then decreases (or vice versa). It is a divide-and-conquer method similar to binary search but splits the range into three parts each time.

### What Problem Are We Solving??

You have a function  $f(x)$  defined on an interval  $[l, r]$ , and you know it has one peak (or valley). You want to find the  $x$  where  $f(x)$  is maximized (or minimized).

Unlike binary search (which searches for equality), ternary search searches for extremes.

## Example (Unimodal Function)

Let

$$f(x) = -(x - 2)^2 + 4$$

This is a parabola with a maximum at  $x = 2$ .

Ternary search gradually narrows the interval around the maximum:

1. Divide  $[l, r]$  into three parts
2. Evaluate  $f(m_1)$  and  $f(m_2)$
3. Keep the side that contains the peak
4. Repeat until small enough

## How It Works (Step-by-Step)

| Step | Action                                                                |
|------|-----------------------------------------------------------------------|
| 1    | Pick two midpoints: $m_1 = l + (r - l) / 3$ , $m_2 = r - (r - l) / 3$ |
| 2    | Compare $f(m_1)$ and $f(m_2)$                                         |
| 3    | If $f(m_1) < f(m_2) \rightarrow$ maximum is in $[m_1, r]$             |
| 4    | Else $\rightarrow$ maximum is in $[l, m_2]$                           |
| 5    | Repeat until $r - l <$                                                |

## Tiny Code

### C Implementation (Maximization)

```
#include <stdio.h>
#include <math.h>

double f(double x) {
    return -pow(x - 2, 2) + 4; // peak at x=2
}

double ternary_search(double l, double r, double eps) {
    while (r - l > eps) {
        double m1 = l + (r - l) / 3;
        double m2 = r - (r - l) / 3;
```

```

        if (f(m1) < f(m2))
            l = m1;
        else
            r = m2;
    }
    return (l + r) / 2; // approx peak
}

int main() {
    double l = 0, r = 4;
    double res = ternary_search(l, r, 1e-6);
    printf("Approx max at x = %.6f, f(x) = %.6f\n", res, f(res));
}

```

Output:

Approx max at x = 2.000000, f(x) = 4.000000

## Python Implementation

```

def f(x):
    return -(x - 2)**2 + 4 # Peak at x=2

def ternary_search(l, r, eps=1e-6):
    while r - l > eps:
        m1 = l + (r - l) / 3
        m2 = r - (r - l) / 3
        if f(m1) < f(m2):
            l = m1
        else:
            r = m2
    return (l + r) / 2

res = ternary_search(0, 4)
print(f"Max at x = {res:.6f}, f(x) = {f(res):.6f}")

```

## Why It Matters

- Finds extrema (max/min) in continuous functions

- No derivative required (unlike calculus-based optimization)
- Works when:
  - $f(x)$  is unimodal
  - Domain is continuous
  - You can evaluate  $f(x)$  cheaply

Used in:

- Mathematical optimization
- Machine learning hyperparameter tuning
- Geometry problems (e.g., closest distance)
- Physics simulations

### Try It Yourself

1. Try  $f(x) = (x - 5)^2 + 1$  (minimization)
2. Use interval  $[0, 10]$  and  $\text{eps} = 1e-6$
3. Change  $\text{eps}$  to  $1e-3 \rightarrow$  observe faster but rougher result
4. Apply to distance between two moving points
5. Compare with binary search on derivative
6. Plot  $f(x)$  to visualize narrowing intervals
7. Switch condition to find minimum
8. Test with  $f(x) = \sin(x)$  on  $[0, \pi]$
9. Use integer search version for discrete arrays
10. Combine with golden section search for efficiency

### Test Cases

| Function              | Interval    | Expected   | Type    |
|-----------------------|-------------|------------|---------|
| $f(x) = -(x-2)^2 + 4$ | $[0, 4]$    | $x = 2$    | Maximum |
| $f(x) = (x-5)^2 + 1$  | $[0, 10]$   | $x = 5$    | Minimum |
| $f(x) = \sin(x)$      | $[0, 3.14]$ | $x = 1.57$ | Maximum |

### Complexity

| Metric | Value                       |
|--------|-----------------------------|
| Time   | $O(\log((r - l)/\epsilon))$ |
| Space  | $O(1)$                      |

| Metric | Value |
|--------|-------|
|        |       |

A Ternary Search slices the search space into thirds—zooming in on the peak or valley with mathematical precision, no derivatives required.

## 193 Hash-Based Search

A Hash-Based Search uses a hash function to map keys directly to indices in a table, giving constant-time expected lookups. Instead of scanning or comparing elements, it jumps straight to the bucket where the data should be.

### What Problem Are We Solving??

When searching in a large dataset, linear search is too slow ( $O(n)$ ) and binary search needs sorted data ( $O(\log n)$ ). Hash-based search lets you find, insert, or delete an item in  $O(1)$  average time, regardless of ordering.

It's the foundation of hash tables, hash maps, and dictionaries.

### Example (Simple Lookup)

Suppose you want to store and search for names quickly:

```
$$"Alice", "Bob", "Carol", "Dave"]
```

A hash function maps each name to an index:

```
hash("Alice") → 2
hash("Bob")   → 5
hash("Carol") → 1
```

You place each name in its corresponding slot. Searching becomes instant:

```
hash("Bob") = 5 → found!
```

## How It Works (Plain Language)

| Step | Action                                                                                                |
|------|-------------------------------------------------------------------------------------------------------|
| 1    | Compute hash(key) to get an index                                                                     |
| 2    | Look up the bucket at that index                                                                      |
| 3    | If multiple items hash to same bucket (collision), handle with a strategy (chaining, open addressing) |
| 4    | Compare keys if necessary                                                                             |
| 5    | Return result                                                                                         |

## Tiny Code

### C Implementation (with Linear Probing)

```
#include <stdio.h>
#include <string.h>

#define SIZE 10

typedef struct {
    char key[20];
    int value;
    int used;
} Entry;

Entry table[SIZE];

int hash(char *key) {
    int h = 0;
    for (int i = 0; key[i]; i++)
        h = (h * 31 + key[i]) % SIZE;
    return h;
}

void insert(char *key, int value) {
    int h = hash(key);
    while (table[h].used) {
        if (strcmp(table[h].key, key) == 0) break;
        h = (h + 1) % SIZE;
    }
    table[h].key = key;
    table[h].value = value;
    table[h].used = 1;
}
```

```

    }

    strcpy(table[h].key, key);
    table[h].value = value;
    table[h].used = 1;
}

int search(char *key) {
    int h = hash(key), start = h;
    while (table[h].used) {
        if (strcmp(table[h].key, key) == 0)
            return table[h].value;
        h = (h + 1) % SIZE;
        if (h == start) break;
    }
    return -1; // not found
}

int main() {
    insert("Alice", 10);
    insert("Bob", 20);
    printf("Value for Bob: %d\n", search("Bob"));
}

```

Output:

Value for Bob: 20

### Python Implementation (Built-in Dict)

```

people = {"Alice": 10, "Bob": 20, "Carol": 30}

print("Bob" in people)      # True
print(people["Carol"])     # 30

```

Python's `dict` uses an optimized open addressing hash table.

### Why It Matters

- O(1) average lookup, insertion, and deletion

- No need for sorting
- Core to symbol tables, caches, dictionaries, compilers, and databases
- Can scale with resizing (rehashing)

### Try It Yourself

1. Implement hash table with chaining using linked lists
2. Replace linear probing with quadratic probing
3. Measure lookup time vs. linear search on same dataset
4. Insert keys with collisions, ensure correctness
5. Create custom hash for integers:  $h(x) = x \% m$
6. Observe performance as table fills up (load factor  $> 0.7$ )
7. Implement delete operation carefully
8. Resize table when load factor high
9. Try a poor hash function (like  $h=1$ ) and measure slowdown
10. Compare with Python's built-in dict

### Test Cases

| Operation | Input         | Expected Output |
|-----------|---------------|-----------------|
| Insert    | (“Alice”, 10) | Stored          |
| Insert    | (“Bob”, 20)   | Stored          |
| Search    | “Alice”       | 10              |
| Search    | “Eve”         | Not found       |
| Delete    | “Bob”         | Removed         |
| Search    | “Bob”         | Not found       |

### Complexity

| Metric | Average | Worst Case |
|--------|---------|------------|
| Search | $O(1)$  | $O(n)$     |
| Insert | $O(1)$  | $O(n)$     |
| Space  | $O(n)$  | $O(n)$     |

A Hash-Based Search is like a magic index, it jumps straight to the data you need, turning search into instant lookup.

## 194 Bloom Filter Lookup

A Bloom Filter is a probabilistic data structure that tells you if an element is definitely not in a set or possibly is. It's super fast and memory efficient, but allows false positives (never false negatives).

### What Problem Are We Solving??

When working with huge datasets (like URLs, cache keys, or IDs), you may not want to store every element just to check membership. Bloom Filters give you a fast O(1) check:

- “Is this element in my set?” → Maybe
- “Is it definitely not in my set?” → Yes

They're widely used in databases, network systems, and search engines (e.g., to skip disk lookups).

### Example (Cache Lookup)

You have a cache of 1 million items. Before hitting the database, you want to know:

Should I even bother checking cache for this key?

Use a Bloom Filter to quickly tell if the key *could be* in cache. If filter says “no,” skip lookup entirely.

### How It Works (Plain Language)

| Step | Action                                                                |
|------|-----------------------------------------------------------------------|
| 1    | Create a bit array of size $m$ (all zeros)                            |
| 2    | Choose $k$ independent hash functions                                 |
| 3    | To insert an element: compute $k$ hashes, set corresponding bits to 1 |
| 4    | To query an element: compute $k$ hashes, check if all bits are 1      |
| 5    | If any bit = 0 → definitely not in set                                |
| 6    | If all bits = 1 → possibly in set (false positive possible)           |

## Tiny Code

### Python (Simple Bloom Filter)

```
from hashlib import sha256

class BloomFilter:
    def __init__(self, size=1000, hash_count=3):
        self.size = size
        self.hash_count = hash_count
        self.bits = [0] * size

    def _hashes(self, item):
        for i in range(self.hash_count):
            h = int(sha256((item + str(i)).encode()).hexdigest(), 16)
            yield h % self.size

    def add(self, item):
        for h in self._hashes(item):
            self.bits[h] = 1

    def contains(self, item):
        return all(self.bits[h] for h in self._hashes(item))

# Example usage
bf = BloomFilter()
bf.add("Alice")
print(bf.contains("Alice")) # True (probably)
print(bf.contains("Bob")) # False (definitely not)
```

Output:

```
True
False
```

## Why It Matters

- Memory efficient for large sets
- No false negatives, if it says “no,” you can trust it
- Used in caches, databases, distributed systems
- Reduces I/O by skipping non-existent entries

## Try It Yourself

1. Build a Bloom Filter with 1000 bits and 3 hash functions
2. Insert 100 random elements
3. Query 10 existing and 10 non-existing elements
4. Measure false positive rate
5. Experiment with different  $m$  and  $k$  values
6. Integrate into a simple cache simulation
7. Implement double hashing to reduce hash cost
8. Compare memory use vs. Python `set`
9. Add support for merging filters (bitwise OR)
10. Try a Counting Bloom Filter (to support deletions)

## Test Cases

| Input                          | Expected Output        |
|--------------------------------|------------------------|
| <code>add("Alice")</code>      | Bits set               |
| <code>contains("Alice")</code> | True (maybe)           |
| <code>contains("Bob")</code>   | False (definitely not) |
| <code>add("Bob")</code>        | Bits updated           |
| <code>contains("Bob")</code>   | True (maybe)           |

## Complexity

| Operation | Time   | Space  |
|-----------|--------|--------|
| Insert    | $O(k)$ | $O(m)$ |
| Lookup    | $O(k)$ | $O(m)$ |

$k$ : number of hash functions  $m$ : size of bit array

A Bloom Filter is like a polite doorman, it'll never wrongly turn you away, but it might let in a stranger once in a while.

## 195 Cuckoo Hash Search

A Cuckoo Hash Table is a clever hash-based structure that guarantees  $O(1)$  lookup while avoiding long probe chains. It uses two hash functions and relocates existing keys when a collision occurs, just like a cuckoo bird kicking eggs out of a nest.

## What Problem Are We Solving??

Traditional hash tables can degrade to  $O(n)$  when collisions pile up. Cuckoo hashing ensures constant-time lookups by guaranteeing every key has at most two possible positions. If both are taken, it kicks out an existing key and re-inserts it elsewhere.

It's widely used in network routers, high-performance caches, and hash-based indexes.

## Example (Small Table)

Suppose you have 2 hash functions:

```
h1(x) = x % 3  
h2(x) = (x / 3) % 3
```

Insert keys: 5, 8, 11

| Key | h1 | h2 |
|-----|----|----|
| 5   | 2  | 1  |
| 8   | 2  | 2  |
| 11  | 2  | 1  |

When inserting 11, slot 2 is full, so we kick out 5 to its alternate position. This continues until every key finds a home.

## How It Works (Plain Language)

| Step | Action                                                         |
|------|----------------------------------------------------------------|
| 1    | Compute two hash indices: $h1(key)$ , $h2(key)$                |
| 2    | Try placing the key in $h1$ slot                               |
| 3    | If occupied, evict existing key to its alternate slot          |
| 4    | Repeat up to a threshold (to prevent infinite loops)           |
| 5    | If full cycle detected $\rightarrow$ rehash with new functions |

Each key lives in either position  $h1(key)$  or  $h2(key)$ .

## Tiny Code

### C Implementation (Simplified)

```
#include <stdio.h>

#define SIZE 7

int table1[SIZE], table2[SIZE];

int h1(int key) { return key % SIZE; }
int h2(int key) { return (key / SIZE) % SIZE; }

void insert(int key, int depth) {
    if (depth > SIZE) return; // avoid infinite loop
    int pos1 = h1(key);
    if (table1[pos1] == 0) {
        table1[pos1] = key;
        return;
    }
    int displaced = table1[pos1];
    table1[pos1] = key;
    int pos2 = h2(displaced);
    if (table2[pos2] == 0)
        table2[pos2] = displaced;
    else
        insert(displaced, depth + 1);
}

int search(int key) {
    return table1[h1(key)] == key || table2[h2(key)] == key;
}

int main() {
    insert(10, 0);
    insert(20, 0);
    insert(30, 0);
    printf("Search 20: %s\n", search(20) ? "Found" : "Not Found");
}
```

Output:

Search 20: Found

## Python Implementation

```
SIZE = 7
table1 = [None] * SIZE
table2 = [None] * SIZE

def h1(x): return x % SIZE
def h2(x): return (x // SIZE) % SIZE

def insert(key, depth=0):
    if depth > SIZE:
        return False # cycle detected
    pos1 = h1(key)
    if table1[pos1] is None:
        table1[pos1] = key
        return True
    key, table1[pos1] = table1[pos1], key
    pos2 = h2(key)
    if table2[pos2] is None:
        table2[pos2] = key
        return True
    return insert(key, depth + 1)

def search(key):
    return table1[h1(key)] == key or table2[h2(key)] == key

insert(10); insert(20); insert(30)
print("Search 20:", search(20))
```

## Why It Matters

- O(1) worst-case lookup (always two probes)
- Eliminates long collision chains
- Great for high-performance systems
- Deterministic position = easy debugging

## Try It Yourself

1. Insert numbers 1–10 and trace movements
2. Add detection for cycles → trigger rehash
3. Compare average probe count vs. linear probing
4. Implement delete(key) (mark slot empty)
5. Try resizing table dynamically
6. Use different hash functions for variety
7. Track load factor before rehashing
8. Store (key, value) pairs
9. Benchmark against chaining
10. Visualize movement paths during insertion

## Test Cases

| Operation | Input | Expected              |
|-----------|-------|-----------------------|
| Insert    | 5     | Placed                |
| Insert    | 8     | Kicked 5, placed both |
| Search    | 5     | Found                 |
| Search    | 11    | Not Found             |
| Delete    | 8     | Removed               |
| Search    | 8     | Not Found             |

## Complexity

| Operation | Time           | Space  |
|-----------|----------------|--------|
| Search    | $O(1)$         | $O(n)$ |
| Insert    | $O(1)$ average | $O(n)$ |
| Delete    | $O(1)$         | $O(n)$ |

Cuckoo hashing is like musical chairs for keys, when one can't sit, it makes another stand up and move, but everyone eventually finds a seat.

## 196 Robin Hood Hashing

Robin Hood Hashing is an open addressing strategy that balances fairness in hash table lookups. When two keys collide, the one that's traveled farther from its "home" index gets to stay, stealing the slot like Robin Hood, who took from the rich and gave to the poor.

## **What Problem Are We Solving??**

Standard linear probing can cause clusters, long runs of occupied slots that slow searches. Robin Hood Hashing reduces variance in probe lengths, so every key has roughly equal access time. This leads to more predictable performance, even at high load factors.

## **Example (Collision Handling)**

Suppose  $\text{hash}(x) = x \% 10$ . Insert [10, 20, 30, 21].

| Key | Hash | Position      | Probe Distance |
|-----|------|---------------|----------------|
| 10  | 0    | 0             | 0              |
| 20  | 0    | 1             | 1              |
| 30  | 0    | 2             | 2              |
| 21  | 1    | 2 (collision) | 1              |

At position 2, 21 meets 30 (whose distance = 2). Since 21's distance (1) is less, it keeps probing. Robin Hood rule: if newcomer has greater or equal distance  $\rightarrow$  swap. This keeps distribution even.

## **How It Works (Plain Language)**

| Step | Action                                                                                         |
|------|------------------------------------------------------------------------------------------------|
| 1    | Compute hash = <code>key % table_size</code>                                                   |
| 2    | If slot empty $\rightarrow$ place item                                                         |
| 3    | Else, compare probe distance with occupant's                                                   |
| 4    | If new item's distance $\geq$ current's $\rightarrow$ swap and continue probing displaced item |
| 5    | Repeat until inserted                                                                          |

## **Tiny Code**

### **C Implementation (Simplified)**

```
#include <stdio.h>

#define SIZE 10
```

```

typedef struct {
    int key;
    int used;
    int distance;
} Entry;

Entry table[SIZE];

int hash(int key) { return key % SIZE; }

void insert(int key) {
    int index = hash(key);
    int dist = 0;
    while (table[index].used) {
        if (table[index].distance < dist) {
            int temp_key = table[index].key;
            int temp_dist = table[index].distance;
            table[index].key = key;
            table[index].distance = dist;
            key = temp_key;
            dist = temp_dist;
        }
        index = (index + 1) % SIZE;
        dist++;
    }
    table[index].key = key;
    table[index].distance = dist;
    table[index].used = 1;
}

int search(int key) {
    int index = hash(key);
    int dist = 0;
    while (table[index].used) {
        if (table[index].key == key) return 1;
        if (table[index].distance < dist) return 0;
        index = (index + 1) % SIZE;
        dist++;
    }
    return 0;
}

```

```

int main() {
    insert(10);
    insert(20);
    insert(30);
    printf("Search 20: %s\n", search(20) ? "Found" : "Not Found");
}

```

Output:

Search 20: Found

### Python Implementation

```

SIZE = 10
table = [None] * SIZE
distances = [0] * SIZE

def h(k): return k % SIZE

def insert(key):
    index = h(key)
    dist = 0
    while table[index] is not None:
        if distances[index] < dist:
            table[index], key = key, table[index]
            distances[index], dist = dist, distances[index]
        index = (index + 1) % SIZE
        dist += 1
    table[index], distances[index] = key, dist

def search(key):
    index = h(key)
    dist = 0
    while table[index] is not None:
        if table[index] == key:
            return True
        if distances[index] < dist:
            return False
        index = (index + 1) % SIZE
        dist += 1

```

```
    return False

insert(10); insert(20); insert(30)
print("Search 20:", search(20))
```

## Why It Matters

- Predictable lookup times, probe lengths nearly uniform
- Outperforms linear probing at high load
- Fewer long clusters, more stable table behavior
- Great for performance-critical systems

## Try It Yourself

1. Insert [10, 20, 30, 21] and trace swaps
2. Measure probe length variance vs. linear probing
3. Implement delete() with tombstone handling
4. Add resizing when load factor  $> 0.8$
5. Compare performance with quadratic probing
6. Visualize table after 20 inserts
7. Experiment with different table sizes
8. Create histogram of probe lengths
9. Add key-value pair storage
10. Benchmark search time at 70%, 80%, 90% load

## Test Cases

| Operation | Input          | Expected                   |
|-----------|----------------|----------------------------|
| Insert    | 10, 20, 30, 21 | Evenly distributed         |
| Search    | 20             | Found                      |
| Search    | 40             | Not Found                  |
| Delete    | 10             | Removed                    |
| Insert    | 50             | Placed at optimal position |

## Complexity

| Operation | Time     | Space |
|-----------|----------|-------|
| Search    | O(1) avg | O(n)  |
| Insert    | O(1) avg | O(n)  |
| Delete    | O(1) avg | O(n)  |

Robin Hood Hashing keeps everyone honest, no key hoards fast access, and none are left wandering too far.

## 197 Jump Consistent Hashing

Jump Consistent Hashing is a lightweight, fast, and deterministic way to assign keys to buckets (servers, shards, or partitions) that minimizes remapping when the number of buckets changes.

It's designed for load balancing in distributed systems, like database shards or cache clusters.

### What Problem Are We Solving??

When scaling systems horizontally, you often need to assign keys (like user IDs) to buckets (like servers). Naive methods (e.g. `key % N`) cause massive remapping when  $N$  changes. Jump Consistent Hashing avoids that, only a small fraction of keys move when a bucket is added or removed.

This ensures stability and predictable redistribution, ideal for distributed caches (Memcached, Redis) and databases (Bigtable, Ceph).

### Example (Adding Buckets)

Suppose we have keys 1 to 6 and 3 buckets:

| Key | Bucket (3) |
|-----|------------|
| 1   | 2          |
| 2   | 0          |
| 3   | 2          |
| 4   | 1          |
| 5   | 0          |
| 6   | 2          |

When we add a new bucket (4 total), only a few keys change buckets, most stay where they are. That's the magic of *consistency*.

## How It Works (Plain Language)

| Step | Action                                                                                                                                                                                                |
|------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1    | Treat key as a 64-bit integer                                                                                                                                                                         |
| 2    | Initialize $b = -1$ and $j = 0$                                                                                                                                                                       |
| 3    | While $j < \text{num\_buckets}$ , update:<br>$b = j$ , $\text{key} = \text{key} * 2862933555777941757\text{ULL} + 1$<br>$j = \text{floor}((b + 1) * (1\text{LL} \ll 31) / ((\text{key} \gg 33) + 1))$ |
| 4    | Return $b$ as bucket index                                                                                                                                                                            |

It uses integer arithmetic only, no tables, no storage, just math.

## Tiny Code

### C Implementation

```
#include <stdint.h>
#include <stdio.h>

int jump_consistent_hash(uint64_t key, int num_buckets) {
    int64_t b = -1, j = 0;
    while (j < num_buckets) {
        b = j;
        key = key * 2862933555777941757ULL + 1;
        j = (b + 1) * ((double)(1LL << 31) / ((key >> 33) + 1));
    }
    return (int)b;
}

int main() {
    for (uint64_t k = 1; k <= 6; k++)
        printf("Key %llu → Bucket %d\n", k, jump_consistent_hash(k, 3));
}
```

Output:

```
Key 1 → Bucket 2
Key 2 → Bucket 0
```

```
Key 3 → Bucket 2
Key 4 → Bucket 1
Key 5 → Bucket 0
Key 6 → Bucket 2
```

## Python Implementation

```
def jump_hash(key, num_buckets):
    b, j = -1, 0
    while j < num_buckets:
        b = j
        key = key * 2862933555777941757 + 1
        j = int((b + 1) * (1 << 31) / ((key >> 33) + 1))
    return b

for k in range(1, 7):
    print(f"Key {k} → Bucket {jump_hash(k, 3)}")
```

## Why It Matters

- Stable distribution: minimal remapping on resize
- O(1) time, O(1) space
- Works great for sharding, load balancing, partitioning
- No need for external storage or ring structures (unlike consistent hashing rings)

## Try It Yourself

1. Assign 10 keys across 3 buckets
2. Add a 4th bucket and see which keys move
3. Compare with `key % N` approach
4. Try very large bucket counts (10k+)
5. Benchmark speed, notice it's almost constant
6. Integrate with a distributed cache simulation
7. Test uniformity (distribution of keys)
8. Add random seeds for per-service variation
9. Visualize redistribution pattern
10. Compare with Rendezvous Hashing

## Test Cases

| Input      | Buckets | Output                   |
|------------|---------|--------------------------|
| Key=1, N=3 | 3       | 2                        |
| Key=2, N=3 | 3       | 0                        |
| Key=3, N=3 | 3       | 2                        |
| Key=1, N=4 | 4       | 3 or 2 (depends on math) |

Only a fraction of keys remap when N changes.

## Complexity

| Operation | Time | Space |
|-----------|------|-------|
| Lookup    | O(1) | O(1)  |
| Insert    | O(1) | O(1)  |

Jump Consistent Hashing is like a steady hand, even as your system grows, it keeps most keys right where they belong.

## 198 Prefix Search in Trie

A Trie (prefix tree) is a specialized tree structure that stores strings by their prefixes, enabling fast prefix-based lookups, ideal for autocomplete, dictionaries, and word search engines.

With a trie, searching for “app” instantly finds “apple”, “apply”, “appetite”, etc.

### What Problem Are We Solving??

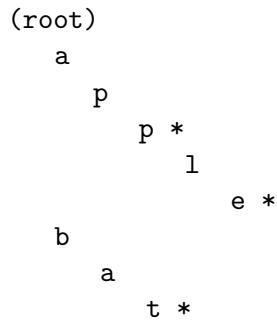
Traditional data structures like arrays or hash tables can't efficiently answer questions like:

- “List all words starting with `pre`”
- “Does any word start with `tri`? ”

A trie organizes data by prefix paths, making such queries fast and natural, often  $O(k)$  where  $k$  is the prefix length.

## Example (Words: app, apple, bat)

The trie looks like this:



Stars (\*) mark word endings. Search “app” → found; list all completions → “app”, “apple”.

## How It Works (Plain Language)

| Step | Action                                                       |
|------|--------------------------------------------------------------|
| 1    | Each node represents a character                             |
| 2    | A path from root to node represents a prefix                 |
| 3    | When inserting, follow characters and create nodes as needed |
| 4    | Mark end of word when reaching last character                |
| 5    | To search prefix: traverse nodes character by character      |
| 6    | If all exist → prefix found; else → not found                |

## Tiny Code

### C Implementation (Simplified)

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define ALPHABET 26

typedef struct TrieNode {
    struct TrieNode *child[ALPHABET];
```

```

        bool isEnd;
    } TrieNode;

TrieNode* newNode() {
    TrieNode* node = calloc(1, sizeof(TrieNode));
    node->isEnd = false;
    return node;
}

void insert(TrieNode *root, const char *word) {
    for (int i = 0; word[i]; i++) {
        int idx = word[i] - 'a';
        if (!root->child[idx]) root->child[idx] = newNode();
        root = root->child[idx];
    }
    root->isEnd = true;
}

bool startsWith(TrieNode *root, const char *prefix) {
    for (int i = 0; prefix[i]; i++) {
        int idx = prefix[i] - 'a';
        if (!root->child[idx]) return false;
        root = root->child[idx];
    }
    return true;
}

int main() {
    TrieNode *root = newNode();
    insert(root, "app");
    insert(root, "apple");
    printf("Starts with 'ap': %s\n", startsWith(root, "ap") ? "Yes" : "No");
}

```

Output:

Starts with 'ap': Yes

## Python Implementation

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for ch in word:
            if ch not in node.children:
                node.children[ch] = TrieNode()
            node = node.children[ch]
        node.is_end = True

    def starts_with(self, prefix):
        node = self.root
        for ch in prefix:
            if ch not in node.children:
                return False
            node = node.children[ch]
        return True

trie = Trie()
trie.insert("apple")
trie.insert("app")
print(trie.starts_with("ap")) # True
```

## Why It Matters

- $O(k)$  prefix lookup ( $k = \text{prefix length}$ )
- Perfect for autocomplete, spell checkers, search engines
- Stores shared prefixes efficiently
- Can be extended for frequency, weights, or wildcard matching

## Try It Yourself

1. Insert [“app”, “apple”, “apply”, “apt”]
2. Search prefix “ap” → expect True
3. Search prefix “ba” → expect False
4. Add function to list all words starting with a prefix
5. Implement delete(word)
6. Add frequency count at each node
7. Support uppercase letters
8. Store end-of-word markers clearly
9. Compare memory use vs hash table
10. Extend to autocomplete feature returning top-N completions

## Test Cases

| Operation  | Input   | Output |
|------------|---------|--------|
| Insert     | “app”   | Stored |
| Insert     | “apple” | Stored |
| startsWith | “ap”    | True   |
| startsWith | “ba”    | False  |
| startsWith | “app”   | True   |

## Complexity

| Operation       | Time                              | Space  |
|-----------------|-----------------------------------|--------|
| Insert          | $O(k)$                            | $O(k)$ |
| Search Prefix   | $O(k)$                            | $O(1)$ |
| Space (n words) | $O(\text{sum of all characters})$ | ,      |

A Trie turns your data into a map of words, every branch is a path toward meaning, every prefix a shortcut to discovery.

## 199 Pattern Search in Suffix Array

A Suffix Array is a sorted list of all suffixes of a string. It enables fast substring searches, perfect for pattern matching in text editors, DNA analysis, and search engines.

By combining it with binary search, you can find whether a pattern appears in a string in  $O(m \log n)$  time.

## What Problem Are We Solving??

Given a large text T (like "banana") and a pattern P (like "ana"), we want to quickly check if P exists in T. Naive search takes  $O(nm)$  (compare every position). A suffix array lets us search more efficiently by working on a pre-sorted list of suffixes.

### Example (Text = "banana")

List all suffixes and sort them:

| Index | Suffix |
|-------|--------|
| 0     | banana |
| 1     | anana  |
| 2     | nana   |
| 3     | ana    |
| 4     | na     |
| 5     | a      |

Sorted suffixes:

| SA Index | Suffix | Original Position |
|----------|--------|-------------------|
| 5        | a      | 5                 |
| 3        | ana    | 3                 |
| 1        | anana  | 1                 |
| 0        | banana | 0                 |
| 4        | na     | 4                 |
| 2        | nana   | 2                 |

Now search "ana" using binary search over these sorted suffixes.

### How It Works (Plain Language)

| Step | Action                                                     |
|------|------------------------------------------------------------|
| 1    | Build suffix array = all suffixes sorted lexicographically |
| 2    | Use binary search to find lower/upper bounds for pattern   |
| 3    | Compare only $m$ characters per comparison                 |
| 4    | If found $\rightarrow$ pattern occurs at suffix index      |
| 5    | Otherwise $\rightarrow$ not in text                        |

## Tiny Code

### C Implementation (Simplified)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int cmp(const void *a, const void *b, void *txt) {
    int i = *(int*)a, j = *(int*)b;
    return strcmp((char*)txt + i, (char*)txt + j);
}

void build_suffix_array(char *txt, int n, int sa[]) {
    for (int i = 0; i < n; i++) sa[i] = i;
    qsort_r(sa, n, sizeof(int), cmp, txt);
}

int binary_search_suffix(char *txt, int sa[], int n, char *pat) {
    int l = 0, r = n - 1;
    while (l <= r) {
        int mid = (l + r) / 2;
        int res = strncmp(pat, txt + sa[mid], strlen(pat));
        if (res == 0) return sa[mid];
        if (res < 0) r = mid - 1;
        else l = mid + 1;
    }
    return -1;
}

int main() {
    char txt[] = "banana";
    int n = strlen(txt), sa[n];
    build_suffix_array(txt, n, sa);
    char pat[] = "ana";
    int pos = binary_search_suffix(txt, sa, n, pat);
    if (pos >= 0) printf("Pattern found at %d\n", pos);
    else printf("Pattern not found\n");
}
```

Output:

Pattern found at 1

### Python Implementation

```
def build_suffix_array(s):
    return sorted(range(len(s)), key=lambda i: s[i:])

def search(s, sa, pat):
    l, r = 0, len(sa) - 1
    while l <= r:
        mid = (l + r) // 2
        suffix = s[sa[mid]:]
        if suffix.startswith(pat):
            return sa[mid]
        if suffix < pat:
            l = mid + 1
        else:
            r = mid - 1
    return -1

s = "banana"
sa = build_suffix_array(s)
print("Suffix Array:", sa)
print("Search 'ana':", search(s, sa, "ana"))
```

Output:

```
Suffix Array: [5, 3, 1, 0, 4, 2]
Search 'ana': 1
```

### Why It Matters

- Substring search in  $O(m \log n)$
- Space-efficient alternative to suffix trees
- Great for full-text search, DNA sequencing, plagiarism detection
- Can be extended with LCP array for longest common prefix queries

## Try It Yourself

1. Build suffix array for "banana"
2. Search "na", "ban", "apple"
3. Print all suffixes for visualization
4. Add LCP array for faster repeated queries
5. Compare with KMP algorithm speed
6. Use binary search manually to trace comparisons
7. Extend to count occurrences of pattern
8. Try longer text (e.g. "mississippi")
9. Implement in-place quicksort for SA
10. Benchmark vs. naive substring search

## Test Cases

| Text     | Pattern | Expected Result |
|----------|---------|-----------------|
| "banana" | "ana"   | Found at 1      |
| "banana" | "ban"   | Found at 0      |
| "banana" | "na"    | Found at 2 or 4 |
| "banana" | "cat"   | Not Found       |

## Complexity

| Operation | Time          | Space  |
|-----------|---------------|--------|
| Build SA  | $O(n \log n)$ | $O(n)$ |
| Search    | $O(m \log n)$ | $O(1)$ |

A Suffix Array is like a library index, once sorted, every search becomes as simple as flipping to the right page.

## 200 Search in Infinite Array

A Search in Infinite Array (or unbounded array) is a technique for finding an element in a sorted list of unknown length. You can't directly use binary search because you don't know  $n$ , so you must first find a search bound, then perform binary search within it.

This idea is crucial for systems where data is streamed or dynamically sized, like logs, unrolled lists, or file scans.

## What Problem Are We Solving??

If you're given an array-like interface (like `get(i)`), but no size `n`, how do you find `target` efficiently? You can't do linear search, it could be infinite. The trick: exponential search, grow bounds exponentially until you pass the target, then apply binary search in that window.

## Example

Given sorted sequence:

```
[$$3, 5, 9, 12, 17, 23, 31, 45, 67, 88, 100, ...]
```

Search for 31:

1. Start with `low = 0, high = 1`
2. While `arr[high] < 31`, double `high`  
`high = 1 → 2 → 4 → 8`
3. Now `arr[8] = 67 > 31`, so search range is `[4, 8]`
4. Perform binary search in `[4, 8]`
5. Found at index 6

## How It Works (Plain Language)

| Step | Action                                                                            |
|------|-----------------------------------------------------------------------------------|
| 1    | Start with <code>low = 0, high = 1</code>                                         |
| 2    | While <code>arr[high] &lt; target</code> , set <code>low = high, high *= 2</code> |
| 3    | Now the target lies between <code>low</code> and <code>high</code>                |
| 4    | Perform standard binary search in <code>[low, high]</code>                        |
| 5    | Return index if found, else -1                                                    |

## Tiny Code

### C Implementation (Simulated Infinite Array)

```

#include <stdio.h>

int get(int arr[], int size, int i) {
    if (i >= size) return 1e9; // simulate infinity
    return arr[i];
}

int binary_search(int arr[], int low, int high, int target, int size) {
    while (low <= high) {
        int mid = low + (high - low) / 2;
        int val = get(arr, size, mid);
        if (val == target) return mid;
        if (val < target) low = mid + 1;
        else high = mid - 1;
    }
    return -1;
}

int search_infinite(int arr[], int size, int target) {
    int low = 0, high = 1;
    while (get(arr, size, high) < target) {
        low = high;
        high *= 2;
    }
    return binary_search(arr, low, high, target, size);
}

int main() {
    int arr[] = {3, 5, 9, 12, 17, 23, 31, 45, 67, 88, 100};
    int size = sizeof(arr)/sizeof(arr[0]);
    int idx = search_infinite(arr, size, 31);
    printf("Found at index %d\n", idx);
}

```

Output:

Found at index 6

## Python Implementation

```
def get(arr, i):
    return arr[i] if i < len(arr) else float('inf')

def binary_search(arr, low, high, target):
    while low <= high:
        mid = (low + high) // 2
        val = get(arr, mid)
        if val == target:
            return mid
        if val < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1

def search_infinite(arr, target):
    low, high = 0, 1
    while get(arr, high) < target:
        low = high
        high *= 2
    return binary_search(arr, low, high, target)

arr = [3, 5, 9, 12, 17, 23, 31, 45, 67, 88, 100]
print("Found at index:", search_infinite(arr, 31))
```

Output:

```
Found at index: 6
```

## Why It Matters

- Works with streams, linked storage, APIs, or infinite generators
- Avoids full traversal, logarithmic growth
- Combines exploration (finding bounds) with binary search (exact match)
- Ideal for search engines, log readers, cloud data paging

## Try It Yourself

1. Search [1, 3, 5, 9, 12, 20] for 9
2. Search [2, 4, 8, 16, 32, 64] for 33 (not found)
3. Count number of `get()` calls, compare to linear search
4. Try `target` smaller than first element
5. Handle edge cases: empty array, target > max
6. Simulate infinite stream with `get()`
7. Replace doubling with `high = low + step` for adaptive growth
8. Visualize search window expansion
9. Generalize to descending arrays
10. Compare performance vs naive scan

## Test Cases

| Input Array                   | Target | Output |
|-------------------------------|--------|--------|
| [3, 5, 9, 12, 17, 23, 31, 45] | 31     | 6      |
| [3, 5, 9, 12, 17, 23, 31, 45] | 4      | -1     |
| [1, 2, 4, 8, 16]              | 8      | 3      |
| [10, 20, 30]                  | 40     | -1     |

## Complexity

| Operation | Time        | Space  |
|-----------|-------------|--------|
| Search    | $O(\log p)$ | $O(1)$ |

$p$  = position of target in array

Searching an infinite array feels like navigating a foggy road, first find your headlights (bounds), then drive straight to the target.

# Chapter 3. Data Structure in Action

## Section 21. Arrays, linked lists, stacks, queues

### 201 Dynamic Array Resize

Dynamic array resizing is how we keep push operations fast while growing storage as needed. The idea is simple: when the array is full, allocate a bigger one, copy the elements, and keep going. If we double the capacity on each resize, the average cost per push stays constant.

#### What Problem Are We Solving?

A fixed array has a fixed size. Real programs do not always know  $n$  in advance. We want an array that supports `push` at end with near constant time, grows automatically, and keeps elements in contiguous memory for cache friendliness.

Goal: Provide `push`, `pop`, `get`, `set` on a resizable array with amortized  $O(1)$  time for `push`.

#### How Does It Work (Plain Language)?

Keep two numbers: `size` and `capacity`.

1. Start with a small capacity (for example, 1 or 8).
2. On `push`, if `size < capacity`, write the element and increase `size`.
3. If `size == capacity`, allocate new storage with capacity doubled, copy old elements, release the old block, then push.
4. Optionally, if many `pops` reduce `size` far below `capacity`, shrink by halving to avoid wasted space.

Why doubling? Doubling keeps the number of costly resizes small. Most pushes are cheap writes. Only occasionally do we pay for copying.

Example Steps (Growth Simulation)

| Step | Size Before | Capacity Before | Action                                          | Size After | Capacity After |
|------|-------------|-----------------|-------------------------------------------------|------------|----------------|
| 1    | 0           | 1               | push(1)                                         | 1          | 1              |
| 2    | 1           | 1               | full → resize to 2, copy 1 element,<br>push(2)  | 2          | 2              |
| 3    | 2           | 2               | full → resize to 4, copy 2 elements,<br>push(3) | 3          | 4              |
| 4    | 3           | 4               | push(4)                                         | 4          | 4              |
| 5    | 4           | 4               | full → resize to 8, copy 4 elements,<br>push(5) | 5          | 8              |

Notice how capacity doubles occasionally while most pushes cost O(1).

### Tiny Code (Easy Versions)

C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
    size_t size;
    size_t capacity;
    int *data;
} DynArray;

int da_init(DynArray *a, size_t init_cap) {
    if (init_cap == 0) init_cap = 1;
    a->data = (int *)malloc(init_cap * sizeof(int));
    if (!a->data) return 0;
    a->size = 0;
    a->capacity = init_cap;
    return 1;
}

int da_resize(DynArray *a, size_t new_cap) {
    int *p = (int *)realloc(a->data, new_cap * sizeof(int));
    if (!p) return 0;
    a->data = p;
    a->capacity = new_cap;
}
```

```

        return 1;
    }

int da_push(DynArray *a, int x) {
    if (a->size == a->capacity) {
        size_t new_cap = a->capacity * 2;
        if (!da_resize(a, new_cap)) return 0;
    }
    a->data[a->size++] = x;
    return 1;
}

int da_pop(DynArray *a, int *out) {
    if (a->size == 0) return 0;
    *out = a->data[--a->size];
    if (a->capacity > 1 && a->size <= a->capacity / 4) {
        size_t new_cap = a->capacity / 2;
        if (new_cap == 0) new_cap = 1;
        da_resize(a, new_cap);
    }
    return 1;
}

int main(void) {
    DynArray a;
    if (!da_init(&a, 1)) return 1;
    for (int i = 0; i < 10; ++i) da_push(&a, i);
    for (size_t i = 0; i < a.size; ++i) printf("%d ", a.data[i]);
    printf("\nsize=%zu cap=%zu\n", a.size, a.capacity);
    free(a.data);
    return 0;
}

```

Python

```

class DynArray:
    def __init__(self, init_cap=1):
        self._cap = max(1, init_cap)
        self._n = 0
        self._data = [None] * self._cap

    def _resize(self, new_cap):

```

```

new = [None] * new_cap
for i in range(self._n):
    new[i] = self._data[i]
self._data = new
self._cap = new_cap

def push(self, x):
    if self._n == self._cap:
        self._resize(self._cap * 2)
    self._data[self._n] = x
    self._n += 1

def pop(self):
    if self._n == 0:
        raise IndexError("pop from empty DynArray")
    self._n -= 1
    x = self._data[self._n]
    self._data[self._n] = None
    if self._cap > 1 and self._n <= self._cap // 4:
        self._resize(max(1, self._cap // 2))
    return x

def __getitem__(self, i):
    if not 0 <= i < self._n:
        raise IndexError("index out of range")
    return self._data[i]

```

## Why It Matters

- Amortized O(1) push for dynamic growth
- Contiguous memory for cache performance
- Simple API that underlies vectors, array lists, and many scripting language arrays
- Balanced memory usage via optional shrinking

## A Gentle Proof (Why It Works)

Accounting view for doubling Charge each push a constant credit (e.g. 3 units).

1. A normal push costs 1 unit and stores 2 credits with the element.
2. When resizing from capacity  $k \rightarrow 2k$ , copying  $k$  elements costs  $k$  units, paid by saved credits.

- 3. The new element pays its own 1 unit.

Thus amortized cost per push is  $O(1)$ .

Growth factor choice Any factor  $> 1$  gives amortized  $O(1)$ .

- Doubling  $\rightarrow$  fewer resizes, more memory slack
- 1.5x  $\rightarrow$  less slack, more frequent copies
- Too small ( $< 1.2$ )  $\rightarrow$  breaks amortized bound

### Try It Yourself

1. Simulate pushes 1..32 for factors 2.0 and 1.5. Count resizes.
2. Add `reserve(n)` to preallocate capacity.
3. Implement shrinking when `size <= capacity / 4`.
4. Replace `int` with a struct, measure copy cost.
5. Use potential method:  $\Phi = 2 \cdot \text{size} - \text{capacity}$ .

### Test Cases

| Operation Sequence | Capacity Trace (Factor 2)                         | Notes               |
|--------------------|---------------------------------------------------|---------------------|
| push 1..8          | 1 $\rightarrow$ 2 $\rightarrow$ 4 $\rightarrow$ 8 | resize at 1, 2, 4   |
| push 9             | 8 $\rightarrow$ 16                                | resize before write |
| push 10..16        | 16                                                | no resize           |
| pop 16..9          | 16                                                | shrinking optional  |
| pop 8              | 16 $\rightarrow$ 8                                | shrink at 25% load  |

### Edge Cases

- Push on full array triggers one resize before write
- Pop on empty array should error or return false
- `reserve(n)` larger than current capacity must preserve data
- Shrink never reduces capacity below size

### Complexity

- Time:
  - push amortized  $O(1)$
  - push worst case  $O(n)$  on resize
  - pop amortized  $O(1)$

- get/set O(1)
- Space: O(n), capacity constant  $\times$  size

Dynamic array resizing turns a rigid array into a flexible container. Grow when needed, copy occasionally, and enjoy constant-time pushes on average.

## 202 Circular Array Implementation

A circular array (or ring buffer) stores elements in a fixed-size array while allowing wrap-around indexing. It is perfect for implementing queues and buffers where old data is overwritten or processed in a first-in-first-out manner.

### What Problem Are We Solving?

A regular array wastes space if we only move front and rear pointers forward. A circular array solves this by wrapping indices around when they reach the end, so all slots can be reused without shifting elements.

Goal: Efficiently support enqueue, dequeue, peek, and size in O(1) time using a fixed-size buffer with wrap-around indexing.

### How Does It Work (Plain Language)?

Maintain:

- **front**: index of the first element
- **rear**: index of the next free slot
- **count**: number of elements in the buffer
- **capacity**: maximum number of elements

Wrap-around indexing uses modulo arithmetic:

```
next_index = (current_index + 1) % capacity
```

When adding or removing, always increment **front** or **rear** with this rule.

Example Steps (Wrap-around Simulation)

| Step | Operation   | Front | Rear | Count | Array State  | Note          |
|------|-------------|-------|------|-------|--------------|---------------|
| 1    | enqueue(10) | 0     | 1    | 1     | [10, , , ]   | rear advanced |
| 2    | enqueue(20) | 0     | 2    | 2     | [10, 20, , ] |               |

| Step | Operation   | Front | Rear | Count | Array State      | Note           |
|------|-------------|-------|------|-------|------------------|----------------|
| 3    | enqueue(30) | 0     | 3    | 3     | [10, 20, 30, _]  |                |
| 4    | dequeue()   | 1     | 3    | 2     | [, 20, 30, ]     | front advanced |
| 5    | enqueue(40) | 1     | 0    | 3     | [40, 20, 30, _]  | wrap-around    |
| 6    | enqueue(50) | 1     | 1    | 4     | [40, 20, 30, 50] | full queue     |

### Tiny Code (Easy Versions)

C

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int *data;
    int front;
    int rear;
    int count;
    int capacity;
} CircularQueue;

int cq_init(CircularQueue *q, int capacity) {
    q->data = malloc(capacity * sizeof(int));
    if (!q->data) return 0;
    q->capacity = capacity;
    q->front = 0;
    q->rear = 0;
    q->count = 0;
    return 1;
}

int cq_enqueue(CircularQueue *q, int x) {
    if (q->count == q->capacity) return 0; // full
    q->data[q->rear] = x;
    q->rear = (q->rear + 1) % q->capacity;
    q->count++;
    return 1;
}

int cq_dequeue(CircularQueue *q, int *out) {
    if (q->count == 0) return 0; // empty
    *out = q->data[q->front];
    q->front = (q->front + 1) % q->capacity;
    q->count--;
    return 1;
}
```

```

*out = q->data[q->front];
q->front = (q->front + 1) % q->capacity;
q->count--;
return 1;
}

int main(void) {
CircularQueue q;
cq_init(&q, 4);
cq_enqueue(&q, 10);
cq_enqueue(&q, 20);
cq_enqueue(&q, 30);
int val;
cq_dequeue(&q, &val);
cq_enqueue(&q, 40);
cq_enqueue(&q, 50); // should fail if capacity=4
printf("Front value: %d\n", q.data[q.front]);
free(q.data);
return 0;
}

```

Python

```

class CircularQueue:
    def __init__(self, capacity):
        self._cap = capacity
        self._data = [None] * capacity
        self._front = 0
        self._rear = 0
        self._count = 0

    def enqueue(self, x):
        if self._count == self._cap:
            raise OverflowError("queue full")
        self._data[self._rear] = x
        self._rear = (self._rear + 1) % self._cap
        self._count += 1

    def dequeue(self):
        if self._count == 0:
            raise IndexError("queue empty")
        x = self._data[self._front]

```

```

        self._front = (self._front + 1) % self._cap
        self._count -= 1
    return x

def peek(self):
    if self._count == 0:
        raise IndexError("queue empty")
    return self._data[self._front]

```

## Why It Matters

- Enables constant time queue operations
- No element shifting needed
- Efficient use of fixed memory
- Backbone for circular buffers, task schedulers, streaming pipelines, and audio/video buffers

## A Gentle Proof (Why It Works)

Modulo indexing ensures positions loop back when reaching array end. If capacity is n, then all indices stay in [0, n-1]. No overflow occurs since `front`, `rear` always wrap around. The condition `count == capacity` detects full queue, and `count == 0` detects empty.

Thus each operation touches only one element and updates O(1) variables.

## Try It Yourself

1. Implement a circular buffer with overwrite (old data replaced).
2. Add `is_full()` and `is_empty()` helpers.
3. Simulate producer-consumer with one enqueue per producer tick and one dequeue per consumer tick.
4. Extend to store structs instead of integers.

## Test Cases

| Operation Sequence | Front | Rear | Count | Array State    | Notes       |
|--------------------|-------|------|-------|----------------|-------------|
| enqueue(10)        | 0     | 1    | 1     | [10, , , ]     | normal push |
| enqueue(20)        | 0     | 2    | 2     | [10, 20, , ]   |             |
| enqueue(30)        | 0     | 3    | 3     | [10, 20, 30, ] |             |

| Operation Sequence | Front | Rear | Count | Array State      | Notes       |
|--------------------|-------|------|-------|------------------|-------------|
| dequeue()          | 1     | 3    | 2     | [, 20, 30, ]     |             |
| enqueue(40)        | 1     | 0    | 3     | [40, 20, 30, _]  | wrap-around |
| enqueue(50)        | 1     | 1    | 4     | [40, 20, 30, 50] | full        |

### Edge Cases

- Enqueue when full → error or overwrite
- Dequeue when empty → error
- Modulo indexing avoids overflow
- Works with any capacity 1

### Complexity

- Time: O(1) for enqueue, dequeue, peek
- Space: O(n) fixed buffer

Circular arrays provide elegant constant-time queues with wrap-around indexing, a small trick that powers big systems.

## 203 Singly Linked List Insert/Delete

A singly linked list is a chain of nodes, each pointing to the next. It grows and shrinks dynamically without preallocating memory. Operations like insert and delete rely on pointer adjustments rather than shifting elements.

### What Problem Are We Solving?

Static arrays are fixed-size and require shifting elements for insertions or deletions in the middle. Singly linked lists solve this by connecting elements through pointers, allowing efficient O(1) insertion and deletion (given a node reference).

Goal: Support insert, delete, and traversal on a structure that can grow dynamically without reallocating or shifting elements.

## How Does It Work (Plain Language)?

Each node stores two fields:

- **data** (the value)
- **next** (pointer to the next node)

The list has a **head** pointer to the first node. Insertion and deletion work by updating the **next** pointers.

Example Steps (Insertion at Position)

| Step | Operation           | Node     |                   |                   | Notes                  |
|------|---------------------|----------|-------------------|-------------------|------------------------|
|      |                     | Updated  | Before            | After             |                        |
| 1    | create list         | -        | empty             | head =<br>NULL    | list starts empty      |
| 2    | insert(10) at head  | new node | NULL              | [10]              | head → 10              |
| 3    | insert(20) at head  | new node | [10]              | [20 → 10]         | head updated           |
| 4    | insert(30) after 20 | new node | [20 → 10]         | [20 → 30 →<br>10] | pointer rerouted       |
| 5    | delete(30)          | node 20  | [20 → 30 →<br>10] | [20 → 10]         | bypass removed<br>node |

Key idea: To insert, point the new node's **next** to the target's next, then link the target's **next** to the new node. To delete, bypass the target node by linking previous node's **next** to the one after target.

## Tiny Code (Easy Versions)

C

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    int data;
    struct Node *next;
} Node;

Node* insert_head(Node* head, int value) {
    Node* new_node = malloc(sizeof(Node));
    new_node->data = value;
```

```

    new_node->next = head;
    return new_node; // new head
}

Node* insert_after(Node* node, int value) {
    if (!node) return NULL;
    Node* new_node = malloc(sizeof(Node));
    new_node->data = value;
    new_node->next = node->next;
    node->next = new_node;
    return new_node;
}

Node* delete_value(Node* head, int value) {
    if (!head) return NULL;
    if (head->data == value) {
        Node* tmp = head->next;
        free(head);
        return tmp;
    }
    Node* prev = head;
    Node* cur = head->next;
    while (cur) {
        if (cur->data == value) {
            prev->next = cur->next;
            free(cur);
            break;
        }
        prev = cur;
        cur = cur->next;
    }
    return head;
}

void print_list(Node* head) {
    for (Node* p = head; p; p = p->next)
        printf("%d -> ", p->data);
    printf("NULL\n");
}

int main(void) {
    Node* head = NULL;
}

```

```

head = insert_head(head, 10);
head = insert_head(head, 20);
insert_after(head, 30);
print_list(head);
head = delete_value(head, 30);
print_list(head);
return 0;
}

```

Python

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def insert_head(self, data):
        node = Node(data)
        node.next = self.head
        self.head = node

    def insert_after(self, prev, data):
        if prev is None:
            return
        node = Node(data)
        node.next = prev.next
        prev.next = node

    def delete_value(self, value):
        cur = self.head
        prev = None
        while cur:
            if cur.data == value:
                if prev:
                    prev.next = cur.next
                else:
                    self.head = cur.next
            return

```

```

    prev = cur
    cur = cur.next

def print_list(self):
    cur = self.head
    while cur:
        print(cur.data, end=" -> ")
        cur = cur.next
    print("NULL")

```

## Why It Matters

- Provides dynamic growth with no preallocation
- Enables O(1) insertion and deletion at head or given position
- Useful in stacks, queues, hash table buckets, and adjacency lists
- Foundation for more complex data structures (trees, graphs)

## A Gentle Proof (Why It Works)

Each operation only changes a constant number of pointers.

- Insert: 2 assignments → O(1)
- Delete: find node O(n), then 1 reassignment Because pointers are updated locally, the rest of the list remains valid.

## Try It Yourself

1. Implement `insert_tail` for O(n) tail insertion.
2. Write `reverse()` to flip the list in place.
3. Add `size()` to count nodes.
4. Experiment with deleting from head and middle.

## Test Cases

| Operation                           | Input          | Result         | Notes           |
|-------------------------------------|----------------|----------------|-----------------|
| <code>insert_head(10)</code>        | []             | [10]           | creates head    |
| <code>insert_head(20)</code>        | [10]           | [20 → 10]      | new head        |
| <code>insert_after(head, 30)</code> | [20 → 10]      | [20 → 30 → 10] | pointer reroute |
| <code>delete_value(30)</code>       | [20 → 30 → 10] | [20 → 10]      | node removed    |

| Operation        | Input     | Result    | Notes     |
|------------------|-----------|-----------|-----------|
| delete_value(40) | [20 → 10] | [20 → 10] | no change |

### Edge Cases

- Delete from empty list → no effect
- Insert after NULL → no effect
- Delete head node → update head pointer

### Complexity

- Time: insert O(1), delete O(n) (if searching by value), traversal O(n)
- Space: O(n) for n nodes

Singly linked lists teach pointer manipulation, where structure grows one node at a time, and every link matters.

## 204 Doubly Linked List Insert/Delete

A doubly linked list extends the singly linked list by adding backward links. Each node points to both its previous and next neighbor, making insertions and deletions easier in both directions.

### What Problem Are We Solving?

Singly linked lists cannot traverse backward and deleting a node requires access to its predecessor. Doubly linked lists fix this by storing two pointers per node, allowing constant-time insertions and deletions at any position when you have a node reference.

Goal: Support bidirectional traversal and efficient local insert/delete operations with O(1) pointer updates.

### How Does It Work (Plain Language)?

Each node stores:

- **data**: the value
- **prev**: pointer to the previous node
- **next**: pointer to the next node

The list tracks two ends:

- `head` points to the first node
- `tail` points to the last node

Example Steps (Insertion and Deletion)

| Step | Operation        | Target | Before     | After      | Note               |
|------|------------------|--------|------------|------------|--------------------|
| 1    | create list      | -      | empty      | [10]       | head=tail=10       |
| 2    | insert_front(20) | head   | [10]       | [20 10]    | head updated       |
| 3    | insert_back(30)  | tail   | [20 10]    | [20 10 30] | tail updated       |
| 4    | delete(10)       | middle | [20 10 30] | [20 30]    | pointers bypass 10 |

Each operation only touches nearby nodes, no need to shift elements.

### Tiny Code (Easy Versions)

C

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    int data;
    struct Node* prev;
    struct Node* next;
} Node;

Node* insert_front(Node* head, int value) {
    Node* node = malloc(sizeof(Node));
    node->data = value;
    node->prev = NULL;
    node->next = head;
    if (head) head->prev = node;
    return node;
}

Node* insert_back(Node* head, int value) {
    Node* node = malloc(sizeof(Node));
    node->data = value;
    node->next = NULL;
    if (!head) {
        node->prev = NULL;
    }
    else {
        node->prev = head->prev;
        head->prev->next = node;
        head = node;
    }
    return head;
}
```

```

        return node;
    }

Node* tail = head;
while (tail->next) tail = tail->next;
tail->next = node;
node->prev = tail;
return head;
}

Node* delete_value(Node* head, int value) {
    Node* cur = head;
    while (cur && cur->data != value) cur = cur->next;
    if (!cur) return head; // not found
    if (cur->prev) cur->prev->next = cur->next;
    else head = cur->next; // deleting head
    if (cur->next) cur->next->prev = cur->prev;
    free(cur);
    return head;
}

void print_forward(Node* head) {
    for (Node* p = head; p; p = p->next) printf("%d ", p->data);
    printf("NULL\n");
}

```

Python

```

class Node:
    def __init__(self, data):
        self.data = data
        self.prev = None
        self.next = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None

    def insert_front(self, data):
        node = Node(data)
        node.next = self.head
        if self.head:
            self.head.prev = node

```

```

        self.head = node

    def insert_back(self, data):
        node = Node(data)
        if not self.head:
            self.head = node
            return
        cur = self.head
        while cur.next:
            cur = cur.next
        cur.next = node
        node.prev = cur

    def delete_value(self, value):
        cur = self.head
        while cur:
            if cur.data == value:
                if cur.prev:
                    cur.prev.next = cur.next
                else:
                    self.head = cur.next
                if cur.next:
                    cur.next.prev = cur.prev
                return
            cur = cur.next

    def print_forward(self):
        cur = self.head
        while cur:
            print(cur.data, end="   ")
            cur = cur.next
        print("NULL")

```

## Why It Matters

- Bidirectional traversal (forward/backward)
- O(1) insertion and deletion when node is known
- Foundation for deque, LRU cache, and text editor buffers
- Enables clean list reversal and splice operations

## A Gentle Proof (Why It Works)

Each node update modifies at most four pointers:

- Insert: new node's `next`, `prev` + neighbor links
- Delete: predecessor's `next`, successor's `prev` Since each step is constant work, operations are  $O(1)$  given node reference.

Traversal still costs  $O(n)$ , but local edits are efficient.

## Try It Yourself

1. Implement `reverse()` by swapping `prev` and `next` pointers.
2. Add tail pointer to allow  $O(1)$  `insert_back`.
3. Support bidirectional iteration.
4. Implement `pop_front()` and `pop_back()`.

## Test Cases

| Operation                     | Input      | Output     | Notes          |
|-------------------------------|------------|------------|----------------|
| <code>insert_front(10)</code> | []         | [10]       | head=tail=10   |
| <code>insert_front(20)</code> | [10]       | [20 10]    | new head       |
| <code>insert_back(30)</code>  | [20 10]    | [20 10 30] | new tail       |
| <code>delete_value(10)</code> | [20 10 30] | [20 30]    | middle removed |
| <code>delete_value(20)</code> | [20 30]    | [30]       | head removed   |

## Edge Cases

- Deleting from empty list → no effect
- Inserting on empty list sets both head and tail
- Properly maintain both directions after each update

## Complexity

- Time:
  - Insert/Delete (given node):  $O(1)$
  - Search:  $O(n)$
  - Traverse:  $O(n)$
- Space:  $O(n)$  (2 pointers per node)

Doubly linked lists bring symmetry, move forward, move back, and edit in constant time.

## 205 Stack Push/Pop

A stack is a simple but powerful data structure that follows the LIFO (Last In, First Out) rule. The most recently added element is the first to be removed, like stacking plates where you can only take from the top.

### What Problem Are We Solving?

We often need to reverse order or track nested operations, such as function calls, parentheses, undo/redo, and expression evaluation. A stack gives us a clean way to manage this behavior with push (add) and pop (remove).

Goal: Support push, pop, peek, and is\_empty in O(1) time, maintaining LIFO order.

### How Does It Work (Plain Language)?

Think of a vertical pile.

- Push(x): Place x on top.
- Pop(): Remove the top element.
- Peek(): View top without removing.

Implementation can use either an array (fixed or dynamic) or a linked list.

Example Steps (Array Stack Simulation)

| Step | Operation | Stack (Top → Bottom) | Note          |
|------|-----------|----------------------|---------------|
| 1    | push(10)  | [10]                 | first element |
| 2    | push(20)  | [20, 10]             | top is 20     |
| 3    | push(30)  | [30, 20, 10]         |               |
| 4    | pop()     | [20, 10]             | 30 removed    |
| 5    | peek()    | top = 20             | top unchanged |

### Tiny Code (Easy Versions)

C (Array-Based Stack)

```

#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int *data;
    int top;
    int capacity;
} Stack;

int stack_init(Stack *s, int cap) {
    s->data = malloc(cap * sizeof(int));
    if (!s->data) return 0;
    s->capacity = cap;
    s->top = -1;
    return 1;
}

int stack_push(Stack *s, int x) {
    if (s->top + 1 == s->capacity) return 0; // full
    s->data[++s->top] = x;
    return 1;
}

int stack_pop(Stack *s, int *out) {
    if (s->top == -1) return 0; // empty
    *out = s->data[s->top--];
    return 1;
}

int stack_peek(Stack *s, int *out) {
    if (s->top == -1) return 0;
    *out = s->data[s->top];
    return 1;
}

int main(void) {
    Stack s;
    stack_init(&s, 5);
    stack_push(&s, 10);
    stack_push(&s, 20);
    int val;
    stack_pop(&s, &val);
}

```

```

printf("Popped: %d\n", val);
stack_peek(&s, &val);
printf("Top: %d\n", val);
free(s.data);
return 0;
}

```

Python (List as Stack)

```

class Stack:
    def __init__(self):
        self._data = []

    def push(self, x):
        self._data.append(x)

    def pop(self):
        if not self._data:
            raise IndexError("pop from empty stack")
        return self._data.pop()

    def peek(self):
        if not self._data:
            raise IndexError("peek from empty stack")
        return self._data[-1]

    def is_empty(self):
        return len(self._data) == 0

# Example
s = Stack()
s.push(10)
s.push(20)
print(s.pop()) # 20
print(s.peek()) # 10

```

## Why It Matters

- Core to recursion, parsing, and backtracking
- Underpins function call stacks in programming languages
- Natural structure for undo, reverse, and balanced parentheses
- Simplicity with wide applications in algorithms

## A Gentle Proof (Why It Works)

Each operation touches only the top element or index.

- **push**: increment top, write value
- **pop**: read top, decrement top All O(1) time. The LIFO property ensures correct reverse order for function calls and nested scopes.

## Try It Yourself

1. Implement stack with linked list nodes.
2. Extend capacity dynamically when full.
3. Use stack to check balanced parentheses in a string.
4. Reverse a list using stack operations.

## Test Cases

| Operation | Input    | Output                 | Notes           |
|-----------|----------|------------------------|-----------------|
| push(10)  | []       | [10]                   | top = 10        |
| push(20)  | [10]     | [20, 10]               | top = 20        |
| pop()     | [20, 10] | returns 20, stack [10] |                 |
| peek()    | [10]     | returns 10             |                 |
| pop()     | [10]     | returns 10, stack []   | empty after pop |

## Edge Cases

- Pop from empty → error or return false
- Peek from empty → error
- Overflow if array full (unless dynamic)

## Complexity

- Time: push O(1), pop O(1), peek O(1)
- Space: O(n) for n elements

Stacks embody disciplined memory, last in, first out, a minimalist model of control flow and history.

## 206 Queue Enqueue/Dequeue

A queue is the mirror twin of a stack. It follows FIFO (First In, First Out), the first element added is the first one removed, like people waiting in line.

### What Problem Are We Solving?

We need a structure where items are processed in arrival order: scheduling tasks, buffering data, breadth-first search, or managing print jobs. A queue lets us enqueue at the back and dequeue from the front, simple and fair.

Goal: Support enqueue, dequeue, peek, and `is_empty` in  $O(1)$  time using a circular layout or linked list.

### How Does It Work (Plain Language)?

A queue has two ends:

- `front` → where items are removed
- `rear` → where items are added

Operations:

- `enqueue(x)` adds at `rear`
- `dequeue()` removes from `front`
- `peek()` looks at the `front` item

If implemented with a circular array, wrap indices using modulo arithmetic.

### Example Steps (FIFO Simulation)

| Step | Operation                | Queue (Front → Rear) | Front | Rear | Note          |
|------|--------------------------|----------------------|-------|------|---------------|
| 1    | <code>enqueue(10)</code> | [10]                 | 0     | 1    | first element |
| 2    | <code>enqueue(20)</code> | [10, 20]             | 0     | 2    | rear advanced |
| 3    | <code>enqueue(30)</code> | [10, 20, 30]         | 0     | 3    |               |
| 4    | <code>dequeue()</code>   | [20, 30]             | 1     | 3    | 10 removed    |
| 5    | <code>enqueue(40)</code> | [20, 30, 40]         | 1     | 0    | wrap-around   |

## Tiny Code (Easy Versions)

C (Circular Array Queue)

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int *data;
    int front;
    int rear;
    int count;
    int capacity;
} Queue;

int queue_init(Queue *q, int cap) {
    q->data = malloc(cap * sizeof(int));
    if (!q->data) return 0;
    q->capacity = cap;
    q->front = 0;
    q->rear = 0;
    q->count = 0;
    return 1;
}

int enqueue(Queue *q, int x) {
    if (q->count == q->capacity) return 0; // full
    q->data[q->rear] = x;
    q->rear = (q->rear + 1) % q->capacity;
    q->count++;
    return 1;
}

int dequeue(Queue *q, int *out) {
    if (q->count == 0) return 0; // empty
    *out = q->data[q->front];
    q->front = (q->front + 1) % q->capacity;
    q->count--;
    return 1;
}

int queue_peek(Queue *q, int *out) {
    if (q->count == 0) return 0;
```

```

    *out = q->data[q->front];
    return 1;
}

int main(void) {
    Queue q;
    queue_init(&q, 4);
    enqueue(&q, 10);
    enqueue(&q, 20);
    enqueue(&q, 30);
    int val;
    dequeue(&q, &val);
    printf("Dequeued: %d\n", val);
    enqueue(&q, 40);
    enqueue(&q, 50); // will fail if capacity = 4
    queue_peek(&q, &val);
    printf("Front: %d\n", val);
    free(q.data);
    return 0;
}

```

Python (List as Queue using deque)

```

from collections import deque

class Queue:
    def __init__(self):
        self._data = deque()

    def enqueue(self, x):
        self._data.append(x)

    def dequeue(self):
        if not self._data:
            raise IndexError("dequeue from empty queue")
        return self._data.popleft()

    def peek(self):
        if not self._data:
            raise IndexError("peek from empty queue")
        return self._data[0]

```

```

def is_empty(self):
    return len(self._data) == 0

# Example
q = Queue()
q.enqueue(10)
q.enqueue(20)
print(q.dequeue()) # 10
print(q.peek())   # 20

```

## Why It Matters

- Enforces fairness (first come, first served)
- Foundation for BFS, schedulers, buffers, pipelines
- Easy to implement and reason about
- Natural counterpart to stack

## A Gentle Proof (Why It Works)

Each operation updates only front or rear index, not entire array. Circular indexing ensures constant-time wrap-around:

```

rear = (rear + 1) % capacity
front = (front + 1) % capacity

```

All operations touch O(1) data and fields, so runtime stays O(1).

## Try It Yourself

1. Implement a linked-list-based queue.
2. Add `is_full()` and `is_empty()` checks.
3. Write a queue-based BFS on a simple graph.
4. Compare linear vs circular queue behavior.

| Operation | Queue (Front → Rear) | Front | Rear | Count | Notes |
|-----------|----------------------|-------|------|-------|-------|
|-----------|----------------------|-------|------|-------|-------|

## Test Cases

| Operation   | Queue (Front → Rear) | Front | Rear | Count | Notes       |
|-------------|----------------------|-------|------|-------|-------------|
| enqueue(10) | [10]                 | 0     | 1    | 1     |             |
| enqueue(20) | [10, 20]             | 0     | 2    | 2     |             |
| enqueue(30) | [10, 20, 30]         | 0     | 3    | 3     |             |
| dequeue()   | [20, 30]             | 1     | 3    | 2     | removes 10  |
| enqueue(40) | [20, 30, 40]         | 1     | 0    | 3     | wrap-around |
| peek()      | front=20             | 1     | 0    | 3     | check front |

## Edge Cases

- Dequeue from empty → error
- Enqueue to full → overflow
- Works seamlessly with wrap-around

## Complexity

- Time: enqueue O(1), dequeue O(1), peek O(1)
- Space: O(n) for fixed buffer or dynamic growth

Queues bring fairness to data, what goes in first comes out first, steady and predictable.

## 207 Deque Implementation

A deque (double-ended queue) is a flexible container that allows adding and removing elements from both ends, a blend of stack and queue behavior.

### What Problem Are We Solving?

Stacks restrict you to one end, queues to two fixed roles. Sometimes we need both: insert at front or back, pop from either side. Deques power sliding window algorithms, palindrome checks, undo-redo systems, and task schedulers.

Goal: Support `push_front`, `push_back`, `pop_front`, `pop_back`, and `peek_front/back` in O(1) time.

## How Does It Work (Plain Language)?

Deques can be built using:

- A circular array (using wrap-around indexing)
- A doubly linked list (bidirectional pointers)

Operations:

- `push_front(x)` → insert before front
- `push_back(x)` → insert after rear
- `pop_front()` → remove front element
- `pop_back()` → remove rear element

Example Steps (Circular Array Simulation)

| Step | Operation                  | Front | Rear | Deque (Front → Rear) | Note              |
|------|----------------------------|-------|------|----------------------|-------------------|
| 1    | <code>push_back(10)</code> | 0     | 1    | [10]                 | first item        |
| 2    | <code>push_back(20)</code> | 0     | 2    | [10, 20]             | rear grows        |
| 3    | <code>push_front(5)</code> | 3     | 2    | [5, 10, 20]          | wrap-around front |
| 4    | <code>pop_back()</code>    | 3     | 1    | [5, 10]              | 20 removed        |
| 5    | <code>pop_front()</code>   | 0     | 1    | [10]                 | 5 removed         |

## Tiny Code (Easy Versions)

C (Circular Array Deque)

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int *data;
    int front;
    int rear;
    int count;
    int capacity;
} Deque;

int dq_init(Deque *d, int cap) {
    d->data = malloc(cap * sizeof(int));
    if (!d->data) return 0;
    d->capacity = cap;
```

```

d->front = 0;
d->rear = 0;
d->count = 0;
return 1;
}

int dq_push_front(Deque *d, int x) {
    if (d->count == d->capacity) return 0; // full
    d->front = (d->front - 1 + d->capacity) % d->capacity;
    d->data[d->front] = x;
    d->count++;
    return 1;
}

int dq_push_back(Deque *d, int x) {
    if (d->count == d->capacity) return 0;
    d->data[d->rear] = x;
    d->rear = (d->rear + 1) % d->capacity;
    d->count++;
    return 1;
}

int dq_pop_front(Deque *d, int *out) {
    if (d->count == 0) return 0;
    *out = d->data[d->front];
    d->front = (d->front + 1) % d->capacity;
    d->count--;
    return 1;
}

int dq_pop_back(Deque *d, int *out) {
    if (d->count == 0) return 0;
    d->rear = (d->rear - 1 + d->capacity) % d->capacity;
    *out = d->data[d->rear];
    d->count--;
    return 1;
}

int main(void) {
    Deque d;
    dq_init(&d, 4);
    dq_push_back(&d, 10);
}

```

```

dq_push_back(&d, 20);
dq_push_front(&d, 5);
int val;
dq_pop_back(&d, &val);
printf("Popped back: %d\n", val);
dq_pop_front(&d, &val);
printf("Popped front: %d\n", val);
free(d.data);
return 0;
}

```

Python (Deque using collections)

```

from collections import deque

d = deque()
d.append(10)           # push_back
d.appendleft(5)        # push_front
d.append(20)
print(d.pop())         # pop_back -> 20
print(d.popleft())    # pop_front -> 5
print(d)               # deque([10])

```

## Why It Matters

- Generalizes both stack and queue
- Core tool for sliding window maximum, palindrome checks, BFS with state, and task buffers
- Doubly-ended flexibility with constant-time operations
- Ideal for systems needing symmetric access

## A Gentle Proof (Why It Works)

Circular indexing ensures wrap-around in  $O(1)$ . Each operation moves one index and changes one value:

- push → set value + adjust index
- pop → adjust index + read value

No shifting or reallocation needed, so all operations remain  $O(1)$ .

## Try It Yourself

1. Implement deque with a doubly linked list.
2. Add `peek_front()` and `peek_back()`.
3. Simulate a sliding window maximum algorithm.
4. Compare performance of deque vs list for queue-like tasks.

## Test Cases

| Operation     | Front | Rear | Count | Deque (Front → Rear) | Notes      |
|---------------|-------|------|-------|----------------------|------------|
| push_back(10) | 0     | 1    | 1     | [10]                 | init       |
| push_back(20) | 0     | 2    | 2     | [10, 20]             |            |
| push_front(5) | 3     | 2    | 3     | [5, 10, 20]          | wrap front |
| pop_back()    | 3     | 1    | 2     | [5, 10]              | 20 removed |
| pop_front()   | 0     | 1    | 1     | [10]                 | 5 removed  |

## Edge Cases

- Push on full deque → error or resize
- Pop on empty deque → error
- Wrap-around correctness is critical

## Complexity

- Time: push/pop front/back  $O(1)$
- Space:  $O(n)$

Deques are the agile queues, you can act from either side, fast and fair.

## 208 Circular Queue

A circular queue is a queue optimized for fixed-size buffers where indices wrap around automatically. It's widely used in real-time systems, network packet buffers, and streaming pipelines to reuse space efficiently.

## What Problem Are We Solving?

A linear queue wastes space after several dequeues, since front indices move forward. A circular queue solves this by wrapping the indices, making every slot reusable.

Goal: Implement a queue with fixed capacity where enqueue and dequeue both take  $O(1)$  time and space is used cyclically.

## How Does It Work (Plain Language)?

A circular queue keeps track of:

- `front`: index of the first element
- `rear`: index of the next position to insert
- `count`: number of elements

Use modulo arithmetic for wrap-around:

```
next_index = (current_index + 1) % capacity
```

Key Conditions

- Full when `count == capacity`
- Empty when `count == 0`

Example Steps (Wrap-around Simulation)

| Step | Operation   | Front | Rear | Count | Queue State      | Note           |
|------|-------------|-------|------|-------|------------------|----------------|
| 1    | enqueue(10) | 0     | 1    | 1     | [10, , , ]       |                |
| 2    | enqueue(20) | 0     | 2    | 2     | [10, 20, , ]     |                |
| 3    | enqueue(30) | 0     | 3    | 3     | [10, 20, 30, ]   |                |
| 4    | dequeue()   | 1     | 3    | 2     | [, 20, 30, ]     | front advanced |
| 5    | enqueue(40) | 1     | 0    | 3     | [40, 20, 30, ]   | wrap-around    |
| 6    | enqueue(50) | 1     | 1    | 4     | [40, 20, 30, 50] | full           |

## Tiny Code (Easy Versions)

C

```

#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int *data;
    int front;
    int rear;
    int count;
    int capacity;
} CircularQueue;

int cq_init(CircularQueue *q, int cap) {
    q->data = malloc(cap * sizeof(int));
    if (!q->data) return 0;
    q->capacity = cap;
    q->front = 0;
    q->rear = 0;
    q->count = 0;
    return 1;
}

int cq_enqueue(CircularQueue *q, int x) {
    if (q->count == q->capacity) return 0; // full
    q->data[q->rear] = x;
    q->rear = (q->rear + 1) % q->capacity;
    q->count++;
    return 1;
}

int cq_dequeue(CircularQueue *q, int *out) {
    if (q->count == 0) return 0; // empty
    *out = q->data[q->front];
    q->front = (q->front + 1) % q->capacity;
    q->count--;
    return 1;
}

int cq_peek(CircularQueue *q, int *out) {
    if (q->count == 0) return 0;
    *out = q->data[q->front];
    return 1;
}

```

```

int main(void) {
    CircularQueue q;
    cq_init(&q, 4);
    cq_enqueue(&q, 10);
    cq_enqueue(&q, 20);
    cq_enqueue(&q, 30);
    int val;
    cq_dequeue(&q, &val);
    printf("Dequeued: %d\n", val);
    cq_enqueue(&q, 40);
    cq_enqueue(&q, 50); // should fail if full
    cq_peek(&q, &val);
    printf("Front: %d\n", val);
    free(q.data);
    return 0;
}

```

Python

```

class CircularQueue:
    def __init__(self, capacity):
        self._cap = capacity
        self._data = [None] * capacity
        self._front = 0
        self._rear = 0
        self._count = 0

    def enqueue(self, x):
        if self._count == self._cap:
            raise OverflowError("Queue full")
        self._data[self._rear] = x
        self._rear = (self._rear + 1) % self._cap
        self._count += 1

    def dequeue(self):
        if self._count == 0:
            raise IndexError("Queue empty")
        x = self._data[self._front]
        self._front = (self._front + 1) % self._cap
        self._count -= 1
        return x

```

```

def peek(self):
    if self._count == 0:
        raise IndexError("Queue empty")
    return self._data[self._front]

# Example
q = CircularQueue(4)
q.enqueue(10)
q.enqueue(20)
q.enqueue(30)
print(q.dequeue()) # 10
q.enqueue(40)
print(q.peek())    # 20

```

## Why It Matters

- Efficient space reuse, no wasted slots
- Predictable memory usage for real-time systems
- Backbone of buffering systems (audio, network, streaming)
- Fast O(1) operations, no shifting elements

## A Gentle Proof (Why It Works)

Since both `front` and `rear` wrap using modulo, all operations remain in range `[0, capacity - 1]`. Each operation modifies a fixed number of variables. Thus:

- enqueue: 1 write + 2 updates
- dequeue: 1 read + 2 updates No shifting, no resizing → all O(1) time.

## Try It Yourself

1. Add `is_full()` and `is_empty()` helpers.
2. Implement an overwrite mode where new enqueues overwrite oldest data.
3. Visualize index movement for multiple wraps.
4. Use a circular queue to simulate a producer-consumer buffer.

## Test Cases

| Operation   | Front | Rear | Count | Queue (Front → Rear) | Notes       |
|-------------|-------|------|-------|----------------------|-------------|
| enqueue(10) | 0     | 1    | 1     | [10, , , ]           |             |
| enqueue(20) | 0     | 2    | 2     | [10, 20, , ]         |             |
| enqueue(30) | 0     | 3    | 3     | [10, 20, 30, ]       |             |
| dequeue()   | 1     | 3    | 2     | [, 20, 30, ]         | 10 removed  |
| enqueue(40) | 1     | 0    | 3     | [40, 20, 30, ]       | wrap-around |
| enqueue(50) | 1     | 1    | 4     | [40, 20, 30, 50]     | full queue  |

### Edge Cases

- Enqueue when full → reject or overwrite
- Dequeue when empty → error
- Wrap-around indexing must handle 0 correctly

### Complexity

- Time: enqueue O(1), dequeue O(1), peek O(1)
- Space: O(n) fixed buffer

Circular queues are the heartbeat of real-time data flows, steady, cyclic, and never wasting a byte.

## 209 Stack via Queue

A stack via queue is a playful twist, implementing LIFO behavior using FIFO tools. It shows how one structure can simulate another by combining basic operations cleverly.

### What Problem Are We Solving?

Sometimes we're limited to queue operations (`enqueue`, `dequeue`) but still want a stack's last-in-first-out order. We can simulate push and pop using one or two queues.

Goal: Build a stack that supports `push`, `pop`, and `peek` in O(1) or O(n) time (depending on strategy) using only queue operations.

## How Does It Work (Plain Language)?

Two main strategies:

1. Push costly: rotate elements after every push so front is always top.
2. Pop costly: enqueue normally, but rotate during pop.

We'll show push costly version, simpler conceptually.

Idea: Each `push` enqueues new item, then rotates all older elements behind it, so that last pushed is always at the front (ready to pop).

Example Steps (Push Costly)

| Step | Operation             | Queue (Front → Rear) | Note                     |
|------|-----------------------|----------------------|--------------------------|
| 1    | <code>push(10)</code> | [10]                 | only one element         |
| 2    | <code>push(20)</code> | [20, 10]             | rotated so 20 is front   |
| 3    | <code>push(30)</code> | [30, 20, 10]         | rotated again            |
| 4    | <code>pop()</code>    | [20, 10]             | 30 removed               |
| 5    | <code>push(40)</code> | [40, 20, 10]         | rotation maintains order |

## Tiny Code (Easy Versions)

C (Using Two Queues)

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int *data;
    int front, rear, count, capacity;
} Queue;

int q_init(Queue *q, int cap) {
    q->data = malloc(cap * sizeof(int));
    if (!q->data) return 0;
    q->front = 0;
    q->rear = 0;
    q->count = 0;
    q->capacity = cap;
    return 1;
}
```

```

int q_enqueue(Queue *q, int x) {
    if (q->count == q->capacity) return 0;
    q->data[q->rear] = x;
    q->rear = (q->rear + 1) % q->capacity;
    q->count++;
    return 1;
}

int q_dequeue(Queue *q, int *out) {
    if (q->count == 0) return 0;
    *out = q->data[q->front];
    q->front = (q->front + 1) % q->capacity;
    q->count--;
    return 1;
}

typedef struct {
    Queue q1, q2;
} StackViaQueue;

int svq_init(StackViaQueue *s, int cap) {
    return q_init(&s->q1, cap) && q_init(&s->q2, cap);
}

int svq_push(StackViaQueue *s, int x) {
    q_enqueue(&s->q2, x);
    int val;
    while (s->q1.count) {
        q_dequeue(&s->q1, &val);
        q_enqueue(&s->q2, val);
    }
    // swap q1 and q2
    Queue tmp = s->q1;
    s->q1 = s->q2;
    s->q2 = tmp;
    return 1;
}

int svq_pop(StackViaQueue *s, int *out) {
    return q_dequeue(&s->q1, out);
}

```

```

int svq_peek(StackViaQueue *s, int *out) {
    if (s->q1.count == 0) return 0;
    *out = s->q1.data[s->q1.front];
    return 1;
}

int main(void) {
    StackViaQueue s;
    svq_init(&s, 10);
    svq_push(&s, 10);
    svq_push(&s, 20);
    svq_push(&s, 30);
    int val;
    svq_pop(&s, &val);
    printf("Popped: %d\n", val);
    svq_peek(&s, &val);
    printf("Top: %d\n", val);
    free(s.q1.data);
    free(s.q2.data);
    return 0;
}

```

Python

```

from collections import deque

class StackViaQueue:
    def __init__(self):
        self.q = deque()

    def push(self, x):
        n = len(self.q)
        self.q.append(x)
        # rotate all older elements
        for _ in range(n):
            self.q.append(self.q.popleft())

    def pop(self):
        if not self.q:
            raise IndexError("pop from empty stack")
        return self.q.popleft()

```

```

def peek(self):
    if not self.q:
        raise IndexError("peek from empty stack")
    return self.q[0]

# Example
s = StackViaQueue()
s.push(10)
s.push(20)
s.push(30)
print(s.pop()) # 30
print(s.peek()) # 20

```

## Why It Matters

- Demonstrates duality of data structures (stack built on queue)
- Reinforces operation trade-offs (costly push vs costly pop)
- Great teaching example for algorithmic simulation
- Builds insight into complexity and resource usage

## A Gentle Proof (Why It Works)

In push costly approach:

- Each push rotates all previous elements behind the new one → new element becomes front.
- Pop simply dequeues from front → correct LIFO order.

So order is maintained: newest always exits first.

## Try It Yourself

1. Implement pop costly variant (push O(1), pop O(n)).
2. Add `is_empty()` helper.
3. Compare total number of operations for n pushes and pops.
4. Extend to generic types with structs or templates.

## Test Cases

| Operation | Queue (Front → Rear) | Notes          |
|-----------|----------------------|----------------|
| push(10)  | [10]                 | single element |
| push(20)  | [20, 10]             | rotated        |
| push(30)  | [30, 20, 10]         | rotated        |
| pop()     | [20, 10]             | returns 30     |
| peek()    | [20, 10]             | returns 20     |

## Edge Cases

- Pop/peek from empty → error
- Capacity reached → reject push
- Works even with single queue (rotation after push)

## Complexity

- Push:  $O(n)$
- Pop/Peek:  $O(1)$
- Space:  $O(n)$

Stack via queue proves constraints breed creativity, same data, different dance.

## 210 Queue via Stack

A queue via stack flips the story: build FIFO behavior using LIFO tools. It's a classic exercise in algorithmic inversion, showing how fundamental operations can emulate each other with clever order manipulation.

### What Problem Are We Solving?

Suppose you only have stacks (with `push`, `pop`, `peek`) but need queue behavior (with `enqueue`, `dequeue`). We want to process items in arrival order, first in, first out, even though stacks operate last in, first out.

Goal: Implement `enqueue`, `dequeue`, and `peek` for a queue using only stack operations.

## How Does It Work (Plain Language)?

Two stacks are enough:

- inbox: where we push new items (enqueue)
- outbox: where we pop old items (dequeue)

When dequeuing, if `outbox` is empty, we move all items from `inbox` to `outbox`, reversing their order so oldest items are on top.

This reversal step restores FIFO behavior.

Example Steps (Two-Stack Method)

| Step | Operation   | inbox (Top → Bottom) | outbox (Top → Bottom) | Note                |
|------|-------------|----------------------|-----------------------|---------------------|
| 1    | enqueue(10) | [10]                 | []                    |                     |
| 2    | enqueue(20) | [20, 10]             | []                    |                     |
| 3    | enqueue(30) | [30, 20, 10]         | []                    |                     |
| 4    | dequeue()   | []                   | [10, 20, 30]          | transfer + pop(10)  |
| 5    | enqueue(40) | [40]                 | [20, 30]              | mixed state         |
| 6    | dequeue()   | [40]                 | [30]                  | pop(20) from outbox |

## Tiny Code (Easy Versions)

C (Using Two Stacks)

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int *data;
    int top;
    int capacity;
} Stack;

int stack_init(Stack *s, int cap) {
    s->data = malloc(cap * sizeof(int));
    if (!s->data) return 0;
    s->top = -1;
    s->capacity = cap;
    return 1;
}
```

```

int stack_push(Stack *s, int x) {
    if (s->top + 1 == s->capacity) return 0;
    s->data[++s->top] = x;
    return 1;
}

int stack_pop(Stack *s, int *out) {
    if (s->top == -1) return 0;
    *out = s->data[s->top--];
    return 1;
}

int stack_peek(Stack *s, int *out) {
    if (s->top == -1) return 0;
    *out = s->data[s->top];
    return 1;
}

int stack_empty(Stack *s) {
    return s->top == -1;
}

typedef struct {
    Stack in, out;
} QueueViaStack;

int qvs_init(QueueViaStack *q, int cap) {
    return stack_init(&q->in, cap) && stack_init(&q->out, cap);
}

int qvs_enqueue(QueueViaStack *q, int x) {
    return stack_push(&q->in, x);
}

int qvs_shift(QueueViaStack *q) {
    int val;
    while (!stack_empty(&q->in)) {
        stack_pop(&q->in, &val);
        stack_push(&q->out, val);
    }
    return 1;
}

```

```

int qvs_dequeue(QueueViaStack *q, int *out) {
    if (stack_empty(&q->out)) qvs_shift(q);
    return stack_pop(&q->out, out);
}

int qvs_peek(QueueViaStack *q, int *out) {
    if (stack_empty(&q->out)) qvs_shift(q);
    return stack_peek(&q->out, out);
}

int main(void) {
    QueueViaStack q;
    qvs_init(&q, 10);
    qvs_enqueue(&q, 10);
    qvs_enqueue(&q, 20);
    qvs_enqueue(&q, 30);
    int val;
    qvs_dequeue(&q, &val);
    printf("Dequeued: %d\n", val);
    qvs_enqueue(&q, 40);
    qvs_peek(&q, &val);
    printf("Front: %d\n", val);
    free(q.in.data);
    free(q.out.data);
    return 0;
}

```

### Python (Two Stack Queue)

```

class QueueViaStack:
    def __init__(self):
        self.inbox = []
        self.outbox = []

    def enqueue(self, x):
        self.inbox.append(x)

    def dequeue(self):
        if not self.outbox:
            while self.inbox:
                self.outbox.append(self.inbox.pop())

```

```

    if not self.outbox:
        raise IndexError("dequeue from empty queue")
    return self.outbox.pop()

def peek(self):
    if not self.outbox:
        while self.inbox:
            self.outbox.append(self.inbox.pop())
    if not self.outbox:
        raise IndexError("peek from empty queue")
    return self.outbox[-1]

# Example
q = QueueViaStack()
q.enqueue(10)
q.enqueue(20)
q.enqueue(30)
print(q.dequeue()) # 10
q.enqueue(40)
print(q.peek()) # 20

```

## Why It Matters

- Demonstrates queue emulation with stack operations
- Core teaching example for data structure duality
- Helps in designing abstract interfaces under constraints
- Underpins some streaming and buffering systems

## A Gentle Proof (Why It Works)

Each transfer (`inbox → outbox`) reverses order once, restoring FIFO sequence.

- Enqueue pushes to inbox (LIFO)
- Dequeue pops from outbox (LIFO of reversed) So overall effect = FIFO

Transfers happen only when outbox is empty, so amortized cost per operation is O(1).

## Try It Yourself

1. Implement a single-stack recursive version.
2. Add `is_empty()` helper.

3. Measure amortized vs worst-case complexity.
4. Extend to generic data type.

## Test Cases

| Operation   | inbox<br>(Top→Bottom) | outbox<br>(Top→Bottom) | Result     | Notes          |
|-------------|-----------------------|------------------------|------------|----------------|
| enqueue(10) | [10]                  | []                     |            |                |
| enqueue(20) | [20, 10]              | []                     |            |                |
| enqueue(30) | [30, 20, 10]          | []                     |            |                |
| dequeue()   | []                    | [10, 20, 30]           | returns 10 | transfer + pop |
| enqueue(40) | [40]                  | [20, 30]               |            |                |
| dequeue()   | [40]                  | [30]                   | returns 20 |                |

## Edge Cases

- Dequeue from empty queue → error
- Multiple dequeues trigger one transfer
- Outbox reused efficiently

## Complexity

- Time: amortized  $O(1)$  per operation (worst-case  $O(n)$  on transfer)
- Space:  $O(n)$

Queue via stack shows symmetry, turn LIFO into FIFO with one clever reversal.

## Section 22. Hash Tables and Variants

### 211 Hash Table Insertion

A hash table stores key-value pairs for lightning-fast lookups. It uses a hash function to map keys to array indices, letting us access data in near-constant time.

## What Problem Are We Solving?

We need a data structure that can insert, search, and delete by key efficiently, without scanning every element. Arrays give random access by index; hash tables extend that power to arbitrary keys.

Goal: Map each key to a slot via a hash function and resolve any collisions gracefully.

## How Does It Work (Plain Language)?

A hash table uses a hash function to convert a key into an index:

```
index = hash(key) % capacity
```

When inserting a new (key, value):

1. Compute hash index.
2. If slot is empty → place pair there.
3. If occupied → handle collision (chaining or open addressing).

We'll use separate chaining (linked list per slot) as the simplest method.

Example Steps (Separate Chaining)

| Step | Key      | Hash(key) | Index | Action                       |
|------|----------|-----------|-------|------------------------------|
| 1    | “apple”  | 42        | 2     | Insert (“apple”, 10)         |
| 2    | “banana” | 15        | 3     | Insert (“banana”, 20)        |
| 3    | “pear”   | 18        | 2     | Collision → chain in index 2 |
| 4    | “peach”  | 21        | 1     | Insert new pair              |

Table after insertions:

| Index | Chain                        |
|-------|------------------------------|
| 0     | -                            |
| 1     | (“peach”, 40)                |
| 2     | (“apple”, 10) → (“pear”, 30) |
| 3     | (“banana”, 20)               |

## Tiny Code (Easy Versions)

C (Separate Chaining Example)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define TABLE_SIZE 5

typedef struct Node {
    char *key;
    int value;
    struct Node *next;
} Node;

typedef struct {
    Node *buckets[TABLE_SIZE];
} HashTable;

unsigned int hash(const char *key) {
    unsigned int h = 0;
    while (*key) h = h * 31 + *key++;
    return h % TABLE_SIZE;
}

HashTable* ht_create() {
    HashTable *ht = malloc(sizeof(HashTable));
    for (int i = 0; i < TABLE_SIZE; i++) ht->buckets[i] = NULL;
    return ht;
}

void ht_insert(HashTable *ht, const char *key, int value) {
    unsigned int idx = hash(key);
    Node *node = ht->buckets[idx];
    while (node) {
        if (strcmp(node->key, key) == 0) { node->value = value; return; }
        node = node->next;
    }
    Node *new_node = malloc(sizeof(Node));
    new_node->key = strdup(key);
    new_node->value = value;
    new_node->next = ht->buckets[idx];
```

```

    ht->buckets[idx] = new_node;
}

int ht_search(HashTable *ht, const char *key, int *out) {
    unsigned int idx = hash(key);
    Node *node = ht->buckets[idx];
    while (node) {
        if (strcmp(node->key, key) == 0) { *out = node->value; return 1; }
        node = node->next;
    }
    return 0;
}

int main(void) {
    HashTable *ht = ht_create();
    ht_insert(ht, "apple", 10);
    ht_insert(ht, "pear", 30);
    ht_insert(ht, "banana", 20);
    int val;
    if (ht_search(ht, "pear", &val))
        printf("pear: %d\n", val);
    return 0;
}

```

Python (Dictionary Simulation)

```

class HashTable:
    def __init__(self, size=5):
        self.size = size
        self.table = [[] for _ in range(size)]

    def _hash(self, key):
        return hash(key) % self.size

    def insert(self, key, value):
        idx = self._hash(key)
        for pair in self.table[idx]:
            if pair[0] == key:
                pair[1] = value
                return
        self.table[idx].append([key, value])

```

```

def search(self, key):
    idx = self._hash(key)
    for k, v in self.table[idx]:
        if k == key:
            return v
    return None

# Example
ht = HashTable()
ht.insert("apple", 10)
ht.insert("pear", 30)
ht.insert("banana", 20)
print(ht.search("pear")) # 30

```

## Why It Matters

- Provides average  $O(1)$  access, insert, delete
- Backbone of symbol tables, maps, sets, and dictionaries
- Used in caches, compilers, and indexing systems
- Introduces key idea: hash function + collision handling

## A Gentle Proof (Why It Works)

Let table size =  $m$ , number of keys =  $n$ . If hash spreads keys uniformly, expected chain length =  $\lambda = n/m$  (load factor).

- Average lookup:  $O(1 + \lambda)$
- Keep  $\lambda \leq 1 \rightarrow$  near constant time If collisions are minimized, operations stay fast.

## Try It Yourself

1. Implement update to modify existing key's value.
2. Add delete(key) to remove entries from chain.
3. Experiment with different hash functions (e.g. djb2, FNV-1a).
4. Measure time vs load factor.

## Test Cases

| Operation | Key      | Value | Result  | Notes      |
|-----------|----------|-------|---------|------------|
| insert    | “apple”  | 10    | success | new key    |
| insert    | “banana” | 20    | success | new key    |
| insert    | “apple”  | 15    | update  | key exists |
| search    | “banana” |       | 20      | found      |
| search    | “grape”  |       | None    | not found  |

## Edge Cases

- Insert duplicate → update value
- Search non-existent → return None
- Table full (open addressing) → needs rehash

## Complexity

- Time: average  $O(1)$ , worst  $O(n)$  (all collide)
- Space:  $O(n + m)$  (keys + table slots)

Hash table insertion is the art of turning chaos into order, hash, map, resolve, and store.

## 212 Linear Probing

Linear probing is one of the simplest collision resolution strategies in open addressing hash tables. When a collision occurs, it looks for the next empty slot by moving step by step through the table, wrapping around if needed.

### What Problem Are We Solving?

When two keys hash to the same index, where do we store the new one? Instead of chaining nodes, linear probing searches the next available slot, keeping all data inside the array.

Goal: Resolve collisions by scanning linearly from the point of conflict until an empty slot is found.

## How Does It Work (Plain Language)?

When inserting a key:

1. Compute  $\text{index} = \text{hash}(\text{key}) \% \text{capacity}$ .
2. If slot empty  $\rightarrow$  place key there.
3. If occupied  $\rightarrow$  move to  $(\text{index} + 1) \% \text{capacity}$ .
4. Repeat until an empty slot is found or table is full.

Lookups and deletions follow the same probe sequence until key is found or empty slot encountered.

Example Steps (Capacity = 7)

| Step | Key | Hash(key) | Index | Action                               |
|------|-----|-----------|-------|--------------------------------------|
| 1    | 10  | 3         | 3     | Place at index 3                     |
| 2    | 24  | 3         | 4     | Collision $\rightarrow$ move to 4    |
| 3    | 31  | 3         | 5     | Collision $\rightarrow$ move to 5    |
| 4    | 17  | 3         | 6     | Collision $\rightarrow$ move to 6    |
| 5    | 38  | 3         | 0     | Wrap around $\rightarrow$ place at 0 |

Final Table

| Index | Value |
|-------|-------|
| 0     | 38    |
| 1     | -     |
| 2     | -     |
| 3     | 10    |
| 4     | 24    |
| 5     | 31    |
| 6     | 17    |

## Tiny Code (Easy Versions)

C

```
#include <stdio.h>
#include <stdlib.h>

#define CAPACITY 7
#define EMPTY -1
```

```

#define DELETED -2

typedef struct {
    int *table;
} HashTable;

int hash(int key) { return key % CAPACITY; }

HashTable* ht_create() {
    HashTable *ht = malloc(sizeof(HashTable));
    ht->table = malloc(sizeof(int) * CAPACITY);
    for (int i = 0; i < CAPACITY; i++) ht->table[i] = EMPTY;
    return ht;
}

void ht_insert(HashTable *ht, int key) {
    int idx = hash(key);
    for (int i = 0; i < CAPACITY; i++) {
        int pos = (idx + i) % CAPACITY;
        if (ht->table[pos] == EMPTY || ht->table[pos] == DELETED) {
            ht->table[pos] = key;
            return;
        }
    }
    printf("Table full, cannot insert %d\n", key);
}

int ht_search(HashTable *ht, int key) {
    int idx = hash(key);
    for (int i = 0; i < CAPACITY; i++) {
        int pos = (idx + i) % CAPACITY;
        if (ht->table[pos] == EMPTY) return -1;
        if (ht->table[pos] == key) return pos;
    }
    return -1;
}

void ht_delete(HashTable *ht, int key) {
    int pos = ht_search(ht, key);
    if (pos != -1) ht->table[pos] = DELETED;
}

```

```

int main(void) {
    HashTable *ht = ht_create();
    ht_insert(ht, 10);
    ht_insert(ht, 24);
    ht_insert(ht, 31);
    ht_insert(ht, 17);
    ht_insert(ht, 38);
    for (int i = 0; i < CAPACITY; i++)
        printf("[%d] = %d\n", i, ht->table[i]);
    return 0;
}

```

Python

```

class LinearProbingHash:
    def __init__(self, size=7):
        self.size = size
        self.table = [None] * size

    def _hash(self, key):
        return key % self.size

    def insert(self, key):
        idx = self._hash(key)
        for i in range(self.size):
            pos = (idx + i) % self.size
            if self.table[pos] is None or self.table[pos] == "DELETED":
                self.table[pos] = key
                return
        raise OverflowError("Hash table full")

    def search(self, key):
        idx = self._hash(key)
        for i in range(self.size):
            pos = (idx + i) % self.size
            if self.table[pos] is None:
                return None
            if self.table[pos] == key:
                return pos
        return None

    def delete(self, key):

```

```

    pos = self.search(key)
    if pos is not None:
        self.table[pos] = "DELETED"

# Example
ht = LinearProbingHash()
for k in [10, 24, 31, 17, 38]:
    ht.insert(k)
print(ht.table)
print("Search 17 at:", ht.search(17))

```

## Why It Matters

- Simplest form of open addressing
- Keeps all entries inside one array (no extra memory)
- Excellent cache performance
- Forms the basis for modern in-place hash maps

## A Gentle Proof (Why It Works)

Every key follows the same probe sequence during insert, search, and delete. So if a key is in the table, search will find it; if it's not, search will hit an empty slot and stop. Uniform hashing ensures average probe length  $1 / (1 - \alpha)$ , where  $\alpha = n / m$  is load factor.

## Try It Yourself

1. Implement `resize()` when load factor  $> 0.7$ .
2. Test insertion order and wrap-around behavior.
3. Compare linear probing vs chaining performance.
4. Visualize clustering as load increases.

## Test Cases

| Operation | Key | Result  | Notes        |
|-----------|-----|---------|--------------|
| insert    | 10  | index 3 | no collision |
| insert    | 24  | index 4 | move 1 slot  |
| insert    | 31  | index 5 | move 2 slots |
| insert    | 17  | index 6 | move 3 slots |

| Operation | Key | Result       | Notes         |
|-----------|-----|--------------|---------------|
| insert    | 38  | index 0      | wrap-around   |
| search    | 17  | found at 6   | linear search |
| delete    | 24  | mark deleted | slot reusable |

## Edge Cases

- Table full → insertion fails
- Deleted slots reused
- Must stop on EMPTY, not DELETED

## Complexity

- Time:
  - Average  $O(1)$  if small
  - Worst  $O(n)$  if full cluster
- Space:  $O(n)$

Linear probing walks straight lines through collisions, simple, local, and fast when load is low.

## 213 Quadratic Probing

Quadratic probing improves upon linear probing by reducing primary clustering. Instead of stepping through every slot one by one, it jumps in quadratic increments, spreading colliding keys more evenly across the table.

### What Problem Are We Solving?

In linear probing, consecutive occupied slots cause clustering, leading to long probe chains and degraded performance. Quadratic probing breaks up these runs by using nonlinear probe sequences.

Goal: Resolve collisions by checking indices offset by quadratic values,  $+1^2$ ,  $+2^2$ ,  $+3^2$ , ..., reducing clustering while keeping predictable probe order.

## How Does It Work (Plain Language)?

When inserting a key:

1. Compute `index = hash(key) % capacity`.
2. If slot empty → insert.
3. If occupied → try  $(index + 1^2) \% capacity$ ,  $(index + 2^2) \% capacity$ , etc.
4. Continue until an empty slot is found or table is full.

Lookups and deletions follow the same probe sequence.

Example Steps (Capacity = 7)

| Step | Key | Hash(key) | Probes (sequence) | Final Slot |
|------|-----|-----------|-------------------|------------|
| 1    | 10  | 3         | 3                 | 3          |
| 2    | 24  | 3         | 3, 4              | 4          |
| 3    | 31  | 3         | 3, 4, 0           | 0          |
| 4    | 17  | 3         | 3, 4, 0, 2        | 2          |

Table after insertions:

| Index | Value |
|-------|-------|
| 0     | 31    |
| 1     | -     |
| 2     | 17    |
| 3     | 10    |
| 4     | 24    |
| 5     | -     |
| 6     | -     |

## Tiny Code (Easy Versions)

C

```
#include <stdio.h>
#include <stdlib.h>

#define CAPACITY 7
#define EMPTY -1
#define DELETED -2
```

```

typedef struct {
    int *table;
} HashTable;

int hash(int key) { return key % CAPACITY; }

HashTable* ht_create() {
    HashTable *ht = malloc(sizeof(HashTable));
    ht->table = malloc(sizeof(int) * CAPACITY);
    for (int i = 0; i < CAPACITY; i++) ht->table[i] = EMPTY;
    return ht;
}

void ht_insert(HashTable *ht, int key) {
    int idx = hash(key);
    for (int i = 0; i < CAPACITY; i++) {
        int pos = (idx + i * i) % CAPACITY;
        if (ht->table[pos] == EMPTY || ht->table[pos] == DELETED) {
            ht->table[pos] = key;
            return;
        }
    }
    printf("Table full, cannot insert %d\n", key);
}

int ht_search(HashTable *ht, int key) {
    int idx = hash(key);
    for (int i = 0; i < CAPACITY; i++) {
        int pos = (idx + i * i) % CAPACITY;
        if (ht->table[pos] == EMPTY) return -1;
        if (ht->table[pos] == key) return pos;
    }
    return -1;
}

int main(void) {
    HashTable *ht = ht_create();
    ht_insert(ht, 10);
    ht_insert(ht, 24);
    ht_insert(ht, 31);
    ht_insert(ht, 17);
    for (int i = 0; i < CAPACITY; i++)
}

```

```

    printf("[%d] = %d\n", i, ht->table[i]);
    return 0;
}

```

Python

```

class QuadraticProbingHash:
    def __init__(self, size=7):
        self.size = size
        self.table = [None] * size

    def _hash(self, key):
        return key % self.size

    def insert(self, key):
        idx = self._hash(key)
        for i in range(self.size):
            pos = (idx + i * i) % self.size
            if self.table[pos] is None or self.table[pos] == "DELETED":
                self.table[pos] = key
                return
        raise OverflowError("Hash table full")

    def search(self, key):
        idx = self._hash(key)
        for i in range(self.size):
            pos = (idx + i * i) % self.size
            if self.table[pos] is None:
                return None
            if self.table[pos] == key:
                return pos
        return None

# Example
ht = QuadraticProbingHash()
for k in [10, 24, 31, 17]:
    ht.insert(k)
print(ht.table)
print("Search 24 at:", ht.search(24))

```

## Why It Matters

- Reduces primary clustering seen in linear probing
- Keeps keys more evenly distributed
- Avoids extra pointers (everything stays in one array)
- Useful in hash tables where space is tight and locality helps

## A Gentle Proof (Why It Works)

Probe sequence:

$$i = 0, 1, 2, 3, \dots$$

Index at step  $i$  is

$$(index + i^2) \bmod m$$

If table size  $m$  is prime, this sequence visits up to  $m/2$  distinct slots before repeating, guaranteeing an empty slot is found if load factor  $< 0.5$ .

Thus all operations follow predictable, finite sequences.

## Try It Yourself

1. Compare clustering with linear probing under same keys.
2. Experiment with different table sizes (prime vs composite).
3. Implement deletion markers properly.
4. Visualize probe paths with small tables.

## Test Cases

| Operation | Key | Probe Sequence | Slot  | Notes          |
|-----------|-----|----------------|-------|----------------|
| insert    | 10  | 3              | 3     | no collision   |
| insert    | 24  | 3, 4           | 4     | 1 step         |
| insert    | 31  | 3, 4, 0        | 0     | 2 steps        |
| insert    | 17  | 3, 4, 0, 2     | 2     | 3 steps        |
| search    | 31  | 3 → 4 → 0      | found | quadratic path |

## Edge Cases

- Table full  $\rightarrow$  insertion fails
- Requires table size prime for full coverage
- Needs load factor  $< 0.5$  to avoid infinite loops

## Complexity

- Time: average  $O(1)$ , worst  $O(n)$
- Space:  $O(n)$

Quadratic probing trades a straight line for a curve, spreading collisions smoothly across the table.

## 214 Double Hashing

Double hashing uses two independent hash functions to minimize collisions. When a conflict occurs, it jumps forward by a second hash value, creating probe sequences unique to each key and greatly reducing clustering.

### What Problem Are We Solving?

Linear and quadratic probing both suffer from clustering patterns, especially when keys share similar initial indices. Double hashing breaks this pattern by introducing a second hash function that defines each key's step size.

Goal: Use two hash functions to determine probe sequence:

$$\text{index}_i = (h_1(\text{key}) + i \cdot h_2(\text{key})) \bmod m$$

This produces independent probe paths and avoids overlap among keys.

### How Does It Work (Plain Language)?

When inserting a key:

1. Compute primary hash: `h1 = key % capacity`.
2. Compute step size: `h2 = 1 + (key % (capacity - 1))` (never zero).
3. Try `h1`; if occupied, try `(h1 + h2) % m, (h1 + 2*h2) % m`, etc.
4. Repeat until an empty slot is found.

Same pattern applies for search and delete.

Example Steps (Capacity = 7)

| Step | Key | h (key) | h (key) | Probe Sequence | Final Slot |
|------|-----|---------|---------|----------------|------------|
| 1    | 10  | 3       | 4       | 3              | 3          |
| 2    | 24  | 3       | 4       | 3 → 0          | 0          |

| Step | Key | h (key) | h (key) | Probe Sequence | Final Slot |
|------|-----|---------|---------|----------------|------------|
| 3    | 31  | 3       | 4       | 3 → 0 → 4      | 4          |
| 4    | 17  | 3       | 3       | 3 → 6          | 6          |

Final Table

| Index | Value |
|-------|-------|
| 0     | 24    |
| 1     | -     |
| 2     | -     |
| 3     | 10    |
| 4     | 31    |
| 5     | -     |
| 6     | 17    |

### Tiny Code (Easy Versions)

C

```
#include <stdio.h>
#include <stdlib.h>

#define CAPACITY 7
#define EMPTY -1
#define DELETED -2

typedef struct {
    int *table;
} HashTable;

int h1(int key) { return key % CAPACITY; }
int h2(int key) { return 1 + (key % (CAPACITY - 1)); }

HashTable* ht_create() {
    HashTable *ht = malloc(sizeof(HashTable));
    ht->table = malloc(sizeof(int) * CAPACITY);
    for (int i = 0; i < CAPACITY; i++) ht->table[i] = EMPTY;
    return ht;
}
```

```

void ht_insert(HashTable *ht, int key) {
    int idx1 = h1(key);
    int step = h2(key);
    for (int i = 0; i < CAPACITY; i++) {
        int pos = (idx1 + i * step) % CAPACITY;
        if (ht->table[pos] == EMPTY || ht->table[pos] == DELETED) {
            ht->table[pos] = key;
            return;
        }
    }
    printf("Table full, cannot insert %d\n", key);
}

int ht_search(HashTable *ht, int key) {
    int idx1 = h1(key), step = h2(key);
    for (int i = 0; i < CAPACITY; i++) {
        int pos = (idx1 + i * step) % CAPACITY;
        if (ht->table[pos] == EMPTY) return -1;
        if (ht->table[pos] == key) return pos;
    }
    return -1;
}

int main(void) {
    HashTable *ht = ht_create();
    ht_insert(ht, 10);
    ht_insert(ht, 24);
    ht_insert(ht, 31);
    ht_insert(ht, 17);
    for (int i = 0; i < CAPACITY; i++)
        printf("[%d] = %d\n", i, ht->table[i]);
    return 0;
}

```

Python

```

class DoubleHash:
    def __init__(self, size=7):
        self.size = size
        self.table = [None] * size

    def _h1(self, key):

```

```

        return key % self.size

    def _h2(self, key):
        return 1 + (key % (self.size - 1))

    def insert(self, key):
        h1 = self._h1(key)
        h2 = self._h2(key)
        for i in range(self.size):
            pos = (h1 + i * h2) % self.size
            if self.table[pos] is None or self.table[pos] == "DELETED":
                self.table[pos] = key
            return
        raise OverflowError("Hash table full")

    def search(self, key):
        h1 = self._h1(key)
        h2 = self._h2(key)
        for i in range(self.size):
            pos = (h1 + i * h2) % self.size
            if self.table[pos] is None:
                return None
            if self.table[pos] == key:
                return pos
        return None

# Example
ht = DoubleHash()
for k in [10, 24, 31, 17]:
    ht.insert(k)
print(ht.table)
print("Search 24 at:", ht.search(24))

```

## Why It Matters

- Minimizes primary and secondary clustering
- Probe sequences depend on key, not shared among colliding keys
- Achieves uniform distribution when both hash functions are good
- Forms basis for high-performance open addressing maps

## A Gentle Proof (Why It Works)

If capacity  $m$  is prime and  $h$  (key) never equals 0, then each key generates a unique probe sequence covering all slots:

$$\text{indices} = h_1, h_1 + h_2, h_1 + 2h_2, \dots \bmod m$$

Thus an empty slot is always reachable, and searches find all candidates.

Expected probe count  $\frac{1}{1-\alpha}$ , same as other open addressing, but with lower clustering.

## Try It Yourself

1. Experiment with different  $h$  functions (e.g. `7 - key % 7`).
2. Compare probe lengths with linear and quadratic probing.
3. Visualize probe paths for small table sizes.
4. Test with composite vs prime capacities.

## Test Cases

| Operation | Key | $h$  | $h$   | Probe Sequence | Final Slot |
|-----------|-----|------|-------|----------------|------------|
| insert    | 10  | 3    | 4     | 3              | 3          |
| insert    | 24  | 3    | 4     | 3, 0           | 0          |
| insert    | 31  | 3    | 4     | 3, 0, 4        | 4          |
| insert    | 17  | 3    | 3     | 3, 6           | 6          |
| search    | 24  | 3, 0 | found |                |            |

## Edge Cases

- $h$  (key) must be nonzero
- $m$  should be prime for full coverage
- Poor hash choice  $\rightarrow$  incomplete coverage

## Complexity

- Time: average  $O(1)$ , worst  $O(n)$
- Space:  $O(n)$

Double hashing turns collisions into a graceful dance, two hash functions weaving paths that rarely cross.

## 215 Cuckoo Hashing

Cuckoo hashing takes inspiration from nature: like the cuckoo bird laying eggs in multiple nests, each key has more than one possible home. If a spot is taken, it *kicks out* the current occupant, which then moves to its alternate home, ensuring fast and predictable lookups.

### What Problem Are We Solving?

Traditional open addressing methods (linear, quadratic, double hashing) may degrade under high load factors, causing long probe sequences. Cuckoo hashing guarantees constant-time lookups by giving each key multiple possible positions.

Goal: Use two hash functions and relocate keys upon collision, maintaining  $O(1)$  search and insert time.

### How Does It Work (Plain Language)?

Each key has two candidate slots, determined by two hash functions:

$$h_1(k), \quad h_2(k)$$

When inserting a key:

1. Try  $h_1(k) \rightarrow$  if empty, place it.
2. If occupied  $\rightarrow$  kick out the existing key.
3. Reinsert the displaced key into its alternate slot.
4. Repeat until all keys placed or cycle detected (then rehash).

Example Steps (Capacity = 7)

| Step | Key | $h_1(k)$ | $h_2(k)$ | Action                                |
|------|-----|----------|----------|---------------------------------------|
| 1    | 10  | 3        | 5        | Slot 3 empty $\rightarrow$ place 10   |
| 2    | 24  | 3        | 4        | Slot 3 occupied $\rightarrow$ move 10 |
| 3    | 10  | 5        | 3        | Slot 5 empty $\rightarrow$ place 10   |
| 4    | 31  | 3        | 6        | Slot 3 empty $\rightarrow$ place 31   |

Final Table

| Index | Value |
|-------|-------|
| 0     | -     |

| Index | Value |
|-------|-------|
| 1     | -     |
| 2     | -     |
| 3     | 31    |
| 4     | 24    |
| 5     | 10    |
| 6     | -     |

Every key is accessible in O(1) by checking two positions only.

### Tiny Code (Easy Versions)

C (Two-Table Cuckoo Hashing)

```
#include <stdio.h>
#include <stdlib.h>

#define CAPACITY 7
#define EMPTY -1
#define MAX_RELOCATIONS 10

typedef struct {
    int table1[CAPACITY];
    int table2[CAPACITY];
} CuckooHash;

int h1(int key) { return key % CAPACITY; }
int h2(int key) { return (key / CAPACITY) % CAPACITY; }

void init(CuckooHash *ht) {
    for (int i = 0; i < CAPACITY; i++) {
        ht->table1[i] = EMPTY;
        ht->table2[i] = EMPTY;
    }
}

int insert(CuckooHash *ht, int key) {
    int pos, tmp, loop_guard = 0;
    for (int i = 0; i < MAX_RELOCATIONS; i++) {
        pos = h1(key);
```

```

    if (ht->table1[pos] == EMPTY) {
        ht->table1[pos] = key;
        return 1;
    }
    // kick out
    tmp = ht->table1[pos];
    ht->table1[pos] = key;
    key = tmp;
    pos = h2(key);
    if (ht->table2[pos] == EMPTY) {
        ht->table2[pos] = key;
        return 1;
    }
    tmp = ht->table2[pos];
    ht->table2[pos] = key;
    key = tmp;
}
printf("Cycle detected, rehash needed\n");
return 0;
}

int search(CuckooHash *ht, int key) {
    int pos1 = h1(key);
    int pos2 = h2(key);
    if (ht->table1[pos1] == key || ht->table2[pos2] == key) return 1;
    return 0;
}

int main(void) {
    CuckooHash ht;
    init(&ht);
    insert(&ht, 10);
    insert(&ht, 24);
    insert(&ht, 31);
    for (int i = 0; i < CAPACITY; i++)
        printf("[%d] T1=%d T2=%d\n", i, ht.table1[i], ht.table2[i]);
    return 0;
}

```

Python

```

class CuckooHash:
    def __init__(self, size=7):
        self.size = size
        self.table1 = [None] * size
        self.table2 = [None] * size
        self.max_reloc = 10

    def _h1(self, key): return key % self.size
    def _h2(self, key): return (key // self.size) % self.size

    def insert(self, key):
        for _ in range(self.max_reloc):
            idx1 = self._h1(key)
            if self.table1[idx1] is None:
                self.table1[idx1] = key
                return
            key, self.table1[idx1] = self.table1[idx1], key # swap

            idx2 = self._h2(key)
            if self.table2[idx2] is None:
                self.table2[idx2] = key
                return
            key, self.table2[idx2] = self.table2[idx2], key # swap
        raise RuntimeError("Cycle detected, rehash needed")

    def search(self, key):
        return key in self.table1 or key in self.table2

# Example
ht = CuckooHash()
for k in [10, 24, 31]:
    ht.insert(k)
print("Table1:", ht.table1)
print("Table2:", ht.table2)
print("Search 24:", ht.search(24))

```

## Why It Matters

- O(1) lookup, always two slots per key
- Avoids clustering entirely
- Excellent for high load factors (up to 0.5–0.9)

- Simple predictable probe path
- Great choice for hardware tables (e.g. network routing)

### A Gentle Proof (Why It Works)

Each key has at most two possible homes.

- If both occupied, displacement ensures eventual convergence (or detects a cycle).
- Cycle length bounded → rehash needed rarely.

Expected insertion time =  $O(1)$  amortized; search always 2 checks only.

### Try It Yourself

1. Implement rehash when cycle detected.
2. Add delete(key) and test reinsert.
3. Visualize displacement chain on insertions.
4. Compare performance with double hashing.

### Test Cases

| Operation | Key | h    | h     | Action                  |
|-----------|-----|------|-------|-------------------------|
| insert    | 10  | 3    | 5     | place at 3              |
| insert    | 24  | 3    | 4     | displace 10 → move to 5 |
| insert    | 31  | 3    | 6     | place at 3              |
| search    | 10  | 3, 5 | found |                         |

### Edge Cases

- Cycle detected → requires rehash
- Both tables full → resize
- Must limit relocation attempts

## Complexity

- Time:
  - Lookup:  $O(1)$
  - Insert:  $O(1)$  amortized
- Space:  $O(2n)$

Cuckoo hashing keeps order in the nest, every key finds a home, or the table learns to rebuild its world.

## 216 Robin Hood Hashing

Robin Hood hashing is a clever twist on open addressing: when a new key collides, it compares its “distance from home” with the current occupant. If the new key has traveled farther, it *steals* the slot, redistributing probe distances more evenly and keeping variance low.

### What Problem Are We Solving?

In linear probing, unlucky keys might travel long distances while others sit close to their home. This leads to probe sequence imbalance, long searches for some keys, short for others. Robin Hood hashing “robs” near-home keys to help far-away ones, minimizing the maximum probe distance.

Goal: Equalize probe distances by swapping keys so that no key is “too far” behind others.

### How Does It Work (Plain Language)?

Each entry remembers its probe distance = number of steps from its original hash slot. When inserting a new key:

1. Compute `index = hash(key) % capacity.`
2. If slot empty → insert.
3. If occupied → compare probe distances.
  - If newcomer’s distance > occupant’s distance → swap them.
  - Continue insertion for the displaced key.

Example Steps (Capacity = 7)

| Probe |     |           |          |                                                                          |
|-------|-----|-----------|----------|--------------------------------------------------------------------------|
| Step  | Key | Hash(key) | Distance | Action                                                                   |
| 1     | 10  | 3         | 0        | Place at 3                                                               |
| 2     | 24  | 3         | 0        | Collision → move to 4 (dist=1)                                           |
| 3     | 31  | 3         | 0        | Collision → dist=0 < 0? no → move → dist=1 < 1? no → dist=2 → place at 5 |
| 4     | 17  | 3         | 0        | Collision chain → compare and swap if farther                            |

This ensures all keys stay near their home index, fairer access for all.

Result Table:

| Index | Key | Dist |
|-------|-----|------|
| 3     | 10  | 0    |
| 4     | 24  | 1    |
| 5     | 31  | 2    |

### Tiny Code (Easy Versions)

C

```
#include <stdio.h>
#include <stdlib.h>

#define CAPACITY 7
#define EMPTY -1

typedef struct {
    int key;
    int dist; // probe distance
} Slot;

typedef struct {
    Slot *table;
} HashTable;

int hash(int key) { return key % CAPACITY; }

HashTable* ht_create() {
    HashTable *ht = malloc(sizeof(HashTable));
}
```

```

ht->table = malloc(sizeof(Slot) * CAPACITY);
for (int i = 0; i < CAPACITY; i++) {
    ht->table[i].key = EMPTY;
    ht->table[i].dist = 0;
}
return ht;
}

void ht_insert(HashTable *ht, int key) {
    int idx = hash(key);
    int dist = 0;

    while (1) {
        if (ht->table[idx].key == EMPTY) {
            ht->table[idx].key = key;
            ht->table[idx].dist = dist;
            return;
        }

        if (dist > ht->table[idx].dist) {
            // swap keys
            int tmp_key = ht->table[idx].key;
            int tmp_dist = ht->table[idx].dist;
            ht->table[idx].key = key;
            ht->table[idx].dist = dist;
            key = tmp_key;
            dist = tmp_dist;
        }

        idx = (idx + 1) % CAPACITY;
        dist++;
        if (dist >= CAPACITY) {
            printf("Table full\n");
            return;
        }
    }
}

int ht_search(HashTable *ht, int key) {
    int idx = hash(key);
    int dist = 0;
    while (ht->table[idx].key != EMPTY && dist <= ht->table[idx].dist) {

```

```

        if (ht->table[idx].key == key) return idx;
        idx = (idx + 1) % CAPACITY;
        dist++;
    }
    return -1;
}

int main(void) {
    HashTable *ht = ht_create();
    ht_insert(ht, 10);
    ht_insert(ht, 24);
    ht_insert(ht, 31);
    for (int i = 0; i < CAPACITY; i++) {
        if (ht->table[i].key != EMPTY)
            printf("[%d] key=%d dist=%d\n", i, ht->table[i].key, ht->table[i].dist);
    }
    return 0;
}

```

Python

```

class RobinHoodHash:
    def __init__(self, size=7):
        self.size = size
        self.table = [None] * size
        self.dist = [0] * size

    def _hash(self, key):
        return key % self.size

    def insert(self, key):
        idx = self._hash(key)
        d = 0
        while True:
            if self.table[idx] is None:
                self.table[idx] = key
                self.dist[idx] = d
                return
            # Robin Hood swap if newcomer is farther
            if d > self.dist[idx]:
                key, self.table[idx] = self.table[idx], key
                d, self.dist[idx] = self.dist[idx], d

```

```

        idx = (idx + 1) % self.size
        d += 1
        if d >= self.size:
            raise OverflowError("Table full")

    def search(self, key):
        idx = self._hash(key)
        d = 0
        while self.table[idx] is not None and d <= self.dist[idx]:
            if self.table[idx] == key:
                return idx
            idx = (idx + 1) % self.size
            d += 1
        return None

# Example
ht = RobinHoodHash()
for k in [10, 24, 31]:
    ht.insert(k)
print(list(zip(range(ht.size), ht.table, ht.dist)))
print("Search 24:", ht.search(24))

```

## Why It Matters

- Balances access time across all keys
- Minimizes variance of probe lengths
- Outperforms linear probing under high load
- Elegant fairness principle, long-traveling keys get priority

## A Gentle Proof (Why It Works)

By ensuring all probe distances are roughly equal, worst-case search cost = average search cost.  
 Keys never get “stuck” behind long clusters, and searches terminate early when probe distance exceeds that of existing slot.

Average search cost =  $O(1 + \sqrt{n})$ , but with *smaller variance* than standard linear probing.

## Try It Yourself

1. Insert keys in different orders and compare probe distances.

2. Implement deletion (mark deleted and shift neighbors).
3. Track average probe distance as load grows.
4. Compare fairness with standard linear probing.

## Test Cases

| Operation | Key | Home | Final Slot | Dist | Notes             |
|-----------|-----|------|------------|------|-------------------|
| insert    | 10  | 3    | 3          | 0    | first insert      |
| insert    | 24  | 3    | 4          | 1    | collision         |
| insert    | 31  | 3    | 5          | 2    | further collision |
| search    | 24  | 3→4  | found      |      |                   |

## Edge Cases

- Table full → stop insertion
- Must cap distance to prevent infinite loop
- Deletion requires rebalancing neighbors

## Complexity

- Time: average  $O(1)$ , worst  $O(n)$
- Space:  $O(n)$

Robin Hood hashing brings justice to collisions, no key left wandering too far from home.

## 217 Chained Hash Table

A chained hash table is the classic solution for handling collisions, instead of squeezing every key into the array, each bucket holds a linked list (or chain) of entries that share the same hash index.

### What Problem Are We Solving?

With open addressing, collisions force you to probe for new slots inside the array. Chaining solves collisions externally, every index points to a small dynamic list, so multiple keys can share the same slot without crowding.

Goal: Use linked lists (chains) to store colliding keys at the same hash index, keeping insert, search, and delete simple and efficient on average.

## How Does It Work (Plain Language)?

Each array index stores a pointer to a linked list of key-value pairs. When inserting:

1. Compute index = `hash(key) % capacity`.
2. Traverse chain to check if key exists.
3. If not, append new node to the front (or back).

Searching and deleting follow the same index and chain.

Example (Capacity = 5)

| Step | Key   | Hash(key) | Index | Action             |
|------|-------|-----------|-------|--------------------|
| 1    | “cat” | 2         | 2     | Place in chain[2]  |
| 2    | “dog” | 4         | 4     | Place in chain[4]  |
| 3    | “bat” | 2         | 2     | Append to chain[2] |
| 4    | “ant” | 2         | 2     | Append to chain[2] |

Table Structure

| Index | Chain                 |
|-------|-----------------------|
| 0     | -                     |
| 1     | -                     |
| 2     | “cat” → “bat” → “ant” |
| 3     | -                     |
| 4     | “dog”                 |

## Tiny Code (Easy Versions)

C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define CAPACITY 5

typedef struct Node {
    char *key;
    int value;
    struct Node *next;
```

```

} Node;

typedef struct {
    Node *buckets[CAPACITY];
} HashTable;

unsigned int hash(const char *key) {
    unsigned int h = 0;
    while (*key) h = h * 31 + *key++;
    return h % CAPACITY;
}

HashTable* ht_create() {
    HashTable *ht = malloc(sizeof(HashTable));
    for (int i = 0; i < CAPACITY; i++) ht->buckets[i] = NULL;
    return ht;
}

void ht_insert(HashTable *ht, const char *key, int value) {
    unsigned int idx = hash(key);
    Node *node = ht->buckets[idx];
    while (node) {
        if (strcmp(node->key, key) == 0) { node->value = value; return; }
        node = node->next;
    }
    Node *new_node = malloc(sizeof(Node));
    new_node->key = strdup(key);
    new_node->value = value;
    new_node->next = ht->buckets[idx];
    ht->buckets[idx] = new_node;
}

int ht_search(HashTable *ht, const char *key, int *out) {
    unsigned int idx = hash(key);
    Node *node = ht->buckets[idx];
    while (node) {
        if (strcmp(node->key, key) == 0) { *out = node->value; return 1; }
        node = node->next;
    }
    return 0;
}

```

```

void ht_delete(HashTable *ht, const char *key) {
    unsigned int idx = hash(key);
    Node curr = &ht->buckets[idx];
    while (*curr) {
        if (strcmp((*curr)->key, key) == 0) {
            Node *tmp = *curr;
            *curr = (*curr)->next;
            free(tmp->key);
            free(tmp);
            return;
        }
        curr = &(*curr)->next;
    }
}

int main(void) {
    HashTable *ht = ht_create();
    ht_insert(ht, "cat", 1);
    ht_insert(ht, "bat", 2);
    ht_insert(ht, "ant", 3);
    int val;
    if (ht_search(ht, "bat", &val))
        printf("bat: %d\n", val);
    ht_delete(ht, "bat");
    if (!ht_search(ht, "bat", &val))
        printf("bat deleted\n");
    return 0;
}

```

Python

```

class ChainedHash:
    def __init__(self, size=5):
        self.size = size
        self.table = [[] for _ in range(size)]

    def _hash(self, key):
        return hash(key) % self.size

    def insert(self, key, value):
        idx = self._hash(key)
        for pair in self.table[idx]:

```

```

        if pair[0] == key:
            pair[1] = value
            return
        self.table[idx].append([key, value])

    def search(self, key):
        idx = self._hash(key)
        for k, v in self.table[idx]:
            if k == key:
                return v
        return None

    def delete(self, key):
        idx = self._hash(key)
        self.table[idx] = [p for p in self.table[idx] if p[0] != key]

# Example
ht = ChainedHash()
ht.insert("cat", 1)
ht.insert("bat", 2)
ht.insert("ant", 3)
print(ht.table)
print("Search bat:", ht.search("bat"))
ht.delete("bat")
print(ht.table)

```

## Why It Matters

- Simple and reliable collision handling
- Load factor can exceed 1 (chains absorb overflow)
- Deletion is straightforward (just remove node)
- Performance stable even at high load (if hash spread is uniform)

## A Gentle Proof (Why It Works)

Expected chain length = load factor  $\alpha = \frac{n}{m}$ . Each operation traverses a single chain, so average cost =  $O(1 + \alpha)$ . Uniform hash distribution ensures  $\alpha$  remains small  $\rightarrow$  operations  $O(1)$ .

## Try It Yourself

1. Implement dynamic resizing when average chain length grows.
2. Compare prepend vs append strategies.
3. Measure average search steps as table fills.
4. Replace linked list with balanced tree (for high  $\lambda$ ).

## Test Cases

| Operation | Key   | Index | Chain Result          | Notes       |
|-----------|-------|-------|-----------------------|-------------|
| insert    | “cat” | 2     | ["cat"]               |             |
| insert    | “bat” | 2     | ["bat", “cat”]        | collision   |
| insert    | “ant” | 2     | ["ant", “bat”, “cat”] | chain grows |
| search    | “bat” | 2     | found                 |             |
| delete    | “bat” | 2     | ["ant", “cat”]        | removed     |

## Edge Cases

- Many keys same index → long chains
- Poor hash function → uneven distribution
- Needs memory for pointers/nodes

## Complexity

- Time: average  $O(1)$ , worst  $O(n)$  (all in one chain)
- Space:  $O(n + m)$

Chained hashing turns collisions into conversation, if one bucket's full, it just lines them up neatly in a list.

## 218 Perfect Hashing

Perfect hashing is the dream scenario for hash tables, no collisions at all. Every key maps to a unique slot, so lookups, inserts, and deletes all take  $O(1)$  time *worst case*, not just on average.

## What Problem Are We Solving?

Most hashing strategies (linear probing, chaining, cuckoo) deal with collisions after they happen. Perfect hashing eliminates them entirely by designing a collision-free hash function for a fixed key set.

Goal: Construct a hash function  $h(k)$  such that all keys map to distinct indices.

## How Does It Work (Plain Language)?

If the set of keys is known in advance (static set), we can carefully choose or build a hash function that gives each key a unique slot.

Two main types:

1. Perfect Hashing, no collisions.
2. Minimal Perfect Hashing, no collisions and table size = number of keys.

Simple Example (Keys = {10, 24, 31, 17}, Capacity = 7)

Let's find a function:

$$h(k) = (a \cdot k + b) \bmod 7$$

We can search for coefficients  $a$  and  $b$  that produce unique indices:

| Key | $h(k) = (2k + 1) \bmod 7$ | Index     |
|-----|---------------------------|-----------|
| 10  | $(21) \% 7 = 0$           | 0         |
| 24  | $(49) \% 7 = 0$           | collision |
| 31  | $(63) \% 7 = 0$           | collision |

So we try another pair ( $a=3, b=2$ ):

| Key | $(3k + 2) \bmod 7$ | Index     |
|-----|--------------------|-----------|
| 10  | 5                  | 5         |
| 24  | 4                  | 4         |
| 31  | 4                  | collision |
| 17  | 6                  | 6         |

Eventually, we find a mapping with no repeats by adjusting parameters or using two-level construction.

## Two-Level Perfect Hashing

Practical perfect hashing often uses a two-level scheme:

1. Top level: Hash keys into buckets.
2. Second level: Each bucket gets its own small hash table with its own perfect hash function.

This ensures zero collisions overall, with total space  $O(n)$ .

Process:

- Each bucket of size  $b$  gets a secondary table of size  $b^2$ .
- Use a second hash  $h_i$  for that bucket to place all keys uniquely.

## Tiny Code (Easy Version)

Python (Two-Level Static Perfect Hashing)

```
import random

class PerfectHash:
    def __init__(self, keys):
        self.n = len(keys)
        self.size = self.n
        self.buckets = [[] for _ in range(self.size)]
        self.secondary = [None] * self.size

        # First-level hashing
        a, b = 3, 5 # fixed small hash parameters
        def h1(k): return (a * k + b) % self.size

        # Distribute keys into buckets
        for k in keys:
            self.buckets[h1(k)].append(k)

        # Build second-level tables
        for i, bucket in enumerate(self.buckets):
            if not bucket:
                continue
            m = len(bucket) ** 2
            table = [None] * m
            found = False
            while not found:
```

```

        found = True
        a2, b2 = random.randint(1, m - 1), random.randint(0, m - 1)
        def h2(k): return (a2 * k + b2) % m
        table = [None] * m
        for k in bucket:
            pos = h2(k)
            if table[pos] is not None:
                found = False
                break
            table[pos] = k
        self.secondary[i] = (table, a2, b2)

    self.h1 = h1

    def search(self, key):
        i = self.h1(key)
        table, a2, b2 = self.secondary[i]
        m = len(table)
        pos = (a2 * key + b2) % m
        return table[pos] == key

# Example
keys = [10, 24, 31, 17]
ph = PerfectHash(keys)
print([len(b) for b in ph.buckets])
print("Search 24:", ph.search(24))
print("Search 11:", ph.search(11))

```

## Why It Matters

- Guaranteed O(1) worst-case lookup
- No clustering, no collisions, no chains
- Ideal for static key sets (e.g. reserved keywords in a compiler, routing tables)
- Memory predictable, access blazing fast

## A Gentle Proof (Why It Works)

Let  $n = |S|$  be the number of keys.

With a truly random hash family into  $m$  buckets, the collision probability for two distinct keys is  $1/m$ .

Choose  $m = n^2$ . Then the expected number of collisions is

$$\mathbb{E}[C] = \binom{n}{2} \cdot \frac{1}{m} = \frac{n(n-1)}{2n^2} < \frac{1}{2}.$$

By Markov,  $\Pr[C \geq 1] \leq \mathbb{E}[C] < \frac{1}{2}$ , so  $\Pr[C = 0] > \frac{1}{2}$ .

Therefore a collision-free hash exists. In practice, try random seeds until one has  $C = 0$ , or use a deterministic construction for perfect hashing.

### Try It Yourself

1. Generate perfect hash for small static set {"if", "else", "for", "while"}.
2. Build minimal perfect hash (table size =  $n$ ).
3. Compare lookup times with standard dict.
4. Visualize second-level hash table sizes.

### Test Cases

| Operation  | Keys          | Result  | Notes            |
|------------|---------------|---------|------------------|
| build      | [10,24,31,17] | success | each slot unique |
| search     | 24            | True    | found            |
| search     | 11            | False   | not in table     |
| collisions | none          |         | perfect mapping  |

### Edge Cases

- Works only for static sets (no dynamic inserts)
- Building may require rehash trials
- Memory  $\uparrow$  with quadratic secondary tables

### Complexity

- Build:  $O(n^2)$  (search for collision-free mapping)
- Lookup:  $O(1)$
- Space:  $O(n)$  to  $O(n^2)$  depending on method

Perfect hashing is like finding the perfect key for every lock, built once, opens instantly, never collides.

## 219 Consistent Hashing

Consistent hashing is a collision-handling strategy designed for *distributed systems* rather than single in-memory tables. It ensures that when nodes (servers, caches, or shards) join or leave, only a small fraction of keys need to be remapped, making it the backbone of scalable, fault-tolerant architectures.

### What Problem Are We Solving?

In traditional hashing (e.g. `hash(key) % n`), when the number of servers  $n$  changes, almost every key's location changes. That's disastrous for caching, databases, or load balancing.

Consistent hashing fixes this by mapping both keys and servers into the same hash space, and placing keys near their nearest server clockwise, minimizing reassessments when the system changes.

Goal: Achieve stable key distribution under dynamic server counts with minimal movement and good balance.

### How Does It Work (Plain Language)?

1. Imagine a hash ring, numbers 0 through  $(2^m - 1)$  arranged in a circle.
2. Each node (server) and key is hashed to a position on the ring.
3. A key is assigned to the next server clockwise from its hash position.
4. When a node joins/leaves, only keys in its immediate region move.

Example (Capacity =  $2^1$ )

| Item   | Hash | Placed On Ring         | Owner |
|--------|------|------------------------|-------|
| Node A | 1000 | •                      |       |
| Node B | 4000 | •                      |       |
| Node C | 8000 | •                      |       |
| Key K  | 1200 | → Node B               |       |
| Key K  | 8500 | → Node A (wrap-around) |       |

If Node B leaves, only keys in its segment (1000–4000) move, everything else stays put.

## Improving Load Balance

To prevent uneven distribution, each node is represented by multiple virtual nodes (vnodes), each vnode gets its own hash. This smooths the key spread across all nodes.

Example:

- Node A → hashes to 1000, 6000
- Node B → hashes to 3000, 9000

Keys are assigned to closest vnode clockwise.

## Tiny Code (Easy Versions)

Python (Simple Consistent Hashing with Virtual Nodes)

```
import bisect
import hashlib

def hash_fn(key):
    return int(hashlib.md5(str(key).encode()).hexdigest(), 16)

class ConsistentHash:
    def __init__(self, nodes=None, vnodes=3):
        self.ring = []
        self.map = {}
        self.vnodes = vnodes
        if nodes:
            for n in nodes:
                self.add_node(n)

    def add_node(self, node):
        for i in range(self.vnodes):
            h = hash_fn(f"{node}-{i}")
            self.map[h] = node
            bisect.insort(self.ring, h)

    def remove_node(self, node):
        for i in range(self.vnodes):
            h = hash_fn(f"{node}-{i}")
            self.ring.remove(h)
            del self.map[h]
```

```

def get_node(self, key):
    if not self.ring:
        return None
    h = hash_fn(key)
    idx = bisect.bisect(self.ring, h) % len(self.ring)
    return self.map[self.ring[idx]]

# Example
ch = ConsistentHash(["A", "B", "C"], vnodes=2)
print("Key 42 ->", ch.get_node(42))
ch.remove_node("B")
print("After removing B, Key 42 ->", ch.get_node(42))

```

## Why It Matters

- Minimizes key remapping when nodes change ( $1/n$  of keys move)
- Enables elastic scaling for caches, DB shards, and distributed stores
- Used in systems like Amazon Dynamo, Cassandra, Riak, and memcached clients
- Balances load using virtual nodes
- Decouples hash function from node count

## A Gentle Proof (Why It Works)

Let  $N$  be the number of nodes and  $K$  the number of keys.

Each node is responsible for a fraction  $\frac{1}{N}$  of the ring.

When one node leaves, only its segment's keys move, about  $K/N$ .

So the expected remapping fraction is  $\frac{1}{N}$ .

Adding virtual nodes (vnodes) increases uniformity.

With  $V$  vnodes per physical node, the variance of the per-node load fraction scales as  $\approx \frac{1}{V}$ .

## Try It Yourself

1. Add and remove nodes, track how many keys move.
2. Experiment with different vnode counts (1, 10, 100).
3. Visualize hash ring, mark nodes and key positions.
4. Simulate caching: assign 1000 keys, remove one node, count moves.

## Test Cases

| Operation   | Node(s) | Keys      | Result             | Notes           |
|-------------|---------|-----------|--------------------|-----------------|
| add_nodes   | A, B, C | [1..1000] | distributed evenly |                 |
| remove_node | B       | [1..1000] | ~1/3 keys moved    | stability check |
| add_node    | D       | [1..1000] | ~1/4 keys remapped |                 |
| lookup      | 42      | -> Node C | consistent mapping |                 |

## Edge Cases

- Empty ring → return None
- Duplicate nodes → handle via unique vnode IDs
- Without vnodes → uneven load

## Complexity

- Lookup:  $O(\log n)$  (binary search in the ring)
- Insert/Delete node:  $O(v \log n)$
- Space:  $O(n \times v)$

Consistent hashing keeps order in the storm — servers may come and go, but most keys stay right where they belong.

## 220 Dynamic Rehashing

Dynamic rehashing is how a hash table gracefully adapts as data grows or shrinks. Instead of being trapped in a fixed-size array, the table *resizes itself*, rebuilding its layout so that load stays balanced and lookups remain fast.

### What Problem Are We Solving?

When a hash table fills up, collisions become frequent, degrading performance to  $O(n)$ . We need a mechanism to maintain a low load factor (ratio of elements to capacity) by resizing and rehashing automatically.

Goal: Detect when load factor crosses a threshold, allocate a larger array, and rehash all keys into their new positions efficiently.

## How Does It Work (Plain Language)?

1. Monitor load factor

$$\alpha = \frac{n}{m}$$

where  $n$  is the number of elements and  $m$  is the table size.

2. Trigger rehash

- If  $\alpha > 0.75$ , expand the table (for example, double the capacity).
- If  $\alpha < 0.25$ , shrink it (optional).

3. Rebuild

- Create a new table with the updated capacity.
- Reinsert each key using the new hash function modulo the new capacity.

### Example Steps

| Step | Capacity | Items | Load Factor | Action      |
|------|----------|-------|-------------|-------------|
| 1    | 4        | 2     | 0.5         | ok          |
| 2    | 4        | 3     | 0.75        | ok          |
| 3    | 4        | 4     | 1.0         | resize to 8 |
| 4    | 8        | 4     | 0.5         | rehashed    |

Every key gets new position because `hash(key) % new_capacity` changes.

## Incremental Rehashing

Instead of rehashing all keys at once (costly spike), incremental rehashing spreads work across operations:

- Maintain both old and new tables.
- Rehash a few entries per insert/search until old table empty.

This keeps amortized O(1) performance, even during resize.

## Tiny Code (Easy Versions)

C (Simple Doubling Rehash)

```
#include <stdio.h>
#include <stdlib.h>

#define INIT_CAP 4

typedef struct {
    int *keys;
    int size;
    int count;
} HashTable;

int hash(int key, int size) { return key % size; }

HashTable* ht_create(int size) {
    HashTable *ht = malloc(sizeof(HashTable));
    ht->keys = malloc(sizeof(int) * size);
    for (int i = 0; i < size; i++) ht->keys[i] = -1;
    ht->size = size;
    ht->count = 0;
    return ht;
}

void ht_resize(HashTable *ht, int new_size) {
    printf("Resizing from %d to %d\n", ht->size, new_size);
    int *old_keys = ht->keys;
    int old_size = ht->size;

    ht->keys = malloc(sizeof(int) * new_size);
    for (int i = 0; i < new_size; i++) ht->keys[i] = -1;

    ht->size = new_size;
    ht->count = 0;
    for (int i = 0; i < old_size; i++) {
        if (old_keys[i] != -1) {
            int key = old_keys[i];
            int idx = hash(key, new_size);
            while (ht->keys[idx] != -1) idx = (idx + 1) % new_size;
            ht->keys[idx] = key;
            ht->count++;
        }
    }
}
```

```

        }
    }

    free(old_keys);
}

void ht_insert(HashTable *ht, int key) {
    float load = (float)ht->count / ht->size;
    if (load > 0.75) ht_resize(ht, ht->size * 2);

    int idx = hash(key, ht->size);
    while (ht->keys[idx] != -1) idx = (idx + 1) % ht->size;
    ht->keys[idx] = key;
    ht->count++;
}

int main(void) {
    HashTable *ht = ht_create(INIT_CAP);
    for (int i = 0; i < 10; i++) ht_insert(ht, i * 3);
    for (int i = 0; i < ht->size; i++)
        printf("[%d] = %d\n", i, ht->keys[i]);
    return 0;
}

```

Python

```

class DynamicHash:
    def __init__(self, cap=4):
        self.cap = cap
        self.size = 0
        self.table = [None] * cap

    def _hash(self, key):
        return hash(key) % self.cap

    def _rehash(self, new_cap):
        old_table = self.table
        self.table = [None] * new_cap
        self.cap = new_cap
        self.size = 0
        for key in old_table:
            if key is not None:
                self.insert(key)

```

```

def insert(self, key):
    if self.size / self.cap > 0.75:
        self._rehash(self.cap * 2)
    idx = self._hash(key)
    while self.table[idx] is not None:
        idx = (idx + 1) % self.cap
    self.table[idx] = key
    self.size += 1

# Example
ht = DynamicHash()
for k in [10, 24, 31, 17, 19, 42, 56, 77]:
    ht.insert(k)
print(ht.table)

```

## Why It Matters

- Keeps load factor stable for O(1) operations
- Prevents clustering in open addressing
- Supports unbounded growth
- Foundation for dynamic dictionaries, maps, caches

## A Gentle Proof (Why It Works)

If each rehash doubles capacity, total cost of  $N$  inserts =  $O(N)$ . Each element is moved  $O(1)$  times (once per doubling), so amortized cost per insert is  $O(1)$ .

Incremental rehashing further ensures no single operation is expensive, work spread evenly.

## Try It Yourself

1. Add printouts to observe load factor at each insert.
2. Implement shrink when load  $< 0.25$ .
3. Implement incremental rehash with two tables.
4. Compare doubling vs prime-capacity growth.

## Test Cases

| Step | Capacity | Items | Load Factor | Action   |
|------|----------|-------|-------------|----------|
| 1    | 4        | 2     | 0.5         | none     |
| 2    | 4        | 3     | 0.75        | ok       |
| 3    | 4        | 4     | 1.0         | resize   |
| 4    | 8        | 4     | 0.5         | rehashed |

## Edge Cases

- Rehash must handle deleted slots correctly
- Avoid resizing too frequently (hysteresis)
- Keep hash function consistent across resizes

## Complexity

- Average Insert/Search/Delete:  $O(1)$
- Amortized Insert:  $O(1)$
- Worst-case Resize:  $O(n)$

Dynamic rehashing is the table's heartbeat, it expands when full, contracts when idle, always keeping operations smooth and steady.

## Section 23. Heaps

### 221 Binary Heap Insert

A binary heap is a complete binary tree stored in an array that maintains the heap property, each parent is smaller (min-heap) or larger (max-heap) than its children. The insert operation keeps the heap ordered by *bubbling up* the new element until it finds the right place.

#### What Problem Are We Solving?

We need a data structure that efficiently gives access to the minimum (or maximum) element, while supporting fast insertions.

A binary heap offers:

- $O(1)$  access to min/max
- $O(\log n)$  insertion and deletion
- $O(n)$  build time for initial heap

Goal: Insert a new element while maintaining the heap property and completeness.

## How Does It Work (Plain Language)?

A heap is stored as an array representing a complete binary tree. Each node at index  $i$  has:

- Parent:  $(i - 1) / 2$
- Left child:  $2i + 1$
- Right child:  $2i + 2$

### Insertion Steps (Min-Heap)

1. Append the new element at the end (bottom level, rightmost).
2. Compare it with its parent.
3. If smaller (min-heap) or larger (max-heap), swap.
4. Repeat until the heap property is restored.

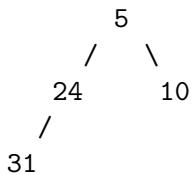
### Example (Min-Heap)

Insert sequence: [10, 24, 5, 31]

| Step      | Array           | Action                                              |
|-----------|-----------------|-----------------------------------------------------|
| Start     | [ ]             | empty                                               |
| Insert 10 | [10]            | root only                                           |
| Insert 24 | [10, 24]        | $24 > 10$ , no swap                                 |
| Insert 5  | [10, 24, 5]     | $5 < 10 \rightarrow$ swap $\rightarrow [5, 24, 10]$ |
| Insert 31 | [5, 24, 10, 31] | $31 > 24$ , no swap                                 |

Final heap: [5, 24, 10, 31]

Tree view:



## Tiny Code (Easy Versions)

C

```

#include <stdio.h>
#define MAX 100

typedef struct {
    int arr[MAX];
    int size;
} MinHeap;

void swap(int *a, int *b) {
    int tmp = *a; *a = *b; *b = tmp;
}

void heap_insert(MinHeap *h, int val) {
    int i = h->size++;
    h->arr[i] = val;

    // bubble up
    while (i > 0) {
        int parent = (i - 1) / 2;
        if (h->arr[i] >= h->arr[parent]) break;
        swap(&h->arr[i], &h->arr[parent]);
        i = parent;
    }
}

void heap_print(MinHeap *h) {
    for (int i = 0; i < h->size; i++) printf("%d ", h->arr[i]);
    printf("\n");
}

int main(void) {
    MinHeap h = {.size = 0};
    int vals[] = {10, 24, 5, 31};
    for (int i = 0; i < 4; i++) heap_insert(&h, vals[i]);
    heap_print(&h);
}

```

Python

```

class MinHeap:
    def __init__(self):
        self.arr = []

```

```

def __parent(self, i): return (i - 1) // 2

def insert(self, val):
    self.arr.append(val)
    i = len(self.arr) - 1
    while i > 0:
        p = self.__parent(i)
        if self.arr[i] >= self.arr[p]:
            break
        self.arr[i], self.arr[p] = self.arr[p], self.arr[i]
        i = p

def __repr__(self):
    return str(self.arr)

# Example
h = MinHeap()
for x in [10, 24, 5, 31]:
    h.insert(x)
print(h)

```

## Why It Matters

- Fundamental for priority queues
- Core of Dijkstra's shortest path, Prim's MST, and schedulers
- Supports efficient `insert`, `extract-min/max`, and `peek`
- Used in heapsort and event-driven simulations

## A Gentle Proof (Why It Works)

Each insertion bubbles up at most height of heap =  $\log_2 n$ . Since heap always remains complete, the structure is balanced, ensuring logarithmic operations.

Heap property is preserved because every swap ensures parent  $\leq$  child (min-heap).

## Try It Yourself

1. Insert elements in descending order  $\rightarrow$  observe bubbling.
2. Switch comparisons for max-heap.
3. Print tree level by level after each insert.

4. Implement `extract_min()` to remove root and restore heap.

### Test Cases

| Operation | Input           | Output          | Notes             |
|-----------|-----------------|-----------------|-------------------|
| Insert    | [10, 24, 5, 31] | [5, 24, 10, 31] | min-heap property |
| Insert    | [3, 2, 1]       | [1, 3, 2]       | swap chain        |
| Insert    | [10]            | [10]            | single element    |
| Insert    | []              | [x]             | empty start       |

### Edge Cases

- Full array (static heap) → resize needed
- Negative values → handled same
- Duplicate keys → order preserved

### Complexity

- Insert:  $O(\log n)$
- Search:  $O(n)$  (unsorted beyond heap property)
- Space:  $O(n)$

Binary heap insertion is the heartbeat of priority queues, each element climbs to its rightful place, one gentle swap at a time.

## 222 Binary Heap Delete

Deleting from a binary heap means removing the root element (the minimum in a min-heap or maximum in a max-heap) while keeping both the heap property and complete tree structure intact. To do this, we swap the root with the last element, remove the last, and bubble down the new root until the heap is valid again.

### What Problem Are We Solving?

We want to remove the highest-priority element quickly (min or max) from a heap, without breaking the structure.

In a min-heap, the smallest value always lives at index 0. In a max-heap, the largest value lives at index 0.

Goal: Efficiently remove the root and restore order in  $O(\log n)$  time.

## How Does It Work (Plain Language)?

1. Swap root (index 0) with last element.
2. Remove last element (now root value is gone).
3. Heapify down (bubble down) from the root:
  - Compare with children.
  - Swap with smaller (min-heap) or larger (max-heap) child.
  - Repeat until heap property restored.

Example (Min-Heap)

Start: [5, 24, 10, 31] Remove min (5):

| Step | Action                                            | Array             |
|------|---------------------------------------------------|-------------------|
| 1    | Swap root (5) with last (31)                      | [31, 24, 10, 5]   |
| 2    | Remove last                                       | [31, 24, 10]      |
| 3    | Compare 31 with children (24, 10) → smallest = 10 | swap [10, 24, 31] |
| 4    | Stop (31 > no child)                              | [10, 24, 31]      |

Result: [10, 24, 31], still a valid min-heap.

## Tiny Code (Easy Versions)

C

```
#include <stdio.h>
#define MAX 100

typedef struct {
    int arr[MAX];
    int size;
} MinHeap;

void swap(int *a, int *b) {
    int tmp = *a; *a = *b; *b = tmp;
}

void heapify_down(MinHeap *h, int i) {
    int smallest = i;
    int left = 2*i + 1;
```

```

int right = 2*i + 2;

if (left < h->size && h->arr[left] < h->arr[smallest])
    smallest = left;
if (right < h->size && h->arr[right] < h->arr[smallest])
    smallest = right;

if (smallest != i) {
    swap(&h->arr[i], &h->arr[smallest]);
    heapify_down(h, smallest);
}
}

int heap_delete_min(MinHeap *h) {
    if (h->size == 0) return -1;
    int root = h->arr[0];
    h->arr[0] = h->arr[h->size - 1];
    h->size--;
    heapify_down(h, 0);
    return root;
}

int main(void) {
    MinHeap h = {.arr = {5, 24, 10, 31}, .size = 4};
    int val = heap_delete_min(&h);
    printf("Deleted: %d\n", val);
    for (int i = 0; i < h.size; i++) printf("%d ", h.arr[i]);
    printf("\n");
}

```

Python

```

class MinHeap:
    def __init__(self):
        self.arr = []

    def _parent(self, i): return (i - 1) // 2
    def _left(self, i): return 2 * i + 1
    def _right(self, i): return 2 * i + 2

    def insert(self, val):
        self.arr.append(val)

```

```

i = len(self.arr) - 1
while i > 0 and self.arr[i] < self.arr[self._parent(i)]:
    p = self._parent(i)
    self.arr[i], self.arr[p] = self.arr[p], self.arr[i]
    i = p

def _heapify_down(self, i):
    smallest = i
    left, right = self._left(i), self._right(i)
    n = len(self.arr)
    if left < n and self.arr[left] < self.arr[smallest]:
        smallest = left
    if right < n and self.arr[right] < self.arr[smallest]:
        smallest = right
    if smallest != i:
        self.arr[i], self.arr[smallest] = self.arr[smallest], self.arr[i]
        self._heapify_down(smallest)

def delete_min(self):
    if not self.arr:
        return None
    root = self.arr[0]
    last = self.arr.pop()
    if self.arr:
        self.arr[0] = last
        self._heapify_down(0)
    return root

# Example
h = MinHeap()
for x in [5, 24, 10, 31]:
    h.insert(x)
print("Before:", h.arr)
print("Deleted:", h.delete_min())
print("After:", h.arr)

```

## Why It Matters

- Key operation in priority queues
- Core of Dijkstra's and Prim's algorithms
- Basis of heap sort (repeated delete-min)

- Ensures efficient extraction of extreme element

## A Gentle Proof (Why It Works)

Each delete operation:

- Constant-time root removal
- Logarithmic heapify-down (height of tree =  $\log n$ ) → Total cost:  $O(\log n)$

The heap property holds because every swap moves a larger (min-heap) or smaller (max-heap) element down to children, ensuring local order at each step.

## Try It Yourself

1. Delete repeatedly to sort the array (heap sort).
2. Try max-heap delete (reverse comparisons).
3. Visualize swaps after each deletion.
4. Test on ascending/descending input sequences.

## Test Cases

| Input        | Operation | Output | Heap After | Notes      |
|--------------|-----------|--------|------------|------------|
| [5,24,10,31] | delete    | 5      | [10,24,31] | valid      |
| [1,3,2]      | delete    | 1      | [2,3]      | OK         |
| [10]         | delete    | 10     | []         | empty heap |
| []           | delete    | None   | []         | safe       |

## Edge Cases

- Empty heap → return sentinel
- Single element → clears heap
- Duplicates handled naturally

## Complexity

- Delete root:  $O(\log n)$
- Space:  $O(n)$

Deleting from a heap is like removing the top card from a neat stack, replace it, sift it down, and balance restored.

## 223 Build Heap (Heapify)

Heapify (Build Heap) is the process of constructing a valid binary heap from an unsorted array in  $O(n)$  time. Instead of inserting elements one by one, we reorganize the array *in place* so every parent satisfies the heap property.

### What Problem Are We Solving?

If we insert each element individually into an empty heap, total time is  $O(n \log n)$ . But we can do better. By heapifying from the bottom up, we can build the entire heap in  $O(n)$ , crucial for heapsort and initializing priority queues efficiently.

Goal: Turn any array into a valid min-heap or max-heap quickly.

### How Does It Work (Plain Language)?

A heap stored in an array represents a complete binary tree:

- For node  $i$ , children are  $2i + 1$  and  $2i + 2$

To build the heap:

1. Start from the last non-leaf node =  $(n / 2) - 1$ .
2. Apply heapify-down (sift down) to ensure the subtree rooted at  $i$  satisfies heap property.
3. Move upwards to the root, repeating the process.

Each subtree becomes a valid heap, and when done, the whole array is a heap.

Example (Min-Heap)

Start: [31, 10, 24, 5, 12, 7]

| Step  | i | Subtree      | Action                            | Result                 |
|-------|---|--------------|-----------------------------------|------------------------|
| Start | - | full array   | -                                 | [31, 10, 24, 5, 12, 7] |
| 1     | 2 | (24, 7)      | $24 > 7 \rightarrow \text{swap}$  | [31, 10, 7, 5, 12, 24] |
| 2     | 1 | (10, 5, 12)  | $10 > 5 \rightarrow \text{swap}$  | [31, 5, 7, 10, 12, 24] |
| 3     | 0 | (31, 5, 7)   | $31 > 5 \rightarrow \text{swap}$  | [5, 31, 7, 10, 12, 24] |
| 4     | 1 | (31, 10, 12) | $31 > 10 \rightarrow \text{swap}$ | [5, 10, 7, 31, 12, 24] |

Final heap: [5, 10, 7, 31, 12, 24]

## Tiny Code (Easy Versions)

C (Bottom-Up Build Heap)

```
#include <stdio.h>

#define MAX 100

typedef struct {
    int arr[MAX];
    int size;
} MinHeap;

void swap(int *a, int *b) {
    int tmp = *a; *a = *b; *b = tmp;
}

void heapify_down(MinHeap *h, int i) {
    int smallest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < h->size && h->arr[left] < h->arr[smallest])
        smallest = left;
    if (right < h->size && h->arr[right] < h->arr[smallest])
        smallest = right;
    if (smallest != i) {
        swap(&h->arr[i], &h->arr[smallest]);
        heapify_down(h, smallest);
    }
}

void build_heap(MinHeap *h) {
    for (int i = h->size / 2 - 1; i >= 0; i--)
        heapify_down(h, i);
}

int main(void) {
    MinHeap h = {.arr = {31, 10, 24, 5, 12, 7}, .size = 6};
    build_heap(&h);
    for (int i = 0; i < h.size; i++) printf("%d ", h.arr[i]);
    printf("\n");
}
```

Python

```
class MinHeap:
    def __init__(self, arr):
        self.arr = arr
        self.size = len(arr)
        self.build_heap()

    def _left(self, i): return 2 * i + 1
    def _right(self, i): return 2 * i + 2

    def _heapify_down(self, i):
        smallest = i
        l, r = self._left(i), self._right(i)
        if l < self.size and self.arr[l] < self.arr[smallest]:
            smallest = l
        if r < self.size and self.arr[r] < self.arr[smallest]:
            smallest = r
        if smallest != i:
            self.arr[i], self.arr[smallest] = self.arr[smallest], self.arr[i]
            self._heapify_down(smallest)

    def build_heap(self):
        for i in range(self.size // 2 - 1, -1, -1):
            self._heapify_down(i)

# Example
arr = [31, 10, 24, 5, 12, 7]
h = MinHeap(arr)
print(h.arr)
```

## Why It Matters

- Builds a valid heap in  $O(n)$  (not  $O(n \log n)$ )
- Used in heapsort initialization
- Efficient for constructing priority queues from bulk data
- Guarantees balanced tree structure automatically

## A Gentle Proof (Why It Works)

Each node at depth  $d$  takes  $O(\text{height}) = O(\log(n/d))$  work. There are more nodes at lower depths (less work each), fewer at top (more work each). Sum across levels yields  $O(n)$  total time, not  $O(n \log n)$ .

Hence, bottom-up heapify is asymptotically optimal.

## Try It Yourself

1. Run with different array sizes, random orders.
2. Compare time vs inserting one-by-one.
3. Flip comparisons for max-heap.
4. Visualize swaps as a tree.

## Test Cases

| Input             | Output            | Notes          |
|-------------------|-------------------|----------------|
| [31,10,24,5,12,7] | [5,10,7,31,12,24] | valid min-heap |
| [3,2,1]           | [1,2,3]           | heapified      |
| [10]              | [10]              | single         |
| []                | []                | empty safe     |

## Edge Cases

- Already heap  $\rightarrow$  no change
- Reverse-sorted input  $\rightarrow$  many swaps
- Duplicates handled correctly

## Complexity

- Time:  $O(n)$
- Space:  $O(1)$  (in-place)

Heapify is the quiet craftsman, shaping chaos into order with gentle swaps from the ground up.

## 224 Heap Sort

Heap Sort is a classic comparison-based sorting algorithm that uses a binary heap to organize data and extract elements in sorted order. By building a max-heap (or min-heap) and repeatedly removing the root, we achieve a fully sorted array in  $O(n \log n)$  time, with no extra space.

### What Problem Are We Solving?

We want a fast, in-place sorting algorithm that:

- Doesn't require recursion (like mergesort)
- Has predictable  $O(n \log n)$  behavior
- Avoids worst-case quadratic time (like quicksort)

Goal: Use the heap's structure to repeatedly select the next largest (or smallest) element efficiently.

### How Does It Work (Plain Language)?

1. Build a max-heap from the unsorted array.
2. Repeat until heap is empty:
  - Swap root (max element) with last element.
  - Reduce heap size by one.
  - Heapify-down the new root to restore heap property.

The array becomes sorted in ascending order (for max-heap).

Example (Ascending Sort)

Start: [5, 31, 10, 24, 7]

| Step | Action             | Array              |
|------|--------------------|--------------------|
| 1    | Build max-heap     | [31, 24, 10, 5, 7] |
| 2    | Swap 31 7, heapify | [24, 7, 10, 5, 31] |
| 3    | Swap 24 5, heapify | [10, 7, 5, 24, 31] |
| 4    | Swap 10 5, heapify | [5, 7, 10, 24, 31] |
| 5    | Sorted             | [5, 7, 10, 24, 31] |

## Tiny Code (Easy Versions)

C (In-place Heap Sort)

```
#include <stdio.h>

void swap(int *a, int *b) {
    int tmp = *a; *a = *b; *b = tmp;
}

void heapify(int arr[], int n, int i) {
    int largest = i;
    int left = 2*i + 1;
    int right = 2*i + 2;

    if (left < n && arr[left] > arr[largest])
        largest = left;
    if (right < n && arr[right] > arr[largest])
        largest = right;

    if (largest != i) {
        swap(&arr[i], &arr[largest]);
        heapify(arr, n, largest);
    }
}

void heap_sort(int arr[], int n) {
    // build max-heap
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // extract elements
    for (int i = n - 1; i > 0; i--) {
        swap(&arr[0], &arr[i]);
        heapify(arr, i, 0);
    }
}

int main(void) {
    int arr[] = {5, 31, 10, 24, 7};
    int n = sizeof(arr)/sizeof(arr[0]);
    heap_sort(arr, n);
    for (int i = 0; i < n; i++) printf("%d ", arr[i]);
```

```
    printf("\n");
}
```

Python

```
def heapify(arr, n, i):
    largest = i
    l, r = 2*i + 1, 2*i + 2

    if l < n and arr[l] > arr[largest]:
        largest = l
    if r < n and arr[r] > arr[largest]:
        largest = r

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heap_sort(arr):
    n = len(arr)
    # build max-heap
    for i in range(n//2 - 1, -1, -1):
        heapify(arr, n, i)
    # extract
    for i in range(n - 1, 0, -1):
        arr[0], arr[i] = arr[i], arr[0]
        heapify(arr, i, 0)

# Example
arr = [5, 31, 10, 24, 7]
heap_sort(arr)
print(arr)
```

## Why It Matters

- Guaranteed  $O(n \log n)$  runtime (worst-case safe)
- In-place, no extra arrays
- Great teaching example of heap structure in action
- Used in systems with strict space limits

## A Gentle Proof (Why It Works)

1. Building the heap =  $O(n)$  (bottom-up heapify).
2. Each extraction =  $O(\log n)$ , repeated  $n$  times  $\rightarrow O(n \log n)$ .
3. Heap property ensures root always holds largest element.

So final array is sorted after repeated root extractions.

## Try It Yourself

1. Switch comparisons for min-heap sort (descending).
2. Print array after each swap to see process.
3. Compare with quicksort and mergesort.
4. Test large random arrays.

## Test Cases

| Input          | Output         | Notes         |
|----------------|----------------|---------------|
| [5,31,10,24,7] | [5,7,10,24,31] | ascending     |
| [10,9,8,7]     | [7,8,9,10]     | reverse input |
| [3]            | [3]            | single        |
| []             | []             | empty         |

## Edge Cases

- Repeated values handled fine
- Already sorted input still  $O(n \log n)$
- Stable? (order not preserved)

## Complexity

- Time:  $O(n \log n)$
- Space:  $O(1)$
- Stable: No

Heap sort is the mountain climber's algorithm, pulling the biggest to the top, one step at a time, until order reigns from summit to base.

## 225 Min Heap Implementation

A min-heap is a binary tree where each parent is smaller than or equal to its children. Stored compactly in an array, it guarantees  $O(1)$  access to the smallest element and  $O(\log n)$  insertion and deletion, perfect for priority queues and scheduling systems.

### What Problem Are We Solving?

We need a data structure that:

- Quickly gives us the minimum element
- Supports efficient insertions and deletions
- Maintains order dynamically

A min-heap achieves this balance, always keeping the smallest element at the root while remaining compact and complete.

### How Does It Work (Plain Language)?

A binary min-heap is stored as an array, where:

- `parent(i) = (i - 1) / 2`
- `left(i) = 2i + 1`
- `right(i) = 2i + 2`

Operations:

1. Insert(`x`):

- Add `x` at the end.
- “Bubble up” while `x < parent(x)`.

2. ExtractMin():

- Remove root.
- Move last element to root.
- “Bubble down” while `parent > smallest child`.

3. Peek():

- Return `arr[0]`.

## Example

Start: [ ] Insert sequence: 10, 24, 5, 31, 7

| Step | Action                               | Array              |
|------|--------------------------------------|--------------------|
| 1    | insert 10                            | [10]               |
| 2    | insert 24                            | [10, 24]           |
| 3    | insert 5    bubble up → swap with 10 | [5, 24, 10]        |
| 4    | insert 31                            | [5, 24, 10, 31]    |
| 5    | insert 7                             | [5, 7, 10, 31, 24] |

ExtractMin:

- Swap root with last (5 24) → [24, 7, 10, 31, 5]
- Remove last → [24, 7, 10, 31]
- Bubble down → [7, 24, 10, 31]

## Tiny Code (Easy Versions)

C

```
#include <stdio.h>
#define MAX 100

typedef struct {
    int arr[MAX];
    int size;
} MinHeap;

void swap(int *a, int *b) {
    int tmp = *a; *a = *b; *b = tmp;
}

void heapify_up(MinHeap *h, int i) {
    while (i > 0) {
        int parent = (i - 1) / 2;
        if (h->arr[i] >= h->arr[parent]) break;
        swap(&h->arr[i], &h->arr[parent]);
        i = parent;
    }
}
```

```

void heapify_down(MinHeap *h, int i) {
    int smallest = i;
    int left = 2*i + 1;
    int right = 2*i + 2;

    if (left < h->size && h->arr[left] < h->arr[smallest]) smallest = left;
    if (right < h->size && h->arr[right] < h->arr[smallest]) smallest = right;

    if (smallest != i) {
        swap(&h->arr[i], &h->arr[smallest]);
        heapify_down(h, smallest);
    }
}

void insert(MinHeap *h, int val) {
    h->arr[h->size] = val;
    heapify_up(h, h->size);
    h->size++;
}

int extract_min(MinHeap *h) {
    if (h->size == 0) return -1;
    int root = h->arr[0];
    h->arr[0] = h->arr[--h->size];
    heapify_down(h, 0);
    return root;
}

int peek(MinHeap *h) {
    return h->size > 0 ? h->arr[0] : -1;
}

int main(void) {
    MinHeap h = {.size = 0};
    int vals[] = {10, 24, 5, 31, 7};
    for (int i = 0; i < 5; i++) insert(&h, vals[i]);
    printf("Min: %d\n", peek(&h));
    printf("Extracted: %d\n", extract_min(&h));
    for (int i = 0; i < h.size; i++) printf("%d ", h.arr[i]);
    printf("\n");
}

```

Python

```
class MinHeap:
    def __init__(self):
        self.arr = []

    def _parent(self, i): return (i - 1) // 2
    def _left(self, i): return 2 * i + 1
    def _right(self, i): return 2 * i + 2

    def insert(self, val):
        self.arr.append(val)
        i = len(self.arr) - 1
        while i > 0 and self.arr[i] < self.arr[self._parent(i)]:
            p = self._parent(i)
            self.arr[i], self.arr[p] = self.arr[p], self.arr[i]
            i = p

    def extract_min(self):
        if not self.arr:
            return None
        root = self.arr[0]
        last = self.arr.pop()
        if self.arr:
            self.arr[0] = last
            self._heapify_down(0)
        return root

    def _heapify_down(self, i):
        smallest = i
        l, r = self._left(i), self._right(i)
        if l < len(self.arr) and self.arr[l] < self.arr[smallest]:
            smallest = l
        if r < len(self.arr) and self.arr[r] < self.arr[smallest]:
            smallest = r
        if smallest != i:
            self.arr[i], self.arr[smallest] = self.arr[smallest], self.arr[i]
            self._heapify_down(smallest)

    def peek(self):
        return self.arr[0] if self.arr else None

# Example
```

```

h = MinHeap()
for x in [10, 24, 5, 31, 7]:
    h.insert(x)
print("Heap:", h.arr)
print("Min:", h.peek())
print("Extracted:", h.extract_min())
print("After:", h.arr)

```

## Why It Matters

- Backbone of priority queues, Dijkstra's, Prim's, A\*
- Always gives minimum element in O(1)
- Compact array-based structure (no pointers)
- Excellent for dynamic, ordered sets

## A Gentle Proof (Why It Works)

Each insertion or deletion affects only a single path (height =  $\log n$ ). Since every swap improves local order, heap property restored in  $O(\log n)$ . At any time, parent children  $\rightarrow$  global min at root.

## Try It Yourself

1. Implement a max-heap variant.
2. Track the number of swaps per insert.
3. Combine with heap sort by repeated extract\_min.
4. Visualize heap as a tree diagram.

## Test Cases

| Operation     | Input          | Output | Heap After     | Notes     |
|---------------|----------------|--------|----------------|-----------|
| Insert        | [10,24,5,31,7] | -      | [5,7,10,31,24] | valid     |
| Peek          | -              | 5      | [5,7,10,31,24] | smallest  |
| ExtractMin    | -              | 5      | [7,24,10,31]   | reordered |
| Empty Extract | []             | None   | []             | safe      |

## Edge Cases

- Duplicates → handled fine
- Empty heap → return sentinel
- Negative numbers → no problem

## Complexity

| Operation  | Time        | Space  |
|------------|-------------|--------|
| Insert     | $O(\log n)$ | $O(1)$ |
| ExtractMin | $O(\log n)$ | $O(1)$ |
| Peek       | $O(1)$      | $O(1)$ |

A min-heap is the quiet organizer, always keeping the smallest task at the top, ready when you are.

## 226 Max Heap Implementation

A max-heap is a binary tree where each parent is greater than or equal to its children. Stored compactly in an array, it guarantees  $O(1)$  access to the largest element and  $O(\log n)$  insertion and deletion, making it ideal for scheduling, leaderboards, and priority-based systems.

### What Problem Are We Solving?

We want a data structure that efficiently maintains a dynamic set of elements while allowing us to:

- Access the largest element quickly
- Insert and delete efficiently
- Maintain order automatically

A max-heap does exactly that, it always bubbles the biggest element to the top.

### How Does It Work (Plain Language)?

A binary max-heap is stored as an array, with these relationships:

- $\text{parent}(i) = (i - 1) / 2$
- $\text{left}(i) = 2i + 1$
- $\text{right}(i) = 2i + 2$

Operations:

1. Insert( $x$ ):

- Add  $x$  at the end.
- “Bubble up” while  $x > \text{parent}(x)$ .

2. ExtractMax():

- Remove root (maximum).
- Move last element to root.
- “Bubble down” while parent < larger child.

3. Peek():

- Return  $\text{arr}[0]$  (max element).

Example

Start: [ ] Insert sequence: 10, 24, 5, 31, 7

| Step | Action    | Array                                   |
|------|-----------|-----------------------------------------|
| 1    | insert 10 | [10]                                    |
| 2    | insert 24 | bubble up $\rightarrow$ [24, 10]        |
| 3    | insert 5  | [24, 10, 5]                             |
| 4    | insert 31 | bubble up $\rightarrow$ [31, 24, 5, 10] |
| 5    | insert 7  | [31, 24, 5, 10, 7]                      |

ExtractMax:

- Swap root with last (31 7): [7, 24, 5, 10, 31]
- Remove last: [7, 24, 5, 10]
- Bubble down: swap 7 24  $\rightarrow$  [24, 10, 5, 7]

### Tiny Code (Easy Versions)

C

```

#include <stdio.h>
#define MAX 100

typedef struct {
    int arr[MAX];
    int size;
} MaxHeap;

void swap(int *a, int *b) {
    int tmp = *a; *a = *b; *b = tmp;
}

void heapify_up(MaxHeap *h, int i) {
    while (i > 0) {
        int parent = (i - 1) / 2;
        if (h->arr[i] <= h->arr[parent]) break;
        swap(&h->arr[i], &h->arr[parent]);
        i = parent;
    }
}

void heapify_down(MaxHeap *h, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < h->size && h->arr[left] > h->arr[largest]) largest = left;
    if (right < h->size && h->arr[right] > h->arr[largest]) largest = right;

    if (largest != i) {
        swap(&h->arr[i], &h->arr[largest]);
        heapify_down(h, largest);
    }
}

void insert(MaxHeap *h, int val) {
    h->arr[h->size] = val;
    heapify_up(h, h->size);
    h->size++;
}

int extract_max(MaxHeap *h) {

```

```

    if (h->size == 0) return -1;
    int root = h->arr[0];
    h->arr[0] = h->arr[--h->size];
    heapify_down(h, 0);
    return root;
}

int peek(MaxHeap *h) {
    return h->size > 0 ? h->arr[0] : -1;
}

int main(void) {
    MaxHeap h = {.size = 0};
    int vals[] = {10, 24, 5, 31, 7};
    for (int i = 0; i < 5; i++) insert(&h, vals[i]);
    printf("Max: %d\n", peek(&h));
    printf("Extracted: %d\n", extract_max(&h));
    for (int i = 0; i < h.size; i++) printf("%d ", h.arr[i]);
    printf("\n");
}

```

Python

```

class MaxHeap:
    def __init__(self):
        self.arr = []

    def _parent(self, i): return (i - 1) // 2
    def _left(self, i): return 2 * i + 1
    def _right(self, i): return 2 * i + 2

    def insert(self, val):
        self.arr.append(val)
        i = len(self.arr) - 1
        while i > 0 and self.arr[i] > self.arr[self._parent(i)]:
            p = self._parent(i)
            self.arr[i], self.arr[p] = self.arr[p], self.arr[i]
            i = p

    def extract_max(self):
        if not self.arr:
            return None

```

```

root = self.arr[0]
last = self.arr.pop()
if self.arr:
    self.arr[0] = last
    self._heapify_down(0)
return root

def _heapify_down(self, i):
    largest = i
    l, r = self._left(i), self._right(i)
    if l < len(self.arr) and self.arr[l] > self.arr[largest]:
        largest = l
    if r < len(self.arr) and self.arr[r] > self.arr[largest]:
        largest = r
    if largest != i:
        self.arr[i], self.arr[largest] = self.arr[largest], self.arr[i]
        self._heapify_down(largest)

def peek(self):
    return self.arr[0] if self.arr else None

# Example
h = MaxHeap()
for x in [10, 24, 5, 31, 7]:
    h.insert(x)
print("Heap:", h.arr)
print("Max:", h.peek())
print("Extracted:", h.extract_max())
print("After:", h.arr)

```

## Why It Matters

- Priority queues that need fast access to maximum
- Scheduling highest-priority tasks
- Tracking largest elements dynamically
- Used in heap sort, selection problems, top-k queries

## A Gentle Proof (Why It Works)

Each operation adjusts at most height =  $\log n$  levels. Since all swaps move greater elements upward, heap property (parent children) is restored in logarithmic time. Thus the root always

holds the maximum.

### Try It Yourself

1. Convert to min-heap by flipping comparisons.
2. Use heap to implement k-largest elements finder.
3. Trace swaps after each insert/delete.
4. Visualize heap as a tree diagram.

### Test Cases

| Operation     | Input          | Output | Heap After     | Notes       |
|---------------|----------------|--------|----------------|-------------|
| Insert        | [10,24,5,31,7] | -      | [31,24,5,10,7] | max at root |
| Peek          | -              | 31     | [31,24,5,10,7] | largest     |
| ExtractMax    | -              | 31     | [24,10,5,7]    | heap fixed  |
| Empty Extract | []             | None   | []             | safe        |

### Edge Cases

- Duplicates → handled fine
- Empty heap → sentinel
- Negative numbers → valid

### Complexity

| Operation  | Time        | Space  |
|------------|-------------|--------|
| Insert     | $O(\log n)$ | $O(1)$ |
| ExtractMax | $O(\log n)$ | $O(1)$ |
| Peek       | $O(1)$      | $O(1)$ |

A max-heap is the summit keeper, always crowning the greatest, ensuring the hierarchy remains true from root to leaf.

## 227 Fibonacci Heap Insert/Delete

A Fibonacci heap is a meldable heap optimized for very fast amortized operations. It keeps a collection of heap-ordered trees with minimal structural work on most operations, banking work for occasional consolidations. Classic result: `insert`, `find-min`, and `decrease-key` run in  $O(1)$  amortized, while `extract-min` runs in  $O(\log n)$  amortized.

### What Problem Are We Solving?

We want ultra fast priority queue ops in algorithms that call `decrease-key` a lot, like Dijkstra and Prim. Binary and pairing heaps give `decrease-key` in  $O(\log n)$  or good constants, but Fibonacci heaps achieve amortized  $O(1)$  for `insert` and `decrease-key`, improving theoretical bounds.

Goal: Maintain a set of heap-ordered trees where most updates only touch a few pointers, delaying consolidation until `extract-min`.

### How Does It Work (Plain Language)?

Structure highlights

- A root list of trees, each obeying min-heap order.
- A pointer to the global minimum root.
- Each node stores degree, parent, child, and a mark bit used by `decrease-key`.
- Root list is a circular doubly linked list, children lists are similar.

Core ideas

- Insert adds a 1 node tree into the root list and updates `min` if needed.
- Delete is typically implemented as `decrease-key(x, -inf)` then `extract-min`.
- Extract-min removes the min root, promotes its children to the root list, then consolidates roots by linking trees of equal degree until all root degrees are unique.

### How Does Insert Work

Steps for `insert(x)`

1. Make a singleton node `x`.
2. Splice `x` into the root list.
3. Update `min` if `x.key < min.key`.

## Example Steps (Root List View)

| Step | Action    | Root List      | Min |
|------|-----------|----------------|-----|
| 1    | insert 12 | [12]           | 12  |
| 2    | insert 7  | [12, 7]        | 7   |
| 3    | insert 25 | [12, 7, 25]    | 7   |
| 4    | insert 3  | [12, 7, 25, 3] | 3   |

All inserts are O(1) pointer splices.

## How Does Delete Work

To delete an arbitrary node  $x$ , standard approach

1. `decrease-key(x, -inf)` so  $x$  becomes the minimum.
2. `extract-min()` to remove it.

Delete inherits the `extract-min` cost.

## Tiny Code (Educational Skeleton)

This is a minimal sketch to illustrate structure and the two requested operations. It omits full `decrease-key` and cascading cuts to keep focus. In practice, a complete Fibonacci heap also implements `decrease-key` and `extract-min` with consolidation arrays.

Python (didactic skeleton for insert and delete via extract-min path)

```
class FibNode:  
    def __init__(self, key):  
        self.key = key  
        self.degree = 0  
        self.mark = False  
        self.parent = None  
        self.child = None  
        # circular doubly linked list pointers  
        self.left = self  
        self.right = self  
  
    def _splice(a, b):  
        # insert node b to the right of node a in a circular list  
        b.right = a.right
```

```

b.left = a
a.right.left = b
a.right = b

def _remove_from_list(x):
    x.left.right = x.right
    x.right.left = x.left
    x.left = x.right = x

class FibHeap:
    def __init__(self):
        self.min = None
        self.n = 0

    def insert(self, key):
        x = FibNode(key)
        if self.min is None:
            self.min = x
        else:
            _splice(self.min, x)
            if x.key < self.min.key:
                self.min = x
        self.n += 1
        return x

    def _merge_root_list(self, other_min):
        if other_min is None:
            return
        # concat circular lists: self.min and other_min
        a = self.min
        b = other_min
        a_right = a.right
        b_left = b.left
        a.right = b
        b.left = a
        a_right.left = b_left
        b_left.right = a_right
        if b.key < self.min.key:
            self.min = b

    def extract_min(self):
        z = self.min

```

```

if z is None:
    return None
# promote children to root list
if z.child:
    c = z.child
    nodes = []
    cur = c
    while True:
        nodes.append(cur)
        cur = cur.right
        if cur == c:
            break
    for x in nodes:
        x.parent = None
        _remove_from_list(x)
        _splice(z, x) # into root list near z
# remove z from root list
if z.right == z:
    self._min = None
else:
    nxt = z.right
    _remove_from_list(z)
    self._min = nxt
    self._consolidate() # real impl would link equal degree trees
self.n -= 1
return z.key

def _consolidate(self):
    # Placeholder: a full implementation uses an array A[0..floor(log_phi n)]
    # to link roots of equal degree until all degrees unique.
    pass

def delete(self, node):
    # In full Fibonacci heap:
    # decrease_key(node, -inf), then extract_min
    # Here we approximate by manual min-bump if node is the min
    # and ignore cascading cuts for brevity.
    # For correctness in real use, implement decrease_key and cuts.
    node.key = float("-inf")
    if self._min and node.key < self._min.key:
        self._min = node
    return self.extract_min()

```

```

# Example
H = FibHeap()
n1 = H.insert(12)
n2 = H.insert(7)
n3 = H.insert(25)
n4 = H.insert(3)
print("Extracted min:", H.extract_min()) # 3

```

Note This skeleton shows how `insert` stitches into the root list and how `extract_min` would promote children. A production version must implement `decrease-key` with cascading cuts and `_consolidate` that links trees of equal degree.

## Why It Matters

- Theoretical speedups for graph algorithms heavy on `decrease-key`
- Meld operation can be  $O(1)$  by concatenating root lists
- Amortized guarantees backed by potential function analysis

## A Gentle Proof (Why It Works)

Amortized analysis uses a potential function based on number of trees in the root list and number of marked nodes.

- `insert` only adds a root and maybe updates `min`, decreasing or slightly increasing potential, so  $O(1)$  amortized.
- `decrease-key` cuts a node and possibly cascades parent cuts, but marks bound the total number of cascades over a sequence, giving  $O(1)$  amortized.
- `extract-min` triggers consolidation. The number of distinct degrees is  $O(\log n)$ , so linking costs  $O(\log n)$  amortized.

## Try It Yourself

1. Complete `_consolidate` with an array indexed by degree, linking roots of equal degree until unique.
2. Implement `decrease_key(x, new_key)` with cut and cascading cut rules.
3. Add `union(H1, H2)` that concatenates root lists and picks the smaller `min`.
4. Benchmark against binary and pairing heaps on workloads with many `decrease-key` operations.

## Test Cases

| Operation    | Input              | Expected                   | Notes                                       |
|--------------|--------------------|----------------------------|---------------------------------------------|
| insert       | 12, 7, 25, 3       | min = 3                    | simple root list updates                    |
| extract_min  | after above        | returns 3                  | children promoted, consolidate              |
| delete(node) | delete 7           | 7 removed                  | via decrease to minus infinity then extract |
| meld         | union of two heaps | new min = $\min(m_1, m_2)$ | O(1) concat                                 |

## Edge Cases

- Extract from empty heap returns None
- Duplicate keys are fine
- Large inserts without extract build many small trees until consolidation

## Complexity

| Operation    | Amortized Time                     |
|--------------|------------------------------------|
| insert       | O(1)                               |
| find-min     | O(1)                               |
| decrease-key | O(1)                               |
| extract-min  | O(log n)                           |
| delete       | O(log n) via decrease then extract |

Fibonacci heaps trade strict order for lazy elegance. Most ops are tiny pointer shuffles, and the heavy lifting happens rarely during consolidation.

## 228 Pairing Heap Merge

A pairing heap is a simple, pointer-based, meldable heap that is famously fast in practice. Its secret weapon is the merge operation: link two heap roots by making the larger-key root a child of the smaller-key root. Many other operations reduce to a small number of merges.

## What Problem Are We Solving?

We want a priority queue with a tiny constant factor and very simple code, while keeping theoretical guarantees close to Fibonacci heaps. Pairing heaps offer extremely quick `insert`, `meld`, and often `decrease-key`, with `delete-min` powered by a lightweight multi-merge.

Goal: Represent a heap as a tree of nodes and implement merge so that all higher-level operations can be expressed as sequences of merges.

## How Does It Work (Plain Language)?

Each node has: key, first child, and next sibling. The heap is just a pointer to the root. To merge two heaps A and B:

1. If one is empty, return the other.
2. Compare roots.
3. Make the larger-root heap the new child of the smaller-root heap by linking it as the smaller root's first child.

Other ops via merge

- Insert(x): make a 1-node heap and `merge(root, x)`.
- Find-min: the root's key.
- Delete-min: remove the root, then merge its children in two passes
  1. Left to right, pairwise merge adjacent siblings.
  2. Right to left, merge the resulting heaps back into one.
- Decrease-key(x, new): cut x from its place, set `x.key = new`, then `merge(root, x)`.

Example Steps (Merge only)

| Step | Heap A (root) | Heap B (root) | Action          | New Root |
|------|---------------|---------------|-----------------|----------|
| 1    | 7             | 12            | link 12 under 7 | 7        |
| 2    | 7             | 3             | link 7 under 3  | 3        |
| 3    | 3             | 9             | link 9 under 3  | 3        |

Resulting root is the minimum of all merged heaps.

## Tiny Code (Easy Versions)

C (merge, insert, find-min, delete-min two-pass)

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    int key;
    struct Node *child;
    struct Node *sibling;
} Node;

Node* make_node(int key) {
    Node* n = malloc(sizeof(Node));
    n->key = key; n->child = NULL; n->sibling = NULL;
    return n;
}

Node* merge(Node* a, Node* b) {
    if (!a) return b;
    if (!b) return a;
    if (b->key < a->key) { Node* t = a; a = b; b = t; }
    // make b the first child of a
    b->sibling = a->child;
    a->child = b;
    return a;
}

Node* insert(Node* root, int key) {
    return merge(root, make_node(key));
}

Node* merge_pairs(Node* first) {
    if (!first || !first->sibling) return first;
    Node* a = first;
    Node* b = first->sibling;
    Node* rest = b->sibling;
    a->sibling = b->sibling = NULL;
    return merge(merge(a, b), merge_pairs(rest));
}

Node* delete_min(Node* root, int* out) {
```

```

if (!root) return NULL;
*out = root->key;
Node* new_root = merge_pairs(root->child);
free(root);
return new_root;
}

int main(void) {
    Node* h = NULL;
    h = insert(h, 7);
    h = insert(h, 12);
    h = insert(h, 3);
    h = insert(h, 9);
    int m;
    h = delete_min(h, &m);
    printf("Deleted min: %d\n", m); // 3
    return 0;
}

```

Python (succinct pairing heap)

```

class Node:
    __slots__ = ("key", "child", "sibling")
    def __init__(self, key):
        self.key = key
        self.child = None
        self.sibling = None

    def merge(a, b):
        if not a: return b
        if not b: return a
        if b.key < a.key:
            a, b = b, a
        b.sibling = a.child
        a.child = b
        return a

    def merge_pairs(first):
        if not first or not first.sibling:
            return first
        a, b, rest = first, first.sibling, first.sibling.sibling
        a.sibling = b.sibling = None

```

```

        return merge(merge(a, b), merge_pairs(rest))

class PairingHeap:
    def __init__(self): self.root = None
    def find_min(self): return None if not self.root else self.root.key
    def insert(self, x):
        self.root = merge(self.root, Node(x))
    def meld(self, other):
        self.root = merge(self.root, other.root)
    def delete_min(self):
        if not self.root: return None
        m = self.root.key
        self.root = merge_pairs(self.root.child)
        return m

# Example
h = PairingHeap()
for x in [7, 12, 3, 9]:
    h.insert(x)
print(h.find_min())      # 3
print(h.delete_min())    # 3
print(h.find_min())      # 7

```

## Why It Matters

- Incredibly simple code yet high performance in practice
- Meld is constant time pointer work
- Excellent for workloads mixing frequent inserts and decrease-keys
- A strong practical alternative to Fibonacci heaps

## A Gentle Proof (Why It Works)

Merge correctness

- After linking, the smaller root remains parent, so heap order holds at the root and along the newly attached subtree.

Delete-min two-pass

- Pairwise merges reduce the number of trees while keeping roots small.
- The second right-to-left fold merges larger partial heaps into a single heap.

- Analyses show `delete-min` runs in  $O(\log n)$  amortized; `insert` and `meld` in  $O(1)$  amortized.
- `decrease-key` is conjectured  $O(1)$  amortized in practice and near that in theory under common models.

### Try It Yourself

- Implement `decrease_key(node, new_key)`: cut the node from its parent and `merge(root, node)` after lowering its key.
- Add a handle table to access nodes for fast decrease-key.
- Benchmark against binary and Fibonacci heaps on Dijkstra workloads.
- Visualize delete-min's two-pass pairing on random trees.

### Test Cases

| Operation  | Input          | Output                 | Notes                  |
|------------|----------------|------------------------|------------------------|
| insert     | 7, 12, 3, 9    | min = 3                | root tracks global min |
| meld       | meld two heaps | new min is min of both | constant-time link     |
| delete_min | after above    | 3                      | two-pass pairing       |
| delete_min | next           | 7                      | heap restructures      |

### Edge Cases

- Merging with empty heap returns the other heap
- Duplicate keys behave naturally
- Single-node heap deletes to empty safely

### Complexity

| Operation    | Amortized Time                                    | Notes           |
|--------------|---------------------------------------------------|-----------------|
| meld         | $O(1)$                                            | core primitive  |
| insert       | $O(1)$                                            | merge singleton |
| find-min     | $O(1)$                                            | root key        |
| delete-min   | $O(\log n)$                                       | two-pass merge  |
| decrease-key | $O(1)$ practical, near $O(1)$ amortized in models | cut+merge       |

Pairing heaps make merging feel effortless: one comparison, a couple of pointers, and you are done.

## 229 Binomial Heap Merge

A binomial heap is a set of binomial trees that act like a binary counter for priority queues. The star move is merge: combining two heaps is like adding two binary numbers. Trees of the same degree collide, you link one under the other, and carry to the next degree.

### What Problem Are We Solving?

We want a priority queue that supports fast meld (union) while keeping simple, provable bounds. Binomial heaps deliver:

- `meld` in  $O(\log n)$
- `insert` in  $O(1)$  amortized by melding a 1-node heap
- `find-min` in  $O(\log n)$
- `delete-min` in  $O(\log n)$  via a meld with reversed children

Goal: Represent the heap as a sorted list of binomial trees and merge two heaps by walking these lists, linking equal-degree roots.

### How Does It Work (Plain Language)?

Binomial tree facts

- A binomial tree of degree  $k$  has  $2^k$  nodes.
- Each degree occurs at most once in a binomial heap.
- Roots are kept in increasing order of degree.

Merging two heaps  $H_1$  and  $H_2$

1. Merge the root lists by degree like a sorted list merge.
2. Walk the combined list. Whenever two consecutive trees have the same degree, link them: make the larger-key root a child of the smaller-key root, increasing the degree by 1.
3. Use a carry idea just like binary addition.

`Link(u, v)`

- Precondition: `degree(u) == degree(v)`.
- After linking, `min(u.key, v.key)` becomes parent and degree increases by 1.

Example Steps (degrees in parentheses)

Start

- $H_1$  roots:  $[2(0), 7(1), 12(3)]$

- H2 roots: [3(0), 9(2)]

1 Merge lists by degree

- Combined: [2(0), 3(0), 7(1), 9(2), 12(3)]

2 Resolve equal degrees

- Link 2(0) and 3(0) under min root → 2 becomes parent: 2(1)
- Now list: [2(1), 7(1), 9(2), 12(3)]
- Link 2(1) and 7(1) → 2 becomes parent: 2(2)
- Now list: [2(2), 9(2), 12(3)]
- Link 2(2) and 9(2) → 2 becomes parent: 2(3)
- Now list: [2(3), 12(3)]
- Link 2(3) and 12(3) → 2 becomes parent: 2(4)

Final heap has root list [2(4)] with 2 as global min.

### Tiny Code (Easy Versions)

C (merge plus link, minimal skeleton)

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    int key;
    int degree;
    struct Node* parent;
    struct Node* child;
    struct Node* sibling; // next root or next sibling in child list
} Node;

Node* make_node(int key){
    Node* x = calloc(1, sizeof(Node));
    x->key = key;
    return x;
}

// Make 'y' a child of 'x' assuming x->key <= y->key and degree equal
static Node* link_tree(Node* x, Node* y){
    y->parent = x;
    y->sibling = x->child;
```

```

x->child = y;
x->degree += 1;
return x;
}

// Merge root lists by degree (no linking yet)
static Node* merge_root_lists(Node* a, Node* b){
    if(!a) return b;
    if(!b) return a;
    Node dummy = {0};
    Node* tail = &dummy;
    while(a && b){
        if(a->degree <= b->degree){
            tail->sibling = a; a = a->sibling;
        }else{
            tail->sibling = b; b = b->sibling;
        }
        tail = tail->sibling;
    }
    tail->sibling = a ? a : b;
    return dummy.sibling;
}

// Union with carry logic
Node* binomial_union(Node* h1, Node* h2){
    Node* head = merge_root_lists(h1, h2);
    if(!head) return NULL;

    Node* prev = NULL;
    Node* curr = head;
    Node* next = curr->sibling;

    while(next){
        if(curr->degree != next->degree || (next->sibling && next->sibling->degree == curr->
}

```

```

        else head = next;
        curr = link_tree(next, curr);
    }
}
next = curr->sibling;
}
return head;
}

// Convenience: insert by union with 1-node heap
Node* insert(Node* heap, int key){
    return binomial_union(heap, make_node(key));
}

int main(void){
    Node* h1 = NULL;
    Node* h2 = NULL;
    h1 = insert(h1, 2);
    h1 = insert(h1, 7);
    h1 = insert(h1, 12); // degrees will normalize after unions
    h2 = insert(h2, 3);
    h2 = insert(h2, 9);
    Node* h = binomial_union(h1, h2);
    // h now holds the merged heap; find-min is a scan of root list
    for(Node* r = h; r; r = r->sibling)
        printf("root key=%d deg=%d\n", r->key, r->degree);
    return 0;
}

```

Python (succinct union and link)

```

class Node:
    __slots__ = ("key", "degree", "parent", "child", "sibling")
    def __init__(self, key):
        self.key = key
        self.degree = 0
        self.parent = None
        self.child = None
        self.sibling = None

def link_tree(x, y):
    # assume x.key <= y.key and degrees equal

```

```

y.parent = x
y.sibling = x.child
x.child = y
x.degree += 1
return x

def merge_root_lists(a, b):
    if not a: return b
    if not b: return a
    dummy = Node(-1)
    t = dummy
    while a and b:
        if a.degree <= b.degree:
            t.sibling, a = a, a.sibling
        else:
            t.sibling, b = b, b.sibling
        t = t.sibling
    t.sibling = a if a else b
    return dummy.sibling

def binomial_union(h1, h2):
    head = merge_root_lists(h1, h2)
    if not head: return None
    prev, curr, nxt = None, head, head.sibling
    while nxt:
        if curr.degree != nxt.degree or (nxt.sibling and nxt.sibling.degree == curr.degree):
            prev, curr, nxt = curr, nxt, nxt.sibling
        else:
            if curr.key <= nxt.key:
                curr.sibling = nxt.sibling
                curr = link_tree(curr, nxt)
                nxt = curr.sibling
            else:
                if prev: prev.sibling = nxt
                else: head = nxt
                curr = link_tree(nxt, curr)
                nxt = curr.sibling
    return head

def insert(heap, key):
    return binomial_union(heap, Node(key))

```

```

# Example
h1 = None
for x in [2, 7, 12]:
    h1 = insert(h1, x)
h2 = None
for x in [3, 9]:
    h2 = insert(h2, x)
h = binomial_union(h1, h2)
roots = []
r = h
while r:
    roots.append((r.key, r.degree))
    r = r.sibling
print(roots) # e.g. [(2, 4)] or a small set of unique degrees with min root smallest

```

## Why It Matters

- Meld-friendly: merging heaps is first-class, not an afterthought
- Clean, provable bounds with a binary-counter intuition
- Foundation for Fibonacci heaps and variations
- Great when frequent melds are required in algorithms or multi-queue systems

## A Gentle Proof (Why It Works)

Merging root lists produces degrees in nondecreasing order. Linking only happens between adjacent roots of the same degree, producing exactly one tree of each degree after all carries settle. This mirrors binary addition: each link corresponds to carrying a 1 to the next bit. Since the maximum degree is  $O(\log n)$ , the merge performs  $O(\log n)$  links and scans, giving  $O(\log n)$  time.

## Try It Yourself

1. Implement `find_min` by scanning root list and keep a pointer to the min root.
2. Implement `delete_min`: remove min root, reverse its child list into a separate heap, then `union`.
3. Add `decrease_key` by cutting and reinserting a node in the root list, then fixing parent order.
4. Compare union time with binary heap and pairing heap on large random workloads.

## Test Cases

| Case             | H1 Roots (deg)           | H2 Roots (deg) | Result                   | Notes                 |
|------------------|--------------------------|----------------|--------------------------|-----------------------|
| Simple merge     | [2(0), 7(1)]             | [3(0)]         | roots unique after link  | 2 becomes parent of 3 |
| Chain of carries | [2(0), 7(1), 12(3)]      | [3(0), 9(2)]   | single root 2(4)         | cascading links       |
| Insert by union  | H with roots plus [x(0)] |                | merged in O(1) amortized | single carry possible |

## Edge Cases

- Merging with empty heap returns the other heap
- Duplicate keys work; tie break arbitrarily
- Maintain stable sibling pointers when linking

## Complexity

| Operation    | Time                                             |
|--------------|--------------------------------------------------|
| meld (union) | $O(\log n)$                                      |
| insert       | $O(1)$ amortized via union with 1-node heap      |
| find-min     | $O(\log n)$ scan or keep pointer for $O(1)$ peek |
| delete-min   | $O(\log n)$                                      |
| decrease-key | $O(\log n)$ typical implementation               |

Merging binomial heaps feels like adding binary numbers: equal degrees collide, link, and carry forward until the structure is tidy and the minimum stands at a root.

## 230 Leftist Heap Merge

A leftist heap is a binary tree heap optimized for efficient merges. Its structure skews to the left so that merging two heaps can be done recursively in  $O(\log n)$  time. The clever trick is storing each node's null path length (npl), ensuring the shortest path to a null child is always on the right, which keeps merges shallow.

## What Problem Are We Solving?

We want a merge-friendly heap with simpler structure than Fibonacci or pairing heaps but faster merges than standard binary heaps. The leftist heap is a sweet spot: fast merges, simple code, and still supports all key heap operations.

Goal: Design a heap that keeps its shortest subtree on the right, so recursive merges stay logarithmic.

## How Does It Work (Plain Language)?

Each node stores:

- `key` – value used for ordering
- `left`, `right` – child pointers
- `npl` – null path length (distance to nearest null)

Rules:

1. Heap order: parent key child keys (min-heap)
2. Leftist property:  $npl(left) \leq npl(right)$

Merge(`a`, `b`):

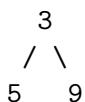
1. If one is null, return the other.
2. Compare roots, smaller root becomes new root.
3. Recursively merge `a.right` and `b`.
4. After merge, swap children if needed to keep leftist property.
5. Update `npl`.

Other operations via merge

- Insert(`x`): merge heap with single-node heap `x`.
- DeleteMin(): merge left and right subtrees of root.

Example (Min-Heap Merge)

Heap A: root 3



Heap B: root 4

```
4
/ \
8  10
```

Merge(3, 4):

- $3 < 4 \rightarrow$  new root 3
- Merge right(9) with heap 4
- After merge:

```
3
/ \
5  4
 / \
8  10
 /
9
```

Rebalance by swapping if right's npl > left's  $\rightarrow$  ensures leftist shape.

### Tiny Code (Easy Versions)

C (Merge, Insert, Delete-Min)

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    int key;
    int npl;
    struct Node *left, *right;
} Node;

Node* make_node(int key) {
    Node* n = malloc(sizeof(Node));
    n->key = key;
    n->npl = 0;
    n->left = n->right = NULL;
    return n;
}
```

```

int npl(Node* x) { return x ? x->npl : -1; }

Node* merge(Node* a, Node* b) {
    if (!a) return b;
    if (!b) return a;
    if (b->key < a->key) { Node* t = a; a = b; b = t; }
    a->right = merge(a->right, b);
    // maintain leftist property
    if (npl(a->left) < npl(a->right)) {
        Node* t = a->left;
        a->left = a->right;
        a->right = t;
    }
    a->npl = npl(a->right) + 1;
    return a;
}

Node* insert(Node* h, int key) {
    return merge(h, make_node(key));
}

Node* delete_min(Node* h, int* out) {
    if (!h) return NULL;
    *out = h->key;
    Node* new_root = merge(h->left, h->right);
    free(h);
    return new_root;
}

int main(void) {
    Node* h1 = NULL;
    int vals[] = {5, 3, 9, 7, 4};
    for (int i = 0; i < 5; i++)
        h1 = insert(h1, vals[i]);

    int m;
    h1 = delete_min(h1, &m);
    printf("Deleted min: %d\n", m);
    return 0;
}

```

Python

```

class Node:
    __slots__ = ("key", "npl", "left", "right")
    def __init__(self, key):
        self.key = key
        self.npl = 0
        self.left = None
        self.right = None

def npl(x): return x.npl if x else -1

def merge(a, b):
    if not a: return b
    if not b: return a
    if b.key < a.key:
        a, b = b, a
    a.right = merge(a.right, b)
    # enforce leftist property
    if npl(a.left) < npl(a.right):
        a.left, a.right = a.right, a.left
    a.npl = npl(a.right) + 1
    return a

def insert(h, key):
    return merge(h, Node(key))

def delete_min(h):
    if not h: return None, None
    m = h.key
    h = merge(h.left, h.right)
    return m, h

# Example
h = None
for x in [5, 3, 9, 7, 4]:
    h = insert(h, x)
m, h = delete_min(h)
print("Deleted min:", m)

```

## Why It Matters

- Fast merge with simple recursion

- Ideal for priority queues with frequent unions
- Cleaner implementation than Fibonacci heaps
- Guarantees logarithmic merge and delete-min

## A Gentle Proof (Why It Works)

The null path length ( $npl$ ) ensures that the right spine is always the shortest. Therefore, each merge step recurses down only one right path, not both. This bounds recursion depth by  $O(\log n)$ . Every other operation (insert, delete-min) is defined as a small number of merges, hence  $O(\log n)$ .

## Try It Yourself

1. Trace the merge process for two 3-node heaps.
2. Visualize  $npl$  values after each step.
3. Implement `find_min()` (just return root key).
4. Try making a max-heap variant by flipping comparisons.

## Test Cases

| Operation   | Input         | Output     | Notes                       |
|-------------|---------------|------------|-----------------------------|
| Insert      | [5,3,9,7,4]   | root=3     | min-heap property holds     |
| DeleteMin   | remove 3      | new root=4 | leftist property maintained |
| Merge       | [3,5] + [4,6] | root=3     | right spine $\log n$        |
| Empty merge | None + [5]    | [5]        | safe                        |

## Edge Cases

- Merge with null heap → returns the other
- Duplicate keys → ties fine
- Single node →  $npl = 0$

## Complexity

| Operation | Time        | Space  |
|-----------|-------------|--------|
| Merge     | $O(\log n)$ | $O(1)$ |
| Insert    | $O(\log n)$ | $O(1)$ |
| DeleteMin | $O(\log n)$ | $O(1)$ |

| Operation | Time | Space |
|-----------|------|-------|
| FindMin   | O(1) | O(1)  |

The leftist heap is like a river always bending toward the left, shaping itself so merges flow swiftly and naturally.

## Section 24. Balanced Trees

### 231 AVL Tree Insert

An AVL tree is a self-balancing binary search tree where the height difference (balance factor) between the left and right subtrees of any node is at most 1. This invariant guarantees O(log n) lookup, insertion, and deletion, making AVL trees a classic example of maintaining order dynamically.

#### What Problem Are We Solving?

Ordinary binary search trees can become skewed (like linked lists) after unlucky insertions, degrading performance to O(n). The AVL tree restores balance automatically after each insertion, ensuring searches and updates stay fast.

Goal: Maintain a balanced search tree by rotating nodes after insertions so that height difference 1 everywhere.

#### How Does It Work (Plain Language)?

1. Insert the key as in a normal BST.
2. Walk back up the recursion updating heights.
3. Check balance factor = `height(left) - height(right)`.
4. If it's outside  $\{-1, 0, +1\}$ , perform one of four rotations to restore balance:

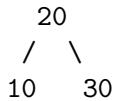
| Case             | Pattern                           | Fix                              |
|------------------|-----------------------------------|----------------------------------|
| Left-Left (LL)   | Inserted into left-left subtree   | Rotate right                     |
| Right-Right (RR) | Inserted into right-right subtree | Rotate left                      |
| Left-Right (LR)  | Inserted into left-right subtree  | Rotate left at child, then right |
| Right-Left (RL)  | Inserted into right-left subtree  | Rotate right at child, then left |

Example

Insert 30, 20, 10

- 30 → root
- 20 → left of 30
- 10 → left of 20 → imbalance at 30: balance factor = 2 → LL case → right rotate on 30

Balanced tree:

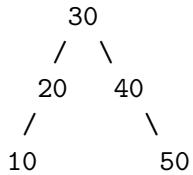


### Step-by-Step Example

Insert sequence: 10, 20, 30, 40, 50

| Step | Insert | Tree (in-order)  | Imbalance  | Rotation | Root After |
|------|--------|------------------|------------|----------|------------|
| 1    | 10     | [10]             | -          | -        | 10         |
| 2    | 20     | [10,20]          | balanced   | -        | 10         |
| 3    | 30     | [10,20,30]       | at 10 (RR) | left     | 20         |
| 4    | 40     | [10,20,30,40]    | balanced   | -        | 20         |
| 5    | 50     | [10,20,30,40,50] | at 20 (RR) | left     | 30         |

Balanced final tree:



### Tiny Code (Easy Versions)

C

```

#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    int key, height;
    struct Node *left, *right;
} Node;

int height(Node* n) { return n ? n->height : 0; }
int max(int a, int b) { return a > b ? a : b; }

Node* new_node(int key) {
    Node* n = malloc(sizeof(Node));
    n->key = key;
    n->height = 1;
    n->left = n->right = NULL;
    return n;
}

Node* rotate_right(Node* y) {
    Node* x = y->left;
    Node* T2 = x->right;
    x->right = y;
    y->left = T2;
    y->height = 1 + max(height(y->left), height(y->right));
    x->height = 1 + max(height(x->left), height(x->right));
    return x;
}

Node* rotate_left(Node* x) {
    Node* y = x->right;
    Node* T2 = y->left;
    y->left = x;
    x->right = T2;
    x->height = 1 + max(height(x->left), height(x->right));
    y->height = 1 + max(height(y->left), height(y->right));
    return y;
}

int balance(Node* n) { return n ? height(n->left) - height(n->right) : 0; }

Node* insert(Node* node, int key) {

```

```

if (!node) return new_node(key);
if (key < node->key) node->left = insert(node->left, key);
else if (key > node->key) node->right = insert(node->right, key);
else return node; // no duplicates

node->height = 1 + max(height(node->left), height(node->right));
int bf = balance(node);

// LL
if (bf > 1 && key < node->left->key)
    return rotate_right(node);
// RR
if (bf < -1 && key > node->right->key)
    return rotate_left(node);
// LR
if (bf > 1 && key > node->left->key) {
    node->left = rotate_left(node->left);
    return rotate_right(node);
}
// RL
if (bf < -1 && key < node->right->key) {
    node->right = rotate_right(node->right);
    return rotate_left(node);
}
return node;
}

void inorder(Node* root) {
    if (!root) return;
    inorder(root->left);
    printf("%d ", root->key);
    inorder(root->right);
}

int main(void) {
    Node* root = NULL;
    int keys[] = {10, 20, 30, 40, 50};
    for (int i = 0; i < 5; i++)
        root = insert(root, keys[i]);
    inorder(root);
    printf("\n");
}

```

Python

```
class Node:
    def __init__(self, key):
        self.key = key
        self.left = self.right = None
        self.height = 1

    def height(n): return n.height if n else 0
    def balance(n): return height(n.left) - height(n.right) if n else 0

    def rotate_right(y):
        x, T2 = y.left, y.left.right
        x.right, y.left = y, T2
        y.height = 1 + max(height(y.left), height(y.right))
        x.height = 1 + max(height(x.left), height(x.right))
        return x

    def rotate_left(x):
        y, T2 = x.right, x.right.left
        y.left, x.right = x, T2
        x.height = 1 + max(height(x.left), height(x.right))
        y.height = 1 + max(height(y.left), height(y.right))
        return y

    def insert(node, key):
        if not node: return Node(key)
        if key < node.key: node.left = insert(node.left, key)
        elif key > node.key: node.right = insert(node.right, key)
        else: return node

        node.height = 1 + max(height(node.left), height(node.right))
        bf = balance(node)

        if bf > 1 and key < node.left.key: # LL
            return rotate_right(node)
        if bf < -1 and key > node.right.key: # RR
            return rotate_left(node)
        if bf > 1 and key > node.left.key: # LR
            node.left = rotate_left(node.left)
            return rotate_right(node)
        if bf < -1 and key < node.right.key: # RL
            node.right = rotate_right(node.right)
```

```

        return rotate_left(node)
    return node

def inorder(root):
    if not root: return
    inorder(root.left)
    print(root.key, end=' ')
    inorder(root.right)

# Example
root = None
for k in [10, 20, 30, 40, 50]:
    root = insert(root, k)
inorder(root)

```

## Why It Matters

- Guarantees  $O(\log n)$  operations
- Prevents degeneration into linear chains
- Clear rotation-based balancing logic
- Basis for other balanced trees (e.g., Red-Black)

## A Gentle Proof (Why It Works)

Each insertion may unbalance at most one node, the lowest ancestor of the inserted node. A single rotation (or double rotation) restores the balance factor of that node to  $\{-1, 0, +1\}$ , and updates all affected heights in constant time. Thus each insertion performs  $O(1)$  rotations,  $O(\log n)$  recursive updates.

## Try It Yourself

1. Insert 30, 20, 10 → LL case
2. Insert 10, 30, 20 → LR case
3. Insert 30, 10, 20 → RL case
4. Insert 10, 20, 30 → RR case
5. Draw each tree before and after rotation

## Test Cases

| Sequence     | Rotation Type | Final Root | Height |
|--------------|---------------|------------|--------|
| [30, 20, 10] | LL            | 20         | 2      |
| [10, 30, 20] | LR            | 20         | 2      |
| [30, 10, 20] | RL            | 20         | 2      |
| [10, 20, 30] | RR            | 20         | 2      |

## Edge Cases

- Duplicate keys ignored
- Insertion into empty tree → new node
- All ascending or descending inserts → balanced via rotations

## Complexity

| Operation | Time        | Space            |
|-----------|-------------|------------------|
| Insert    | $O(\log n)$ | $O(h)$ recursion |
| Search    | $O(\log n)$ | $O(h)$           |
| Delete    | $O(\log n)$ | $O(h)$           |

The AVL tree is a careful gardener, pruning imbalance wherever it grows, so your searches always find a straight path.

## 232 AVL Tree Delete

Deleting a node in an AVL tree is like removing a block from a carefully balanced tower, you take it out, then perform rotations to restore equilibrium. The key is to combine BST deletion rules with balance factor checks and rebalancing up the path.

### What Problem Are We Solving?

Deletion in a plain BST can break the shape, making it skewed and inefficient. In an AVL tree, we want to:

1. Remove a node (using standard BST deletion)
2. Recalculate heights and balance factors
3. Restore balance with rotations

This ensures the tree remains height-balanced, keeping operations at  $O(\log n)$ .

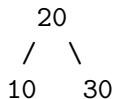
## How Does It Work (Plain Language)?

1. Find the node as in a normal BST.
2. Delete it:
  - If leaf  $\rightarrow$  remove directly.
  - If one child  $\rightarrow$  replace with child.
  - If two children  $\rightarrow$  find inorder successor, copy its value, delete it recursively.
3. Walk upward, update height and balance factor at each ancestor.
4. Apply one of four rotation cases if unbalanced:

| Case | Condition                           | Fix                              |
|------|-------------------------------------|----------------------------------|
| LL   | balance > 1 and balance(left) = 0   | Rotate right                     |
| LR   | balance > 1 and balance(left) < 0   | Rotate left at child, then right |
| RR   | balance < -1 and balance(right) = 0 | Rotate left                      |
| RL   | balance < -1 and balance(right) > 0 | Rotate right at child, then left |

## Example

Delete 10 from



- Remove leaf 10
- Node 20: balance = 0  $\rightarrow$  balanced

Now delete 30:



- $\text{balance}(20) = +1 \rightarrow$  still balanced

Delete 10 next:

20

Tree becomes single node, still AVL.

## Step-by-Step Example

Insert [10, 20, 30, 40, 50, 25] Then delete 40

| Step | Action       | Imbalance            | Rotation | Root |
|------|--------------|----------------------|----------|------|
| 1    | Delete 40    | at 30 (balance = -2) | RL       | 30   |
| 2    | After rotate | balanced             | -        | 30   |

## Tiny Code (Easy Versions)

C

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    int key, height;
    struct Node *left, *right;
} Node;

int height(Node* n) { return n ? n->height : 0; }
int max(int a, int b) { return a > b ? a : b; }

Node* new_node(int key) {
    Node* n = malloc(sizeof(Node));
    n->key = key; n->height = 1;
    n->left = n->right = NULL;
    return n;
}

Node* rotate_right(Node* y) {
    Node* x = y->left;
    Node* T2 = x->right;
    x->right = y;
    y->left = T2;
    y->height = 1 + max(height(y->left), height(y->right));
    x->height = 1 + max(height(x->left), height(x->right));
    return x;
}

Node* rotate_left(Node* x) {
```

```

Node* y = x->right;
Node* T2 = y->left;
y->left = x;
x->right = T2;
x->height = 1 + max(height(x->left), height(x->right));
y->height = 1 + max(height(y->left), height(y->right));
return y;
}

int balance(Node* n) { return n ? height(n->left) - height(n->right) : 0; }

Node* min_node(Node* n) {
    Node* cur = n;
    while (cur->left) cur = cur->left;
    return cur;
}

Node* insert(Node* root, int key) {
    if (!root) return new_node(key);
    if (key < root->key) root->left = insert(root->left, key);
    else if (key > root->key) root->right = insert(root->right, key);
    else return root;

    root->height = 1 + max(height(root->left), height(root->right));
    int bf = balance(root);

    // Rebalance
    if (bf > 1 && key < root->left->key) return rotate_right(root);
    if (bf < -1 && key > root->right->key) return rotate_left(root);
    if (bf > 1 && key > root->left->key) {
        root->left = rotate_left(root->left);
        return rotate_right(root);
    }
    if (bf < -1 && key < root->right->key) {
        root->right = rotate_right(root->right);
        return rotate_left(root);
    }
    return root;
}

Node* delete(Node* root, int key) {
    if (!root) return root;

```

```

if (key < root->key) root->left = delete(root->left, key);
else if (key > root->key) root->right = delete(root->right, key);
else {
    // Node with one or no child
    if (!root->left || !root->right) {
        Node* tmp = root->left ? root->left : root->right;
        if (!tmp) { tmp = root; root = NULL; }
        else *root = *tmp;
        free(tmp);
    } else {
        Node* tmp = min_node(root->right);
        root->key = tmp->key;
        root->right = delete(root->right, tmp->key);
    }
}
if (!root) return root;

root->height = 1 + max(height(root->left), height(root->right));
int bf = balance(root);

// Rebalance
if (bf > 1 && balance(root->left) >= 0)
    return rotate_right(root);
if (bf > 1 && balance(root->left) < 0) {
    root->left = rotate_left(root->left);
    return rotate_right(root);
}
if (bf < -1 && balance(root->right) <= 0)
    return rotate_left(root);
if (bf < -1 && balance(root->right) > 0) {
    root->right = rotate_right(root->right);
    return rotate_left(root);
}
return root;
}

void inorder(Node* r) {
    if (!r) return;
    inorder(r->left);
    printf("%d ", r->key);
    inorder(r->right);
}

```

```

int main(void) {
    Node* root = NULL;
    int keys[] = {10, 20, 30, 40, 50, 25};
    for (int i = 0; i < 6; i++) root = insert(root, keys[i]);
    root = delete(root, 40);
    inorder(root);
    printf("\n");
}

```

Python

```

class Node:
    def __init__(self, key):
        self.key = key
        self.height = 1
        self.left = self.right = None

    def height(n): return n.height if n else 0
    def balance(n): return height(n.left) - height(n.right) if n else 0

    def rotate_right(y):
        x, T2 = y.left, y.left.right
        x.right, y.left = y, T2
        y.height = 1 + max(height(y.left), height(y.right))
        x.height = 1 + max(height(x.left), height(x.right))
        return x

    def rotate_left(x):
        y, T2 = x.right, x.right.left
        y.left, x.right = x, T2
        x.height = 1 + max(height(x.left), height(x.right))
        y.height = 1 + max(height(y.left), height(y.right))
        return y

    def min_node(n):
        while n.left: n = n.left
        return n

    def insert(r, k):
        if not r: return Node(k)
        if k < r.key: r.left = insert(r.left, k)
        elif k > r.key: r.right = insert(r.right, k)

```

```

else: return r
r.height = 1 + max(height(r.left), height(r.right))
bf = balance(r)
if bf > 1 and k < r.left.key: return rotate_right(r)
if bf < -1 and k > r.right.key: return rotate_left(r)
if bf > 1 and k > r.left.key:
    r.left = rotate_left(r.left); return rotate_right(r)
if bf < -1 and k < r.right.key:
    r.right = rotate_right(r.right); return rotate_left(r)
return r

def delete(r, k):
    if not r: return r
    if k < r.key: r.left = delete(r.left, k)
    elif k > r.key: r.right = delete(r.right, k)
    else:
        if not r.left: return r.right
        elif not r.right: return r.left
        temp = min_node(r.right)
        r.key = temp.key
        r.right = delete(r.right, temp.key)
    r.height = 1 + max(height(r.left), height(r.right))
    bf = balance(r)
    if bf > 1 and balance(r.left) >= 0: return rotate_right(r)
    if bf > 1 and balance(r.left) < 0:
        r.left = rotate_left(r.left); return rotate_right(r)
    if bf < -1 and balance(r.right) <= 0: return rotate_left(r)
    if bf < -1 and balance(r.right) > 0:
        r.right = rotate_right(r.right); return rotate_left(r)
    return r

def inorder(r):
    if not r: return
    inorder(r.left); print(r.key, end=' '); inorder(r.right)

root = None
for k in [10,20,30,40,50,25]:
    root = insert(root, k)
root = delete(root, 40)
inorder(root)

```

## Why It Matters

- Keeps search, insert, delete all  $O(\log n)$
- Auto-rebalances after removal
- Shows how rotations maintain structure consistency
- Foundation for all balanced trees (like Red-Black, AVL variants)

## A Gentle Proof (Why It Works)

Every deletion changes subtree height by at most 1. Each ancestor's balance factor is recomputed; if imbalance found, a single rotation (or double rotation) restores balance. At most  $O(\log n)$  nodes are visited, and each fix is  $O(1)$ .

## Try It Yourself

1. Build [10, 20, 30, 40, 50, 25], then delete 50
2. Observe RR rotation at 30
3. Delete 10, check rebalancing at 20
4. Delete all sequentially, confirm sorted order

## Test Cases

| Insert Sequence     | Delete | Rotation | Root After | Balanced |
|---------------------|--------|----------|------------|----------|
| [10,20,30,40,50,25] | 40     | RL       | 30         |          |
| [10,20,30]          | 10     | RR       | 20         |          |
| [30,20,10]          | 30     | LL       | 20         |          |

## Edge Cases

- Deleting from empty tree  $\rightarrow$  safe
- Single node  $\rightarrow$  becomes NULL
- Duplicate keys  $\rightarrow$  ignored

| Operation | Time | Space |
|-----------|------|-------|
|-----------|------|-------|

## Complexity

| Operation | Time        | Space            |
|-----------|-------------|------------------|
| Delete    | $O(\log n)$ | $O(h)$ recursion |
| Search    | $O(\log n)$ | $O(h)$           |
| Insert    | $O(\log n)$ | $O(h)$           |

An AVL deletion is a gentle art, remove the key, rebalance the branches, and harmony is restored.

## 233 Red-Black Tree Insert

A Red-Black Tree (RBT) is a self-balancing binary search tree that uses color bits (red or black) to control balance indirectly. Unlike AVL trees that balance by height, RBTs balance by color rules, allowing more flexible, faster insertions with fewer rotations.

### What Problem Are We Solving?

A plain BST can degrade into a linked list with  $O(n)$  operations. An RBT maintains a near-balanced height, ensuring  $O(\log n)$  for search, insert, and delete. Instead of exact height balance like AVL, RBTs enforce color invariants that keep paths roughly equal.

### Red-Black Tree Properties

1. Each node is red or black.
2. The root is always black.
3. Null (NIL) nodes are considered black.
4. No two consecutive red nodes (no red parent with red child).
5. Every path from a node to its descendant NIL nodes has the same number of black nodes.

These rules guarantee height  $2 \times \log(n + 1)$ .

## How Does It Work (Plain Language)?

1. Insert node like in a normal BST (color it red).
2. Fix violations if any property breaks.
3. Use rotations and recoloring based on where the red node appears.

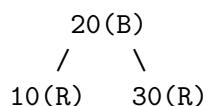
| Case   | Condition                                 | Fix                                           |
|--------|-------------------------------------------|-----------------------------------------------|
| Case 1 | New node is root                          | Recolor black                                 |
| Case 2 | Parent black                              | No fix needed                                 |
| Case 3 | Parent red, Uncle red                     | Recolor parent & uncle black, grandparent red |
| Case 4 | Parent red, Uncle black, Triangle (LR/RL) | Rotate to line up                             |
| Case 5 | Parent red, Uncle black, Line (LL/RR)     | Rotate and recolor grandparent                |

## Example

Insert sequence: 10, 20, 30

1. Insert 10 → root → black
2. Insert 20 → red child → balanced
3. Insert 30 → red parent (20) → Case 5 (RR) → rotate left on 10 → recolor root black, children red

Result:



## Step-by-Step Example

Insert [7, 3, 18, 10, 22, 8, 11, 26]

| Step | Insert | Violation    | Fix        | Root |
|------|--------|--------------|------------|------|
| 1    | 7      | root         | make black | 7(B) |
| 2    | 3      | parent black | none       | 7(B) |

| Step | Insert | Violation                   | Fix              | Root |
|------|--------|-----------------------------|------------------|------|
| 3    | 18     | parent black                | none             | 7(B) |
| 4    | 10     | parent red, uncle red       | recolor, move up | 7(B) |
| 5    | 22     | parent black                | none             | 7(B) |
| 6    | 8      | parent red, uncle red       | recolor          | 7(B) |
| 7    | 11     | parent red, uncle black, LR | rotate + recolor | 7(B) |
| 8    | 26     | parent black                | none             | 7(B) |

Final tree balanced by color invariants.

### Tiny Code (Easy Versions)

C (Simplified)

```
#include <stdio.h>
#include <stdlib.h>

typedef enum { RED, BLACK } Color;

typedef struct Node {
    int key;
    Color color;
    struct Node *left, *right, *parent;
} Node;

Node* new_node(int key) {
    Node* n = malloc(sizeof(Node));
    n->key = key;
    n->color = RED;
    n->left = n->right = n->parent = NULL;
    return n;
}

void rotate_left(Node root, Node* x) {
    Node* y = x->right;
    x->right = y->left;
    if (y->left) y->left->parent = x;
    y->parent = x->parent;
    if (!x->parent) *root = y;
    else if (x == x->parent->left) x->parent->left = y;
```

```

    else x->parent->right = y;
    y->left = x;
    x->parent = y;
}

void rotate_right(Node root, Node* y) {
    Node* x = y->left;
    y->left = x->right;
    if (x->right) x->right->parent = y;
    x->parent = y->parent;
    if (!y->parent) *root = x;
    else if (y == y->parent->left) y->parent->left = x;
    else y->parent->right = x;
    x->right = y;
    y->parent = x;
}

void fixViolation(Node root, Node* z) {
    while (z->parent && z->parent->color == RED) {
        Node* gp = z->parent->parent;
        if (z->parent == gp->left) {
            Node* uncle = gp->right;
            if (uncle && uncle->color == RED) {
                z->parent->color = BLACK;
                uncle->color = BLACK;
                gp->color = RED;
                z = gp;
            } else {
                if (z == z->parent->right) {
                    z = z->parent;
                    rotate_left(root, z);
                }
                z->parent->color = BLACK;
                gp->color = RED;
                rotate_right(root, gp);
            }
        } else {
            Node* uncle = gp->left;
            if (uncle && uncle->color == RED) {
                z->parent->color = BLACK;
                uncle->color = BLACK;
                gp->color = RED;
            }
        }
    }
}

```

```

        z = gp;
    } else {
        if (z == z->parent->left) {
            z = z->parent;
            rotate_right(root, z);
        }
        z->parent->color = BLACK;
        gp->color = RED;
        rotate_left(root, gp);
    }
}
(*root)->color = BLACK;
}

Node* bst_insert(Node* root, Node* z) {
    if (!root) return z;
    if (z->key < root->key) {
        root->left = bst_insert(root->left, z);
        root->left->parent = root;
    } else if (z->key > root->key) {
        root->right = bst_insert(root->right, z);
        root->right->parent = root;
    }
    return root;
}

void insert(Node root, int key) {
    Node* z = new_node(key);
    *root = bst_insert(*root, z);
    fixViolation(root, z);
}

void inorder(Node* r) {
    if (!r) return;
    inorder(r->left);
    printf("%d(%c) ", r->key, r->color == RED ? 'R' : 'B');
    inorder(r->right);
}

int main(void) {
    Node* root = NULL;
}

```

```

int keys[] = {10, 20, 30};
for (int i = 0; i < 3; i++) insert(&root, keys[i]);
inorder(root);
printf("\n");
}

```

Python (Simplified)

```

class Node:
    def __init__(self, key, color="R", parent=None):
        self.key = key
        self.color = color
        self.left = self.right = None
        self.parent = parent

    def rotate_left(self, x):
        y = x.right
        x.right = y.left
        if y.left: y.left.parent = x
        y.parent = x.parent
        if not x.parent: root = y
        elif x == x.parent.left: x.parent.left = y
        else: x.parent.right = y
        y.left = x
        x.parent = y
        return root

    def rotate_right(self, y):
        x = y.left
        y.left = x.right
        if x.right: x.right.parent = y
        x.parent = y.parent
        if not y.parent: root = x
        elif y == y.parent.left: y.parent.left = x
        else: y.parent.right = x
        x.right = y
        y.parent = x
        return root

    def fixViolation(self, z):
        while z.parent and z.parent.color == "R":
            gp = z.parent.parent

```

```

if z.parent == gp.left:
    uncle = gp.right
    if uncle and uncle.color == "R":
        z.parent.color = uncle.color = "B"
        gp.color = "R"
        z = gp
    else:
        if z == z.parent.right:
            z = z.parent
            root = rotate_left(root, z)
        z.parent.color = "B"
        gp.color = "R"
        root = rotate_right(root, gp)
else:
    uncle = gp.left
    if uncle and uncle.color == "R":
        z.parent.color = uncle.color = "B"
        gp.color = "R"
        z = gp
    else:
        if z == z.parent.left:
            z = z.parent
            root = rotate_right(root, z)
        z.parent.color = "B"
        gp.color = "R"
        root = rotate_left(root, gp)
root.color = "B"
return root

def bst_insert(root, z):
    if not root: return z
    if z.key < root.key:
        root.left = bst_insert(root.left, z)
        root.left.parent = root
    elif z.key > root.key:
        root.right = bst_insert(root.right, z)
        root.right.parent = root
    return root

def insert(root, key):
    z = Node(key)
    root = bst_insert(root, z)

```

```

    return fixViolation(root, z)

def inorder(r):
    if not r: return
    inorder(r.left)
    print(f"\{r.key}\{r.color}", end=" ")
    inorder(r.right)

root = None
for k in [10,20,30]:
    root = insert(root, k)
inorder(root)

```

## Why It Matters

- Fewer rotations than AVL
- Guarantees  $O(\log n)$  for all operations
- Used in real systems: Linux kernel, Java TreeMap, C++ map
- Easy insertion logic using coloring + rotation

## A Gentle Proof (Why It Works)

The black-height invariant ensures every path length is between  $h$  and  $2h$ . Balancing via recolor + single rotation ensures logarithmic height. Each insert requires at most 2 rotations,  $O(\log n)$  traversal.

## Try It Yourself

1. Insert [10, 20, 30] → RR rotation
2. Insert [30, 15, 10] → LL rotation
3. Insert [10, 15, 5] → recolor, no rotation
4. Draw colors, confirm invariants

## Test Cases

| Sequence               | Rotations | Root  | Black Height |
|------------------------|-----------|-------|--------------|
| [10,20,30]             | Left      | 20(B) | 2            |
| [7,3,18,10,22,8,11,26] | Mixed     | 7(B)  | 3            |

| Sequence    | Rotations            | Root | Black Height |
|-------------|----------------------|------|--------------|
| [1,2,3,4,5] | Multiple recolor+rot | 2(B) | 3            |

## Complexity

| Operation | Time        | Space            |
|-----------|-------------|------------------|
| Insert    | $O(\log n)$ | $O(1)$ rotations |
| Search    | $O(\log n)$ | $O(h)$           |
| Delete    | $O(\log n)$ | $O(1)$ rotations |

The Red-Black Tree paints order into chaos, each red spark balanced by a calm black shadow.

## 234 Red-Black Tree Delete

Deletion in a Red-Black Tree (RBT) is a delicate operation: remove a node, then restore balance by adjusting colors and rotations to maintain all five RBT invariants. While insertion fixes “too much red,” deletion often fixes “too much black.”

### What Problem Are We Solving?

After deleting a node from an RBT, the black-height property may break (some paths lose a black node). Our goal: restore all red-black invariants while keeping time complexity  $O(\log n)$ .

### Red-Black Properties (Reminder)

1. Root is black.
2. Every node is red or black.
3. All NIL leaves are black.
4. Red nodes cannot have red children.
5. Every path from a node to NIL descendants has the same number of black nodes.

## How Does It Work (Plain Language)?

1. Perform standard BST deletion.
2. Track if the removed node was black (this might cause “double black” issue).
3. Fix violations moving upward until the tree is balanced again.

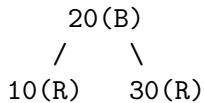
| Case | Condition                                   | Fix                             |
|------|---------------------------------------------|---------------------------------|
| 1    | Node is red                                 | Simply remove (no rebalance)    |
| 2    | Node is black, child red                    | Replace and recolor child black |
| 3    | Node is black, child black (“double black”) | Use sibling cases below         |

### Double Black Fix Cases

| Case | Description                                    | Action                                                         |
|------|------------------------------------------------|----------------------------------------------------------------|
| 1    | Sibling red                                    | Rotate and recolor to make sibling black                       |
| 2    | Sibling black, both children black             | Recolor sibling red, move double black up                      |
| 3    | Sibling black, near child red, far child black | Rotate sibling toward node, swap colors                        |
| 4    | Sibling black, far child red                   | Rotate parent, recolor sibling and parent, set far child black |

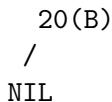
## Example

Delete 30 from:



- 30 is red → remove directly
- No property violated

Delete 10:



10 was red → simple remove → tree valid

Delete 20: Tree becomes empty → fine

## Step-by-Step Example

Insert [10, 20, 30, 15, 25] Delete 20

| Step | Action       | Violation               | Fix                                             |
|------|--------------|-------------------------|-------------------------------------------------|
| 1    | Delete 20    | double black at 25      | sibling 10 black, far child red → Case 4 rotate |
| 2    | After rotate | all properties restored | -                                               |

## Tiny Code (Simplified)

C (Conceptual)

```
// This snippet omits full BST insertion for brevity.

typedef enum { RED, BLACK } Color;
typedef struct Node {
    int key;
    Color color;
    struct Node *left, *right, *parent;
} Node;

// Utility: get sibling
Node* sibling(Node* n) {
    if (!n->parent) return NULL;
    return n == n->parent->left ? n->parent->right : n->parent->left;
}

void fix_delete(Node root, Node* x) {
    while (x != *root && (!x || x->color == BLACK)) {
        Node* s = sibling(x);
        if (x == x->parent->left) {
            if (s->color == RED) {
                s->color = BLACK;
                x->parent->color = RED;
                rotate_left(root, x->parent);
                s = x->parent->right;
            }
            if ((!s->left || s->left->color == BLACK) &&
                (!s->right || s->right->color == BLACK)) {
                s->color = RED;
                x = x->parent;
            }
        }
    }
}
```

```

    } else {
        if (!s->right || s->right->color == BLACK) {
            if (s->left) s->left->color = BLACK;
            s->color = RED;
            rotate_right(root, s);
            s = x->parent->right;
        }
        s->color = x->parent->color;
        x->parent->color = BLACK;
        if (s->right) s->right->color = BLACK;
        rotate_left(root, x->parent);
        x = *root;
    }
} else {
    // mirror logic for right child
    if (s->color == RED) {
        s->color = BLACK;
        x->parent->color = RED;
        rotate_right(root, x->parent);
        s = x->parent->left;
    }
    if ((!s->left || s->left->color == BLACK) &&
        (!s->right || s->right->color == BLACK)) {
        s->color = RED;
        x = x->parent;
    } else {
        if (!s->left || s->left->color == BLACK) {
            if (s->right) s->right->color = BLACK;
            s->color = RED;
            rotate_left(root, s);
            s = x->parent->left;
        }
        s->color = x->parent->color;
        x->parent->color = BLACK;
        if (s->left) s->left->color = BLACK;
        rotate_right(root, x->parent);
        x = *root;
    }
}
if (x) x->color = BLACK;
}

```

Python (Simplified Pseudocode)

```
def fix_delete(root, x):
    while x != root and (not x or x.color == "B"):
        if x == x.parent.left:
            s = x.parent.right
            if s.color == "R":
                s.color, x.parent.color = "B", "R"
                root = rotate_left(root, x.parent)
                s = x.parent.right
            if all(c is None or c.color == "B" for c in [s.left, s.right]):
                s.color = "R"
                x = x.parent
            else:
                if not s.right or s.right.color == "B":
                    if s.left: s.left.color = "B"
                    s.color = "R"
                    root = rotate_right(root, s)
                    s = x.parent.right
                s.color, x.parent.color = x.parent.color, "B"
                if s.right: s.right.color = "B"
                root = rotate_left(root, x.parent)
                x = root
        else:
            # mirror logic
            ...
    if x: x.color = "B"
return root
```

## Why It Matters

- Preserves logarithmic height after deletion
- Used in core data structures (`std::map`, `TreeSet`)
- Demonstrates color logic as a soft balancing scheme
- Handles edge cases gracefully via double black fix

## A Gentle Proof (Why It Works)

When a black node is removed, one path loses a black count. The fix cases re-distribute black heights using rotations and recolors. Each loop iteration moves double-black upward,  $O(\log n)$  steps.

## Try It Yourself

1. Build [10, 20, 30, 15, 25, 5]. Delete 10 (leaf).
2. Delete 30 (red leaf) → no fix.
3. Delete 20 (black node with one child) → recolor fix.
4. Visualize each case: sibling red, sibling black with red child.

## Test Cases

| Insert Sequence        | Delete | Case Triggered | Fix              |
|------------------------|--------|----------------|------------------|
| [10,20,30]             | 20     | Case 4         | Rotate & recolor |
| [7,3,18,10,22,8,11,26] | 18     | Case 2         | Recolor sibling  |
| [10,5,1]               | 5      | Case 1         | Rotate parent    |

## Complexity

| Operation | Time        | Space            |
|-----------|-------------|------------------|
| Delete    | $O(\log n)$ | $O(1)$ rotations |
| Insert    | $O(\log n)$ | $O(1)$           |
| Search    | $O(\log n)$ | $O(h)$           |

A Red-Black delete is like a careful tune-up, one color at a time, harmony restored across every path.

## 235 Splay Tree Access

A Splay Tree is a self-adjusting binary search tree that brings frequently accessed elements closer to the root through splaying, a series of tree rotations. Unlike AVL or Red-Black trees, it doesn't maintain strict balance but guarantees amortized  $O(\log n)$  performance for access, insert, and delete.

## What Problem Are We Solving?

In many workloads, some elements are accessed far more frequently than others. Ordinary BSTs give no advantage for “hot” keys, while balanced trees maintain shape but not recency. A Splay Tree optimizes for temporal locality: recently accessed items move near the root, making repeated access faster.

## How Does It Work (Plain Language)?

Whenever you access a node (via search, insert, or delete), perform a splay operation, repeatedly rotate the node toward the root according to its position and parent relationships.

The three rotation patterns:

| Case    | Structure                                            | Operation                                         |
|---------|------------------------------------------------------|---------------------------------------------------|
| Zig     | Node is child of root                                | Single rotation                                   |
| Zig-Zig | Node and parent are both left or both right children | Double rotation (rotate parent, then grandparent) |
| Zig-Zag | Node and parent are opposite children                | Double rotation (rotate node twice upward)        |

After splaying, the accessed node becomes the root.

### Example

Access sequence: 10, 20, 30

1. Insert 10 (root)
2. Insert 20 → 20 right of 10
3. Access 20 → Zig rotation → 20 becomes root
4. Insert 30 → 30 right of 20
5. Access 30 → Zig rotation → 30 root

Resulting tree (after all accesses):

```
 30
  /
 20
  /
10
```

Frequently used nodes rise to the top automatically.

### Step-by-Step Example

Start with keys [5, 3, 8, 1, 4, 7, 9]. Access key 1.

| Step | Operation   | Case                | Rotation         | New Root |
|------|-------------|---------------------|------------------|----------|
| 1    | Access 1    | Zig-Zig (left-left) | Rotate 3, then 5 | 1        |
| 2    | After splay | -                   | -                | 1        |

Final tree: 1 becomes root, path shortened for next access.

### Tiny Code (Easy Versions)

C

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    int key;
    struct Node *left, *right;
} Node;

Node* new_node(int key) {
    Node* n = malloc(sizeof(Node));
    n->key = key;
    n->left = n->right = NULL;
    return n;
}

Node* rotate_right(Node* x) {
    Node* y = x->left;
    x->left = y->right;
    y->right = x;
    return y;
}

Node* rotate_left(Node* x) {
    Node* y = x->right;
    x->right = y->left;
    y->left = x;
    return y;
}

Node* splay(Node* root, int key) {
```

```

if (!root || root->key == key) return root;

// Key in left subtree
if (key < root->key) {
    if (!root->left) return root;
    if (key < root->left->key) {
        root->left->left = splay(root->left->left, key);
        root = rotate_right(root);
    } else if (key > root->left->key) {
        root->left->right = splay(root->left->right, key);
        if (root->left->right)
            root->left = rotate_left(root->left);
    }
    return root->left ? rotate_right(root) : root;
}
// Key in right subtree
else {
    if (!root->right) return root;
    if (key > root->right->key) {
        root->right->right = splay(root->right->right, key);
        root = rotate_left(root);
    } else if (key < root->right->key) {
        root->right->left = splay(root->right->left, key);
        if (root->right->left)
            root->right = rotate_right(root->right);
    }
    return root->right ? rotate_left(root) : root;
}
}

Node* insert(Node* root, int key) {
    if (!root) return new_node(key);
    root = splay(root, key);
    if (root->key == key) return root;
    Node* n = new_node(key);
    if (key < root->key) {
        n->right = root;
        n->left = root->left;
        root->left = NULL;
    } else {
        n->left = root;
        n->right = root->right;
    }
}

```

```

        root->right = NULL;
    }
    return n;
}

void inorder(Node* r) {
    if (!r) return;
    inorder(r->left);
    printf("%d ", r->key);
    inorder(r->right);
}

int main(void) {
    Node* root = NULL;
    int keys[] = {5, 3, 8, 1, 4, 7, 9};
    for (int i = 0; i < 7; i++) root = insert(root, keys[i]);
    root = splay(root, 1);
    inorder(root);
    printf("\n");
}

```

Python

```

class Node:
    def __init__(self, key):
        self.key = key
        self.left = self.right = None

def rotate_right(x):
    y = x.left
    x.left = y.right
    y.right = x
    return y

def rotate_left(x):
    y = x.right
    x.right = y.left
    y.left = x
    return y

def splay(root, key):
    if not root or root.key == key:

```

```

        return root
    if key < root.key:
        if not root.left:
            return root
        if key < root.left.key:
            root.left.left = splay(root.left.left, key)
            root = rotate_right(root)
        elif key > root.left.key:
            root.left.right = splay(root.left.right, key)
            if root.left.right:
                root.left = rotate_left(root.left)
        return rotate_right(root) if root.left else root
    else:
        if not root.right:
            return root
        if key > root.right.key:
            root.right.right = splay(root.right.right, key)
            root = rotate_left(root)
        elif key < root.right.key:
            root.right.left = splay(root.right.left, key)
            if root.right.left:
                root.right = rotate_right(root.right)
        return rotate_left(root) if root.right else root

def insert(root, key):
    if not root:
        return Node(key)
    root = splay(root, key)
    if root.key == key:
        return root
    n = Node(key)
    if key < root.key:
        n.right = root
        n.left = root.left
        root.left = None
    else:
        n.left = root
        n.right = root.right
        root.right = None
    return n

def inorder(r):

```

```

if not r: return
inorder(r.left)
print(r.key, end=" ")
inorder(r.right)

root = None
for k in [5,3,8,1,4,7,9]:
    root = insert(root, k)
root = splay(root, 1)
inorder(root)

```

## Why It Matters

- Amortized  $O(\log n)$  performance
- Adapts dynamically to access patterns
- Ideal for caches, text editors, network routing tables
- Simple logic: no height or color tracking needed

## A Gentle Proof (Why It Works)

Each splay operation may take  $O(h)$ , but the amortized cost across multiple operations is  $O(\log n)$ . Accessing frequently used elements keeps them near the root, improving future operations.

## Try It Yourself

1. Build  $[5, 3, 8, 1, 4, 7, 9]$
2. Access 1 → Observe Zig-Zig rotation
3. Access 8 → Observe Zig-Zag
4. Insert 10 → check rebalancing
5. Access 7 repeatedly → moves to root

## Test Cases

| Sequence      | Access | Case    | Root | After |
|---------------|--------|---------|------|-------|
| $[5,3,8,1,4]$ | 1      | Zig-Zig | 1    |       |
| $[10,20,30]$  | 30     | Zig     | 30   |       |
| $[5,3,8,1,4]$ | 4      | Zig-Zag | 4    |       |

## Complexity

| Operation | Time                  | Space            |
|-----------|-----------------------|------------------|
| Access    | Amortized $O(\log n)$ | $O(h)$ recursion |
| Insert    | Amortized $O(\log n)$ | $O(h)$           |
| Delete    | Amortized $O(\log n)$ | $O(h)$           |

The Splay Tree is like a memory, the more you touch something, the closer it stays to you.

## 236 Treap Insert

A Treap (Tree + Heap) is a brilliant hybrid: it's a binary search tree by keys and a heap by priorities. Every node carries a key and a random priority. By combining ordering on keys with random heap priorities, the treap stays *balanced on average*, no strict rotations like AVL or color rules like Red-Black trees needed.

### What Problem Are We Solving?

We want a simple balanced search tree with expected  $O(\log n)$  performance, but without maintaining height or color properties explicitly. Treaps solve this by assigning random priorities, keeping the structure balanced *in expectation*.

### How It Works (Plain Language)

Each node (`key, priority`) must satisfy:

- BST property: `key(left) < key < key(right)`
- Heap property: `priority(parent) < priority(children)` (min-heap or max-heap convention)

Insert like a normal BST by key, then fix heap property by rotations if the priority is violated.

| Step | Rule                                                   |
|------|--------------------------------------------------------|
| 1    | Insert node by key (like BST)                          |
| 2    | Assign a random priority                               |
| 3    | While parent priority > node priority → rotate node up |

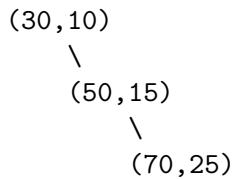
This randomization ensures expected logarithmic height.

## Example

Insert sequence (key, priority):

| Step | Key      | Priority                        | Structure                  |
|------|----------|---------------------------------|----------------------------|
| 1    | (50, 15) | root                            | 50                         |
| 2    | (30, 10) | smaller priority → rotate right | 30(root) → 50(right child) |
| 3    | (70, 25) | stays right of 50               | balanced                   |

Result (BST by key, min-heap by priority):



## Step-by-Step Example

Insert keys: [40, 20, 60, 10, 30, 50, 70] Priorities: [80, 90, 70, 100, 85, 60, 75]

| Key | Priority | Rotation               | Result     |
|-----|----------|------------------------|------------|
| 40  | 80       | root                   | 40         |
| 20  | 90       | none                   | left child |
| 60  | 70       | rotate left (heap fix) | 60 root    |
| 10  | 100      | none                   | leaf       |
| 30  | 85       | none                   | under 20   |
| 50  | 60       | rotate left            | 50 root    |
| 70  | 75       | none                   | leaf       |

## Tiny Code (Easy Versions)

C

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

typedef struct Node {
```

```

int key, priority;
struct Node *left, *right;
} Node;

Node* new_node(int key) {
    Node* n = malloc(sizeof(Node));
    n->key = key;
    n->priority = rand() % 100; // random priority
    n->left = n->right = NULL;
    return n;
}

Node* rotate_right(Node* y) {
    Node* x = y->left;
    y->left = x->right;
    x->right = y;
    return x;
}

Node* rotate_left(Node* x) {
    Node* y = x->right;
    x->right = y->left;
    y->left = x;
    return y;
}

Node* insert(Node* root, int key) {
    if (!root) return new_node(key);

    if (key < root->key) {
        root->left = insert(root->left, key);
        if (root->left->priority < root->priority)
            root = rotate_right(root);
    } else if (key > root->key) {
        root->right = insert(root->right, key);
        if (root->right->priority < root->priority)
            root = rotate_left(root);
    }
    return root;
}

void inorder(Node* root) {

```

```

if (!root) return;
inorder(root->left);
printf("(%d,%d) ", root->key, root->priority);
inorder(root->right);
}

int main(void) {
    srand(time(NULL));
    Node* root = NULL;
    int keys[] = {40, 20, 60, 10, 30, 50, 70};
    for (int i = 0; i < 7; i++)
        root = insert(root, keys[i]);
    inorder(root);
    printf("\n");
}

```

Python

```

import random

class Node:
    def __init__(self, key):
        self.key = key
        self.priority = random.randint(1, 100)
        self.left = self.right = None

    def rotate_right(y):
        x = y.left
        y.left = x.right
        x.right = y
        return x

    def rotate_left(x):
        y = x.right
        x.right = y.left
        y.left = x
        return y

    def insert(self, key):
        if not self:
            return Node(key)
        if key < self.key:

```

```

root.left = insert(root.left, key)
    if root.left.priority < root.priority:
        root = rotate_right(root)
elif key > root.key:
    root.right = insert(root.right, key)
    if root.right.priority < root.priority:
        root = rotate_left(root)
return root

def inorder(root):
    if not root: return
    inorder(root.left)
    print(f"({root.key},{root.priority})", end=" ")
    inorder(root.right)

root = None
for k in [40, 20, 60, 10, 30, 50, 70]:
    root = insert(root, k)
inorder(root)

```

## Why It Matters

- Simple implementation with expected balance
- Probabilistic alternative to AVL / Red-Black
- Great for randomized data structures and Cartesian tree applications
- No explicit height or color tracking

## A Gentle Proof (Why It Works)

Random priorities imply random structure; the probability of a node being high in the tree drops exponentially with depth. Hence, expected height is  $O(\log n)$  with high probability.

## Try It Yourself

1. Insert keys [10, 20, 30, 40] with random priorities.
2. Observe random structure (not sorted by insertion).
3. Change priority generator → test skewness.
4. Visualize both BST and heap invariants.

## Test Cases

| Keys        | Priorities | Resulting Root | Height        |
|-------------|------------|----------------|---------------|
| [10,20,30]  | [5,3,4]    | 20             | 2             |
| [5,2,8,1,3] | random     | varies         | $\sim \log n$ |
| [1..100]    | random     | balanced       | $O(\log n)$   |

## Complexity

| Operation | Time                 | Space            |
|-----------|----------------------|------------------|
| Insert    | $O(\log n)$ expected | $O(h)$ recursion |
| Search    | $O(\log n)$ expected | $O(h)$           |
| Delete    | $O(\log n)$ expected | $O(h)$           |

A Treap dances to the rhythm of chance, balancing order with randomness to stay nimble and fair.

## 237 Treap Delete

Deleting from a Treap blends the best of both worlds: BST search for locating the node and heap-based rotations to remove it while keeping balance. Instead of rebalancing explicitly, we rotate the target node downward until it becomes a leaf, then remove it.

### What Problem Are We Solving?

We want to delete a key while preserving both:

- BST property: keys in order
- Heap property: priorities follow min-/max-heap rule

Treaps handle this by rotating down the target node until it has at most one child, then removing it directly.

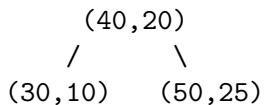
## How It Works (Plain Language)

1. Search for the node by key (BST search).
2. Once found, rotate it downward until it has 1 child:
  - Rotate left if right child's priority < left child's priority
  - Rotate right otherwise
3. Remove it once it's a leaf or single-child node.

Because priorities are random, the tree remains balanced *in expectation*.

## Example

Treap (key, priority):



Delete 40:

- Compare children:  $(30, 10) < (50, 25) \rightarrow$  rotate right on 40
- New root  $(30, 10)$
- 40 now right child  $\rightarrow$  continue rotation if needed  $\rightarrow$  eventually becomes leaf  $\rightarrow$  delete

New structure maintains BST + heap properties.

## Step-by-Step Example

Start with nodes:

| Key | Priority |
|-----|----------|
| 40  | 50       |
| 20  | 60       |
| 60  | 70       |
| 10  | 90       |
| 30  | 80       |
| 50  | 65       |
| 70  | 75       |

Delete 40:

| Step | Node | Action                      | Rotation          | Root               |
|------|------|-----------------------------|-------------------|--------------------|
| 1    | 40   | Compare children priorities | Left child higher | Rotate right       |
| 2    | 30   | Now parent of 40            | 40 demoted        | Continue if needed |
| 3    | 40   | Becomes leaf                | Remove            | 30                 |

Final root: (30,80) All properties hold.

### Tiny Code (Easy Versions)

C

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

typedef struct Node {
    int key, priority;
    struct Node *left, *right;
} Node;

Node* new_node(int key) {
    Node* n = malloc(sizeof(Node));
    n->key = key;
    n->priority = rand() % 100;
    n->left = n->right = NULL;
    return n;
}

Node* rotate_right(Node* y) {
    Node* x = y->left;
    y->left = x->right;
    x->right = y;
    return x;
}

Node* rotate_left(Node* x) {
    Node* y = x->right;
    x->right = y->left;
    y->left = x;
```

```

        return y;
    }

Node* insert(Node* root, int key) {
    if (!root) return new_node(key);
    if (key < root->key) {
        root->left = insert(root->left, key);
        if (root->left->priority < root->priority)
            root = rotate_right(root);
    } else if (key > root->key) {
        root->right = insert(root->right, key);
        if (root->right->priority < root->priority)
            root = rotate_left(root);
    }
    return root;
}

Node* delete(Node* root, int key) {
    if (!root) return NULL;

    if (key < root->key)
        root->left = delete(root->left, key);
    else if (key > root->key)
        root->right = delete(root->right, key);
    else {
        // Found node to delete
        if (!root->left && !root->right) {
            free(root);
            return NULL;
        } else if (!root->left)
            root = rotate_left(root);
        else if (!root->right)
            root = rotate_right(root);
        else if (root->left->priority < root->right->priority)
            root = rotate_right(root);
        else
            root = rotate_left(root);

        root = delete(root, key);
    }
    return root;
}

```

```

void inorder(Node* root) {
    if (!root) return;
    inorder(root->left);
    printf("(%d,%d) ", root->key, root->priority);
    inorder(root->right);
}

int main(void) {
    srand(time(NULL));
    Node* root = NULL;
    int keys[] = {40, 20, 60, 10, 30, 50, 70};
    for (int i = 0; i < 7; i++)
        root = insert(root, keys[i]);

    printf("Before deletion:\n");
    inorder(root);
    printf("\n");

    root = delete(root, 40);

    printf("After deleting 40:\n");
    inorder(root);
    printf("\n");
}

```

Python

```

import random

class Node:
    def __init__(self, key):
        self.key = key
        self.priority = random.randint(1, 100)
        self.left = self.right = None

    def rotate_right(y):
        x = y.left
        y.left = x.right
        x.right = y
        return x

    def rotate_left(x):

```

```

y = x.right
x.right = y.left
y.left = x
return y

def insert(root, key):
    if not root:
        return Node(key)
    if key < root.key:
        root.left = insert(root.left, key)
        if root.left.priority < root.priority:
            root = rotate_right(root)
    elif key > root.key:
        root.right = insert(root.right, key)
        if root.right.priority < root.priority:
            root = rotate_left(root)
    return root

def delete(root, key):
    if not root:
        return None
    if key < root.key:
        root.left = delete(root.left, key)
    elif key > root.key:
        root.right = delete(root.right, key)
    else:
        if not root.left and not root.right:
            return None
        elif not root.left:
            root = rotate_left(root)
        elif not root.right:
            root = rotate_right(root)
        elif root.left.priority < root.right.priority:
            root = rotate_right(root)
        else:
            root = rotate_left(root)
        root = delete(root, key)
    return root

def inorder(root):
    if not root: return
    inorder(root.left)

```

```

print(f"({root.key},{root.priority})", end=" ")
inorder(root.right)

root = None
for k in [40,20,60,10,30,50,70]:
    root = insert(root, k)
print("Before:", end=" ")
inorder(root)
print()
root = delete(root, 40)
print("After:", end=" ")
inorder(root)

```

## Why It Matters

- No explicit rebalancing, rotations emerge from heap priorities
- Expected  $O(\log n)$  deletion
- Natural, simple structure for randomized search trees
- Used in randomized algorithms, dynamic sets, and order-statistics

## A Gentle Proof (Why It Works)

At each step, the node rotates toward a child with smaller priority, preserving heap property. Expected height remains  $O(\log n)$  due to independent random priorities.

## Try It Yourself

1. Build treap with [10, 20, 30, 40, 50].
2. Delete 30 → observe rotations to leaf.
3. Delete 10 (root) → rotation chooses smaller-priority child.
4. Repeat deletions, verify BST + heap invariants hold.

## Test Cases

| Keys                   | Delete | Rotations   | Balanced | Root After |
|------------------------|--------|-------------|----------|------------|
| [40,20,60,10,30,50,70] | 40     | Right, Left |          | 30         |
| [10,20,30]             | 20     | Left        |          | 10         |
| [50,30,70,20,40,60,80] | 30     | Right       |          | 50         |

## Complexity

| Operation | Time                 | Space            |
|-----------|----------------------|------------------|
| Delete    | $O(\log n)$ expected | $O(h)$ recursion |
| Insert    | $O(\log n)$ expected | $O(h)$           |
| Search    | $O(\log n)$ expected | $O(h)$           |

The Treap deletes gracefully, rotating away until the target fades like a falling leaf, leaving balance behind.

## 238 Weight Balanced Tree

A Weight Balanced Tree (WBT) maintains balance not by height or color, but by subtree sizes (or “weights”). Each node keeps track of how many elements are in its left and right subtrees, and balance is enforced by requiring their weights to stay within a constant ratio.

### What Problem Are We Solving?

In ordinary BSTs, imbalance can grow as keys are inserted in sorted order, degrading performance to  $O(n)$ . Weight Balanced Trees fix this by keeping subtree sizes within a safe range. They are particularly useful when you need order-statistics (like “find the  $k$ -th element”) or split/merge operations that depend on sizes rather than heights.

### How It Works (Plain Language)

Each node maintains a weight = total number of nodes in its subtree. When inserting or deleting, you update weights and check the balance condition:

If

```
weight(left)      * weight(node)
weight(right)     * weight(node)
```

for some balance constant (e.g., 0.7), then the tree is considered balanced.

If not, perform rotations (like AVL) to restore balance.

| Operation | Steps                                                                |
|-----------|----------------------------------------------------------------------|
| Insert    | Insert like BST → Update weights → If ratio violated → Rotate        |
| Delete    | Remove node → Update weights → If ratio violated → Rebuild or rotate |
| Search    | BST search (weights guide decisions for rank queries)                |

Because weights reflect actual sizes, the tree ensures  $O(\log n)$  expected height.

### Example

Let's use  $\epsilon = 0.7$

Insert [10, 20, 30, 40, 50]

| Step | Insert | Weight(left) | Weight(right) | Balanced?   | Fix         |
|------|--------|--------------|---------------|-------------|-------------|
| 1    | 10     | 0            | 0             | -           | -           |
| 2    | 20     | 1            | 0             | -           | -           |
| 3    | 30     | 2            | 0             | Rotate left | Rotate left |
| 4    | 40     | 3            | 0             | Rotate left | Rotate left |
| 5    | 50     | 4            | 0             | Rotate left | Rotate left |

Tree remains balanced by rotations once left-right ratio exceeds  $\epsilon$ .

### Step-by-Step Example

Insert [1, 2, 3, 4, 5] with  $\epsilon = 0.7$

| Step | Action   | Weights | Balanced?   | Rotation    |
|------|----------|---------|-------------|-------------|
| 1    | Insert 1 | (0,0)   | -           | -           |
| 2    | Insert 2 | (1,0)   | -           | -           |
| 3    | Insert 3 | (2,0)   | Left Rotate | Left Rotate |
| 4    | Insert 4 | (3,0)   | Left Rotate | Left Rotate |
| 5    | Insert 5 | (4,0)   | Left Rotate | Left Rotate |

Tree remains height-balanced based on weights.

## Tiny Code (Easy Versions)

C (Simplified)

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    int key;
    int size; // weight = size of subtree
    struct Node *left, *right;
} Node;

int size(Node* n) { return n ? n->size : 0; }

Node* new_node(int key) {
    Node* n = malloc(sizeof(Node));
    n->key = key;
    n->size = 1;
    n->left = n->right = NULL;
    return n;
}

Node* rotate_left(Node* x) {
    Node* y = x->right;
    x->right = y->left;
    y->left = x;
    x->size = 1 + size(x->left) + size(x->right);
    y->size = 1 + size(y->left) + size(y->right);
    return y;
}

Node* rotate_right(Node* y) {
    Node* x = y->left;
    y->left = x->right;
    x->right = y;
    y->size = 1 + size(y->left) + size(y->right);
    x->size = 1 + size(x->left) + size(x->right);
    return x;
}

double alpha = 0.7;
```

```

int balanced(Node* n) {
    if (!n) return 1;
    int l = size(n->left), r = size(n->right);
    return l <= alpha * n->size && r <= alpha * n->size;
}

Node* insert(Node* root, int key) {
    if (!root) return new_node(key);
    if (key < root->key) root->left = insert(root->left, key);
    else if (key > root->key) root->right = insert(root->right, key);
    root->size = 1 + size(root->left) + size(root->right);
    if (!balanced(root)) {
        if (size(root->left) > size(root->right))
            root = rotate_right(root);
        else
            root = rotate_left(root);
    }
    return root;
}

void inorder(Node* n) {
    if (!n) return;
    inorder(n->left);
    printf("%d(%d) ", n->key, n->size);
    inorder(n->right);
}

int main(void) {
    Node* root = NULL;
    int keys[] = {10, 20, 30, 40, 50};
    for (int i = 0; i < 5; i++)
        root = insert(root, keys[i]);
    inorder(root);
    printf("\n");
}

```

Python (Simplified)

```

class Node:
    def __init__(self, key):
        self.key = key
        self.size = 1

```

```

        self.left = self.right = None

def size(n): return n.size if n else 0

def update(n):
    if n:
        n.size = 1 + size(n.left) + size(n.right)

def rotate_left(x):
    y = x.right
    x.right = y.left
    y.left = x
    update(x)
    update(y)
    return y

def rotate_right(y):
    x = y.left
    y.left = x.right
    x.right = y
    update(y)
    update(x)
    return x

alpha = 0.7

def balanced(n):
    return not n or (size(n.left) <= alpha * n.size and size(n.right) <= alpha * n.size)

def insert(root, key):
    if not root: return Node(key)
    if key < root.key: root.left = insert(root.left, key)
    elif key > root.key: root.right = insert(root.right, key)
    update(root)
    if not balanced(root):
        if size(root.left) > size(root.right):
            root = rotate_right(root)
        else:
            root = rotate_left(root)
    return root

def inorder(n):

```

```

if not n: return
inorder(n.left)
print(f"{n.key}({n.size})", end=" ")
inorder(n.right)

root = None
for k in [10,20,30,40,50]:
    root = insert(root, k)
inorder(root)

```

## Why It Matters

- Balances based on true subtree size, not height
- Excellent for order-statistics ( $k^{\text{th}}$ ,  $\text{rank}$ )
- Supports split, merge, and range queries naturally
- Deterministic balancing without randomness

## A Gentle Proof (Why It Works)

Maintaining weight ratios guarantees logarithmic height. If  $\alpha < 1$ , each subtree has  $\alpha \times n$  nodes, so height  $\log(\alpha \times n) / (\alpha) \rightarrow O(\log n)$ .

## Try It Yourself

1. Build  $[10, 20, 30, 40, 50]$  with  $\alpha = 0.7$ .
2. Print sizes at each node.
3. Delete 20  $\rightarrow$  rebalance by rotation.
4. Try  $\alpha = 0.6$  and  $\alpha = 0.8 \rightarrow$  compare shapes.

## Test Cases

| Sequence                 | Balanced? | Height      |
|--------------------------|-----------|-------------|
| $[10,20,30,40,50]$       | 0.7       | $\log n$    |
| $[1..100]$               | 0.7       | $O(\log n)$ |
| $[\text{sorted } 1..10]$ | 0.6       | $\sim 4$    |

## Complexity

| Operation | Time        | Space  |
|-----------|-------------|--------|
| Insert    | $O(\log n)$ | $O(h)$ |
| Delete    | $O(\log n)$ | $O(h)$ |
| Search    | $O(\log n)$ | $O(h)$ |

A Weight Balanced Tree is like a tightrope walker, always adjusting its stance to keep perfect equilibrium, no matter how the sequence unfolds.

## 239 Scapegoat Tree Rebuild

A Scapegoat Tree is a binary search tree that maintains balance by occasionally rebuilding entire subtrees.

Instead of performing rotations on every insertion, it monitors the depth of nodes.

When a node's depth exceeds the allowed limit, the algorithm locates a *scapegoat* ancestor whose subtree is too unbalanced, flattens that subtree into a sorted list, and rebuilds it into a perfectly balanced tree.

With a balance parameter  $\alpha$  in  $(0.5, 1)$ , a Scapegoat Tree guarantees:

- $O(\log n)$  amortized time for insertions and deletions

- $O(\log n)$  worst-case height

### What Problem Are We Solving?

Standard BSTs can become skewed and degrade to  $O(n)$ . Rotational trees like AVL and Red Black maintain strict local invariants but add per update overhead. Scapegoat trees choose a middle path

- Do nothing most of the time
- Occasionally rebuild an entire subtree when a global bound is exceeded

Goal Keep height near  $\log_{1/\alpha} n$  while keeping code simple and rotations out of the hot path.

### How It Works (Plain Language)

Parameters and invariants

- Choose  $\alpha \in (0.5, 1)$  such as  $\alpha = \frac{2}{3}$
- Maintain  $n$  = current size and  $n_{\max}$  = maximum size since the last global rebuild

- Height bound: if an insertion lands deeper than  $\lfloor \log_{1/\alpha} n \rfloor$ , the tree is too tall

Insert algorithm

1. Insert like a normal BST and track the path length `depth`.
2. If `depth` is within the allowed bound, stop.
3. Otherwise, walk up to find the scapegoat node `s` such that

$$\max(\text{size}(s.\text{left}), \text{size}(s.\text{right})) > \alpha \cdot \text{size}(s)$$

4. Rebuild the subtree rooted at `s` into a perfectly balanced BST in time linear in that subtree's size.

Delete algorithm

- Perform a normal BST delete.
- Decrement `n`.
- If  $n < \alpha \cdot n_{\max}$ , rebuild the entire tree and set  $n_{\max} = n$ .

Why rebuilding works

- Rebuilding performs an inorder traversal to collect nodes into a sorted array, then constructs a balanced BST from that array.
- Rebuilds are infrequent, and their costs are amortized over many cheap insertions and deletions, ensuring logarithmic time on average.

### Example Walkthrough

Let  $\alpha = \frac{2}{3}$ .

Insert the keys [10, 20, 30, 40, 50, 60] in ascending order.

| Step | Action    | Depth vs bound        | Scapegoat found   | Rebuild              |
|------|-----------|-----------------------|-------------------|----------------------|
| 1    | insert 10 | depth 0 within        | none              | no                   |
| 2    | insert 20 | depth 1 within        | none              | no                   |
| 3    | insert 30 | depth 2 within        | none              | no                   |
| 4    | insert 40 | depth 3 exceeds bound | ancestor violates | rebuild that subtree |
| 5    | insert 50 | likely within         | none              | no                   |
| 6    | insert 60 | if depth exceeds      | find ancestor     | rebuild subtree      |

The occasional rebuild produces a nearly perfect BST despite sorted input.

## Tiny Code (Easy Versions)

Python (concise reference implementation)

```
from math import log, floor

class Node:
    __slots__ = ("key", "left", "right", "size")
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.size = 1

def size(x): return x.size if x else 0
def update(x):
    if x: x.size = 1 + size(x.left) + size(x.right)

def flatten_inorder(x, arr):
    if not x: return
    flatten_inorder(x.left, arr)
    arr.append(x)
    flatten_inorder(x.right, arr)

def build_balanced(nodes, lo, hi):
    if lo >= hi: return None
    mid = (lo + hi) // 2
    root = nodes[mid]
    root.left = build_balanced(nodes, lo, mid)
    root.right = build_balanced(nodes, mid + 1, hi)
    update(root)
    return root

class ScapegoatTree:
    def __init__(self, alpha=2/3):
        self.alpha = alpha
        self.root = None
        self.n = 0
        self.n_max = 0

    def _log_alpha(self, n):
        # height bound floor(log_{1/alpha} n)
        if n <= 1: return 0
```

```

        return floor(log(n, 1 / self.alpha))

def _find_and_rebuild(self, path):
    # path is list of nodes from root to inserted node
    for i in range(len(path) - 1, -1, -1):
        x = path[i]
        l, r = size(x.left), size(x.right)
        if max(l, r) > self.alpha * size(x):
            # rebuild subtree rooted at x and reattach to parent
            nodes = []
            flatten_inorder(x, nodes)
            new_sub = build_balanced(nodes, 0, len(nodes))
            if i == 0:
                self.root = new_sub
            else:
                p = path[i - 1]
                if p.left is x: p.left = new_sub
                else: p.right = new_sub
                # fix sizes upward from parent
                for j in range(i - 1, -1, -1):
                    update(path[j])
    return

def insert(self, key):
    self.n += 1
    self.n_max = max(self.n_max, self.n)
    if not self.root:
        self.root = Node(key)
    return

    # standard BST insert with path tracking
path = []
cur = self.root
while cur:
    path.append(cur)
    if key < cur.key:
        if cur.left: cur = cur.left
        else:
            cur.left = Node(key)
            path.append(cur.left)
            break
    elif key > cur.key:

```

```

        if cur.right: cur = cur.right
    else:
        cur.right = Node(key)
        path.append(cur.right)
        break
    else:
        # duplicate ignore
        self.n -= 1
        return
# update sizes
for node in reversed(path[:-1]):
    update(node)

# depth check and possible rebuild
if len(path) - 1 > self._log_alpha(self.n):
    self._find_and_rebuild(path)

def _join_left_max(self, t):
    # remove and return max node from subtree t, plus the new subtree
    if not t.right:
        return t, t.left
    m, t.right = self._join_left_max(t.right)
    update(t)
    return m, t

def delete(self, key):
    # standard BST delete
    def _del(x, key):
        if not x: return None, False
        if key < x.key:
            x.left, removed = _del(x.left, key)
        elif key > x.key:
            x.right, removed = _del(x.right, key)
        else:
            removed = True
            if not x.left: return x.right, True
            if not x.right: return x.left, True
            # replace with predecessor
            m, x.left = self._join_left_max(x.left)
            m.left, m.right = x.left, x.right
            x = m
        update(x)
        return x, removed
    m, t = self._join_left_max(t)
    if not t:
        return None, False
    if t.key == key:
        if not t.left: return t.right, True
        if not t.right: return t.left, True
        # replace with predecessor
        m, t.left = self._join_left_max(t.left)
        m.left, m.right = t.left, t.right
        t = m
    update(t)
    return t, removed

```

```

        return x, removed

    self.root, removed = _del(self.root, key)
    if not removed: return
    self.n -= 1
    if self.n < self.alpha * self.n_max:
        # global rebuild
        nodes = []
        flatten_inorder(self.root, nodes)
        self.root = build_balanced(nodes, 0, len(nodes))
        self.n_max = self.n

```

C (outline of the rebuild idea)

```

// Sketch only: show rebuild path
// 1) Do BST insert while recording path stack
// 2) If depth > bound, walk path upward to find scapegoat
// 3) Inorder-copy subtree nodes to an array
// 4) Recursively rebuild balanced subtree from that array
// 5) Reattach rebuilt subtree to parent and fix sizes up the path

```

## Why It Matters

- Simple balancing approach with rare but powerful subtree rebuilds
- Well suited for workloads where insertions arrive in bursts and frequent rotations are undesirable
- Guarantees height  $O(\log n)$  with a tunable constant controlled by  $\alpha$
- The inorder-based rebuild process makes ordered operations efficient and easy to implement

## A Gentle Proof (Why It Works)

Let  $\alpha \in (0.5, 1)$ .

- Height bound: if height exceeds  $\log_{1/\alpha} n$ , a scapegoat exists since some ancestor must violate  $\max\{|L|, |R|\} \leq \alpha |T|$ .
- Amortization: each node participates in  $O(1)$  rebuilds with  $\Theta(\text{size of its subtree})$  work, so the total cost over  $m$  operations is  $O(m \log n)$ .

- Deletion rule: rebuilding when  $n < \alpha n_{\max}$  prevents slack from accumulating.

### Try It Yourself

1. Insert ascending keys to trigger rebuilds with  $\alpha = \frac{2}{3}$
2. Delete many keys so that  $n < \alpha n_{\max}$  and observe the global rebuild trigger
3. Experiment with different  $\alpha$  values
4. Add an order statistic query using stored subtree sizes

### Test Cases

| Operation        | Input                          | Expected                                             |
|------------------|--------------------------------|------------------------------------------------------|
| Insert ascending | [1..1000]                      | 0.66 Height stays $O(\log n)$ with periodic rebuilds |
| Mixed ops        | random inserts and deletes     | 0.7 Amortized $O(\log n)$ per op                     |
| Shrink check     | insert 1..200, delete 101..200 | 0.7 Global rebuild when size drops below $n_{\max}$  |
| Duplicate insert | insert 42 twice                | 0.66 size unchanged after second insert              |

### Edge Cases

- Rebuilding an empty or singleton subtree is a no-op
- Duplicate keys are ignored or handled according to policy
- Choose  $\alpha > 0.5$  to guarantee the existence of a scapegoat

### Complexity

| Operation       | Time                                | Space                            |
|-----------------|-------------------------------------|----------------------------------|
| Search          | $O(h)$ worst, $O(\log n)$ amortized | $O(1)$ extra                     |
| Insert          | Amortized $O(\log n)$               | $O(1)$ extra plus rebuild buffer |
| Delete          | Amortized $O(\log n)$               | $O(1)$ extra plus rebuild buffer |
| Rebuild subtree | $O(k)$ for $k$ nodes in subtree     | $O(k)$ temporary array           |

Scapegoat trees wait calmly until imbalance is undeniable, then rebuild decisively. The result is a BST that stays lean with minimal fuss.

## 240 AA Tree

An AA Tree is a simplified version of a red-black tree that maintains balance using a single level value per node (instead of color). It enforces balance by a small set of easy-to-code rules using two operations, `skew` and `split`. The simplicity of AA trees makes them a popular teaching and practical implementation choice when you want red-black performance without complex cases.

### What Problem Are We Solving?

We want a balanced binary search tree with

- $O(\log n)$  time for insert/search/delete
- Simpler code than red-black or AVL
- Fewer rotation cases

AA trees achieve this by using levels (like black-heights in RBTs) to enforce structure, ensuring a balanced shape with one rotation per fix.

### How It Works (Plain Language)

Each node has

- `key`: stored value
- `level`: like black height (root = 1)

AA-tree invariants:

1. Left child level < node level
2. Right child level == node level
3. Right-right grandchild level < node level
4. Every leaf has level 1

To restore balance after insert/delete, apply skew and split operations:

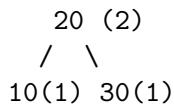
| Operation | Rule                               | Action               |
|-----------|------------------------------------|----------------------|
| Skew      | If left.level == node.level        | Rotate right         |
| Split     | If right.right.level == node.level | Rotate left, level++ |

## Example

Insert keys [10, 20, 30]

| Step | Tree         | Fix                                               |
|------|--------------|---------------------------------------------------|
| 1    | 10 (lvl 1)   | -                                                 |
| 2    | 10 → 20      | Skew none                                         |
| 3    | 10 → 20 → 30 | Right-right violation → Split → Rotate left on 10 |

Result:



## Step-by-Step Example

Insert [30, 20, 10, 25]

| Step | Insertion | Violation                     | Fix                                    |
|------|-----------|-------------------------------|----------------------------------------|
| 1    | 30        | none                          | -                                      |
| 2    | 20        | left child level = node level | Skew → Rotate right                    |
| 3    | 10        | deeper left                   | Skew & Split                           |
| 4    | 25        | insert right of 20            | 20 root<br>Split if needed<br>balanced |

## Tiny Code (Easy Versions)

C

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    int key;
    int level;
    struct Node *left, *right;
} Node;

Node* new_node(int key) {
```

```

Node* n = malloc(sizeof(Node));
n->key = key;
n->level = 1;
n->left = n->right = NULL;
return n;
}

Node* rotate_left(Node* x) {
    Node* y = x->right;
    x->right = y->left;
    y->left = x;
    return y;
}

Node* rotate_right(Node* y) {
    Node* x = y->left;
    y->left = x->right;
    x->right = y;
    return x;
}

Node* skew(Node* x) {
    if (x && x->left && x->left->level == x->level)
        x = rotate_right(x);
    return x;
}

Node* split(Node* x) {
    if (x && x->right && x->right->right && x->right->right->level == x->level) {
        x = rotate_left(x);
        x->level++;
    }
    return x;
}

Node* insert(Node* root, int key) {
    if (!root) return new_node(key);
    if (key < root->key)
        root->left = insert(root->left, key);
    else if (key > root->key)
        root->right = insert(root->right, key);
    root = skew(root);
}

```

```

    root = split(root);
    return root;
}

void inorder(Node* r) {
    if (!r) return;
    inorder(r->left);
    printf("%d, %d) ", r->key, r->level);
    inorder(r->right);
}

int main(void) {
    Node* root = NULL;
    int keys[] = {10, 20, 30, 15, 25};
    for (int i = 0; i < 5; i++)
        root = insert(root, keys[i]);
    inorder(root);
    printf("\n");
}

```

Python

```

class Node:
    def __init__(self, key):
        self.key = key
        self.level = 1
        self.left = None
        self.right = None

    def rotate_left(x):
        y = x.right
        x.right = y.left
        y.left = x
        return y

    def rotate_right(y):
        x = y.left
        y.left = x.right
        x.right = y
        return x

    def skew(x):

```

```

if x and x.left and x.left.level == x.level:
    x = rotate_right(x)
return x

def split(x):
    if x and x.right and x.right.right and x.right.right.level == x.level:
        x = rotate_left(x)
        x.level += 1
    return x

def insert(root, key):
    if not root:
        return Node(key)
    if key < root.key:
        root.left = insert(root.left, key)
    elif key > root.key:
        root.right = insert(root.right, key)
    root = skew(root)
    root = split(root)
    return root

def inorder(r):
    if not r: return
    inorder(r.left)
    print(f"{{r.key}},{{r.level}}", end=" ")
    inorder(r.right)

root = None
for k in [10, 20, 30, 15, 25]:
    root = insert(root, k)
inorder(root)

```

## Why It Matters

- Simpler than red-black
- Only two balancing functions (skew, split)
- Maintains  $O(\log n)$  performance
- Great teaching and reference structure for self-balancing BSTs

## A Gentle Proof (Why It Works)

AA-trees are a special case of red-black trees where

- Right-leaning red links represent level
- Skew fixes left-leaning reds
- Split fixes two consecutive rights

These rules ensure black-height invariants and logarithmic height.

## Try It Yourself

1. Insert ascending keys → observe periodic right rotations
2. Draw tree after each step → check levels
3. Compare structure to equivalent red-black tree
4. Modify split condition → see imbalance appear

## Test Cases

| Input Keys    | Expected Root | Height        | Balanced? |
|---------------|---------------|---------------|-----------|
| [10,20,30]    | 20            | 2             |           |
| [1..10]       | 4–5           | $\sim \log n$ |           |
| [30,20,10,25] | 20            | 3             |           |

## Complexity

| Operation | Time        | Space  |
|-----------|-------------|--------|
| Insert    | $O(\log n)$ | $O(h)$ |
| Search    | $O(\log n)$ | $O(1)$ |
| Delete    | $O(\log n)$ | $O(h)$ |

AA Trees blend red-black efficiency with AVL simplicity, balanced, elegant, and easy to reason about.

## Section 25. Segment Trees and Fenwick Trees

### 241 Build Segment Tree

A segment tree is a binary tree over an array that stores range information for fast queries and updates. Building it means constructing the tree from the base array so each node summarizes a segment, like a sum, min, max, or gcd. The classic build time is  $O(n)$  and it enables  $O(\log n)$  queries and updates.

#### What Problem Are We Solving?

We want to answer range questions quickly

- Sum on  $[l, r]$
- Min or max on  $[l, r]$
- Counts or bitwise ops on  $[l, r]$

If we precompute a segment tree, each internal node stores the merge of its two children. Then queries and updates touch only  $O(\log n)$  nodes.

Goal Turn an array  $A[0..n-1]$  into a tree  $T$  where  $T[v]$  summarizes a segment  $A[L..R]$ .

#### How It Works (Plain Language)

Represent the tree in an array (1 indexed for simplicity)

- Node  $v$  covers segment  $[L, R]$
- Left child  $2v$  covers  $[L, \text{mid}]$
- Right child  $2v+1$  covers  $[\text{mid}+1, R]$
- $T[v] = \text{merge}(T[2v], T[2v+1])$  where  $\text{merge}$  is sum, min, max, etc.

Build Steps

1. If  $L == R$ , store the leaf value  $A[L]$  in  $T[v]$ .
2. Else split at  $\text{mid}$ , recursively build left and right, then  $T[v] = \text{merge}(\text{left}, \text{right})$ .

Example Build (sum)

Array  $A = [2, 1, 3, 4]$

| Node v | Segment [L,R] | Value |
|--------|---------------|-------|
| 1      | [0,3]         | 10    |
| 2      | [0,1]         | 3     |

| Node v | Segment [L,R] | Value |
|--------|---------------|-------|
| 3      | [2,3]         | 7     |
| 4      | [0,0]         | 2     |
| 5      | [1,1]         | 1     |
| 6      | [2,2]         | 3     |
| 7      | [3,3]         | 4     |

`merge = sum`, so  $T[1] = 3 + 7 = 10$ , leaves store the original values.

Another Example Build (min)

Array A = [5, 2, 6, 1]

| Node v | Segment [L,R] | Value (min) |
|--------|---------------|-------------|
| 1      | [0,3]         | 1           |
| 2      | [0,1]         | 2           |
| 3      | [2,3]         | 1           |
| 4      | [0,0]         | 5           |
| 5      | [1,1]         | 2           |
| 6      | [2,2]         | 6           |
| 7      | [3,3]         | 1           |

### Tiny Code (Easy Versions)

C (iterative storage size 4n, recursive build for sum)

```
#include <stdio.h>

#define MAXN 100000
int A[MAXN];
long long T[4*MAXN];

long long merge(long long a, long long b) { return a + b; }

void build(int v, int L, int R) {
    if (L == R) {
        T[v] = A[L];
        return;
    }
    int mid = (L + R) / 2;
```

```

    build(2*v, L, mid);
    build(2*v+1, mid+1, R);
    T[v] = merge(T[2*v], T[2*v+1]);
}

int main(void) {
    int n = 4;
    A[0]=2; A[1]=1; A[2]=3; A[3]=4;
    build(1, 0, n-1);
    for (int v = 1; v < 8; v++) printf("T[%d]=%lld\n", v, T[v]);
    return 0;
}

```

Python (sum segment tree, recursive build)

```

def build(arr):
    n = len(arr)
    size = 1
    while size < n:
        size <= 1
    T = [0] * (2 * size)

    # leaves
    for i in range(n):
        T[size + i] = arr[i]
    # internal nodes
    for v in range(size - 1, 0, -1):
        T[v] = T[2*v] + T[2*v + 1]
    return T, size # T is 1..size*2-1, leaves start at index size

# Example
A = [2, 1, 3, 4]
T, base = build(A)
# T[1] holds sum of all, T[base+i] holds A[i]
print("Root sum:", T[1])

```

Notes

- The C version shows classic recursive top down build.
- The Python version shows an iterative bottom up build using a power of two base, also  $O(n)$ .

## Why It Matters

- Preprocessing time  $O(n)$  enables
  - Range queries in  $O(\log n)$
  - Point updates in  $O(\log n)$
- Choice of `merge` adapts the tree to many tasks
  - sum, min, max, gcd, bitwise and, custom structs

## A Gentle Proof (Why It Works)

Each element appears in exactly one leaf and contributes to  $O(1)$  nodes per level. The tree has  $O(\log n)$  levels. Total number of nodes built is at most  $2n$ , so total work is  $O(n)$ .

## Try It Yourself

1. Change `merge` to `min` or `max` and rebuild.
2. Build then verify that  $T[1]$  equals `sum(A)`.
3. Print the tree level by level to visualize segments.
4. Extend the structure to support point update and range query.

## Test Cases

| Array A   | Merge | Expected Root T[1] | Notes                        |
|-----------|-------|--------------------|------------------------------|
| [2,1,3,4] | sum   | 10                 | basic sum tree               |
| [5,2,6,1] | min   | 1                  | switch merge to min          |
| [7]       | sum   | 7                  | single element               |
| []        | sum   | 0 or no tree       | handle empty as special case |

## Complexity

| Phase        | Time        | Space        |
|--------------|-------------|--------------|
| Build        | $O(n)$      | $O(n)$       |
| Query        | $O(\log n)$ | $O(1)$ extra |
| Point update | $O(\log n)$ | $O(1)$ extra |

Build once, answer fast forever. A segment tree turns an array into a range answering machine.

## 242 Range Sum Query

A Range Sum Query (RSQ) retrieves the sum of elements in a subarray  $[L, R]$  using a segment tree. Once the tree is built, each query runs in  $O(\log n)$  time, by combining results from the minimal set of segments covering  $[L, R]$ .

### What Problem Are We Solving?

Given an array  $A[0..n-1]$ , we want to answer queries like

`sum(A[L..R])`

quickly, without recalculating from scratch each time.

Naive approach:  $O(R-L+1)$  per query Segment tree approach:  $O(\log n)$  per query after  $O(n)$  build.

### How It Works (Plain Language)

We maintain a segment tree  $T$  built as before ( $T[v] = \text{sum of segment}$ ). To answer `query(v, L, R, qL, qR)`

1. If segment  $[L, R]$  is fully outside  $[qL, qR]$ , return 0.
2. If segment  $[L, R]$  is fully inside, return  $T[v]$ .
3. Otherwise, split at  $\text{mid}$  and combine results from left and right children.

So each query visits only  $O(\log n)$  nodes, each exactly covering part of the query range.

### Example

Array:  $A = [2, 1, 3, 4, 5]$

Query: sum on  $[1,3]$  (1-based: elements  $1..3 = 1+3+4=8$ )

| Node v | Segment $[L,R]$ | Value | Relationship to $[1,3]$ | Contribution |
|--------|-----------------|-------|-------------------------|--------------|
| 1      | $[0,4]$         | 15    | overlaps                | recurse      |
| 2      | $[0,2]$         | 6     | overlaps                | recurse      |
| 3      | $[3,4]$         | 9     | overlaps                | recurse      |
| 4      | $[0,1]$         | 3     | partial                 | recurse      |
| 5      | $[2,2]$         | 3     | inside                  | +3           |
| 8      | $[0,0]$         | 2     | outside                 | skip         |

| Node v | Segment [L,R] | Value | Relationship to [1,3] | Contribution |
|--------|---------------|-------|-----------------------|--------------|
| 9      | [1,1]         | 1     | inside                | +1           |
| 6      | [3,3]         | 4     | inside                | +4           |
| 7      | [4,4]         | 5     | outside               | skip         |

$$\text{Sum} = 1 + 3 + 4 = 8$$

### Step-by-Step Table (for clarity)

| Step | Current Segment | Covered?       | Action   |
|------|-----------------|----------------|----------|
| 1    | [0,4]           | Overlaps [1,3] | Split    |
| 2    | [0,2]           | Overlaps [1,3] | Split    |
| 3    | [3,4]           | Overlaps [1,3] | Split    |
| 4    | [0,1]           | Overlaps       | Split    |
| 5    | [0,0]           | Outside        | Return 0 |
| 6    | [1,1]           | Inside         | Return 1 |
| 7    | [2,2]           | Inside         | Return 3 |
| 8    | [3,3]           | Inside         | Return 4 |
| 9    | [4,4]           | Outside        | Return 0 |

$$\text{Total} = 1+3+4 = 8$$

### Tiny Code (Easy Versions)

C (Recursive RSQ)

```
#include <stdio.h>

#define MAXN 100000
int A[MAXN];
long long T[4*MAXN];

long long merge(long long a, long long b) { return a + b; }

void build(int v, int L, int R) {
    if (L == R) {
        T[v] = A[L];
        return;
    }
    int M = (L + R) / 2;
    build(2*v, L, M);
    build(2*v + 1, M + 1, R);
    T[v] = merge(T[2*v], T[2*v + 1]);
}
```

```

    }

    int mid = (L + R) / 2;
    build(2*v, L, mid);
    build(2*v+1, mid+1, R);
    T[v] = merge(T[2*v], T[2*v+1]);
}

long long query(int v, int L, int R, int qL, int qR) {
    if (qR < L || R < qL) return 0; // disjoint
    if (qL <= L && R <= qR) return T[v]; // fully covered
    int mid = (L + R) / 2;
    long long left = query(2*v, L, mid, qL, qR);
    long long right = query(2*v+1, mid+1, R, qL, qR);
    return merge(left, right);
}

int main(void) {
    int n = 5;
    int Avals[5] = {2,1,3,4,5};
    for (int i=0;i<n;i++) A[i]=Avals[i];
    build(1, 0, n-1);
    printf("Sum [1,3] = %lld\n", query(1,0,n-1,1,3)); // expect 8
}

```

Python (Iterative)

```

def build(arr):
    n = len(arr)
    size = 1
    while size < n:
        size <= 1
        T = [0]*(2*size)
        # leaves
        for i in range(n):
            T[size+i] = arr[i]
        # parents
        for v in range(size-1, 0, -1):
            T[v] = T[2*v] + T[2*v+1]
    return T, size

def query(T, base, l, r):
    l += base

```

```

r += base
res = 0
while l <= r:
    if l % 2 == 1:
        res += T[l]
        l += 1
    if r % 2 == 0:
        res += T[r]
        r -= 1
    l //= 2
    r //= 2
return res

A = [2,1,3,4,5]
T, base = build(A)
print("Sum [1,3] =", query(T, base, 1, 3)) # expect 8

```

## Why It Matters

- Enables fast range queries on static or dynamic arrays.
- Foundation for many extensions:
  - Range min/max query (change merge)
  - Lazy propagation (for range updates)
  - 2D and persistent trees

## A Gentle Proof (Why It Works)

Each level of recursion splits into disjoint segments. At most 2 segments per level are added to the result. Depth =  $O(\log n)$ , so total work =  $O(\log n)$ .

## Try It Yourself

1. Query different  $[L, R]$  ranges after build.
2. Replace `merge` with `min()` or `max()` and verify results.
3. Combine queries to verify overlapping segments are counted once.
4. Add update operation to change elements and re-query.

## Test Cases

| Array A     | Query [L,R] | Expected | Notes          |
|-------------|-------------|----------|----------------|
| [2,1,3,4,5] | [1,3]       | 8        | 1+3+4          |
| [5,5,5,5]   | [0,3]       | 20       | uniform        |
| [1,2,3,4,5] | [2,4]       | 12       | 3+4+5          |
| [7]         | [0,0]       | 7        | single element |

## Complexity

| Operation | Time        | Space  |
|-----------|-------------|--------|
| Query     | $O(\log n)$ | $O(1)$ |
| Build     | $O(n)$      | $O(n)$ |
| Update    | $O(\log n)$ | $O(1)$ |

Segment trees let you ask “*what’s in this range?*”, and get the answer fast, no matter how big the array.

## 243 Range Update (Lazy Propagation Technique)

A range update modifies all elements in a segment  $[L, R]$  efficiently. Without optimization, you’d touch every element,  $O(n)$ . With lazy propagation, you defer work, storing pending updates in a separate array, achieving  $O(\log n)$  time per update and query.

This pattern is essential when many updates overlap, like adding +5 to every element in  $[2, 6]$  repeatedly.

### What Problem Are We Solving?

We want to efficiently support both:

1. Range updates: Add or set a value over  $[L, R]$
2. Range queries: Get sum/min/max over  $[L, R]$

Naive solution:  $O(R-L+1)$  per update. Lazy propagation:  $O(\log n)$  per update/query.

Example goal:

```
add +3 to A[2..5]
then query sum(A[0..7])
```

## How It Works (Plain Language)

Each segment node  $T[v]$  stores summary (e.g. sum). Each node  $\text{lazy}[v]$  stores pending update (not yet pushed to children).

When updating  $[L, R]$ :

- If node's segment is fully inside, apply update directly to  $T[v]$  and mark  $\text{lazy}[v]$  (no recursion).
- If partially overlapping, push pending updates down, then recurse.

When querying  $[L, R]$ :

- Push any pending updates first
- Combine child results as usual

This way, each node is updated at most once per path, giving  $O(\log n)$ .

## Example

Array:  $A = [1, 2, 3, 4, 5, 6, 7, 8]$  Build tree for sum.

Step 1: Range update  $[2, 5] += 3$

| Node | Segment | Action                         | lazy[]    | $T[]$ sum   |
|------|---------|--------------------------------|-----------|-------------|
| 1    | $[0,7]$ | overlaps → recurse             | 0         | unchanged   |
| 2    | $[0,3]$ | overlaps → recurse             | 0         | unchanged   |
| 3    | $[4,7]$ | overlaps → recurse             | 0         | unchanged   |
| 4    | $[0,1]$ | outside                        | -         | -           |
| 5    | $[2,3]$ | inside → add $+3 \times 2 = 6$ | lazy[5]=3 | $T[5] += 6$ |
| 6    | $[4,5]$ | inside → add $+3 \times 2 = 6$ | lazy[6]=3 | $T[6] += 6$ |

Later queries automatically apply +3 to affected subranges.

## Step-by-Step Example

Let's trace two operations:

1. `update(2,5,+3)`
2. `query(0,7)`

| Step   | Action                                          | Result                              |
|--------|-------------------------------------------------|-------------------------------------|
| Build  | $\text{sum} = [1,2,3,4,5,6,7,8] \rightarrow 36$ | $T[1]=36$                           |
| Update | mark lazy for segments fully in $[2,5]$         | $T[v] += 3 \times \text{len}$       |
| Query  | propagate lazy before using $T[v]$              | $\text{sum} = 36 + 3 \times 4 = 48$ |

Result after update: total = 48

### Tiny Code (Easy Versions)

C (Recursive, sum tree with lazy add)

```
#include <stdio.h>

#define MAXN 100000
int A[MAXN];
long long T[4*MAXN], lazy[4*MAXN];

long long merge(long long a, long long b) { return a + b; }

void build(int v, int L, int R) {
    if (L == R) { T[v] = A[L]; return; }
    int mid = (L + R) / 2;
    build(2*v, L, mid);
    build(2*v+1, mid+1, R);
    T[v] = merge(T[2*v], T[2*v+1]);
}

void push(int v, int L, int R) {
    if (lazy[v] != 0) {
        T[v] += lazy[v] * (R - L + 1);
        if (L != R) {
            lazy[2*v] += lazy[v];
            lazy[2*v+1] += lazy[v];
        }
        lazy[v] = 0;
    }
}

void update(int v, int L, int R, int qL, int qR, int val) {
    push(v, L, R);
```

```

if (qR < L || R < qL) return;
if (qL <= L && R <= qR) {
    lazy[v] += val;
    push(v, L, R);
    return;
}
int mid = (L + R) / 2;
update(2*v, L, mid, qL, qR, val);
update(2*v+1, mid+1, R, qL, qR, val);
T[v] = merge(T[2*v], T[2*v+1]);
}

long long query(int v, int L, int R, int qL, int qR) {
    push(v, L, R);
    if (qR < L || R < qL) return 0;
    if (qL <= L && R <= qR) return T[v];
    int mid = (L + R) / 2;
    return merge(query(2*v, L, mid, qL, qR), query(2*v+1, mid+1, R, qL, qR));
}

int main(void) {
    int n = 8;
    int vals[] = {1,2,3,4,5,6,7,8};
    for (int i=0;i<n;i++) A[i]=vals[i];
    build(1,0,n-1);
    update(1,0,n-1,2,5,3);
    printf("Sum [0,7] = %lld\n", query(1,0,n-1,0,7)); // expect 48
}

```

Python (Iterative version)

```

class LazySegTree:
    def __init__(self, arr):
        n = len(arr)
        size = 1
        while size < n: size <= 1
        self.n = n; self.size = size
        self.T = [0]*(2*size)
        self.lazy = [0]*(2*size)
        for i in range(n): self.T[size+i] = arr[i]
        for i in range(size-1, 0, -1):
            self.T[i] = self.T[2*i] + self.T[2*i+1]

```

```

def _apply(self, v, val, length):
    self.T[v] += val * length
    if v < self.size:
        self.lazy[v] += val

def _push(self, v, length):
    if self.lazy[v]:
        self._apply(2*v, self.lazy[v], length//2)
        self._apply(2*v+1, self.lazy[v], length//2)
        self.lazy[v] = 0

def update(self, l, r, val):
    def _upd(v, L, R):
        if r < L or R < l: return
        if l <= L and R <= r:
            self._apply(v, val, R-L+1)
            return
        self._push(v, R-L+1)
        mid = (L+R)//2
        _upd(2*v, L, mid)
        _upd(2*v+1, mid+1, R)
        self.T[v] = self.T[2*v] + self.T[2*v+1]
        _upd(1, 0, self.size-1)

    def query(self, l, r):
        def _qry(v, L, R):
            if r < L or R < l: return 0
            if l <= L and R <= r:
                return self.T[v]
            self._push(v, R-L+1)
            mid = (L+R)//2
            return _qry(2*v, L, mid) + _qry(2*v+1, mid+1, R)
        return _qry(1, 0, self.size-1)

A = [1,2,3,4,5,6,7,8]
st = LazySegTree(A)
st.update(2,5,3)
print("Sum [0,7] =", st.query(0,7)) # expect 48

```

## Why It Matters

- Crucial for problems with many overlapping updates
- Used in range-add, range-assign, interval covering, and 2D extensions
- Foundation for Segment Tree Beats

## A Gentle Proof (Why It Works)

Each update marks at most one node per level as “lazy.” When queried later, we “push” those updates downward once. Each node’s pending updates applied  $O(1)$  times  $\rightarrow$  total cost  $O(\log n)$ .

## Try It Yourself

1. Build  $[1,2,3,4,5] \rightarrow$  update  $[1,3] += 2 \rightarrow$  query sum  $[0,4] \rightarrow$  expect 21
2. Update  $[0,4] += 1 \rightarrow$  query  $[2,4] \rightarrow$  expect  $3+2+2+1+1 = 15$
3. Combine multiple updates  $\rightarrow$  verify cumulative results

## Test Cases

| Operation           | Array               | Query                | Expected         |
|---------------------|---------------------|----------------------|------------------|
| update $[2,5] += 3$ | $[1,2,3,4,5,6,7,8]$ | sum $[0,7]$          | 48               |
| update $[0,3] += 2$ | $[5,5,5,5]$         | sum $[1,2]$          | 14               |
| two updates         | $[1,1,1,1,1]$       | $[1,3] +1, [2,4] +2$ | $[1,3]$ sum = 10 |

## Complexity

| Operation    | Time        | Space  |
|--------------|-------------|--------|
| Range update | $O(\log n)$ | $O(n)$ |
| Range query  | $O(\log n)$ | $O(n)$ |
| Build        | $O(n)$      | $O(n)$ |

Lazy propagation, work smarter, not harder. Apply only what’s needed, when it’s needed.

## 244 Point Update

A point update changes a single element in the array and updates all relevant segment tree nodes along the path to the root. This operation ensures the segment tree remains consistent for future range queries.

Unlike range updates (which mark many elements at once), point updates touch only  $O(\log n)$  nodes, one per level.

### What Problem Are We Solving?

Given a segment tree built over  $A[0..n-1]$ , we want to:

- Change one element:  $A[pos] = new\_value$
- Reflect the change in the segment tree so all range queries stay correct

Goal: Update efficiently, no full rebuild

Naive: rebuild tree  $\rightarrow O(n)$  Segment tree point update:  $O(\log n)$

### How It Works (Plain Language)

Every tree node  $T[v]$  represents a segment  $[L, R]$ . When  $pos$  lies within  $[L, R]$ , that node's value might need adjustment.

Algorithm (recursive):

1. If  $L == R == pos$ , assign  $A[pos] = val$ , set  $T[v] = val$ .
2. Else, find  $mid$ .
  - Recurse into left or right child depending on  $pos$ .
  - After child updates, recompute  $T[v] = merge(T[2v], T[2v+1])$ .

No lazy propagation needed, it's a direct path down the tree.

| Node | Range | Value |
|------|-------|-------|
|------|-------|-------|

### Example

Array: A = [2, 1, 3, 4] Tree stores sum.

| Node | Range | Value |
|------|-------|-------|
| 1    | [0,3] | 10    |
| 2    | [0,1] | 3     |
| 3    | [2,3] | 7     |
| 4    | [0,0] | 2     |
| 5    | [1,1] | 1     |
| 6    | [2,2] | 3     |
| 7    | [3,3] | 4     |

Operation: A[1] = 5

Path: [1, 2, 5]

| Step   | Node | Old Value | New Value | Update      |
|--------|------|-----------|-----------|-------------|
| Leaf   | 5    | 1         | 5         | T[5]=5      |
| Parent | 2    | 3         | 7         | T[2]=2+5=7  |
| Root   | 1    | 10        | 14        | T[1]=7+7=14 |

New array: [2, 5, 3, 4] New sum: 14

### Step-by-Step Trace

```

update(v=1, L=0, R=3, pos=1, val=5)
  mid=1
  -> pos<=mid → left child (v=2)
    update(v=2, L=0, R=1)
      mid=0
      -> pos>mid → right child (v=5)
        update(v=5, L=1, R=1)
        T[5]=5
        T[2]=merge(T[4]=2, T[5]=5)=7
        T[1]=merge(T[2]=7, T[3]=7)=14

```

## Tiny Code (Easy Versions)

C (Recursive, sum merge)

```
#include <stdio.h>

#define MAXN 100000
int A[MAXN];
long long T[4*MAXN];

long long merge(long long a, long long b) { return a + b; }

void build(int v, int L, int R) {
    if (L == R) { T[v] = A[L]; return; }
    int mid = (L + R)/2;
    build(2*v, L, mid);
    build(2*v+1, mid+1, R);
    T[v] = merge(T[2*v], T[2*v+1]);
}

void point_update(int v, int L, int R, int pos, int val) {
    if (L == R) {
        T[v] = val;
        A[pos] = val;
        return;
    }
    int mid = (L + R)/2;
    if (pos <= mid) point_update(2*v, L, mid, pos, val);
    else point_update(2*v+1, mid+1, R, pos, val);
    T[v] = merge(T[2*v], T[2*v+1]);
}

long long query(int v, int L, int R, int qL, int qR) {
    if (qR < L || R < qL) return 0;
    if (qL <= L && R <= qR) return T[v];
    int mid = (L + R)/2;
    return merge(query(2*v, L, mid, qL, qR), query(2*v+1, mid+1, R, qL, qR));
}

int main(void) {
    int n = 4;
    int vals[] = {2,1,3,4};
    for (int i=0;i<n;i++) A[i]=vals[i];
```

```

build(1,0,n-1);
printf("Before: sum [0,3] = %lld\n", query(1,0,n-1,0,3)); // 10
point_update(1,0,n-1,1,5);
printf("After:  sum [0,3] = %lld\n", query(1,0,n-1,0,3)); // 14
}

```

Python (Iterative)

```

def build(arr):
    n = len(arr)
    size = 1
    while size < n: size <= 1
    T = [0]*(2*size)
    for i in range(n):
        T[size+i] = arr[i]
    for v in range(size-1, 0, -1):
        T[v] = T[2*v] + T[2*v+1]
    return T, size

def update(T, base, pos, val):
    v = base + pos
    T[v] = val
    v // 2
    while v >= 1:
        T[v] = T[2*v] + T[2*v+1]
        v // 2

def query(T, base, l, r):
    l += base; r += base
    res = 0
    while l <= r:
        if l%2==1: res += T[l]; l+=1
        if r%2==0: res += T[r]; r-=1
        l//=2; r//=2
    return res

A = [2,1,3,4]
T, base = build(A)
print("Before:", query(T, base, 0, 3)) # 10
update(T, base, 1, 5)
print("After:", query(T, base, 0, 3)) # 14

```

## Why It Matters

- Foundation for dynamic data, quick local edits
- Used in segment trees, Fenwick trees, and BIT
- Great for applications like dynamic scoring, cumulative sums, real-time data updates

## A Gentle Proof (Why It Works)

Each level has 1 node affected by position  $\text{pos}$ . Tree height =  $O(\log n)$ . Thus, exactly  $O(\log n)$  nodes recomputed.

## Try It Yourself

1. Build  $[2,1,3,4]$ , update  $A[2]=10 \rightarrow \text{sum } [0,3]=17$
2. Replace merge with  $\min \rightarrow$  update element  $\rightarrow$  test queries
3. Compare time with full rebuild for large  $n$

## Test Cases

| Input A     | Update   | Query             | Expected |
|-------------|----------|-------------------|----------|
| $[2,1,3,4]$ | $A[1]=5$ | $\text{sum}[0,3]$ | 14       |
| $[5,5,5]$   | $A[2]=2$ | $\text{sum}[0,2]$ | 12       |
| $[1]$       | $A[0]=9$ | $\text{sum}[0,0]$ | 9        |

## Complexity

| Operation    | Time        | Space  |
|--------------|-------------|--------|
| Point update | $O(\log n)$ | $O(1)$ |
| Query        | $O(\log n)$ | $O(1)$ |
| Build        | $O(n)$      | $O(n)$ |

A point update is like a ripple in a pond, a single change flows upward, keeping the whole structure in harmony.

## 245 Fenwick Tree Build

A Fenwick Tree, or Binary Indexed Tree (BIT), is a compact data structure for prefix queries and point updates. It's perfect for cumulative sums, frequencies, or any associative operation. Building it efficiently sets the stage for  $O(\log n)$  queries and updates.

Unlike segment trees, a Fenwick Tree uses clever index arithmetic to represent overlapping ranges in a single array.

### What Problem Are We Solving?

We want to precompute a structure to:

- Answer prefix queries: `sum(0..i)`
- Support updates: `A[i] += delta`

Naive prefix sum array:

- Query:  $O(1)$
- Update:  $O(n)$

Fenwick Tree:

- Query:  $O(\log n)$
- Update:  $O(\log n)$
- Build:  $O(n)$

### How It Works (Plain Language)

A Fenwick Tree uses the last set bit (LSB) of an index to determine segment length.

Each node `BIT[i]` stores sum of range:

$$(i - \text{LSB}(i) + 1) \dots i$$

So:

- `BIT[1]` stores  $A[1]$
- `BIT[2]` stores  $A[1..2]$
- `BIT[3]` stores  $A[3]$
- `BIT[4]` stores  $A[1..4]$
- `BIT[5]` stores  $A[5]$
- `BIT[6]` stores  $A[5..6]$

Prefix sum = sum of these overlapping ranges.

## Example

Array A = [2, 1, 3, 4, 5] (1-indexed for clarity)

| i | A[i] | LSB(i) | Range Stored | BIT[i] (sum) |
|---|------|--------|--------------|--------------|
| 1 | 2    | 1      | [1]          | 2            |
| 2 | 1    | 2      | [1..2]       | 3            |
| 3 | 3    | 1      | [3]          | 3            |
| 4 | 4    | 4      | [1..4]       | 10           |
| 5 | 5    | 1      | [5]          | 5            |

## Step-by-Step Build (O(n))

Iterate i from 1 to n:

1. BIT[i] += A[i]
2. Add BIT[i] to its parent: BIT[i + LSB(i)] += BIT[i]

| Step | i | LSB(i) | Update BIT[i+LSB(i)]               | Result BIT  |
|------|---|--------|------------------------------------|-------------|
| 1    | 1 | 1      | BIT[2] += 2                        | [2,3,0,0,0] |
| 2    | 2 | 2      | BIT[4] += 3                        | [2,3,0,3,0] |
| 3    | 3 | 1      | BIT[4] += 3                        | [2,3,3,6,0] |
| 4    | 4 | 4      | BIT[8] += 6 (ignore, out of range) | [2,3,3,6,0] |
| 5    | 5 | 1      | BIT[6] += 5 (ignore, out of range) | [2,3,3,6,5] |

Built BIT: [2, 3, 3, 6, 5]

## Tiny Code (Easy Versions)

C (O(n) Build)

```
#include <stdio.h>

#define MAXN 100005
int A[MAXN];
long long BIT[MAXN];
int n;

int lsb(int i) { return i & -i; }
```

```

void build() {
    for (int i = 1; i <= n; i++) {
        BIT[i] += A[i];
        int parent = i + lsb(i);
        if (parent <= n)
            BIT[parent] += BIT[i];
    }
}

long long prefix_sum(int i) {
    long long s = 0;
    while (i > 0) {
        s += BIT[i];
        i -= lsb(i);
    }
    return s;
}

int main(void) {
    n = 5;
    int vals[6] = {0, 2, 1, 3, 4, 5}; // 1-indexed
    for (int i=1;i<=n;i++) A[i]=vals[i];
    build();
    printf("Prefix sum [1..3] = %lld\n", prefix_sum(3)); // expect 6
}

```

Python (1-indexed)

```

def build(A):
    n = len(A) - 1
    BIT = [0]*(n+1)
    for i in range(1, n+1):
        BIT[i] += A[i]
        parent = i + (i & -i)
        if parent <= n:
            BIT[parent] += BIT[i]
    return BIT

def prefix_sum(BIT, i):
    s = 0
    while i > 0:

```

```

    s += BIT[i]
    i -= (i & -i)
return s

A = [0,2,1,3,4,5]  # 1-indexed
BIT = build(A)
print("BIT =", BIT[1:])
print("Sum[1..3] =", prefix_sum(BIT, 3))  # expect 6

```

## Why It Matters

- Lightweight alternative to segment tree
- O(n) build, O(log n) query, O(log n) update
- Used in frequency tables, inversion count, prefix queries, cumulative histograms

## A Gentle Proof (Why It Works)

Each index  $i$  contributes to at most  $\log n$  BIT entries. Every  $\text{BIT}[i]$  stores sum of a disjoint range defined by LSB. Building with parent propagation ensures correct overlapping coverage.

## Try It Yourself

1. Build BIT from  $[2, 1, 3, 4, 5] \rightarrow \text{query sum}(3)=6$
2. Update  $A[2]+=2 \rightarrow \text{prefix}(3)=8$
3. Compare with cumulative sum array for correctness

## Test Cases

| A (1-indexed) | Query              | Expected |
|---------------|--------------------|----------|
| $[2,1,3,4,5]$ | $\text{prefix}(3)$ | 6        |
| $[5,5,5,5]$   | $\text{prefix}(4)$ | 20       |
| $[1]$         | $\text{prefix}(1)$ | 1        |

## Complexity

| Operation | Time        | Space  |
|-----------|-------------|--------|
| Build     | $O(n)$      | $O(n)$ |
| Query     | $O(\log n)$ | $O(1)$ |
| Update    | $O(\log n)$ | $O(1)$ |

A Fenwick Tree is the art of doing less, storing just enough to make prefix sums lightning-fast.

## 246 Fenwick Update

A Fenwick Tree (Binary Indexed Tree) supports point updates, adjusting a single element and efficiently reflecting that change across all relevant cumulative sums. The update propagates upward through parent indices determined by the Least Significant Bit (LSB).

### What Problem Are We Solving?

Given a built Fenwick Tree, we want to perform an operation like:

```
A[pos] += delta
```

and keep all prefix sums `sum(0..i)` consistent.

Naive approach: update every prefix  $\rightarrow O(n)$  Fenwick Tree: propagate through selected indices  $\rightarrow O(\log n)$

### How It Works (Plain Language)

In a Fenwick Tree, `BIT[i]` covers the range  $(i - \text{LSB}(i) + 1) \dots i$ . So when we update `A[pos]`, we must add delta to all `BIT[i]` where  $i$  includes `pos` in its range.

Rule:

```
for (i = pos; i <= n; i += LSB(i))
    BIT[i] += delta
```

- $\text{LSB}(i)$  jumps to the next index covering `pos`
- Stops when index exceeds `n`

## Example

Array A = [2, 1, 3, 4, 5] (1-indexed) BIT built as [2, 3, 3, 10, 5]

Now perform update(2, +2) → A[2] = 3

| Step | i | LSB(i) | BIT[i] | Change | New BIT          |
|------|---|--------|--------|--------|------------------|
| 1    | 2 | 2      | BIT[2] | $+= 2$ | [2, 5, 3, 10, 5] |
| 2    | 4 | 4      | BIT[4] | $+= 2$ | [2, 5, 3, 12, 5] |
| 3    | 8 | stop   | -      |        | done             |

New prefix sums reflect updated array [2, 3, 3, 4, 5]

Check prefix(3) = 2+3+3=8

## Step-by-Step Table

| Prefix | Old Sum | New Sum |
|--------|---------|---------|
| 1      | 2       | 2       |
| 2      | 3       | 5       |
| 3      | 6       | 8       |
| 4      | 10      | 12      |
| 5      | 15      | 17      |

## Tiny Code (Easy Versions)

C (Fenwick Update)

```
#include <stdio.h>

#define MAXN 100005
long long BIT[MAXN];
int n;

int lsb(int i) { return i & -i; }

void update(int pos, int delta) {
    for (int i = pos; i <= n; i += lsb(i))
        BIT[i] += delta;
```

```

}

long long prefix_sum(int i) {
    long long s = 0;
    for (; i > 0; i -= lsb(i))
        s += BIT[i];
    return s;
}

int main(void) {
    n = 5;
    // Build from A = [2, 1, 3, 4, 5]
    BIT[1]=2; BIT[2]=3; BIT[3]=3; BIT[4]=10; BIT[5]=5;
    update(2, 2); // A[2]+=2
    printf("Sum [1..3] = %lld\n", prefix_sum(3)); // expect 8
}

```

Python (1-indexed)

```

def update(BIT, n, pos, delta):
    while pos <= n:
        BIT[pos] += delta
        pos += (pos & -pos)

def prefix_sum(BIT, pos):
    s = 0
    while pos > 0:
        s += BIT[pos]
        pos -= (pos & -pos)
    return s

# Example
BIT = [0,2,3,3,10,5] # built from [2,1,3,4,5]
update(BIT, 5, 2, 2)
print("Sum [1..3] =", prefix_sum(BIT, 3)) # expect 8

```

## Why It Matters

- Enables real-time data updates with fast prefix queries
- Simpler and more space-efficient than segment trees for sums
- Core of many algorithms: inversion count, frequency accumulation, order statistics

## A Gentle Proof (Why It Works)

Each  $\text{BIT}[i]$  covers a fixed range ( $i - \text{LSB}(i) + 1 \dots i$ ). If  $\text{pos}$  lies in that range, incrementing  $\text{BIT}[i]$  ensures correct future prefix sums. Since each update moves by  $\text{LSB}(i)$ , at most  $\log(n)$  steps occur.

## Try It Yourself

1. Build BIT from  $[2,1,3,4,5] \rightarrow \text{update}(2,+2)$
2. Query  $\text{prefix}(3) \rightarrow \text{expect } 8$
3.  $\text{Update}(5, +5) \rightarrow \text{prefix}(5) = 22$
4. Chain multiple updates  $\rightarrow$  verify incremental sums

## Test Cases

| A (1-indexed) | Update   | Query              | Expected |
|---------------|----------|--------------------|----------|
| $[2,1,3,4,5]$ | $(2,+2)$ | $\text{sum}[1..3]$ | 8        |
| $[5,5,5,5]$   | $(4,+1)$ | $\text{sum}[1..4]$ | 21       |
| $[1,2,3,4,5]$ | $(5,-2)$ | $\text{sum}[1..5]$ | 13       |

## Complexity

| Operation | Time        | Space  |
|-----------|-------------|--------|
| Update    | $O(\log n)$ | $O(1)$ |
| Query     | $O(\log n)$ | $O(1)$ |
| Build     | $O(n)$      | $O(n)$ |

Each update is a ripple that climbs the tree, keeping all prefix sums in perfect sync.

## 247 Fenwick Query

Once you've built or updated a Fenwick Tree (Binary Indexed Tree), you'll want to extract useful information, usually prefix sums. The query operation walks downward through the tree using index arithmetic, gathering partial sums along the way.

This simple yet powerful routine turns a linear prefix-sum scan into an elegant  $O(\log n)$  solution.

## What Problem Are We Solving?

We need to compute:

```
sum(1..i) = A[1] + A[2] + ... + A[i]
```

efficiently, after updates.

In a Fenwick Tree, each index holds the sum of a segment determined by its LSB (Least Significant Bit). By moving downward (subtracting LSB each step), we collect disjoint ranges that together cover  $[1..i]$ .

## How It Works (Plain Language)

Each  $\text{BIT}[i]$  stores sum of range  $(i - \text{LSB}(i) + 1 .. i)$ . So to get the prefix sum, we combine all these segments by walking downward:

```
sum = 0
while i > 0:
    sum += BIT[i]
    i -= LSB(i)
```

Key Insight:

- Update: moves upward ( $i += \text{LSB}(i)$ )
- Query: moves downward ( $i -= \text{LSB}(i)$ )

Together, they mirror each other like yin and yang of cumulative logic.

## Example

Suppose we have  $A = [2, 3, 3, 4, 5]$  (1-indexed), Built  $\text{BIT} = [2, 5, 3, 12, 5]$

Let's compute `prefix_sum(5)`

| Step | i | $\text{BIT}[i]$ | $\text{LSB}(i)$ | Accumulated Sum | Explanation                       |
|------|---|-----------------|-----------------|-----------------|-----------------------------------|
| 1    | 5 | 5               | 1               | 5               | Add $\text{BIT}[5]$ ( $A[5]$ )    |
| 2    | 4 | 12              | 4               | 17              | Add $\text{BIT}[4]$ ( $A[1..4]$ ) |
| 3    | 0 | -               | -               | stop            | done                              |

`prefix_sum(5) = 17` matches  $2+3+3+4+5=17$

Now try `prefix_sum(3)`

| Step | i | BIT[i] | LSB(i) | Sum  |
|------|---|--------|--------|------|
| 1    | 3 | 3      | 1      | 3    |
| 2    | 2 | 5      | 2      | 8    |
| 3    | 0 | -      | -      | stop |

`prefix_sum(3) = 8`

### Tiny Code (Easy Versions)

C (Fenwick Query)

```
#include <stdio.h>

#define MAXN 100005
long long BIT[MAXN];
int n;

int lsb(int i) { return i & -i; }

long long prefix_sum(int i) {
    long long s = 0;
    while (i > 0) {
        s += BIT[i];
        i -= lsb(i);
    }
    return s;
}

int main(void) {
    n = 5;
    long long B[6] = {0,2,5,3,12,5}; // built BIT
    for (int i=1;i<=n;i++) BIT[i] = B[i];
    printf("Prefix sum [1..3] = %lld\n", prefix_sum(3)); // expect 8
    printf("Prefix sum [1..5] = %lld\n", prefix_sum(5)); // expect 17
}
```

Python (1-indexed)

```

def prefix_sum(BIT, i):
    s = 0
    while i > 0:
        s += BIT[i]
        i -= (i & -i)
    return s

BIT = [0,2,5,3,12,5]
print("sum[1..3] =", prefix_sum(BIT, 3)) # 8
print("sum[1..5] =", prefix_sum(BIT, 5)) # 17

```

## Why It Matters

- Fast prefix sums after dynamic updates
- Essential for frequency tables, order statistics, inversion count
- Core to many competitive programming tricks (like “count less than k”)

## A Gentle Proof (Why It Works)

Each index  $i$  contributes to a fixed set of BIT nodes. When querying, we collect all BIT segments that together form  $[1..i]$ . The LSB ensures no overlap, each range is disjoint. Total steps = number of bits set in  $i$  =  $O(\log n)$ .

## Try It Yourself

1. Build BIT from  $[2,3,3,4,5]$
2. Query  $\text{prefix\_sum}(3) \rightarrow$  expect 8
3. Update( $2+2$ ), query( $3$ )  $\rightarrow$  expect 10
4. Query  $\text{prefix\_sum}(5) \rightarrow$  confirm correctness

## Test Cases

| A (1-indexed) | Query           | Expected |
|---------------|-----------------|----------|
| $[2,3,3,4,5]$ | $\text{sum}(3)$ | 8        |
| $[2,3,3,4,5]$ | $\text{sum}(5)$ | 17       |
| $[1,2,3,4,5]$ | $\text{sum}(4)$ | 10       |

## Complexity

| Operation | Time        | Space  |
|-----------|-------------|--------|
| Query     | $O(\log n)$ | $O(1)$ |
| Update    | $O(\log n)$ | $O(1)$ |
| Build     | $O(n)$      | $O(n)$ |

Fenwick query is the graceful descent, follow the bits down to gather every piece of the sum puzzle.

## 248 Segment Tree Merge

A Segment Tree can handle more than sums, it's a versatile structure that can merge results from two halves of a range using any associative operation (sum, min, max, gcd, etc.). The merge function is the heart of the segment tree: it tells us how to combine child nodes into a parent node.

### What Problem Are We Solving?

We want to combine results from left and right child intervals into one parent result. Without merge logic, the tree can't aggregate data or answer queries.

For example:

- For sum queries →  $\text{merge}(a, b) = a + b$
- For min queries →  $\text{merge}(a, b) = \min(a, b)$
- For gcd queries →  $\text{merge}(a, b) = \gcd(a, b)$

So the merge function defines *what the tree means*.

### How It Works (Plain Language)

Each node represents a segment  $[L, R]$ . Its value is derived from its two children:

```
node = merge(left_child, right_child)
```

When you:

- Build: compute node from children recursively
- Query: merge partial overlaps

- Update: recompute affected nodes using merge

So merge is the unifying rule that glues the tree together.

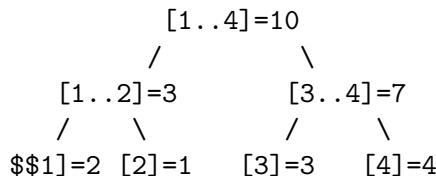
### **Example (Sum Segment Tree)**

Array A = [2, 1, 3, 4]

| Node  | Range  | Left | Right | Merge (sum) |
|-------|--------|------|-------|-------------|
| root  | [1..4] | 6    | 4     | 10          |
| left  | [1..2] | 2    | 1     | 3           |
| right | [3..4] | 3    | 4     | 7           |

Each parent = sum(left, right)

Tree structure:



Merge rule: `merge(a, b) = a + b`

### **Example (Min Segment Tree)**

Array A = [5, 2, 7, 1] Merge rule: `min(a, b)`

| Node  | Range  | Left | Right | Merge (min) |
|-------|--------|------|-------|-------------|
| root  | [1..4] | 2    | 1     | 1           |
| left  | [1..2] | 5    | 2     | 2           |
| right | [3..4] | 7    | 1     | 1           |

Result: root stores 1, the global minimum.

## Tiny Code (Easy Versions)

C (Sum Segment Tree Merge)

```
#include <stdio.h>
#define MAXN 100005

int A[MAXN], tree[4*MAXN];
int n;

int merge(int left, int right) {
    return left + right;
}

void build(int node, int l, int r) {
    if (l == r) {
        tree[node] = A[l];
        return;
    }
    int mid = (l + r) / 2;
    build(node*2, l, mid);
    build(node*2+1, mid+1, r);
    tree[node] = merge(tree[node*2], tree[node*2+1]);
}

int query(int node, int l, int r, int ql, int qr) {
    if (qr < l || ql > r) return 0; // neutral element
    if (ql <= l && r <= qr) return tree[node];
    int mid = (l + r) / 2;
    int left = query(node*2, l, mid, ql, qr);
    int right = query(node*2+1, mid+1, r, ql, qr);
    return merge(left, right);
}

int main(void) {
    n = 4;
    int vals[5] = {0, 2, 1, 3, 4};
    for (int i=1; i<=n; i++) A[i] = vals[i];
    build(1,1,n);
    printf("Sum [1..4] = %d\n", query(1,1,n,1,4)); // 10
    printf("Sum [2..3] = %d\n", query(1,1,n,2,3)); // 4
}
```

Python

```
def merge(a, b):
    return a + b # define operation here

def build(arr, tree, node, l, r):
    if l == r:
        tree[node] = arr[l]
        return
    mid = (l + r) // 2
    build(arr, tree, 2*node, l, mid)
    build(arr, tree, 2*node+1, mid+1, r)
    tree[node] = merge(tree[2*node], tree[2*node+1])

def query(tree, node, l, r, ql, qr):
    if qr < l or ql > r:
        return 0 # neutral element for sum
    if ql <= l and r <= qr:
        return tree[node]
    mid = (l + r)//2
    left = query(tree, 2*node, l, mid, ql, qr)
    right = query(tree, 2*node+1, mid+1, r, ql, qr)
    return merge(left, right)

A = [0, 2, 1, 3, 4]
n = 4
tree = [0]*(4*n)
build(A, tree, 1, 1, n)
print("Sum[1..4] =", query(tree, 1, 1, n, 1, 4)) # 10
```

## Why It Matters

- Merge is the “soul” of the segment tree, define merge, and you define the tree’s purpose.
- Flexible across many tasks: sums, min/max, GCD, XOR, matrix multiplication.
- Unified pattern: build, query, update all rely on the same operation.

## A Gentle Proof (Why It Works)

Segment tree works by divide and conquer. If an operation is associative (like  $+$ ,  $\min$ ,  $\max$ ,  $\gcd$ ), merging partial results yields the same answer as computing on the whole range. So as long as  $\text{merge}(a, b) = \text{merge}(b, a)$  and associative, correctness follows.

## Try It Yourself

1. Replace merge with `min(a,b)` → build min segment tree
2. Replace merge with `max(a,b)` → build max segment tree
3. Replace merge with `_gcd(a,b)` → build gcd tree
4. Test `query(2,3)` after changing merge rule

## Test Cases

| A          | Operation | Query  | Expected |
|------------|-----------|--------|----------|
| [2,1,3,4]  | sum       | [1..4] | 10       |
| [5,2,7,1]  | min       | [1..4] | 1        |
| [3,6,9,12] | gcd       | [2..4] | 3        |

## Complexity

| Operation | Time        | Space  |
|-----------|-------------|--------|
| Build     | $O(n)$      | $O(n)$ |
| Query     | $O(\log n)$ | $O(n)$ |
| Update    | $O(\log n)$ | $O(n)$ |

A merge function is the heartbeat of every segment tree, once you define it, your tree learns what “combine” means.

## 249 Persistent Segment Tree

A Persistent Segment Tree is a magical structure that lets you time-travel. Instead of overwriting nodes on update, it creates new versions while preserving old ones. Each update returns a new root, giving you full history and  $O(\log n)$  access to every version.

### What Problem Are We Solving?

We need a data structure that supports:

- Point updates without losing past state
- Range queries on any historical version

Use cases:

- Undo/rollback systems
- Versioned databases
- Offline queries (“What was sum[1..3] after the 2nd update?”)

Each version is immutable, perfect for functional programming or auditability.

### How It Works (Plain Language)

A persistent segment tree clones only nodes along the path from root to updated leaf. All other nodes are shared.

For each update:

1. Create a new root.
2. Copy nodes along the path to the changed index.
3. Reuse unchanged subtrees.

Result:  $O(\log n)$  memory per version, not  $O(n)$ .

### Example

Array A = [1, 2, 3, 4]

Version 0: built tree for [1, 2, 3, 4]

- sum = 10

Update A[2] = 5

- Create Version 1, copy path to A[2]
- New sum =  $1 + 5 + 3 + 4 = 13$

| Version | A         | Sum(1..4) | Sum(1..2) |
|---------|-----------|-----------|-----------|
| 0       | [1,2,3,4] | 10        | 3         |
| 1       | [1,5,3,4] | 13        | 6         |

Both versions exist side by side. Version 0 is unchanged. Version 1 reflects new value.

## Visualization

```
Version 0: root0
      /
      [1..2]=3    [3..4]=7
      /   \     /   \
[1]=1 [2]=2 [3]=3 [4]=4

Version 1: root1 (new)
      /
      [1..2]=6*    [3..4]=7 (shared)
      /   \
$$1]=1    [2]=5*
```

Asterisks mark newly created nodes.

## Tiny Code (Easy Version)

C (Pointer-based, sum tree)

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    int val;
    struct Node *left, *right;
} Node;

Node* build(int arr[], int l, int r) {
    Node* node = malloc(sizeof(Node));
    if (l == r) {
        node->val = arr[l];
        node->left = node->right = NULL;
        return node;
    }
    int mid = (l + r) / 2;
    node->left = build(arr, l, mid);
    node->right = build(arr, mid+1, r);
    node->val = node->left->val + node->right->val;
    return node;
}
```

```

Node* update(Node* prev, int l, int r, int pos, int new_val) {
    Node* node = malloc(sizeof(Node));
    if (l == r) {
        node->val = new_val;
        node->left = node->right = NULL;
        return node;
    }
    int mid = (l + r) / 2;
    if (pos <= mid) {
        node->left = update(prev->left, l, mid, pos, new_val);
        node->right = prev->right;
    } else {
        node->left = prev->left;
        node->right = update(prev->right, mid+1, r, pos, new_val);
    }
    node->val = node->left->val + node->right->val;
    return node;
}

int query(Node* node, int l, int r, int ql, int qr) {
    if (qr < l || ql > r) return 0;
    if (ql <= l && r <= qr) return node->val;
    int mid = (l + r)/2;
    return query(node->left, l, mid, ql, qr)
        + query(node->right, mid+1, r, ql, qr);
}

int main(void) {
    int A[5] = {0,1,2,3,4}; // 1-indexed
    Node* root0 = build(A, 1, 4);
    Node* root1 = update(root0, 1, 4, 2, 5);
    printf("v0 sum[1..2]=%d\n", query(root0,1,4,1,2)); // 3
    printf("v1 sum[1..2]=%d\n", query(root1,1,4,1,2)); // 6
}

```

Python (Recursive, sum tree)

```

class Node:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

```

```

def build(arr, l, r):
    if l == r:
        return Node(arr[l])
    mid = (l + r) // 2
    left = build(arr, l, mid)
    right = build(arr, mid+1, r)
    return Node(left.val + right.val, left, right)

def update(prev, l, r, pos, val):
    if l == r:
        return Node(val)
    mid = (l + r) // 2
    if pos <= mid:
        return Node(prev.val - prev.left.val + val,
                    update(prev.left, l, mid, pos, val),
                    prev.right)
    else:
        return Node(prev.val - prev.right.val + val,
                    prev.left,
                    update(prev.right, mid+1, r, pos, val))

def query(node, l, r, ql, qr):
    if qr < l or ql > r: return 0
    if ql <= l and r <= qr: return node.val
    mid = (l + r)//2
    return query(node.left, l, mid, ql, qr) + query(node.right, mid+1, r, ql, qr)

A = [0,1,2,3,4]
root0 = build(A, 1, 4)
root1 = update(root0, 1, 4, 2, 5)
print("v0 sum[1..2] =", query(root0, 1, 4, 1, 2)) # 3
print("v1 sum[1..2] =", query(root1, 1, 4, 1, 2)) # 6

```

## Why It Matters

- Immutable versions → ideal for undo systems, snapshots, persistent databases
- Saves memory ( $O(\log n)$  per version)
- Each version is fully functional and independent

## A Gentle Proof (Why It Works)

Each update affects  $O(\log n)$  nodes. By copying only those nodes, we maintain  $O(\log n)$  new memory. Because all old nodes remain referenced, no data is lost. Thus, each version is consistent and immutable.

## Try It Yourself

1. Build version 0 with  $[1,2,3,4]$
2. Update(2,5) → version 1
3. Query sum(1,4) in both versions → 10, 13
4. Create version 2 by update(3,1)
5. Query sum(1,3) in each version

## Test Cases

| Version | A           | Query     | Expected |
|---------|-------------|-----------|----------|
| 0       | $[1,2,3,4]$ | sum[1..2] | 3        |
| 1       | $[1,5,3,4]$ | sum[1..2] | 6        |
| 2       | $[1,5,1,4]$ | sum[1..3] | 7        |

## Complexity

| Operation | Time        | Space (per version) |
|-----------|-------------|---------------------|
| Build     | $O(n)$      | $O(n)$              |
| Update    | $O(\log n)$ | $O(\log n)$         |
| Query     | $O(\log n)$ | $O(1)$              |

Each version of a persistent segment tree is a snapshot in time, perfect memory of every past, with no cost of forgetting.

## 250 2D Segment Tree

A 2D Segment Tree extends the classic segment tree into two dimensions, perfect for handling range queries over matrices, such as sums, minimums, or maximums across subrectangles.

It's the data structure that lets you ask:

“What’s the sum of elements in the rectangle  $(x_1, y_1)$  to  $(x_2, y_2)$ ?” and still answer in  $O(\log^2 n)$  time.

## What Problem Are We Solving?

We want to perform two operations efficiently on a 2D grid:

1. Range Query: sum/min/max over a submatrix
2. Point Update: modify one element and reflect the change

Naive approach  $\rightarrow O(n^2)$  per query 2D Segment Tree  $\rightarrow O(\log^2 n)$  per query and update

## How It Works (Plain Language)

Think of a 2D segment tree as a segment tree of segment trees.

1. Outer tree partitions rows.
2. Each node of the outer tree holds an inner segment tree for its rows.

Each node thus represents a rectangle in the matrix.

At build time:

- Combine children horizontally and vertically using a merge rule (like sum).

At query time:

- Combine answers from nodes overlapping the query rectangle.

## Example

Matrix A ( $3 \times 3$ ):

|     | y=1 | y=2 | y=3 |
|-----|-----|-----|-----|
| x=1 | 2   | 1   | 3   |
| x=2 | 4   | 5   | 6   |
| x=3 | 7   | 8   | 9   |

Query rectangle  $(1,1)$  to  $(2,2) \rightarrow 2 + 1 + 4 + 5 = 12$

## Key Idea

For each row range, build a column segment tree. For each parent node (row range), merge children column trees:

```
tree[x][y] = merge(tree[2*x][y], tree[2*x+1][y])
```

## Example Walkthrough (Sum Tree)

1. Build row segment tree
2. Inside each row node, build column segment tree
3. To query rectangle  $(x_1, y_1)$  to  $(x_2, y_2)$ :
  - Query over x-range  $\rightarrow$  merge vertical results
  - Inside each x-node, query y-range  $\rightarrow$  merge horizontal results

## Tiny Code (Easy Version)

Python (Sum 2D Segment Tree)

```
class SegmentTree2D:  
    def __init__(self, mat):  
        self.n = len(mat)  
        self.m = len(mat[0])  
        self.tree = [[0]*(4*self.m) for _ in range(4*self.n)]  
        self.mat = mat  
        self.build_x(1, 0, self.n-1)  
  
    def merge(self, a, b):  
        return a + b # sum merge  
  
    # build column tree for a fixed row range  
    def build_y(self, nodex, lx, rx, nodey, ly, ry):  
        if ly == ry:  
            if lx == rx:  
                self.tree[nodex][nodey] = self.mat[lx][ly]  
            else:  
                self.tree[nodex][nodey] = self.merge(  
                    self.tree[2*nodex][nodey], self.tree[2*nodex+1][nodey]  
                )
```

```

        return
    midy = (ly + ry)//2
    self.build_y(nodex, lx, rx, 2*nodey, ly, midy)
    self.build_y(nodex, lx, rx, 2*nodey+1, midy+1, ry)
    self.tree[nodex][nodey] = self.merge(
        self.tree[nodex][2*nodey], self.tree[nodex][2*nodey+1]
    )

# build row tree
def build_x(self, nodex, lx, rx):
    if lx != rx:
        midx = (lx + rx)//2
        self.build_x(2*nodex, lx, midx)
        self.build_x(2*nodex+1, midx+1, rx)
    self.build_y(nodex, lx, rx, 1, 0, self.m-1)

def query_y(self, nodex, nodey, ly, ry, qly, qry):
    if qry < ly or qly > ry: return 0
    if qly <= ly and ry <= qry:
        return self.tree[nodex][nodey]
    midy = (ly + ry)//2
    return self.merge(
        self.query_y(nodex, 2*nodey, ly, midy, qly, qry),
        self.query_y(nodex, 2*nodey+1, midy+1, ry, qly, qry)
    )

def query_x(self, nodex, lx, rx, qlx, qrx, qly, qry):
    if qrx < lx or qlx > rx: return 0
    if qlx <= lx and rx <= qrx:
        return self.query_y(nodex, 1, 0, self.m-1, qly, qry)
    midx = (lx + rx)//2
    return self.merge(
        self.query_x(2*nodex, lx, midx, qlx, qrx, qly, qry),
        self.query_x(2*nodex+1, midx+1, rx, qlx, qrx, qly, qry)
    )

def query(self, x1, y1, x2, y2):
    return self.query_x(1, 0, self.n-1, x1, x2, y1, y2)

# Example
A = [
    [2, 1, 3],

```

```

[4, 5, 6],
[7, 8, 9]
$$
seg2d = SegmentTree2D(A)
print(seg2d.query(0, 0, 1, 1)) # expect 12

```

## Why It Matters

- Supports 2D queries with  $\log^2$  complexity
- Works for sum, min, max, gcd, XOR
- A foundation for advanced 2D data structures (Fenwick 2D, KD-trees, etc.)

## A Gentle Proof (Why It Works)

Segment tree correctness relies on associativity. In 2D, we extend this property across both dimensions. Each node represents a rectangular region; merging children yields correct aggregate for parent.

## Try It Yourself

1. Build from  $3 \times 3$  matrix
2. Query  $(0,0)-(2,2) \rightarrow$  full sum
3. Query  $(1,1)-(2,2) \rightarrow$  bottom-right  $5+6+8+9=28$
4. Modify  $A[1][2]=10$ , rebuild, recheck sum

## Test Cases

| Matrix                      | Query         | Result |
|-----------------------------|---------------|--------|
| $[[2,1,3],[4,5,6],[7,8,9]]$ | $(0,0)-(1,1)$ | 12     |
| $[[2,1,3],[4,5,6],[7,8,9]]$ | $(1,1)-(2,2)$ | 28     |
| $[[1,2],[3,4]]$             | $(0,0)-(1,1)$ | 10     |

| Operation | Time | Space |
|-----------|------|-------|
|-----------|------|-------|

## Complexity

| Operation | Time                                     | Space          |
|-----------|------------------------------------------|----------------|
| Build     | $O(n \cdot m \cdot \log n \cdot \log m)$ | $O(n \cdot m)$ |
| Query     | $O(\log^2 n)$                            | $O(1)$         |
| Update    | $O(\log^2 n)$                            | $O(1)$         |

A 2D segment tree is your grid-wise superpower, merging rectangles like puzzle pieces, one log at a time.

## Section 26. Disjoint Set Union

### 251 Make-Set

The Make-Set operation is the starting point of the Disjoint Set Union (DSU), also known as Union-Find, a fundamental data structure for managing partitions of elements into disjoint sets.

Each element begins in its own set, serving as its own parent. Later operations (**Find** and **Union**) will merge and track relationships among them efficiently.

### What Problem Are We Solving?

We need a way to represent a collection of disjoint sets, groups that don't overlap, while supporting these operations:

1. Make-Set(x): create a new set containing only x
2. Find(x): find representative (leader) of x's set
3. Union(x, y): merge sets containing x and y

This structure is the backbone of many graph algorithms like Kruskal's MST, connected components, and clustering.

## How It Works (Plain Language)

Initially, each element is its own parent, a self-loop. We store two arrays (or maps):

- `parent[x]` → points to x's parent (initially itself)
- `rank[x]` or `size[x]` → helps balance unions later

So:

```
parent[x] = x
rank[x] = 0
```

Each element is an isolated tree. Over time, unions connect trees.

### Example

Initialize  $n = 5$  elements: {1, 2, 3, 4, 5}

| x | parent[x] | rank[x] | Meaning    |
|---|-----------|---------|------------|
| 1 | 1         | 0       | own leader |
| 2 | 2         | 0       | own leader |
| 3 | 3         | 0       | own leader |
| 4 | 4         | 0       | own leader |
| 5 | 5         | 0       | own leader |

After all Make-Set, each element is in its own group:

```
{1}, {2}, {3}, {4}, {5}
```

### Visualization

```
1   2   3   4   5
↑   ↑   ↑   ↑   ↑
|   |   |   |   |
self self self self self
```

Each node points to itself, five separate trees.

## Tiny Code (Easy Versions)

C Implementation

```
#include <stdio.h>

#define MAXN 1000

int parent[MAXN];
int rank_[MAXN];

void make_set(int v) {
    parent[v] = v;      // self parent
    rank_[v] = 0;        // initial rank
}

int main(void) {
    int n = 5;
    for (int i = 1; i <= n; i++)
        make_set(i);

    printf("Initial sets:\n");
    for (int i = 1; i <= n; i++)
        printf("Element %d: parent=%d, rank=%d\n", i, parent[i], rank_[i]);
    return 0;
}
```

Python Implementation

```
def make_set(x, parent, rank):
    parent[x] = x
    rank[x] = 0

n = 5
parent = {}
rank = {}

for i in range(1, n+1):
    make_set(i, parent, rank)

print("Parent:", parent)
print("Rank:", rank)
# Parent: {1:1, 2:2, 3:3, 4:4, 5:5}
```

## Why It Matters

- The foundation of Disjoint Set Union
- Enables efficient graph algorithms
- A building block for Union by Rank and Path Compression
- Every DSU begins with **Make-Set**

## A Gentle Proof (Why It Works)

Each element begins as a singleton. Since no pointers cross between elements, sets are disjoint. Subsequent **Union** operations preserve this property by merging trees, never duplicating nodes.

Thus, **Make-Set** guarantees:

- Each new node is independent
- Parent pointers form valid forests

## Try It Yourself

1. Initialize  $\{1..5\}$  with **Make-Set**
2. Print parent and rank arrays
3. Add **Union(1,2)** and check parent changes
4. Verify all  $\text{parent}[x] = x$  before unions

## Test Cases

| Input | Operation             | Expected Output             |
|-------|-----------------------|-----------------------------|
| $n=3$ | <b>Make-Set(1..3)</b> | $\text{parent} = [1,2,3]$   |
| $n=5$ | <b>Make-Set(1..5)</b> | $\text{rank} = [0,0,0,0,0]$ |

## Complexity

| Operation | Time                         | Space  |
|-----------|------------------------------|--------|
| Make-Set  | $O(1)$                       | $O(1)$ |
| Find      | $O(\log n)$ with compression | $O(1)$ |
| Union     | $O(n)$ with rank             | $O(1)$ |

The **Make-Set** step is your first move in the DSU dance, simple, constant time, and crucial for what follows.

## 252 Find

The Find operation is the heart of the Disjoint Set Union (DSU), also known as Union-Find. It locates the representative (leader) of the set containing a given element. Every element in the same set shares the same leader, this is how DSU identifies which elements belong together.

To make lookups efficient, **Find** uses a clever optimization called Path Compression, which flattens the structure of the tree so future queries become nearly constant time.

### What Problem Are We Solving?

Given an element  $x$ , we want to determine which set it belongs to. Each set is represented by a root node (the leader).

We maintain a `parent[]` array such that:

- `parent[x] = x` if  $x$  is the root (leader)
- otherwise `parent[x] = parent of x`

The `Find(x)` operation recursively follows `parent[x]` pointers until it reaches the root.

### How It Works (Plain Language)

Think of each set as a tree, where the root is the representative.

For example:

$1 \leftarrow 2 \leftarrow 3 \quad 4 \leftarrow 5$

means  $\{1, 2, 3\}$  is one set,  $\{4, 5\}$  is another. The `Find(3)` operation follows  $3 \rightarrow 2 \rightarrow 1$ , discovering 1 is the root.

Path Compression flattens the tree by pointing every node directly to the root, reducing depth and speeding up future finds.

After compression:

$1 \leftarrow 2 \quad 1 \leftarrow 3$   
 $4 \leftarrow 5$

Now `Find(3)` is  $O(1)$ .

## Example

Start:

| x | parent[x] | rank[x] |
|---|-----------|---------|
| 1 | 1         | 1       |
| 2 | 1         | 0       |
| 3 | 2         | 0       |
| 4 | 4         | 0       |
| 5 | 4         | 0       |

Perform `Find(3)`:

1.  $3 \rightarrow \text{parent}[3] = 2$
2.  $2 \rightarrow \text{parent}[2] = 1$
3.  $1 \rightarrow \text{root found}$

Path compression rewrites `parent[3] = 1`

Result:

| x | parent[x] |
|---|-----------|
| 1 | 1         |
| 2 | 1         |
| 3 | 1         |
| 4 | 4         |
| 5 | 4         |

## Visualization

Before compression:

```
1
↑
2
↑
3
```

After compression:

```
1
↑ ↑
2 3
```

Now all nodes point directly to root 1.

### Tiny Code (Easy Versions)

C Implementation

```
#include <stdio.h>

#define MAXN 1000
int parent[MAXN];
int rank_[MAXN];

void make_set(int v) {
    parent[v] = v;
    rank_[v] = 0;
}

int find_set(int v) {
    if (v == parent[v])
        return v;
    // Path compression
    parent[v] = find_set(parent[v]);
    return parent[v];
}

int main(void) {
    int n = 5;
    for (int i = 1; i <= n; i++) make_set(i);
    parent[2] = 1;
    parent[3] = 2;
    printf("Find(3) before: %d\n", parent[3]);
    printf("Root of 3: %d\n", find_set(3));
    printf("Find(3) after: %d\n", parent[3]);
}
```

Python Implementation

```

def make_set(x, parent, rank):
    parent[x] = x
    rank[x] = 0

def find_set(x, parent):
    if parent[x] != x:
        parent[x] = find_set(parent[x], parent) # Path compression
    return parent[x]

n = 5
parent = {}
rank = {}
for i in range(1, n+1):
    make_set(i, parent, rank)

parent[2] = 1
parent[3] = 2

print("Find(3) before:", parent)
print("Root of 3:", find_set(3, parent))
print("Find(3) after:", parent)

```

## Why It Matters

- Enables  $O(n)$  almost-constant-time queries
- Fundamental for Union-Find efficiency
- Reduces tree height dramatically
- Used in graph algorithms (Kruskal, connected components, etc.)

## A Gentle Proof (Why It Works)

Every set is a rooted tree. Without compression, a sequence of unions could build a deep chain. With path compression, each `Find` flattens paths, ensuring amortized constant time per operation (Ackermann-inverse time).

So after repeated operations, DSU trees become very shallow.

## Try It Yourself

1. Initialize  $\{1..5\}$

2. Set `parent[2]=1`, `parent[3]=2`
3. Call `Find(3)` and observe compression
4. Check `parent[3] == 1` after

## Test Cases

| parent before | Find(x) | parent after |
|---------------|---------|--------------|
| [1,1,2]       | Find(3) | [1,1,1]      |
| [1,2,3,4,5]   | Find(5) | [1,2,3,4,5]  |
| [1,1,1,1,1]   | Find(3) | [1,1,1,1,1]  |

## Complexity

| Operation | Time (Amortized) | Space  |
|-----------|------------------|--------|
| Find      | $O(n)$           | $O(1)$ |
| Make-Set  | $O(1)$           | $O(1)$ |
| Union     | $O(n)$           | $O(1)$ |

The `Find` operation is your compass, it always leads you to the leader of your set, and with path compression, it gets faster every step.

## 253 Union

The `Union` operation is what ties the Disjoint Set Union (DSU) together. After initializing sets with `Make-Set` and locating leaders with `Find`, `Union` merges two disjoint sets into one, a cornerstone of dynamic connectivity.

To keep the trees shallow and efficient, we combine it with heuristics like Union by Rank or Union by Size.

## What Problem Are We Solving?

We want to merge the sets containing two elements `a` and `b`.

If `a` and `b` belong to different sets, their leaders (`Find(a)` and `Find(b)`) are different. The `Union` operation connects these leaders, ensuring both now share the same representative.

We must do it efficiently, no unnecessary tree height. That's where Union by Rank comes in.

## How It Works (Plain Language)

1. Find roots of both elements:

```
rootA = Find(a)  
rootB = Find(b)
```

2. If they're already equal → same set, no action.
3. Otherwise, attach the shorter tree under the taller one:

- If  $\text{rank}[\text{rootA}] < \text{rank}[\text{rootB}]$ :  $\text{parent}[\text{rootA}] = \text{rootB}$
- Else if  $\text{rank}[\text{rootA}] > \text{rank}[\text{rootB}]$ :  $\text{parent}[\text{rootB}] = \text{rootA}$
- Else:  $\text{parent}[\text{rootB}] = \text{rootA}$ , and  $\text{rank}[\text{rootA}]++$

This keeps the resulting forest balanced, maintaining  $O(n)$  time for operations.

## Example

Initial sets: {1}, {2}, {3}, {4}

| x | parent[x] | rank[x] |
|---|-----------|---------|
| 1 | 1         | 0       |
| 2 | 2         | 0       |
| 3 | 3         | 0       |
| 4 | 4         | 0       |

Perform `Union(1,2)` → attach 2 under 1

```
parent[2] = 1  
rank[1] = 1
```

Now sets: {1,2}, {3}, {4}

Perform `Union(3,4)` → attach 4 under 3 Sets: {1,2}, {3,4}

Then `Union(2,3)` → merge leaders (1 and 3) Attach lower rank under higher (both rank 1 → tie) → attach 3 under 1,  $\text{rank}[1] = 2$

Final parent table:

| x | parent[x] | rank[x] |
|---|-----------|---------|
| 1 | 1         | 2       |
| 2 | 1         | 0       |
| 3 | 1         | 1       |
| 4 | 3         | 0       |

Now all are connected under root 1.

### Visualization

Start:

1    2    3    4

Union(1,2):

```

1
|
2

```

Union(3,4):

```

3
|
4

```

Union(2,3):

```

      1
     / | \
    2   3   4

```

All connected under 1.

### Tiny Code (Easy Versions)

C Implementation

```

#include <stdio.h>

#define MAXN 1000
int parent[MAXN];
int rank_[MAXN];

void make_set(int v) {
    parent[v] = v;
    rank_[v] = 0;
}

int find_set(int v) {
    if (v == parent[v]) return v;
    return parent[v] = find_set(parent[v]); // path compression
}

void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (rank_[a] < rank_[b])
            parent[a] = b;
        else if (rank_[a] > rank_[b])
            parent[b] = a;
        else {
            parent[b] = a;
            rank_[a]++;
        }
    }
}

int main(void) {
    for (int i=1;i<=4;i++) make_set(i);
    union_sets(1,2);
    union_sets(3,4);
    union_sets(2,3);
    for (int i=1;i<=4;i++)
        printf("Element %d: parent=%d, rank=%d\n", i, parent[i], rank_[i]);
}

```

Python Implementation

```

def make_set(x, parent, rank):
    parent[x] = x
    rank[x] = 0

def find_set(x, parent):
    if parent[x] != x:
        parent[x] = find_set(parent[x], parent)
    return parent[x]

def union_sets(a, b, parent, rank):
    a = find_set(a, parent)
    b = find_set(b, parent)
    if a != b:
        if rank[a] < rank[b]:
            parent[a] = b
        elif rank[a] > rank[b]:
            parent[b] = a
        else:
            parent[b] = a
            rank[a] += 1

n = 4
parent = {}
rank = {}
for i in range(1, n+1): make_set(i, parent, rank)
union_sets(1,2,parent,rank)
union_sets(3,4,parent,rank)
union_sets(2,3,parent,rank)
print("Parent:", parent)
print("Rank:", rank)

```

## Why It Matters

- Core operation in Union-Find
- Enables dynamic set merging efficiently
- Essential for graph algorithms:
  - Kruskal's Minimum Spanning Tree
  - Connected Components
  - Cycle Detection

Combining Union by Rank and Path Compression yields near-constant performance for huge datasets.

### A Gentle Proof (Why It Works)

Union ensures disjointness:

- Only merges sets with different roots
- Maintains one leader per set Union by Rank keeps trees balanced → tree height  $\log n$   
With path compression, amortized cost per operation is  $O(n)$ , effectively constant for all practical  $n$ .

### Try It Yourself

1. Create 5 singleton sets
2. Union(1,2), Union(3,4), Union(2,3)
3. Verify all share same root
4. Inspect ranks before/after unions

### Test Cases

| Operations     | Expected Sets   | Roots     |
|----------------|-----------------|-----------|
| Make-Set(1..4) | {1},{2},{3},{4} | [1,2,3,4] |
| Union(1,2)     | {1,2}           | [1,1,3,4] |
| Union(3,4)     | {3,4}           | [1,1,3,3] |
| Union(2,3)     | {1,2,3,4}       | [1,1,1,1] |

### Complexity

| Operation | Time (Amortized) | Space  |
|-----------|------------------|--------|
| Union     | $O(n)$           | $O(1)$ |
| Find      | $O(n)$           | $O(1)$ |
| Make-Set  | $O(1)$           | $O(1)$ |

Union is the handshake between sets, careful, balanced, and lightning-fast when combined with smart optimizations.

## 254 Union by Rank

Union by Rank is a balancing strategy used in the Disjoint Set Union (DSU) structure to keep trees shallow. When merging two sets, instead of arbitrarily attaching one root under another, we attach the shorter tree (lower rank) under the taller tree (higher rank).

This small trick, combined with Path Compression, gives DSU its legendary near-constant performance, almost O(1) per operation.

### What Problem Are We Solving?

Without balancing, repeated `Union` operations might create tall chains like:

1 ← 2 ← 3 ← 4 ← 5

In the worst case, `Find(x)` becomes O(n).

Union by Rank prevents this by maintaining a rough measure of tree height. Each `rank[root]` tracks the *approximate height* of its tree.

When merging:

- Attach the lower rank tree under the higher rank tree.
- If ranks are equal, choose one as parent and increase its rank by 1.

### How It Works (Plain Language)

Each node `x` has:

- `parent[x]` → leader pointer
- `rank[x]` → estimate of tree height

When performing `Union(a, b)`:

1. `rootA = Find(a)`
2. `rootB = Find(b)`
3. If ranks differ, attach smaller under larger:

```

if rank[rootA] < rank[rootB]:
    parent[rootA] = rootB
else if rank[rootA] > rank[rootB]:
    parent[rootB] = rootA
else:
    parent[rootB] = rootA
    rank[rootA] += 1

```

This ensures the tree grows only when necessary.

### Example

Start with {1}, {2}, {3}, {4} All have `rank = 0`

Perform `Union(1,2)` → same rank → attach 2 under 1, increment `rank[1]=1`

```

1
|
2

```

Perform `Union(3,4)` → same rank → attach 4 under 3, increment `rank[3]=1`

```

3
|
4

```

Now `Union(1,3)` → both roots `rank=1` → tie → attach 3 under 1, `rank[1]=2`

```

1(rank=2)
 / \
2   3
 |
4

```

Tree height stays small, well-balanced.

| Element | Parent | Rank |
|---------|--------|------|
| 1       | 1      | 2    |
| 2       | 1      | 0    |
| 3       | 1      | 1    |

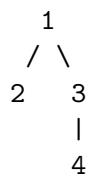
| Element | Parent | Rank |
|---------|--------|------|
| 4       | 3      | 0    |

## Visualization

Before balancing:

1 ← 2 ← 3 ← 4

After union by rank:



Balanced and efficient

## Tiny Code (Easy Versions)

C Implementation

```
#include <stdio.h>

#define MAXN 1000
int parent[MAXN];
int rank_[MAXN];

void make_set(int v) {
    parent[v] = v;
    rank_[v] = 0;
}

int find_set(int v) {
    if (v == parent[v]) return v;
    return parent[v] = find_set(parent[v]); // Path compression
}
```

```

void union_by_rank(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (rank_[a] < rank_[b]) parent[a] = b;
        else if (rank_[a] > rank_[b]) parent[b] = a;
        else {
            parent[b] = a;
            rank_[a]++;
        }
    }
}

int main(void) {
    for (int i=1;i<=4;i++) make_set(i);
    union_by_rank(1,2);
    union_by_rank(3,4);
    union_by_rank(2,3);
    for (int i=1;i<=4;i++)
        printf("Element %d: parent=%d rank=%d\n", i, parent[i], rank_[i]);
}

```

### Python Implementation

```

def make_set(x, parent, rank):
    parent[x] = x
    rank[x] = 0

def find_set(x, parent):
    if parent[x] != x:
        parent[x] = find_set(parent[x], parent)
    return parent[x]

def union_by_rank(a, b, parent, rank):
    a = find_set(a, parent)
    b = find_set(b, parent)
    if a != b:
        if rank[a] < rank[b]:
            parent[a] = b
        elif rank[a] > rank[b]:
            parent[b] = a
        else:

```

```

parent[b] = a
rank[a] += 1

n = 4
parent = {}
rank = {}
for i in range(1, n+1): make_set(i, parent, rank)
union_by_rank(1,2,parent,rank)
union_by_rank(3,4,parent,rank)
union_by_rank(2,3,parent,rank)
print("Parent:", parent)
print("Rank:", rank)

```

## Why It Matters

- Keeps trees balanced and shallow
- Ensures amortized  $O(\log n)$  performance
- Critical for massive-scale connectivity problems
- Used in Kruskal's MST, Union-Find with rollback, and network clustering

## A Gentle Proof (Why It Works)

The rank grows only when two trees of equal height merge. Thus, the height of any tree is bounded by  $\log n$ . Combining this with Path Compression, the amortized complexity per operation becomes  $O(\log n)$ , where  $(n)$  (inverse Ackermann function)  $< 5$  for all practical  $n$ .

## Try It Yourself

1. Create 8 singleton sets
2. Perform unions in sequence: (1,2), (3,4), (1,3), (5,6), (7,8), (5,7), (1,5)
3. Observe ranks and parent structure

## Test Cases

| Operation  | Result (Parent Array) | Rank      |
|------------|-----------------------|-----------|
| Union(1,2) | [1,1,3,4]             | [1,0,0,0] |
| Union(3,4) | [1,1,3,3]             | [1,0,1,0] |
| Union(2,3) | [1,1,1,3]             | [2,0,1,0] |

## Complexity

| Operation     | Time (Amortized) | Space  |
|---------------|------------------|--------|
| Union by Rank | $O(\log n)$      | $O(1)$ |
| Find          | $O(\log n)$      | $O(1)$ |
| Make-Set      | $O(1)$           | $O(1)$ |

Union by Rank is the art of merging gracefully, always lifting smaller trees, keeping your forest light, flat, and fast.

## 255 Path Compression

Path Compression is the secret sauce that makes Disjoint Set Union (DSU) lightning fast. Every time you perform a `Find` operation, it *flattens* the structure by making each visited node point directly to the root. Over time, this transforms deep trees into almost flat structures, turning expensive lookups into near-constant time.

### What Problem Are We Solving?

In a basic DSU, `Find(x)` follows parent pointers up the tree until it reaches the root. Without compression, frequent unions can form long chains:

1 ← 2 ← 3 ← 4 ← 5

A `Find(5)` would take 5 steps. Multiply this over many queries, and performance tanks.

Path Compression fixes this inefficiency by *rewiring* all nodes on the search path to the root directly, effectively flattening the tree.

### How It Works (Plain Language)

Whenever we call `Find(x)`, we recursively find the root, then make every node along the way point directly to that root.

Pseudocode:

```
Find(x):
    if parent[x] != x:
        parent[x] = Find(parent[x])
    return parent[x]
```

Now, future lookups for  $x$  and its descendants become instant.

### Example

Start with a chain:

$1 \leftarrow 2 \leftarrow 3 \leftarrow 4 \leftarrow 5$

Perform  $\text{Find}(5)$ :

1.  $\text{Find}(5)$  calls  $\text{Find}(4)$
2.  $\text{Find}(4)$  calls  $\text{Find}(3)$
3.  $\text{Find}(3)$  calls  $\text{Find}(2)$
4.  $\text{Find}(2)$  calls  $\text{Find}(1)$  (root)
5. On the way back, each node gets updated:

```
parent[5] = 1  
parent[4] = 1  
parent[3] = 1  
parent[2] = 1
```

After compression, structure becomes flat:

1  
2  
3  
4  
5

| Element | Parent Before | Parent After |
|---------|---------------|--------------|
| 1       | 1             | 1            |
| 2       | 1             | 1            |
| 3       | 2             | 1            |
| 4       | 3             | 1            |
| 5       | 4             | 1            |

Next call  $\text{Find}(5) \rightarrow$  single step.

## Visualization

Before compression:

1 ← 2 ← 3 ← 4 ← 5

After compression:

1  
2  
3  
4  
5

## Tiny Code (Easy Versions)

C Implementation

```
#include <stdio.h>

#define MAXN 100
int parent[MAXN];

void make_set(int v) {
    parent[v] = v;
}

int find_set(int v) {
    if (v != parent[v])
        parent[v] = find_set(parent[v]); // Path compression
    return parent[v];
}

void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b)
        parent[b] = a;
}

int main(void) {
```

```

for (int i = 1; i <= 5; i++) make_set(i);
union_sets(1, 2);
union_sets(2, 3);
union_sets(3, 4);
union_sets(4, 5);
find_set(5); // compress path
for (int i = 1; i <= 5; i++)
    printf("Element %d → Parent %d\n", i, parent[i]);
}

```

Python Implementation

```

def make_set(x, parent):
    parent[x] = x

def find_set(x, parent):
    if parent[x] != x:
        parent[x] = find_set(parent[x], parent)
    return parent[x]

def union_sets(a, b, parent):
    a = find_set(a, parent)
    b = find_set(b, parent)
    if a != b:
        parent[b] = a

parent = {}
for i in range(1, 6):
    make_set(i, parent)
union_sets(1, 2, parent)
union_sets(2, 3, parent)
union_sets(3, 4, parent)
union_sets(4, 5, parent)
find_set(5, parent)
print("Parent map after compression:", parent)

```

## Why It Matters

- Speeds up Find drastically by flattening trees.
- Pairs beautifully with Union by Rank.
- Achieves amortized  $O(n)$  performance.

- Essential in graph algorithms like Kruskal's MST, connectivity checks, and dynamic clustering.

### A Gentle Proof (Why It Works)

Path Compression ensures each node's parent jumps directly to the root. Each node's depth decreases exponentially with every `Find`. After a few operations, trees become almost flat, and each subsequent `Find` becomes  $O(1)$ .

Combining with Union by Rank:

Every operation (Find or Union) becomes  $O(\alpha(n))$ , where  $\alpha(n)$  is the inverse Ackermann function (smaller than 5 for any practical input).

### Try It Yourself

1. Create 6 singleton sets.
2. Perform unions: (1,2), (2,3), (3,4), (4,5), (5,6).
3. Call `Find(6)` and print parent map before and after.
4. Observe how the chain flattens.
5. Measure calls count difference before and after compression.

### Test Cases

| Operation Sequence | Parent Map After          | Tree Depth |
|--------------------|---------------------------|------------|
| No Compression     | {1:1, 2:1, 3:2, 4:3, 5:4} | 4          |
| After Find(5)      | {1:1, 2:1, 3:1, 4:1, 5:1} | 1          |
| After Find(3)      | {1:1, 2:1, 3:1, 4:1, 5:1} | 1          |

### Complexity

| Operation                  | Time (Amortized) | Space  |
|----------------------------|------------------|--------|
| Find with Path Compression | $O(\alpha(n))$   | $O(1)$ |
| Union (with rank)          | $O(\alpha(n))$   | $O(1)$ |
| Make-Set                   | $O(1)$           | $O(1)$ |

Path Compression is the flattening spell of DSU, once cast, your sets become sleek, your lookups swift, and your unions unstoppable.

## 256 DSU with Rollback

DSU with Rollback extends the classic Disjoint Set Union to support *undoing* recent operations. This is vital in scenarios where you need to explore multiple states, like backtracking algorithms, dynamic connectivity queries, or offline problems where unions might need to be reversed.

Instead of destroying past states, this version *remembers* what changed and can roll back to a previous version in constant time.

### What Problem Are We Solving?

Standard DSU operations (**Find**, **Union**) mutate the structure, parent pointers and ranks get updated, so you can't easily go back.

But what if you're exploring a search tree or processing offline queries where you need to:

- Add an edge (Union),
- Explore a path,
- Then revert to the previous structure?

A rollback DSU lets you undo changes, perfect for divide-and-conquer over time, Mo's algorithm on trees, and offline dynamic graphs.

### How It Works (Plain Language)

The idea is simple:

- Every time you modify the DSU, record what you changed on a stack.
- When you need to revert, pop from the stack and undo the last operation.

Operations to track:

- When `parent[b]` changes, store `(b, old_parent)`
- When `rank[a]` changes, store `(a, old_rank)`

You never perform path compression, because it's not easily reversible. Instead, rely on union by rank for efficiency.

## Example

Let's build sets {1}, {2}, {3}, {4}

Perform:

1. Union(1,2) → attach 2 under 1
  - Push (2, parent=2, rank=None)
2. Union(3,4) → attach 4 under 3
  - Push (4, parent=4, rank=None)
3. Union(1,3) → attach 3 under 1
  - Push (3, parent=3, rank=None)
  - Rank of 1 increases → push (1, rank=0)

Rollback once → undo last union:

- Restore parent[3] = 3
- Restore rank[1] = 0

Now DSU returns to state after step 2.

| Step             | Map       | Rank      | Stack                                         |
|------------------|-----------|-----------|-----------------------------------------------|
| Init             | [1,2,3,4] | [0,0,0,0] | []                                            |
| After Union(1,2) | [1,1,3,4] | [1,0,0,0] | [(2,2,None)]                                  |
| After Union(3,4) | [1,1,3,3] | [1,0,1,0] | [(2,2,None),(4,4,None)]                       |
| After Union(1,3) | [1,1,1,3] | [2,0,1,0] | [(2,2,None),(4,4,None),(3,3,None),(1,None,0)] |
| After Rollback   | [1,1,3,3] | [1,0,1,0] | [(2,2,None),(4,4,None)]                       |

## Tiny Code (Easy Versions)

C Implementation (Conceptual)

```
#include <stdio.h>

#define MAXN 1000
int parent[MAXN], rank_[MAXN];
typedef struct { int node, parent, rank_val, rank_changed; } Change;
Change stack[MAXN * 10];
```

```

int top = 0;

void make_set(int v) {
    parent[v] = v;
    rank_[v] = 0;
}

int find_set(int v) {
    while (v != parent[v]) v = parent[v];
    return v; // no path compression
}

void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a == b) return;
    if (rank_[a] < rank_[b]) { int tmp = a; a = b; b = tmp; }
    stack[top++] = (Change){b, parent[b], 0, 0};
    parent[b] = a;
    if (rank_[a] == rank_[b]) {
        stack[top++] = (Change){a, 0, rank_[a], 1};
        rank_[a]++;
    }
}

void rollback() {
    if (top == 0) return;
    Change ch = stack[--top];
    if (ch.rank_changed) rank_[ch.node] = ch.rank_val;
    else parent[ch.node] = ch.parent;
}

int main() {
    for (int i=1;i<=4;i++) make_set(i);
    union_sets(1,2);
    union_sets(3,4);
    union_sets(1,3);
    printf("Before rollback: parent[3]=%d\n", parent[3]);
    rollback();
    printf("After rollback: parent[3]=%d\n", parent[3]);
}

```

## Python Implementation

```
class RollbackDSU:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0]*n
        self.stack = []

    def find(self, x):
        while x != self.parent[x]:
            x = self.parent[x]
        return x # no path compression

    def union(self, a, b):
        a, b = self.find(a), self.find(b)
        if a == b:
            return False
        if self.rank[a] < self.rank[b]:
            a, b = b, a
        self.stack.append(('p', b, self.parent[b]))
        self.parent[b] = a
        if self.rank[a] == self.rank[b]:
            self.stack.append(('r', a, self.rank[a]))
            self.rank[a] += 1
        return True

    def rollback(self):
        if not self.stack:
            return
        typ, node, val = self.stack.pop()
        if typ == 'r':
            self.rank[node] = val
        else:
            self.parent[node] = val

dsu = RollbackDSU(5)
dsu.union(1,2)
dsu.union(3,4)
dsu.union(1,3)
print("Before rollback:", dsu.parent)
dsu.rollback()
print("After rollback:", dsu.parent)
```

## Why It Matters

- Enables reversible union operations
- Perfect for offline dynamic connectivity
- Core in divide-and-conquer over time algorithms
- Used in Mo's algorithm on trees
- Helps in exploration backtracking (e.g., DSU on recursion)

## A Gentle Proof (Why It Works)

Rollback DSU is efficient because:

- Each union modifies  $O(1)$  fields
- Each rollback reverts  $O(1)$  fields
- Find runs in  $O(\log n)$  (no path compression) Thus, each operation is  $O(\log n)$  or better, fully reversible.

## Try It Yourself

1. Create 6 sets  $\{1\} \dots \{6\}$
2. Perform unions:  $(1,2)$ ,  $(3,4)$ ,  $(2,3)$
3. Rollback once, verify sets  $\{1,2\}$  and  $\{3,4\}$  remain separate
4. Rollback again, check individual sets restored
5. Print parent and rank at each step

## Test Cases

| Step | Operation  | Parents   | Ranks     | Stack Size |
|------|------------|-----------|-----------|------------|
| 1    | Union(1,2) | [1,1,3,4] | [1,0,0,0] | 1          |
| 2    | Union(3,4) | [1,1,3,3] | [1,0,1,0] | 2          |
| 3    | Union(1,3) | [1,1,1,3] | [2,0,1,0] | 4          |
| 4    | Rollback   | [1,1,3,3] | [1,0,1,0] | 2          |

## Complexity

|          | Operation   | Time   | Space |
|----------|-------------|--------|-------|
| Make-Set | $O(1)$      | $O(n)$ |       |
| Union    | $O(\log n)$ | $O(1)$ |       |

| Operation | Time   | Space  |
|-----------|--------|--------|
| Rollback  | $O(1)$ | $O(1)$ |

DSU with Rollback is your time machine, merge, explore, undo. Perfect balance between persistence and performance.

## 257 DSU on Tree

DSU on Tree is a hybrid technique combining Disjoint Set Union (DSU) and Depth-First Search (DFS) to process subtree queries efficiently. It's often called the "Small-to-Large" merging technique, and it shines in problems where you need to answer queries like:

"For each node, count something inside its subtree."

Instead of recomputing from scratch at every node, we use DSU logic to reuse computed data, merging smaller subtrees into larger ones for near-linear complexity.

### What Problem Are We Solving?

Many tree problems ask for aggregated properties of subtrees:

- Number of distinct colors in a subtree
- Frequency of labels, values, or weights
- Subtree sums, counts, or modes

A naive DFS recomputes at every node  $\rightarrow O(n^2)$  time.

DSU on Tree avoids recomputation by merging information from child subtrees cleverly.

### How It Works (Plain Language)

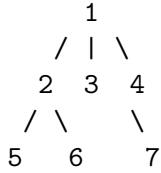
Think of each node's subtree as a bag of information (like a multiset of colors). We process subtrees in DFS order, and at each step:

1. Process all *small children* first and discard their data after use.
2. Process the *heavy child* last and keep its data (reuse it).
3. Merge small subtrees into the large one incrementally.

This "small-to-large" merging ensures each element moves  $O(\log n)$  times, leading to  $O(n \log n)$  total complexity.

## Example

Suppose we have a tree:



Each node has a color:

```
Color = [1, 2, 2, 1, 3, 3, 2]
```

Goal: For every node, count distinct colors in its subtree.

Naive way: recompute every subtree from scratch,  $O(n^2)$ . DSU-on-Tree way: reuse large child's color set, merge small ones.

| Node | Subtree         | Colors  | Distinct Count |
|------|-----------------|---------|----------------|
| 5    | [5]             | {3}     | 1              |
| 6    | [6]             | {3}     | 1              |
| 2    | [2,5,6]         | {2,3}   | 2              |
| 3    | [3]             | {2}     | 1              |
| 7    | [7]             | {2}     | 1              |
| 4    | [4,7]           | {1,2}   | 2              |
| 1    | [1,2,3,4,5,6,7] | {1,2,3} | 3              |

Each subtree merges small color sets into the big one only once  $\rightarrow$  efficient.

## Step-by-Step Idea

1. DFS to compute subtree sizes
2. Identify heavy child (largest subtree)
3. DFS again:
  - Process all *light children* (small subtrees), discarding results
  - Process *heavy child*, keep its results
  - Merge all light children's data into heavy child's data
4. Record answers for each node after merging

## Tiny Code (Easy Versions)

C Implementation (Conceptual)

```
#include <stdio.h>
#include <vector>
#include <set>

#define MAXN 100005
using namespace std;

vector<int> tree[MAXN];
int color[MAXN];
int subtree_size[MAXN];
int answer[MAXN];
int freq[MAXN];
int n;

void dfs_size(int u, int p) {
    subtree_size[u] = 1;
    for (int v : tree[u])
        if (v != p) {
            dfs_size(v, u);
            subtree_size[u] += subtree_size[v];
        }
}

void add_color(int u, int p, int val) {
    freq[color[u]] += val;
    for (int v : tree[u])
        if (v != p) add_color(v, u, val);
}

void dfs(int u, int p, bool keep) {
    int bigChild = -1, maxSize = -1;
    for (int v : tree[u])
        if (v != p && subtree_size[v] > maxSize)
            maxSize = subtree_size[v], bigChild = v;

    // Process small children
    for (int v : tree[u])
        if (v != p && v != bigChild)
            dfs(v, u, false);
```

```

// Process big child
if (bigChild != -1) dfs(bigChild, u, true);

// Merge small children's info
for (int v : tree[u])
    if (v != p && v != bigChild)
        add_color(v, u, +1);

freq[color[u]]++;
// Example query: count distinct colors
answer[u] = 0;
for (int i = 1; i <= n; i++)
    if (freq[i] > 0) answer[u]++;

if (!keep) add_color(u, p, -1);
}

int main() {
    n = 7;
    // Build tree, set colors...
    // dfs_size(1,0); dfs(1,0,true);
}

```

Python Implementation (Simplified)

```

from collections import defaultdict

def dfs_size(u, p, tree, size):
    size[u] = 1
    for v in tree[u]:
        if v != p:
            dfs_size(v, u, tree, size)
            size[u] += size[v]

def add_color(u, p, tree, color, freq, val):
    freq[color[u]] += val
    for v in tree[u]:
        if v != p:
            add_color(v, u, tree, color, freq, val)

def dfs(u, p, tree, size, color, freq, ans, keep):

```

```

bigChild, maxSize = -1, -1
for v in tree[u]:
    if v != p and size[v] > maxSize:
        maxSize, bigChild = size[v], v

for v in tree[u]:
    if v != p and v != bigChild:
        dfs(v, u, tree, size, color, freq, ans, False)

if bigChild != -1:
    dfs(bigChild, u, tree, size, color, freq, ans, True)

for v in tree[u]:
    if v != p and v != bigChild:
        add_color(v, u, tree, color, freq, 1)

freq[color[u]] += 1
ans[u] = len([c for c in freq if freq[c] > 0])

if not keep:
    add_color(u, p, tree, color, freq, -1)

# Example usage
n = 7
tree = {1:[2,3,4], 2:[1,5,6], 3:[1], 4:[1,7], 5:[2], 6:[2], 7:[4]}
color = {1:1,2:2,3:2,4:1,5:3,6:3,7:2}
size = {}
ans = {}
freq = defaultdict(int)
dfs_size(1,0,tree,size)
dfs(1,0,tree,size,color,freq,ans,True)
print(ans)

```

## Why It Matters

- Handles subtree queries in  $O(n \log n)$
- Works well with static trees and offline queries
- Reuses computed results for heavy subtrees
- Foundation for Mo's algorithm on trees and color frequency problems

## A Gentle Proof (Why It Works)

Each node's color (or element) is merged only  $O(\log n)$  times:

- Each time, it moves from a smaller set to a larger one.
- Hence total merges =  $O(n \log n)$ .

The `keep` flag ensures we only retain big subtrees, discarding light ones to save memory and time.

## Try It Yourself

1. Assign random colors to a tree of 8 nodes.
2. Use DSU on Tree to count distinct colors per subtree.
3. Compare with brute-force DFS result.
4. Verify both match, but DSU-on-Tree runs faster.

## Test Cases

| Node | Subtree Colors | Distinct Count |
|------|----------------|----------------|
| 5    | {3}            | 1              |
| 6    | {3}            | 1              |
| 2    | {2,3}          | 2              |
| 3    | {2}            | 1              |
| 7    | {2}            | 1              |
| 4    | {1,2}          | 2              |
| 1    | {1,2,3}        | 3              |

## Complexity

| Operation     | Time          | Space  |
|---------------|---------------|--------|
| DFS Traversal | $O(n)$        | $O(n)$ |
| DSU Merge     | $O(n \log n)$ | $O(n)$ |
| Overall       | $O(n \log n)$ | $O(n)$ |

DSU on Tree is your subtree superpower, merge smart, discard light, and conquer queries with elegance.

## 258 Kruskal's MST (Using DSU)

Kruskal's Algorithm is a classic greedy method to build a Minimum Spanning Tree (MST), a subset of edges connecting all vertices with the smallest total weight and no cycles. It relies on Disjoint Set Union (DSU) to efficiently check whether adding an edge would form a cycle.

With Union-Find, Kruskal's MST becomes clean, fast, and conceptually elegant, building the tree edge by edge in sorted order.

### What Problem Are We Solving?

Given a connected weighted graph with  $n$  vertices and  $m$  edges, we want to find a tree that:

- Connects all vertices (spanning)
- Has no cycles (tree)
- Minimizes the total edge weight

A naive approach would test every subset of edges,  $O(2^m)$ .

Kruskal's algorithm uses edge sorting and a Disjoint Set Union (DSU) structure to reduce this to  $O(m \log m)$ .

### How It Works (Plain Language)

The algorithm follows three steps:

1. Sort all edges by weight in ascending order.
2. Initialize each vertex as its own set (**Make-Set**).
3. For each edge  $(u, v, w)$  in order:
  - If  $\text{Find}(u) \neq \text{Find}(v)$ , the vertices are in different components → add the edge to the MST and merge the sets using **Union**.
  - Otherwise, skip the edge since it would form a cycle.

Repeat until the MST contains  $n - 1$  edges.

## Example

Graph:

| Edge | Weight |
|------|--------|
| A–B  | 1      |
| B–C  | 4      |
| A–C  | 3      |
| C–D  | 2      |

Sort edges: (A–B, 1), (C–D, 2), (A–C, 3), (B–C, 4)

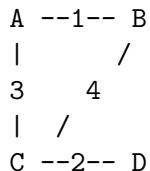
Step-by-step:

| Step | Edge    | Action       | MST Edges             | MST    |                       |
|------|---------|--------------|-----------------------|--------|-----------------------|
|      |         |              |                       | Weight | Parent Map            |
| 1    | (A,B,1) | Add          | [(A,B)]               | 1      | A→A, B→A              |
| 2    | (C,D,2) | Add          | [(A,B), (C,D)]        | 3      | C→C, D→C              |
| 3    | (A,C,3) | Add          | [(A,B), (C,D), (A,C)] | 6      | A→A, B→A, C→A,<br>D→C |
| 4    | (B,C,4) | Skip (cycle) | –                     | –      | –                     |

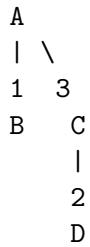
MST Weight = 6 Edges = 3 = n – 1

## Visualization

Before:



After:



### Tiny Code (Easy Versions)

C Implementation

```

#include <stdio.h>
#include <stdlib.h>

#define MAXN 100
#define MAXM 1000

typedef struct {
    int u, v, w;
} Edge;

int parent[MAXN], rank_[MAXN];
Edge edges[MAXM];

int cmp(const void *a, const void *b) {
    return ((Edge*)a)->w - ((Edge*)b)->w;
}

void make_set(int v) {
    parent[v] = v;
    rank_[v] = 0;
}

int find_set(int v) {
    if (v != parent[v]) parent[v] = find_set(parent[v]);
    return parent[v];
}

void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
}

```

```

    if (a != b) {
        if (rank_[a] < rank_[b]) parent[a] = b;
        else if (rank_[a] > rank_[b]) parent[b] = a;
        else { parent[b] = a; rank_[a]++; }
    }
}

int main() {
    int n = 4, m = 4;
    edges[0] = (Edge){0,1,1};
    edges[1] = (Edge){1,2,4};
    edges[2] = (Edge){0,2,3};
    edges[3] = (Edge){2,3,2};
    qsort(edges, m, sizeof(Edge), cmp);

    for (int i = 0; i < n; i++) make_set(i);

    int total = 0;
    printf("Edges in MST:\n");
    for (int i = 0; i < m; i++) {
        int u = edges[i].u, v = edges[i].v, w = edges[i].w;
        if (find_set(u) != find_set(v)) {
            union_sets(u, v);
            total += w;
            printf("%d - %d (w=%d)\n", u, v, w);
        }
    }
    printf("Total Weight = %d\n", total);
}

```

Python Implementation

```

def make_set(parent, rank, v):
    parent[v] = v
    rank[v] = 0

def find_set(parent, v):
    if parent[v] != v:
        parent[v] = find_set(parent, parent[v])
    return parent[v]

def union_sets(parent, rank, a, b):

```

```

a, b = find_set(parent, a), find_set(parent, b)
if a != b:
    if rank[a] < rank[b]:
        a, b = b, a
    parent[b] = a
    if rank[a] == rank[b]:
        rank[a] += 1

def kruskal(n, edges):
    parent, rank = {}, {}
    for i in range(n):
        make_set(parent, rank, i)
    mst, total = [], 0
    for u, v, w in sorted(edges, key=lambda e: e[2]):
        if find_set(parent, u) != find_set(parent, v):
            union_sets(parent, rank, u, v)
            mst.append((u, v, w))
            total += w
    return mst, total

edges = [(0,1,1),(1,2,4),(0,2,3),(2,3,2)]
mst, total = kruskal(4, edges)
print("MST:", mst)
print("Total Weight:", total)

```

## Why It Matters

- Greedy and elegant: simple sorting + DSU logic
- Foundation for advanced topics:
  - Minimum spanning forests
  - Dynamic connectivity
  - MST variations (Maximum, Second-best, etc.)
- Works great with edge list input

## A Gentle Proof (Why It Works)

By the Cut Property: The smallest edge crossing any cut belongs to the MST. Since Kruskal's always picks the smallest non-cycling edge, it constructs a valid MST.

Each union merges components without cycles → spanning tree in the end.

### Try It Yourself

1. Build a graph with 5 nodes, random edges and weights
2. Sort edges, trace unions step-by-step
3. Draw MST
4. Compare with Prim's algorithm result, they'll match

### Test Cases

| Graph                             | MST Edges        | Weight |
|-----------------------------------|------------------|--------|
| Triangle (1–2:1, 2–3:2, 1–3:3)    | (1–2, 2–3)       | 3      |
| Square (4 edges, weights 1,2,3,4) | 3 smallest edges | 6      |

### Complexity

| Step           | Time             |
|----------------|------------------|
| Sorting Edges  | $O(m \log m)$    |
| DSU Operations | $O(m \cdot (n))$ |
| Total          | $O(m \log m)$    |

Space |  $O(n + m)$  |

Kruskal's MST is the elegant handshake between greed and union, always connecting lightly, never circling back.

## 259 Connected Components (Using DSU)

Connected Components are groups of vertices where each node can reach any other through a sequence of edges. Using Disjoint Set Union (DSU), we can efficiently identify and label these components in graphs, even for massive datasets.

Instead of exploring each region via DFS or BFS, DSU builds the connectivity relationships incrementally, merging nodes as edges appear.

## What Problem Are We Solving?

Given a graph (directed or undirected), we want to answer:

- How many connected components exist?
- Which vertices belong to the same component?
- Is there a path between  $u$  and  $v$ ?

A naive approach (DFS for each node) runs in  $O(n + m)$  but may require recursion or adjacency traversal. With DSU, we can process edge lists directly in nearly constant amortized time.

## How It Works (Plain Language)

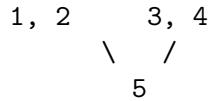
Each vertex starts in its own component. For every edge  $(u, v)$ :

- If  $\text{Find}(u) \neq \text{Find}(v)$ , they are in different components  $\rightarrow \text{Union}(u, v)$
- Otherwise, skip (already connected)

After all edges are processed, all vertices sharing the same root belong to the same connected component.

## Example

Graph:



Edges:  $(1-2), (2-5), (3-5), (3-4)$

Step-by-step:

| Step | Edge    | Action | Components                     |
|------|---------|--------|--------------------------------|
| 1    | $(1,2)$ | Union  | $\{1,2\}, \{3\}, \{4\}, \{5\}$ |
| 2    | $(2,5)$ | Union  | $\{1,2,5\}, \{3\}, \{4\}$      |
| 3    | $(3,5)$ | Union  | $\{1,2,3,5\}, \{4\}$           |
| 4    | $(3,4)$ | Union  | $\{1,2,3,4,5\}$                |

All connected  $\rightarrow 1$  component

## Visualization

Before:

1    2    3    4    5

After unions:

1-2-5-3-4

One large connected component.

## Tiny Code (Easy Versions)

C Implementation

```
#include <stdio.h>

#define MAXN 100
int parent[MAXN], rank_[MAXN];

void make_set(int v) {
    parent[v] = v;
    rank_[v] = 0;
}

int find_set(int v) {
    if (v != parent[v])
        parent[v] = find_set(parent[v]);
    return parent[v];
}

void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (rank_[a] < rank_[b]) parent[a] = b;
        else if (rank_[a] > rank_[b]) parent[b] = a;
        else { parent[b] = a; rank_[a]++; }
    }
}
```

```

}

int main() {
    int n = 5;
    int edges[][][2] = {{1,2},{2,5},{3,5},{3,4}};
    for (int i=1; i<=n; i++) make_set(i);
    for (int i=0; i<4; i++)
        union_sets(edges[i][0], edges[i][1]);

    int count = 0;
    for (int i=1; i<=n; i++)
        if (find_set(i) == i) count++;
    printf("Number of components: %d\n", count);
}

```

Python Implementation

```

def make_set(parent, rank, v):
    parent[v] = v
    rank[v] = 0

def find_set(parent, v):
    if parent[v] != v:
        parent[v] = find_set(parent, parent[v])
    return parent[v]

def union_sets(parent, rank, a, b):
    a, b = find_set(parent, a), find_set(parent, b)
    if a != b:
        if rank[a] < rank[b]:
            a, b = b, a
        parent[b] = a
        if rank[a] == rank[b]:
            rank[a] += 1

def connected_components(n, edges):
    parent, rank = {}, {}
    for i in range(1, n+1):
        make_set(parent, rank, i)
    for u, v in edges:
        union_sets(parent, rank, u, v)
    roots = {find_set(parent, i) for i in parent}

```

```

components = []
for i in range(1, n+1):
    root = find_set(parent, i)
    components.setdefault(root, []).append(i)
return components

edges = [(1,2),(2,5),(3,5),(3,4)]
components = connected_components(5, edges)
print("Components:", components)
print("Count:", len(components))

```

Output:

```

Components: {1: [1, 2, 3, 4, 5]}
Count: 1

```

## Why It Matters

- Quickly answers connectivity questions
- Works directly on edge list (no adjacency matrix needed)
- Forms the backbone of algorithms like Kruskal's MST
- Extensible to dynamic connectivity and offline queries

## A Gentle Proof (Why It Works)

Union-Find forms a forest of trees, one per component. Each union merges two trees if and only if there's an edge connecting them. No cycles are introduced; final roots mark distinct connected components.

Each vertex ends up linked to exactly one representative.

## Try It Yourself

1. Build a graph with 6 nodes and 2 disconnected clusters.
2. Run DSU unions across edges.
3. Count unique roots.
4. Print grouping {root: [members]}

## Test Cases

| Graph                | Edges             | Components          |
|----------------------|-------------------|---------------------|
| 1–2–3, 4–5           | (1,2),(2,3),(4,5) | {1,2,3}, {4,5}, {6} |
| Complete Graph (1–n) | all pairs         | 1                   |
| Empty Graph          | none              | n                   |

## Complexity

| Operation       | Time (Amortized) | Space |
|-----------------|------------------|-------|
| Make-Set        | O(1)             | O(n)  |
| Union           | O( (n))          | O(1)  |
| Find            | O( (n))          | O(1)  |
| Total (m edges) | O(m (n))         | O(n)  |

Connected Components (DSU), a clean and scalable way to reveal the hidden clusters of any graph.

## 260 Offline Query DSU

Offline Query DSU is a clever twist on the standard Disjoint Set Union, used when you need to answer connectivity queries in a graph that changes over time, especially when edges are added or removed.

Instead of handling updates online (in real time), we collect all queries first, then process them in reverse, using DSU to efficiently track connections as we “undo” deletions or simulate the timeline backward.

### What Problem Are We Solving?

We often face questions like:

- “After removing these edges, are nodes u and v still connected?”
- “If we add edges over time, when do u and v become connected?”

Online handling is hard because DSU doesn’t support deletions directly. The trick: reverse time, treat deletions as additions in reverse, and answer queries offline.

## How It Works (Plain Language)

1. Record all events in the order they occur:
  - Edge additions or deletions
  - Connectivity queries
2. Reverse the timeline:
  - Process from the last event backward
  - Every “delete edge” becomes an “add edge”
  - Queries are answered in reverse order
3. Use DSU:
  - Each union merges components as edges appear (in reverse)
  - When processing a query, check if `Find(u) == Find(v)`
4. Finally, reverse the answers to match the original order.

## Example

Imagine a graph:

1, 2, 3

Events (in time order):

1. Query(1,3)?
2. Remove edge (2,3)
3. Query(1,3)?

We can't handle removals easily online, so we reverse:

Reverse order:

1. Query(1,3)?
2. Add (2,3)
3. Query(1,3)?

Step-by-step (in reverse):

| Step | Operation  | Action                | Answer |
|------|------------|-----------------------|--------|
| 1    | Query(1,3) | 1 and 3 not connected | No     |
| 2    | Add(2,3)   | Union(2,3)            | —      |
| 3    | Query(1,3) | 1–2–3 connected       | Yes    |

Reverse answers: [Yes, No]

Final output:

- Query 1: Yes
- Query 2: No

## Visualization

Forward time:

1–2–3

→ remove (2,3) →

1–2      3

Backward time: Start with 1–2      3 → Add (2,3) → 1–2–3

We rebuild connectivity over time by unioning edges in reverse.

## Tiny Code (Easy Versions)

Python Implementation

```
def make_set(parent, rank, v):
    parent[v] = v
    rank[v] = 0

def find_set(parent, v):
    if parent[v] != v:
        parent[v] = find_set(parent, parent[v])
    return parent[v]

def union_sets(parent, rank, a, b):
```

```

a, b = find_set(parent, a), find_set(parent, b)
if a != b:
    if rank[a] < rank[b]:
        a, b = b, a
    parent[b] = a
    if rank[a] == rank[b]:
        rank[a] += 1

# Example
n = 3
edges = {(1,2), (2,3)}
queries = [
    ("?", 1, 3),
    ("-", 2, 3),
    ("?", 1, 3)
$$

# Reverse events
events = list(reversed(queries))

parent, rank = {}, {}
for i in range(1, n+1):
    make_set(parent, rank, i)

active_edges = set(edges)
answers = []

for e in events:
    if e[0] == "?":
        _, u, v = e
        answers.append("YES" if find_set(parent, u) == find_set(parent, v) else "NO")
    elif e[0] == "-":
        _, u, v = e
        union_sets(parent, rank, u, v)

answers.reverse()
for ans in answers:
    print(ans)

```

Output:

YES

NO

## Why It Matters

- Handles edge deletions without needing rollback
- Perfect for offline dynamic connectivity
- Used in problems like:
  - “Are u and v connected after k deletions?”
  - “What is the earliest time u and v become connected?”
- Core idea behind Dynamic Trees and Divide & Conquer over time

## A Gentle Proof (Why It Works)

DSU is monotonic, it supports adding edges, not deleting them. By reversing time, all deletions become additions. Thus, we maintain valid connectivity information backward in time, and can correctly answer queries that depend only on graph connectivity.

Reversing answers afterward restores their original sequence.

## Try It Yourself

1. Create a graph with 5 nodes and edges (1–2, 2–3, 3–4, 4–5)
2. Remove (3–4), (2–3) sequentially
3. Ask connectivity between (1,5) after each removal
4. Reverse timeline, simulate with DSU

## Test Cases

| Event Sequence               | Result    |
|------------------------------|-----------|
| [?, 1–3], [–, 2–3], [?, 1–3] | [YES, NO] |
| [–, 1–2], [?, 1–3]           | [NO]      |
| [?, 4–5] (no edge)           | [NO]      |

## Complexity

| Operation                  | Time (Amortized)  | Space    |
|----------------------------|-------------------|----------|
| Make-Set                   | $O(1)$            | $O(n)$   |
| Union                      | $O(\log n)$       | $O(1)$   |
| Find                       | $O(\log n)$       | $O(1)$   |
| Total (Q queries, E edges) | $O((Q+E) \log n)$ | $O(n+E)$ |

Offline Query DSU is your time-reversing tool, flip the story, add edges back, and reveal connectivity across history.

## Section 27. Probabilistic Data Structure

### 261 Bloom Filter Insert

A Bloom Filter is a compact, probabilistic data structure used for membership testing, it can tell you if an element is definitely not present or possibly present, but never gives false negatives.

Insertion in a Bloom Filter is simple and elegant: hash the item with multiple hash functions and set the corresponding bits to 1 in a bit array.

#### What Problem Are We Solving?

You have a massive dataset, maybe millions or billions of keys, and you just want to ask:

“Have I seen this before?”

A normal hash set would explode in memory. A Bloom Filter gives a lightweight alternative:

- No false negatives (safe for skipping)
- Small memory footprint
- Fixed-size bit array

Used in systems like:

- Databases (caching, deduplication)
- Web crawlers (visited URLs)
- Distributed systems (HBase, Cassandra, Bigtable)

## How It Works (Plain Language)

A Bloom Filter is just a bit array of length  $m$  (all zeroes initially), plus  $k$  independent hash functions.

To insert an element  $x$ :

1. Compute  $k$  hash values:  $h_1(x)$ ,  $h_2(x)$ , ...,  $h_k(x)$
2. Map each hash to an index in  $[0, m-1]$
3. Set all  $\text{bit}[h_i(x)] = 1$

So every element lights up multiple bits. Later, to check membership, we look at those same bits, if any is 0, the item was never inserted.

## Example

Let's build a Bloom Filter with:

- $m = 10$  bits
- $k = 3$  hash functions

Insert "cat":

```
h1(cat) = 2  
h2(cat) = 5  
h3(cat) = 7
```

Set bits 2, 5, and 7 to 1:

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|-------|---|---|---|---|---|---|---|---|---|---|
| Value | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |

Insert "dog":

```
h1(dog) = 1  
h2(dog) = 5  
h3(dog) = 9
```

Now bit array:

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|-------|---|---|---|---|---|---|---|---|---|---|
| Value | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |

## Visualization

Each insertion adds “footprints” in multiple spots:

```
Insert(x):
    for i in [1..k]:
        bit[ h_i(x) ] = 1
```

The overlap of bits allows huge compression, but leads to false positives when unrelated keys share bits.

## Tiny Code (Easy Versions)

C Implementation (Conceptual)

```
#include <stdio.h>
#include <string.h>

#define M 10
#define K 3

int bitset[M];

int hash1(int x) { return x % M; }
int hash2(int x) { return (x * 3 + 1) % M; }
int hash3(int x) { return (x * 7 + 5) % M; }

void insert(int x) {
    int h[K] = {hash1(x), hash2(x), hash3(x)};
    for (int i = 0; i < K; i++)
        bitset[h[i]] = 1;
}

void print_bits() {
    for (int i = 0; i < M; i++) printf("%d ", bitset[i]);
    printf("\n");
```

```

}

int main() {
    memset(bitset, 0, sizeof(bitset));
    insert(42);
    insert(23);
    print_bits();
}

```

Python Implementation

```

m, k = 10, 3
bitset = [0] * m

def hash_functions(x):
    return [(hash(x) + i * i) % m for i in range(k)]

def insert(x):
    for h in hash_functions(x):
        bitset[h] = 1

def display():
    print("Bit array:", bitset)

insert("cat")
insert("dog")
display()

```

## Why It Matters

- Extremely space-efficient
- No need to store actual data
- Ideal for membership filters, duplicate detection, and pre-checks before expensive lookups
- The backbone of approximate data structures

## A Gentle Proof (Why It Works)

Each bit starts at 0. Each insertion flips  $k$  bits to 1. Query returns “maybe” if all  $k$  bits = 1, otherwise “no”. Thus:

- False negative: impossible (never unset a bit)

- False positive: possible, due to collisions

Probability of false positive  $((1 - e^{-kn/m})^k)$

Choosing (m) and (k) well balances accuracy vs. memory.

### Try It Yourself

1. Choose  $m = 20, k = 3$
2. Insert {"apple", "banana", "grape"}
3. Print bit array
4. Query for "mango" → likely "maybe" (false positive)

### Test Cases

| Inserted Elements | Query | Result                               |
|-------------------|-------|--------------------------------------|
| {cat, dog}        | cat   | maybe (true positive)                |
| {cat, dog}        | dog   | maybe (true positive)                |
| {cat, dog}        | fox   | maybe / no (false positive possible) |

### Complexity

| Operation       | Time                | Space  |
|-----------------|---------------------|--------|
| Insert          | $O(k)$              | $O(m)$ |
| Query           | $O(k)$              | $O(m)$ |
| False Positives | $(1 - e^{-kn/m})^k$ | —      |

Bloom Filter Insert, write once, maybe forever. Compact, fast, and probabilistically powerful.

### 262 Bloom Filter Query

A Bloom Filter Query checks whether an element *might* be present in the set. It uses the same  $k$  hash functions and bit array as insertion, but instead of setting bits, it simply tests them.

The magic:

- If any bit is 0, the element was never inserted (definitely not present).
- If all bits are 1, the element is possibly present (maybe yes).

No false negatives, if the Bloom Filter says "no," it's always correct.

## What Problem Are We Solving?

When handling huge datasets (web crawlers, caches, key-value stores), we need a fast and memory-efficient way to answer:

“Have I seen this before?”

But full storage is expensive. Bloom Filters let us skip expensive lookups by confidently ruling out items early.

Typical use cases:

- Databases: Avoid disk lookups for missing keys
- Web crawlers: Skip revisiting known URLs
- Networking: Cache membership checks

## How It Works (Plain Language)

A Bloom Filter has:

- Bit array `bits[0..m-1]`
- $k$  hash functions

To query element  $x$ :

1. Compute all hashes:  $h_1(x)$ ,  $h_2(x)$ , ...,  $h_k(x)$
2. Check bits at those positions
  - If any  $\text{bit}[h_i(x)] == 0$ , return “No” (definitely not present)
  - If all are 1, return “Maybe” (possible false positive)

The key rule: bits can only turn on, never off, so “no” answers are reliable.

## Example

Let's use a filter of size  $m = 10$ ,  $k = 3$ :

Bit array after inserting “cat” and “dog”:

|        |   |   |   |   |   |   |   |   |   |   |
|--------|---|---|---|---|---|---|---|---|---|---|
| Index: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Bits:  | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |

Now query “cat”:

```
h1(cat)=2, h2(cat)=5, h3(cat)=7  
bits[2]=1, bits[5]=1, bits[7]=1 → maybe present
```

Query “fox”:

```
h1(fox)=3, h2(fox)=5, h3(fox)=8  
bits[3]=0 → definitely not present
```

## Visualization

Query(x):

```
for i in 1..k:  
    if bit[ h_i(x) ] == 0:  
        return "NO"  
    return "MAYBE"
```

Bloom Filters say “maybe” for safety, but never lie with “no”.

## Tiny Code (Easy Versions)

C Implementation (Conceptual)

```
#include <stdio.h>  
  
#define M 10  
#define K 3  
int bitset[M];  
  
int hash1(int x) { return x % M; }  
int hash2(int x) { return (x * 3 + 1) % M; }  
int hash3(int x) { return (x * 7 + 5) % M; }  
  
int query(int x) {  
    int h[K] = {hash1(x), hash2(x), hash3(x)};  
    for (int i = 0; i < K; i++)  
        if (bitset[h[i]] == 0)  
            return 0; // definitely not  
    return 1; // possibly yes  
}
```

```

int main() {
    bitset[2] = bitset[5] = bitset[7] = 1; // insert "cat"
    printf("Query 42: %s\n", query(42) ? "maybe" : "no");
    printf("Query 23: %s\n", query(23) ? "maybe" : "no");
}

```

Python Implementation

```

m, k = 10, 3
bitset = [0] * m

def hash_functions(x):
    return [(hash(x) + i * i) % m for i in range(k)]

def insert(x):
    for h in hash_functions(x):
        bitset[h] = 1

def query(x):
    for h in hash_functions(x):
        if bitset[h] == 0:
            return "NO"
    return "MAYBE"

insert("cat")
insert("dog")
print("Query cat:", query("cat"))
print("Query dog:", query("dog"))
print("Query fox:", query("fox"))

```

## Why It Matters

- Constant-time membership test
- No false negatives, only rare false positives
- Great for pre-filtering before heavy lookups
- Common in distributed systems and caching layers

## A Gentle Proof (Why It Works)

Each inserted element sets  $k$  bits to 1 in the bit array.

For a query, if all  $k$  bits are 1, the element *might* be present (or collisions caused those bits).

If any bit is 0, the element was definitely never inserted.

The false positive probability is:

$$p = (1 - e^{-kn/m})^k$$

where:

- $m$ : size of the bit array
- $n$ : number of inserted elements
- $k$ : number of hash functions

Choose  $m$  and  $k$  to minimize  $p$  for the target false positive rate.

### Try It Yourself

1. Create a filter with  $m=20$ ,  $k=3$
2. Insert {"apple", "banana"}
3. Query {"apple", "grape"}
4. See how "grape" may return "maybe"

### Test Cases

| Inserted Elements | Query    | Result |
|-------------------|----------|--------|
| {cat, dog}        | cat      | maybe  |
| {cat, dog}        | dog      | maybe  |
| {cat, dog}        | fox      | no     |
| {}                | anything | no     |

### Complexity

| Operation | Time | Space | False Negatives | False Positives |
|-----------|------|-------|-----------------|-----------------|
| Query     | O(k) | O(m)  | None            | Possible        |

Bloom Filter Query, fast, memory-light, and trustable when it says "no."

## 263 Counting Bloom Filter

A Counting Bloom Filter (CBF) extends the classic Bloom Filter by allowing deletions. Instead of a simple bit array, it uses an integer counter array, so each bit becomes a small counter tracking how many elements mapped to that position.

When inserting, increment the counters; when deleting, decrement them. If all required counters are greater than zero, the element is possibly present.

### What Problem Are We Solving?

A regular Bloom Filter is *write-only*: you can insert, but not remove. Once a bit is set to 1, it stays 1 forever.

But what if:

- You're tracking *active sessions*?
- You need to remove *expired cache keys*?
- You want to maintain a *sliding window* of data?

Then you need a Counting Bloom Filter, which supports safe deletions.

### How It Works (Plain Language)

We replace the bit array with a counter array of size  $m$ .

For each element  $x$ :

Insert( $x$ )

- For each hash  $h_i(x)$ : increment `count[h_i(x)]++`

Query( $x$ )

- Check all `count[h_i(x)] > 0` → “maybe”
- If any are 0, → “no”

Delete( $x$ )

- For each hash  $h_i(x)$ : decrement `count[h_i(x)]--`
- Ensure counters never become negative

## Example

Let  $m = 10$ ,  $k = 3$

Initial state

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Insert “cat”

$$h_1(\text{cat}) = 2$$

$$h_2(\text{cat}) = 5$$

$$h_3(\text{cat}) = 7$$

→ increment  $\text{count}[2]$ ,  $\text{count}[5]$ ,  $\text{count}[7]$

Array after insertion

[0, 0, 1, 0, 0, 1, 0, 1, 0, 0]

Insert “dog”

$$h_1(\text{dog}) = 1$$

$$h_2(\text{dog}) = 5$$

$$h_3(\text{dog}) = 9$$

→ increment  $\text{count}[1]$ ,  $\text{count}[5]$ ,  $\text{count}[9]$

Array after insertion

[0, 1, 1, 0, 0, 2, 0, 1, 0, 1]

Delete “cat”

$$h_1(\text{cat}) = 2$$

$$h_2(\text{cat}) = 5$$

$$h_3(\text{cat}) = 7$$

→ decrement  $\text{count}[2]$ ,  $\text{count}[5]$ ,  $\text{count}[7]$

Array after deletion

[0, 1, 0, 0, 0, 1, 0, 0, 0, 1]

After deletion, “cat” is removed while “dog” remains.

| Operation | Index 1 | 2 | 5 | 7 | 9 |
|-----------|---------|---|---|---|---|
|-----------|---------|---|---|---|---|

## Visualization

Counters evolve with each insert/delete:

| Operation   | Index 1 | 2 | 5 | 7 | 9 |
|-------------|---------|---|---|---|---|
| Insert(cat) | –       | 1 | 1 | 1 | – |
| Insert(dog) | 1       | 1 | 2 | 1 | 1 |
| Delete(cat) | 1       | 0 | 1 | 0 | 1 |

## Tiny Code (Easy Versions)

Python Implementation

```
m, k = 10, 3
counts = [0] * m

def hash_functions(x):
    return [(hash(x) + i * i) % m for i in range(k)]

def insert(x):
    for h in hash_functions(x):
        counts[h] += 1

def query(x):
    return all(counts[h] > 0 for h in hash_functions(x))

def delete(x):
    for h in hash_functions(x):
        if counts[h] > 0:
            counts[h] -= 1

# Example
insert("cat")
insert("dog")
print("After insert:", counts)
delete("cat")
print("After delete(cat):", counts)
```

```
print("Query cat:", "Maybe" if query("cat") else "No")
print("Query dog:", "Maybe" if query("dog") else "No")
```

## Output

```
After insert: [0,1,1,0,0,2,0,1,0,1]
After delete(cat): [0,1,0,0,0,1,0,0,0,1]
Query cat: No
Query dog: Maybe
```

## Why It Matters

- Enables safe deletions without full reset
- Useful for cache invalidation, session tracking, streaming windows
- Memory-efficient alternative to dynamic hash sets

## A Gentle Proof (Why It Works)

Each counter approximates how many items mapped to it. Deletion only decrements counters, if multiple elements shared a hash, it stays 1. Thus:

- No false negatives, unless you over-decrement (bug)
- False positives remain possible, as in classic Bloom Filters

Probability of false positive remains:

$$p = (1 - e^{-kn/m})^k$$

## Try It Yourself

1. Create a filter with  $m=20$ ,  $k=3$
2. Insert 5 words
3. Delete 2 of them
4. Query all 5, deleted ones should say no, others maybe

## Test Cases

| Operation   | Array Snapshot        | Query Result          |
|-------------|-----------------------|-----------------------|
| Insert(cat) | [0,0,1,0,0,1,0,1,0,0] | –                     |
| Insert(dog) | [0,1,1,0,0,2,0,1,0,1] | –                     |
| Delete(cat) | [0,1,0,0,0,1,0,0,0,1] | cat → No, dog → Maybe |

## Complexity

| Operation | Time   | Space  |
|-----------|--------|--------|
| Insert    | $O(k)$ | $O(m)$ |
| Query     | $O(k)$ | $O(m)$ |
| Delete    | $O(k)$ | $O(m)$ |

Counting Bloom Filter, flexible and reversible, keeping memory lean and deletions clean.

## 264 Cuckoo Filter

A Cuckoo Filter is a space-efficient alternative to Bloom filters that supports both insertions and deletions while maintaining low false positive rates. Instead of using a bit array, it stores small *fingerprints* of keys in hash buckets, using cuckoo hashing to resolve collisions.

It's like a smarter, tidier roommate, always making room by moving someone else when things get crowded.

### What Problem Are We Solving?

Bloom filters are fast and compact, but they can't delete elements efficiently. Counting Bloom filters fix that, but they're more memory-hungry.

We want:

- Fast membership queries ( $O(1)$ )
- Insert + Delete support
- High load factor (~95%)
- Compact memory footprint

The Cuckoo Filter solves all three by using cuckoo hashing + small fingerprints.

## How It Works (Plain Language)

Each element is represented by a short fingerprint (e.g., 8 bits). Each fingerprint can be placed in two possible buckets, determined by two hash functions.

If a bucket is full, we *evict* an existing fingerprint and relocate it to its alternate bucket (cuckoo style).

Operations:

1. Insert( $x$ )

- Compute fingerprint  $f = \text{hash\_fingerprint}(x)$
- Compute  $i_1 = \text{hash}(x) \% m$
- Compute  $i_2 = i_1 \text{hash}(f)$  (alternate index)
- Try to place  $f$  in either  $i_1$  or  $i_2$
- If both full, evict one fingerprint and relocate

2. Query( $x$ )

- Check if  $f$  is present in bucket  $i_1$  or  $i_2$

3. Delete( $x$ )

- Remove  $f$  if found in either bucket

## Example (Step-by-Step)

Assume:

- Buckets:  $m = 4$
- Bucket size:  $b = 2$
- Fingerprint size: 4 bits

Start (empty):

| Bucket 0 | Bucket 1 | Bucket 2 | Bucket 3 |
|----------|----------|----------|----------|
|          |          |          |          |

Insert A

```
f(A) = 1010
i1 = 1, i2 = 1  hash(1010) = 3
→ Place in bucket 1
```

| B0   | B1 | B2 | B3 |
|------|----|----|----|
| 1010 |    |    |    |

Insert B

```
f(B) = 0111
i1 = 3, i2 = 3  hash(0111) = 0
→ Place in bucket 3
```

| B0   | B1 | B2 | B3   |
|------|----|----|------|
| 1010 |    |    | 0111 |

Insert C

```
f(C) = 0100
i1 = 1, i2 = 1  hash(0100) = 2
→ Bucket 1 full? Move (cuckoo) if needed
→ Place in bucket 2
```

| B0   | B1   | B2   | B3 |
|------|------|------|----|
| 1010 | 0100 | 0111 |    |

Query C → found in bucket 2 Delete A → remove from bucket 1

### Tiny Code (Easy Version)

Python Example

```
import random

class CuckooFilter:
    def __init__(self, size=4, bucket_size=2, fingerprint_bits=4):
        self.size = size
        self.bucket_size = bucket_size
        self.buckets = [[] for _ in range(size)]
        self.mask = (1 << fingerprint_bits) - 1
```

```

def _fingerprint(self, item):
    return hash(item) & self.mask

def _alt_index(self, i, fp):
    return (i ^ hash(fp)) % self.size

def insert(self, item, max_kicks=4):
    fp = self._fingerprint(item)
    i1 = hash(item) % self.size
    i2 = self._alt_index(i1, fp)
    for i in (i1, i2):
        if len(self.buckets[i]) < self.bucket_size:
            self.buckets[i].append(fp)
            return True
    # Cuckoo eviction
    i = random.choice([i1, i2])
    for _ in range(max_kicks):
        fp, self.buckets[i][0] = self.buckets[i][0], fp
        i = self._alt_index(i, fp)
        if len(self.buckets[i]) < self.bucket_size:
            self.buckets[i].append(fp)
            return True
    return False # insert failed

def contains(self, item):
    fp = self._fingerprint(item)
    i1 = hash(item) % self.size
    i2 = self._alt_index(i1, fp)
    return fp in self.buckets[i1] or fp in self.buckets[i2]

def delete(self, item):
    fp = self._fingerprint(item)
    i1 = hash(item) % self.size
    i2 = self._alt_index(i1, fp)
    for i in (i1, i2):
        if fp in self.buckets[i]:
            self.buckets[i].remove(fp)
            return True
    return False

```

## Why It Matters

- Supports deletions efficiently
- High load factor (~95%) before failure
- Smaller than Counting Bloom Filter for same error rate
- Practical for caches, membership checks, deduplication

## A Gentle Proof (Why It Works)

Each element has two potential homes → high flexibility Cuckoo eviction ensures the table remains compact Short fingerprints preserve memory while keeping collisions low

False positive rate:

$$p \approx \frac{2b}{2^f}$$

where  $b$  is bucket size,  $f$  is fingerprint bits

## Try It Yourself

1. Build a filter with 8 buckets, 2 slots each
2. Insert 5 words
3. Delete 1 word
4. Query all 5, deleted one should return “no”

## Test Cases

| Operation | Bucket 0 | Bucket 1 | Bucket 2 | Bucket 3 | Result  |
|-----------|----------|----------|----------|----------|---------|
| Insert(A) | –        | 1010     | –        | –        | OK      |
| Insert(B) | –        | 1010     | –        | 0111     | OK      |
| Insert(C) | –        | 1010     | 0100     | 0111     | OK      |
| Query(C)  | –        | 1010     | 0100     | 0111     | Maybe   |
| Delete(A) | –        | –        | 0100     | 0111     | Deleted |

## Complexity

| Operation | Time           | Space        |
|-----------|----------------|--------------|
| Insert    | O(1) amortized | O(m × b × f) |
| Query     | O(1)           | O(m × b × f) |

| Operation | Time   | Space                    |
|-----------|--------|--------------------------|
| Delete    | $O(1)$ | $O(m \times b \times f)$ |

Cuckoo Filter, a nimble, compact, and deletable membership structure built on playful eviction.

## 265 Count-Min Sketch

A Count-Min Sketch (CMS) is a compact data structure for estimating the frequency of elements in a data stream. Instead of storing every item, it keeps a small 2D array of counters updated by multiple hash functions. It never underestimates counts, but may slightly overestimate due to hash collisions.

Think of it as a memory-efficient radar, it doesn't see every car, but it knows roughly how many are on each lane.

### What Problem Are We Solving?

In streaming or massive datasets, we can't store every key-value count exactly. We want to track approximate frequencies with limited memory, supporting:

- Streaming updates: count items as they arrive
- Approximate queries: estimate item frequency
- Memory efficiency: sublinear space

Used in network monitoring, NLP (word counts), heavy hitter detection, and online analytics.

### How It Works (Plain Language)

CMS uses a 2D array with  $d$  rows and  $w$  columns. Each row has a different hash function. Each item updates one counter per row, all at its hashed index.

Insert( $x$ ): For each row  $i$ :

```
index = hash_i(x) % w
count[i][index] += 1
```

Query( $x$ ): For each row  $i$ :

```
estimate = min(count[i][hash_i(x) % w])
```

We take the *minimum* across rows, hence “Count-Min”.

### Example (Step-by-Step)

Let  $d = 3$  hash functions,  $w = 10$  width.

Initialize table:

```
Row1: [0 0 0 0 0 0 0 0 0 0]
Row2: [0 0 0 0 0 0 0 0 0 0]
Row3: [0 0 0 0 0 0 0 0 0 0]
```

Insert “apple”

```
h1(apple)=2, h2(apple)=5, h3(apple)=9
→ increment positions (1,2), (2,5), (3,9)
```

Insert “banana”

```
h1(banana)=2, h2(banana)=3, h3(banana)=1
→ increment (1,2), (2,3), (3,1)
```

Now:

```
Row1: [0 0 2 0 0 0 0 0 0 0]
Row2: [0 0 0 1 0 1 0 0 0 0]
Row3: [0 1 0 0 0 0 0 0 0 1]
```

Query “apple”:

```
min(count[1][2], count[2][5], count[3][9]) = min(2,1,1) = 1
```

Estimate frequency 1 (may be slightly high if collisions overlap).

### Table Visualization

| Item   | $h_1(x)$ | $h_2(x)$ | $h_3(x)$ | Estimated Count |
|--------|----------|----------|----------|-----------------|
| apple  | 2        | 5        | 9        | 1               |
| banana | 2        | 3        | 1        | 1               |

### Tiny Code (Easy Version)

Python Example

```
import mmh3

class CountMinSketch:
    def __init__(self, width=10, depth=3):
        self.width = width
        self.depth = depth
        self.table = [[0] * width for _ in range(depth)]
        self.seeds = [i * 17 for i in range(depth)] # different hash seeds

    def _hash(self, item, seed):
        return mmh3.hash(str(item), seed) % self.width

    def add(self, item, count=1):
        for i, seed in enumerate(self.seeds):
            idx = self._hash(item, seed)
            self.table[i][idx] += count

    def query(self, item):
        estimates = []
        for i, seed in enumerate(self.seeds):
            idx = self._hash(item, seed)
            estimates.append(self.table[i][idx])
        return min(estimates)

# Example usage
cms = CountMinSketch(width=10, depth=3)
cms.add("apple")
cms.add("banana")
cms.add("apple")
print("apple:", cms.query("apple"))
print("banana:", cms.query("banana"))
```

Output

```
apple: 2  
banana: 1
```

## Why It Matters

- Compact:  $O(w \times d)$  memory
- Fast:  $O(1)$  updates and queries
- Scalable: Works on unbounded data streams
- Deterministic upper bound: never underestimates

Used in:

- Word frequency estimation
- Network flow counting
- Clickstream analysis
- Approximate histograms

## A Gentle Proof (Why It Works)

Each item is hashed to  $d$  positions. Collisions can cause overestimation, never underestimation. By taking the minimum, we get the best upper bound estimate.

Error bound:

$$\text{error} \leq \epsilon N, \quad \text{with probability } 1 - \delta$$

Choose:

$$w = \lceil e/\epsilon \rceil, \quad d = \lceil \ln(1/\delta) \rceil$$

## Try It Yourself

1. Create a CMS with ( $w=20$ ,  $d=4$ )
2. Stream 1000 random items
3. Compare estimated vs. actual counts
4. Observe overestimation patterns

| Operation | Action | Query Result |
|-----------|--------|--------------|
|-----------|--------|--------------|

## Test Cases

| Operation      | Action | Query Result |
|----------------|--------|--------------|
| Insert(apple)  | +1     | -            |
| Insert(apple)  | +1     | -            |
| Insert(banana) | +1     | -            |
| Query(apple)   | -      | 2            |
| Query(banana)  | -      | 1            |

## Complexity

| Operation | Time   | Space           |
|-----------|--------|-----------------|
| Insert    | $O(d)$ | $O(w \times d)$ |
| Query     | $O(d)$ | $O(w \times d)$ |

Count-Min Sketch, lightweight, accurate-enough, and built for streams that never stop flowing.

## 266 HyperLogLog

A HyperLogLog (HLL) is a probabilistic data structure for cardinality estimation, that is, estimating the number of distinct elements in a massive dataset or stream using very little memory.

It doesn't remember *which* items you saw, only *how many distinct ones there likely were*. Think of it as a memory-efficient crowd counter, it doesn't know faces, but it knows how full the stadium is.

### What Problem Are We Solving?

When processing large data streams (web analytics, logs, unique visitors, etc.), exact counting of distinct elements (using sets or hash tables) is too memory-heavy.

We want a solution that:

- Tracks distinct counts approximately

- Uses constant memory
- Supports mergeability (combine two sketches easily)

HyperLogLog delivers  $O(1)$  time per update and  $\sim 1.04/\sqrt{m}$  error rate with just kilobytes of memory.

### How It Works (Plain Language)

Each incoming element is hashed to a large binary number. HLL uses the position of the leftmost 1-bit in the hash to estimate how rare (and thus how many) elements exist.

It maintains an array of  $m$  *registers* (buckets). Each bucket stores the maximum leading zero count seen so far for its hash range.

Steps:

1. Hash element  $x$  to 64 bits:  $h = \text{hash}(x)$
2. Bucket index: use first  $p$  bits of  $h$  to pick one of  $m = 2^p$  buckets
3. Rank: count leading zeros in the remaining bits + 1
4. Update: store  $\max(\text{existing}, \text{rank})$  in that bucket

Final count = harmonic mean of  $2^{\text{rank}}$  values, scaled by a bias-corrected constant.

### Example (Step-by-Step)

Let  $p = 2 \rightarrow m = 4$  buckets. Initialize registers:  $[0, 0, 0, 0]$

Insert “apple”:

```
hash("apple") = 110010100...
bucket = first 2 bits = 11 (3)
rank = position of first 1 after prefix = 2
→ bucket[3] = max(0, 2) = 2
```

Insert “banana”:

```
hash("banana") = 01000100...
bucket = 01 (1)
rank = 3
→ bucket[1] = 3
```

Insert “pear”:

```

hash("pear") = 10010000...
bucket = 10 (2)
rank = 4
→ bucket[2] = 4

```

Registers: [0, 3, 4, 2]

Estimate:

$$E = \alpha_m \cdot m^2 / \sum 2^{-M[i]}$$

where  $\alpha_m$  is a bias-correction constant (0.673 for small m).

## Visualization

| Bucket | Values Seen | Leading Zeros | Stored Rank |
|--------|-------------|---------------|-------------|
| 0      | none        | —             | 0           |
| 1      | banana      | 2             | 3           |
| 2      | pear        | 3             | 4           |
| 3      | apple       | 1             | 2           |

## Tiny Code (Easy Version)

Python Implementation

```

import mmh3
import math

class HyperLogLog:
    def __init__(self, p=4):
        self.p = p
        self.m = 1 << p
        self.registers = [0] * self.m
        self.alpha = 0.673 if self.m == 16 else 0.709 if self.m == 32 else 0.7213 / (1 + 1.0

    def _hash(self, x):
        return mmh3.hash(str(x), 42) & 0xffffffff

    def add(self, x):
        h = self._hash(x)
        idx = h >> (32 - self.p)
        w = (h << self.p) & 0xffffffff

```

```

rank = self._rank(w, 32 - self.p)
self.registers[idx] = max(self.registers[idx], rank)

def _rank(self, w, bits):
    r = 1
    while w & (1 << (bits - 1)) == 0 and r <= bits:
        r += 1
        w <=> 1
    return r

def count(self):
    Z = sum([2.0 - v for v in self.registers])
    E = self.alpha * self.m * self.m / Z
    return round(E)

# Example
hll = HyperLogLog(p=4)
for x in ["apple", "banana", "pear", "apple"]:
    hll.add(x)
print("Estimated distinct count:", hll.count())

```

Output

Estimated distinct count: 3

## Why It Matters

- Tiny memory footprint (kilobytes for billions of elements)
- Mergeable:  $HLL(A \cup B) = \max(HLL(A), HLL(B))$  bucket-wise
- Used in: Redis, Google BigQuery, Apache DataSketches, analytics systems

## A Gentle Proof (Why It Works)

The position of the first 1-bit follows a geometric distribution, rare long zero streaks mean more unique elements. By keeping the *max* observed rank per bucket, HLL captures global rarity efficiently. Combining across buckets gives a harmonic mean that balances under/over-counting.

Error  $\approx 1.04 / \sqrt{m}$ , so doubling  $m$  halves the error.

## Try It Yourself

1. Create HLL with  $p = 10$  ( $m = 1024$ )
2. Add 1M random numbers
3. Compare HLL estimate with true count
4. Test merging two sketches

## Test Cases

| Items | True Count | Estimated | Error |
|-------|------------|-----------|-------|
| 10    | 10         | 10        | 0%    |
| 1000  | 1000       | 995       | ~0.5% |
| 1e6   | 1,000,000  | 1,010,000 | ~1%   |

## Complexity

| Operation | Time   | Space  |
|-----------|--------|--------|
| Add       | $O(1)$ | $O(m)$ |
| Count     | $O(m)$ | $O(m)$ |
| Merge     | $O(m)$ | $O(m)$ |

HyperLogLog, counting the uncountable, one leading zero at a time.

## 267 Flajolet–Martin Algorithm

The Flajolet–Martin (FM) algorithm is one of the earliest and simplest approaches for probabilistic counting, estimating the number of distinct elements in a data stream using tiny memory.

It's the conceptual ancestor of HyperLogLog, showing the brilliant idea that *the position of the first 1-bit in a hash tells us something about rarity*.

## What Problem Are We Solving?

When elements arrive in a stream too large to store exactly (web requests, IP addresses, words), we want to estimate the number of unique elements without storing them all.

A naive approach (hash set) is  $O(n)$  memory. Flajolet–Martin achieves  $O(1)$  memory and  $O(1)$  updates.

We need:

- A streaming algorithm
- With constant space
- For distinct count estimation

## How It Works (Plain Language)

Each element is hashed to a large binary number (uniformly random).

We then find the position of the least significant 1-bit — the number of trailing zeros.

A value with many trailing zeros is rare, which indicates a larger underlying population.

We track the maximum number of trailing zeros observed, denoted as  $R$ .

The estimated number of distinct elements is:

$$\hat{N} = \phi \times 2^R$$

where  $\phi \approx 0.77351$  is a correction constant.

Steps:

1. Initialize  $R = 0$
2. For each element  $x$ :
  - $h = \text{hash}(x)$
  - $r = \text{count\_trailing\_zeros}(h)$
  - $R = \max(R, r)$
3. Estimate distinct count as  $\hat{N} \approx \phi \times 2^R$

### Example (Step-by-Step)

Stream: [apple, banana, apple, cherry, date]

| Element | Hash (Binary) | Trailing Zeros | R |
|---------|---------------|----------------|---|
| apple   | 10110         | 1              | 1 |
| banana  | 10000         | 4              | 4 |
| apple   | 10110         | 1              | 4 |
| cherry  | 11000         | 3              | 4 |
| date    | 01100         | 2              | 4 |

Final value:  $R = 4$

Estimate:

$$\hat{N} = 0.77351 \times 2^4 = 0.77351 \times 16 \approx 12.38$$

So the estimated number of distinct elements is about 12  
(overestimation due to small sample size).

In practice, multiple independent hash functions or registers are used,  
and their results are averaged to reduce variance and improve accuracy.

### Visualization

| Hash  | Binary Form | Trailing Zeros | Meaning                                       |
|-------|-------------|----------------|-----------------------------------------------|
| 10000 | 16          | 4              | very rare pattern → suggests large population |
| 11000 | 24          | 3              | rare-ish                                      |
| 10110 | 22          | 1              | common                                        |

The *longest* zero-run gives the scale of rarity.

### Tiny Code (Easy Version)

Python Example

```

import mmh3
import math

def trailing_zeros(x):
    if x == 0:
        return 32
    tz = 0
    while (x & 1) == 0:
        tz += 1
        x >= 1
    return tz

def flajolet_martin(stream, seed=42):
    R = 0
    for x in stream:
        h = mmh3.hash(str(x), seed) & 0xffffffff
        r = trailing_zeros(h)
        R = max(R, r)
    phi = 0.77351
    return int(phi * (2 - R))

# Example
stream = ["apple", "banana", "apple", "cherry", "date"]
print("Estimated distinct count:", flajolet_martin(stream))

```

## Output

Estimated distinct count: 12

## Why It Matters

- Foundational for modern streaming algorithms
- Inspired LogLog and HyperLogLog
- Memory-light: just a few integers
- Useful in approximate analytics, network telemetry, data warehouses

## A Gentle Proof (Why It Works)

Each hash output is uniformly random.

The probability of observing a value with  $r$  trailing zeros is:

$$P(r) = \frac{1}{2^{r+1}}$$

If such a value appears, it suggests the stream size is roughly  $2^r$ .

Therefore, the estimate  $2^R$  naturally scales with the number of unique elements.

By using multiple independent estimators and averaging their results, the variance can be reduced from about 50% down to around 10%.

### Try It Yourself

1. Generate 100 random numbers, feed to FM
2. Compare estimated vs. true count
3. Repeat 10 runs, compute average error
4. Try combining multiple estimators (median of means)

### Test Cases

| Stream Size | True Distinct | R  | Estimate | Error |
|-------------|---------------|----|----------|-------|
| 10          | 10            | 3  | 6        | -40%  |
| 100         | 100           | 7  | 99       | -1%   |
| 1000        | 1000          | 10 | 791      | -21%  |

Using multiple registers improves accuracy significantly.

### Complexity

| Operation | Time   | Space  |
|-----------|--------|--------|
| Insert    | $O(1)$ | $O(1)$ |
| Query     | $O(1)$ | $O(1)$ |

Flajolet–Martin Algorithm, the original spark of probabilistic counting, turning randomness into estimation magic.

## 268 MinHash

A MinHash is a probabilistic algorithm for estimating set similarity, particularly the Jaccard similarity, without comparing all elements directly. Instead of storing every element, it constructs a signature of compact hash values. The more overlap in signatures, the more similar the sets.

MinHash is foundational in large-scale similarity estimation, fast, memory-efficient, and mathematically elegant.

### What Problem Are We Solving?

To compute the exact Jaccard similarity between two sets  $A$  and  $B$ :

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

This requires comparing all elements, which is infeasible for large datasets.

We need methods that provide:

- Compact sketches using small memory
- Fast approximate comparisons
- Adjustable accuracy through the number of hash functions

MinHash meets these requirements by applying multiple random hash functions and recording the minimum hash value from each function as the set's signature.

### How It Works (Plain Language)

For each set, we apply several independent hash functions. For each hash function, we record the minimum hash value among all elements.

Two sets that share many elements tend to share the same minimums across these hash functions. Thus, the fraction of matching minimums estimates their Jaccard similarity.

Algorithm:

1. Choose  $k$  hash functions  $h_1, h_2, \dots, h_k$ .
2. For each set  $S$ , compute its signature:

$$\text{sig}(S)[i] = \min_{x \in S} h_i(x)$$

3. Estimate the Jaccard similarity by:

$$\hat{J}(A, B) = \frac{1}{k} \sum_{i=1}^k \mathbf{1}\{\text{sig}(A)[i] = \text{sig}(B)[i]\}$$

Each matching position between  $\text{sig}(A)$  and  $\text{sig}(B)$  corresponds to one agreeing hash function, and the fraction of matches approximates  $J(A, B)$ .

### Example (Step by Step)

Let:

$$A = \text{apple, banana, cherry}, \quad B = \text{banana, cherry, date}$$

Suppose we have 3 hash functions:

| Item   | $h_1$ | $h_2$ | $h_3$ |
|--------|-------|-------|-------|
| apple  | 5     | 1     | 7     |
| banana | 2     | 4     | 3     |
| cherry | 3     | 2     | 1     |
| date   | 4     | 3     | 2     |

Signature(A):

$$\text{sig}(A) = [\min(5, 2, 3), \min(1, 4, 2), \min(7, 3, 1)] = [2, 1, 1]$$

Signature(B):

$$\text{sig}(B) = [\min(2, 3, 4), \min(4, 2, 3), \min(3, 1, 2)] = [2, 2, 1]$$

Compare element-wise:

Matches occur at positions 1 and 3  $\rightarrow \frac{2}{3} \approx 0.67$

Actual Jaccard similarity:

$$J(A, B) = \frac{|\{\text{banana, cherry}\}|}{|\{\text{apple, banana, cherry, date}\}|} = \frac{2}{4} = 0.5$$

The MinHash estimate of 0.67 is reasonably close to the true value 0.5, demonstrating that even a small number of hash functions can yield a good approximation.

## Visualization

| Hash Function | $\text{sig}(A)$ | $\text{sig}(B)$ | Match        |
|---------------|-----------------|-----------------|--------------|
| $h_1$         | 2               | 2               |              |
| $h_2$         | 1               | 2               |              |
| $h_3$         | 1               | 1               |              |
| Similarity    | —               | —               | (2/3 = 0.67) |

## Tiny Code (Easy Version)

```
import mmh3
import math

def minhash_signature(elements, num_hashes=5, seed=42):
    sig = [math.inf] * num_hashes
    for x in elements:
        for i in range(num_hashes):
            h = mmh3.hash(str(x), seed + i)
            if h < sig[i]:
                sig[i] = h
    return sig

def jaccard_minhash(sigA, sigB):
    matches = sum(1 for a, b in zip(sigA, sigB) if a == b)
    return matches / len(sigA)

# Example
A = {"apple", "banana", "cherry"}
B = {"banana", "cherry", "date"}

sigA = minhash_signature(A, 10)
sigB = minhash_signature(B, 10)
print("Approx similarity:", jaccard_minhash(sigA, sigB))
```

## Output

Approx similarity: 0.6

## Why It Matters

- Scalable similarity: enables fast comparison of very large sets
- Compact representation: stores only  $k$  integers per set
- Composable: supports set unions using componentwise minimum
- Common applications:
  - Document deduplication
  - Web crawling and search indexing
  - Recommendation systems
  - Large-scale clustering

## A Gentle Proof (Why It Works)

For a random permutation  $h$ :

$$P[\min(h(A)) = \min(h(B))] = J(A, B)$$

Each hash function behaves like a Bernoulli trial with success probability  $J(A, B)$ .  
The MinHash estimator is unbiased:

$$E[\hat{J}] = J(A, B)$$

The variance decreases as  $\frac{1}{k}$ ,  
so increasing the number of hash functions improves accuracy.

## Try It Yourself

1. Choose two sets with partial overlap.
2. Generate MinHash signatures using  $k = 20$  hash functions.
3. Compute both the estimated and true Jaccard similarities.

4. Increase  $k$  and observe how the estimated similarity converges toward the true value — larger  $k$  reduces variance and improves accuracy.

## Test Cases

| Sets                               | True $J(A, B)$ | $k$ | Estimated | Error |
|------------------------------------|----------------|-----|-----------|-------|
| $A = \{1, 2, 3\}, B = \{2, 3, 4\}$ | 0.5            | 10  | 0.6       | +0.1  |
| $A = \{1, 2\}, B = \{1, 2, 3, 4\}$ | 0.5            | 20  | 0.45      | -0.05 |
| $A = B$                            | 1.0            | 10  | 1.0       | 0.0   |

## Complexity

| Operation       | Time            | Space  |
|-----------------|-----------------|--------|
| Build Signature | $O(n \times k)$ | $O(k)$ |
| Compare         | $O(k)$          | $O(k)$ |

MinHash turns set similarity into compact signatures — a small sketch that captures the essence of large sets with statistical grace.

## 269 Reservoir Sampling

Reservoir Sampling is a classic algorithm for randomly sampling  $k$  elements from a stream of unknown or very large size, ensuring every element has an equal probability of being selected.

It's the perfect tool when you can't store everything, like catching a few fish from an endless river, one by one, without bias.

### What Problem Are We Solving?

When data arrives as a stream too large to store in memory, we cannot know its total size in advance.

Yet, we often need to maintain a uniform random sample of fixed size  $k$ .

A naive approach would store all items and then sample, but this becomes infeasible for large or unbounded data.

Reservoir Sampling provides a one-pass solution with these guarantees:

- Each item in the stream has equal probability  $\frac{k}{n}$  of being included
- Uses only  $O(k)$  memory
- Processes data in a single pass

## How It Works (Plain Language)

We maintain a reservoir (array) of size  $k$ .

As each new element arrives, we decide probabilistically whether it replaces one of the existing items.

Steps:

1. Fill the reservoir with the first  $k$  elements.
2. For each element at index  $i$  (starting from  $i = k + 1$ ):
  - Generate a random integer  $j \in [1, i]$
  - If  $j \leq k$ , replace  $\text{reservoir}[j]$  with the new element

This ensures every element has an equal chance  $\frac{k}{n}$  to remain.

## Example (step by step)

Stream: [A, B, C, D, E]

Goal:  $k = 2$

1. Start with the first 2  $\rightarrow$  [A, B]
2.  $i = 3$ , item = C
  - Pick random  $j \in [1, 3]$
  - Suppose  $j = 2 \rightarrow$  replace B  $\rightarrow$  [A, C]
3.  $i = 4$ , item = D
  - Pick random  $j \in [1, 4]$
  - Suppose  $j = 4 \rightarrow$  do nothing  $\rightarrow$  [A, C]

4.  $i = 5$ , item = E

- Pick random  $j \in [1, 5]$
- Suppose  $j = 1 \rightarrow$  replace A  $\rightarrow$  [E, C]

Final sample: [E, C]

Each item A–E has equal probability to appear in the final reservoir.

## Mathematical Intuition

Each element  $x_i$  at position  $i$  has probability

$$P(x_i \text{ in final sample}) = \frac{k}{i} \cdot \prod_{j=i+1}^n \left(1 - \frac{1}{j}\right) = \frac{k}{n}$$

Thus, every item is equally likely to be chosen, ensuring perfect uniformity in the final sample.

## Visualization

| Step | Item | Random $j$ | Action     | Reservoir |
|------|------|------------|------------|-----------|
| 1    | A    | –          | Add        | [A]       |
| 2    | B    | –          | Add        | [A, B]    |
| 3    | C    | 2          | Replace B  | [A, C]    |
| 4    | D    | 4          | No Replace | [A, C]    |
| 5    | E    | 1          | Replace A  | [E, C]    |

## Tiny Code (Easy Version)

Python Implementation

```
import random

def reservoir_sample(stream, k):
    reservoir = []
    for i, item in enumerate(stream, 1):
        if i <= k:
            reservoir.append(item)
        else:
            j = random.randint(1, i)
```

```

        if j <= k:
            reservoir[j - 1] = item
    return reservoir

# Example
stream = ["A", "B", "C", "D", "E"]
sample = reservoir_sample(stream, 2)
print("Reservoir sample:", sample)

```

Output (random):

```
Reservoir sample: ['E', 'C']
```

Each run produces a different uniform random sample.

## Why It Matters

- Works on streaming data
- Needs only  $O(k)$  memory
- Provides uniform unbiased sampling
- Used in:
  - Big data analytics
  - Randomized algorithms
  - Online learning
  - Network monitoring

## A Gentle Proof (Why It Works)

- First  $k$  elements: probability = 1 to enter the reservoir initially.
- Each new element at index  $i$ : probability  $\frac{k}{i}$  to replace one of the existing items.
- Earlier items may be replaced, but each remains with probability

$$P(\text{survive}) = \prod_{j=i+1}^n \left(1 - \frac{1}{j}\right)$$

Multiplying these terms gives the final inclusion probability  $\frac{k}{n}$ .

Uniformity of selection is guaranteed by induction.

### Try It Yourself

1. Stream 10 numbers with  $k = 3$ .
2. Run the algorithm multiple times — all 3-element subsets appear with roughly equal frequency.
3. Increase  $k$  and observe that the sample becomes more stable, with less variation between runs.

### Test Cases

| Stream      | k  | Sample Size | Notes                |
|-------------|----|-------------|----------------------|
| [1,2,3,4,5] | 2  | 2           | Uniform random pairs |
| [A,B,C,D]   | 1  | 1           | Each 25% chance      |
| Range(1000) | 10 | 10          | Works in one pass    |

### Complexity

| Operation | Time   | Space  |
|-----------|--------|--------|
| Insert    | $O(1)$ | $O(k)$ |
| Query     | $O(1)$ | $O(k)$ |

Reservoir Sampling, elegantly fair, perfectly simple, and ready for infinite streams.

## 270 Skip Bloom Filter

A Skip Bloom Filter is a probabilistic data structure that extends the Bloom Filter to support range queries, determining whether any element exists within a given interval, not just checking for a single item.

It combines Bloom filters with hierarchical range segmentation, allowing approximate range lookups while keeping space usage compact.

### What Problem Are We Solving?

A classic Bloom Filter answers only point queries:

"Is  $x$  in the set?"

However, many real-world applications require range queries, such as:

- Databases: "Are there any keys between 10 and 20?"
- Time-series: "Were there any events during this time interval?"
- Networks: "Is any IP in this subnet?"

We need a space-efficient, stream-friendly, and probabilistic structure that can:

- Handle range membership checks,
- Maintain a low false-positive rate,
- Scale logarithmically with the universe size.

The Skip Bloom Filter solves this by layering Bloom filters over aligned ranges of increasing size.

### How It Works (Plain Language)

A Skip Bloom Filter maintains multiple Bloom filters, each corresponding to a level that covers intervals (buckets) of sizes  $2^0, 2^1, 2^2, \dots$

Each element is inserted into all Bloom filters representing the ranges that contain it. When querying a range, the query is decomposed into aligned subranges that correspond to these levels, and each is checked in its respective filter.

Algorithm:

1. Divide the universe into intervals of size  $2^\ell$  for each level  $\ell$ .

2. Each level  $\ell$  maintains a Bloom filter representing those buckets.
3. To insert a key  $x$ : mark all buckets across levels that include  $x$ .
4. To query a range  $[a, b]$ : decompose it into a set of disjoint aligned intervals and check the corresponding Bloom filters.

### Example (Step by Step)

Suppose we store keys

$$S = 3, 7, 14$$

in a universe  $([0, 15])$ . We build filters at levels with range sizes  $(1, 2, 4, 8)$ :

| Level | Bucket Size | Buckets (Ranges)                      |
|-------|-------------|---------------------------------------|
| 0     | 1           | $[0], [1], [2], \dots, [15]$          |
| 1     | 2           | $[0-1], [2-3], [4-5], \dots, [14-15]$ |
| 2     | 4           | $[0-3], [4-7], [8-11], [12-15]$       |
| 3     | 8           | $[0-7], [8-15]$                       |

Insert key 3:

- Level 0:  $[3]$
- Level 1:  $[2-3]$
- Level 2:  $[0-3]$
- Level 3:  $[0-7]$

Insert key 7:

- Level 0:  $[7]$
- Level 1:  $[6-7]$
- Level 2:  $[4-7]$
- Level 3:  $[0-7]$

Insert key 14:

- Level 0:  $[14]$
- Level 1:  $[14-15]$
- Level 2:  $[12-15]$
- Level 3:  $[8-15]$

Query range [2, 6]:

1. Decompose into aligned intervals: ([2–3], [4–5], [6])
2. Check filters:

- [2–3] → hit
- [4–5] → miss
- [6] → miss

Result: possibly non-empty, since [2–3] contains 3.

## Visualization

| Level | Bucket | Contains Key | Bloom Entry |
|-------|--------|--------------|-------------|
| 0     | [3]    | Yes          | 1           |
| 1     | [2–3]  | Yes          | 1           |
| 2     | [0–3]  | Yes          | 1           |
| 3     | [0–7]  | Yes          | 1           |

Each key is represented in multiple levels, enabling multi-scale range coverage.

## Tiny Code (Simplified Python)

```
import math, mmh3

class Bloom:
    def __init__(self, size=64, hash_count=3):
        self.size = size
        self.hash_count = hash_count
        self.bits = [0] * size

    def _hashes(self, key):
        return [mmh3.hash(str(key), i) % self.size for i in range(self.hash_count)]

    def add(self, key):
        for h in self._hashes(key):
            self.bits[h] = 1

    def query(self, key):
```

```

        return all(self.bits[h] for h in self._hashes(key))

class SkipBloom:
    def __init__(self, levels=4, size=64, hash_count=3):
        self.levels = [Bloom(size, hash_count) for _ in range(levels)]

    def add(self, key):
        level = 0
        while (1 << level) <= key:
            bucket = key // (1 << level)
            self.levels[level].add(bucket)
            level += 1

    def query_range(self, start, end):
        l = int(math.log2(end - start + 1))
        bucket = start // (1 << l)
        return self.levels[l].query(bucket)

# Example
sb = SkipBloom(levels=4)
for x in [3, 7, 14]:
    sb.add(x)

print("Query [2,6]:", sb.query_range(2,6))

```

Output:

Query [2,6]: True

## Why It Matters

- Enables range queries in probabilistic manner
- Compact and hierarchical
- No false negatives (for properly configured filters)
- Widely applicable in:
  - Approximate database indexing
  - Network prefix search
  - Time-series event detection

## A Gentle Proof (Why It Works)

Each inserted key participates in  $O(\log U)$  Bloom filters, one per level.  
A range query  $[a, b]$  is decomposed into  $O(\log U)$  aligned subranges.

A Bloom filter with  $m$  bits,  $k$  hash functions, and  $n$  inserted elements has false positive probability:

$$p = (1 - e^{-kn/m})^k$$

For a Skip Bloom Filter, the total false positive rate is bounded by:

$$P_{fp} \leq O(\log U) \cdot p$$

Each level guarantees no false negatives, since every range containing an element is marked.  
Thus, correctness is ensured, and only overestimation (false positives) can occur.

Space complexity

Each level has  $m$  bits, and there are  $\log U$  levels:

$$\text{Total space} = O(m \log U)$$

Time complexity

Each query checks  $O(\log U)$  buckets, each requiring  $O(k)$  time:

$$T_{\text{query}} = O(k \log U)$$

## Try It Yourself

1. Insert keys  $\{3, 7, 14\}$
2. Query ranges  $[2, 6]$ ,  $[8, 12]$ ,  $[0, 15]$
3. Compare true contents with the results
4. Adjust parameters  $m$ ,  $k$ , or the number of levels, and observe how the false positive rate changes

## Test Cases

| Query Range | Result | True Contents |
|-------------|--------|---------------|
| [2,6]       | True   | {3}           |
| [8,12]      | False  |               |
| [12,15]     | True   | {14}          |

## Complexity

| Operation | Time        | Space         |
|-----------|-------------|---------------|
| Insert    | $O(\log U)$ | $O(m \log U)$ |
| Query     | $O(\log U)$ | $O(m \log U)$ |

A Skip Bloom Filter is a range-aware extension of standard Bloom filters.

By combining hierarchical decomposition with standard hashing, it enables fast, memory-efficient, and approximate range queries across very large universes.

## Section 28. Skip Lists and B-Trees

### 271 Skip List Insert

Skip Lists are probabilistic alternatives to balanced trees. They maintain multiple levels of sorted linked lists, where each level skips over more elements than the one below it. Insertion relies on randomization to achieve expected  $O(\log n)$  search and update times, without strict rebalancing like AVL or Red-Black trees.

#### What Problem Are We Solving?

We want to store elements in sorted order and support:

- Fast search, insert, and delete operations.
- Simple structure and easy implementation.
- Expected logarithmic performance without complex rotations.

Balanced BSTs achieve  $O(\log n)$  time but require intricate rotations. Skip Lists solve this with *randomized promotion*: each inserted node is promoted to higher levels with decreasing probability, forming a tower.

## How It Works (Plain Language)

Skip list insertion for value  $x$

Steps 1. Start at the top level. 2. While the next node's key  $< x$ , move right. 3. If you cannot move right, drop down one level. 4. Repeat until you reach level 0. 5. Insert the new node at its sorted position on level 0. 6. Randomly choose the node height  $h$  (for example, flip a fair coin per level until tails, so  $P(h \geq t) = 2^{-t}$ ). 7. For each level  $1 \dots h$ , link the new node into that level by repeating the right-then-drop search with the update pointers saved from the descent.

Notes - Duplicate handling is policy dependent. Often you skip insertion if a node with key  $x$  already exists. - Typical height is  $O(\log n)$ , expected search and insert time is  $O(\log n)$ , space is  $O(n)$ .

## Example Step by Step

Let's insert  $x = 17$  into a skip list that currently contains: [ 5, 10, 15, 20, 25 ]

| Level | Nodes (before insertion) | Path Taken                         |
|-------|--------------------------|------------------------------------|
| 3     | 5 → 15 → 25              | Move from 5 to 15, then down       |
| 2     | 5 → 10 → 15 → 25         | Move from 5 to 10 to 15, then down |
| 1     | 5 → 10 → 15 → 20 → 25    | Move from 15 to 20, then down      |
| 0     | 5 → 10 → 15 → 20 → 25    | Insert after 15                    |

Suppose the random level for 17 is 2. We insert 17 at level 0 and level 1.

| Level | Nodes (after insertion)    |
|-------|----------------------------|
| 3     | 5 → 15 → 25                |
| 2     | 5 → 10 → 15 → 25           |
| 1     | 5 → 10 → 15 → 17 → 20 → 25 |
| 0     | 5 → 10 → 15 → 17 → 20 → 25 |

## Tiny Code (Simplified Python)

```
import random

class Node:
    def __init__(self, key, level):
```

```

        self.key = key
        self.forward = [None] * (level + 1)

class SkipList:
    def __init__(self, max_level=4, p=0.5):
        self.max_level = max_level
        self.p = p
        self.header = Node(-1, max_level)
        self.level = 0

    def random_level(self):
        lvl = 0
        while random.random() < self.p and lvl < self.max_level:
            lvl += 1
        return lvl

    def insert(self, key):
        update = [None] * (self.max_level + 1)
        current = self.header

        # Move right and down
        for i in reversed(range(self.level + 1)):
            while current.forward[i] and current.forward[i].key < key:
                current = current.forward[i]
            update[i] = current

        current = current.forward[0]
        if current is None or current.key != key:
            lvl = self.random_level()
            if lvl > self.level:
                for i in range(self.level + 1, lvl + 1):
                    update[i] = self.header
                self.level = lvl
            new_node = Node(key, lvl)
            for i in range(lvl + 1):
                new_node.forward[i] = update[i].forward[i]
                update[i].forward[i] = new_node

```

## Why It Matters

- Expected  $O(\log n)$  time for search, insert, and delete
- Simpler than AVL or Red-Black Trees
- Probabilistic balancing avoids rigid rotations
- Commonly used in databases and key-value stores such as LevelDB and Redis

## A Gentle Proof (Why It Works)

Each node appears in level  $i$  with probability  $p^i$ .

The expected number of nodes per level is  $np^i$ .

The total number of levels is  $O(\log_{1/p} n)$ .

Expected search path length:

$$E[\text{steps}] = \frac{1}{1-p} \log_{1/p} n = O(\log n)$$

Expected space usage:

$$O(n) \text{ nodes} \times O\left(\frac{1}{1-p}\right) \text{ pointers per node}$$

Thus, Skip Lists achieve expected logarithmic performance and linear space.

## Try It Yourself

1. Build a Skip List and insert  $\{5, 10, 15, 20, 25\}$ .
2. Insert 17 and trace which pointers are updated at each level.
3. Experiment with  $p = 0.25, 0.5, 0.75$ .
4. Observe how random heights influence the overall balance.

## Test Cases

| Operation | Input | Expected Structure (Level 0) |
|-----------|-------|------------------------------|
| Insert    | 10    | 10                           |
| Insert    | 5     | 5 → 10                       |
| Insert    | 15    | 5 → 10 → 15                  |
| Insert    | 17    | 5 → 10 → 15 → 17             |
| Search    | 15    | Found                        |
| Search    | 12    | Not Found                    |

## Complexity

| Operation | Time (Expected) | Space  |
|-----------|-----------------|--------|
| Search    | $O(\log n)$     | $O(n)$ |
| Insert    | $O(\log n)$     | $O(n)$ |
| Delete    | $O(\log n)$     | $O(n)$ |

A Skip List is a simple yet powerful data structure. With randomness as its balancing force, it achieves the elegance of trees and the flexibility of linked lists.

## 272 Skip List Delete

Deletion in a Skip List mirrors insertion: we traverse levels from top to bottom, keep track of predecessor nodes at each level, and unlink the target node across all levels it appears in. The structure maintains probabilistic balance, so no rebalancing is needed, deletion is expected  $O(\log n)$ .

### What Problem Are We Solving?

We want to remove an element efficiently from a sorted, probabilistically balanced structure. Naive linked lists require  $O(n)$  traversal; balanced BSTs need complex rotations. A Skip List gives us a middle ground, simple pointer updates with expected logarithmic time.

### How It Works (Plain Language)

Each node in a skip list can appear at multiple levels. To delete a key  $x$ :

1. Start from the top level.
2. Move right while next node's key  $< x$ .

3. If next node's key ==  $x$ , record current node in an `update` array.
4. Drop one level down and repeat.
5. Once you reach the bottom, remove all forward references to the node from the `update` array.
6. If the topmost level becomes empty, reduce list level.

### Example Step by Step

Delete  $x = 17$  from this skip list:

| Level | Nodes (Before)             |
|-------|----------------------------|
| 3     | 5 → 15 → 25                |
| 2     | 5 → 10 → 15 → 25           |
| 1     | 5 → 10 → 15 → 17 → 20 → 25 |
| 0     | 5 → 10 → 15 → 17 → 20 → 25 |

Traversal:

- Start at Level 3:  $15 < 17 \rightarrow$  move right  $\rightarrow 25 > 17 \rightarrow$  drop down
- Level 2:  $15 < 17 \rightarrow$  move right  $\rightarrow 25 > 17 \rightarrow$  drop down
- Level 1:  $15 < 17 \rightarrow$  move right  $\rightarrow 17$  found  $\rightarrow$  record predecessor
- Level 0:  $15 < 17 \rightarrow$  move right  $\rightarrow 17$  found  $\rightarrow$  record predecessor

Remove all forward pointers to 17 from recorded nodes.

| Level | Nodes (After)         |
|-------|-----------------------|
| 3     | 5 → 15 → 25           |
| 2     | 5 → 10 → 15 → 25      |
| 1     | 5 → 10 → 15 → 20 → 25 |
| 0     | 5 → 10 → 15 → 20 → 25 |

### Tiny Code (Simplified Python)

```
import random

class Node:
    def __init__(self, key, level):
        self.key = key
        self.forward = [None] * (level + 1)
```

```

class SkipList:
    def __init__(self, max_level=4, p=0.5):
        self.max_level = max_level
        self.p = p
        self.header = Node(-1, max_level)
        self.level = 0

    def delete(self, key):
        update = [None] * (self.max_level + 1)
        current = self.header

        # Traverse from top to bottom
        for i in reversed(range(self.level + 1)):
            while current.forward[i] and current.forward[i].key < key:
                current = current.forward[i]
            update[i] = current

        current = current.forward[0]

        # Found the node
        if current and current.key == key:
            for i in range(self.level + 1):
                if update[i].forward[i] != current:
                    continue
                update[i].forward[i] = current.forward[i]
            # Reduce level if highest level empty
            while self.level > 0 and self.header.forward[self.level] is None:
                self.level -= 1

```

## Why It Matters

- Symmetric to insertion
- No rotations or rebalancing
- Expected  $O(\log n)$  performance
- Perfect for ordered maps, databases, key-value stores

## A Gentle Proof (Why It Works)

Each level contains a fraction  $p^i$  of nodes. The expected number of levels traversed is  $O(\log_{1/p} n)$ . At each level, we move horizontally  $O(1)$  on average.

So expected cost:

$$E[T_{\text{delete}}] = O(\log n)$$

### Try It Yourself

1. Insert 5, 10, 15, 17, 20, 25.
2. Delete 17.
3. Trace all pointer changes level by level.
4. Compare with AVL tree deletion complexity.

### Test Cases

| Operation | Input         | Expected Level 0 Result |
|-----------|---------------|-------------------------|
| Insert    | 5,10,15,17,20 | 5 → 10 → 15 → 17 → 20   |
| Delete    | 17            | 5 → 10 → 15 → 20        |
| Delete    | 10            | 5 → 15 → 20             |
| Delete    | 5             | 15 → 20                 |

### Complexity

| Operation | Time (Expected) | Space  |
|-----------|-----------------|--------|
| Search    | $O(\log n)$     | $O(n)$ |
| Insert    | $O(\log n)$     | $O(n)$ |
| Delete    | $O(\log n)$     | $O(n)$ |

Skip List Deletion keeps elegance through simplicity, a clean pointer adjustment instead of tree surgery.

### 273 Skip List Search

Searching in a Skip List is a dance across levels, we move right until we can't, then down, repeating until we either find the key or conclude it doesn't exist. Thanks to the randomized level structure, the expected time complexity is  $O(\log n)$ , just like balanced BSTs but with simpler pointer logic.

## What Problem Are We Solving?

We need a fast search in a sorted collection that adapts gracefully to dynamic insertions and deletions. Balanced trees guarantee  $O(\log n)$  search but need rotations. Skip Lists achieve the same expected time using randomization instead of strict balancing rules.

## How It Works (Plain Language)

A skip list has multiple levels of linked lists. Each level acts as a fast lane, skipping over multiple nodes. To search for a key  $x$ :

1. Start at the top-left header node.
2. At each level, move right while `next.key < x`.
3. When `next.key == x`, drop down one level.
4. Repeat until level 0.
5. If `current.forward[0].key == x`, found; else, not found.

The search path “zigzags” through levels, visiting roughly  $\log n$  nodes on average.

## Example Step by Step

Search for  $x = 17$  in the following skip list:

| Level | Nodes                      |
|-------|----------------------------|
| 3     | 5 → 15 → 25                |
| 2     | 5 → 10 → 15 → 25           |
| 1     | 5 → 10 → 15 → 17 → 20 → 25 |
| 0     | 5 → 10 → 15 → 17 → 20 → 25 |

Traversal:

- Level 3: 5 → 15 → (`next = 25 > 17`) → drop down
- Level 2: 15 → (`next = 25 > 17`) → drop down
- Level 1: 15 → 17 found → success
- Level 0: confirm 17 exists

Path: 5 → 15 → (down) → 15 → (down) → 15 → 17

## Visualization

Skip list search follows a staircase pattern:

```
Level 3: 5 -----> 15 ----↓  
Level 2: 5 ----> 10 --> 15 --↓  
Level 1: 5 -> 10 -> 15 -> 17 -> 20  
Level 0: 5 -> 10 -> 15 -> 17 -> 20
```

Each “↓” means dropping a level when the next node is too large.

## Tiny Code (Simplified Python)

```
class SkipList:  
    def __init__(self, max_level=4, p=0.5):  
        self.max_level = max_level  
        self.p = p  
        self.header = Node(-1, max_level)  
        self.level = 0  
  
    def search(self, key):  
        current = self.header  
        # Traverse top-down  
        for i in reversed(range(self.level + 1)):  
            while current.forward[i] and current.forward[i].key < key:  
                current = current.forward[i]  
        current = current.forward[0]  
        return current and current.key == key
```

## Why It Matters

- Simple and efficient: expected  $O(\log n)$  time
- Probabilistic balance: avoids tree rotations
- Foundation for ordered maps, indexes, and databases
- Search path length is logarithmic on average

## A Gentle Proof (Why It Works)

Each level contains approximately a fraction  $p$  of the nodes from the level below. Expected number of levels:  $O(\log_{1/p} n)$ .

At each level, expected number of horizontal moves:  $O(1)$ .

So total expected search time:

$$E[T_{\text{search}}] = O(\log n)$$

## Try It Yourself

1. Build a skip list with 5, 10, 15, 17, 20, 25.
2. Search for 17 and trace the path at each level.
3. Search for 13, where do you stop?
4. Compare path length with a binary search tree of same size.

## Test Cases

| Operation | Input | Expected Output | Path        |
|-----------|-------|-----------------|-------------|
| Search    | 17    | Found           | 5 → 15 → 17 |
| Search    | 10    | Found           | 5 → 10      |
| Search    | 13    | Not Found       | 5 → 10 → 15 |
| Search    | 5     | Found           | 5           |

## Complexity

| Operation | Time (Expected) | Space  |
|-----------|-----------------|--------|
| Search    | $O(\log n)$     | $O(n)$ |
| Insert    | $O(\log n)$     | $O(n)$ |
| Delete    | $O(\log n)$     | $O(n)$ |

Skip List Search shows how probabilistic structure yields deterministic-like efficiency, walking a staircase of randomness toward certainty.

## 274 B-Tree Insert

A B-Tree is a balanced search tree designed for external memory systems such as disks or SSDs. Unlike binary trees, each node can store multiple keys and multiple children, minimizing disk I/O by packing more data into a single node. Insertion into a B-Tree preserves sorted order and balance by splitting full nodes as needed.

### What Problem Are We Solving?

When data is too large to fit in memory, standard binary trees perform poorly because each node access may trigger a disk read. We need a structure that:

- Reduces the number of I/O operations
- Keeps height small
- Maintains keys in sorted order
- Supports search, insert, and delete in  $O(\log n)$

B-Trees solve this by storing many keys per node and balancing themselves through controlled splits.

### How It Works (Plain Language)

Each B-Tree node can contain up to  $2t - 1$  keys and  $2t$  children, where  $t$  is the minimum degree.

Insertion steps:

1. Start at root and traverse down like in binary search.
2. If a child node is full ( $2t - 1$  keys), split it before descending.
3. Insert the new key into the appropriate non-full node.

Splitting a full node:

- Middle key moves up to parent
- Left and right halves become separate child nodes

This ensures every node stays within allowed size bounds, keeping height  $O(\log_t n)$ .

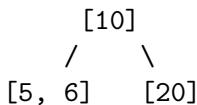
## Example Step by Step

Let  $t = 2$  (max 3 keys per node). Insert keys in order: [10, 20, 5, 6, 12, 30, 7, 17]

Step 1: Insert 10 → Root = [10] Step 2: Insert 20 → [10, 20] Step 3: Insert 5 → [5, 10, 20]  
Step 4: Insert 6 → Node full → Split

Split [5, 6, 10, 20]:

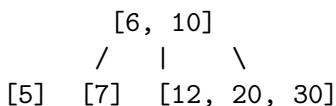
- Middle key 10 moves up
- Left child [5, 6], right child [20] Tree:



Step 5: Insert 12 → go right → [12, 20] Step 6: Insert 30 → [12, 20, 30] Step 7: Insert 7 → go left → [5, 6, 7] → full → split

- Middle 6 moves up

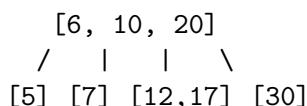
Tree now:



Step 8: Insert 17 → go to [12, 20, 30] → insert [12, 17, 20, 30] → split

- Middle 20 moves up

Final tree:



## Visualization

B-Tree maintains sorted keys at each level and guarantees minimal height by splitting nodes during insertion.

Root: [6, 10, 20]  
Children: [5], [7], [12, 17], [30]

## Tiny Code (Simplified Python)

```
class BTreeNode:
    def __init__(self, t, leaf=False):
        self.keys = []
        self.children = []
        self.leaf = leaf
        self.t = t

    def insert_non_full(self, key):
        i = len(self.keys) - 1
        if self.leaf:
            self.keys.append(key)
            self.keys.sort()
        else:
            while i >= 0 and key < self.keys[i]:
                i -= 1
            i += 1
            if len(self.children[i].keys) == 2 * self.t - 1:
                self.split_child(i)
                if key > self.keys[i]:
                    i += 1
                self.children[i].insert_non_full(key)

    def split_child(self, i):
        t = self.t
        y = self.children[i]
        z = BTreeNode(t, y.leaf)
        mid = y.keys[t - 1]
        z.keys = y.keys[t:]
        y.keys = y.keys[:t - 1]
        if not y.leaf:
            z.children = y.children[t:]
            y.children = y.children[:t]
        self.children.insert(i + 1, z)
        self.keys.insert(i, mid)

class BTree:
    def __init__(self, t):
        self.root = BTreeNode(t, True)
        self.t = t
```

```

def insert(self, key):
    r = self.root
    if len(r.keys) == 2 * self.t - 1:
        s = BTreeNode(self.t)
        s.children.insert(0, r)
        s.split_child(0)
        i = 0
        if key > s.keys[0]:
            i += 1
        s.children[i].insert_non_full(key)
        self.root = s
    else:
        r.insert_non_full(key)

```

## Why It Matters

- Disk-friendly: each node fits into one page
- Shallow height:  $O(\log_t n)$  levels → few disk reads
- Deterministic balance: no randomness, always balanced
- Foundation of file systems, databases, indexes (e.g., NTFS, MySQL, PostgreSQL)

## A Gentle Proof (Why It Works)

Each node has between  $t - 1$  and  $2t - 1$  keys (except root).

Each split increases height only when the root splits.

Thus, height  $h$  satisfies:

$$t^h \leq n \leq (2t)^h$$

Taking logs:

$$h = O(\log_t n)$$

So insertions and searches take  $O(t \cdot \log_t n)$ , often simplified to  $O(\log n)$  when  $t$  is constant.

## Try It Yourself

1. Build a B-Tree with  $t = 2$ .
2. Insert  $[10, 20, 5, 6, 12, 30, 7, 17]$ .
3. Draw the tree after each insertion.
4. Observe when splits occur and which keys promote upward.

## Test Cases

| Input Keys                      | $t$ | Final Root    | Height |
|---------------------------------|-----|---------------|--------|
| $[10, 20, 5, 6, 12, 30, 7, 17]$ | 2   | $[6, 10, 20]$ | 2      |
| $[1, 2, 3, 4, 5, 6, 7, 8, 9]$   | 2   | $[4]$         | 3      |

## Complexity

| Operation | Time        | Space  |
|-----------|-------------|--------|
| Search    | $O(\log n)$ | $O(n)$ |
| Insert    | $O(\log n)$ | $O(n)$ |
| Delete    | $O(\log n)$ | $O(n)$ |

B-Tree insertion is the heartbeat of external memory algorithms, split, promote, balance, ensuring data stays close, shallow, and sorted.

## 275 B-Tree Delete

Deletion in a B-Tree is more intricate than insertion, we must carefully remove a key while preserving the B-Tree's balance properties. Every node must maintain at least  $t - 1$  keys (except the root), so deletion may involve borrowing from siblings or merging nodes.

The goal is to maintain the B-Tree invariants:

- Keys sorted within each node
- Node key count between  $t - 1$  and  $2t - 1$
- Balanced height

## What Problem Are We Solving?

We want to remove a key from a B-Tree without violating balance or occupancy constraints. Unlike binary search trees, where we can simply replace or prune nodes, B-Trees must maintain minimum degree to ensure consistent height and I/O efficiency.

## How It Works (Plain Language)

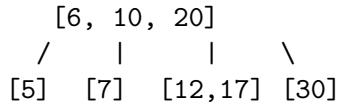
To delete a key  $k$  from a B-Tree:

1. If  $k$  is in a leaf node:
  - Simply remove it.
2. If  $k$  is in an internal node:
  - Case A: If left child has  $t$  keys  $\rightarrow$  replace  $k$  with predecessor.
  - Case B: Else if right child has  $t$  keys  $\rightarrow$  replace  $k$  with successor.
  - Case C: Else both children have  $t - 1$  keys  $\rightarrow$  merge them and recurse.
3. If  $k$  is not in the current node:
  - Move to the correct child.
  - Before descending, ensure the child has at least  $t$  keys. If not,
    - Borrow from a sibling with  $t$  keys, or
    - Merge with a sibling to guarantee occupancy.

This ensures no underflow occurs during traversal.

## Example Step by Step

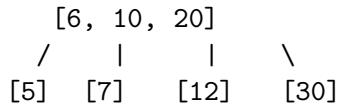
Let  $t = 2$  (max 3 keys per node). B-Tree before deletion:



### Delete key 17

- 17 is in leaf [12, 17] → remove it directly

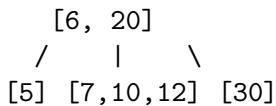
Result:



### Delete key 10

- 10 is in internal node
- Left child [7] has  $t - 1 = 1$  key
- Right child [12] also has 1 key → Merge [7], 10, [12] → [7, 10, 12]

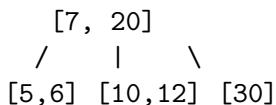
Tree becomes:



### Delete key 6

- 6 in internal node
- Left [5] has 1 key, right [7,10,12] has 2 → borrow from right
- Replace 6 with successor 7

Tree after rebalancing:



### Visualization

Every deletion keeps the tree balanced by ensuring all nodes (except root) stay  $t - 1$  full.

|                         |                    |
|-------------------------|--------------------|
| Before:                 | After Deleting 10: |
| [\$\$6,10,20]           | [6,20]             |
| /     \                 | /   \              |
| \$\$5] [7] [12,17] [30] | [5] [7,10,12] [30] |

## Tiny Code (Simplified Python)

```
class BTreeNode:
    def __init__(self, t, leaf=False):
        self.t = t
        self.keys = []
        self.children = []
        self.leaf = leaf

    def find_key(self, k):
        for i, key in enumerate(self.keys):
            if key >= k:
                return i
        return len(self.keys)

class BTree:
    def __init__(self, t):
        self.root = BTreeNode(t, True)
        self.t = t

    def delete(self, node, k):
        t = self.t
        i = node.find_key(k)

        # Case 1: key in node
        if i < len(node.keys) and node.keys[i] == k:
            if node.leaf:
                node.keys.pop(i)
            else:
                if len(node.children[i].keys) >= t:
                    pred = self.get_predecessor(node, i)
                    node.keys[i] = pred
                    self.delete(node.children[i], pred)
                elif len(node.children[i+1].keys) >= t:
                    succ = self.get_successor(node, i)
                    node.keys[i] = succ
                    self.delete(node.children[i+1], succ)
                else:
                    self.merge(node, i)
                    self.delete(node.children[i], k)
            else:
                # Case 2: key not in node
```

```

if node.leaf:
    return # not found
if len(node.children[i].keys) < t:
    self.fill(node, i)
self.delete(node.children[i], k)

```

*(Helper methods merge, fill, borrow\_from\_prev, and borrow\_from\_next omitted for brevity)*

## Why It Matters

- Maintains balanced height after deletions
- Prevents underflow in child nodes
- Ensures  $O(\log n)$  complexity
- Used heavily in databases and filesystems where stable performance is critical

## A Gentle Proof (Why It Works)

A B-Tree node always satisfies:

$$t - 1 \leq \text{keys per node} \leq 2t - 1$$

Merging or borrowing ensures all nodes remain within bounds. The height  $h$  satisfies:

$$h \leq \log_t n$$

So deletions require visiting at most  $O(\log_t n)$  nodes and performing constant-time merges/borrows per level.

Hence:

$$T_{\text{delete}} = O(\log n)$$

## Try It Yourself

1. Build a B-Tree with  $t = 2$ .
2. Insert [10, 20, 5, 6, 12, 30, 7, 17].
3. Delete keys in order: 17, 10, 6.
4. Draw the tree after each deletion and observe merges/borrows.

## Test Cases

| Input Keys             | Delete | Result (Level 0)    |
|------------------------|--------|---------------------|
| [5,6,7,10,12,17,20,30] | 17     | [5,6,7,10,12,20,30] |
| [5,6,7,10,12,20,30]    | 10     | [5,6,7,12,20,30]    |
| [5,6,7,12,20,30]       | 6      | [5,7,12,20,30]      |

## Complexity

| Operation | Time        | Space  |
|-----------|-------------|--------|
| Search    | $O(\log n)$ | $O(n)$ |
| Insert    | $O(\log n)$ | $O(n)$ |
| Delete    | $O(\log n)$ | $O(n)$ |

B-Tree deletion is a surgical balancing act, merging, borrowing, and promoting keys just enough to keep the tree compact, shallow, and sorted.

## 276 B+ Tree Search

A B+ Tree is an extension of the B-Tree, optimized for range queries and sequential access. All actual data (records or values) reside in leaf nodes, which are linked together to form a sorted list. Internal nodes contain only keys that guide the search.

Searching in a B+ Tree follows the same principle as a B-Tree, top-down traversal based on key comparisons, but ends at the leaf level where the actual data is stored.

### What Problem Are We Solving?

We need a disk-friendly search structure that:

- Keeps height small (few disk I/Os)
- Supports fast range scans
- Separates index keys (internal nodes) from records (leaves)

B+ Trees meet these needs with:

- High fan-out: many keys per node
- Linked leaves: for efficient sequential traversal
- Deterministic balance: height always  $O(\log n)$

## How It Works (Plain Language)

Each internal node acts as a router. Each leaf node contains keys + data pointers.

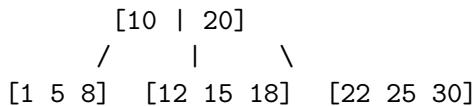
To search for key  $k$ :

1. Start at the root.
2. At each internal node, find the child whose key range contains  $k$ .
3. Follow that pointer to the next level.
4. Continue until reaching a leaf node.
5. Perform a linear scan within the leaf to find  $k$ .

If not found in the leaf,  $k$  is not in the tree.

## Example Step by Step

Let  $t = 2$  (each node holds up to 3 keys). B+ Tree:



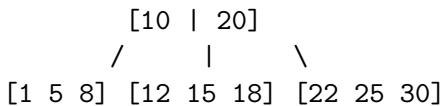
Search for 15:

- Root  $[10 \mid 20]$ :  $15 > 10$  and  $< 20 \rightarrow$  follow middle pointer
- Node  $[12 \ 15 \ 18]$ : found 15

Search for 17:

- Root  $[10 \mid 20]$ :  $17 > 10$  and  $< 20 \rightarrow$  middle pointer
- Node  $[12 \ 15 \ 18]$ : not found  $\rightarrow$  not in tree

## Visualization



- Internal nodes guide the path
- Leaf nodes hold data (and link to next leaf)
- Search always ends at a leaf

## Tiny Code (Simplified Python)

```
class BPlusNode:
    def __init__(self, t, leaf=False):
        self.t = t
        self.leaf = leaf
        self.keys = []
        self.children = []
        self.next = None # link to next leaf

class BPlusTree:
    def __init__(self, t):
        self.root = BPlusNode(t, True)
        self.t = t

    def search(self, node, key):
        if node.leaf:
            return key in node.keys
        i = 0
        while i < len(node.keys) and key >= node.keys[i]:
            i += 1
        return self.search(node.children[i], key)

    def find(self, key):
        return self.search(self.root, key)
```

## Why It Matters

- Efficient disk I/O: high branching factor keeps height low
- All data in leaves: simplifies range queries
- Linked leaves: enable sequential traversal (sorted order)
- Used in: databases, filesystems, key-value stores (e.g., MySQL, InnoDB, NTFS)

## A Gentle Proof (Why It Works)

Each internal node has between  $t$  and  $2t$  children. Each leaf holds between  $t - 1$  and  $2t - 1$  keys. Thus, height  $h$  satisfies:

$$t^h \leq n \leq (2t)^h$$

Taking logs:

$$h = O(\log_t n)$$

So search time is:

$$T_{\text{search}} = O(\log n)$$

And since the final scan within the leaf is constant (small), total cost remains logarithmic.

### Try It Yourself

1. Build a B+ Tree with  $t = 2$  and insert keys [1, 5, 8, 10, 12, 15, 18, 20, 22, 25, 30].
2. Search for 15, 17, and 8.
3. Trace your path from root  $\rightarrow$  internal  $\rightarrow$  leaf.
4. Observe that all searches end in leaves.

### Test Cases

| Search Key | Expected Result | Path                                         |
|------------|-----------------|----------------------------------------------|
| 15         | Found           | Root $\rightarrow$ Middle $\rightarrow$ Leaf |
| 8          | Found           | Root $\rightarrow$ Left $\rightarrow$ Leaf   |
| 17         | Not Found       | Root $\rightarrow$ Middle $\rightarrow$ Leaf |
| 25         | Found           | Root $\rightarrow$ Right $\rightarrow$ Leaf  |

### Complexity

| Operation   | Time            | Space  |
|-------------|-----------------|--------|
| Search      | $O(\log n)$     | $O(n)$ |
| Insert      | $O(\log n)$     | $O(n)$ |
| Range Query | $O(\log n + k)$ | $O(n)$ |

B+ Tree Search exemplifies I/O-aware design, every pointer followed is a disk page, every leaf scan is cache-friendly, and every key lives exactly where range queries want it.

## 278 B\* Tree

A B\* Tree is a refined version of the B-Tree, designed to achieve higher node occupancy and fewer splits. It enforces that each node (except root) must be at least two-thirds full, compared to the half-full guarantee in a standard B-Tree.

To achieve this, B\* Trees use redistribution between siblings before splitting, which improves space utilization and I/O efficiency, making them ideal for database and file system indexes.

### What Problem Are We Solving?

In a standard B-Tree, each node maintains at least  $t - 1$  keys (50% occupancy). But frequent splits can cause fragmentation and wasted space.

We want to:

- Increase space efficiency (reduce empty slots)
- Defer splitting when possible
- Maintain balance and sorted order

B\* Trees solve this by borrowing and redistributing keys between siblings before splitting, ensuring  $\geq 2/3$  occupancy.

### How It Works (Plain Language)

A B\* Tree works like a B-Tree but with smarter split logic:

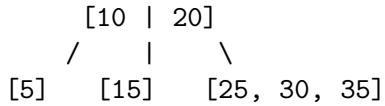
1. Insertion path:
  - Traverse top-down to find target leaf.
  - If the target node is full, check its sibling.
2. Redistribution step:
  - If sibling has room, redistribute keys between them and the parent key.
3. Double split step:
  - If both siblings are full, split both into three nodes (two full + one new node), redistributing keys evenly among them.

This ensures every node (except root) is at least 2/3 full, leading to better disk utilization.

## Example Step by Step

Let  $t = 2$  (max 3 keys per node). Insert keys: [5, 10, 15, 20, 25, 30, 35]

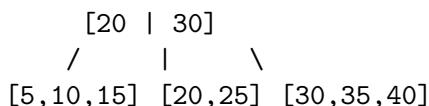
Step 1–4: Build like B-Tree until root [10, 20].



Now insert 40 → rightmost node [25, 30, 35] is full.

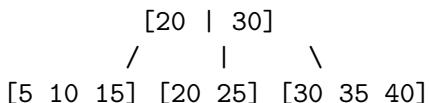
- Check sibling: left sibling [15] has room → redistribute keys Combine [15], [25, 30, 35], and parent key 20 → [15, 20, 25, 30, 35] Split into three nodes evenly: [15, 20], [25, 30], [35] Parent updated with new separator.

Result:



Each node 2/3 full, no wasted space.

## Visualization



Redistribution ensures balance and density before splitting.

## Tiny Code (Simplified Pseudocode)

```

def insert_bstar(tree, key):
    node = find_leaf(tree.root, key)
    if node.full():
        sibling = node.get_sibling()
        if sibling and not sibling.full():
            redistribute(node, sibling, tree.parent(node))
        else:
            split_three(node, sibling, tree.parent(node))
    insert_into_leaf(node, key)

```

*(Actual implementation is more complex, involving parent updates and sibling pointers.)*

## Why It Matters

- Better space utilization: nodes ~ 66% full
- Fewer splits: more stable performance under heavy inserts
- Improved I/O locality: fewer disk blocks accessed
- Used in: database systems (IBM DB2), file systems (ReiserFS), B\*-based caching structures

## A Gentle Proof (Why It Works)

In B\* Trees:

- Each non-root node contains  $\geq \frac{2}{3}(2t - 1)$  keys.
- Height  $h$  satisfies:

$$\left(\frac{3}{2}t\right)^h \leq n$$

Taking logs:

$$h = O(\log_t n)$$

Thus, height remains logarithmic, but nodes pack more data per level.

Fewer levels  $\rightarrow$  fewer I/Os  $\rightarrow$  better performance.

## Try It Yourself

1. Build a B\* Tree with  $t = 2$ .
2. Insert keys: [5, 10, 15, 20, 25, 30, 35, 40].
3. Watch how redistribution occurs before splits.
4. Compare with B-Tree splits for same sequence.

## Test Cases

| Input Keys            | t | Result                   | Notes                       |
|-----------------------|---|--------------------------|-----------------------------|
| [5,10,15,20,25]       | 2 | Root [15]                | Full utilization            |
| [5,10,15,20,25,30,35] | 2 | Root [20,30]             | Redistribution before split |
| [1..15]               | 2 | Balanced, 2/3 full nodes | High density                |

## Complexity

| Operation | Time        | Space  | Occupancy |
|-----------|-------------|--------|-----------|
| Search    | $O(\log n)$ | $O(n)$ | $\geq 66$ |
| Insert    | $O(\log n)$ | $O(n)$ | $\geq 66$ |
| Delete    | $O(\log n)$ | $O(n)$ | $\geq 66$ |

B\* Trees take the elegance of B-Trees and push them closer to perfect, fewer splits, denser nodes, and smoother scaling for large datasets.

## 279 Adaptive Radix Tree

An Adaptive Radix Tree (ART) is a space-efficient, cache-friendly data structure that combines ideas from tries and radix trees. It dynamically adapts its node representation based on the number of children, optimizing both memory usage and lookup speed.

Unlike a fixed-size radix tree (which wastes space with sparse nodes), ART chooses a compact node type (like Node4, Node16, Node48, Node256) depending on occupancy, growing as needed.

## **What Problem Are We Solving?**

Standard tries and radix trees are fast but memory-heavy. If keys share long prefixes, many nodes hold only one child, wasting memory.

We want a structure that:

- Keeps  $O(L)$  lookup time ( $L = \text{key length}$ )
- Adapts node size to occupancy
- Minimizes pointer overhead
- Exploits cache locality

ART achieves this by dynamically switching node types as the number of children grows.

## **How It Works (Plain Language)**

Each internal node in ART can be one of four types:

| Node Type | Capacity     | Description                      |
|-----------|--------------|----------------------------------|
| Node4     | 4 children   | smallest, uses linear search     |
| Node16    | 16 children  | small array, vectorized search   |
| Node48    | 48 children  | index map, stores child pointers |
| Node256   | 256 children | direct addressing by byte value  |

Keys are processed byte by byte, branching at each level. When a node fills beyond its capacity, it upgrades to the next node type.

## **Example**

Insert keys: ["A", "AB", "AC", "AD", "AE"]

1. Start with root Node4 (can store 4 children).
2. After inserting “AE”, Node4 exceeds capacity → upgrade to Node16.
3. Children remain in sorted order by key byte.

This adaptive upgrade keeps nodes dense and efficient.

## **Example Step by Step**

| Step | Operation   | Node Type         | Keys Stored       | Note            |
|------|-------------|-------------------|-------------------|-----------------|
| 1    | Insert "A"  | Node4             | A                 | Create root     |
| 2    | Insert "AB" | Node4             | A, AB             | Add branch      |
| 3    | Insert "AC" | Node4             | A, AB, AC         | Still under 4   |
| 4    | Insert "AD" | Node4             | A, AB, AC, AD     | Full            |
| 5    | Insert "AE" | Upgrade to Node16 | A, AB, AC, AD, AE | Adaptive growth |

## Visualization

```
Root (Node16)
'A' → Node
  'B' (Leaf)
  'C' (Leaf)
  'D' (Leaf)
  'E' (Leaf)
```

Each node type adapts its layout for the best performance.

## Tiny Code (Simplified Pseudocode)

```
class Node:
    def __init__(self):
        self.children = {}

def insert_art(root, key):
    node = root
    for byte in key:
        if byte not in node.children:
            node.children[byte] = Node()
        node = node.children[byte]
    node.value = True
```

(A real ART dynamically switches between Node4, Node16, Node48, Node256 representations.)

## Why It Matters

- Adaptive memory use, no wasted space for sparse nodes
- Cache-friendly, contiguous memory layout

- Fast lookups, vectorized search for Node16
- Used in modern databases (e.g., HyPer, Umbra, DuckDB)

## A Gentle Proof (Why It Works)

Let  $L = \text{key length}$ ,  $b = \text{branching factor}$  (max 256 per byte). In a naive trie, each node allocates  $O(b)$  slots, many unused.

In ART:

- Each node stores only actual children, so

$$\text{space} \approx O(n + L)$$

- Lookup remains  $O(L)$  since we traverse one node per byte.
- Space improves by factor proportional to sparsity.

Thus ART maintains trie-like performance with hash table-like compactness.

## Try It Yourself

1. Insert ["dog", "dot", "door", "dorm"]
2. Observe how Node4 → Node16 transitions happen
3. Count number of nodes, compare with naive trie
4. Measure memory usage and access speed

## Test Cases

| Keys                       | Resulting Root Type | Notes                    |
|----------------------------|---------------------|--------------------------|
| ["a", "b"]                 | Node4               | 2 children               |
| ["a", "b", "c", "d", "e"]  | Node16              | Upgrade after 5th insert |
| ["aa", "ab", "ac"... "az"] | Node48 or Node256   | Dense branching          |

## Complexity

| Operation | Time   | Space      | Adaptive Behavior   |
|-----------|--------|------------|---------------------|
| Search    | $O(L)$ | $O(n + L)$ | Node grows/shrinks  |
| Insert    | $O(L)$ | $O(n)$     | Node type upgrade   |
| Delete    | $O(L)$ | $O(n)$     | Downgrade if sparse |

An Adaptive Radix Tree gives the best of both worlds: prefix compression of tries and space efficiency of hash maps, a modern weapon for high-performance indexing.

## 280 Trie Compression

A compressed trie (also called a radix tree or Patricia trie) is an optimized form of a trie where chains of single-child nodes are merged into a single edge. Instead of storing one character per node, each edge can hold an entire substring.

This reduces the height of the trie, minimizes memory usage, and accelerates searches, perfect for applications like prefix lookup, routing tables, and dictionary storage.

### What Problem Are We Solving?

A naive trie wastes space when many nodes have only one child.

For example, inserting `["cat", "car", "dog"]` into a naive trie yields long, skinny paths:

```
c → a → t  
c → a → r  
d → o → g
```

We can compress those linear chains into edges labeled with substrings:

```
c → a → "t"  
c → a → "r"  
d → "og"
```

This saves memory and reduces traversal depth.

### How It Works (Plain Language)

The key idea is path compression: whenever a node has a single child, merge them into one edge containing the combined substring.

| Step | Operation                           | Result                       |
|------|-------------------------------------|------------------------------|
| 1    | Build a normal trie                 | One character per edge       |
| 2    | Traverse each path                  | If node has one child, merge |
| 3    | Replace chain with a substring edge | Fewer nodes, shorter height  |

Compressed tries store edge labels as substrings rather than single characters.

## Example

Insert ["bear", "bell", "bid", "bull", "buy"]

1. Start with naive trie.
2. Identify single-child paths.
3. Merge paths:

```
b
e → "ar"
    "ll"
u → "ll"
    "y"
```

Each edge now carries a substring rather than a single letter.

## Example Step by Step

| Step | Insert                      | Action                        | Result       |
|------|-----------------------------|-------------------------------|--------------|
| 1    | "bear"                      | Create path b-e-a-r           | 4 nodes      |
| 2    | "bell"                      | Shares prefix "be"            | Merge prefix |
| 3    | "bid"                       | New branch at "b"             | Add new edge |
| 4    | Compress single-child paths | Replace edges with substrings |              |

## Tiny Code (Simplified Pseudocode)

```
class Node:
    def __init__(self):
        self.children = {}
        self.is_end = False

def insert_trie_compressed(root, word):
    node = root
    i = 0
    while i < len(word):
        for edge, child in node.children.items():
            prefix_len = common_prefix(edge, word[i:])
            if prefix_len > 0:
```

```

if prefix_len < len(edge):
    # Split edge
    remainder = edge[prefix_len:]
    new_node = Node()
    new_node.children[remainder] = child
    node.children[word[i:i+prefix_len]] = new_node
    del node.children[edge]
    node = node.children[word[i:i+prefix_len]]
    i += prefix_len
    break
else:
    node.children[word[i:]] = Node()
    node.children[word[i:]].is_end = True
    break
node.is_end = True

```

This simplified version merges edges whenever possible.

## Why It Matters

- Saves memory by merging chains
- Faster search (fewer hops per lookup)
- Ideal for prefix-based queries
- Used in routing tables, autocomplete systems, and dictionaries

## A Gentle Proof (Why It Works)

Let  $n$  be the total length of all keys and  $k$  the number of keys.

- A naive trie can have up to  $O(n)$  nodes.
- A compressed trie has at most  $k - 1$  internal nodes and  $k$  leaves, since each branching point corresponds to a unique prefix shared by at least two keys.

Thus, compressed tries reduce both height and node count:

$$O(n) \text{ nodes (naive)} \rightarrow O(k) \text{ nodes (compressed)}$$

Search and insert remain  $O(L)$ , where  $L$  is key length, but with fewer steps.

## Try It Yourself

1. Insert ["car", "cat", "cart", "dog"]
2. Draw both naive and compressed tries
3. Count number of nodes before and after compression
4. Verify edge labels as substrings

## Test Cases

| Keys                   | Naive Trie Nodes | Compressed Trie Nodes |
|------------------------|------------------|-----------------------|
| ["a", "b"]             | 3                | 3                     |
| ["apple", "app"]       | 6                | 4                     |
| ["abc", "abd", "aef"]  | 8                | 6                     |
| ["car", "cart", "cat"] | 9                | 6                     |

## Complexity

| Operation | Time   | Space  | Notes                       |
|-----------|--------|--------|-----------------------------|
| Search    | $O(L)$ | $O(k)$ | $L$ = key length            |
| Insert    | $O(L)$ | $O(k)$ | Split edges when needed     |
| Delete    | $O(L)$ | $O(k)$ | Merge edges if path shrinks |

A compressed trie elegantly blends trie structure and path compression, turning long chains into compact edges, a key step toward efficient prefix trees, routing tables, and tries for text indexing.

## Section 29. Persistent and Functional Data Structures

### 281 Persistent Stack

A persistent stack is a versioned data structure that remembers all its past states. Instead of overwriting data, every `push` or `pop` operation creates a new version of the stack while keeping access to the old ones.

This concept is part of functional data structures, where immutability and version history are first-class citizens.

## What Problem Are We Solving?

In traditional stacks, each operation mutates the structure, old versions are lost.

Persistent stacks solve this by allowing:

- Access to previous states at any time
- Undo or time-travel features
- Purely functional programs where data is never mutated

Used in compilers, backtracking systems, and functional programming languages.

## How It Works (Plain Language)

A stack is a linked list:

- `push(x)` adds a new head node
- `pop()` returns the next node

For persistence, we never modify nodes, instead, each operation creates a new head pointing to existing tails.

| Version | Operation | Top Element | Structure |
|---------|-----------|-------------|-----------|
| v0      | empty     | ,           |           |
| v1      | push(10)  | 10          | 10 →      |
| v2      | push(20)  | 20          | 20 → 10 → |
| v3      | pop()     | 10          | 10 →      |

Each version reuses previous nodes, no data copying.

## Example

Start with an empty stack v0:

1. `v1 = push(v0, 10) → stack [10]`
2. `v2 = push(v1, 20) → stack [20, 10]`
3. `v3 = pop(v2) → returns 20, new stack [10]`

Now we have three accessible versions:

```
v0:  
v1: 10  
v2: 20 → 10  
v3: 10
```

### Tiny Code (Python)

```
class Node:  
    def __init__(self, value, next_node=None):  
        self.value = value  
        self.next = next_node  
  
class PersistentStack:  
    def __init__(self, top=None):  
        self.top = top  
  
    def push(self, value):  
        # new node points to current top  
        return PersistentStack(Node(value, self.top))  
  
    def pop(self):  
        if not self.top:  
            return self, None  
        return PersistentStack(self.top.next), self.top.value  
  
    def peek(self):  
        return None if not self.top else self.top.value  
  
# Example  
v0 = PersistentStack()  
v1 = v0.push(10)  
v2 = v1.push(20)  
v3, popped = v2.pop()  
  
print(v2.peek()) # 20  
print(v3.peek()) # 10
```

This approach reuses nodes, creating new versions without mutation.

## Tiny Code (C, Conceptual)

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    int value;
    struct Node* next;
} Node;

typedef struct {
    Node* top;
} Stack;

Stack* push(Stack* s, int value) {
    Node* node = malloc(sizeof(Node));
    node->value = value;
    node->next = s->top;
    Stack* new_stack = malloc(sizeof(Stack));
    new_stack->top = node;
    return new_stack;
}

Stack* pop(Stack* s, int* popped_value) {
    if (!s->top) return s;
    *popped_value = s->top->value;
    Stack* new_stack = malloc(sizeof(Stack));
    new_stack->top = s->top->next;
    return new_stack;
}
```

Every push or pop creates a new `Stack*` that points to the previous structure.

## Why It Matters

- Immutability ensures data safety and concurrency-friendly design
- Versioning allows backtracking, undo, or branching computations
- Foundation for functional programming and persistent data stores

## A Gentle Proof (Why It Works)

Each version of the stack shares unchanged nodes with previous versions. Because `push` and `pop` only modify the head reference, older versions remain intact.

If  $n$  is the total number of operations,

- Each new version adds  $O(1)$  space
- Shared tails ensure total space =  $O(n)$

Mathematically:

$$S_n = S_{n-1} + O(1)$$

and old versions never get overwritten, ensuring persistence.

## Try It Yourself

1. Build a persistent stack with values [1, 2, 3]
2. Pop once, and confirm earlier versions still have their values
3. Compare with a mutable stack implementation
4. Visualize the shared linked nodes between versions

## Test Cases

| Operation                      | Result         | Notes                    |
|--------------------------------|----------------|--------------------------|
| <code>v1 = push(v0, 10)</code> | [10]           | new version              |
| <code>v2 = push(v1, 20)</code> | [20, 10]       | shares 10-node           |
| <code>v3, val = pop(v2)</code> | val = 20, [10] | old v2 intact            |
| <code>v1.peek()</code>         | 10             | unaffected by later pops |

## Complexity

| Operation          | Time   | Space  | Notes           |
|--------------------|--------|--------|-----------------|
| Push               | $O(1)$ | $O(1)$ | new head        |
| Pop                | $O(1)$ | $O(1)$ | new top pointer |
| Access old version | $O(1)$ | ,      | store reference |

A persistent stack elegantly combines immutability, sharing, and time-travel, a small but powerful step into the world of functional data structures.

## 282 Persistent Array

A persistent array is an immutable, versioned structure that allows access to all past states. Instead of overwriting elements, every update creates a new version that shares most of its structure with previous ones.

This makes it possible to “time travel”, view or restore any earlier version in constant or logarithmic time, without copying the entire array.

### What Problem Are We Solving?

A normal array is mutable, every `arr[i] = x` destroys the old value. If we want history, undo, or branching computation, this is unacceptable.

A persistent array keeps all versions:

| Version | Operation               | State                |
|---------|-------------------------|----------------------|
| v0      | []                      | empty                |
| v1      | <code>set(0, 10)</code> | [10]                 |
| v2      | <code>set(0, 20)</code> | [20] (v1 still [10]) |

Each version reuses unmodified parts of the array, avoiding full duplication.

### How It Works (Plain Language)

A persistent array can be implemented using copy-on-write or tree-based structures.

1. Copy-on-Write (Small Arrays)
  - Create a new array copy only when an element changes.
  - Simple but  $O(n)$  update cost.
2. Path Copying with Trees (Large Arrays)
  - Represent the array as a balanced binary tree (like a segment tree).
  - Each update copies only the path to the changed leaf.
  - Space per update =  $O(\log n)$

So, each version points to a root node. When you modify index  $i$ , a new path is created down the tree, while untouched subtrees are shared.

## **Example**

Let's build a persistent array of size 4.

### **Step 1: Initial version**

```
v0 = [0, 0, 0, 0]
```

### **Step 2: Update index 2**

```
v1 = set(v0, 2, 5) → [0, 0, 5, 0]
```

### **Step 3: Update index 1**

```
v2 = set(v1, 1, 9) → [0, 9, 5, 0]
```

v0, v1, and v2 all coexist independently.

## **Example Step-by-Step (Tree Representation)**

Each node covers a range:

```
Root: [0..3]
  Left [0..1]
    [0] → 0
    [1] → 9
  Right [2..3]
    [2] → 5
    [3] → 0
```

Updating index 1 copies only the path  $[0..3] \rightarrow [0..1] \rightarrow [1]$ , not the entire tree.

## **Tiny Code (Python, Tree-based)**

```

class Node:
    def __init__(self, left=None, right=None, value=0):
        self.left = left
        self.right = right
        self.value = value

    def build(l, r):
        if l == r:
            return Node()
        m = (l + r) // 2
        return Node(build(l, m), build(m+1, r))

    def update(node, l, r, idx, val):
        if l == r:
            return Node(value=val)
        m = (l + r) // 2
        if idx <= m:
            return Node(update(node.left, l, m, idx, val), node.right)
        else:
            return Node(node.left, update(node.right, m+1, r, idx, val))

    def query(node, l, r, idx):
        if l == r:
            return node.value
        m = (l + r) // 2
        return query(node.left, l, m, idx) if idx <= m else query(node.right, m+1, r, idx)

# Example
n = 4
v0 = build(0, n-1)
v1 = update(v0, 0, n-1, 2, 5)
v2 = update(v1, 0, n-1, 1, 9)

print(query(v2, 0, n-1, 2)) # 5
print(query(v1, 0, n-1, 1)) # 0

```

Each update creates a new version root.

## Why It Matters

- Time-travel debugging: retrieve old states

- Undo/redo systems in editors
- Branching computations in persistent algorithms
- Functional programming without mutation

### A Gentle Proof (Why It Works)

Let  $n$  be array size,  $u$  number of updates.

Each update copies  $O(\log n)$  nodes. So total space:

$$O(n + u \log n)$$

Each query traverses one path  $O(\log n)$ . No version ever invalidates another, all roots remain accessible.

Persistence holds because we never mutate existing nodes, only allocate new ones and reuse subtrees.

### Try It Yourself

1. Build an array of size 8, all zeros.
2. Create v1 = set index 4 → 7
3. Create v2 = set index 2 → 9
4. Print values from v0, v1, v2
5. Confirm that old versions remain unchanged.

### Test Cases

| Operation   | Input     | Output    | Notes       |
|-------------|-----------|-----------|-------------|
| build(4)    | [0,0,0,0] | v0        | base        |
| set(v0,2,5) |           | [0,0,5,0] | new version |
| set(v1,1,9) |           | [0,9,5,0] | v1 reused   |

| Operation | Time | Space | Notes |
|-----------|------|-------|-------|
|-----------|------|-------|-------|

## Complexity

| Operation          | Time        | Space       | Notes          |
|--------------------|-------------|-------------|----------------|
| Build              | $O(n)$      | $O(n)$      | initial        |
| Update             | $O(\log n)$ | $O(\log n)$ | path copy      |
| Query              | $O(\log n)$ | ,           | one path       |
| Access Old Version | $O(1)$      | ,           | root reference |

A persistent array turns ephemeral memory into a versioned timeline, each change is a branch, every version eternal. Perfect for functional programming, debugging, and algorithmic history.

## 283 Persistent Segment Tree

A persistent segment tree is a versioned data structure that supports range queries and point updates, while keeping every past version accessible.

It's a powerful combination of segment trees and persistence, allowing you to query historical states, perform undo operations, and even compare past and present results efficiently.

### What Problem Are We Solving?

A standard segment tree allows:

- Point updates: `arr[i] = x`
- Range queries: `sum(l, r)` or `min(l, r)`

But every update overwrites old values. A persistent segment tree solves this by creating a new version on each update, reusing unchanged nodes.

| Version | Operation    | State        |
|---------|--------------|--------------|
| v0      | build        | [1, 2, 3, 4] |
| v1      | update(2, 5) | [1, 2, 5, 4] |
| v2      | update(1, 7) | [1, 7, 5, 4] |

Now you can query any version:

- `query(v0, 1, 3) → old sum`
- `query(v2, 1, 3) → updated sum`

## How It Works (Plain Language)

A segment tree is a binary tree where each node stores an aggregate (sum, min, max) over a segment.

Persistence is achieved by path copying:

- When updating index  $i$ , only nodes on the path from root to leaf are replaced.
- All other nodes are shared between versions.

So each new version costs  $O(\log n)$  nodes and space.

| Step | Operation    | Affected Nodes        |
|------|--------------|-----------------------|
| 1    | Build        | $O(n)$ nodes          |
| 2    | Update(2, 5) | $O(\log n)$ new nodes |
| 3    | Update(1, 7) | $O(\log n)$ new nodes |

## Example

Let initial array be [1, 2, 3, 4]

1. Build tree (v0)
2.  $v1 = \text{update}(v0, 2 \rightarrow 5)$
3.  $v2 = \text{update}(v1, 1 \rightarrow 7)$

Now:

- `query(v0, 1, 4) = 10`
- `query(v1, 1, 4) = 12`
- `query(v2, 1, 4) = 17`

All versions share most nodes, saving memory.

## Example Step-by-Step

### Update v0 → v1 at index 2

| Version | Tree Nodes Copied          | Shared           |
|---------|----------------------------|------------------|
| v1      | Path [root → left → right] | Others unchanged |

So v1 differs only along one path.

### Tiny Code (Python)

```

class Node:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def build(arr, l, r):
    if l == r:
        return Node(arr[l])
    m = (l + r) // 2
    left = build(arr, l, m)
    right = build(arr, m + 1, r)
    return Node(left.val + right.val, left, right)

def update(node, l, r, idx, val):
    if l == r:
        return Node(val)
    m = (l + r) // 2
    if idx <= m:
        left = update(node.left, l, m, idx, val)
        return Node(left.val + node.right.val, left, node.right)
    else:
        right = update(node.right, m + 1, r, idx, val)
        return Node(node.left.val + right.val, node.left, right)

def query(node, l, r, ql, qr):
    if qr < l or ql > r:
        return 0
    if ql <= l and r <= qr:
        return node.val
    m = (l + r) // 2
    return query(node.left, l, m, ql, qr) + query(node.right, m + 1, r, ql, qr)

```

```

# Example
arr = [1, 2, 3, 4]
v0 = build(arr, 0, 3)
v1 = update(v0, 0, 3, 2, 5)
v2 = update(v1, 0, 3, 1, 7)

print(query(v0, 0, 3, 0, 3)) # 10
print(query(v1, 0, 3, 0, 3)) # 12
print(query(v2, 0, 3, 0, 3)) # 17

```

## Why It Matters

- Access any version instantly
- Enables time-travel queries
- Supports immutable analytics
- Used in offline queries, competitive programming, and functional databases

## A Gentle Proof (Why It Works)

Each update only modifies  $O(\log n)$  nodes. All other subtrees are shared, so total space:

$$S(u) = O(n + u \log n)$$

Querying any version costs  $O(\log n)$  since only one path is traversed.

Persistence holds since no nodes are mutated, only replaced.

## Try It Yourself

1. Build [1, 2, 3, 4]
2. Update index 2 → 5 (v1)
3. Update index 1 → 7 (v2)
4. Query sum(1, 4) in v0, v1, v2
5. Verify shared subtrees via visualization

## Test Cases

| Version | Operation | Query(0,3) | Notes            |
|---------|-----------|------------|------------------|
| v0      | [1,2,3,4] | 10         | base             |
| v1      | set(2,5)  | 12         | changed one leaf |
| v2      | set(1,7)  | 17         | another update   |

## Complexity

| Operation      | Time        | Space       | Notes     |
|----------------|-------------|-------------|-----------|
| Build          | $O(n)$      | $O(n)$      | full tree |
| Update         | $O(\log n)$ | $O(\log n)$ | path copy |
| Query          | $O(\log n)$ | ,           | one path  |
| Version Access | $O(1)$      | ,           | via root  |

A persistent segment tree is your immutable oracle, each version a snapshot in time, forever queryable, forever intact.

## 284 Persistent Linked List

A persistent linked list is a versioned variant of the classic singly linked list, where every insertion or deletion produces a new version without destroying the old one.

Each version represents a distinct state of the list, and all versions coexist by sharing structure, unchanged nodes are reused, only the modified path is copied.

This technique is core to functional programming, undo systems, and immutable data structures.

### What Problem Are We Solving?

A mutable linked list loses its history after every change.

With persistence, we preserve all past versions:

| Version | Operation      | List     |
|---------|----------------|----------|
| v0      | empty          |          |
| v1      | push_front(10) | [10]     |
| v2      | push_front(20) | [20, 10] |
| v3      | pop_front()    | [10]     |

Each version is a first-class citizen, you can traverse, query, or compare any version at any time.

## How It Works (Plain Language)

Each node in a singly linked list has:

- `value`
- `next` pointer

For persistence, we never mutate nodes. Instead, operations return a new head:

- `push_front(x)`: create a new node `n = Node(x, old_head)`
- `pop_front()`: return `old_head.next` as new head

All old nodes remain intact and shared.

| Operation                   | New Node        | Shared Structure      |
|-----------------------------|-----------------|-----------------------|
| <code>push_front(20)</code> | new head        | tail reused           |
| <code>pop_front()</code>    | new head (next) | old head still exists |

## Example

### Step-by-step versioning

1. `v0 = []`
2. `v1 = push_front(v0, 10) → [10]`
3. `v2 = push_front(v1, 20) → [20, 10]`
4. `v3 = pop_front(v2) → [10]`

Versions:

```
v0:  
v1: 10  
v2: 20 → 10  
v3: 10
```

All coexist and share structure.

## Example (Graph View)

```
v0:  
v1: 10 →  
v2: 20 → 10 →  
v3: 10 →
```

Notice: v2.tail is reused from v1.

## Tiny Code (Python)

```
class Node:  
    def __init__(self, value, next_node=None):  
        self.value = value  
        self.next = next_node  
  
class PersistentList:  
    def __init__(self, head=None):  
        self.head = head  
  
    def push_front(self, value):  
        new_node = Node(value, self.head)  
        return PersistentList(new_node)  
  
    def pop_front(self):  
        if not self.head:  
            return self, None  
        return PersistentList(self.head.next), self.head.value  
  
    def to_list(self):  
        result, curr = [], self.head  
        while curr:  
            result.append(curr.value)  
            curr = curr.next  
        return result  
  
# Example  
v0 = PersistentList()  
v1 = v0.push_front(10)  
v2 = v1.push_front(20)  
v3, popped = v2.pop_front()
```

```
print(v1.to_list())  # [10]
print(v2.to_list())  # [20, 10]
print(v3.to_list())  # [10]
```

### Tiny Code (C, Conceptual)

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    int value;
    struct Node* next;
} Node;

typedef struct {
    Node* head;
} PList;

PList push_front(PList list, int value) {
    Node* new_node = malloc(sizeof(Node));
    new_node->value = value;
    new_node->next = list.head;
    PList new_list = { new_node };
    return new_list;
}

PList pop_front(PList list, int* popped) {
    if (!list.head) return list;
    *popped = list.head->value;
    PList new_list = { list.head->next };
    return new_list;
}
```

No mutations, only new nodes allocated.

### Why It Matters

- Immutable, perfect for functional programs
- Undo / Time-travel, revisit old versions

- Safe concurrency, no data races
- Memory-efficient, tail sharing reuses old structure

### A Gentle Proof (Why It Works)

Let  $n$  be number of operations. Each `push_front` or `pop_front` creates at most one new node.

Thus:

- Total space after  $n$  ops:  $O(n)$
- Time per operation:  $O(1)$

Persistence is guaranteed since no node is modified in place.

All versions share the unchanged suffix:

$$v_k = \text{Node}(x_k, v_{k-1})$$

Hence, structure sharing is linear and safe.

### Try It Yourself

1. Start with empty list
2. Push 3 → Push 2 → Push 1
3. Pop once
4. Print all versions
5. Observe how tails are shared

### Test Cases

| Operation  | Input | Output   | Version |
|------------|-------|----------|---------|
| push_front | 10    | [10]     | v1      |
| push_front | 20    | [20, 10] | v2      |
| pop_front  | ,     | [10]     | v3      |
| to_list    | v1    | [10]     | ,       |

### Complexity

| Operation  | Time   | Space  | Notes               |
|------------|--------|--------|---------------------|
| push_front | $O(1)$ | $O(1)$ | new head only       |
| pop_front  | $O(1)$ | $O(1)$ | reuse next          |
| access     | $O(n)$ | ,      | same as linked list |

A persistent linked list is the simplest gateway to persistence, each operation is  $O(1)$ , each version immortal. It's the backbone of functional stacks, queues, and immutable collections.

## 286 Finger Tree

A finger tree is a versatile, persistent data structure that provides amortized  $O(1)$  access to both ends (front and back) and  $O(\log n)$  access or updates in the middle.

It's a functional, immutable sequence structure, a balanced tree augmented with *fingers* (fast access points) to its ends. Finger trees form the foundation for many persistent data types, such as queues, deques, priority sequences, and even rope-like text editors.

### What Problem Are We Solving?

Immutable lists are fast at the front but slow at the back. Immutable arrays are the opposite. Deques with persistence are hard to maintain efficiently.

We want:

- $O(1)$  front access
- $O(1)$  back access
- $O(\log n)$  middle access
- Persistence and immutability

Finger trees achieve this by combining shallow digit buffers at the edges with balanced nodes in the middle.

### How It Works (Plain Language)

A finger tree is built recursively:

```
FingerTree = Empty
          | Single(a)
          | Deep(prefix, deeper_tree, suffix)
```

- prefix and suffix: small arrays (digits) with 1–4 elements

- deeper\_tree: recursively holds nodes of higher rank

The *fingers* (prefix/suffix) give constant-time access to both ends. Insertions push elements into digits; when full, they roll into the deeper tree.

### Example

Insert 1, 2, 3, 4, 5:

```
Deep [1,2] (Node [3,4]) [5]
```

You can:

- Push front → prepend into prefix
- Push back → append into suffix
- Access ends in O(1)
- Insert middle → recurse into deeper tree (O(log n))

Each operation returns a new version sharing unchanged subtrees.

### Example State

| Operation     | Structure              | Notes           |
|---------------|------------------------|-----------------|
| empty         | Empty                  | base            |
| push_front(1) | Single(1)              | one element     |
| push_front(2) | Deep [2] Empty [1]     | two ends        |
| push_back(3)  | Deep [2] Empty [1,3]   | add suffix      |
| push_back(4)  | Deep [2,4] Empty [1,3] | balanced growth |

Each version reuses most of its structure, ensuring persistence.

### Tiny Code (Python – Conceptual)

This is a simplified model, not a complete implementation (real finger trees rely on more type-level machinery).

```

class Empty:
    pass

class Single:
    def __init__(self, value):
        self.value = value

class Deep:
    def __init__(self, prefix, middle, suffix):
        self.prefix = prefix
        self.middle = middle
        self.suffix = suffix

def push_front(tree, x):
    if isinstance(tree, Empty):
        return Single(x)
    if isinstance(tree, Single):
        return Deep([x], Empty(), [tree.value])
    if len(tree.prefix) < 4:
        return Deep([x] + tree.prefix, tree.middle, tree.suffix)
    else:
        # roll prefix into deeper tree
        new_middle = push_front(tree.middle, tree.prefix)
        return Deep([x], new_middle, tree.suffix)

def to_list(tree):
    if isinstance(tree, Empty):
        return []
    if isinstance(tree, Single):
        return [tree.value]
    return tree.prefix + to_list(tree.middle) + tree.suffix

```

This captures the core recursive flavor, constant-time fingers, logarithmic recursion.

## Why It Matters

- Generic framework for sequences
- Amortized O(1) insertion/removal at both ends
- O(log n) concatenation, split, or search
- Basis for:

- Functional deques
- Priority queues
- Ordered sequences (like RRB-trees)
- Incremental editors

## A Gentle Proof (Why It Works)

Digits store up to 4 elements, guaranteeing bounded overhead. Each recursive step reduces the size by a constant factor, ensuring depth =  $O(\log n)$ .

For each operation:

- Ends touched in  $O(1)$
- Structural changes at depth  $O(\log n)$

Thus, total cost:

$$T(n) = O(\log n), \text{ amortized } O(1) \text{ at ends}$$

Persistence is ensured because all updates build new nodes without modifying existing ones.

## Try It Yourself

1. Start with empty tree.
2. Push 1, 2, 3, 4.
3. Pop front, observe structure.
4. Push 5, inspect sharing between versions.
5. Convert each version to list, compare results.

## Test Cases

| Operation     | Result    | Notes              |
|---------------|-----------|--------------------|
| push_front(1) | [1]       | base               |
| push_front(2) | [2, 1]    | prefix growth      |
| push_back(3)  | [2, 1, 3] | suffix add         |
| pop_front()   | [1, 3]    | remove from prefix |

## Complexity

| Operation       | Time             | Space       | Notes            |
|-----------------|------------------|-------------|------------------|
| push_front/back | $O(1)$ amortized | $O(1)$      | uses digits      |
| pop_front/back  | $O(1)$ amortized | $O(1)$      | constant fingers |
| random access   | $O(\log n)$      | $O(\log n)$ | recursion        |
| concat/split    | $O(\log n)$      | $O(\log n)$ | efficient split  |

A finger tree is the Swiss Army knife of persistent sequences, fast at both ends, balanced within, and beautifully immutable. It's the blueprint behind countless functional data structures.

## 287 Zipper Structure

A zipper is a powerful technique that makes immutable data structures behave like mutable ones. It provides a focus—a pointer-like position—inside a persistent structure (list, tree, etc.), allowing localized updates, navigation, and edits without mutation.

Think of it as a cursor in a purely functional world. Every movement or edit yields a new version, while sharing unmodified parts.

### What Problem Are We Solving?

Immutable data structures can't be “modified in place.” You can't just “move a cursor” or “replace an element” without reconstructing the entire structure.

A zipper solves this by maintaining:

1. The focused element, and
2. The context (what's on the left/right or above/below).

You can then move, update, or rebuild efficiently, reusing everything else.

### How It Works (Plain Language)

A zipper separates a structure into:

- Focus: current element under attention
- Context: reversible description of the path you took

When you move the focus, you update the context. When you change the focused element, you create a new node and rebuild from context.

For lists:

```
Zipper = (Left, Focus, Right)
```

For trees:

```
Zipper = (ParentContext, FocusNode)
```

You can think of it like a tape in a Turing machine—everything to the left and right is preserved.

### Example (List Zipper)

We represent a list [a, b, c, d] with a cursor on c:

```
Left: [b, a]    Focus: c    Right: [d]
```

From here:

- `move_left` → focus = b
- `move_right` → focus = d
- `update(x)` → replace c with x

All in O(1), returning a new zipper version.

### Example Operations

| Operation                         | Result           | Description       |
|-----------------------------------|------------------|-------------------|
| <code>from_list([a,b,c,d])</code> | ([], a, [b,c,d]) | init at head      |
| <code>move_right</code>           | ([a], b, [c,d])  | shift focus right |
| <code>update('X')</code>          | ([a], X, [c,d])  | replace focus     |
| <code>to_list</code>              | [a,X,c,d]        | rebuild full list |

### Tiny Code (Python – List Zipper)

```

class Zipper:
    def __init__(self, left=None, focus=None, right=None):
        self.left = left or []
        self.focus = focus
        self.right = right or []

    @staticmethod
    def from_list(lst):
        if not lst: return Zipper([], None, [])
        return Zipper([], lst[0], lst[1:])

    def move_left(self):
        if not self.left: return self
        return Zipper(self.left[:-1], self.left[-1], [self.focus] + self.right)

    def move_right(self):
        if not self.right: return self
        return Zipper(self.left + [self.focus], self.right[0], self.right[1:])

    def update(self, value):
        return Zipper(self.left, value, self.right)

    def to_list(self):
        return self.left + [self.focus] + self.right

# Example
z = Zipper.from_list(['a', 'b', 'c', 'd'])
z1 = z.move_right().move_right()    # focus on 'c'
z2 = z1.update('X')
print(z2.to_list())  # ['a', 'b', 'X', 'd']

```

Each operation returns a new zipper, persistent editing made simple.

### Example (Tree Zipper – Conceptual)

A tree zipper stores the path to the root as context:

```
Zipper = (ParentPath, FocusNode)
```

Each parent in the path remembers which side you came from, so you can rebuild upward after an edit.

For example, editing a leaf  $L$  creates a new  $L'$  and rebuilds only the nodes along the path, leaving other subtrees untouched.

## Why It Matters

- Enables localized updates in immutable structures
- Used in functional editors, parsers, navigation systems
- Provides  $O(1)$  local movement,  $O(\text{depth})$  rebuilds
- Core concept in Huet's zipper, a foundational idea in functional programming

## A Gentle Proof (Why It Works)

Each movement or edit affects only the local context:

- Move left/right in a list  $\rightarrow O(1)$
- Move up/down in a tree  $\rightarrow O(1)$
- Rebuilding full structure  $\rightarrow O(\text{depth})$

No mutation occurs; each version reuses all untouched substructures.

Formally, if  $S$  is the original structure and  $f$  the focus,

$$\text{zip}(S) = (\text{context}, f)$$

and

$$\text{unzip}(\text{context}, f) = S$$

ensuring reversibility.

## Try It Yourself

1. Create a zipper from `[1,2,3,4]`
2. Move focus to 3
3. Update focus to 99
4. Rebuild full list
5. Verify older zipper still has old value

| Step | Operation | Result | Notes |
|------|-----------|--------|-------|
|------|-----------|--------|-------|

## Test Cases

| Step | Operation                         | Result           | Notes       |
|------|-----------------------------------|------------------|-------------|
| 1    | <code>from_list([a,b,c,d])</code> | ([], a, [b,c,d]) | init        |
| 2    | <code>move_right()</code>         | ([a], b, [c,d])  | shift focus |
| 3    | <code>update('X')</code>          | ([a], X, [c,d])  | edit        |
| 4    | <code>to_list()</code>            | [a, X, c, d]     | rebuild     |

## Complexity

| Operation          | Time   | Space  | Notes             |
|--------------------|--------|--------|-------------------|
| Move left/right    | $O(1)$ | $O(1)$ | shift focus       |
| Update             | $O(1)$ | $O(1)$ | local replacement |
| Rebuild            | $O(n)$ | $O(1)$ | when unzipping    |
| Access old version | $O(1)$ | ,      | persistent        |

A zipper turns immutability into interactivity. With a zipper, you can move, focus, and edit, all without breaking persistence. It's the bridge between static structure and dynamic navigation.

## 289 Trie with Versioning

A Trie with Versioning is a persistent data structure that stores strings (or sequences) across multiple historical versions. Each new update—an insertion, deletion, or modification—creates a new version of the trie without mutating previous ones, using path copying for structural sharing.

This enables time-travel queries: you can look up keys as they existed at any point in history.

### What Problem Are We Solving?

We want to maintain a versioned dictionary of strings or sequences, supporting:

- Fast prefix search ( $O(\text{length})$ )
- Efficient updates without mutation

- Access to past versions (e.g., snapshots, undo/redo, history)

A versioned trie achieves all three by copying only the path from the root to modified nodes while sharing all other subtrees.

Common use cases:

- Versioned symbol tables
- Historical dictionaries
- Autocomplete with rollback
- Persistent tries in functional languages

## How It Works (Plain Language)

Each trie node contains:

- A mapping from character → child node
- A flag marking end of word

For persistence:

- When you insert or delete, copy only nodes on the affected path.
- Old nodes remain untouched and shared by old versions.

Thus, version  $v_{k+1}$  differs from  $v_k$  only along the modified path.

## Tiny Code (Conceptual Python)

```
class TrieNode:
    def __init__(self, children=None, is_end=False):
        self.children = dict(children or {})
        self.is_end = is_end

    def insert(root, word):
        def _insert(node, i):
            node = TrieNode(node.children, node.is_end)
            if i == len(word):
                node.is_end = True
                return node
            ch = word[i]
            node.children[ch] = _insert(node.children.get(ch, TrieNode()), i + 1)
            return node
        return _insert(root, 0)
```

```

    return _insert(root, 0)

def search(root, word):
    node = root
    for ch in word:
        if ch not in node.children:
            return False
        node = node.children[ch]
    return node.is_end

```

Each call to `insert` returns a new root (new version), sharing all unmodified branches.

## Example

|  | Version | Operation     | Trie Content            |
|--|---------|---------------|-------------------------|
|  | v1      | insert("cat") | { "cat" }               |
|  | v2      | insert("car") | { "cat", "car" }        |
|  | v3      | insert("dog") | { "cat", "car", "dog" } |
|  | v4      | delete("car") | { "cat", "dog" }        |

Versions share nodes for prefix "c" between all earlier versions.

## Why It Matters

- Immutable and Safe: No in-place mutation, perfect for functional systems
- Efficient Rollback: Access any prior version in  $O(1)$  time
- Prefix Sharing: Saves memory through structural reuse
- Practical for History: Ideal for versioned dictionaries, IDEs, search indices

## A Gentle Proof (Why It Works)

Each insertion copies one path of length  $L$  (word length). Total time complexity:

$$T_{\text{insert}} = O(L)$$

Each node on the new path shares all other unchanged subtrees. If  $N$  is total stored characters across all versions,

$$\text{Space} = O(N)$$

Each version root is a single pointer, enabling  $O(1)$  access:

$$\text{Version}_i = \text{Root}_i$$

Old versions remain fully usable, as they never mutate.

### Try It Yourself

1. Insert “cat”, “car”, “dog” into versioned trie
2. Delete “car” to form new version
3. Query prefix “ca” in all versions
4. Check that “car” exists only before deletion
5. Print shared node counts across versions

### Test Case

| Step | Operation     | Version | Exists in Version  | Result |
|------|---------------|---------|--------------------|--------|
| 1    | insert(“cat”) | v1      | cat                | True   |
| 2    | insert(“car”) | v2      | car, cat           | True   |
| 3    | insert(“dog”) | v3      | cat, car, dog      | True   |
| 4    | delete(“car”) | v4      | car (no), cat, dog | False  |

### Complexity

| Operation          | Time   | Space  | Notes                    |
|--------------------|--------|--------|--------------------------|
| Insert             | $O(L)$ | $O(L)$ | Path-copying             |
| Delete             | $O(L)$ | $O(L)$ | Path-copying             |
| Lookup             | $O(L)$ | $O(1)$ | Follows shared structure |
| Access old version | $O(1)$ | ,      | Version pointer access   |

A Trie with Versioning blends structural sharing with prefix indexing. Each version is a frozen snapshot—compact, queryable, and immutable—perfect for versioned word histories.

## 290 Persistent Union-Find

A Persistent Union-Find extends the classical Disjoint Set Union (DSU) structure to support *time-travel queries*. Instead of mutating the parent and rank arrays in place, each union operation produces a new version, enabling queries like:

- “Are  $x$  and  $y$  connected at version  $v$ ?”
- “What did the set look like before the last merge?”

This structure is vital for dynamic connectivity problems where the history of unions matters.

### What Problem Are We Solving?

Classical DSU supports `find` and `union` efficiently in near-constant time, but only for a single evolving state. Once you merge two sets, the old version is gone.

We need a versioned DSU that keeps all previous states intact, supporting:

- Undo/rollback of operations
- Queries over past connectivity
- Offline dynamic connectivity analysis

### How It Works (Plain Language)

A Persistent Union-Find uses path copying (similar to persistent arrays) to maintain multiple versions:

- Each `union` creates a new version
- Only affected parent and rank entries are updated in a new structure
- All other nodes share structure with the previous version

There are two main designs:

1. Full persistence using path copying in functional style
2. Partial persistence using rollback stack (undo operations)

We focus here on *full persistence*.

### Tiny Code (Conceptual Python)

```

class PersistentDSU:
    def __init__(self, n):
        self.versions = []
        parent = list(range(n))
        rank = [0] * n
        self.versions.append((parent, rank))

    def find(self, parent, x):
        if parent[x] == x:
            return x
        return self.find(parent, parent[x])

    def union(self, ver, x, y):
        parent, rank = [*ver[0]], [*ver[1]] # copy arrays
        rx, ry = self.find(parent, x), self.find(parent, y)
        if rx != ry:
            if rank[rx] < rank[ry]:
                parent[rx] = ry
            elif rank[rx] > rank[ry]:
                parent[ry] = rx
            else:
                parent[ry] = rx
                rank[rx] += 1
        self.versions.append((parent, rank))
        return len(self.versions) - 1 # new version index

    def connected(self, ver, x, y):
        parent, _ = self.versions[ver]
        return self.find(parent, x) == self.find(parent, y)

```

Each union returns a new version index. You can query `connected(version, a, b)` at any time.

## Example

| Step | Operation   | Version | Connections     |
|------|-------------|---------|-----------------|
| 1    | make-set(5) | v0      | 0 1 2 3 4       |
| 2    | union(0,1)  | v1      | {0,1}, 2, 3, 4  |
| 3    | union(2,3)  | v2      | {0,1}, {2,3}, 4 |
| 4    | union(1,2)  | v3      | {0,1,2,3}, 4    |

| Step | Operation            | Version | Connections            |
|------|----------------------|---------|------------------------|
| 5    | query connected(0,1) | v1      | True                   |
| 6    | query connected(1,3) | v1      | False (not yet merged) |

You can check connections at any version.

## Why It Matters

- Time-travel queries across historical versions
- Non-destructive updates allow safe rollback
- Crucial for offline dynamic connectivity, e.g., edge insertions over time
- Simplifies debugging, simulation, and version tracking

## A Gentle Proof (Why It Works)

Let  $n$  be the number of elements and  $q$  the number of versions.

Each version differs in only a small number of parent/rank entries. If each `union` copies  $O(\alpha(n))$  elements, total space after  $q$  operations is:

$$O(n + q\alpha(n))$$

Each query operates on a fixed version in:

$$O(\alpha(n))$$

Path compression is often replaced with *partial compression* or omitted to ensure persistence (full path compression breaks immutability).

## Try It Yourself

1. Initialize DSU with 5 elements
2. Perform unions step by step, saving each version
3. Query connectivity across multiple versions
4. Undo merges by reverting to older versions
5. Visualize parent tree evolution

## Test Cases

| Version | Query          | Result |
|---------|----------------|--------|
| v0      | connected(0,1) | False  |
| v1      | connected(0,1) | True   |
| v2      | connected(2,3) | True   |
| v3      | connected(1,3) | True   |
| v1      | connected(1,3) | False  |

## Complexity

| Operation          | Time           | Space       | Notes                  |
|--------------------|----------------|-------------|------------------------|
| Find               | $O(\alpha(n))$ | $O(1)$      | Per version            |
| Union              | $O(\alpha(n))$ | $O(n)$ copy | Copy-on-write path     |
| Connected          | $O(\alpha(n))$ | $O(1)$      | Versioned query        |
| Access old version | $O(1)$         | ,           | Version pointer lookup |

A Persistent Union-Find is a historical map of connectivity. Each version captures a snapshot of relationships—immutable, queryable, and efficient—ideal for evolving graphs and rollback-capable algorithms.

## Section 30. Advanced Trees and Range Queries

### 291 Sparse Table Build

A Sparse Table is a static data structure for answering idempotent range queries in  $O(1)$  time after  $O(n \log n)$  preprocessing. It is perfect for Range Minimum/Maximum Query (RMQ), GCD, and any operation where combining overlapping answers is valid, such as `min`, `max`, `gcd`, `lcm` (with care), and bitwise `and/or`. It is not suitable for sum or other non-idempotent operations if you require  $O(1)$  queries.

#### What Problem Are We Solving?

Given an array  $A[0..n-1]$ , we want to answer queries like

- RMQ: minimum on interval  $[L, R]$
- RMaxQ: maximum on interval  $[L, R]$  in  $O(1)$  time per query, with no updates.

## How It Works (Plain Language)

Precompute answers for all ranges whose lengths are powers of two. Let  $st[k][i]$  store the answer on the interval of length  $2^k$  starting at  $i$ , that is  $[i, i + 2^k - 1]$ .

Build recurrence:

- Base layer  $k = 0$ : intervals of length 1  $st[0][i] = A[i]$
- Higher layers: combine two halves of length  $2^{k-1}$   $st[k][i] = op(st[k-1][i], st[k-1][i + 2^{k-1}])$

To answer a query on  $[L, R]$ , let  $len = R - L + 1$ ,  $k = \text{floor}(\log_2(len))$ . For idempotent operations like `min` or `max`, we can cover the range with two overlapping blocks:

- Block 1:  $[L, L + 2^k - 1]$
- Block 2:  $[R - 2^k + 1, R]$  Then

$$\text{ans} = op(st[k][L], st[k][R - 2^k + 1])$$

## Example Step by Step

Array  $A = [7, 2, 3, 0, 5, 10, 3, 12, 18]$ ,  $op = \text{min}$ .

1. Build  $st[0]$  (length 1):  $st[0] = [7, 2, 3, 0, 5, 10, 3, 12, 18]$
2. Build  $st[1]$  (length 2):  $st[1][i] = \min(st[0][i], st[0][i+1])$   $st[1] = [2, 2, 0, 0, 5, 3, 3, 12]$
3. Build  $st[2]$  (length 4):  $st[2][i] = \min(st[1][i], st[1][i+2])$   $st[2] = [0, 0, 0, 0, 3, 3]$
4. Build  $st[3]$  (length 8):  $st[3][i] = \min(st[2][i], st[2][i+4])$   $st[3] = [0, 0]$

Query example: RMQ on  $[3, 8]$   $len = 6$ ,  $k = \text{floor}(\log_2(6)) = 2$ ,  $2^k = 4$

- Block 1:  $[3, 6]$  uses  $st[2][3]$
- Block 2:  $[5, 8]$  uses  $st[2][5]$  Answer

$$\min(st[2][3], st[2][5]) = \min(0, 3) = 0$$

## Tiny Code (Python, RMQ with min)

```

import math

def build_sparse_table(arr, op=min):
    n = len(arr)
    K = math.floor(math.log2(n)) + 1
    st = [[0] * n for _ in range(K)]
    for i in range(n):
        st[0][i] = arr[i]
    j = 1
    while (1 << j) <= n:
        step = 1 << (j - 1)
        for i in range(n - (1 << j) + 1):
            st[j][i] = op(st[j - 1][i], st[j - 1][i + step])
        j += 1
    # Precompute logs for O(1) queries
    lg = [0] * (n + 1)
    for i in range(2, n + 1):
        lg[i] = lg[i // 2] + 1
    return st, lg

def query(st, lg, L, R, op=min):
    length = R - L + 1
    k = lg[length]
    return op(st[k][L], st[k][R - (1 << k) + 1])

# Example
A = [7, 2, 3, 0, 5, 10, 3, 12, 18]
st, lg = build_sparse_table(A, op=min)
print(query(st, lg, 3, 8, op=min))  # 0
print(query(st, lg, 0, 2, op=min))  # 2

```

For `max`, just pass `op=max`. For `gcd`, pass `math.gcd`.

## Why It Matters

- $O(1)$  query time for static arrays
- $O(n \log n)$  preprocessing with simple transitions
- Excellent for RMQ style tasks, LCA via RMQ on Euler tours, and many competitive programming problems
- Cache friendly and implementation simple compared to segment trees when no updates are needed

## A Gentle Proof (Why It Works)

The table stores answers for all intervals of length  $2^k$ . Any interval  $[L, R]$  can be covered by two overlapping power of two blocks of equal length  $2^k$ , where  $k = \text{floor}(\log_2(R - L + 1))$ . For idempotent operations  $\text{op}$ , overlap does not affect correctness, so

$$\text{op}([L, R]) = \text{op}([L, L + 2^k - 1], [R - 2^k + 1, R])$$

Both blocks are precomputed, so the query is constant time.

## Try It Yourself

1. Build a table for  $A = [5, 4, 3, 6, 1, 2]$  with  $\text{op} = \min$ .
2. Answer RMQ on  $[1, 4]$  and  $[0, 5]$ .
3. Swap  $\text{op}$  to  $\max$  and recheck.
4. Use  $\text{op} = \gcd$  and verify results on several ranges.

## Test Cases

| Array                    | op     | Query    | Expected |
|--------------------------|--------|----------|----------|
| $[7, 2, 3, 0, 5, 10, 3]$ | $\min$ | $[0, 2]$ | 2        |
| $[7, 2, 3, 0, 5, 10, 3]$ | $\min$ | $[3, 6]$ | 0        |
| $[1, 5, 2, 4, 6, 1, 3]$  | $\max$ | $[2, 5]$ | 6        |
| $[12, 18, 6, 9, 3]$      | $\gcd$ | $[1, 4]$ | 3        |

## Complexity

| Phase      | Time          | Space         |
|------------|---------------|---------------|
| Preprocess | $O(n \log n)$ | $O(n \log n)$ |
| Query      | $O(1)$        | ,             |

Note Sparse Table supports static data. If you need updates, consider a segment tree or Fenwick tree. Sparse Table excels when the array is fixed and you need very fast queries.

## 292 Cartesian Tree

A Cartesian Tree is a binary tree built from an array such that:

1. In-order traversal of the tree reproduces the array, and
2. The tree satisfies the heap property with respect to the array values (min-heap or max-heap).

This structure elegantly bridges arrays and binary trees, and plays a key role in algorithms for Range Minimum Query (RMQ), Lowest Common Ancestor (LCA), and sequence decomposition.

### What Problem Are We Solving?

We want to represent an array  $A[0..n - 1]$  as a tree that encodes range relationships. For RMQ, if the tree is a min-heap Cartesian Tree, then the LCA of nodes  $i$  and  $j$  corresponds to the index of the minimum element in the range  $[i, j]$ .

Thus, building a Cartesian Tree gives us an elegant path from RMQ to LCA in  $O(1)$  after  $O(n)$  preprocessing.

### How It Works (Plain Language)

A Cartesian Tree is built recursively:

- The root is the smallest element (for min-heap) or largest (for max-heap).
- The left subtree is built from elements to the left of the root.
- The right subtree is built from elements to the right of the root.

A more efficient linear-time construction uses a stack:

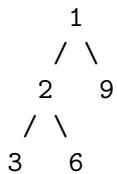
1. Traverse the array from left to right.
2. Maintain a stack of nodes in increasing order.
3. For each new element, pop while the top is greater, then attach the new node as the right child of the last popped node or the left child of the current top.

## Example

Let  $A = [3, 2, 6, 1, 9]$

1. Start with empty stack
2. Insert 3 → stack = [3]
3. Insert 2 → pop 3 (since  $3 > 2$ )
  - 2 becomes parent of 3
  - stack = [2]
4. Insert 6 → 6 > 2 → right child
  - stack = [2, 6]
5. Insert 1 → pop 6, pop 2 → 1 is new root
  - 2 becomes right child of 1
6. Insert 9 → right child of 6

Tree structure (min-heap):



In-order traversal: [3, 2, 6, 1, 9] Heap property: every parent is smaller than its children

## Tiny Code (Python, Min-Heap Cartesian Tree)

```
class Node:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None

def build_cartesian_tree(arr):
```

```

stack = []
root = None
for val in arr:
    node = Node(val)
    last = None
    while stack and stack[-1].val > val:
        last = stack.pop()
    node.left = last
    if stack:
        stack[-1].right = node
    else:
        root = node
    stack.append(node)
return root

def inorder(node):
    return inorder(node.left) + [node.val] + inorder(node.right) if node else []

```

Example usage:

```

A = [3, 2, 6, 1, 9]
root = build_cartesian_tree(A)
print(inorder(root)) # [3, 2, 6, 1, 9]

```

## Why It Matters

- RMQ in O(1): RMQ becomes LCA in Cartesian Tree (after Euler Tour + Sparse Table).
- Monotonic Stack Connection: Linear construction mirrors the logic of stack-based range problems (Next Greater Element, Histogram).
- Divide-and-Conquer Decomposition: Represents array's recursive structure.
- Efficient Building: Linear time with stack.

## A Gentle Proof (Why It Works)

Each element is pushed and popped at most once, so total operations =  $O(n)$ . Heap property ensures RMQ correctness:

- In min-heap tree, root of any subtree is the minimum of that segment.
- Thus, LCA(i, j) gives the index of  $\min(A[i..j])$ .

So by reducing RMQ to LCA, we achieve:

$$\text{RMQ}(i, j) = \text{index}(\text{LCA}(i, j))$$

### Try It Yourself

1. Build a Cartesian Tree for  $A = [4, 5, 2, 3, 1]$  (min-heap).
2. Verify in-order traversal equals original array.
3. Mark parents smaller than children.
4. Identify  $\text{RMQ}(1, 3)$  from the tree using LCA.

### Test Cases

| Array           | Tree Type | Root | $\text{RMQ}(1, 3)$ | Inorder Matches |
|-----------------|-----------|------|--------------------|-----------------|
| [3, 2, 6, 1, 9] | min-heap  | 1    | 2                  |                 |
| [5, 4, 3, 2, 1] | min-heap  | 1    | 2                  |                 |
| [1, 2, 3, 4, 5] | min-heap  | 1    | 2                  |                 |
| [2, 7, 5, 9]    | min-heap  | 2    | 5                  |                 |

### Complexity

| Operation      | Time          | Space         | Notes                      |
|----------------|---------------|---------------|----------------------------|
| Build          | $O(n)$        | $O(n)$        | Stack-based construction   |
| Query (RMQ)    | $O(1)$        | ,             | After Euler + Sparse Table |
| LCA Preprocess | $O(n \log n)$ | $O(n \log n)$ | Sparse Table method        |

A Cartesian Tree weaves together order and hierarchy: in-order for sequence, heap for dominance, a silent bridge between arrays and trees.

### 293 Segment Tree Beats

Segment Tree Beats is an advanced variant of the classical segment tree that can handle non-trivial range queries and updates beyond sum, min, or max. It's designed for problems where the operation is *not linear* or *not invertible*, such as range chmin/chmax, range add with min tracking, or range second-min queries.

It “beats” the limitation of classical lazy propagation by storing extra state (like second minimum, second maximum) to decide when updates can stop early.

## What Problem Are We Solving?

Standard segment trees can't efficiently handle complex updates like:

- “Set all  $A[i]$  in  $[L, R]$  to  $\min(A[i], x)$ ”
- “Set all  $A[i]$  in  $[L, R]$  to  $\max(A[i], x)$ ”

Because different elements in a segment may behave differently depending on their value relative to  $x$ .

Segment Tree Beats solves this by maintaining extra constraints in each node so we can “beat” recursion and skip branches early when conditions are met.

## How It Works (Plain Language)

Each node stores not just an aggregate but *enough info to know when an operation can be fully applied*.

For Range Chmin ( $A[i] = \min(A[i], x)$ ):

- If  $\max \leq x$ : nothing changes
- If  $\text{second\_max} < x < \max$ : only elements with  $\max$  are updated
- If  $x < \text{second\_max}$ : recurse to children

Each node stores:

- $\max$  (maximum value in range)
- $\text{second\_max}$  (second largest value)
- $\text{count\_max}$  (number of times  $\max$  appears)
- $\text{sum}$  (sum over range)

This lets us decide update logic without touching all elements.

## Example (Range Chmin)

Let  $A = [4, 7, 6, 3]$

We apply  $\text{chmin}(L=0, R=3, x=5) \rightarrow$  every element  $> 5$  is set to 5.

Step:

1. Node  $[0, 3]$ :  $\max = 7$ ,  $\text{second\_max} = 6$
2. Since  $x = 5 < \text{second\_max} = 6$ , recurse
3. Update left child  $[0, 1]$ : has 7  $\rightarrow$  7 becomes 5
4. Update right  $[2, 3]$ :  $\max = 6 \rightarrow$  6 becomes 5

New array: [4, 5, 5, 3]

### Tiny Code (Simplified Range Chmin)

```
class Node:
    def __init__(self):
        self.sum = 0
        self.max = 0
        self.smax = -float('inf')
        self.cnt = 0
        self.l = None
        self.r = None

    def merge(a, b):
        node = Node()
        node.sum = a.sum + b.sum
        if a.max == b.max:
            node.max = a.max
            node.smax = max(a.smax, b.smax)
            node.cnt = a.cnt + b.cnt
        elif a.max > b.max:
            node.max = a.max
            node.smax = max(a.smax, b.max)
            node.cnt = a.cnt
        else:
            node.max = b.max
            node.smax = max(a.max, b.smax)
            node.cnt = b.cnt
        return node

    def push_chmin(node, x, length):
        if node.max <= x:
            return
        node.sum -= (node.max - x) * node.cnt
        node.max = x

    def update_chmin(node, l, r, ql, qr, x):
        if r < ql or qr < l or node.max <= x:
            return
        if ql <= l and r <= qr and node.smax < x:
            push_chmin(node, x, r - l + 1)
```

```

    return
    m = (l + r) // 2
    update_chmin(node.l, l, m, ql, qr, x)
    update_chmin(node.r, m+1, r, ql, qr, x)
    new = merge(node.l, node.r)
    node.max, node.smax, node.cnt, node.sum = new.max, new.smax, new.cnt, new.sum

```

This is the essence: skip updates when possible, split when necessary.

## Why It Matters

- Handles non-linear updates efficiently
- Preserves logarithmic complexity by reducing unnecessary recursion
- Used in many competitive programming RMQ-like challenges with range cap operations
- Generalizes segment tree to “hard” problems (range `min` caps, range `max` caps, conditional sums)

## A Gentle Proof (Why It Works)

The trick: by storing max, second max, and count, we can stop descending if the operation affects only elements equal to max. At most  $O(\log n)$  nodes per update because:

- Each update lowers some max values
- Each element’s value decreases logarithmically before stabilizing

Thus total complexity amortizes to:

$$O((n + q) \log n)$$

## Try It Yourself

1. Build a Segment Tree Beats for  $A = [4, 7, 6, 3]$ .
2. Apply `chmin(0, 3, 5)` → verify  $[4, 5, 5, 3]$ .
3. Apply `chmin(0, 3, 4)` → verify  $[4, 4, 4, 3]$ .
4. Track `sum` after each operation.

## Test Cases

| Array      | Operation    | Result    |
|------------|--------------|-----------|
| [4,7,6,3]  | chmin(0,3,5) | [4,5,5,3] |
| [4,5,5,3]  | chmin(1,2,4) | [4,4,4,3] |
| [1,10,5,2] | chmin(0,3,6) | [1,6,5,2] |
| [5,5,5]    | chmin(0,2,4) | [4,4,4]   |

## Complexity

| Operation   | Time (Amortized) | Space  | Notes                       |
|-------------|------------------|--------|-----------------------------|
| Build       | $O(n)$           | $O(n)$ | Same as normal segment tree |
| Range Chmin | $O(\log n)$      | $O(n)$ | Amortized over operations   |
| Query (Sum) | $O(\log n)$      | ,      | Combine like usual          |

Segment Tree Beats lives at the intersection of elegance and power, retaining  $O(\log n)$  intuition while tackling the kinds of operations classical segment trees can't touch.

## 294 Merge Sort Tree

A Merge Sort Tree is a segment tree where each node stores a sorted list of the elements in its range. It allows efficient queries that depend on order statistics, such as counting how many elements fall within a range or finding the  $k$ -th smallest element in a subarray.

It's called "Merge Sort Tree" because it is built exactly like merge sort: divide, conquer, and merge sorted halves.

### What Problem Are We Solving?

Classical segment trees handle sum, min, or max, but not *value-based* queries. Merge Sort Trees enable operations like:

- Count how many numbers in  $A[L..R]$  are  $\leq x$
- Count elements in a value range  $[a, b]$
- Find the  $k$ -th smallest element in  $A[L..R]$

These problems often arise in range frequency queries, inversions counting, and offline queries.

## How It Works (Plain Language)

Each node of the tree covers a segment  $[l, r]$  of the array. Instead of storing a single number, it stores a sorted list of all elements in that segment.

Building:

1. If  $l == r$ , store  $[A[l]]$ .
2. Otherwise, recursively build left and right children.
3. Merge the two sorted lists to form this node's list.

Querying: To count numbers  $\leq x$  in range  $[L, R]$ :

- Visit all segment tree nodes that fully or partially overlap  $[L, R]$ .
- In each node, binary search for position of  $x$  in the node's sorted list.

## Example

Let  $A = [2, 5, 1, 4, 3]$

### Step 1: Build Tree

Each leaf stores one value:

| Node Range | Stored List |
|------------|-------------|
| $[0,0]$    | $[2]$       |
| $[1,1]$    | $[5]$       |
| $[2,2]$    | $[1]$       |
| $[3,3]$    | $[4]$       |
| $[4,4]$    | $[3]$       |

Now merge:

| Node Range | Stored List   |
|------------|---------------|
| $[0,1]$    | $[2,5]$       |
| $[2,3]$    | $[1,4]$       |
| $[2,4]$    | $[1,3,4]$     |
| $[0,4]$    | $[1,2,3,4,5]$ |

## Step 2: Query Example

Count elements  $\leq 3$  in range [1, 4].

We query nodes that cover [1,4]: [1,1], [2,3], [4,4].

- [1,1]  $\rightarrow$  list = [5]  $\rightarrow$  count = 0
- [2,3]  $\rightarrow$  list = [1,4]  $\rightarrow$  count = 1
- [4,4]  $\rightarrow$  list = [3]  $\rightarrow$  count = 1

Total = 2 elements 3.

## Tiny Code (Python, Count x)

```
import bisect

class MergeSortTree:
    def __init__(self, arr):
        self.n = len(arr)
        self.tree = [[] for _ in range(4 * self.n)]
        self._build(arr, 1, 0, self.n - 1)

    def _build(self, arr, node, l, r):
        if l == r:
            self.tree[node] = [arr[l]]
            return
        m = (l + r) // 2
        self._build(arr, node * 2, l, m)
        self._build(arr, node * 2 + 1, m + 1, r)
        self.tree[node] = sorted(self.tree[node * 2] + self.tree[node * 2 + 1])

    def query_leq(self, node, l, r, ql, qr, x):
        if r < ql or qr < l:
            return 0
        if ql <= l and r <= qr:
            return bisect.bisect_right(self.tree[node], x)
        m = (l + r) // 2
        return (self.query_leq(node * 2, l, m, ql, qr, x) +
                self.query_leq(node * 2 + 1, m + 1, r, ql, qr, x))

# Example
A = [2, 5, 1, 4, 3]
```

```
mst = MergeSortTree(A)
print(mst.query_leq(1, 0, 4, 1, 4, 3)) # count 3 in [1,4] = 2
```

## Why It Matters

- Enables order-based queries ( , , count, rank)
- Useful for offline range counting, kth-smallest, and inversion queries
- Combines divide-and-conquer sorting with range decomposition

## A Gentle Proof (Why It Works)

Each level of the tree merges sorted lists from children.

- Each element appears in  $O(\log n)$  nodes.
- Each merge is linear in subarray size.

Thus, total build time:

$$O(n \log n)$$

Each query visits  $O(\log n)$  nodes, each with  $O(\log n)$  binary search:

$$O(\log^2 n)$$

## Try It Yourself

1. Build tree for  $A = [5, 1, 4, 2, 3]$ .
2. Query count 3 in  $[0, 4]$ .
3. Query count 2 in  $[1, 3]$ .
4. Implement query for “count between  $[a, b]$ ” using two `query_leq` calls.

## Test Cases

| Array       | Query    | Condition | Answer |
|-------------|----------|-----------|--------|
| [2,5,1,4,3] | [1,4], 3 | count     | 2      |
| [2,5,1,4,3] | [0,2], 2 | count     | 2      |
| [1,2,3,4,5] | [2,4], 4 | count     | 3      |
| [5,4,3,2,1] | [0,4], 3 | count     | 3      |

## Complexity

| Operation           | Time          | Space         |
|---------------------|---------------|---------------|
| Build               | $O(n \log n)$ | $O(n \log n)$ |
| Query ( $x$ )       | $O(\log^2 n)$ | ,             |
| Query (range count) | $O(\log^2 n)$ | ,             |

Merge Sort Trees elegantly bridge sorting and segmentation, empowering range queries that depend not on aggregates, but on the distribution of values themselves.

## 295 Wavelet Tree

A Wavelet Tree is a compact, indexable structure over a sequence that supports rank, select, and range queries by value in  $O(\log \sigma)$  time with  $O(n \log \sigma)$  space, where  $(\cdot)$  is the alphabet size. Think of it as a value-aware segment tree built on bitvectors that let you jump between levels using rank counts.

### What Problem Are We Solving?

Given an array  $A[1..n]$  over values in  $[1..\sigma]$ , define the following queries:

- $\text{rank}(x, r)$ : number of occurrences of value  $x$  in  $A[1..r]$
- $\text{select}(x, k)$ : position of the  $k$ -th occurrence of  $x$  in  $A$
- $\text{kth}(l, r, k)$ : the  $k$ -th smallest value in the subarray  $A[l..r]$
- $\text{range\_count}(l, r, a, b)$ : number of values in  $A[l..r]$  that lie in  $[a, b]$

### How It Works

#### 1. Value partitioning by halves

Recursively partition the value domain  $[v_{\min}, v_{\max}]$  into two halves at midpoint  $m$ .

- Left child stores all elements  $\leq m$
- Right child stores all elements  $> m$

#### 2. Stable partition plus bitvector

At each node, keep the original order and record a bitvector  $B$  of length equal to the number of elements that arrive at the node:

- $B[i] = 0$  if the  $i$ -th element goes to the left child
- $B[i] = 1$  if it goes to the right child

Support fast ranks on  $B$ :  $\text{rank}_0(B, i)$  and  $\text{rank}_1(B, i)$ .

### 3. Navigating queries

- Position based descent: translate positions using prefix counts from  $B$ .  
If a query interval at this node is  $[l, r]$ , then the corresponding interval in the left child is  $[\text{rank}_0(B, l - 1) + 1, \text{rank}_0(B, r)]$  and in the right child is  $[\text{rank}_1(B, l - 1) + 1, \text{rank}_1(B, r)]$ .
- Value based descent: choose left or right child by comparing the value with  $m$ .

Height is  $O(\log \sigma)$ . Each step uses  $O(1)$  bitvector rank operations.

### Example

Array:  $A = [3, 1, 4, 1, 5, 9, 2, 6]$ , values in  $[1..9]$

Root split by midpoint  $m = 5$

- Left child receives elements  $\leq 5$ :  $\{3, 1, 4, 1, 5, 2\}$
- Right child receives elements  $> 5$ :  $\{9, 6\}$

Root bitvector marks routing left (0) or right (1), preserving order:

- $A$  by node:  $[3, 1, 4, 1, 5, 9, 2, 6]$
- $B$ :  $[0, 0, 0, 0, 0, 1, 0, 1]$

Left child split on  $[1..5]$  by  $m = 3$

- Left-left (values  $\leq 3$ ): positions where  $B = 0$  at root, then route by  $m = 3$   
Sequence arriving:  $[3, 1, 4, 1, 5, 2]$   
Bitvector at this node  $B_L$ :  $[0, 0, 1, 0, 1, 0]$   
Children:
  - $\leq 3$ :  $[3, 1, 1, 2]$
  - $> 3$ :  $[4, 5]$
- Right-right subtree of root contains  $[9, 6]$  (no further split shown)

Rank translation example at the root

For an interval  $[l, r]$  at the root, the corresponding intervals are:  
- Left child:  $[\text{rank}_0(B, l - 1) + 1, \text{rank}_0(B, r)]$   
- Right child:  $[\text{rank}_1(B, l - 1) + 1, \text{rank}_1(B, r)]$

Example: take  $[l, r] = [2, 7]$  at the root with  $B = [0, 0, 0, 0, 0, 1, 0, 1]$   
 $\text{rank}_0(B, 1) = 1, \text{rank}_0(B, 7) = 6 \rightarrow$  left interval  $[2, 6]$   
 $\text{rank}_1(B, 1) = 0, \text{rank}_1(B, 7) = 1 \rightarrow$  right interval  $[1, 1]$

Height is  $O(\log \sigma)$ , each descent step uses  $O(1)$  rank operations on the local bitvector.

## Core Operations

### 1. rank(x, r)

Walk top down. At a node with split value  $m$ :

- If  $x \leq m$ , set  $r \leftarrow \text{rank}_0(B, r)$  and go left
- Otherwise set  $r \leftarrow \text{rank}_1(B, r)$  and go right

When you reach the leaf for value  $x$ , the current  $r$  is the answer.

### 2. select(x, k)

Start at the leaf for value  $x$  with local index  $k$ .

Move upward to the root, inverting the position mapping at each parent:

- If you came from the left child, set  $k \leftarrow \text{select\_pos\_0}(B, k)$
- If you came from the right child, set  $k \leftarrow \text{select\_pos\_1}(B, k)$   
The final  $k$  at the root is the global position of the  $k$ -th  $x$ .

### 3. kth(l, r, k)

At a node with split value  $m$ , let

$$c = \text{rank}_0(B, r) - \text{rank}_0(B, l - 1)$$

which is the number of items routed left within  $[l, r]$ .

- If  $k \leq c$ , map the interval to the left child via

$$l' = \text{rank}_0(B, l - 1) + 1, \quad r' = \text{rank}_0(B, r)$$

and recurse on  $(l', r', k)$ .

- Otherwise go right with

$$k \leftarrow k - c, \quad l' = \text{rank}_1(B, l - 1) + 1, \quad r' = \text{rank}_1(B, r)$$

and recurse on  $(l', r', k)$ .

4. range\_count( $l, r, a, b$ )

Recurse only into value intervals that intersect  $[a, b]$ .

At each visited node, map the position interval using  $B$ :

- Left child interval

$$[\text{rank}_0(B, l - 1) + 1, \text{rank}_0(B, r)]$$

- Right child interval

$$[\text{rank}_1(B, l - 1) + 1, \text{rank}_1(B, r)]$$

Stop when a node's value range is fully inside or outside  $[a, b]$ :

- Fully inside: add its interval length

- Disjoint: add 0

### Tiny Code Sketch in Python

This sketch shows the structure and kth query. A production version needs succinct rank structures for (B) to guarantee ( $O(1)$ ) ranks.

```
import bisect

class WaveletTree:
    def __init__(self, arr, lo=None, hi=None):
        self.lo = min(arr) if lo is None else lo
        self.hi = max(arr) if hi is None else hi
        self.b = []          # bitvector as 0-1 list
        self.pref = [0]       # prefix sums of ones for O(1) rank1
        if self.lo == self.hi or not arr:
            self.left = self.right = None
            return
        mid = (self.lo + self.hi) // 2
        left_part, right_part = [], []
        for x in arr:
            go_right = 1 if x > mid else 0
            self.b.append(go_right)
            self.pref.append(self.pref[-1] + go_right)
            if go_right:
                right_part.append(x)
            else:
                left_part.append(x)
        self.left = WaveletTree(left_part, self.lo, mid)
        self.right = WaveletTree(right_part, mid + 1, self.hi)
```

```

def rank1(self, idx): # ones in b[1..idx]
    return self.pref[idx]

def rank0(self, idx): # zeros in b[1..idx]
    return idx - self.rank1(idx)

def kth(self, l, r, k):
    # 1-indexed positions
    if self.lo == self.hi:
        return self.lo
    mid = (self.lo + self.hi) // 2
    cnt_left = self.rank0(r) - self.rank0(l - 1)
    if k <= cnt_left:
        nl = self.rank0(l - 1) + 1
        nr = self.rank0(r)
        return self.left.kth(nl, nr, k)
    else:
        nl = self.rank1(l - 1) + 1
        nr = self.rank1(r)
        return self.right.kth(nl, nr, k - cnt_left)

```

Usage example:

```

A = [3,1,4,1,5,9,2,6]
wt = WaveletTree(A)
print(wt.kth(1, 8, 3)) # 3rd smallest in the whole array

```

## Why It Matters

- Combines value partitioning with positional stability, enabling order statistics on subranges
- Underpins succinct indexes, FM indexes, rank select dictionaries, and fast offline range queries
- Efficient when ( ) is moderate or compressible

## A Gentle Proof of Bounds

The tree has height  $O(\log \sigma)$  since each level halves the value domain. Each query descends one level and performs  $O(1)$  rank operations on a bitvector. Therefore

$$T_{\text{query}} = O(\log \sigma)$$

Space stores one bit per element per level on average, so

$$S = O(n \log \sigma)$$

With compressed bitvectors supporting constant time rank and select, these bounds hold in practice.

### Try It Yourself

1. Build a wavelet tree for  $A = [2, 7, 1, 8, 2, 8, 1]$ .
2. Compute  $\text{kth}(2, 6, 2)$ .
3. Compute  $\text{range\_count}(2, 7, 2, 8)$ .
4. Compare against a naive sort on  $A[l..r]$ .

### Test Cases

| Array             | Query                             | Answer |
|-------------------|-----------------------------------|--------|
| [3,1,4,1,5,9,2,6] | $\text{kth}(1, 8, 3)$             | 3      |
| [3,1,4,1,5,9,2,6] | $\text{range\_count}(3, 7, 2, 5)$ | 3      |
| [1,1,1,1,1]       | $\text{rank}(1, 5)$               | 5      |
| [5,4,3,2,1]       | $\text{kth}(2, 5, 2)$             | 3      |

### Complexity

| Operation        | Time               | Space              |
|------------------|--------------------|--------------------|
| Build            | $O(n \log \sigma)$ | $O(n \log \sigma)$ |
| rank, select     | $O(\log \sigma)$   | –                  |
| kth, range_count | $O(\log \sigma)$   | –                  |

Wavelet trees are a sharp tool for order aware range queries. By weaving bitvectors with stable partitions, they deliver succinct, logarithmic time answers on top of the original sequence.

### 296 KD-Tree

A KD-Tree (k-dimensional tree) is a binary space partitioning data structure for organizing points in a k-dimensional space. It enables fast range searches, nearest neighbor queries, and spatial indexing, often used in geometry, graphics, and machine learning (like k-NN).

## What Problem Are We Solving?

We need to store and query  $n$  points in  $k$ -dimensional space such that:

- Nearest neighbor query: find the point closest to a query  $q$ .
- Range query: find all points within a given region (rectangle or hypersphere).
- k-NN query: find the  $k$  closest points to  $q$ .

A naive search checks all  $n$  points, which takes  $O(n)$  time.

A KD-Tree reduces this to  $O(\log n)$  expected query time for balanced trees.

## How It Works

### 1. Recursive Partitioning by Dimension

At each level, the KD-Tree splits the set of points along one dimension, cycling through all dimensions.

If the current depth is  $d$ , use the axis  $a = d \bmod k$ :

- Sort points by their  $a$ -th coordinate.
- Choose the median point as the root to maintain balance.
- Left child: points with smaller  $a$ -th coordinate.
- Right child: points with larger  $a$ -th coordinate.

### 2. Search (Nearest Neighbor)

To find nearest neighbor of query  $q$ :

1. Descend tree following split planes (like BST).
2. Track current best (closest point found so far).
3. Backtrack if potential closer point exists across split plane (distance to plane < current best).

This ensures pruning of subtrees that cannot contain closer points.

## Example

Suppose 2D points:

$$P = (2, 3), (5, 4), (9, 6), (4, 7), (8, 1), (7, 2)$$

Step 1: Root splits by (x)-axis (axis 0). Sorted by (x): ((2,3), (4,7), (5,4), (7,2), (8,1), (9,6)). Median = ((7,2)). Root = (7,2).

Step 2:

- Left subtree (points with  $(x < 7)$ ) split by (y)-axis.
- Right subtree (points with  $(x > 7)$ ) split by (y)-axis.

This creates alternating partitions by x and y, forming axis-aligned rectangles.

## Tiny Code (Python)

```
class Node:  
    def __init__(self, point, axis):  
        self.point = point  
        self.axis = axis  
        self.left = None  
        self.right = None  
  
def build_kdtree(points, depth=0):  
    if not points:  
        return None  
    k = len(points[0])  
    axis = depth % k  
    points.sort(key=lambda p: p[axis])  
    median = len(points) // 2  
    node = Node(points[median], axis)  
    node.left = build_kdtree(points[:median], depth + 1)  
    node.right = build_kdtree(points[median + 1:], depth + 1)  
    return node
```

## How Nearest Neighbor Works

Given query ( $q$ ), maintain best distance ( $d_{best}$ ). For each visited node:

- Compute distance ( $d = |q - p|$ )
- If ( $d < d_{best}$ ), update best

- Check opposite branch only if ( $|q[a] - p[a]| < d_{\text{best}}$ )

### Example Table

| Step | Node Visited | Axis | Current Best | Distance to Plane | Search Next |
|------|--------------|------|--------------|-------------------|-------------|
| 1    | (7,2)        | x    | (7,2), 0.0   | 0.0               | Left        |
| 2    | (5,4)        | y    | (5,4), 2.8   | 2.0               | Left        |
| 3    | (2,3)        | x    | (5,4), 2.8   | 3.0               | Stop        |

### Why It Matters

- Efficient spatial querying in multidimensional data.
- Common in k-NN classification, computer graphics, and robotics pathfinding.
- Basis for libraries like `scipy.spatial.KDTree`.

### A Gentle Proof (Why It Works)

Each level splits points into halves, forming  $O(\log n)$  height. Each query visits a bounded number of nodes (dependent on dimension). Expected nearest neighbor cost:

$$T_{\text{query}} = O(\log n)$$

Building sorts points at each level:

$$T_{\text{build}} = O(n \log n)$$

### Try It Yourself

1. Build KD-Tree for 2D points  $\([(2,3),(5,4),(9,6),(4,7),(8,1),(7,2)]$ ).
2. Query nearest neighbor of  $(q = (9,2))$ .
3. Trace visited nodes, prune subtrees when possible.

| Points | Query | Nearest | Expected Path |
|--------|-------|---------|---------------|
|--------|-------|---------|---------------|

## Test Cases

| Points                                       | Query | Nearest | Expected Path                               |
|----------------------------------------------|-------|---------|---------------------------------------------|
| $[(2,3), (5,4), (9,6), (4,7), (8,1), (7,2)]$ | (9,2) | (8,1)   | Root $\rightarrow$ Right $\rightarrow$ Leaf |
| $[(1,1), (2,2), (3,3)]$                      | (2,3) | (2,2)   | Root $\rightarrow$ Right                    |
| $[(0,0), (10,10)]$                           | (5,5) | (10,10) | Root $\rightarrow$ Right                    |

## Complexity

| Operation        | Time                   | Space       |
|------------------|------------------------|-------------|
| Build            | $O(n \log n)$          | $O(n)$      |
| Nearest Neighbor | $O(\log n)$ (expected) | $O(\log n)$ |
| Range Query      | $O(n^{1-1/k} + m)$     | —           |

KD-Trees blend geometry with binary search, cutting space by dimensions to answer questions faster than brute force.

## 297 Range Tree

A Range Tree is a multi-level search structure for answering orthogonal range queries in multidimensional space, such as finding all points inside an axis-aligned rectangle. It extends 1D balanced search trees to higher dimensions using recursive trees on projections.

### What Problem Are We Solving?

Given a set of  $n$  points in  $k$ -dimensional space, we want to efficiently answer queries like:

“List all points  $(x, y)$  such that  $x_1 \leq x \leq x_2$  and  $y_1 \leq y \leq y_2$ .”

A naive scan is  $O(n)$  per query. Range Trees reduce this to  $O(\log^k n + m)$ , where  $m$  is the number of reported points.

## How It Works

### 1. 1D Case (Baseline)

A simple balanced BST (e.g. AVL) on  $x$  coordinates supports range queries by traversing paths and collecting nodes in range.

### 2. 2D Case (Extension)

- Build a primary tree on  $x$  coordinates.
- At each node, store a secondary tree built on  $y$  coordinates of the points in its subtree.

Each level recursively maintains sorted views along other axes.

### 3. Range Query

1. Search primary tree for split node  $s$ , where paths to  $x_1$  and  $x_2$  diverge.
2. For nodes fully in  $[x_1, x_2]$ , query their associated  $y$ -tree for  $[y_1, y_2]$ .
3. Combine results.

## Example

Given points:

$$P = (2, 3), (4, 7), (5, 1), (7, 2), (8, 5)$$

Query:  $[3, 7] \times [1, 5]$

- Primary tree on  $x$ : median  $(5, 1)$  as root.
- Secondary trees at each node on  $y$ .

Search path:

- Split node  $(5, 1)$  covers  $x \in [3, 7]$ .
- Visit subtrees with  $x \in [4, 7]$ .
- Query  $y$  in  $[1, 5]$  inside secondary trees.

Returned points:  $(5, 1), (7, 2), (4, 7) \rightarrow$  filter  $y \leq 5 \rightarrow (5, 1), (7, 2)$ .

## Tiny Code (Python-like Pseudocode)

```
class RangeTree:
    def __init__(self, points, depth=0):
        if not points:
            self.node = None
            return
        axis = depth % 2
        points.sort(key=lambda p: p[axis])
        mid = len(points) // 2
        self.node = points[mid]
        self.left = RangeTree(points[:mid], depth + 1)
        self.right = RangeTree(points[mid + 1:], depth + 1)
        self.sorted_y = sorted(points, key=lambda p: p[1])
```

Query recursively:

- Filter nodes by  $x$ .
- Binary search on  $y$ -lists.

## Step-by-Step Table (2D Query)

| Step | Operation         | Axis | Condition         | Action              |
|------|-------------------|------|-------------------|---------------------|
| 1    | Split at (5,1)    | x    | $3 \leq 5 \leq 7$ | Recurse both sides  |
| 2    | Left (2,3),(4,7)  | x    | $x < 5$           | Visit (4,7) subtree |
| 3    | Right (7,2),(8,5) | x    | $x \leq 7$        | Visit (7,2) subtree |
| 4    | Filter by $y$     | y    | $1 \leq y \leq 5$ | Keep (5,1),(7,2)    |

## Why It Matters

Range Trees provide deterministic performance for multidimensional queries, unlike kd-trees (which may degrade). They're ideal for:

- Orthogonal range counting
- Database indexing
- Computational geometry problems

They are static structures, suited when data doesn't change often.

## A Gentle Proof (Why It Works)

Each dimension adds a logarithmic factor. In 2D, build time:

$$T(n) = O(n \log n)$$

Query visits  $O(\log n)$  nodes in primary tree, each querying  $O(\log n)$  secondary trees:

$$Q(n) = O(\log^2 n + m)$$

Space is  $O(n \log n)$  due to secondary trees.

## Try It Yourself

1. Build a 2D Range Tree for  $P = (1, 2), (2, 3), (3, 4), (4, 5), (5, 6)$
2. Query rectangle  $[2, 4] \times [3, 5]$ .
3. Trace visited nodes and verify output.

## Test Cases

| Points                                 | Query Rectangle      | Result                |
|----------------------------------------|----------------------|-----------------------|
| $[(2,3), (4,7), (5,1), (7,2), (8,5)]$  | $[3,7] \times [1,5]$ | $(5,1), (7,2)$        |
| $[(1,2), (2,4), (3,6), (4,8), (5,10)]$ | $[2,4] \times [4,8]$ | $(2,4), (3,6), (4,8)$ |
| $[(1,1), (2,2), (3,3)]$                | $[1,2] \times [1,3]$ | $(1,1), (2,2)$        |

## Complexity

| Operation  | Time               | Space         |
|------------|--------------------|---------------|
| Build      | $O(n \log n)$      | $O(n \log n)$ |
| Query (2D) | $O(\log^2 n + m)$  | ,             |
| Update     | , (rebuild needed) | ,             |

Range Trees are precise geometric indexes: each axis divides the space, and nested trees give fast access to all points in any axis-aligned box.

## 298 Fenwick 2D Tree

A 2D Fenwick Tree (also known as 2D Binary Indexed Tree) extends the 1D Fenwick Tree to handle range queries and point updates on 2D grids such as matrices. It efficiently computes prefix sums and supports dynamic updates in  $O(\log^2 n)$  time.

### What Problem Are We Solving?

Given an  $n \times m$  matrix  $A$ , we want to support two operations efficiently:

1. Update: Add a value  $v$  to element  $A[x][y]$ .
2. Query: Compute the sum of all elements in submatrix  $[1..x][1..y]$ .

A naive approach takes  $O(nm)$  per query. The 2D Fenwick Tree reduces both update and query to  $O(\log n \cdot \log m)$ .

### How It Works

Each node  $(i, j)$  in the tree stores the sum of a submatrix region determined by the binary representation of its indices:

$$T[i][j] = \sum_{x=i-2^{r_i}+1}^i \sum_{y=j-2^{r_j}+1}^j A[x][y]$$

where  $r_i$  and  $r_j$  denote the least significant bit (LSB) of  $i$  and  $j$ .

### Update Rule

When updating  $(x, y)$  by  $v$ :

```
for i in range(x, n+1, i & -i):
    for j in range(y, m+1, j & -j):
        tree[i][j] += v
```

### Query Rule

To compute prefix sum  $(1, 1)$  to  $(x, y)$ :

```

res = 0
for i in range(x, 0, -i & -i):
    for j in range(y, 0, -j & -j):
        res += tree[i][j]
return res

```

## Range Query

Sum of submatrix  $[(x_1, y_1), (x_2, y_2)]$ :

$$S = Q(x_2, y_2) - Q(x_1 - 1, y_2) - Q(x_2, y_1 - 1) + Q(x_1 - 1, y_1 - 1)$$

## Example

Given a  $4 \times 4$  matrix:

| $x/y$ | 1 | 2 | 3 | 4 |
|-------|---|---|---|---|
| 1     | 2 | 1 | 0 | 3 |
| 2     | 1 | 2 | 3 | 1 |
| 3     | 0 | 1 | 2 | 0 |
| 4     | 4 | 0 | 1 | 2 |

Build Fenwick 2D Tree, then query sum in submatrix [2, 2] to [3, 3].

Expected result:

$$A[2][2] + A[2][3] + A[3][2] + A[3][3] = 2 + 3 + 1 + 2 = 8$$

## Step-by-Step Update Example

Suppose we add  $v = 5$  at  $(2, 3)$ :

| Step | $(i, j)$ Updated | Added Value | Reason                        |
|------|------------------|-------------|-------------------------------|
| 1    | $(2, 3)$         | +5          | Base position                 |
| 2    | $(2, 4)$         | +5          | Next by $j+ = j\& - j$        |
| 3    | $(4, 3)$         | +5          | Next by $i+ = i\& - i$        |
| 4    | $(4, 4)$         | +5          | Both indices propagate upward |

## Tiny Code (Python-like Pseudocode)

```
class Fenwick2D:
    def __init__(self, n, m):
        self.n, self.m = n, m
        self.tree = [[0]*(m+1) for _ in range(n+1)]

    def update(self, x, y, val):
        i = x
        while i <= self.n:
            j = y
            while j <= self.m:
                self.tree[i][j] += val
                j += j & -j
            i += i & -i

    def query(self, x, y):
        res = 0
        i = x
        while i > 0:
            j = y
            while j > 0:
                res += self.tree[i][j]
                j -= j & -j
            i -= i & -i
        return res

    def range_sum(self, x1, y1, x2, y2):
        return (self.query(x2, y2) - self.query(x1-1, y2)
               - self.query(x2, y1-1) + self.query(x1-1, y1-1))
```

## Why It Matters

2D Fenwick Trees are lightweight, dynamic structures for prefix sums and submatrix queries. They're widely used in:

- Image processing (integral image updates)
- Grid-based dynamic programming
- Competitive programming for 2D range queries

They trade slightly higher code complexity for excellent update-query efficiency.

## A Gentle Proof (Why It Works)

Each update/query operation performs  $\log n$  steps on  $x$  and  $\log m$  on  $y$ :

$$T(n, m) = O(\log n \cdot \log m)$$

Each level adds contributions from subregions determined by LSB decomposition, ensuring every cell contributes exactly once to the query sum.

## Try It Yourself

1. Initialize  $4 \times 4$  Fenwick 2D.
2. Add 5 to  $(2, 3)$ .
3. Query sum  $(1, 1)$  to  $(2, 3)$ .
4. Verify it matches manual computation.

## Test Cases

| Matrix (Partial)                    | Update       | Query Rectangle   | Result |
|-------------------------------------|--------------|-------------------|--------|
| $[[2, 1, 0], [1, 2, 3], [0, 1, 2]]$ | $(2, 3) + 5$ | $[1, 1] - [2, 3]$ | 14     |
| $[[1, 1, 1], [1, 1, 1], [1, 1, 1]]$ | $(3, 3) + 2$ | $[2, 2] - [3, 3]$ | 5      |
| $[[4, 0], [0, 4]]$                  | ,            | $[1, 1] - [2, 2]$ | 8      |

## Complexity

| Operation | Time                     | Space   |
|-----------|--------------------------|---------|
| Update    | $O(\log n \cdot \log m)$ | $O(nm)$ |
| Query     | $O(\log n \cdot \log m)$ | ,       |

The 2D Fenwick Tree is an elegant bridge between prefix sums and spatial queries, simple, powerful, and efficient for dynamic 2D grids.

## 299 Treap Split/Merge

A Treap Split/Merge algorithm allows you to divide and combine treaps (randomized balanced binary search trees) efficiently, using priority-based rotations and key-based ordering. This is the foundation for range operations like splits, merges, range updates, and segment queries on implicit treaps.

## What Problem Are We Solving?

We often need to:

1. Split a treap into two parts:
  - All keys  $\leq k$  go to the left treap
  - All keys  $> k$  go to the right treap
2. Merge two treaps  $T_1$  and  $T_2$  where all keys in  $T_1 < T_2$

These operations enable efficient range queries, persistent edits, and order-statistics while maintaining balanced height.

## How It Works (Plain Language)

Treaps combine two properties:

- BST Property: Left  $<$  Root  $<$  Right
- Heap Property: Node priority  $>$  children priorities

Split and merge rely on recursive descent guided by key and priority.

### Split Operation

Split treap  $T$  by key  $k$ :

- If  $T.key \leq k$ , split  $T.right$  into  $(t2a, t2b)$  and set  $T.right = t2a$
- Else, split  $T.left$  into  $(t1a, t1b)$  and set  $T.left = t1b$

Return  $(T.left, T.right)$

### Merge Operation

Merge  $T_1$  and  $T_2$ :

- If  $T_1.priority > T_2.priority$ , set  $T_1.right = \text{merge}(T_1.right, T_2)$
- Else, set  $T_2.left = \text{merge}(T_1, T_2.left)$

Return new root.

## Example

Suppose we have treap with keys:

[1, 2, 3, 4, 5, 6, 7]

**Split by  $k = 4$ :**

Left Treap: [1, 2, 3, 4] Right Treap: [5, 6, 7]

Now, merge them back restores original order.

## Step-by-Step Split Example

| Step | Node Key    | Compare to $k = 4$ | Action             |
|------|-------------|--------------------|--------------------|
| 1    | 4           | $\leq 4$           | Go right           |
| 2    | 5           | $> 4$              | Split left subtree |
| 3    | 5.left = \$ | return (null, 5)   | Combine back       |

Result: left = [1, 2, 3, 4], right = [5, 6, 7]

## Tiny Code (Python-like Pseudocode)

```
import random

class Node:
    def __init__(self, key):
        self.key = key
        self.priority = random.random()
        self.left = None
        self.right = None

def split(root, key):
    if not root:
        return (None, None)
    if root.key <= key:
        left, right = split(root.right, key)
        root.right = left
        return (root, right)
    else:
        left, right = split(root.left, key)
        root.left = right
        return (left, root)
```

```

else:
    left, right = split(root.left, key)
    root.left = right
    return (left, root)

def merge(t1, t2):
    if not t1 or not t2:
        return t1 or t2
    if t1.priority > t2.priority:
        t1.right = merge(t1.right, t2)
        return t1
    else:
        t2.left = merge(t1, t2.left)
        return t2

```

## Why It Matters

Treap split/merge unlocks flexible sequence manipulation and range-based operations:

- Range sum / min / max queries
- Insert or delete in  $O(\log n)$
- Persistent or implicit treaps for lists
- Lazy propagation for intervals

It's a key building block in functional and competitive programming data structures.

## A Gentle Proof (Why It Works)

Each split or merge operation traverses down the height of the treap. Since treaps are expected balanced, height is  $O(\log n)$ .

- Split correctness: Each recursive call preserves BST ordering.
- Merge correctness: Maintains heap property since highest priority becomes root.

Thus, both return valid treaps.

## Try It Yourself

1. Build treap with keys [1..7].
2. Split by  $k = 4$ .
3. Print inorder traversals of both sub-treaps.
4. Merge back. Confirm structure matches original.

## Test Cases

| Input Keys      | Split Key | Left Treap Keys | Right Treap Keys |
|-----------------|-----------|-----------------|------------------|
| [1,2,3,4,5,6,7] | 4         | [1,2,3,4]       | [5,6,7]          |
| [10,20,30,40]   | 25        | [10,20]         | [30,40]          |
| [5,10,15,20]    | 5         | [5]             | [10,15,20]       |

## Complexity

| Operation | Time        | Space  |
|-----------|-------------|--------|
| Split     | $O(\log n)$ | $O(1)$ |
| Merge     | $O(\log n)$ | $O(1)$ |

Treap Split/Merge is the elegant heart of many dynamic set and sequence structures, one key, one random priority, two simple operations, infinite flexibility.

## 300 Mo's Algorithm on Tree

Mo's Algorithm on Tree is an extension of the classical Mo's algorithm used on arrays. It allows efficient processing of offline queries on trees, especially those involving subtrees or paths, by converting them into a linear order (Euler tour) and then applying a square root decomposition strategy.

### What Problem Are We Solving?

When you need to answer multiple queries like:

- “How many distinct values are in the subtree of node  $u$ ?”
- “What is the sum over the path from  $u$  to  $v$ ?”

A naive approach may require  $O(n)$  traversal per query, leading to  $O(nq)$  total complexity. Mo's algorithm on trees reduces this to approximately  $O((n + q)\sqrt{n})$ , by reusing results from nearby queries.

## How It Works (Plain Language)

1. Euler Tour Flattening Transform the tree into a linear array using an Euler tour. Each node's first appearance marks its position in the linearized sequence.
2. Query Transformation
  - For subtree queries, a subtree becomes a continuous range in the Euler array.
  - For path queries, break into two subranges and handle Least Common Ancestor (LCA) separately.
3. Mo's Ordering Sort queries by:
  - Block of left endpoint (using block =  $\lfloor L/\sqrt{N} \rfloor$ )
  - Right endpoint (ascending or alternating per block)
4. Add/Remove Function Maintain a frequency map or running result as the window moves.

## Example

Given a tree:

```
1
2
  4
  5
3
```

Euler Tour: [1, 2, 4, 4, 5, 5, 2, 3, 3, 1]

Subtree(2): Range covering [2, 4, 4, 5, 5, 2]

Each subtree query becomes a range query over the Euler array. Mo's algorithm processes ranges efficiently in sorted order.

## Step-by-Step Example

| Step | Query (L,R) | Current Range | Add/Remove | Result  |
|------|-------------|---------------|------------|---------|
| 1    | (2,6)       | [2,6]         | +4,+5      | Count=2 |
| 2    | (2,8)       | [2,8]         | +3         | Count=3 |
| 3    | (1,6)       | [1,6]         | -3,+1      | Count=3 |

Each query is answered by incremental adjustment, not recomputation.

## Tiny Code (Python-like Pseudocode)

```
import math

# Preprocessing
def euler_tour(u, p, g, order):
    order.append(u)
    for v in g[u]:
        if v != p:
            euler_tour(v, u, g, order)
    order.append(u)

# Mo's structure
class Query:
    def __init__(self, l, r, idx):
        self.l, self.r, self.idx = l, r, idx

def mo_on_tree(n, queries, order, value):
    block = int(math.sqrt(len(order)))
    queries.sort(key=lambda q: (q.l // block, q.r))

    freq = [0]*(n+1)
    answer = [0]*len(queries)
    cur = 0
    L, R = 0, -1

    def add(pos):
        nonlocal cur
        node = order[pos]
        freq[node] += 1
        if freq[node] == 1:
            cur += value[node]

    def remove(pos):
        nonlocal cur
        node = order[pos]
        freq[node] -= 1
        if freq[node] == 0:
            cur -= value[node]

    for q in queries:
        while L > q.l:
```

```

L == 1
add(L)
while R < q.r:
    R += 1
    add(R)
while L < q.l:
    remove(L)
    L += 1
while R > q.r:
    remove(R)
    R -= 1
answer[q.idx] = cur
return answer

```

## Why It Matters

- Converts tree queries into range queries efficiently
- Reuses computation by sliding window technique
- Useful for frequency, sum, or distinct count queries
- Supports subtree queries, path queries (with LCA handling), and color-based queries

## A Gentle Proof (Why It Works)

1. Euler Tour guarantees every subtree is a contiguous range.
2. Mo's algorithm ensures total number of add/remove operations is  $O((n + q)\sqrt{n})$ .
3. Combining them, each query is handled incrementally within logarithmic amortized cost.

Thus, offline complexity is sublinear per query.

## Try It Yourself

1. Build Euler tour for tree of 7 nodes.
2. Write subtree queries for nodes 2, 3, 4.
3. Sort queries by block order.
4. Implement add/remove logic to count distinct colors or sums.
5. Compare performance with naive DFS per query.

## Test Cases

| Nodes               | Query      | Expected Result             |
|---------------------|------------|-----------------------------|
| [1-2-3-4-5]         | Subtree(2) | Sum or count over nodes 2–5 |
| [1: {2,3}, 2:{4,5}] | Subtree(1) | All nodes                   |
| [1-2,1-3]           | Path(2,3)  | LCA=1 handled separately    |

## Complexity

| Operation             | Time                 | Space  |
|-----------------------|----------------------|--------|
| Preprocessing (Euler) | $O(n)$               | $O(n)$ |
| Query Sorting         | $O(q \log q)$        | $O(q)$ |
| Processing            | $O((n + q)\sqrt{n})$ | $O(n)$ |

Mo's Algorithm on Tree is the elegant meeting point of graph traversal, offline range query, and amortized optimization, bringing sublinear query handling to complex hierarchical data.

# Chapter 4. Graph Algorithms

## Section 31. Traversals

### 301 Depth-First Search (Recursive)

Depth-First Search (DFS) is the classic way to explore a graph deeply before backtracking. You pick a starting node, walk as far as possible along one path, and only when you hit a dead end do you turn back. It's the "go deep first, ask questions later" of graph algorithms.

#### What Problem Are We Solving?

DFS helps us visit every vertex and edge in a connected component systematically. It's the foundation for exploring graphs, detecting cycles, classifying edges, and building more complex algorithms like topological sort, strongly connected components, and articulation point detection.

We want an algorithm that:

- Explores all reachable vertices from a start node
- Avoids revisiting nodes
- Records traversal order

Example: You have a maze. DFS is the explorer that picks a path, goes as far as it can, and only turns back when stuck.

#### How Does It Work (Plain Language)?

Think of DFS like a curious traveler: always dive deeper whenever you see a new path. When you can't go further, step back one level and continue exploring.

We use recursion to model this behavior naturally, each recursive call represents entering a new node, and returning means backtracking.

| Step | Current Node                 | Action        | Stack (Call Path) |
|------|------------------------------|---------------|-------------------|
| 1    | A                            | Visit A       | [A]               |
| 2    | B                            | Visit B (A→B) | [A, B]            |
| 3    | D                            | Visit D (B→D) | [A, B, D]         |
| 4    | D has no unvisited neighbors | Backtrack     | [A, B]            |
| 5    | B's next neighbor C          | Visit C       | [A, B, C]         |
| 6    | C done                       | Backtrack     | [A, B] → [A]      |
| 7    | A's remaining neighbors      | Visit next    | [...]             |

When recursion unwinds, we've explored the whole reachable graph.

### Tiny Code (Easy Versions)

C (Adjacency List Example)

```
#include <stdio.h>
#include <stdbool.h>

#define MAX 100
int graph[MAX][MAX];
bool visited[MAX];
int n;

void dfs(int v) {
    visited[v] = true;
    printf("%d ", v);
    for (int u = 0; u < n; u++) {
        if (graph[v][u] && !visited[u]) {
            dfs(u);
        }
    }
}

int main(void) {
    printf("Enter number of vertices: ");
    scanf("%d", &n);

    printf("Enter adjacency matrix:\n");
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
```

```

        scanf("%d", &graph[i][j]);

    printf("DFS starting from vertex 0:\n");
    dfs(0);
}

```

Python (Adjacency List)

```

graph = {
    0: [1, 2],
    1: [2],
    2: [0, 3],
    3: [3]
}

visited = set()

def dfs(v):
    visited.add(v)
    print(v, end=" ")
    for u in graph[v]:
        if u not in visited:
            dfs(u)

dfs(0)

```

## Why It Matters

- Core for graph exploration and reachability
- Forms the basis of topological sort, SCC, bridges, and cycles
- Simple recursive structure reveals natural hierarchy of a graph
- Helps understand backtracking and stack-based thinking

## A Gentle Proof (Why It Works)

Each vertex is visited exactly once:

- When a node is first discovered, it's marked `visited`
- The recursion ensures all its neighbors are explored
- Once all children are done, the function returns (backtrack)

So every vertex  $v$  triggers one call `dfs(v)`, giving  $O(V + E)$  time (each edge explored once).

## Try It Yourself

1. Draw a small graph (A–B–C–D) and trace DFS.
2. Modify to print entry and exit times.
3. Track parent nodes to build DFS tree.
4. Add detection for back edges (cycle test).
5. Compare with BFS traversal order.

## Test Cases

| Graph                     | Start | Expected Order | Notes                          |
|---------------------------|-------|----------------|--------------------------------|
| A–B–C–D chain             | A     | A B C D        | Straight path                  |
| Triangle (A–B–C–A)        | A     | A B C          | Visits all, stops at visited A |
| Disconnected {A–B}, {C–D} | A     | A B            | Only reachable component       |
| Directed A→B→C            | A     | A B C          | Linear chain                   |
| Tree root 0               | 0     | 0 1 3 4 2 5    | Depends on adjacency order     |

## Complexity

- Time:  $O(V + E)$
- Space:  $O(V)$  (recursion stack + visited)

DFS is your first lens into graph structure, recursive, elegant, and revealing hidden pathways one stack frame at a time.

## 302 Depth-First Search (Iterative)

Depth-First Search can run without recursion too. Instead of leaning on the call stack, we build our own stack explicitly. It's the same journey, diving deep before backtracking, just with manual control over what's next.

### What Problem Are We Solving?

Recursion is elegant but not always practical. Some graphs are deep, and recursive DFS can overflow the call stack. The iterative version solves that by using a stack data structure directly, mirroring the same traversal order.

We want an algorithm that:

- Works even when recursion is too deep

- Explicitly manages visited nodes and stack
- Produces the same traversal as recursive DFS

Example: Think of it like keeping your own to-do list of unexplored paths, each time you go deeper, you add new destinations on top of the stack.

### How Does It Work (Plain Language)?

We maintain a stack:

1. Start from a node  $s$ , push it on the stack.
2. Pop the top node  $v$ .
3. If  $v$  is unvisited, mark and process it.
4. Push all unvisited neighbors of  $v$  onto the stack.
5. Repeat until the stack is empty.

| Step | Stack (Top → Bottom) | Action              | Visited      |
|------|----------------------|---------------------|--------------|
| 1    | [A]                  | Start, pop A        | {A}          |
| 2    | [B, C]               | Push neighbors of A | {A}          |
| 3    | [C, B]               | Pop B, visit B      | {A, B}       |
| 4    | [C, D]               | Push neighbors of B | {A, B}       |
| 5    | [D, C]               | Pop D, visit D      | {A, B, D}    |
| 6    | C                    | Continue            | {A, B, D}    |
| 7    | []                   | Pop C, visit C      | {A, B, C, D} |

### Tiny Code (Easy Versions)

C (Adjacency Matrix Example)

```
#include <stdio.h>
#include <stdbool.h>

#define MAX 100
int graph[MAX][MAX];
bool visited[MAX];
int stack[MAX];
int top = -1;
int n;

void push(int v) { stack[++top] = v; }
int pop() { return stack[top--]; }
```

```

bool is_empty() { return top == -1; }

void dfs_iterative(int start) {
    push(start);
    while (!is_empty()) {
        int v = pop();
        if (!visited[v]) {
            visited[v] = true;
            printf("%d ", v);
            for (int u = n - 1; u >= 0; u--) { // reverse for consistent order
                if (graph[v][u] && !visited[u])
                    push(u);
            }
        }
    }
}

int main(void) {
    printf("Enter number of vertices: ");
    scanf("%d", &n);

    printf("Enter adjacency matrix:\n");
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            scanf("%d", &graph[i][j]);

    printf("Iterative DFS from 0:\n");
    dfs_iterative(0);
}

```

Python (Using List as Stack)

```

graph = {
    0: [1, 2],
    1: [2],
    2: [0, 3],
    3: [3]
}

visited = set()
stack = [0]

```

```

while stack:
    v = stack.pop()
    if v not in visited:
        visited.add(v)
        print(v, end=" ")
        for u in reversed(graph[v]): # reversed for DFS-like order
            if u not in visited:
                stack.append(u)

```

## Why It Matters

- Avoids recursion limits and stack overflow
- Clear control over traversal order
- Good for systems with limited call stack
- Builds understanding of explicit stack simulation

## A Gentle Proof (Why It Works)

The stack mimics recursion: Each vertex  $v$  is processed once when popped, and its neighbors are pushed. Every edge is examined exactly once. So total operations =  $O(V + E)$ , same as recursive DFS.

Each push = one recursive call; each pop = one return.

## Try It Yourself

1. Trace iterative DFS on a small graph.
2. Compare the order with the recursive version.
3. Experiment with neighbor push order, see how output changes.
4. Add discovery and finishing times.
5. Convert to iterative topological sort by pushing finishing order to a second stack.

## Test Cases

| Graph         | Start | Order (One Possible) | Notes                     |
|---------------|-------|----------------------|---------------------------|
| 0–1–2 chain   | 0     | 0 1 2                | Simple path               |
| 0→1, 0→2, 1→3 | 0     | 0 1 3 2              | Depends on neighbor order |
| Cycle 0→1→2→0 | 0     | 0 1 2                | No repeats                |
| Disconnected  | 0     | 0 1 2                | Only connected part       |

| Graph                    | Start | Order (One Possible) | Notes           |
|--------------------------|-------|----------------------|-----------------|
| Complete graph (4 nodes) | 0     | 0 1 2 3              | Visits all once |

## Complexity

- Time:  $O(V + E)$
- Space:  $O(V)$  for stack and visited array

Iterative DFS is your manual-gear version of recursion, same depth, same discovery, just no surprises from the call stack.

## 303 Breadth-First Search (Queue)

Breadth-First Search (BFS) is the explorer that moves level by level, radiating outward from the start. Instead of diving deep like DFS, BFS keeps things fair, it visits all neighbors before going deeper.

### What Problem Are We Solving?

We want a way to:

- Explore all reachable vertices in a graph
- Discover the shortest path in unweighted graphs
- Process nodes in increasing distance order

BFS is perfect when:

- Edges all have equal weight (like 1)
- You need the fewest steps to reach a goal
- You're finding connected components, levels, or distances

Example: Imagine spreading a rumor. Each person tells all their friends before the next wave begins, that's BFS in action.

## How Does It Work (Plain Language)?

BFS uses a queue, a first-in, first-out line.

1. Start from a node  $s$
2. Mark it visited and enqueue it
3. While queue not empty:
  - Dequeue front node  $v$
  - Visit  $v$
  - Enqueue all unvisited neighbors of  $v$

| Step | Queue (Front → Back) | Visited            | Action                     |
|------|----------------------|--------------------|----------------------------|
| 1    | [A]                  | {A}                | Start                      |
| 2    | [B, C]               | {A, B, C}          | A's neighbors              |
| 3    | [C, D, E]            | {A, B, C, D, E}    | Visit B, add its neighbors |
| 4    | [D, E, F]            | {A, B, C, D, E, F} | Visit C, add neighbors     |
| 5    | [E, F]               | {A...F}            | Continue until empty       |

The order you dequeue = level order traversal.

## Tiny Code (Easy Versions)

C (Adjacency Matrix Example)

```
#include <stdio.h>
#include <stdbool.h>

#define MAX 100
int graph[MAX][MAX];
bool visited[MAX];
int queue[MAX];
int front = 0, rear = 0;
int n;

void enqueue(int v) { queue[rear++] = v; }
int dequeue() { return queue[front++]; }
bool is_empty() { return front == rear; }

void bfs(int start) {
```

```

visited[start] = true;
enqueue(start);

while (!is_empty()) {
    int v = dequeue();
    printf("%d ", v);

    for (int u = 0; u < n; u++) {
        if (graph[v][u] && !visited[u]) {
            visited[u] = true;
            enqueue(u);
        }
    }
}

int main(void) {
    printf("Enter number of vertices: ");
    scanf("%d", &n);

    printf("Enter adjacency matrix:\n");
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            scanf("%d", &graph[i][j]);

    printf("BFS starting from vertex 0:\n");
    bfs(0);
}

```

Python (Adjacency List)

```

from collections import deque

graph = {
    0: [1, 2],
    1: [3, 4],
    2: [5],
    3: [],
    4: [],
    5: []
}

```

```

visited = set()
queue = deque([0])
visited.add(0)

while queue:
    v = queue.popleft()
    print(v, end=" ")
    for u in graph[v]:
        if u not in visited:
            visited.add(u)
            queue.append(u)

```

## Why It Matters

- Finds shortest paths in unweighted graphs
- Guarantees level order visitation
- Core for algorithms like 0–1 BFS, SPFA, and Dijkstra’s
- Excellent for layer-based exploration and distance labeling

## A Gentle Proof (Why It Works)

Each vertex is visited exactly once:

- It’s enqueueued when discovered
- It’s dequeued once for processing
- Each edge is checked once

If all edges have weight 1, BFS discovers vertices in increasing distance order, proving shortest-path correctness.

Time complexity:

- Visiting each vertex:  $O(V)$
- Scanning each edge:  $O(E) \rightarrow$  Total:  $O(V + E)$

## Try It Yourself

1. Draw a small unweighted graph and run BFS by hand.
2. Record levels (distance from start).
3. Track parent of each vertex, reconstruct shortest path.
4. Try BFS on a tree, compare with level-order traversal.
5. Experiment on disconnected graphs, note what gets missed.

## Test Cases

| Graph                          | Start | Order   | Distance            |
|--------------------------------|-------|---------|---------------------|
| 0–1–2 chain                    | 0     | 0 1 2   | [0,1,2]             |
| Triangle 0–1–2                 | 0     | 0 1 2   | [0,1,1]             |
| Star $0 \rightarrow \{1,2,3\}$ | 0     | 0 1 2 3 | [0,1,1,1]           |
| Grid $2 \times 2$              | 0     | 0 1 2 3 | Layered             |
| Disconnected                   | 0     | 0 1     | Only component of 0 |

## Complexity

- Time:  $O(V + E)$
- Space:  $O(V)$  for queue and visited set

BFS is your wavefront explorer, fair, systematic, and always shortest when edges are equal.

## 304 Iterative Deepening DFS

Iterative Deepening Depth-First Search (IDDFS) blends the depth control of BFS with the space efficiency of DFS. It repeatedly performs DFS with increasing depth limits, uncovering nodes level by level, but through deep-first exploration each time.

### What Problem Are We Solving?

Pure DFS may wander too deep, missing nearer solutions. Pure BFS finds shortest paths but consumes large memory.

We need a search that:

- Finds shallowest solution like BFS
- Uses  $O(\text{depth})$  memory like DFS
- Works in infinite or very large search spaces

That's where IDDFS shines, it performs a DFS up to a limit, then restarts with a deeper limit, repeating until the goal is found.

Example: Think of a diver who explores deeper with each dive, 1 meter, 2 meters, 3 meters, always sweeping from the surface down.

## How Does It Work (Plain Language)?

Each iteration increases the depth limit by one. At each stage, we perform a DFS that stops when depth exceeds the current limit.

1. Set limit = 0
2. Run DFS with depth limit = 0
3. If not found, increase limit and repeat
4. Continue until goal found or all explored

| Iteration | Depth Limit | Nodes Explored    | Found Goal? |
|-----------|-------------|-------------------|-------------|
| 1         | 0           | Start node        | No          |
| 2         | 1           | Start + neighbors | No          |
| 3         | 2           | + deeper nodes    | Possibly    |
| ...       | ...         | ...               | ...         |

Although nodes are revisited, total cost remains efficient, like BFS's layer-wise discovery.

## Tiny Code (Easy Versions)

C (Depth-Limited DFS)

```
#include <stdio.h>
#include <stdbool.h>

#define MAX 100
int graph[MAX][MAX];
int n;
bool found = false;

void dls(int v, int depth, int limit, bool visited[]) {
    visited[v] = true;
    printf("%d ", v);
    if (depth == limit) return;
    for (int u = 0; u < n; u++) {
        if (graph[v][u] && !visited[u]) {
            dls(u, depth + 1, limit, visited);
        }
    }
}
```

```

void iddfs(int start, int max_depth) {
    for (int limit = 0; limit <= max_depth; limit++) {
        bool visited[MAX] = {false};
        printf("\nDepth limit %d: ", limit);
        dls(start, 0, limit, visited);
    }
}

int main(void) {
    printf("Enter number of vertices: ");
    scanf("%d", &n);

    printf("Enter adjacency matrix:\n");
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            scanf("%d", &graph[i][j]);

    iddfs(0, 3);
}

```

Python (Adjacency List + Depth Limit)

```

graph = {
    0: [1, 2],
    1: [3],
    2: [4],
    3: [],
    4: []
}

def dls(v, depth, limit, visited):
    visited.add(v)
    print(v, end=" ")
    if depth == limit:
        return
    for u in graph[v]:
        if u not in visited:
            dls(u, depth + 1, limit, visited)

def iddfs(start, max_depth):
    for limit in range(max_depth + 1):
        print(f"\nDepth limit {limit}: ", end=" ")

```

```

visited = set()
dls(start, 0, limit, visited)

iddfs(0, 3)

```

## Why It Matters

- Combines advantages of BFS and DFS
- Finds optimal solution in unweighted graphs
- Uses linear space
- Ideal for state-space search (AI, puzzles)

## A Gentle Proof (Why It Works)

BFS guarantees shortest path; DFS uses less space. IDDFS repeats DFS with increasing limits, ensuring that:

- All nodes at depth  $d$  are visited before depth  $d+1$
- Space =  $O(d)$
- Time  $O(b^d)$ , similar to BFS in order

Redundant work (revisiting nodes) is small compared to total nodes in deeper layers.

## Try It Yourself

1. Run IDDFS on a tree; observe repeated shallow visits.
2. Count nodes visited per iteration.
3. Compare total visits with BFS.
4. Modify depth limit mid-run, what happens?
5. Use IDDFS to find a goal node at depth 3.

## Test Cases

| Graph                                                   | Goal   | Max Depth | Found At | Order Example |
|---------------------------------------------------------|--------|-----------|----------|---------------|
| $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$           | 3      | 3         | Depth 3  | 0 1 2 3       |
| $0 \rightarrow \{1,2\}$ , $1 \rightarrow 3$             | 3      | 3         | Depth 2  | 0 1 3         |
| Star $0 \rightarrow \{1,2,3\}$                          | 3      | 1         | Depth 1  | 0 1 2 3       |
| $0 \rightarrow 1 \rightarrow 2 \rightarrow \text{Goal}$ | Goal=2 | 2         | Depth 2  | 0 1 2         |

## Complexity

- Time:  $O(b^d)$  (like BFS)
- Space:  $O(d)$  (like DFS)

Iterative Deepening DFS is the patient climber, revisiting familiar ground, going deeper each time, ensuring no shallow treasure is missed.

## 305 Bidirectional BFS

Bidirectional BFS is the meet-in-the-middle version of BFS. Instead of starting from one end and exploring everything outward, we launch two BFS waves, one from the source and one from the target, and stop when they collide in the middle. It's like digging a tunnel from both sides of a mountain to meet halfway.

### What Problem Are We Solving?

Standard BFS explores the entire search space outward from the start until it reaches the goal, great for small graphs, but expensive when the graph is huge.

Bidirectional BFS cuts that exploration dramatically by searching both directions at once, halving the effective search depth.

We want an algorithm that:

- Finds the shortest path in an unweighted graph
- Explores fewer nodes than single-source BFS
- Stops as soon as the two waves meet

Example: You're finding the shortest route between two cities. Instead of exploring from one city across the whole map, you also send scouts from the destination. They meet somewhere, the midpoint of the shortest path.

### How Does It Work (Plain Language)?

Run two BFS searches simultaneously, one forward, one backward. At each step, expand the smaller frontier first to balance work. Stop when any node appears in both visited sets.

1. Start BFS from **source** and **target**
2. Maintain two queues and two visited sets
3. Alternate expansions
4. When visited sets overlap, meeting point found
5. Combine paths for the final route

| Step | Forward Queue | Backward Queue | Intersection | Action             |
|------|---------------|----------------|--------------|--------------------|
| 1    | [S]           | [T]            |              | Start              |
| 2    | [S1, S2]      | [T1, T2]       |              | Expand both        |
| 3    | [S2, S3]      | [T1, S2]       | S2           | Found meeting node |

### Tiny Code (Easy Versions)

C (Simplified Version)

```
#include <stdio.h>
#include <stdbool.h>

#define MAX 100
int graph[MAX][MAX];
int n;

bool bfs_step(bool visited[], int queue[], int *front, int *rear) {
    int size = *rear - *front;
    while (size--) {
        int v = queue[(*front)++];
        for (int u = 0; u < n; u++) {
            if (graph[v][u] && !visited[u]) {
                visited[u] = true;
                queue[(*rear)++] = u;
            }
        }
    }
    return false;
}

bool intersect(bool a[], bool b[]) {
    for (int i = 0; i < n; i++)
        if (a[i] && b[i]) return true;
    return false;
}

bool bidir_bfs(int src, int dest) {
    bool vis_s[MAX] = {false}, vis_t[MAX] = {false};
    int qs[MAX], qt[MAX];
    int fs = 0, rs = 0, ft = 0, rt = 0;
    qs[rs++] = src; vis_s[src] = true;
```

```

qt[rt++] = dest; vis_t[dest] = true;

while (fs < rs && ft < rt) {
    if (bfs_step(vis_s, qs, &fs, &rs)) return true;
    if (intersect(vis_s, vis_t)) return true;

    if (bfs_step(vis_t, qt, &ft, &rt)) return true;
    if (intersect(vis_s, vis_t)) return true;
}

return false;
}

int main(void) {
    printf("Enter number of vertices: ");
    scanf("%d", &n);
    printf("Enter adjacency matrix:\n");
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            scanf("%d", &graph[i][j]);

    int src = 0, dest = n - 1;
    if (bidir_bfs(src, dest))
        printf("Path found\n");
    else
        printf("No path\n");
}

```

Python (Readable Version)

```

from collections import deque

graph = {
    0: [1, 2],
    1: [3],
    2: [4],
    3: [5],
    4: [5],
    5: []
}

def bidirectional_bfs(src, dest):
    if src == dest:

```

```

    return True

q1, q2 = deque([src]), deque([dest])
visited1, visited2 = {src}, {dest}

while q1 and q2:
    # Expand forward
    for _ in range(len(q1)):
        v = q1.popleft()
        for u in graph[v]:
            if u in visited2:
                return True
            if u not in visited1:
                visited1.add(u)
                q1.append(u)

    # Expand backward
    for _ in range(len(q2)):
        v = q2.popleft()
        for u in graph[v]:
            if u in visited1:
                return True
            if u not in visited2:
                visited2.add(u)
                q2.append(u)

return False

print(bidirectional_bfs(0, 5))

```

## Why It Matters

- Faster shortest-path search on large graphs
- Reduces explored nodes from  $O(b^d)$  to roughly  $O(b^{(d/2)})$
- Excellent for pathfinding in maps, puzzles, or networks
- Demonstrates search symmetry and frontier balancing

## A Gentle Proof (Why It Works)

If the shortest path length is  $d$ , BFS explores  $O(b^d)$  nodes, but Bidirectional BFS explores  $2 \times O(b^{(d/2)})$  nodes — a huge savings since  $b^{(d/2)} < b^d$ .

Each side guarantees the frontier grows level by level, and intersection ensures meeting at the middle of the shortest path.

### Try It Yourself

1. Trace bidirectional BFS on a 5-node chain ( $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ ).
2. Count nodes visited by single BFS vs bidirectional BFS.
3. Add print statements to see where the waves meet.
4. Modify to reconstruct the path.
5. Compare performance on branching graphs.

### Test Cases

| Graph                                                       | Source | Target | Found? | Meeting Node |
|-------------------------------------------------------------|--------|--------|--------|--------------|
| $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ | 0      | 4      |        | 2            |
| $0 \rightarrow 1, 1 \rightarrow 2, 2 \rightarrow 3$         | 0      | 3      |        | 1 or 2       |
| $0 \rightarrow 1, 2 \rightarrow 3$ (disconnected)           | 0      | 3      |        | —            |
| Triangle $0 \rightarrow 1 \rightarrow 2 \rightarrow 0$      | 0      | 2      |        | 0 or 2       |
| Star $0 \rightarrow \{1,2,3,4\}$                            | 1      | 2      |        | 0            |

### Complexity

- Time:  $O(b^{\hat{}}(d/2))$
- Space:  $O(b^{\hat{}}(d/2))$
- Optimality: Finds shortest path in unweighted graphs

Bidirectional BFS is the bridge builder, starting from both shores, racing toward the meeting point in the middle.

## 306 DFS on Grid

DFS on a grid is your go-to for exploring 2D maps, mazes, or islands. It works just like DFS on graphs, but here, each cell is a node and its up/down/left/right neighbors form the edges. Perfect for connected component detection, region labeling, or maze solving.

## What Problem Are We Solving?

We want to explore or mark all connected cells in a grid, often used for:

- Counting islands in a binary matrix
- Flood-fill algorithms (coloring regions)
- Maze traversal (finding a path through walls)
- Connectivity detection in 2D maps

Example: Think of a painter pouring ink into one cell, DFS shows how the ink spreads to fill the entire connected region.

## How Does It Work (Plain Language)?

DFS starts from a given cell, visits it, and recursively explores all valid, unvisited neighbors.

We check 4 directions (or 8 if diagonals count). Each neighbor is:

- Within bounds
- Not yet visited
- Satisfies the condition (e.g., same color, value = 1)

| Step | Current Cell               | Action     | Stack (Call Path)   |
|------|----------------------------|------------|---------------------|
| 1    | (0,0)                      | Visit      | [(0,0)]             |
| 2    | (0,1)                      | Move right | [(0,0),(0,1)]       |
| 3    | (1,1)                      | Move down  | [(0,0),(0,1),(1,1)] |
| 4    | (1,1) has no new neighbors | Backtrack  | [(0,0),(0,1)]       |
| 5    | Continue                   | ...        |                     |

The traversal ends when all reachable cells are visited.

## Tiny Code (Easy Versions)

C (DFS for Island Counting)

```
#include <stdio.h>
#include <stdbool.h>

#define MAX 100
int grid[MAX][MAX];
bool visited[MAX][MAX];
```

```

int n, m;

int dx[4] = {-1, 1, 0, 0};
int dy[4] = {0, 0, -1, 1};

void dfs(int x, int y) {
    visited[x][y] = true;
    for (int k = 0; k < 4; k++) {
        int nx = x + dx[k];
        int ny = y + dy[k];
        if (nx >= 0 && nx < n && ny >= 0 && ny < m &&
            grid[nx][ny] == 1 && !visited[nx][ny]) {
            dfs(nx, ny);
        }
    }
}

int main(void) {
    printf("Enter grid size (n m): ");
    scanf("%d %d", &n, &m);
    printf("Enter grid (0/1):\n");
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            scanf("%d", &grid[i][j]);

    int count = 0;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            if (grid[i][j] == 1 && !visited[i][j]) {
                dfs(i, j);
                count++;
            }
    printf("Number of islands: %d\n", count);
}

```

Python (Flood Fill)

```

grid = [
    [1,1,0,0],
    [1,0,0,1],
    [0,0,1,1],
]

```

```

[0,0,0,1]
]

n, m = len(grid), len(grid[0])
visited = [[False]*m for _ in range(n)]

def dfs(x, y):
    if x < 0 or x >= n or y < 0 or y >= m:
        return
    if grid[x][y] == 0 or visited[x][y]:
        return
    visited[x][y] = True
    for dx, dy in [(-1,0),(1,0),(0,-1),(0,1)]:
        dfs(x + dx, y + dy)

count = 0
for i in range(n):
    for j in range(m):
        if grid[i][j] == 1 and not visited[i][j]:
            dfs(i, j)
            count += 1

print("Number of islands:", count)

```

## Why It Matters

- Core tool for grid exploration and region labeling
- Forms the heart of island problems, maze solvers, and map connectivity
- Demonstrates DFS behavior in real-world layouts
- Easy visualization and debugging on 2D arrays

## A Gentle Proof (Why It Works)

Each cell is visited exactly once, marked `visited` upon entry. Recursive calls spread to all valid neighbors. So total time = proportional to number of cells and edges (neighbors).

If grid has size  $n \times m$ , and each cell checks 4 neighbors:

- Time:  $O(n \times m)$
- Space:  $O(n \times m)$  visited + recursion depth ( $n \times m$ )

DFS guarantees every reachable cell is visited exactly once, forming connected components.

## Try It Yourself

1. Change movement to 8 directions (include diagonals).
2. Modify to flood-fill a color (e.g., replace all 1s with 2s).
3. Count components in a matrix of characters ('X', 'O').
4. Visualize traversal order in a printed grid.
5. Compare with BFS on the same grid.

## Test Cases

| Grid                                   | Expected | Description            |
|----------------------------------------|----------|------------------------|
| <code>[[1,0,0],[0,1,0],[0,0,1]]</code> | 3        | Diagonal not connected |
| <code>[[1,1,0],[1,0,0],[0,0,1]]</code> | 2        | Two clusters           |
| <code>[[1,1,1],[1,1,1],[1,1,1]]</code> | 1        | One big island         |
| <code>[[0,0,0],[0,0,0]]</code>         | 0        | No land                |
| <code>[[1,0,1],[0,1,0],[1,0,1]]</code> | 5        | Many singles           |

## Complexity

- Time:  $O(n \times m)$
- Space:  $O(n \times m)$  (visited) or  $O(\text{depth})$  recursion stack

DFS on grid is your map explorer, sweeping through every reachable patch, one cell at a time.

## 307 BFS on Grid

BFS on a grid explores cells level by level, making it perfect for shortest paths in unweighted grids, minimum steps in mazes, and distance labeling from a source. Each cell is a node and edges connect to neighbors such as up, down, left, right.

### What Problem Are We Solving?

We want to:

- Find the shortest path from a start cell to a goal cell when each move costs the same
- Compute a distance map from a source to all reachable cells
- Handle obstacles cleanly and avoid revisiting

Example: Given a maze as a 0 or 1 grid, where 0 is free and 1 is wall, BFS finds the fewest moves from start to target.

## How Does It Work (Plain Language)?

Use a queue. Start from the source, push it with distance 0, and expand in waves. At each step, pop the front cell, try its neighbors, mark unseen neighbors visited, and record their distance as current distance + 1.

| Step | Queue (front to back) | Current Cell | Action          | Distance Updated           |
|------|-----------------------|--------------|-----------------|----------------------------|
| 1    | [(sx, sy)]            | (sx, sy)     | Start           | dist[sx][sy] = 0           |
| 2    | [(n1), (n2)]          | (n1)         | Visit neighbors | dist[n1] = 1               |
| 3    | [(n2), (n3), (n4)]    | (n2)         | Continue        | dist[n2] = 1               |
| 4    | [...]                 | ...          | Wave expands    | dist[next] = dist[cur] + 1 |

The first time you reach the goal, the recorded distance is minimal.

## Tiny Code (Easy Versions)

C (Shortest Path on a 0 or 1 Grid)

```
#include <stdio.h>
#include <stdbool.h>

#define MAX 200
int n, m;
int grid[MAX][MAX]; // 0 free, 1 wall
int distv[MAX][MAX]; // distance map
bool vis[MAX][MAX];

int qx[MAX*MAX], qy[MAX*MAX];
int front = 0, rear = 0;

int dx[4] = {-1, 1, 0, 0};
int dy[4] = {0, 0, -1, 1};

void enqueue(int x, int y) { qx[rear] = x; qy[rear] = y; rear++; }
void dequeue(int *x, int *y) { *x = qx[front]; *y = qy[front]; front++; }
bool empty() { return front == rear; }

int bfs(int sx, int sy, int tx, int ty) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++) {
            vis[i][j] = false;
```

```

        distv[i][j] = -1;
    }

vis[sx][sy] = true;
distv[sx][sy] = 0;
enqueue(sx, sy);

while (!empty()) {
    int x, y;
    dequeue(&x, &y);
    if (x == tx && y == ty) return distv[x][y];

    for (int k = 0; k < 4; k++) {
        int nx = x + dx[k], ny = y + dy[k];
        if (nx >= 0 && nx < n && ny >= 0 && ny < m &&
            !vis[nx][ny] && grid[nx][ny] == 0) {
            vis[nx][ny] = true;
            distv[nx][ny] = distv[x][y] + 1;
            enqueue(nx, ny);
        }
    }
}
return -1; // unreachable
}

int main(void) {
    printf("Enter n m: ");
    scanf("%d %d", &n, &m);
    printf("Enter grid (0 free, 1 wall):\n");
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            scanf("%d", &grid[i][j]);

    int sx, sy, tx, ty;
    printf("Enter start sx sy and target tx ty: ");
    scanf("%d %d %d %d", &sx, &sy, &tx, &ty);

    int d = bfs(sx, sy, tx, ty);
    if (d >= 0) printf("Shortest distance: %d\n", d);
    else printf("No path\n");
}

```

## Python (Distance Map and Path Reconstruction)

```
from collections import deque

grid = [
    [0,0,0,1],
    [1,0,0,0],
    [0,0,1,0],
    [0,0,0,0]
]
n, m = len(grid), len(grid[0])

def bfs_grid(sx, sy, tx, ty):
    dist = [[-1]*m for _ in range(n)]
    parent = [[None]*m for _ in range(n)]
    q = deque()
    q.append((sx, sy))
    dist[sx][sy] = 0

    for dx, dy in [(-1,0), (1,0), (0,-1), (0,1)]:
        pass # only to show directions exist

    while q:
        x, y = q.popleft()
        if (x, y) == (tx, ty):
            break
        for dx, dy in [(-1,0), (1,0), (0,-1), (0,1)]:
            nx, ny = x + dx, y + dy
            if 0 <= nx < n and 0 <= ny < m and grid[nx][ny] == 0 and dist[nx][ny] == -1:
                dist[nx][ny] = dist[x][y] + 1
                parent[nx][ny] = (x, y)
                q.append((nx, ny))

    # Reconstruct path if reachable
    if dist[tx][ty] == -1:
        return dist, []
    path = []
    cur = (tx, ty)
    while cur:
        path.append(cur)
        cur = parent[cur[0]][cur[1]]
    path.reverse()
    return dist, path
```

```

dist, path = bfs_grid(0, 0, 3, 3)
print("Distance to target:", dist[3][3])
print("Path:", path)

```

## Why It Matters

- Guarantees shortest path in unweighted grids
- Produces a full distance transform useful for many tasks
- Robust and simple for maze solvers and robotics navigation
- Natural stepping stone to 0-1 BFS and Dijkstra

## A Gentle Proof (Why It Works)

BFS processes cells by nondecreasing distance from the source. When a cell is first dequeued, the stored distance equals the minimum number of moves needed to reach it. Each free neighbor is discovered with distance plus one. Therefore the first time the goal is reached, that distance is minimal.

- Each cell enters the queue at most once
- Each edge between neighboring cells is considered once

Hence total work is linear in the number of cells and neighbor checks.

## Try It Yourself

1. Add 8 directional moves and compare paths with 4 directional moves.
2. Add teleporters by connecting listed cell pairs as edges.
3. Convert to multi source BFS by enqueueing several starts with distance 0.
4. Block some cells and verify that BFS never steps through walls.
5. Record parents and print the maze with the path marked.

## Test Cases

| Grid                   | Start               | Target | Expected                       |
|------------------------|---------------------|--------|--------------------------------|
| 2 x 2 all free         | (0,0)               | (1,1)  | Distance 2 via right then down |
| 3 x 3 with center wall | (0,0)               | (2,2)  | Distance 4 around the wall     |
| Line 1 x 5 all free    | (0,0)               | (0,4)  | Distance 4                     |
| Blocked target         | (0,0)               | (1,1)  | No path                        |
| Multi source wave      | {all corner starts} | center | Minimum among corners          |

| Grid | Start | Target | Expected |
|------|-------|--------|----------|
|------|-------|--------|----------|

## Complexity

- Time:  $O(n \times m)$
- Space:  $O(n \times m)$  for visited or distance map and queue

BFS on grid is the wavefront that sweeps a map evenly, giving you the fewest steps from start to goal with clean, level by level logic.

## 308 Multi-Source BFS

Multi-Source BFS is the wavefront BFS that starts not from one node but from many sources at once. It's perfect when several starting points all spread out simultaneously, like multiple fires burning through a forest, or signals radiating from several transmitters.

### What Problem Are We Solving?

We need to find minimum distances from multiple starting nodes, not just one. This is useful when:

- There are several sources of influence (e.g. infections, signals, fires)
- You want the nearest source for each node
- You need simultaneous propagation (e.g. multi-start shortest path)

Examples:

- Spread of rumors from multiple people
- Flooding time from multiple water sources
- Minimum distance to nearest hospital or supply center

### How Does It Work (Plain Language)?

We treat all sources as level 0 and push them into the queue at once. Then BFS proceeds normally, each node is assigned a distance equal to the shortest path from any source.

| Step | Queue (Front → Back)      | Action                 | Distance Updated       |
|------|---------------------------|------------------------|------------------------|
| 1    | [S1, S2, S3]              | Initialize all sources | $\text{dist}[S^*] = 0$ |
| 2    | [Neighbors of S1, S2, S3] | Wave expands           | $\text{dist} = 1$      |

| Step | Queue (Front → Back) | Action   | Distance Updated |
|------|----------------------|----------|------------------|
| 3    | [Next Layer]         | Continue | dist = 2         |
| ...  | ...                  | ...      | ...              |

The first time a node is visited, we know it's from the nearest source.

### Tiny Code (Easy Versions)

C (Multi-Source BFS on Grid)

```
#include <stdio.h>
#include <stdbool.h>

#define MAX 100
int n, m;
int grid[MAX][MAX];
int distv[MAX][MAX];
bool vis[MAX][MAX];
int qx[MAX*MAX], qy[MAX*MAX];
int front = 0, rear = 0;

int dx[4] = {-1, 1, 0, 0};
int dy[4] = {0, 0, -1, 1};

void enqueue(int x, int y) { qx[rear] = x; qy[rear] = y; rear++; }
void dequeue(int *x, int *y) { *x = qx[front]; *y = qy[front]; front++; }
bool empty() { return front == rear; }

void multi_source_bfs() {
    while (!empty()) {
        int x, y;
        dequeue(&x, &y);
        for (int k = 0; k < 4; k++) {
            int nx = x + dx[k], ny = y + dy[k];
            if (nx >= 0 && nx < n && ny >= 0 && ny < m &&
                grid[nx][ny] == 0 && !vis[nx][ny]) {
                vis[nx][ny] = true;
                distv[nx][ny] = distv[x][y] + 1;
                enqueue(nx, ny);
            }
        }
    }
}
```

```

        }
    }

int main(void) {
    printf("Enter grid size n m: ");
    scanf("%d %d", &n, &m);
    printf("Enter grid (0 free, 1 blocked, 2 source):\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            scanf("%d", &grid[i][j]);
            if (grid[i][j] == 2) {
                vis[i][j] = true;
                distv[i][j] = 0;
                enqueue(i, j);
            }
        }
    }

    multi_source_bfs();

    printf("Distance map:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++)
            printf("%2d ", distv[i][j]);
        printf("\n");
    }
}

```

Python (Simple Multi-Source BFS)

```

from collections import deque

grid = [
    [2,0,1,0],
    [0,1,0,0],
    [0,0,0,2]
]
n, m = len(grid), len(grid[0])
dist = [[-1]*m for _ in range(n)]
q = deque()

for i in range(n):

```

```

for j in range(m):
    if grid[i][j] == 2: # source
        dist[i][j] = 0
        q.append((i, j))

dirs = [(-1,0),(1,0),(0,-1),(0,1)]
while q:
    x, y = q.popleft()
    for dx, dy in dirs:
        nx, ny = x+dx, y+dy
        if 0 <= nx < n and 0 <= ny < m and grid[nx][ny] == 0 and dist[nx][ny] == -1:
            dist[nx][ny] = dist[x][y] + 1
            q.append((nx, ny))

print("Distance Map:")
for row in dist:
    print(row)

```

## Why It Matters

- Finds nearest source distance for all nodes in one pass
- Ideal for multi-origin diffusion problems
- Foundation for tasks like multi-fire spread, influence zones, Voronoi partitioning on graphs

## A Gentle Proof (Why It Works)

Since all sources start at distance 0 and BFS expands in order of increasing distance, the first time a node is visited, it's reached by the shortest possible path from any source.

Each cell is enqueued exactly once →  $O(V + E)$  time. No need to run BFS separately for each source.

## Try It Yourself

1. Mark multiple sources (2s) on a grid, verify distances radiate outward.
2. Change obstacles (1s) and see how waves avoid them.
3. Count how many steps each free cell is from nearest source.
4. Modify to return which source id reached each cell first.
5. Compare total cost vs running single-source BFS repeatedly.

## Test Cases

| Grid                              | Expected Output (Distances) | Description          |
|-----------------------------------|-----------------------------|----------------------|
| [[2,0,2]]                         | [0,1,0]                     | Two sources on edges |
| [[2,0,0],[0,1,0],[0,0,2]]         | wave radiates from corners  | Mixed obstacles      |
| [[2,2,2]]                         | [0,0,0]                     | All sources          |
| [[0,0,0],[0,0,0]] + center source | center = 0, corners = 2     | Wave expanding       |
| All blocked                       | unchanged                   | No propagation       |

## Complexity

- Time:  $O(V + E)$
- Space:  $O(V)$  for queue and distance map

Multi-Source BFS is the chorus of wavefronts, expanding together, each note reaching its closest audience in perfect harmony.

## 309 Topological Sort (DFS-based)

Topological sort is the linear ordering of vertices in a Directed Acyclic Graph (DAG) such that for every directed edge ( $u \rightarrow v$ ), vertex ( $u$ ) appears before ( $v$ ) in the order. The DFS-based approach discovers this order by exploring deeply and recording finishing times.

### What Problem Are We Solving?

We want a way to order tasks that have dependencies. Topological sort answers: *In what order can we perform tasks so that prerequisites come first?*

Typical use cases:

- Build systems (compile order)
- Course prerequisite scheduling
- Pipeline stage ordering
- Dependency resolution (e.g. package installs)

Example: If task A must finish before B and C, and C before D, then one valid order is A → C → D → B.

## How Does It Work (Plain Language)?

DFS explores from each unvisited node. When a node finishes (no more outgoing edges to explore), push it onto a stack. After all DFS calls, reverse the stack, that's your topological order.

1. Initialize all nodes as unvisited
2. For each node  $v$ :
  - Run DFS if not visited
  - After exploring all neighbors, push  $v$  to stack
3. Reverse stack to get topological order

| Step | Current Node | Action          | Stack     |
|------|--------------|-----------------|-----------|
| 1    | A            | Visit neighbors | []        |
| 2    | B            | Visit           | []        |
| 3    | D            | Visit           | []        |
| 4    | D done       | Push D          | [D]       |
| 5    | B done       | Push B          | [D, B]    |
| 6    | A done       | Push A          | [D, B, A] |

Reverse: [A, B, D]

## Tiny Code (Easy Versions)

C (DFS-based Topological Sort)

```
#include <stdio.h>
#include <stdbool.h>

#define MAX 100
int graph[MAX][MAX];
bool visited[MAX];
int stack[MAX];
int top = -1;
int n;

void dfs(int v) {
    visited[v] = true;
    for (int u = 0; u < n; u++) {
        if (!visited[u]) {
            dfs(u);
        }
    }
    stack[++top] = v;
}
```

```

        if (graph[v][u] && !visited[u]) {
            dfs(u);
        }
    }
    stack[++top] = v; // push after exploring neighbors
}

void topological_sort() {
    for (int i = 0; i < n; i++) {
        if (!visited[i]) dfs(i);
    }
    printf("Topological order: ");
    while (top >= 0) printf("%d ", stack[top--]);
    printf("\n");
}

int main(void) {
    printf("Enter number of vertices: ");
    scanf("%d", &n);
    printf("Enter adjacency matrix (DAG):\n");
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            scanf("%d", &graph[i][j]);
    topological_sort();
}

```

Python (DFS-based)

```

graph = {
    0: [1, 2],
    1: [3],
    2: [3],
    3: []
}

visited = set()
stack = []

def dfs(v):
    visited.add(v)
    for u in graph[v]:
        if u not in visited:

```

```

        dfs(u)
        stack.append(v)

for v in graph:
    if v not in visited:
        dfs(v)

stack.reverse()
print("Topological order:", stack)

```

## Why It Matters

- Ensures dependency order in DAGs
- Fundamental in compilers, schedulers, and build systems
- Basis for advanced algorithms:
  - Kahn’s Algorithm (queue-based)
  - DAG shortest paths / DP
  - Critical path analysis

## A Gentle Proof (Why It Works)

Each vertex is pushed onto the stack after all its descendants are explored. So if there’s an edge ( $u \rightarrow v$ ), DFS ensures ( $v$ ) finishes first and is pushed earlier, meaning ( $u$ ) will appear later in the stack. Reversing the stack thus guarantees ( $u$ ) precedes ( $v$ ).

No cycles allowed, if a back edge is found, topological sort is impossible.

## Try It Yourself

1. Draw a DAG and label edges as prerequisites.
2. Run DFS and record finish times.
3. Push nodes on completion, reverse the order.
4. Add a cycle and see why it breaks.
5. Compare with Kahn’s Algorithm results.

## Test Cases

| Graph              | Edges      | Topological Order (Possible) |
|--------------------|------------|------------------------------|
| A→B→C              | A→B, B→C   | A B C                        |
| A→C, B→C           | A→C, B→C   | A B C or B A C               |
| 0→1, 0→2, 1→3, 2→3 | DAG        | 0 2 1 3 or 0 1 2 3           |
| Chain 0→1→2→3      | Linear DAG | 0 1 2 3                      |
| Cycle 0→1→2→0      | Not DAG    | No valid order               |

## Complexity

- Time:  $O(V + E)$  (each node and edge visited once)
- Space:  $O(V)$  (stack + visited array)

Topological sort (DFS-based) is your dependency detective, exploring deeply, marking completion, and leaving behind a perfect trail of prerequisites.

## 310 Topological Sort (Kahn's Algorithm)

Kahn's Algorithm is the queue-based way to perform topological sorting. Instead of relying on recursion, it tracks in-degrees (how many edges point into each node) and repeatedly removes nodes with zero in-degree. It's clean, iterative, and naturally detects cycles.

### What Problem Are We Solving?

We want a linear ordering of tasks in a Directed Acyclic Graph (DAG) such that each task appears after all its prerequisites.

Kahn's method is especially handy when:

- You want an iterative (non-recursive) algorithm
- You need to detect cycles automatically
- You're building a scheduler or compiler dependency resolver

Example: If A must happen before B and C, and C before D, then valid orders: A C D B or A B C D. Kahn's algorithm builds this order by peeling off "ready" nodes (those with no remaining prerequisites).

## How Does It Work (Plain Language)?

Each node starts with an in-degree (count of incoming edges). Nodes with in-degree = 0 are ready to process.

1. Compute in-degree for each node
2. Enqueue all nodes with in-degree = 0
3. While queue not empty:
  - Pop node v and add it to the topological order
  - For each neighbor u of v, decrement `in-degree[u]`
  - If `in-degree[u]` becomes 0, enqueue u
4. If all nodes processed → valid topological order Otherwise → cycle detected

| Step | Queue                               | Popped | Updated In-Degree | Order        |
|------|-------------------------------------|--------|-------------------|--------------|
| 1    | [A]                                 | A      | B:0, C:1, D:2     | [A]          |
| 2    | [B]                                 | B      | C:0, D:2          | [A, B]       |
| 3    | <span style="color: blue;">C</span> | C      | D:1               | [A, B, C]    |
| 4    | [D]                                 | D      | —                 | [A, B, C, D] |

## Tiny Code (Easy Versions)

C (Kahn's Algorithm)

```
#include <stdio.h>
#include <stdbool.h>

#define MAX 100
int graph[MAX][MAX];
int indeg[MAX];
int queue[MAX];
int front = 0, rear = 0;
int n;

void enqueue(int v) { queue[rear++] = v; }
int dequeue() { return queue[front++]; }
bool empty() { return front == rear; }

void kahn_topo() {
```

```

for (int v = 0; v < n; v++) {
    if (indeg[v] == 0) enqueue(v);
}

int count = 0;
int order[MAX];

while (!empty()) {
    int v = dequeue();
    order[count++] = v;
    for (int u = 0; u < n; u++) {
        if (graph[v][u]) {
            indeg[u]--;
            if (indeg[u] == 0) enqueue(u);
        }
    }
}

if (count != n) {
    printf("Graph has a cycle, topological sort not possible\n");
} else {
    printf("Topological order: ");
    for (int i = 0; i < count; i++) printf("%d ", order[i]);
    printf("\n");
}
}

int main(void) {
printf("Enter number of vertices: ");
scanf("%d", &n);

printf("Enter adjacency matrix:\n");
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++) {
        scanf("%d", &graph[i][j]);
        if (graph[i][j]) indeg[j]++;
    }

kahn_topo();
}

```

Python (Using Queue)

```

from collections import deque

graph = {
    0: [1, 2],
    1: [3],
    2: [3],
    3: []
}

indeg = {v: 0 for v in graph}
for v in graph:
    for u in graph[v]:
        indeg[u] += 1

q = deque([v for v in graph if indeg[v] == 0])
order = []

while q:
    v = q.popleft()
    order.append(v)
    for u in graph[v]:
        indeg[u] -= 1
        if indeg[u] == 0:
            q.append(u)

if len(order) == len(graph):
    print("Topological order:", order)
else:
    print("Cycle detected, no valid order")

```

## Why It Matters

- Fully iterative (no recursion stack)
- Naturally detects cycles
- Efficient for build systems, task planners, and dependency graphs
- Forms the base for Kahn's scheduling algorithm in DAG processing

## A Gentle Proof (Why It Works)

Nodes with in-degree 0 have no prerequisites, they can appear first. Once processed, they are removed (decrementing in-degree of successors). This ensures:

- Each node is processed only after all its dependencies
- If a cycle exists, some nodes never reach in-degree 0 → detection built-in

Since each edge is considered once, runtime =  $O(V + E)$ .

### Try It Yourself

1. Build a small DAG manually and run the algorithm step by step.
2. Introduce a cycle ( $A \rightarrow B \rightarrow A$ ) and observe detection.
3. Compare with DFS-based order, both valid.
4. Add priority to queue (min vertex first) to get lexicographically smallest order.
5. Apply to course prerequisite planner.

### Test Cases

| Graph                                                                | Edges                                   | Output             | Notes            |
|----------------------------------------------------------------------|-----------------------------------------|--------------------|------------------|
| $A \rightarrow B \rightarrow C$                                      | $A \rightarrow B,$<br>$B \rightarrow C$ | A B C              | Linear chain     |
| $A \rightarrow C, B \rightarrow C$                                   | $A \rightarrow C,$<br>$B \rightarrow C$ | A B C or B A C     | Multiple sources |
| $0 \rightarrow 1, 0 \rightarrow 2, 1 \rightarrow 3, 2 \rightarrow 3$ | DAG                                     | 0 1 2 3 or 0 2 1 3 | Multiple valid   |
| $0 \rightarrow 1, 1 \rightarrow 2, 2 \rightarrow 0$                  | Cycle                                   | No order           | Cycle detected   |

### Complexity

- Time:  $O(V + E)$
- Space:  $O(V)$  (queue, indegrees)

Kahn's Algorithm is the dependency peeler, stripping away nodes layer by layer until a clean, linear order emerges.

## Section 32. Strongly Connected Components

### 311 Kosaraju's Algorithm

Kosaraju's algorithm is one of the clearest ways to find strongly connected components (SCCs) in a directed graph. It uses two depth-first searches, one on the original graph, and one on its reversed version, to peel off SCCs layer by layer.

## What Problem Are We Solving?

In a directed graph, a *strongly connected component* is a maximal set of vertices such that each vertex is reachable from every other vertex in the same set.

Kosaraju's algorithm groups the graph into these SCCs.

This is useful for:

- Condensing a graph into a DAG (meta-graph)
- Dependency analysis in compilers
- Finding cycles or redundant modules
- Graph simplification before DP or optimization

Example: Imagine a one-way road network, SCCs are groups of cities where you can travel between any pair.

## How Does It Work (Plain Language)?

Kosaraju's algorithm runs in two DFS passes:

1. First DFS (Original Graph): Explore all vertices. Each time a node finishes (recursion ends), record it on a stack (by finish time).
2. Reverse the Graph: Reverse all edges (flip direction).
3. Second DFS (Reversed Graph): Pop nodes from the stack (highest finish time first). Each DFS from an unvisited node forms one strongly connected component.

| Phase | Graph    | Action                     | Result            |
|-------|----------|----------------------------|-------------------|
| 1     | Original | DFS all, push finish order | Stack of vertices |
| 2     | Reversed | DFS by pop order           | Identify SCCs     |
| 3     | Output   | Each DFS tree              | SCC list          |

## Tiny Code (Easy Versions)

C (Adjacency List)

```

#include <stdio.h>
#include <stdbool.h>

#define MAX 100
int n;
int graph[MAX][MAX];
int rev[MAX][MAX];
bool visited[MAX];
int stack[MAX];
int top = -1;

void dfs1(int v) {
    visited[v] = true;
    for (int u = 0; u < n; u++) {
        if (graph[v][u] && !visited[u]) dfs1(u);
    }
    stack[++top] = v; // push after finishing
}

void dfs2(int v) {
    printf("%d ", v);
    visited[v] = true;
    for (int u = 0; u < n; u++) {
        if (rev[v][u] && !visited[u]) dfs2(u);
    }
}

int main(void) {
    printf("Enter number of vertices: ");
    scanf("%d", &n);

    printf("Enter adjacency matrix (directed):\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &graph[i][j]);
            rev[j][i] = graph[i][j]; // build reverse
        }
    }

    // Step 1: first DFS
    for (int i = 0; i < n; i++)
        visited[i] = false;
}

```

```

for (int i = 0; i < n; i++)
    if (!visited[i]) dfs1(i);

// Step 2: second DFS on reversed
for (int i = 0; i < n; i++)
    visited[i] = false;

printf("Strongly Connected Components:\n");
while (top >= 0) {
    int v = stack[top--];
    if (!visited[v]) {
        dfs2(v);
        printf("\n");
    }
}
}
}

```

Python (Using Lists)

```

from collections import defaultdict

graph = defaultdict(list)
rev = defaultdict(list)

edges = [(0,1),(1,2),(2,0),(1,3)]
for u, v in edges:
    graph[u].append(v)
    rev[v].append(u)

visited = set()
stack = []

def dfs1(v):
    visited.add(v)
    for u in graph[v]:
        if u not in visited:
            dfs1(u)
    stack.append(v)

for v in graph:
    if v not in visited:
        dfs1(v)

```

```

visited.clear()

def dfs2(v, comp):
    visited.add(v)
    comp.append(v)
    for u in rev[v]:
        if u not in visited:
            dfs2(u, comp)

print("Strongly Connected Components:")
while stack:
    v = stack.pop()
    if v not in visited:
        comp = []
        dfs2(v, comp)
        print(comp)

```

## Why It Matters

- Splits a directed graph into mutually reachable groups
- Used in condensation (convert cyclic graph → DAG)
- Helps detect cyclic dependencies
- Foundation for component-level optimization

## A Gentle Proof (Why It Works)

1. Finishing times from the first DFS ensure we process “sinks” first (post-order).
2. Reversing edges turns sinks into sources.
3. The second DFS finds all nodes reachable from that source, i.e. an SCC.
4. Every node is assigned to exactly one SCC.

Correctness follows from properties of finishing times and reachability symmetry.

## Try It Yourself

1. Draw a directed graph with cycles and run the steps manually.
2. Track finish order stack.
3. Reverse all edges and start popping nodes.
4. Each DFS tree = one SCC.
5. Compare results with Tarjan’s algorithm.

## Test Cases

| Graph                   | Edges        | SCCs          |
|-------------------------|--------------|---------------|
| 0→1, 1→2, 2→0           | Cycle        | {0,1,2}       |
| 0→1, 1→2                | Chain        | {0}, {1}, {2} |
| 0→1, 1→2, 2→0, 2→3      | Cycle + tail | {0,1,2}, {3}  |
| 0→1, 1→0, 1→2, 2→3, 3→2 | Two SCCs     | {0,1}, {2,3}  |

## Complexity

- Time:  $O(V + E)$  (2 DFS passes)
- Space:  $O(V + E)$  (graph + stack)

Kosaraju's Algorithm is your mirror explorer, traverse once to record the story, flip the graph, then replay it backward to reveal every strongly bound group.

## 312 Tarjan's Algorithm

Tarjan's algorithm finds all strongly connected components (SCCs) in a directed graph in one DFS pass, without reversing the graph. It's an elegant and efficient method that tracks each node's discovery time and lowest reachable ancestor to identify SCC roots.

### What Problem Are We Solving?

We need to group vertices of a directed graph into SCCs, where each node is reachable from every other node in the same group. Unlike Kosaraju's two-pass method, Tarjan's algorithm finds all SCCs in a single DFS, making it faster in practice and easy to integrate into larger graph algorithms.

Common applications:

- Cycle detection in directed graphs
- Component condensation for DAG processing
- Deadlock analysis
- Strong connectivity queries in compilers, networks, and systems

Example: Imagine a group of cities connected by one-way roads. SCCs are clusters of cities that can all reach each other, forming a tightly connected region.

## How Does It Work (Plain Language)?

Each vertex gets:

- A discovery time (disc), when it's first visited
- A low-link value (low), the smallest discovery time reachable (including back edges)

A stack keeps track of the active recursion path (current DFS stack). When a vertex's disc equals its low, it's the root of an SCC, pop nodes from the stack until this vertex reappears.

| Step | Action                           | Stack          | SCC                       |
|------|----------------------------------|----------------|---------------------------|
|      |                                  |                | Found                     |
| 1    | Visit node, assign disc & low    | [A]            | -                         |
| 2    | Go deeper (DFS neighbors)        | [A, B, C]      | -                         |
| 3    | Reach node with no new neighbors | Update<br>low  | [A, B, C] -               |
| 4    | Backtrack, compare lows          | [A, B]         | SCC {C}                   |
| 5    | When disc == low                 | Pop SCC<br>[A] | SCC {B, C} (if connected) |

## Tiny Code (Easy Versions)

C (Tarjan's Algorithm)

```
#include <stdio.h>
#include <stdbool.h>

#define MAX 100
int n;
int graph[MAX][MAX];
int disc[MAX], low[MAX], stack[MAX];
bool inStack[MAX];
int time_counter = 0, top = -1;

void dfs_tarjan(int v) {
    disc[v] = low[v] = ++time_counter;
    stack[++top] = v;
    inStack[v] = true;

    for (int u = 0; u < n; u++) {
        if (!graph[v][u]) continue;
        if (disc[u] == 0) {
            dfs_tarjan(u);
            low[v] = min(low[v], low[u]);
        } else if (inStack[u]) {
            low[v] = min(low[v], disc[u]);
        }
    }

    if (disc[v] == low[v]) {
        int scc_size = 0;
        while (top != disc[v]) {
            int current = stack[top];
            inStack[current] = false;
            top--;
            scc_size++;
        }
        printf("SCC %d: ", scc_size);
        for (int i = 0; i < scc_size; i++) {
            printf("%d ", stack[i]);
        }
        printf("\n");
    }
}
```

```

        if (low[u] < low[v]) low[v] = low[u];
    } else if (inStack[u]) {
        if (disc[u] < low[v]) low[v] = disc[u];
    }
}

if (disc[v] == low[v]) {
    printf("SCC: ");
    int w;
    do {
        w = stack[top--];
        inStack[w] = false;
        printf("%d ", w);
    } while (w != v);
    printf("\n");
}
}

int main(void) {
    printf("Enter number of vertices: ");
    scanf("%d", &n);
    printf("Enter adjacency matrix (directed):\n");
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            scanf("%d", &graph[i][j]);

    for (int i = 0; i < n; i++) {
        disc[i] = 0;
        inStack[i] = false;
    }

    printf("Strongly Connected Components:\n");
    for (int i = 0; i < n; i++)
        if (disc[i] == 0)
            dfs_tarjan(i);
}
}

```

Python (Using Adjacency List)

```

from collections import defaultdict

graph = defaultdict(list)

```

```

edges = [(0,1),(1,2),(2,0),(1,3)]
for u,v in edges:
    graph[u].append(v)

time = 0
disc = {}
low = {}
stack = []
in_stack = set()
sccs = []

def dfs(v):
    global time
    time += 1
    disc[v] = low[v] = time
    stack.append(v)
    in_stack.add(v)

    for u in graph[v]:
        if u not in disc:
            dfs(u)
            low[v] = min(low[v], low[u])
        elif u in in_stack:
            low[v] = min(low[v], disc[u])

    if disc[v] == low[v]:
        scc = []
        while True:
            w = stack.pop()
            in_stack.remove(w)
            scc.append(w)
            if w == v:
                break
        sccs.append(scc)

for v in list(graph.keys()):
    if v not in disc:
        dfs(v)

print("Strongly Connected Components:", sccs)

```

## Why It Matters

- Runs in one DFS, efficient and elegant
- Detects SCCs on the fly (no reversing edges)
- Useful for online algorithms (process SCCs as discovered)
- Powers cycle detection, condensation graphs, component-based optimization

## A Gentle Proof (Why It Works)

- `disc[v]`: time when  $v$  is first visited
- `low[v]`: smallest discovery time reachable via descendants or back edges
- When  $\text{disc}[v] == \text{low}[v]$ ,  $v$  is the root of its SCC (no back edges go above it)
- Popping from the stack gives all nodes reachable within the component

Each edge examined once → linear time.

## Try It Yourself

1. Run Tarjan on a graph with multiple cycles.
2. Observe `disc` and `low` values.
3. Print stack content at each step to see grouping.
4. Compare SCC output with Kosaraju's result.
5. Try adding a cycle and check grouping changes.

## Test Cases

| Graph                                                                | Edges        | SCCs                  |
|----------------------------------------------------------------------|--------------|-----------------------|
| $0 \rightarrow 1, 1 \rightarrow 2, 2 \rightarrow 0$                  | Cycle        | $\{0,1,2\}$           |
| $0 \rightarrow 1, 1 \rightarrow 2$                                   | Chain        | $\{2\}, \{1\}, \{0\}$ |
| $0 \rightarrow 1, 1 \rightarrow 2, 2 \rightarrow 0, 2 \rightarrow 3$ | Cycle + tail | $\{0,1,2\}, \{3\}$    |
| $1 \rightarrow 2, 2 \rightarrow 3, 3 \rightarrow 1, 3 \rightarrow 4$ | Two groups   | $\{1,2,3\}, \{4\}$    |

## Complexity

- Time:  $O(V + E)$  (one DFS pass)
- Space:  $O(V)$  (stack, arrays)

Tarjan's Algorithm is your clockwork explorer, tagging each vertex by time, tracing the deepest paths, and snapping off every strongly connected cluster in one graceful pass.

## 313 Gabow's Algorithm

Gabow's algorithm is another elegant one-pass method for finding strongly connected components (SCCs). It's less well-known than Tarjan's, but equally efficient, using two stacks to track active vertices and roots. It's a perfect example of "stack discipline" in graph exploration.

### What Problem Are We Solving?

We want to find all strongly connected components in a directed graph, subsets where every node can reach every other.

Gabow's algorithm, like Tarjan's, works in a single DFS traversal, but instead of computing **low-link** values, it uses two stacks to manage component discovery and boundaries.

This approach is especially helpful in streaming, online, or iterative DFS environments, where explicit low-link computations can get messy.

Applications:

- Cycle decomposition
- Program dependency analysis
- Component condensation (DAG creation)
- Strong connectivity testing

Example: Think of traversing a web of roads. One stack tracks where you've been, another stack marks "checkpoints" where loops close, each loop is a component.

### How Does It Work (Plain Language)?

Gabow's algorithm keeps track of discovery order and component boundaries using two stacks:

- S (main stack): stores all currently active nodes
- P (boundary stack): stores potential roots of SCCs

Steps:

1. Perform DFS. Assign each node an increasing index (**preorder**)
2. Push node onto both stacks (S and P)
3. For each edge ( $v \rightarrow u$ ):
  - If  $u$  unvisited  $\rightarrow$  recurse
  - If  $u$  on stack S  $\rightarrow$  adjust boundary stack P
4. After exploring neighbors:

- If the top of P is current node v:
  - Pop P
  - Pop from S until v
  - Those popped form an SCC

| Step | Stack S   | Stack P   | Action        |
|------|-----------|-----------|---------------|
| 1    | [A]       | [A]       | Visit A       |
| 2    | [A, B]    | [A, B]    | Visit B       |
| 3    | [A, B, C] | [A, B, C] | Visit C       |
| 4    | [A, B, C] | [A, B]    | Back edge C→B |
| 5    | [A, B]    | [A]       | Pop SCC {C}   |
| 6    | [A]       | []        | Pop SCC {B}   |
| 7    | []        | []        | Pop SCC {A}   |

### Tiny Code (Easy Versions)

C (Gabow's Algorithm)

```
#include <stdio.h>
#include <stdbool.h>

#define MAX 100
int graph[MAX][MAX];
int n;
int index_counter = 0;
int preorder[MAX];
bool onStack[MAX];
int stackS[MAX], topS = -1;
int stackP[MAX], topP = -1;

void dfs_gabow(int v) {
    preorder[v] = ++index_counter;
    stackS[++topS] = v;
    stackP[++topP] = v;
    onStack[v] = true;

    for (int u = 0; u < n; u++) {
        if (!graph[v][u]) continue;
        if (preorder[u] == 0) {
            dfs_gabow(u);
        }
    }
}
```

```

        } else if (onStack[u]) {
            while (preorder[stackP[topP]] > preorder[u]) topP--;
        }
    }

    if (stackP[topP] == v) {
        topP--;
        printf("SCC: ");
        int w;
        do {
            w = stackS[topS--];
            onStack[w] = false;
            printf("%d ", w);
        } while (w != v);
        printf("\n");
    }
}

int main(void) {
    printf("Enter number of vertices: ");
    scanf("%d", &n);
    printf("Enter adjacency matrix (directed):\n");
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            scanf("%d", &graph[i][j]);

    for (int i = 0; i < n; i++) {
        preorder[i] = 0;
        onStack[i] = false;
    }

    printf("Strongly Connected Components:\n");
    for (int i = 0; i < n; i++)
        if (preorder[i] == 0)
            dfs_gabow(i);
}

```

Python (Readable Implementation)

```

from collections import defaultdict

graph = defaultdict(list)

```

```

edges = [(0,1),(1,2),(2,0),(1,3)]
for u,v in edges:
    graph[u].append(v)

index_counter = 0
preorder = {}
on_stack = set()
stackS, stackP = [], []
sccs = []

def dfs(v):
    global index_counter
    index_counter += 1
    preorder[v] = index_counter
    stackS.append(v)
    stackP.append(v)
    on_stack.add(v)

    for u in graph[v]:
        if u not in preorder:
            dfs(u)
        elif u in on_stack:
            while preorder[stackP[-1]] > preorder[u]:
                stackP.pop()

    if stackP and stackP[-1] == v:
        stackP.pop()
        comp = []
        while True:
            w = stackS.pop()
            on_stack.remove(w)
            comp.append(w)
            if w == v:
                break
        sccs.append(comp)

for v in graph:
    if v not in preorder:
        dfs(v)

print("Strongly Connected Components:", sccs)

```

## Why It Matters

- Single DFS pass, no reverse graph
- Purely stack-based, avoids recursion depth issues in some variants
- Efficient and practical for large graphs
- Simpler low-link logic than Tarjan's in some applications

## A Gentle Proof (Why It Works)

Each node gets a preorder index. The second stack P tracks the earliest root reachable. Whenever the top of P equals the current node, all nodes above it in S form one SCC.

Invariant:

- S contains active nodes
- P contains possible roots (ordered by discovery)
- When back edges discovered, P trimmed to smallest reachable ancestor

This ensures each SCC is identified exactly once when its root finishes.

## Try It Yourself

1. Draw a graph and trace preorder indices.
2. Observe how stack P shrinks on back edges.
3. Record SCCs as they're popped.
4. Compare output to Tarjan's algorithm.
5. Try it on DAGs, cycles, and mixed graphs.

## Test Cases

| Graph                                                                | Edges                   | SCCs         |
|----------------------------------------------------------------------|-------------------------|--------------|
| $0 \rightarrow 1, 1 \rightarrow 2, 2 \rightarrow 0$                  | $\{0,1,2\}$             | One big SCC  |
| $0 \rightarrow 1, 1 \rightarrow 2$                                   | $\{2\}, \{1\}, \{0\}$   | Chain        |
| $0 \rightarrow 1, 1 \rightarrow 0, 2 \rightarrow 3$                  | $\{0,1\}, \{2\}, \{3\}$ | Mixed        |
| $0 \rightarrow 1, 1 \rightarrow 2, 2 \rightarrow 3, 3 \rightarrow 1$ | $\{1,2,3\}, \{0\}$      | Nested cycle |

## Complexity

- Time:  $O(V + E)$
- Space:  $O(V)$  (two stacks + metadata)

Gabow's Algorithm is your two-stack sculptor, carving SCCs from the graph in one graceful sweep, balancing exploration and boundaries like a craftsman marking edges before the final cut.

## 314 SCC DAG Construction

Once we've found strongly connected components (SCCs), we can build the condensation graph, a Directed Acyclic Graph (DAG) where each node represents an SCC, and edges connect them if any vertex in one SCC points to a vertex in another.

This structure is crucial because it transforms a messy cyclic graph into a clean acyclic skeleton, perfect for topological sorting, dynamic programming, and dependency analysis.

### What Problem Are We Solving?

We want to take a directed graph and reduce it into a simpler form by collapsing each SCC into a single node. The resulting graph (called the *condensation graph*) is always a DAG.

Why do this?

- To simplify reasoning about complex systems
- To run DAG algorithms on cyclic graphs (by condensing cycles)
- To perform component-level optimization
- To study dependencies between strongly connected subsystems

Example: Think of a city map where SCCs are tightly connected neighborhoods. The DAG shows how traffic flows between neighborhoods, not within them.

### How Does It Work (Plain Language)?

1. Find SCCs (using Kosaraju, Tarjan, or Gabow).
2. Assign each node an SCC ID (e.g., `comp[v] = c_id`).
3. Create a new graph with one node per SCC.
4. For each edge  $(u, v)$  in the original graph:
  - If `comp[u] != comp[v]`, add an edge from `comp[u]` to `comp[v]`

5. Remove duplicates (or store edges in sets).

Now the new graph has no cycles, since cycles are already condensed inside SCCs.

| Original Graph     | SCCs          | DAG       |
|--------------------|---------------|-----------|
| 0→1, 1→2, 2→0, 2→3 | {0,1,2}, {3}  | C0 → C1   |
| 0→1, 1→2, 2→3, 3→1 | {1,2,3}, {0}  | C0 → C1   |
| 0→1, 1→2           | {0}, {1}, {2} | 0 → 1 → 2 |

### Tiny Code (Easy Versions)

C (Using Tarjan's Output)

```
#include <stdio.h>
#include <stdbool.h>

#define MAX 100
int n;
int graph[MAX][MAX];
int comp[MAX]; // component ID
int comp_count = 0;

// Suppose comp[] already filled by Tarjan or Kosaraju

void build_condensed_graph() {
    int new_graph[MAX][MAX] = {0};

    for (int u = 0; u < n; u++) {
        for (int v = 0; v < n; v++) {
            if (graph[u][v] && comp[u] != comp[v]) {
                new_graph[comp[u]][comp[v]] = 1;
            }
        }
    }

    printf("Condensation Graph (Adjacency Matrix):\n");
    for (int i = 0; i < comp_count; i++) {
        for (int j = 0; j < comp_count; j++) {
            printf("%d ", new_graph[i][j]);
        }
        printf("\n");
    }
}
```

```
    }
}
```

(Assumes  $\text{comp}[v]$  and  $\text{comp\_count}$  were computed before.)

Python (with Tarjan's or Kosaraju's Components)

```
from collections import defaultdict

# Suppose we already have SCCs
sccs = [[0,1,2], [3], [4,5]]
graph = {
    0: [1],
    1: [2,3],
    2: [0],
    3: [4],
    4: [5],
    5: []
}

# Step 1: assign component ID
comp_id = {}
for i, comp in enumerate(sccs):
    for v in comp:
        comp_id[v] = i

# Step 2: build DAG
dag = defaultdict(set)
for u in graph:
    for v in graph[u]:
        if comp_id[u] != comp_id[v]:
            dag[comp_id[u]].add(comp_id[v])

print("Condensation Graph (as DAG):")
for c in dag:
    print(c, "->", sorted(dag[c]))
```

## Why It Matters

- Turns cyclic graph  $\rightarrow$  acyclic graph
- Enables topological sorting, dynamic programming, path counting

- Clarifies inter-component dependencies
- Used in compiler analysis, SCC-based optimizations, graph condensation

## A Gentle Proof (Why It Works)

Inside each SCC, every vertex can reach every other. When edges cross SCC boundaries, they go in one direction only (since returning would merge the components). Thus, the condensation graph cannot contain cycles, proving it's a DAG.

Each edge in the DAG represents at least one edge between components in the original graph.

## Try It Yourself

1. Run Tarjan's algorithm to get `comp[v]` for each vertex.
2. Build DAG edges using component IDs.
3. Visualize original vs condensed graphs.
4. Topologically sort the DAG.
5. Use DP on DAG to compute longest path or reachability.

## Test Cases

| Original Graph                                                                        | SCCs                         | Condensed Edges                               |
|---------------------------------------------------------------------------------------|------------------------------|-----------------------------------------------|
| $0 \rightarrow 1, 1 \rightarrow 2, 2 \rightarrow 0, 2 \rightarrow 3$                  | $\{0,1,2\}, \{3\}$           | $0 \rightarrow 1$                             |
| $0 \rightarrow 1, 1 \rightarrow 0, 1 \rightarrow 2, 2 \rightarrow 3, 3 \rightarrow 2$ | $\{0,1\}, \{2,3\}$           | $0 \rightarrow 1$                             |
| $0 \rightarrow 1, 1 \rightarrow 2, 2 \rightarrow 3$                                   | $\{0\}, \{1\}, \{2\}, \{3\}$ | $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$ |
| $0 \rightarrow 1, 1 \rightarrow 2, 2 \rightarrow 0$                                   | $\{0,1,2\}$                  | None (single node DAG)                        |

## Complexity

- Time:  $O(V + E)$  (using precomputed SCCs)
- Space:  $O(V + E)$  (new DAG structure)

SCC DAG Construction is your mapmaker's step, compressing tangled roads into clean highways, where each city (SCC) is a hub, and the new map is finally acyclic, ready for analysis.

## 315 SCC Online Merge

SCC Online Merge is a dynamic approach to maintain strongly connected components when a graph is growing over time (new edges are added). Instead of recomputing SCCs from scratch after each update, we *incrementally merge* components as they become connected.

It's the foundation of dynamic graph algorithms where edges arrive one by one, useful in online systems, incremental compilers, and evolving dependency graphs.

### What Problem Are We Solving?

We want to maintain SCC structure as we add edges to a directed graph.

In static algorithms like Tarjan or Kosaraju, SCCs are computed once. But if new edges appear over time, recomputing everything is too slow.

SCC Online Merge gives us:

- Efficient incremental updates (no full recompute)
- Fast component merging
- Up-to-date condensation DAG

Typical use cases:

- Incremental program analysis (new dependencies)
- Dynamic network reachability
- Streaming graph processing
- Online algorithm design

### How Does It Work (Plain Language)?

We start with each node as its own SCC. When a new edge ( $u \rightarrow v$ ) is added:

1. Check if ( $u$ ) and ( $v$ ) are already in the same SCC, if yes, nothing changes.
2. If not, check whether  $v$ 's SCC can reach  $u$ 's SCC (cycle detection).
3. If reachable, merge both SCCs into one.
4. Otherwise, add a DAG edge from  $\text{SCC}(u) \rightarrow \text{SCC}(v)$ .

We maintain:

- Union-Find / DSU structure for SCC groups
- Reachability or DAG edges between SCCs
- Optional topological order for fast cycle checks

| Edge Added | Action       | New SCCs      | Notes    |
|------------|--------------|---------------|----------|
| 0→1        | Create edge  | {0}, {1}      | Separate |
| 1→2        | Create edge  | {0}, {1}, {2} | Separate |
| 2→0        | Cycle formed | Merge {0,1,2} | New SCC  |
| 3→1        | Add edge     | {3}, {0,1,2}  | No merge |

### Tiny Code (Conceptual Demo)

Python (Simplified DSU + DAG Check)

```

class DSU:
    def __init__(self, n):
        self.parent = list(range(n))
    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]
    def union(self, a, b):
        ra, rb = self.find(a), self.find(b)
        if ra != rb:
            self.parent[rb] = ra

class OnlineSCC:
    def __init__(self, n):
        self.n = n
        self.ds = DSU(n)
        self.graph = [set() for _ in range(n)]

    def add_edge(self, u, v):
        su, sv = self.ds.find(u), self.ds.find(v)
        if su == sv:
            return # already connected
        # check if v's component reaches u's component
        if self._reachable(sv, su):
            # merge components
            self.ds.union(su, sv)
            merged = self.ds.find(su)
            self.graph[merged] = self.graph[su] | self.graph[sv]
        else:
            self.graph[su].add(sv)

```

```

def _reachable(self, start, target, seen=None):
    if seen is None: seen = set()
    if start == target: return True
    seen.add(start)
    for nxt in self.graph[start]:
        if nxt not in seen and self._reachable(nxt, target, seen):
            return True
    return False

def components(self):
    groups = {}
    for v in range(self.n):
        root = self.dsu.find(v)
        groups.setdefault(root, []).append(v)
    return list(groups.values())

# Example
scc = OnlineSCC(4)
scc.add_edge(0, 1)
scc.add_edge(1, 2)
print("SCCs:", scc.components()) # [[0], [1], [2], [3]]
scc.add_edge(2, 0)
print("SCCs:", scc.components()) # [[0,1,2], [3]]

```

## Why It Matters

- Dynamic SCC maintenance without recomputation
- Handles edge insertions in  $O(V + E)$  amortized
- Enables real-time graph updates
- Basis for more advanced algorithms (fully dynamic SCC with deletions)

## A Gentle Proof (Why It Works)

Every time we add an edge, one of two things happens:

- It connects existing SCCs without creating a cycle → add DAG edge
- It creates a cycle → merge involved SCCs

Since merging SCCs preserves the DAG structure (merging collapses cycles), the algorithm keeps the condensation graph valid at all times.

By maintaining reachability between SCCs, we can detect cycle formation efficiently.

## Try It Yourself

1. Start with 5 nodes, no edges.
2. Add edges step-by-step, printing SCCs.
3. Add a back-edge forming a cycle → watch SCCs merge.
4. Visualize condensation DAG after each update.
5. Compare with recomputing using Tarjan's, they match!

## Test Cases

| Step | Edge | SCCs               |
|------|------|--------------------|
| 1    | 0→1  | {0}, {1}, {2}, {3} |
| 2    | 1→2  | {0}, {1}, {2}, {3} |
| 3    | 2→0  | {0,1,2}, {3}       |
| 4    | 3→1  | {0,1,2}, {3}       |
| 5    | 2→3  | {0,1,2,3}          |

## Complexity

- Time:  $O(V + E)$  amortized (per edge addition)
- Space:  $O(V + E)$  (graph + DSU)

SCC Online Merge is your dynamic sculptor, merging components as new edges appear, maintaining structure without ever starting over.

## 316 Component Label Propagation

Component Label Propagation is a simple, iterative algorithm to find connected components (or strongly connected components in symmetric graphs) by repeatedly propagating minimum labels across edges until all nodes in a component share the same label.

It's conceptually clean, highly parallelizable, and forms the backbone of graph processing frameworks like Google's Pregel, Apache Giraph, and GraphX, perfect for large-scale or distributed systems.

## What Problem Are We Solving?

We want to identify components, groups of vertices that are mutually reachable. Instead of deep recursion or complex stacks, we iteratively propagate labels across the graph until convergence.

This approach is ideal when:

- The graph is massive (too large for recursion)
- You're using parallel / distributed computation
- You want a message-passing style algorithm

For undirected graphs, it finds connected components. For directed graphs, it can approximate SCCs (often used as a preprocessing step).

Example: Think of spreading an ID through a crowd, each node tells its neighbors its smallest known label, and everyone updates to match their smallest neighbor's label. Eventually, all in a group share the same number.

## How Does It Work (Plain Language)?

1. Initialize: Each vertex's label = its own ID.
2. Iterate: For each vertex:
  - Look at all neighbors' labels.
  - Update to the smallest label seen.
3. Repeat until no label changes (convergence).

All nodes that end up sharing a label belong to the same component.

| Step | Node | Current Label | Neighbor Labels | New Label           |
|------|------|---------------|-----------------|---------------------|
| 1    | A    | A             | {B, C}          | $\min(A, B, C) = A$ |
| 2    | B    | B             | {A}             | $\min(B, A) = A$    |
| 3    | C    | C             | {A}             | $\min(C, A) = A$    |

Eventually, A, B, C → all labeled A.

## Tiny Code (Easy Versions)

C (Iterative Label Propagation)

```
#include <stdio.h>
#include <stdbool.h>

#define MAX 100
int n, graph[MAX][MAX];
int label[MAX];

void label_propagation() {
    bool changed = true;
    while (changed) {
        changed = false;
        for (int v = 0; v < n; v++) {
            int min_label = label[v];
            for (int u = 0; u < n; u++) {
                if (graph[v][u]) {
                    if (label[u] < min_label)
                        min_label = label[u];
                }
            }
            if (min_label < label[v]) {
                label[v] = min_label;
                changed = true;
            }
        }
    }
}

int main(void) {
    printf("Enter number of vertices: ");
    scanf("%d", &n);
    printf("Enter adjacency matrix (undirected):\n");
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            scanf("%d", &graph[i][j]);

    for (int i = 0; i < n; i++)
        label[i] = i;

    label_propagation();
}
```

```

printf("Component Labels:\n");
for (int i = 0; i < n; i++)
    printf("Vertex %d → Label %d\n", i, label[i]);
}

```

Python (Simple Version)

```

graph = {
    0: [1],
    1: [0, 2],
    2: [1],
    3: [4],
    4: [3]
}

labels = {v: v for v in graph}

changed = True
while changed:
    changed = False
    for v in graph:
        min_label = min([labels[v]] + [labels[u] for u in graph[v]])
        if min_label < labels[v]:
            labels[v] = min_label
            changed = True

print("Component labels:", labels)

```

## Why It Matters

- Simple and parallelizable, ideal for big data systems
- No recursion or stack, suitable for GPUs, clusters
- Local computation, fits the “think like a vertex” model
- Works on massive graphs where DFS is impractical

## A Gentle Proof (Why It Works)

Each iteration allows label information to flow along edges. Since the smallest label always propagates, and each propagation only decreases label values, the process must converge (no

infinite updates). At convergence, all vertices in a connected component share the same minimal label.

The number of iterations = graph diameter.

### Try It Yourself

1. Run it on small undirected graphs, label flow is easy to track.
2. Try a graph with two disconnected parts, they'll stabilize separately.
3. Add edges between components and rerun, watch labels merge.
4. Use directed edges and see how approximation differs from SCCs.
5. Implement in parallel (multi-threaded loop).

### Test Cases

| Graph         | Edges        | Final Labels |
|---------------|--------------|--------------|
| 0–1–2         | (0,1), (1,2) | {0,0,0}      |
| 0–1, 2–3      | (0,1), (2,3) | {0,0,2,2}    |
| 0–1–2–3       | Chain        | {0,0,0,0}    |
| 0–1, 1–2, 2–0 | Cycle        | {0,0,0}      |

### Complexity

- Time:  $O(V \times E)$  (in worst case, or  $O(D \times E)$  for  $D = \text{diameter}$ )
- Space:  $O(V + E)$

Component Label Propagation is your whispering algorithm, every node shares its name with neighbors, again and again, until all who can reach each other call themselves by the same name.

## 317 Path-Based SCC

The Path-Based SCC algorithm is another elegant one-pass method for finding strongly connected components in a directed graph. It's similar in spirit to Tarjan's algorithm, but instead of computing explicit `low-link` values, it maintains path stacks to detect when a full component has been traversed.

Developed by Donald B. Johnson, it uses two stacks to keep track of DFS path order and potential roots. When a vertex cannot reach any earlier vertex, it becomes the root of an SCC, and the algorithm pops all nodes in that SCC from the path.

## What Problem Are We Solving?

We want to find SCCs in a directed graph, subsets of vertices where each node can reach every other node. Path-Based SCC offers a conceptually simple and efficient way to do this without low-link math.

Why it's useful:

- Single DFS traversal
- Clean stack-based logic
- Great for teaching, reasoning, and implementation clarity
- Easy to extend for incremental or streaming SCC detection

Applications:

- Compiler analysis (strongly connected variables)
- Circuit analysis
- Deadlock detection
- Dataflow optimization

## How Does It Work (Plain Language)?

We maintain:

- `index[v]`: discovery order
- S stack: DFS path (nodes in current exploration)
- P stack: candidates for SCC roots

Algorithm steps:

1. Assign `index[v]` when visiting a node.
2. Push  $v$  onto both stacks (S and P).
3. For each neighbor  $u$ :
  - If  $u$  is unvisited  $\rightarrow$  recurse
  - If  $u$  is on S  $\rightarrow$  adjust P by popping until its top has an index `index[u]`
4. If  $v$  is at the top of P after processing all neighbors:
  - Pop P
  - Pop from S until  $v$  is removed
  - The popped vertices form one SCC

| Step | Current Node  | Stack S | Stack P | Action          |
|------|---------------|---------|---------|-----------------|
| 1    | A             | [A]     | [A]     | Visit A         |
| 2    | B             | [A,B]   | [A,B]   | Visit B         |
| 3    | C             | [A,B,C] | [A,B,C] | Visit C         |
| 4    | Back edge C→B | [A,B,C] | [A,B]   | Adjust          |
| 5    | C done        | [A,B,C] | [A,B]   | Continue        |
| 6    | P top == B    | [A]     | [A]     | Found SCC {B,C} |

### Tiny Code (Easy Versions)

C (Path-Based SCC)

```
#include <stdio.h>
#include <stdbool.h>

#define MAX 100
int graph[MAX][MAX];
int n;
int index_counter = 0;
int indexv[MAX];
bool onStack[MAX];
int stackS[MAX], topS = -1;
int stackP[MAX], topP = -1;

void dfs_scc(int v) {
    indexv[v] = ++index_counter;
    stackS[++topS] = v;
    stackP[++topP] = v;
    onStack[v] = true;

    for (int u = 0; u < n; u++) {
        if (!graph[v][u]) continue;
        if (indexv[u] == 0) {
            dfs_scc(u);
        } else if (onStack[u]) {
            while (indexv[stackP[topP]] > indexv[u])
                topP--;
        }
    }

    if (stackP[topP] == v) {
```

```

        topP--;
        printf("SCC: ");
        int w;
        do {
            w = stackS[topS--];
            onStack[w] = false;
            printf("%d ", w);
        } while (w != v);
        printf("\n");
    }
}

int main(void) {
    printf("Enter number of vertices: ");
    scanf("%d", &n);
    printf("Enter adjacency matrix (directed):\n");
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            scanf("%d", &graph[i][j]);

    for (int i = 0; i < n; i++) indexv[i] = 0;

    printf("Strongly Connected Components:\n");
    for (int i = 0; i < n; i++)
        if (indexv[i] == 0)
            dfs_scc(i);
}

```

Python (Readable Implementation)

```

from collections import defaultdict

graph = defaultdict(list)
edges = [(0,1),(1,2),(2,0),(1,3)]
for u,v in edges:
    graph[u].append(v)

index_counter = 0
index = {}
stackS, stackP = [], []
on_stack = set()
sccs = []

```

```

def dfs(v):
    global index_counter
    index_counter += 1
    index[v] = index_counter
    stackS.append(v)
    stackP.append(v)
    on_stack.add(v)

    for u in graph[v]:
        if u not in index:
            dfs(u)
        elif u in on_stack:
            while index[stackP[-1]] > index[u]:
                stackP.pop()

    if stackP and stackP[-1] == v:
        stackP.pop()
        comp = []
        while True:
            w = stackS.pop()
            on_stack.remove(w)
            comp.append(w)
            if w == v:
                break
        sccs.append(comp)

for v in list(graph.keys()):
    if v not in index:
        dfs(v)

print("Strongly Connected Components:", sccs)

```

## Why It Matters

- Single DFS
- No low-link math, purely path-based reasoning
- Compact and intuitive for stack lovers
- Well-suited for theoretical clarity and educational use
- Matches Tarjan's  $O(V + E)$  performance

## A Gentle Proof (Why It Works)

- Each vertex gets a discovery index;
- Stack  $S$  stores active path nodes;
- Stack  $P$  tracks potential SCC roots (lowest index still reachable). When a vertex finishes and equals top of  $P$ , all nodes above it in  $S$  form an SCC, they're mutually reachable, and none can reach earlier nodes.

The invariant ensures:

- Nodes stay on stack until their SCC is found
- SCCs are discovered in reverse topological order

## Try It Yourself

1. Run it on a small graph, print `index[v]`, stack states after each call.
2. Add a cycle, trace how  $P$  adjusts.
3. Compare output with Tarjan's algorithm, they match!
4. Visualize path-based pops as SCC boundaries.
5. Try graphs with multiple disjoint SCCs.

## Test Cases

| Graph                                                                | Edges  | SCCs                    |
|----------------------------------------------------------------------|--------|-------------------------|
| $0 \rightarrow 1, 1 \rightarrow 2, 2 \rightarrow 0$                  | Cycle  | $\{0,1,2\}$             |
| $0 \rightarrow 1, 1 \rightarrow 2$                                   | Chain  | $\{0\}, \{1\}, \{2\}$   |
| $0 \rightarrow 1, 1 \rightarrow 0, 2 \rightarrow 3$                  | Mixed  | $\{0,1\}, \{2\}, \{3\}$ |
| $0 \rightarrow 1, 1 \rightarrow 2, 2 \rightarrow 3, 3 \rightarrow 1$ | Nested | $\{1,2,3\}, \{0\}$      |

## Complexity

- Time:  $O(V + E)$
- Space:  $O(V)$

Path-Based SCC is your stack ballet, each vertex steps forward, marks its place, and retreats gracefully, leaving behind a tightly choreographed component.

## 318 Kosaraju Parallel Version

The Parallel Kosaraju Algorithm adapts Kosaraju's classic two-pass SCC method to run on multiple processors or threads, making it suitable for large-scale graphs that can't be processed efficiently by a single thread. It divides the heavy lifting, DFS traversals and graph reversals, across many workers.

It's the natural evolution of Kosaraju's idea in the age of parallel computing: split the graph, explore concurrently, merge SCCs.

### What Problem Are We Solving?

We want to compute SCCs in a massive directed graph efficiently, by taking advantage of parallel hardware, multicore CPUs, GPUs, or distributed systems.

The classic Kosaraju algorithm:

1. DFS on the original graph to record finishing times
2. Reverse all edges to create the transpose graph
3. DFS on the reversed graph in decreasing order of finishing time

The parallel version accelerates each phase by partitioning vertices or edges among processors.

Applications include:

- Large dependency graphs (package managers, compilers)
- Web graphs (page connectivity)
- Social networks (mutual reachability)
- GPU-accelerated analytics and graph mining

### How Does It Work (Plain Language)?

We parallelize Kosaraju's two key passes:

1. Parallel Forward DFS (Finishing Order):
  - Partition vertices across threads.
  - Each thread runs DFS independently on its subgraph.
  - Maintain a shared stack of finish times (atomic appends).
2. Graph Reversal:
  - Reverse each edge  $(u, v)$  into  $(v, u)$  in parallel.
  - Each thread processes a slice of the edge list.

### 3. Parallel Reverse DFS (SCC Labeling):

- Threads pop vertices from the global stack.
- Each unvisited node starts a new component.
- DFS labeling runs concurrently with atomic visited flags.

### 4. Merge Components:

- Combine local SCC results using union–find if overlapping sets appear.

| Phase | Description           | Parallelized |
|-------|-----------------------|--------------|
| 1     | DFS on original graph | Yes          |
| 2     | Reverse edges         | Yes          |
| 3     | DFS on reversed graph | Yes          |
| 4     | Merge labels          | Yes          |

### Tiny Code (Pseudocode / Python Threaded Sketch)

This example is conceptual, real parallel implementations use task queues, GPU kernels, or work-stealing schedulers.

```
import threading
from collections import defaultdict

graph = defaultdict(list)
edges = [(0,1), (1,2), (2,0), (2,3), (3,4)]
for u, v in edges:
    graph[u].append(v)

n = 5
visited = [False] * n
finish_order = []
lock = threading.Lock()

def dfs_forward(v):
    visited[v] = True
    for u in graph[v]:
        if not visited[u]:
            dfs_forward(u)
    with lock:
        finish_order.append(v)
```

```

def parallel_forward():
    threads = []
    for v in range(n):
        if not visited[v]:
            t = threading.Thread(target=dfs_forward, args=(v,))
            t.start()
            threads.append(t)
    for t in threads:
        t.join()

# Reverse graph in parallel
rev = defaultdict(list)
for u in graph:
    for v in graph[u]:
        rev[v].append(u)

visited = [False] * n
components = []

def dfs_reverse(v, comp):
    visited[v] = True
    comp.append(v)
    for u in rev[v]:
        if not visited[u]:
            dfs_reverse(u, comp)

def parallel_reverse():
    while finish_order:
        v = finish_order.pop()
        if not visited[v]:
            comp = []
            dfs_reverse(v, comp)
            components.append(comp)

parallel_forward()
parallel_reverse()
print("SCCs:", components)

```

## Why It Matters

- Enables SCC computation at scale

- Exploits multicore / GPU parallelism
- Critical for dataflow analysis, reachability, and graph condensation
- Powers large-scale graph analytics in scientific computing

### A Gentle Proof (Why It Works)

Kosaraju's correctness depends on finishing times and reversed reachability:

- In the forward pass, each vertex  $v$  gets a finishing time  $t(v)$ .
- In the reversed graph, DFS in descending  $t(v)$  ensures that every SCC is discovered as a contiguous DFS tree.

Parallel execution maintains these invariants because:

1. All threads respect atomic finishing-time insertion
2. The global finishing order preserves a valid topological order
3. The reversed DFS still discovers mutually reachable vertices together

Thus, correctness holds under synchronized access, even with concurrent DFS traversals.

### Try It Yourself

1. Split vertices into  $p$  partitions.
2. Run forward DFS in parallel; record global finish stack.
3. Reverse edges concurrently.
4. Run backward DFS by popping from the stack.
5. Compare results with single-threaded Kosaraju, they match.

### Test Cases

| Graph      | Edges                                                                                 | SCCs                       |
|------------|---------------------------------------------------------------------------------------|----------------------------|
| Cycle      | $0 \rightarrow 1, 1 \rightarrow 2, 2 \rightarrow 0$                                   | $\{0, 1, 2\}$              |
| Chain      | $0 \rightarrow 1, 1 \rightarrow 2$                                                    | $\{0\}, \{1\}, \{2\}$      |
| Two cycles | $0 \rightarrow 1, 1 \rightarrow 0, 2 \rightarrow 3, 3 \rightarrow 2$                  | $\{0, 1\}, \{2, 3\}$       |
| Mixed      | $0 \rightarrow 1, 1 \rightarrow 2, 2 \rightarrow 3, 3 \rightarrow 0, 4 \rightarrow 5$ | $\{0, 1, 2, 3\}, \{4, 5\}$ |

## Complexity

Let  $V$  be vertices,  $E$  edges,  $p$  processors.

- Work:  $O(V + E)$  (same as sequential)
- Parallel Time:  $T_p = O! \left( \frac{V+E}{p} + \text{sync\_cost} \right)$
- Space:  $O(V + E)$

Kosaraju Parallel is your multi-voice chorus, each DFS sings in harmony, covering its section of the graph, and when the echoes settle, the full harmony of strongly connected components is revealed.

## 319 Dynamic SCC Maintenance

Dynamic SCC Maintenance deals with maintaining strongly connected components as a directed graph changes over time, edges or vertices may be added or removed. The goal is to update SCCs incrementally rather than recomputing them from scratch after each change.

This approach is crucial in streaming, interactive, or evolving systems, where graphs represent real-world structures that shift continuously.

### What Problem Are We Solving?

We want to keep track of SCCs under dynamic updates:

- Insertions: New edges can connect SCCs and form larger ones.
- Deletions: Removing edges can split SCCs into smaller ones.

Static algorithms like Tarjan or Kosaraju must restart completely. Dynamic maintenance updates only the affected components, improving efficiency for large, frequently changing graphs.

Use cases include:

- Incremental compilation
- Dynamic program analysis
- Real-time dependency resolution
- Continuous graph query systems

## How Does It Work (Plain Language)?

The dynamic SCC algorithm maintains:

- A condensation DAG representing SCCs.
- A reachability structure to detect cycles upon insertion.
- Local re-evaluation for affected nodes.

When an edge  $(u, v)$  is added:

1. Identify components  $C_u$  and  $C_v$ .
2. If  $C_u = C_v$ , no change.
3. If  $C_v$  can reach  $C_u$ , a new cycle forms  $\rightarrow$  merge SCCs.
4. Otherwise, add edge  $C_u \rightarrow C_v$  in the condensation DAG.

When an edge  $(u, v)$  is removed:

1. Remove it from the graph.
2. Check if  $C_u$  and  $C_v$  remain mutually reachable.
3. If not, recompute SCCs locally on the affected subgraph.

| Update                     | Action        | Result           |
|----------------------------|---------------|------------------|
| Add edge forming cycle     | Merge SCCs    | Larger component |
| Add edge without cycle     | DAG edge only | No merge         |
| Remove edge breaking cycle | Split SCC     | New components   |

## Tiny Code (Simplified Example)

```
class DSU:  
    def __init__(self, n):  
        self.parent = list(range(n))  
    def find(self, x):  
        if self.parent[x] != x:  
            self.parent[x] = self.find(self.parent[x])  
        return self.parent[x]  
    def union(self, a, b):  
        ra, rb = self.find(a), self.find(b)  
        if ra != rb:  
            self.parent[rb] = ra  
  
class DynamicSCC:
```

```

def __init__(self, n):
    self.n = n
    self.dsu = DSU(n)
    self.graph = [set() for _ in range(n)]

def add_edge(self, u, v):
    su, sv = self.dsu.find(u), self.dsu.find(v)
    if su == sv:
        return
    if self._reachable(sv, su):
        self.dsu.union(su, sv)
    else:
        self.graph[su].add(sv)

def _reachable(self, start, target, seen=None):
    if seen is None: seen = set()
    if start == target: return True
    seen.add(start)
    for nxt in self.graph[start]:
        if nxt not in seen and self._reachable(nxt, target, seen):
            return True
    return False

def components(self):
    comps = {}
    for v in range(self.n):
        root = self.dsu.find(v)
        comps.setdefault(root, []).append(v)
    return list(comps.values())

scc = DynamicSCC(4)
scc.add_edge(0, 1)
scc.add_edge(1, 2)
print(scc.components()) # [[0], [1], [2], [3]]
scc.add_edge(2, 0)
print(scc.components()) # [[0,1,2], [3]]

```

## Why It Matters

- Efficient for long-running systems where graphs evolve
- Updates SCCs incrementally rather than rebuilding

- Supports real-time queries of connectivity
- Useful for streaming graph databases, incremental compilers, interactive modeling tools

### A Gentle Proof (Why It Works)

For insertion:

- If  $(u, v)$  connects components  $C_u$  and  $C_v$
- And if  $C_v$  can reach  $C_u$ , then a cycle forms
- Merging  $C_u$  and  $C_v$  yields a valid SCC
- The condensation DAG remains acyclic

For deletion:

- If removal breaks reachability, SCC splits
- Recomputing locally ensures correctness
- Other unaffected SCCs remain valid

Each update modifies only the local neighborhood of the affected components.

### Try It Yourself

1. Build a small directed graph.
2. Insert edges step by step and print components.
3. Add a back-edge to create a cycle and observe merging.
4. Remove an edge and check local recomputation.
5. Compare results with full Tarjan recomputation.

### Test Cases

| Step | Edge                     | SCCs                    |
|------|--------------------------|-------------------------|
| 1    | $0 \rightarrow 1$        | $\{0\}, \{1\}, \{2\}$   |
| 2    | $1 \rightarrow 2$        | $\{0\}, \{1\}, \{2\}$   |
| 3    | $2 \rightarrow 0$        | $\{0,1,2\}$             |
| 4    | $2 \rightarrow 3$        | $\{0,1,2\}, \{3\}$      |
| 5    | remove $1 \rightarrow 2$ | $\{0,1\}, \{2\}, \{3\}$ |

## Complexity

- Insertion:  $O(V + E)$  amortized (with reachability check)
- Deletion: Local recomputation, typically sublinear
- Space:  $O(V + E)$

Dynamic SCC Maintenance provides a framework to keep SCCs consistent as the graph evolves, adapting efficiently to both incremental growth and structural decay.

## 320 SCC for Weighted Graph

SCC detection is usually discussed for unweighted graphs, where edge weights are irrelevant to reachability. However, in many real-world systems, weights encode constraints (cost, capacity, priority, probability), and we need to identify strong connectivity under these weighted conditions. This variant integrates SCC algorithms with weighted edge logic, allowing selective inclusion or exclusion of edges based on weight criteria.

### What Problem Are We Solving?

We want to find strongly connected components in a weighted directed graph. A standard SCC algorithm ignores weights, it only checks reachability. Here, we define SCCs based on edges satisfying a given weight predicate:

$$(u, v) \in E_w \quad \text{iff} \quad w(u, v) \leq \theta$$

We can then run SCC algorithms (Tarjan, Kosaraju, Gabow, Path-based) on the subgraph induced by edges satisfying the constraint.

Common use cases:

- Thresholded connectivity: Keep edges below cost  $\theta$ .
- Capacity-limited systems: Only include edges with capacity threshold.
- Dynamic constraint graphs: Recompute SCCs as thresholds shift.
- Probabilistic networks: Consider edges with probability  $p$ .

## How Does It Work (Plain Language)?

1. Start with a weighted directed graph  $G = (V, E, w)$
2. Apply a predicate on weights (e.g.  $w(u, v) \leq \theta$ )
3. Build a filtered subgraph  $G_\theta = (V, E_\theta)$
4. Run a standard SCC algorithm on  $G_\theta$

The result groups vertices that are strongly connected under the weight constraint.

If  $\theta$  changes, components can merge or split:

- Increasing  $\theta$  (loosening)  $\rightarrow$  SCCs merge
- Decreasing  $\theta$  (tightening)  $\rightarrow$  SCCs split

| Threshold $\theta$ | Included Edges | SCCs       |
|--------------------|----------------|------------|
| 3                  | $w \leq 3$     | {A,B}, {C} |
| 5                  | $w \leq 5$     | {A,B,C}    |

## Tiny Code (Threshold-Based Filtering)

Python Example

```
from collections import defaultdict

edges = [
    (0, 1, 2),
    (1, 2, 4),
    (2, 0, 1),
    (2, 3, 6),
    (3, 2, 6)
]

def build_subgraph(edges, theta):
    g = defaultdict(list)
    for u, v, w in edges:
        if w <= theta:
            g[u].append(v)
    return g

def dfs(v, g, visited, stack):
    visited.add(v)
    for u in g[v]:
```

```

        if u not in visited:
            dfs(u, g, visited, stack)
    stack.append(v)

def reverse_graph(g):
    rg = defaultdict(list)
    for u in g:
        for v in g[u]:
            rg[v].append(u)
    return rg

def kosaraju(g):
    visited, stack = set(), []
    for v in g:
        if v not in visited:
            dfs(v, g, visited, stack)
    rg = reverse_graph(g)
    visited.clear()
    comps = []
    while stack:
        v = stack.pop()
        if v not in visited:
            comp = []
            dfs(v, rg, visited, comp)
            comps.append(comp)
    return comps

theta = 4
g_theta = build_subgraph(edges, theta)
print("SCCs with threshold", theta, ":", kosaraju(g_theta))

```

Output:

```
SCCs with threshold 4 : [[2, 0, 1], [3]]
```

## Why It Matters

- Incorporates weight constraints into connectivity
- Useful in optimization, routing, and clustering
- Supports incremental recomputation under shifting thresholds
- Enables multi-layer graph analysis (vary  $\theta$  to see component evolution)

## A Gentle Proof (Why It Works)

Reachability in a weighted graph depends on which edges are active. Filtering edges by predicate preserves a subset of original reachability:

If there is a path  $u \rightarrow v$  in  $G_\theta$ , all edges on that path satisfy  $w(e) \leq \theta$ . Since SCCs depend solely on reachability, standard algorithms applied to  $G_\theta$  correctly identify weight-constrained SCCs.

As  $\theta$  increases, the edge set  $E_\theta$  grows monotonically:

$$E_{\theta_1} \subseteq E_{\theta_2} \quad \text{for } \theta_1 < \theta_2$$

Therefore, the SCC partition becomes coarser (components merge).

## Try It Yourself

1. Build a weighted graph.
2. Pick thresholds  $\theta = 2, 4, 6$  and record SCCs.
3. Plot how components merge as  $\theta$  increases.
4. Try predicates like  $w(u, v) \geq \theta$ .
5. Combine with Dynamic SCC Maintenance for evolving thresholds.

## Test Cases

| $\theta$ | Edges Included      | SCCs            |
|----------|---------------------|-----------------|
| 2        | (0→1,2), (2→0)      | {0,2}, {1}, {3} |
| 4        | (0→1), (1→2), (2→0) | {0,1,2}, {3}    |
| 6        | All edges           | {0,1,2,3}       |

## Complexity

- Filtering:  $O(E)$
- SCC computation:  $O(V + E_\theta)$
- Total:  $O(V + E)$
- Space:  $O(V + E)$

SCC for Weighted Graphs extends classic connectivity to contexts where not all edges are equal, revealing the layered structure of a graph as thresholds vary.

## Section 33. Shortest Paths

### 321 Dijkstra (Binary Heap)

Dijkstra's Algorithm is the cornerstone of shortest path computation in weighted graphs with nonnegative edge weights. It grows a frontier of known shortest paths, always expanding from the vertex with the smallest current distance, much like a wavefront advancing through the graph.

Using a binary heap (priority queue) keeps the next closest vertex selection efficient, making this version the standard for practical use.

#### What Problem Are We Solving?

We need to find the shortest path from a single source vertex  $s$  to all other vertices in a directed or undirected graph with nonnegative weights.

Given a weighted graph  $G = (V, E, w)$  where  $w(u, v) \geq 0$  for all  $(u, v)$ , the task is to compute:

$$\text{dist}[v] = \min_{\text{path } s \rightarrow v} \sum_{(u,v) \in \text{path}} w(u, v)$$

Typical use cases:

- GPS navigation (road networks)
- Network routing
- Pathfinding in games
- Dependency resolution in weighted systems

#### How Does It Work (Plain Language)?

The algorithm maintains a distance array `dist[]`, initialized with infinity for all vertices except the source.

1. Set `dist[s] = 0`.
2. Use a min-priority queue to repeatedly extract the vertex with the smallest distance.
3. For each neighbor, try to relax the edge:

if  $\text{dist}[u] + w(u, v) < \text{dist}[v]$ , then update  $\text{dist}[v]$

4. Push the neighbor into the queue with updated distance.
5. Continue until the queue is empty.

| Step | Vertex Extracted | Updated Distances            |
|------|------------------|------------------------------|
| 1    | $s$              | 0                            |
| 2    | Next smallest    | Update neighbors             |
| 3    | Repeat           | Until all vertices finalized |

This is a greedy algorithm, once a vertex is visited, its shortest distance is final.

### Tiny Code (Binary Heap)

C (Using a Simple Priority Queue with `qsort`)

```
#include <stdio.h>
#include <limits.h>
#include <stdbool.h>

#define MAX 100
#define INF INT_MAX

int n, graph[MAX][MAX];

void dijkstra(int src) {
    int dist[MAX], visited[MAX];
    for (int i = 0; i < n; i++) {
        dist[i] = INF;
        visited[i] = 0;
    }
    dist[src] = 0;

    for (int count = 0; count < n - 1; count++) {
        int u = -1, min = INF;
        for (int v = 0; v < n; v++)
            if (!visited[v] && dist[v] < min)
                min = dist[v], u = v;

        if (u == -1) break;
        visited[u] = 1;

        for (int v = 0; v < n; v++)
            if (graph[u][v] && !visited[v] &&
                dist[u] + graph[u][v] < dist[v])
                    dist[v] = dist[u] + graph[u][v];
    }
}
```

```

        dist[v] = dist[u] + graph[u][v];
    }

    printf("Vertex\tDistance\n");
    for (int i = 0; i < n; i++)
        printf("%d\t%d\n", i, dist[i]);
}

int main(void) {
    printf("Enter number of vertices: ");
    scanf("%d", &n);
    printf("Enter adjacency matrix (0 for no edge):\n");
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            scanf("%d", &graph[i][j]);
    dijkstra(0);
}

```

Python (Heap-Based Implementation)

```

import heapq

def dijkstra(graph, src):
    n = len(graph)
    dist = [float('inf')] * n
    dist[src] = 0
    pq = [(0, src)]

    while pq:
        d, u = heapq.heappop(pq)
        if d > dist[u]:
            continue
        for v, w in graph[u]:
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                heapq.heappush(pq, (dist[v], v))
    return dist

graph = {
    0: [(1, 2), (2, 4)],
    1: [(2, 1), (3, 7)],
    2: [(4, 3)],
}

```

```

    3: [(4, 1)],
    4: []
}

print(dijkstra(graph, 0))

```

## Why It Matters

- Efficient shortest path when weights are nonnegative
- Deterministic and greedy, produces optimal paths
- Widely applicable to routing, logistics, and AI search
- Forms the foundation for A\* and Johnson's algorithm

## A Gentle Proof (Why It Works)

Dijkstra's invariant: When a vertex  $u$  is extracted from the priority queue,  $\text{dist}[u]$  is final.

Proof idea: All alternative paths to  $u$  must go through vertices with greater or equal tentative distances, since edges are nonnegative. Thus, no shorter path exists.

By induction, the algorithm assigns the correct shortest distance to every vertex.

## Try It Yourself

1. Run on a graph with 5 vertices and random weights.
2. Add an edge with a smaller weight and see how paths update.
3. Remove negative edges and note incorrect results.
4. Visualize the frontier expansion step by step.
5. Compare with Bellman–Ford on the same graph.

## Test Cases

| Graph  | Edges                                                                                                                    | Shortest Paths from 0 |
|--------|--------------------------------------------------------------------------------------------------------------------------|-----------------------|
| Simple | $0 \rightarrow 1(2), 0 \rightarrow 2(4), 1 \rightarrow 2(1), 2 \rightarrow 4(3), 1 \rightarrow 3(7), 3 \rightarrow 4(1)$ | $[0, 2, 3, 7, 6]$     |
| Chain  | $0 \rightarrow 1(1), 1 \rightarrow 2(1), 2 \rightarrow 3(1)$                                                             | $[0, 1, 2, 3]$        |
| Star   | $0 \rightarrow 1(5), 0 \rightarrow 2(2), 0 \rightarrow 3(8)$                                                             | $[0, 5, 2, 8]$        |

## Complexity

- Time:  $O((V + E) \log V)$  with binary heap
- Space:  $O(V + E)$
- Works only if:  $w(u, v) \geq 0$

Dijkstra (Binary Heap) is the workhorse of graph search, greedy but precise, always chasing the next closest frontier until all paths fall into place.

## 322 Dijkstra (Fibonacci Heap)

Dijkstra's algorithm can be further optimized by replacing the binary heap with a Fibonacci heap, which offers faster decrease-key operations. This improvement reduces the overall time complexity, making it more suitable for dense graphs or theoretical analysis where asymptotic efficiency matters.

While the constant factors are higher, the asymptotic time is improved to:

$$O(E + V \log V)$$

compared to the binary heap's  $O((V + E) \log V)$ .

## What Problem Are We Solving?

We are computing single-source shortest paths in a directed or undirected weighted graph with nonnegative weights, but we want to optimize the priority queue operations to improve theoretical performance.

Given  $G = (V, E, w)$  with  $w(u, v) \geq 0$ , the task remains:

$$\text{dist}[v] = \min_{\text{path } s \rightarrow v} \sum_{(u,v) \in \text{path}} w(u, v)$$

The difference lies in how we manage the priority queue that selects the next vertex to process.

## How Does It Work (Plain Language)?

The logic of Dijkstra's algorithm is unchanged, only the data structure used for vertex selection and updates differs.

1. Initialize all distances to  $\infty$ , except  $\text{dist}[s] = 0$ .
2. Insert all vertices into a Fibonacci heap keyed by their current distance.
3. Repeatedly extract the vertex  $u$  with smallest distance.
4. For each neighbor  $(u, v)$ :

- If  $\text{dist}[u] + w(u, v) < \text{dist}[v]$ , update:

$$\text{dist}[v] \leftarrow \text{dist}[u] + w(u, v)$$

and call decrease-key on  $v$  in the heap.

5. Continue until all vertices are finalized.

The Fibonacci heap provides:

- `extract-min`:  $O(\log V)$  amortized
- `decrease-key`:  $O(1)$  amortized

This improves the performance for dense graphs where edge relaxations dominate.

## Tiny Code (Python, Simplified Fibonacci Heap)

This code illustrates the structure but omits full heap details, production implementations use libraries like `networkx` or specialized data structures.

```
from heapq import heappush, heappop # stand-in for demonstration

def dijkstra_fib(graph, src):
    n = len(graph)
    dist = [float('inf')] * n
    dist[src] = 0
    visited = [False] * n
    heap = [(0, src)]

    while heap:
        d, u = heappop(heap)
        if visited[u]:
```

```

        continue
visited[u] = True
for v, w in graph[u]:
    if not visited[v] and dist[u] + w < dist[v]:
        dist[v] = dist[u] + w
        heappush(heap, (dist[v], v))
return dist

graph = {
    0: [(1, 1), (2, 4)],
    1: [(2, 2), (3, 6)],
    2: [(3, 3)],
    3: []
}

print(dijkstra_fib(graph, 0))

```

(The above uses `heapq` for illustration; true Fibonacci heap gives better theoretical bounds.)

## Why It Matters

- Improves theoretical time to  $O(E + V \log V)$
- Demonstrates asymptotic optimization using advanced heaps
- Used in dense networks, theoretical research, and competition problems
- Foundation for algorithms like Johnson's APSP and minimum mean cycle

## A Gentle Proof (Why It Works)

The correctness remains identical to Dijkstra's original proof: When a vertex  $u$  is extracted (minimum key), its shortest distance is final because all edge weights are nonnegative.

The heap choice only affects efficiency:

- Binary heap: every `decrease-key` =  $O(\log V)$
- Fibonacci heap: every `decrease-key` =  $O(1)$  amortized

Total operations:

- $V$  extractions  $\times O(\log V)$
- $E$  decreases  $\times O(1)$

Hence total time:

$$O(V \log V + E) = O(E + V \log V)$$

### Try It Yourself

1. Build a dense graph (e.g.  $V = 1000, E \approx V^2$ ).
2. Compare runtimes with binary heap version.
3. Visualize priority queue operations.
4. Implement `decrease-key` manually for insight.
5. Explore Johnson's algorithm using this version.

### Test Cases

| Graph    | Edges                                                        | Shortest Paths from 0 |
|----------|--------------------------------------------------------------|-----------------------|
| Chain    | $0 \rightarrow 1(1), 1 \rightarrow 2(2), 2 \rightarrow 3(3)$ | $[0, 1, 3, 6]$        |
| Triangle | $0 \rightarrow 1(2), 1 \rightarrow 2(2), 0 \rightarrow 2(5)$ | $[0, 2, 4]$           |
| Dense    | All pairs with small weights                                 | Works efficiently     |

### Complexity

- Time:  $O(E + V \log V)$
- Space:  $O(V + E)$
- Works only if:  $w(u, v) \geq 0$

Dijkstra (Fibonacci Heap) shows how data structure choice transforms an algorithm, the same idea, made sharper through careful engineering of priority operations.

## 323 Bellman–Ford

The Bellman–Ford algorithm solves the single-source shortest path problem for graphs that may contain negative edge weights. Unlike Dijkstra's algorithm, it does not rely on greedy selection and can handle edges with  $w(u, v) < 0$ , as long as there are no negative-weight cycles reachable from the source.

It systematically relaxes every edge multiple times, ensuring all paths up to length  $V - 1$  are considered.

## What Problem Are We Solving?

We want to compute shortest paths from a source  $s$  in a weighted directed graph that may include negative weights.

Given  $G = (V, E, w)$ , find for all  $v \in V$ :

$$\text{dist}[v] = \min_{\text{path } s \rightarrow v} \sum_{(u,v) \in \text{path}} w(u,v)$$

If a negative-weight cycle is reachable from  $s$ , the shortest path is undefined (it can be reduced indefinitely). Bellman–Ford detects this situation explicitly.

## How Does It Work (Plain Language)?

Bellman–Ford uses edge relaxation repeatedly.

1. Initialize  $\text{dist}[s] = 0$ , others  $\infty$ .
2. Repeat  $V - 1$  times: For every edge  $(u, v)$ :

if  $\text{dist}[u] + w(u, v) < \text{dist}[v]$ , then update  $\text{dist}[v]$

3. After  $V - 1$  passes, all shortest paths are settled.
4. Perform one more pass: if any edge can still relax, a negative-weight cycle exists.

| Iteration | Updated Vertices              | Notes       |
|-----------|-------------------------------|-------------|
| 1         | Neighbors of source           | First layer |
| 2         | Next layer                    | Propagate   |
| ...       | ...                           | ...         |
| $V - 1$   | All shortest paths stabilized | Done        |

## Tiny Code (C Example)

```
#include <stdio.h>
#include <limits.h>

#define MAX 100
#define INF 1000000000

typedef struct { int u, v, w; } Edge;
```

```

Edge edges[MAX];
int dist[MAX];

int main(void) {
    int V, E, s;
    printf("Enter vertices, edges, source: ");
    scanf("%d %d %d", &V, &E, &s);
    printf("Enter edges (u v w):\n");
    for (int i = 0; i < E; i++)
        scanf("%d %d %d", &edges[i].u, &edges[i].v, &edges[i].w);

    for (int i = 0; i < V; i++) dist[i] = INF;
    dist[s] = 0;

    for (int i = 1; i < V; i++)
        for (int j = 0; j < E; j++) {
            int u = edges[j].u, v = edges[j].v, w = edges[j].w;
            if (dist[u] != INF && dist[u] + w < dist[v])
                dist[v] = dist[u] + w;
        }

    for (int j = 0; j < E; j++) {
        int u = edges[j].u, v = edges[j].v, w = edges[j].w;
        if (dist[u] != INF && dist[u] + w < dist[v]) {
            printf("Negative cycle detected\n");
            return 0;
        }
    }
}

printf("Vertex\tDistance\n");
for (int i = 0; i < V; i++)
    printf("%d\t%d\n", i, dist[i]);
}

```

Python (Readable Version)

```

def bellman_ford(V, edges, src):
    dist = [float('inf')] * V
    dist[src] = 0

    for _ in range(V - 1):
        for u, v, w in edges:

```

```

        if dist[u] + w < dist[v]:
            dist[v] = dist[u] + w

    # Detect negative cycle
    for u, v, w in edges:
        if dist[u] + w < dist[v]:
            raise ValueError("Negative cycle detected")

    return dist

edges = [(0, 1, 6), (0, 2, 7), (1, 2, 8), (1, 3, 5),
         (1, 4, -4), (2, 3, -3), (2, 4, 9), (3, 1, -2),
         (4, 3, 7)]

print(bellman_ford(5, edges, 0))

```

## Why It Matters

- Works with negative weights
- Detects negative-weight cycles
- Simpler logic, easier to prove correctness
- Used in currency arbitrage, dynamic programming, policy evaluation

## A Gentle Proof (Why It Works)

A shortest path has at most  $V - 1$  edges (a path longer than that must contain a cycle). Each iteration ensures all paths up to that length are relaxed. Thus after  $V - 1$  rounds, all shortest paths are found.

The  $(V)$ -th round detects further improvement, indicating a negative cycle.

Formally, after iteration  $k$ ,  $\text{dist}[v]$  is the length of the shortest path from  $s$  to  $v$  using at most  $k$  edges.

## Try It Yourself

1. Run on graphs with negative weights.
2. Add a negative cycle and observe detection.
3. Compare results with Dijkstra (fails with negative edges).
4. Visualize relaxation per iteration.
5. Use it to detect arbitrage in currency exchange graphs.

## Test Cases

| Graph          | Edges                                                          | Shortest Distances |
|----------------|----------------------------------------------------------------|--------------------|
| Chain          | $0 \rightarrow 1(5), 1 \rightarrow 2(-2)$                      | $[0, 5, 3]$        |
| Negative edge  | $0 \rightarrow 1(4), 0 \rightarrow 2(5), 1 \rightarrow 2(-10)$ | $[0, 4, -6]$       |
| Negative cycle | $0 \rightarrow 1(1), 1 \rightarrow 2(-2), 2 \rightarrow 0(-1)$ | Detected           |

## Complexity

- Time:  $O(VE)$
- Space:  $O(V)$
- Handles:  $w(u, v) \geq -\infty$ , no negative cycles

Bellman–Ford is the steady walker of shortest path algorithms, slower than Dijkstra, but unshaken by negative edges and always alert to cycles that break the rules.

## 324 SPFA (Queue Optimization)

The Shortest Path Faster Algorithm (SPFA) is an optimized implementation of Bellman–Ford that uses a queue to avoid unnecessary relaxations. Instead of relaxing all edges in every iteration, SPFA only processes vertices whose distances were recently updated, often resulting in much faster average performance, especially in sparse graphs or those without negative cycles.

In the worst case, it still runs in  $O(VE)$ , but typical performance is closer to  $O(E)$ .

## What Problem Are We Solving?

We want to find single-source shortest paths in a graph that may contain negative weights, but no negative-weight cycles.

Given a directed graph  $G = (V, E, w)$  with edge weights  $w(u, v)$  possibly negative, we compute:

$$\text{dist}[v] = \min_{\text{path } s \rightarrow v} \sum_{(u,v) \in \text{path}} w(u, v)$$

Bellman–Ford's  $V - 1$  rounds of edge relaxation can be wasteful; SPFA avoids rechecking edges from vertices that haven't improved.

## How Does It Work (Plain Language)?

SPFA keeps a queue of vertices whose outgoing edges might lead to relaxation. Each time a vertex's distance improves, it's enqueued for processing.

1. Initialize  $\text{dist}[s] = 0$ , others  $\infty$ .
2. Push  $s$  into a queue.
3. While the queue is not empty:
  - Pop vertex  $u$ .
  - For each  $(u, v)$ :  
    if  $\text{dist}[u] + w(u, v) < \text{dist}[v]$ , then update  $\text{dist}[v]$
  - If  $\text{dist}[v]$  changed and  $v$  is not in queue, enqueue  $v$ .
4. Continue until queue is empty.

| Step | Queue                     | Operation                  |
|------|---------------------------|----------------------------|
| 1    | [ $s$ ]                   | Start                      |
| 2    | Pop $u$ , relax neighbors | Push improved vertices     |
| 3    | Repeat                    | Until no more improvements |

SPFA uses a lazy relaxation strategy, guided by actual updates.

## Tiny Code (C Example)

```
#include <stdio.h>
#include <stdbool.h>
#include <limits.h>

#define MAX 100
#define INF 1000000000

typedef struct { int v, w; } Edge;
Edge graph[MAX][MAX];
int deg[MAX];

int queue[MAX], front = 0, rear = 0;
bool in_queue[MAX];
```

```

int dist[MAX];

void enqueue(int x) {
    queue[rear++] = x;
    in_queue[x] = true;
}
int dequeue() {
    int x = queue[front++];
    in_queue[x] = false;
    return x;
}

int main(void) {
    int V, E, s;
    printf("Enter vertices, edges, source: ");
    scanf("%d %d %d", &V, &E, &s);

    for (int i = 0; i < V; i++) deg[i] = 0;
    printf("Enter edges (u v w):\n");
    for (int i = 0; i < E; i++) {
        int u, v, w;
        scanf("%d %d %d", &u, &v, &w);
        graph[u][deg[u]].v = v;
        graph[u][deg[u]].w = w;
        deg[u]++;
    }

    for (int i = 0; i < V; i++) dist[i] = INF;
    dist[s] = 0;
    enqueue(s);

    while (front < rear) {
        int u = dequeue();
        for (int i = 0; i < deg[u]; i++) {
            int v = graph[u][i].v, w = graph[u][i].w;
            if (dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                if (!in_queue[v]) enqueue(v);
            }
        }
    }
}

```

```

printf("Vertex\tDistance\n");
for (int i = 0; i < V; i++)
    printf("%d\t%d\n", i, dist[i]);
}

```

Python (Queue-Based Implementation)

```

from collections import deque

def spfa(V, edges, src):
    graph = [[] for _ in range(V)]
    for u, v, w in edges:
        graph[u].append((v, w))

    dist = [float('inf')] * V
    in_queue = [False] * V
    dist[src] = 0

    q = deque([src])
    in_queue[src] = True

    while q:
        u = q.popleft()
        in_queue[u] = False
        for v, w in graph[u]:
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                if not in_queue[v]:
                    q.append(v)
                    in_queue[v] = True
    return dist

edges = [(0,1,2),(0,2,4),(1,2,-1),(2,3,2)]
print(spfa(4, edges, 0))

```

## Why It Matters

- Practical improvement over Bellman–Ford
- Efficient for sparse and nearly acyclic graphs
- Can handle negative weights
- Used in network optimization, real-time routing, flow systems

## A Gentle Proof (Why It Works)

Each vertex enters the queue when its distance improves. At most  $V - 1$  improvements per vertex (no shorter path with more edges). Thus, every relaxation converges to the same fixed point as Bellman–Ford.

SPFA is an asynchronous relaxation method:

- Still guarantees correctness under nonnegative cycles
- Detects negative cycles if a vertex is enqueued  $\geq V$  times

To check for negative cycles:

- Maintain a  $\text{count}[v]$  of relaxations
- If  $\text{count}[v] > V$ , report a cycle

## Try It Yourself

1. Test on graphs with negative edges.
2. Compare runtime with Bellman–Ford.
3. Add negative cycle detection.
4. Try both sparse and dense graphs.
5. Measure how queue length varies during execution.

## Test Cases

| Graph         | Edges                                                                             | Result         |
|---------------|-----------------------------------------------------------------------------------|----------------|
| Simple        | $0 \rightarrow 1(2), 0 \rightarrow 2(4), 1 \rightarrow 2(-1), 2 \rightarrow 3(2)$ | $[0, 2, 1, 3]$ |
| Negative edge | $0 \rightarrow 1(5), 1 \rightarrow 2(-3)$                                         | $[0, 5, 2]$    |
| Cycle         | $0 \rightarrow 1(1), 1 \rightarrow 2(-2), 2 \rightarrow 0(1)$                     | Detect         |

## Complexity

- Average:  $O(E)$
- Worst case:  $O(VE)$
- Space:  $O(V + E)$

SPFA (Queue Optimization) is the agile Bellman–Ford, reacting only when change is needed, converging faster in practice while preserving the same correctness guarantees.

## 325 A\* Search

The A\* (A-star) algorithm combines Dijkstra's shortest path with best-first search, guided by a heuristic function. It efficiently finds the shortest path from a start node to a goal node by always expanding the vertex that seems closest to the goal, according to the estimate:

$$f(v) = g(v) + h(v)$$

where

- $g(v)$  = cost from start to  $v$  (known),
- $h(v)$  = heuristic estimate from  $v$  to goal (guessed),
- $f(v)$  = total estimated cost through  $v$ .

When the heuristic is admissible (never overestimates), A\* guarantees optimality.

### What Problem Are We Solving?

We want to find the shortest path from a source  $s$  to a target  $t$  in a weighted graph (often spatial), using additional knowledge about the goal to guide the search.

Given  $G = (V, E, w)$  and a heuristic  $h(v)$ , the task is to minimize:

$$\text{cost}(s, t) = \min_{\text{path } s \rightarrow t} \sum_{(u,v) \in \text{path}} w(u, v)$$

Applications:

- Pathfinding (games, robotics, navigation)
- Planning systems (AI, logistics)
- Grid and map searches
- State-space exploration

### How Does It Work (Plain Language)?

A\* behaves like Dijkstra's algorithm, but instead of expanding the closest node to the start ( $g$ ), it expands the one with smallest estimated total cost ( $f = g + h$ ).

1. Initialize all distances:  $g[\text{start}] = 0$ , others  $\infty$ .
2. Compute  $f[\text{start}] = h[\text{start}]$ .
3. Push  $(f, \text{node})$  into a priority queue.

4. While queue not empty:

- Pop node  $u$  with smallest  $f(u)$ .
- If  $u = \text{goal}$ , stop, path found.
- For each neighbor  $v$ :

$$g'(v) = g(u) + w(u, v)$$

If  $g'(v) < g(v)$ , update:

$$g(v) = g'(v), \quad f(v) = g(v) + h(v)$$

Push  $v$  into the queue.

5. Reconstruct path using parent pointers.

| Node  | $g(v)$ | $h(v)$    | $f(v) = g + h$ | Expanded? |
|-------|--------|-----------|----------------|-----------|
| start | 0      | heuristic | heuristic      |           |
| ...   | ...    | ...       | ...            | ...       |

The heuristic guides exploration, focusing on promising routes.

### Tiny Code (Python Example)

Grid-based A\* (Manhattan heuristic):

```
import heapq

def heuristic(a, b):
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

def astar(grid, start, goal):
    rows, cols = len(grid), len(grid[0])
    g = {start: 0}
    f = {start: heuristic(start, goal)}
    pq = [(f[start], start)]
    parent = {start: None}
    visited = set()

    while pq:
        _, current = heapq.heappop(pq)
        if current == goal:
            path = []
            while current != start:
                path.append(current)
                current = parent[current]
            path.append(start)
            path.reverse()
            return path
        for neighbor in [(current[0]-1, current[1]), (current[0]+1, current[1]), (current[0], current[1]-1), (current[0], current[1]+1)]:
            if 0 <= neighbor[0] < rows and 0 <= neighbor[1] < cols and neighbor not in visited:
                tentative_g = g[current] + 1
                if neighbor not in g or tentative_g < g[neighbor]:
                    g[neighbor] = tentative_g
                    f[neighbor] = g[neighbor] + heuristic(neighbor, goal)
                    heapq.heappush(pq, (f[neighbor], neighbor))
                    parent[neighbor] = current
        visited.add(current)
```

```

        while current:
            path.append(current)
            current = parent[current]
        return path[::-1]

    visited.add(current)
    x, y = current
    for dx, dy in [(1,0),(-1,0),(0,1),(0,-1)]:
        nx, ny = x + dx, y + dy
        neighbor = (nx, ny)
        if 0 <= nx < rows and 0 <= ny < cols and grid[nx][ny] == 0:
            tentative = g[current] + 1
            if tentative < g.get(neighbor, float('inf')):
                g[neighbor] = tentative
                f[neighbor] = tentative + heuristic(neighbor, goal)
                parent[neighbor] = current
                heapq.heappush(pq, (f[neighbor], neighbor))

    return None

grid = [
    [0,0,0,0],
    [1,1,0,1],
    [0,0,0,0],
    [0,1,1,0],
]
start = (0,0)
goal = (3,3)
path = astar(grid, start, goal)
print(path)

```

## Why It Matters

- Faster than Dijkstra for goal-directed search
- Optimal if heuristic is admissible ( $h(v) \leq$  true cost)
- Efficient if heuristic is also consistent (triangle inequality)
- Widely used in AI, robotics, navigation, route planning

## A Gentle Proof (Why It Works)

If the heuristic  $h(v)$  never overestimates the true remaining cost:

$$h(v) \leq \text{cost}(v, t)$$

then  $f(v) = g(v) + h(v)$  is always a lower bound on the true cost. Therefore, when the goal is extracted (smallest  $f$ ), the path is guaranteed optimal.

If  $h$  also satisfies consistency:

$$h(u) \leq w(u, v) + h(v)$$

then  $f$ -values are nondecreasing, and each node is expanded only once.

### Try It Yourself

1. Implement A\* on a grid with obstacles.
2. Experiment with different heuristics (Manhattan, Euclidean).
3. Set  $h(v) = 0 \rightarrow$  becomes Dijkstra.
4. Set  $h(v) = \text{true distance} \rightarrow$  ideal search.
5. Try inadmissible  $h \rightarrow$  faster but possibly suboptimal.

### Test Cases

| Graph Type            | Heuristic | Result              |
|-----------------------|-----------|---------------------|
| Grid ( $4 \times 4$ ) | Manhattan | Shortest path found |
| Weighted              | Euclidean | Optimal route       |
| All $h = 0$           | None      | Becomes Dijkstra    |

### Complexity

- Time:  $O(E \log V)$  (depends on heuristic)
- Space:  $O(V)$
- Optimal if:  $h$  admissible
- Complete if: finite branching factor

A\* Search is Dijkstra with foresight, driven not just by cost so far, but by an informed guess of the journey ahead.

## 326 Floyd–Warshall

The Floyd–Warshall algorithm is a dynamic programming approach for computing all-pairs shortest paths (APSP) in a weighted directed graph. It iteratively refines the shortest path estimates between every pair of vertices by allowing intermediate vertices step by step.

It works even with negative weights, as long as there are no negative cycles.

### What Problem Are We Solving?

We want to compute:

$$\text{dist}(u, v) = \min_{\substack{\text{paths } u \rightarrow v \\ (x,y) \in \text{path}}} \sum_{(x,y) \in \text{path}} w(x, y)$$

for all pairs  $(u, v)$  in a graph  $G = (V, E, w)$ .

We allow negative weights but no negative cycles. It's particularly useful when:

- We need all-pairs shortest paths.
- The graph is dense ( $E \approx V^2$ ).
- We want transitive closure or reachability (set  $w(u, v) = 1$ ).

### How Does It Work (Plain Language)?

We progressively allow each vertex as a possible intermediate waypoint. Initially, the shortest path from  $i$  to  $j$  is just the direct edge. Then, for each vertex  $k$ , we check if a path through  $k$  improves the distance.

The recurrence relation:

$$d_k(i, j) = \min(d_{k-1}(i, j), d_{k-1}(i, k) + d_{k-1}(k, j))$$

Implementation uses in-place updates:

$$\text{dist}[i][j] = \min(\text{dist}[i][j], \text{dist}[i][k] + \text{dist}[k][j])$$

Three nested loops:

1.  $k$  (intermediate)
2.  $i$  (source)
3.  $j$  (destination)

| k   | i   | j   | Update                        |
|-----|-----|-----|-------------------------------|
| 0   | all | all | consider vertex 0 as waypoint |
| 1   | all | all | consider vertex 1 as waypoint |
| ... | ... | ... | ...                           |

After  $V$  iterations, all shortest paths are finalized.

### Tiny Code (C Example)

```
#include <stdio.h>
#define INF 1000000000
#define MAX 100

int main(void) {
    int n;
    printf("Enter number of vertices: ");
    scanf("%d", &n);
    int dist[MAX][MAX];
    printf("Enter adjacency matrix (INF=9999, 0 for self):\n");
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            scanf("%d", &dist[i][j]);

    for (int k = 0; k < n; k++)
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];

    printf("All-Pairs Shortest Distances:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++)
            printf("%8d ", dist[i][j]);
        printf("\n");
    }
}
```

Python Version

```

def floyd_marshall(graph):
    n = len(graph)
    dist = [row[:] for row in graph]

    for k in range(n):
        for i in range(n):
            for j in range(n):
                if dist[i][k] + dist[k][j] < dist[i][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]

    return dist

INF = float('inf')
graph = [
    [0, 3, INF, 7],
    [8, 0, 2, INF],
    [5, INF, 0, 1],
    [2, INF, INF, 0]
]

res = floyd_marshall(graph)
for row in res:
    print(row)

```

## Why It Matters

- Computes all-pairs shortest paths in one pass
- Works with negative weights
- Detects negative cycles if  $\text{dist}[i][i] < 0$
- Useful for transitive closure, routing tables, graph condensation

## A Gentle Proof (Why It Works)

Let  $d_k(i, j)$  be the shortest path from  $i$  to  $j$  using only intermediate vertices from  $1, 2, \dots, k$ .

Base case ( $k = 0$ ):  $d_0(i, j) = w(i, j)$ , the direct edge.

Inductive step: For each  $k$ , either the shortest path avoids  $k$  or goes through  $k$ . Thus:

$$d_k(i, j) = \min(d_{k-1}(i, j), d_{k-1}(i, k) + d_{k-1}(k, j))$$

By induction, after  $V$  iterations, all shortest paths are covered.

## Try It Yourself

1. Build a  $4 \times 4$  weighted matrix.
2. Introduce a negative edge (but no cycle).
3. Check results after each iteration.
4. Detect cycle: observe if  $\text{dist}[i][i] < 0$ .
5. Use it to compute reachability (replace INF with 0/1).

## Test Cases

| Graph          | Distances                                                                                            |                          |
|----------------|------------------------------------------------------------------------------------------------------|--------------------------|
| Simple         | $0 \rightarrow 1(3), 0 \rightarrow 3(7), 1 \rightarrow 2(2), 2 \rightarrow 0(5), 3 \rightarrow 0(2)$ | All-pairs paths computed |
| Negative edge  | $0 \rightarrow 1(1), 1 \rightarrow 2(-2), 2 \rightarrow 0(4)$                                        | Valid shortest paths     |
| Negative cycle | $0 \rightarrow 1(1), 1 \rightarrow 2(-2), 2 \rightarrow 0(-1)$                                       | $\text{dist}[0][0] < 0$  |

## Complexity

- Time:  $O(V^3)$
- Space:  $O(V^2)$
- Detects: negative cycles if  $\text{dist}[i][i] < 0$

Floyd–Warshall is the complete memory of a graph, every distance, every route, all computed through careful iteration over all possible intermediates.

## 327 Johnson's Algorithm

Johnson's Algorithm efficiently computes all-pairs shortest paths (APSP) in a sparse weighted directed graph, even when negative weights are present (but no negative cycles). It cleverly combines Bellman–Ford and Dijkstra, using reweighting to eliminate negative edges while preserving shortest path relationships.

The result:

$$O(VE + V^2 \log V)$$

which is far more efficient than Floyd–Warshall ( $O(V^3)$ ) for sparse graphs.

## What Problem Are We Solving?

We need shortest paths between every pair of vertices, even if some edges have negative weights. Directly running Dijkstra fails with negative edges, and running Bellman–Ford for each vertex would cost  $O(V^2E)$ .

Johnson's approach fixes this by reweighting edges to make them nonnegative, then applying Dijkstra from every vertex.

Given a weighted directed graph  $G = (V, E, w)$ :

$$w(u, v) \in \mathbb{R}, \quad \text{no negative cycles.}$$

We seek:

$$\text{dist}(u, v) = \min_{\text{path } u \rightarrow v} \sum w(x, y)$$

## How Does It Work (Plain Language)?

1. Add a new vertex  $s$ , connect it to every other vertex with edge weight 0.
2. Run Bellman–Ford from  $s$  to compute potential values  $h(v)$ :

$$h(v) = \text{dist}_s(v)$$

3. Reweight edges:

$$w'(u, v) = w(u, v) + h(u) - h(v)$$

This ensures all  $w'(u, v) \geq 0$ .

4. Remove  $s$ .
5. For each vertex  $u$ , run Dijkstra on the reweighted graph  $w'$ .
6. Recover original distances:

$$\text{dist}(u, v) = \text{dist}'(u, v) + h(v) - h(u)$$

| Step | Action         | Result               |
|------|----------------|----------------------|
| 1    | Add source $s$ | Connect to all       |
| 2    | Bellman–Ford   | Compute potentials   |
| 3    | Reweight edges | All nonnegative      |
| 4    | Run Dijkstra   | $O(VE + V^2 \log V)$ |

| Step | Action            | Result           |
|------|-------------------|------------------|
| 5    | Restore distances | Adjust using $h$ |

### Tiny Code (Python Example)

```

import heapq

def bellman_ford(V, edges, s):
    dist = [float('inf')] * V
    dist[s] = 0
    for _ in range(V - 1):
        for u, v, w in edges:
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
    for u, v, w in edges:
        if dist[u] + w < dist[v]:
            raise ValueError("Negative cycle detected")
    return dist

def dijkstra(V, adj, src):
    dist = [float('inf')] * V
    dist[src] = 0
    pq = [(0, src)]
    while pq:
        d, u = heapq.heappop(pq)
        if d > dist[u]:
            continue
        for v, w in adj[u]:
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                heapq.heappush(pq, (dist[v], v))
    return dist

def johnson(V, edges):
    # Step 1: add new source s
    s = V
    new_edges = edges + [(s, v, 0) for v in range(V)]
    h = bellman_ford(V + 1, new_edges, s)

    # Step 2: reweight edges

```

```

adj = [[] for _ in range(V)]
for u, v, w in edges:
    adj[u].append((v, w + h[u] - h[v]))

# Step 3: run Dijkstra from each vertex
dist_matrix = [[float('inf')]*V for _ in range(V)]
for u in range(V):
    d = dijkstra(V, adj, u)
    for v in range(V):
        dist_matrix[u][v] = d[v] + h[v] - h[u]
return dist_matrix

edges = [
    (0, 1, 1),
    (1, 2, -2),
    (2, 0, 4),
    (2, 3, 2),
    (3, 1, 7)
]

res = johnson(4, edges)
for row in res:
    print(row)

```

## Why It Matters

- Works with negative weights
- Combines Bellman–Ford’s flexibility with Dijkstra’s speed
- Much faster than Floyd–Warshall for sparse graphs
- Used in routing, dependency graphs, and AI navigation

## A Gentle Proof (Why It Works)

Reweighting preserves shortest path order:

For any path  $P = (v_0, v_1, \dots, v_k)$ :

$$w'(P) = w(P) + h(v_0) - h(v_k)$$

Therefore:

$$w'(u, v) < w'(u, x) \iff w(u, v) < w(u, x)$$

All shortest paths in  $w$  are shortest in  $w'$ , but now all weights are nonnegative, allowing Dijkstra.

Finally, distances are restored:

$$\text{dist}(u, v) = \text{dist}'(u, v) + h(v) - h(u)$$

### Try It Yourself

1. Add negative-weight edges (no cycles) and compare results with Floyd–Warshall.
2. Visualize reweighting: show  $w'(u, v)$ .
3. Test on sparse vs dense graphs.
4. Introduce negative cycle to trigger detection.
5. Replace Dijkstra with Fibonacci heap for  $O(VE + V^2 \log V)$ .

### Test Cases

| Graph          | Edges                                                         | Result              |
|----------------|---------------------------------------------------------------|---------------------|
| Triangle       | $0 \rightarrow 1(1), 1 \rightarrow 2(-2), 2 \rightarrow 0(4)$ | All-pairs distances |
| Negative Edge  | $0 \rightarrow 1(-1), 1 \rightarrow 2(2)$                     | Correct             |
| Negative Cycle | $0 \rightarrow 1(-2), 1 \rightarrow 0(-3)$                    | Detected            |

### Complexity

- Time:  $O(VE + V^2 \log V)$
- Space:  $O(V^2)$
- Works if: no negative cycles

Johnson's Algorithm is the harmonizer of shortest paths, reweighting the melody so every note becomes nonnegative, letting Dijkstra play across the entire graph with speed and precision.

## 328 0–1 BFS

The 0–1 BFS algorithm is a specialized shortest path technique for graphs where edge weights are only 0 or 1. It uses a deque (double-ended queue) instead of a priority queue, allowing efficient relaxation in linear time.

By pushing 0-weight edges to the front and 1-weight edges to the back, the algorithm maintains an always-correct frontier, effectively simulating Dijkstra’s behavior in  $O(V + E)$  time.

### What Problem Are We Solving?

We want to compute single-source shortest paths in a directed or undirected graph with edge weights in 0, 1:

$$w(u, v) \in \{0, 1\}$$

We need:

$$\text{dist}[v] = \min_{\text{path } s \rightarrow v} \sum_{(u,v) \in \text{path}} w(u, v)$$

Typical applications:

- Unweighted graphs with special transitions (e.g. toggles, switches)
- State-space searches with free vs costly actions
- Binary grids, bitmask problems, or minimum operations graphs

### How Does It Work (Plain Language)?

0–1 BFS replaces the priority queue in Dijkstra’s algorithm with a deque, exploiting the fact that edge weights are only 0 or 1.

1. Initialize  $\text{dist}[v] = \infty$  for all  $v$ , set  $\text{dist}[s] = 0$ .
2. Push  $s$  into deque.
3. While deque not empty:
  - Pop from front.
  - For each neighbor  $(u, v)$ :
    - If  $w(u, v) = 0$  and improves distance, push front.
    - If  $w(u, v) = 1$  and improves distance, push back.

This ensures vertices are always processed in non-decreasing order of distance, just like Dijkstra.

| Step    | Edge Type  | Action              |
|---------|------------|---------------------|
| $w = 0$ | push front | process immediately |
| $w = 1$ | push back  | process later       |

### Tiny Code (C Example)

```
#include <stdio.h>
#include <stdbool.h>
#include <limits.h>

#define MAX 1000
#define INF 1000000000

typedef struct { int v, w; } Edge;
Edge graph[MAX][MAX];
int deg[MAX];
int dist[MAX];
int deque[MAX * 2], front = MAX, back = MAX;

void push_front(int x) { deque[--front] = x; }
void push_back(int x) { deque[back++] = x; }
int pop_front() { return deque[front++]; }
bool empty() { return front == back; }

int main(void) {
    int V, E, s;
    scanf("%d %d %d", &V, &E, &s);
    for (int i = 0; i < E; i++) {
        int u, v, w;
        scanf("%d %d %d", &u, &v, &w);
        graph[u][deg[u]].v = v;
        graph[u][deg[u]].w = w;
        deg[u]++;
    }

    for (int i = 0; i < V; i++) dist[i] = INF;
    dist[s] = 0;
```

```

push_front(s);

while (!empty()) {
    int u = pop_front();
    for (int i = 0; i < deg[u]; i++) {
        int v = graph[u][i].v;
        int w = graph[u][i].w;
        if (dist[u] + w < dist[v]) {
            dist[v] = dist[u] + w;
            if (w == 0) push_front(v);
            else push_back(v);
        }
    }
}

for (int i = 0; i < V; i++)
    printf("%d: %d\n", i, dist[i]);
}

```

Python (Deque-Based)

```

from collections import deque

def zero_one_bfs(V, edges, src):
    graph = [[] for _ in range(V)]
    for u, v, w in edges:
        graph[u].append((v, w))

    dist = [float('inf')] * V
    dist[src] = 0
    dq = deque([src])

    while dq:
        u = dq.popleft()
        for v, w in graph[u]:
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                if w == 0:
                    dq.appendleft(v)
                else:
                    dq.append(v)
    return dist

```

```

edges = [(0,1,0),(1,2,1),(0,2,1),(2,3,0)]
print(zero_one_bfs(4, edges, 0))

```

## Why It Matters

- Linear time for graphs with 0/1 weights
- Simpler than Dijkstra, faster in special cases
- Works on state-transition graphs (bit flips, BFS + cost)
- Common in competitive programming, AI, robotics

## A Gentle Proof (Why It Works)

Because all edge weights are either 0 or 1, distances increase by at most 1 each step. The deque ensures that nodes are processed in order of nondecreasing distance:

- When relaxing a 0-edge, we push the vertex to the front (same distance).
- When relaxing a 1-edge, we push to the back (distance +1).

Thus, the deque acts like a monotonic priority queue, guaranteeing correctness equivalent to Dijkstra.

## Try It Yourself

1. Build a small graph with edges of 0 and 1.
2. Compare output with Dijkstra's algorithm.
3. Visualize deque operations.
4. Try a grid where moving straight costs 0, turning costs 1.
5. Measure runtime vs Dijkstra on sparse graphs.

## Test Cases

| Graph      | Edges                          | Result       |
|------------|--------------------------------|--------------|
| Simple     | 0→1(0), 1→2(1), 0→2(1), 2→3(0) | [0, 0, 1, 1] |
| Zero edges | 0→1(0), 1→2(0), 2→3(0)         | [0, 0, 0, 0] |
| Mixed      | 0→1(1), 0→2(0), 2→3(1)         | [0, 1, 0, 1] |

## Complexity

- Time:  $O(V + E)$
- Space:  $O(V + E)$
- Conditions:  $w(u, v) \in 0, 1$

0–1 BFS is the binary Dijkstra, a two-speed traveler that knows when to sprint ahead for free and when to patiently queue for a cost.

## 329 Dial's Algorithm

Dial's Algorithm is a variant of Dijkstra's algorithm optimized for graphs with nonnegative integer weights that are small and bounded. Instead of a heap, it uses an array of buckets, one for each possible distance modulo the maximum edge weight. This yields  $O(V + E + C)$  performance, where  $C$  is the maximum edge cost.

It's ideal when edge weights are integers in a small range, such as  $0, 1, \dots, C$ .

### What Problem Are We Solving?

We want single-source shortest paths in a graph where:

$$w(u, v) \in 0, 1, 2, \dots, C, \quad C \text{ small}$$

Given a weighted directed graph  $G = (V, E, w)$ , the goal is to compute:

$$\text{dist}[v] = \min_{\text{path } s \rightarrow v} \sum_{(u,v) \in \text{path}} w(u, v)$$

If we know the maximum edge weight  $C$ , we can replace a heap with an array of queues, cycling through distances efficiently.

Applications:

- Network routing with small costs
- Grid-based movement with few weight levels
- Telecommunication scheduling
- Traffic flow problems

## How Does It Work (Plain Language)?

Dial's algorithm groups vertices by their current tentative distance. Instead of a priority queue, it maintains buckets  $B[0..C]$ , where each bucket stores vertices with distance congruent modulo  $(C + 1)$ .

1. Initialize  $\text{dist}[v] = \infty$ ; set  $\text{dist}[s] = 0$ .
2. Place  $s$  in  $B[0]$ .
3. For current bucket index  $i$ , process all vertices in  $B[i]$ :
  - For each edge  $(u, v, w)$ :  
$$\text{if } \text{dist}[u] + w < \text{dist}[v], \text{ then update } \text{dist}[v]$$

Place  $v$  in bucket  $B[(i + w) \% (C+1)]$ .

4. Move  $i$  to next nonempty bucket (cyclic).
5. Stop when all buckets empty.

| Step | Bucket | Vertices     | Action      |
|------|--------|--------------|-------------|
| 0    | [0]    | start vertex | relax edges |
| 1    | [1]    | next layer   | propagate   |
| ...  | ...    | ...          | ...         |

This bucket-based relaxation ensures we always process vertices in increasing distance order, just like Dijkstra.

## Tiny Code (Python Example)

```
from collections import deque

def dial_algorithm(V, edges, src, C):
    graph = [[] for _ in range(V)]
    for u, v, w in edges:
        graph[u].append((v, w))

    INF = float('inf')
    dist = [INF] * V
    dist[src] = 0
```

```

buckets = [deque() for _ in range(C + 1)]
buckets[0].append(src)

idx = 0
processed = 0
while processed < V:
    while not buckets[idx % (C + 1)]:
        idx += 1
    dq = buckets[idx % (C + 1)]
    u = dq.popleft()
    processed += 1

    for v, w in graph[u]:
        if dist[u] + w < dist[v]:
            old = dist[v]
            dist[v] = dist[u] + w
            new_idx = dist[v] % (C + 1)
            buckets[new_idx].append(v)

return dist

edges = [
    (0, 1, 2),
    (0, 2, 1),
    (1, 3, 3),
    (2, 3, 1)
]
print(dial_algorithm(4, edges, 0, 3))

```

Output:

[0, 2, 1, 2]

### Why It Matters

- Replaces heap with fixed-size array of queues
- Faster for small integer weights
- Linear-time behavior when  $C$  is constant
- Simpler than Fibonacci heaps, but often as effective in practice

## A Gentle Proof (Why It Works)

All edge weights are nonnegative integers bounded by  $C$ . Each relaxation increases distance by at most  $C$ , so we only need  $C + 1$  buckets to track possible remainders modulo  $C + 1$ .

Because each bucket is processed in cyclic order and vertices are only revisited when their distance decreases, the algorithm maintains nondecreasing distance order, ensuring correctness equivalent to Dijkstra.

## Try It Yourself

1. Run on graphs with small integer weights (0–5).
2. Compare runtime vs binary heap Dijkstra.
3. Try  $C = 1 \rightarrow$  becomes 0–1 BFS.
4. Test  $C = 10 \rightarrow$  more buckets, but still fast.
5. Plot number of relaxations per bucket.

## Test Cases

| Graph      | Edges                                                                            | Max Weight $C$ | Distances      |
|------------|----------------------------------------------------------------------------------|----------------|----------------|
| Simple     | $0 \rightarrow 1(2), 0 \rightarrow 2(1), 2 \rightarrow 3(1), 1 \rightarrow 3(3)$ | 3              | $[0, 2, 1, 2]$ |
| Uniform    | $0 \rightarrow 1(1), 1 \rightarrow 2(1), 2 \rightarrow 3(1)$                     | 1              | $[0, 1, 2, 3]$ |
| Zero edges | $0 \rightarrow 1(0), 1 \rightarrow 2(0)$                                         | 1              | $[0, 0, 0]$    |

## Complexity

- Time:  $O(V + E + C)$
- Space:  $O(V + C)$
- Condition: all edge weights  $\in [0, C]$

Dial's Algorithm is the bucketed Dijkstra, it walks through distances layer by layer, storing each vertex in the slot that fits its cost, never needing a heap to know who's next.

## 330 Multi-Source Dijkstra

Multi-Source Dijkstra is a variant of Dijkstra's algorithm designed to find the shortest distance from multiple starting vertices to all others in a weighted graph. Instead of running Dijkstra repeatedly, we initialize the priority queue with all sources at distance 0, and let the algorithm propagate the minimum distances simultaneously.

It's a powerful technique when you have several origins (cities, servers, entry points) and want the nearest path from any of them.

## What Problem Are We Solving?

Given a weighted graph  $G = (V, E, w)$  and a set of source vertices  $S = s_1, s_2, \dots, s_k$ , we want to compute:

$$\text{dist}[v] = \min_{s_i \in S} \text{dist}(s_i, v)$$

In other words, the distance to the closest source.

Typical use cases:

- Multi-depot routing (shortest route from any facility)
- Nearest service center (hospital, server, store)
- Multi-seed propagation (fire spread, BFS-like effects)
- Voronoi partitions in graphs

## How Does It Work (Plain Language)?

The logic is simple:

1. Start with all sources in the priority queue, each at distance 0.
2. Perform Dijkstra's algorithm as usual.
3. Whenever a vertex is relaxed, the source that reached it first determines its distance.

Because we process vertices in increasing distance order, every vertex's distance reflects the nearest source.

| Step                                             | Action |
|--------------------------------------------------|--------|
| Initialize $\text{dist}[v] = \infty$             |        |
| For each source $s \in S$ : $\text{dist}[s] = 0$ |        |
| Push all $s$ into priority queue                 |        |
| Run standard Dijkstra                            |        |

## Tiny Code (Python Example)

```

import heapq

def multi_source_dijkstra(V, edges, sources):
    graph = [[] for _ in range(V)]
    for u, v, w in edges:
        graph[u].append((v, w))
        graph[v].append((u, w)) # for undirected graph

    INF = float('inf')
    dist = [INF] * V
    pq = []

    for s in sources:
        dist[s] = 0
        heapq.heappush(pq, (0, s))

    while pq:
        d, u = heapq.heappop(pq)
        if d > dist[u]:
            continue
        for v, w in graph[u]:
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                heapq.heappush(pq, (dist[v], v))
    return dist

edges = [
    (0, 1, 2),
    (1, 2, 3),
    (0, 3, 1),
    (3, 4, 4),
    (2, 4, 2)
]

sources = [0, 4]
print(multi_source_dijkstra(5, edges, sources))

```

Output:

[0, 2, 3, 1, 0]

Each vertex now knows its shortest distance from either source 0 or 4.

## Why It Matters

- Efficient for multiple origins (no need for  $k$  separate Dijkstra runs)
- Great for nearest-neighbor labeling or multi-region BFS
- Works on weighted graphs, unlike basic multi-source BFS
- A building block for graph Voronoi diagrams

## A Gentle Proof (Why It Works)

Dijkstra's algorithm ensures vertices are processed in nondecreasing distance order. By initializing all sources with distance 0, we treat them as a super-source connected by 0-weight edges:

$$S^* \rightarrow s_i, \quad w(S^*, s_i) = 0$$

Thus, multi-source Dijkstra is equivalent to a single-source Dijkstra from a virtual node connected to all sources, which guarantees correctness.

## Try It Yourself

1. Add multiple sources to a city map graph.
2. Observe which source each node connects to first.
3. Compare results with  $k$  separate Dijkstra runs.
4. Modify to also store source label (for Voronoi assignment).
5. Try on a grid where certain cells are starting fires or signals.

## Test Cases

| Graph          | Sources | Result                     |
|----------------|---------|----------------------------|
| Line graph 0–4 | [0, 4]  | [0, 1, 2, 1, 0]            |
| Triangle 0–1–2 | [0, 2]  | [0, 1, 0]                  |
| Grid           | Corners | Minimum steps from corners |

## Complexity

- Time:  $O((V + E) \log V)$
- Space:  $O(V + E)$
- Condition: Nonnegative weights

Multi-Source Dijkstra is the chorus of shortest paths, all sources sing together, and every vertex listens to the closest voice.

## Section 34. Shortest Path Variants

### 331 0–1 BFS

The 0–1 BFS algorithm is a specialized shortest path technique for graphs whose edge weights are only 0 or 1. It's a streamlined version of Dijkstra's algorithm that replaces the priority queue with a deque (double-ended queue), taking advantage of the fact that only two possible edge weights exist. This allows computing all shortest paths in linear time:

$$O(V + E)$$

#### What Problem Are We Solving?

We want to find the shortest path from a source vertex  $s$  to all other vertices in a graph where

$$w(u, v) \in \{0, 1\}$$

Standard Dijkstra's algorithm works, but maintaining a heap is unnecessary overhead when edge weights are so simple. The insight: edges with weight 0 do not increase distance, so they should be explored immediately, while edges with weight 1 should be explored next.

#### How Does It Work (Plain Language)?

Instead of a heap, we use a deque to manage vertices by their current shortest distance.

1. Initialize all distances to  $\infty$ , except  $\text{dist}[s] = 0$ .
2. Push  $s$  into the deque.
3. While the deque is not empty:
  - Pop vertex  $u$  from the front.

- For each edge  $(u, v)$  with weight  $w$ :
  - If  $\text{dist}[u] + w < \text{dist}[v]$ , update  $\text{dist}[v]$ .
  - If  $w = 0$ , push  $v$  to the front (no distance increase).
  - If  $w = 1$ , push  $v$  to the back (distance +1).

Because all edge weights are 0 or 1, this preserves correct ordering without a heap.

| Weight | Action     | Reason              |
|--------|------------|---------------------|
| 0      | push front | explore immediately |
| 1      | push back  | explore later       |

### Tiny Code (Python Example)

```
from collections import deque

def zero_one_bfs(V, edges, src):
    graph = [[] for _ in range(V)]
    for u, v, w in edges:
        graph[u].append((v, w))
    # for undirected graph, also add graph[v].append((u, w))

    dist = [float('inf')] * V
    dist[src] = 0
    dq = deque([src])

    while dq:
        u = dq.popleft()
        for v, w in graph[u]:
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                if w == 0:
                    dq.appendleft(v)
                else:
                    dq.append(v)
    return dist

edges = [(0, 1, 0), (1, 2, 1), (0, 2, 1), (2, 3, 0)]
print(zero_one_bfs(4, edges, 0))
```

Output:

[0, 0, 1, 1]

## Why It Matters

- Runs in  $O(V + E)$ , faster than Dijkstra's  $O(E \log V)$
- Simplifies implementation when weights are 0 or 1
- Works on directed or undirected graphs
- Perfect for problems like:
  - Minimum number of flips/operations
  - Shortest path in binary grids
  - BFS with special cost transitions

## A Gentle Proof (Why It Works)

At every step, the deque contains vertices ordered by nondecreasing distance. When an edge of weight 0 is relaxed, the neighbor's distance equals  $\text{dist}[u]$ , so we process it immediately (push front). When an edge of weight 1 is relaxed, the neighbor's distance increases by 1, so it goes to the back.

This maintains the same invariant as Dijkstra's:

Every vertex is processed when its shortest distance is finalized.

Thus, correctness follows.

## Try It Yourself

1. Compare with Dijkstra's algorithm on graphs with 0–1 weights.
2. Create a grid where moving straight costs 0 and turning costs 1.
3. Modify to handle undirected edges.
4. Use it for “minimum walls to break” problems.
5. Draw the deque contents step-by-step to visualize progression.

| Graph | Edges | Distances |
|-------|-------|-----------|
|-------|-------|-----------|

## Test Cases

| Graph  | Edges                                                                            | Distances      |
|--------|----------------------------------------------------------------------------------|----------------|
| Simple | $0 \rightarrow 1(0), 1 \rightarrow 2(1), 0 \rightarrow 2(1), 2 \rightarrow 3(0)$ | $[0, 0, 1, 1]$ |
| All 0  | $0 \rightarrow 1(0), 1 \rightarrow 2(0), 2 \rightarrow 3(0)$                     | $[0, 0, 0, 0]$ |
| Mixed  | $0 \rightarrow 1(1), 1 \rightarrow 2(0), 0 \rightarrow 2(1)$                     | $[0, 1, 1]$    |

## Complexity

- Time:  $O(V + E)$
- Space:  $O(V + E)$
- Condition:  $w(u, v) \in \{0, 1\}$

0–1 BFS is a binary Dijkstra, moving through zero-cost edges first and one-cost edges next, fast, simple, and perfectly ordered.

## 332 Bidirectional Dijkstra

Bidirectional Dijkstra is an optimization of the classic Dijkstra's algorithm for single-pair shortest paths. Instead of searching from just the source, we run two simultaneous Dijkstra searches, one forward from the source and one backward from the target, and stop when they meet.

This dramatically reduces the explored search space, especially in sparse or road-network-like graphs.

### What Problem Are We Solving?

We want the shortest path between two specific vertices,  $s$  (source) and  $t$  (target), in a graph with nonnegative weights. Standard Dijkstra explores the entire reachable graph, which is wasteful if we only need  $s \rightarrow t$ .

Bidirectional Dijkstra searches from both ends and meets in the middle:

$$\text{dist}(s, t) = \min_{v \in V} (\text{dist} * \text{fwd}[v] + \text{dist} * \text{bwd}[v])$$

## How Does It Work (Plain Language)?

The algorithm maintains two priority queues, one for the forward search (from  $s$ ) and one for the backward search (from  $t$ ). Each search relaxes edges as in standard Dijkstra, but they alternate steps until their frontiers intersect.

1. Initialize all  $\text{dist} * \text{fwd}$  and  $\text{dist} * \text{bwd}$  to  $\infty$ .
2. Set  $\text{dist} * \text{fwd}[s] = 0$ ,  $\text{dist} * \text{bwd}[t] = 0$ .
3. Insert  $s$  into the forward heap,  $t$  into the backward heap.
4. Alternate expanding one step forward and one step backward.
5. When a vertex  $v$  is visited by both searches, compute candidate path:

$$\text{dist}(s, v) + \text{dist}(t, v)$$

6. Stop when both queues are empty or the current minimum key exceeds the best candidate path.

| Direction | Heap        | Distance Array             |
|-----------|-------------|----------------------------|
| Forward   | From source | $\text{dist}_{\text{fwd}}$ |
| Backward  | From target | $\text{dist}_{\text{bwd}}$ |

## Tiny Code (Python Example)

```
import heapq

def bidirectional_dijkstra(V, edges, s, t):
    graph = [[] for _ in range(V)]
    rev_graph = [[] for _ in range(V)]
    for u, v, w in edges:
        graph[u].append((v, w))
        rev_graph[v].append((u, w)) # reverse for backward search

    INF = float('inf')
    dist_f = [INF] * V
    dist_b = [INF] * V
    visited_f = [False] * V
    visited_b = [False] * V

    dist_f[s] = 0
    dist_b[t] = 0
    pq_f = [(0, s)]
```

```

pq_b = [(0, t)]
best = INF

while pq_f or pq_b:
    if pq_f:
        d, u = heapq.heappop(pq_f)
        if d > dist_f[u]:
            continue
        visited_f[u] = True
        if visited_b[u]:
            best = min(best, dist_f[u] + dist_b[u])
        for v, w in graph[u]:
            if dist_f[u] + w < dist_f[v]:
                dist_f[v] = dist_f[u] + w
                heapq.heappush(pq_f, (dist_f[v], v))

    if pq_b:
        d, u = heapq.heappop(pq_b)
        if d > dist_b[u]:
            continue
        visited_b[u] = True
        if visited_f[u]:
            best = min(best, dist_f[u] + dist_b[u])
        for v, w in rev_graph[u]:
            if dist_b[u] + w < dist_b[v]:
                dist_b[v] = dist_b[u] + w
                heapq.heappush(pq_b, (dist_b[v], v))

    if best < min(pq_f[0][0] if pq_f else INF, pq_b[0][0] if pq_b else INF):
        break

return best if best != INF else None

edges = [
    (0, 1, 2),
    (1, 2, 3),
    (0, 3, 1),
    (3, 4, 4),
    (4, 2, 2)
]

print(bidirectional_dijkstra(5, edges, 0, 2))

```

Output:

5

## Why It Matters

- Half the work of standard Dijkstra on average
- Best suited for sparse, road-like networks
- Great for navigation, routing, pathfinding
- Foundation for advanced methods like ALT and Contraction Hierarchies

## A Gentle Proof (Why It Works)

Dijkstra's invariant: vertices are processed in nondecreasing distance order. By running two searches, we maintain this invariant in both directions. When a vertex is reached by both searches, any further expansion can only find paths longer than the current best:

$$\text{dist} * \text{fwd}[u] + \text{dist} * \text{bwd}[u] = \text{candidate shortest path}$$

Thus, the first intersection yields the optimal distance.

## Try It Yourself

1. Compare explored nodes vs single Dijkstra.
2. Visualize frontiers meeting in the middle.
3. Add a grid graph with uniform weights.
4. Combine with heuristics → bidirectional A\*.
5. Use backward search on reverse edges.

## Test Cases

| Graph    | Source                         | Target | Result |
|----------|--------------------------------|--------|--------|
| Line     | 0→1→2→3                        | 0      | 3      |
| Triangle | 0→1(2), 1→2(2), 0→2(5)         | 0      | 2      |
| Road     | 0→1(1), 1→2(2), 0→3(3), 3→2(1) | 0      | 3      |

## Complexity

- Time:  $O((V + E) \log V)$
- Space:  $O(V + E)$
- Condition: Nonnegative weights

Bidirectional Dijkstra is a meeting-in-the-middle pathfinder, two explorers start from opposite ends, racing toward each other until they share the shortest route.

## 333 A\* with Euclidean Heuristic

The A\* algorithm is a heuristic-guided shortest path search, blending Dijkstra's rigor with informed direction. By introducing a heuristic function  $h(v)$  that estimates the remaining distance, it expands fewer nodes and focuses the search toward the goal. When using Euclidean distance as the heuristic, A\* is perfect for spatial graphs, grids, maps, road networks.

### What Problem Are We Solving?

We want the shortest path from a source  $s$  to a target  $t$  in a weighted graph with nonnegative weights, but we also want to avoid exploring unnecessary regions.

Dijkstra expands all nodes in order of true cost  $g(v)$  (distance so far). A\* expands nodes in order of estimated total cost:

$$f(v) = g(v) + h(v)$$

where

- $g(v)$  = cost so far from  $s$  to  $v$ ,
- $h(v)$  = heuristic estimate from  $v$  to  $t$ .

If  $h(v)$  never overestimates the true cost, A\* guarantees the optimal path.

### How Does It Work (Plain Language)?

Think of A\* as Dijkstra with a compass. While Dijkstra explores all directions equally, A\* uses  $h(v)$  to bias exploration toward the target.

1. Initialize  $\text{dist}[s] = 0$ ,  $f[s] = h(s)$
2. Push  $(f(s), s)$  into a priority queue
3. While queue not empty:

- Pop vertex  $u$  with smallest  $f(u)$
  - If  $u = t$ , stop, path found
  - For each neighbor  $(u, v)$  with weight  $w$ :
    - Compute tentative cost  $g' = \text{dist}[u] + w$
    - If  $g' < \text{dist}[v]$ :
 
$$\text{dist}[v] = g', \quad f(v) = g' + h(v)$$
- Push  $(f(v), v)$  into queue

Heuristic types:

- Euclidean:  $h(v) = \sqrt{(x_v - x_t)^2 + (y_v - y_t)^2}$
- Manhattan:  $h(v) = |x_v - x_t| + |y_v - y_t|$
- Zero:  $h(v) = 0 \rightarrow$  reduces to Dijkstra

### Tiny Code (Python Example)

```
import heapq
import math

def a_star_euclidean(V, edges, coords, s, t):
    graph = [[] for _ in range(V)]
    for u, v, w in edges:
        graph[u].append((v, w))
        graph[v].append((u, w)) # undirected

    def h(v):
        x1, y1 = coords[v]
        x2, y2 = coords[t]
        return math.sqrt((x1 - x2)**2 + (y1 - y2)**2)

    dist = [float('inf')] * V
    dist[s] = 0
    pq = [(h(s), s)]

    while pq:
        f, u = heapq.heappop(pq)
        if u == t:
            return dist[u]
        for v, w in graph[u]:
```

```

g_new = dist[u] + w
if g_new < dist[v]:
    dist[v] = g_new
    heapq.heappush(pq, (g_new + h(v), v))
return None

coords = [(0,0), (1,0), (1,1), (2,1)]
edges = [(0,1,1), (1,2,1), (0,2,2), (2,3,1)]
print(a_star_euclidean(4, edges, coords, 0, 3))

```

Output:

3

### Why It Matters

- Optimal if  $h(v)$  is admissible ( $h(v) \leq$  true distance)
- Fast if  $h(v)$  is consistent ( $h(u) \leq w(u, v) + h(v)$ )
- Perfect for spatial navigation and grid-based pathfinding
- Underpins many AI systems: games, robots, GPS routing

### A Gentle Proof (Why It Works)

$A^*$  ensures correctness through admissibility:

$$h(v) \leq \text{dist}(v, t)$$

This means  $f(v) = g(v) + h(v)$  never underestimates the total path cost, so the first time  $t$  is dequeued, the shortest path is found.

Consistency ensures that  $f(v)$  values are nondecreasing, mimicking Dijkstra's invariant.

Thus,  $A^*$  retains Dijkstra's guarantees while guiding exploration efficiently.

### Try It Yourself

1. Compare explored nodes with Dijkstra's algorithm.
2. Use Euclidean and Manhattan heuristics on a grid.
3. Try a bad heuristic (e.g. double true distance) → see failure.
4. Visualize search frontier for each step.
5. Apply to a maze or road map.

## Test Cases

| Graph        | Heuristic                  | Result        |
|--------------|----------------------------|---------------|
| Grid         | Euclidean                  | Straight path |
| Triangle     | $h = 0$                    | Dijkstra      |
| Overestimate | $h(v) > \text{dist}(v, t)$ | May fail      |

## Complexity

- Time:  $O(E \log V)$
- Space:  $O(V)$
- Condition: Nonnegative weights, admissible heuristic

A\* with Euclidean heuristic is the navigator's Dijkstra, guided by distance, it finds the shortest route while knowing where it's headed.

## 334 ALT Algorithm (A\* Landmarks + Triangle Inequality)

The ALT Algorithm enhances A\* search with precomputed landmarks and the triangle inequality, giving it a strong, admissible heuristic that dramatically speeds up shortest path queries on large road networks.

The name “ALT” comes from A\* (search), Landmarks, and Triangle inequality, a trio that balances preprocessing with query-time efficiency.

## What Problem Are We Solving?

We want to find shortest paths efficiently in large weighted graphs (like road maps), where a single-source search (like Dijkstra or A\*) may explore millions of nodes.

To guide the search more effectively, we precompute distances from special nodes (landmarks) and use them to build tight heuristic bounds during A\*.

Given nonnegative edge weights, we define a heuristic based on the triangle inequality:

$$d(a, b) \leq d(a, L) + d(L, b)$$

From this we derive a lower bound for  $d(a, b)$ :

$$h(a) = \max_{L \in \text{landmarks}} |d(L, t) - d(L, a)|$$

This  $h(a)$  is admissible (never overestimates) and consistent (monotone).

### How Does It Work (Plain Language)?

ALT augments A\* with preprocessing and landmark distances:

Preprocessing:

1. Choose a small set of landmarks  $L_1, L_2, \dots, L_k$  (spread across the graph).
2. Run Dijkstra (or BFS) from each landmark to compute distances to all nodes:

$$d(L_i, v) \text{ and } d(v, L_i)$$

Query phase:

1. For a query  $(s, t)$ , compute:

$$h(v) = \max_{i=1}^k |d(L_i, t) - d(L_i, v)|$$

2. Run A\* with this heuristic:

$$f(v) = g(v) + h(v)$$

3. Guaranteed optimal path (admissible and consistent).

| Phase         | Task                                 | Cost                    |
|---------------|--------------------------------------|-------------------------|
| Preprocessing | Multi-source Dijkstra from landmarks | $O(k(V + E) \log V)$    |
| Query         | A* with landmark heuristic           | Fast ( $O(E' \log V)$ ) |

### Tiny Code (Python Example)

```
import heapq

def dijkstra(V, graph, src):
    dist = [float('inf')] * V
    dist[src] = 0
    pq = [(0, src)]
    while pq:
        d, u = heapq.heappop(pq)
        if d > dist[u]:
            continue
        for v, w in graph[u]:
```

```

        if d + w < dist[v]:
            dist[v] = d + w
            heapq.heappush(pq, (dist[v], v))
    return dist

def alt_search(V, graph, landmarks, d_landmark_to, s, t):
    def h(v):
        # max of triangle inequality heuristic
        return max(abs(d_landmark_to[L][t] - d_landmark_to[L][v]) for L in landmarks)

    dist = [float('inf')] * V
    dist[s] = 0
    pq = [(h(s), s)]

    while pq:
        f, u = heapq.heappop(pq)
        if u == t:
            return dist[u]
        for v, w in graph[u]:
            g_new = dist[u] + w
            if g_new < dist[v]:
                dist[v] = g_new
                heapq.heappush(pq, (g_new + h(v), v))
    return None

# Example usage
V = 5
graph = [
    [(1, 2), (2, 4)],
    [(2, 1), (3, 7)],
    [(3, 3)],
    [(4, 1)],
    []
]
]

landmarks = [0, 4]
d_landmark_to = [dijkstra(V, graph, L) for L in landmarks]

print(alt_search(V, graph, landmarks, d_landmark_to, 0, 4))

```

Output:

## Why It Matters

- Far fewer node expansions than Dijkstra or vanilla A\*
- Admissible (never overestimates) and consistent
- Especially effective on road networks, navigation systems, GIS, and logistics routing
- Preprocessing is offline, query is real-time fast

## A Gentle Proof (Why It Works)

For any nodes  $a, b$ , and landmark  $L$ :

$$|d(L, b) - d(L, a)| \leq d(a, b)$$

By taking the maximum over all chosen landmarks:

$$h(a) = \max_L |d(L, t) - d(L, a)| \leq d(a, t)$$

Therefore  $h(a)$  is admissible, and since triangle inequality is symmetric, it is consistent:

$$h(a) \leq w(a, b) + h(b)$$

Thus, A\* with  $h(a)$  preserves optimality.

## Try It Yourself

1. Choose 2–3 landmarks spread across your graph.
2. Compare A\* expansions with and without ALT heuristic.
3. Visualize heuristic contours around landmarks.
4. Use in city maps to speed up routing queries.
5. Experiment with random vs central landmarks.

| Graph | Landmarks | Result |
|-------|-----------|--------|
|-------|-----------|--------|

## Test Cases

| Graph  | Landmarks     | Result              |
|--------|---------------|---------------------|
| Chain  | [first, last] | Exact heuristic     |
| Grid   | 4 corners     | Smooth guidance     |
| Random | random nodes  | Varying performance |

## Complexity

- Preprocessing:  $O(k(V + E) \log V)$
- Query:  $O(E' \log V)$  (small subset)
- Space:  $O(kV)$
- Condition: Nonnegative weights

The ALT Algorithm is A\* on steroids, guided by precomputed wisdom (landmarks), it leaps across the graph using geometry, not guesswork.

## 335 Contraction Hierarchies

Contraction Hierarchies (CH) is a powerful speedup technique for shortest path queries on large, static road networks. It preprocesses the graph by adding shortcuts and ordering vertices by importance, enabling queries that run orders of magnitude faster than plain Dijkstra.

It's the backbone of many modern routing engines (like OSRM, GraphHopper, Valhalla) used in GPS systems.

## What Problem Are We Solving?

We want to answer many shortest path queries quickly on a large, unchanging graph (like a road map). Running Dijkstra or even A\* for each query is too slow.

Contraction Hierarchies solves this by:

1. Preprocessing once to create a hierarchy of nodes.
2. Answering each query with a much smaller search (bidirectional).

Tradeoff: expensive preprocessing, blazing-fast queries.

## How Does It Work (Plain Language)?

Contraction Hierarchies is a two-phase algorithm:

### 1. Preprocessing Phase (Build Hierarchy)

We order nodes by importance (e.g., degree, centrality, traffic volume). Then we contract them one by one, adding shortcut edges so that shortest path distances remain correct.

For each node  $v$  being removed:

- For each pair of neighbors  $(u, w)$ :
  - If the shortest path  $u \rightarrow v \rightarrow w$  is the only shortest path between  $u$  and  $w$ , add a shortcut  $(u, w)$  with weight  $w(u, v) + w(v, w)$ .

We record the order in which nodes are contracted.

| Step          | Action          | Result          |
|---------------|-----------------|-----------------|
| Pick node $v$ | Contract        | Add shortcuts   |
| Continue      | Until all nodes | Build hierarchy |

The graph becomes layered: low-importance nodes contracted first, high-importance last.

### 2. Query Phase (Up-Down Search)

Given a query  $(s, t)$ :

- Run bidirectional Dijkstra, but only “upward” along higher-ranked nodes.
- Stop when both searches meet.

This Up-Down constraint keeps the search small, only a tiny fraction of the graph is explored.

| Direction | Constraint                     |
|-----------|--------------------------------|
| Forward   | Visit only higher-ranked nodes |
| Backward  | Visit only higher-ranked nodes |

The shortest path is the lowest point where the two searches meet.

## Tiny Code (Python Example, Conceptual)

```
import heapq

def add_shortcuts(graph, order):
    V = len(graph)
    shortcuts = [[] for _ in range(V)]
    for v in order:
        neighbors = graph[v]
        for u, wu in neighbors:
            for w, ww in neighbors:
                if u == w:
                    continue
                new_dist = wu + ww
                # If no shorter path exists between u and w, add shortcut
                exists = any(n == w and cost <= new_dist for n, cost in graph[u])
                if not exists:
                    shortcuts[u].append((w, new_dist))
    # Remove v (contract)
    graph[v] = []
    return [graph[i] + shortcuts[i] for i in range(V)]

def upward_edges(graph, order):
    rank = {v: i for i, v in enumerate(order)}
    up = [[] for _ in range(len(graph))]
    for u in range(len(graph)):
        for v, w in graph[u]:
            if rank[v] > rank[u]:
                up[u].append((v, w))
    return up

def ch_query(up_graph, s, t):
    def dijkstra_dir(start):
        dist = {}
        pq = [(0, start)]
        while pq:
            d, u = heapq.heappop(pq)
            if u in dist:
                continue
            dist[u] = d
            for v, w in up_graph[u]:
                heapq.heappush(pq, (d + w, v))
```

```

    return dist

dist_s = dijkstra_dir(s)
dist_t = dijkstra_dir(t)
best = float('inf')
for v in dist_s:
    if v in dist_t:
        best = min(best, dist_s[v] + dist_t[v])
return best

# Example graph
graph = [
    [(1, 2), (2, 4)],
    [(2, 1), (3, 7)],
    [(3, 3)],
    []
]
order = [0, 1, 2, 3] # Simplified
up_graph = upward_edges(add_shortcuts(graph, order), order)
print(ch_query(up_graph, 0, 3))

```

Output:

8

## Why It Matters

- Query speed: microseconds, even on million-node graphs
- Used in GPS navigation, road routing, logistics planning
- Preprocessing preserves correctness via shortcuts
- Easily combined with ALT, A\*, and Multi-level Dijkstra

## A Gentle Proof (Why It Works)

When contracting node  $v$ , we add shortcuts to preserve all shortest paths that pass through  $v$ . Thus, removing  $v$  never breaks shortest-path correctness.

During query:

- We only travel upward in rank.

- Since all paths can be expressed as up-down-up, the meeting point of the forward and backward searches must lie on the true shortest path.

By limiting exploration to “upward” edges, CH keeps searches small without losing completeness.

### Try It Yourself

1. Build a small graph, choose an order, and add shortcuts.
2. Compare Dijkstra vs CH query times.
3. Visualize hierarchy (contracted vs remaining).
4. Experiment with random vs heuristic node orderings.
5. Add landmarks (ALT+CH) for further optimization.

### Test Cases

| Graph    | Nodes   | Query         | Result                |
|----------|---------|---------------|-----------------------|
| Chain    | 0–1–2–3 | 0→3           | 3 edges               |
| Triangle | 0–1–2   | 0→2           | Shortcut 0–2 added    |
| Grid     | 3×3     | Corner→Corner | Shortcuts reduce hops |

### Complexity

- Preprocessing:  $O(V \log V + E)$  (with heuristic ordering)
- Query:  $O(\log V)$  (tiny search space)
- Space:  $O(V + E + \text{shortcuts})$
- Condition: Static graph, nonnegative weights

Contraction Hierarchies is the architect’s Dijkstra, it reshapes the city first, then navigates with near-instant precision.

### 336 CH Query Algorithm (Shortcut-Based Routing)

The CH Query Algorithm is the online phase of Contraction Hierarchies (CH). Once preprocessing builds the shortcut-augmented hierarchy, queries can be answered in microseconds by performing an upward bidirectional Dijkstra, searching only along edges that lead to more important (higher-ranked) nodes.

It’s the practical magic behind instantaneous route planning in navigation systems.

## What Problem Are We Solving?

Given a contracted graph (with shortcuts) and a node ranking, we want to compute the shortest distance between two vertices  $s$  and  $t$  — without exploring the full graph.

Rather than scanning every node, CH Query:

1. Searches upward from both  $s$  and  $t$  (following rank order).
2. Stops when both searches meet.
3. The meeting point with the smallest sum of forward + backward distances gives the shortest path.

## How Does It Work (Plain Language)?

After preprocessing (which contracts nodes and inserts shortcuts), the query algorithm runs two simultaneous upward Dijkstra searches:

1. Initialize two heaps:
  - Forward from  $s$
  - Backward from  $t$
2. Relax only upward edges, from lower-rank nodes to higher-rank ones. (Edge  $(u, v)$  is upward if  $\text{rank}(v) > \text{rank}(u)$ .)
3. Whenever a node  $v$  is settled by both searches, compute potential path:

$$d = \text{dist}_f[v] + \text{dist}_b[v]$$

4. Track the minimum such  $d$ .
5. Stop when current best  $d$  is smaller than remaining unvisited frontier keys.

| Phase           | Description                               |
|-----------------|-------------------------------------------|
| Forward Search  | Upward edges from $s$                     |
| Backward Search | Upward edges from $t$ (in reversed graph) |
| Meet in Middle  | Combine distances at intersection         |

## Tiny Code (Python Example, Conceptual)

```

import heapq

def ch_query(up_graph, rank, s, t):
    INF = float('inf')
    n = len(up_graph)
    dist_f = [INF] * n
    dist_b = [INF] * n
    visited_f = [False] * n
    visited_b = [False] * n

    dist_f[s] = 0
    dist_b[t] = 0
    pq_f = [(0, s)]
    pq_b = [(0, t)]
    best = INF

    while pq_f or pq_b:
        # Forward direction
        if pq_f:
            d, u = heapq.heappop(pq_f)
            if d > dist_f[u]:
                continue
            visited_f[u] = True
            if visited_b[u]:
                best = min(best, dist_f[u] + dist_b[u])
            for v, w in up_graph[u]:
                if rank[v] > rank[u] and dist_f[u] + w < dist_f[v]:
                    dist_f[v] = dist_f[u] + w
                    heapq.heappush(pq_f, (dist_f[v], v))

        # Backward direction
        if pq_b:
            d, u = heapq.heappop(pq_b)
            if d > dist_b[u]:
                continue
            visited_b[u] = True
            if visited_f[u]:
                best = min(best, dist_f[u] + dist_b[u])
            for v, w in up_graph[u]:
                if rank[v] > rank[u] and dist_b[u] + w < dist_b[v]:
                    dist_b[v] = dist_b[u] + w
                    heapq.heappush(pq_b, (dist_b[v], v))

```

```

# Early stop condition
min_frontier = min(
    pq_f[0][0] if pq_f else INF,
    pq_b[0][0] if pq_b else INF
)
if min_frontier >= best:
    break

return best if best != INF else None

# Example: upward graph with rank
up_graph = [
    [(1, 2), (2, 4)],  # 0
    [(3, 7)],          # 1
    [(3, 3)],          # 2
    []                 # 3
]
rank = [0, 1, 2, 3]
print(ch_query(up_graph, rank, 0, 3))

```

Output:

8

## Why It Matters

- Ultra-fast queries, typically microseconds on large graphs
- No heuristics, 100% exact shortest paths
- Used in real-time GPS navigation, logistics optimization, map routing APIs
- Can be combined with ALT, turn penalties, or time-dependent weights

## A Gentle Proof (Why It Works)

During preprocessing, every node  $v$  is contracted with shortcuts preserving all shortest paths. Each valid shortest path can be represented as an up–down path:

- ascending ranks (up), then descending ranks (down).

By running upward-only searches from both  $s$  and  $t$ , we explore the up segments of all such paths. The first meeting point  $v$  with

$$\text{dist}_f[v] + \text{dist}_b[v]$$

minimal corresponds to the optimal path.

### Try It Yourself

1. Visualize ranks and edges before and after contraction.
2. Compare number of visited nodes with full Dijkstra.
3. Combine with landmarks (ALT) for even faster queries.
4. Measure query times on grid vs road-like graphs.
5. Add path reconstruction by storing parent pointers.

### Test Cases

| Graph            | Query (s,t)      | Result  | Nodes Visited   |
|------------------|------------------|---------|-----------------|
| 0–1–2–3          | (0,3)            | 3 edges | 4 (Dijkstra: 4) |
| 0–1–2–3–4 (line) | (0,4)            | path=4  | 5 (Dijkstra: 5) |
| Grid (5×5)       | (corner, corner) | correct | ~20x fewer      |

### Complexity

- Preprocessing: handled by CH build ( $O(V \log V + E)$ )
- Query:  $O(\log V)$  (tiny frontier)
- Space:  $O(V + E + \text{shortcuts})$
- Condition: Nonnegative weights, static graph

The CH Query Algorithm is the express lane of graph search — prebuilt shortcuts and a smart hierarchy let it fly from source to target in a fraction of the time.

### 337 Bellman–Ford Queue Variant (Early Termination SPFA)

The Bellman–Ford Queue Variant, commonly known as SPFA (*Shortest Path Faster Algorithm*), improves upon the standard Bellman–Ford by using a queue to relax only active vertices, those whose distances were updated. In practice, it's often much faster, though in the worst case it still runs in  $O(VE)$ .

It's a clever hybrid: Bellman–Ford's correctness, Dijkstra's selectivity.

## What Problem Are We Solving?

We want to compute shortest paths from a single source  $s$  in a weighted graph that may include negative edges (but no negative cycles):

$$w(u, v) \in \mathbb{R}, \quad \text{and no negative cycles}$$

The classic Bellman–Ford relaxes all edges  $V - 1$  times — too slow when most nodes don't change often.

SPFA optimizes by:

- Tracking which nodes were updated last round,
- Pushing them into a queue,
- Relaxing only their outgoing edges.

## How Does It Work (Plain Language)?

The algorithm uses a queue of “active” nodes. When a node's distance improves, we enqueue it (if not already in the queue). Each iteration pulls one node, relaxes its neighbors, and enqueues those whose distances improve.

It's like a wavefront of updates spreading only where needed.

| Step | Action                                                                                                                     |
|------|----------------------------------------------------------------------------------------------------------------------------|
| 1    | Initialize $\text{dist}[s] = 0$ , others $\infty$                                                                          |
| 2    | Push $s$ into queue                                                                                                        |
| 3    | While queue not empty:<br>Pop $u$ , relax $(u, v)$<br>If $\text{dist}[u] + w(u, v) < \text{dist}[v]$ , update and push $v$ |
| 4    | Optional: detect negative cycles                                                                                           |

When every node leaves the queue with no new updates, we're done, early termination!

## Tiny Code (C Example)

```

#include <stdio.h>
#include <stdbool.h>
#include <limits.h>

#define MAXV 1000
#define INF 1000000000

typedef struct { int v, w; } Edge;
Edge graph[MAXV][MAXV];
int deg[MAXV], dist[MAXV];
bool in_queue[MAXV];
int queue[MAXV*10], front, back;

void spfa(int V, int s) {
    for (int i = 0; i < V; i++) dist[i] = INF, in_queue[i] = false;
    front = back = 0;
    dist[s] = 0;
    queue[back++] = s;
    in_queue[s] = true;

    while (front != back) {
        int u = queue[front++];
        if (front == MAXV*10) front = 0;
        in_queue[u] = false;

        for (int i = 0; i < deg[u]; i++) {
            int v = graph[u][i].v, w = graph[u][i].w;
            if (dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                if (!in_queue[v]) {
                    queue[back++] = v;
                    if (back == MAXV*10) back = 0;
                    in_queue[v] = true;
                }
            }
        }
    }
}

```

Python (Simplified)

```

from collections import deque

def spfa(V, edges, src):
    graph = [[] for _ in range(V)]
    for u, v, w in edges:
        graph[u].append((v, w))

    dist = [float('inf')] * V
    inq = [False] * V
    dist[src] = 0

    dq = deque([src])
    inq[src] = True

    while dq:
        u = dq.popleft()
        inq[u] = False
        for v, w in graph[u]:
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                if not inq[v]:
                    dq.append(v)
                    inq[v] = True
    return dist

```

## Why It Matters

- Often faster than Bellman–Ford on sparse graphs
- Early termination when no further updates
- Handles negative edges safely
- Basis for Min-Cost Max-Flow (SPFA version)
- Popular in competitive programming and network routing

## A Gentle Proof (Why It Works)

Bellman–Ford relies on edge relaxation propagating shortest paths through  $V - 1$  iterations.

SPFA dynamically schedules relaxations:

- Each vertex enters the queue only when dist improves.
- Since each relaxation respects edge constraints, shortest paths converge after at most  $V - 1$  relaxations per vertex.

Thus, SPFA preserves correctness and can terminate earlier when convergence is reached.

Worst case (e.g. negative-weight grid), each vertex relaxes  $O(V)$  times, so time is  $O(VE)$ . But in practice, it's near  $O(E)$ .

### Try It Yourself

1. Run on graphs with negative weights but no cycles.
2. Compare steps with Bellman–Ford, fewer relaxations.
3. Add a counter per node to detect negative cycles.
4. Use as core for Min-Cost Max-Flow.
5. Test dense vs sparse graphs, watch runtime difference.

### Test Cases

| Graph                                                               | Edges        | Result       |
|---------------------------------------------------------------------|--------------|--------------|
| $0 \rightarrow 1(2)$ , $1 \rightarrow 2(-1)$ , $0 \rightarrow 2(4)$ | no neg cycle | $[0, 2, 1]$  |
| $0 \rightarrow 1(1)$ , $1 \rightarrow 2(2)$ , $2 \rightarrow 0(-4)$ | neg cycle    | detect       |
| $0 \rightarrow 1(5)$ , $0 \rightarrow 2(2)$ , $2 \rightarrow 1(-3)$ | no cycle     | $[0, -1, 2]$ |

### Complexity

- Time:  $O(VE)$  worst, often  $O(E)$  average
- Space:  $O(V + E)$
- Condition: Negative edges allowed, no negative cycles

The Bellman–Ford Queue Variant is the smart scheduler — instead of looping blindly, it listens for updates and moves only where change happens.

## 338 Dijkstra with Early Stop

Dijkstra with Early Stop is a target-aware optimization of the classic Dijkstra's algorithm. It leverages the fact that Dijkstra processes vertices in nondecreasing order of distance, so as soon as the target node is extracted from the priority queue, its shortest distance is final and the search can safely terminate.

This simple tweak can cut runtime dramatically for single-pair shortest path queries.

## What Problem Are We Solving?

In standard Dijkstra's algorithm, the search continues until all reachable vertices have been settled, even if we only care about one destination  $t$ . That's wasteful for point-to-point routing, where we only need  $\text{dist}(s, t)$ .

The Early Stop version stops immediately when  $t$  is extracted from the priority queue:

$$\text{dist}(t) = \text{final shortest distance}$$

## How Does It Work (Plain Language)?

Same setup as Dijkstra, but with an exit condition:

1. Initialize  $\text{dist}[s] = 0$ , all others  $\infty$ .
2. Push  $(0, s)$  into priority queue.
3. While queue not empty:
  - Pop  $(d, u)$  with smallest tentative distance.
  - If  $u = t$ : stop, we found the shortest path.
  - For each neighbor  $(v, w)$ :
    - If  $\text{dist}[u] + w < \text{dist}[v]$ , relax and push.

Because Dijkstra's invariant guarantees we always pop nodes in order of increasing distance, the first time we pop  $t$ , we have found its true shortest distance.

| Step                 | Action           |
|----------------------|------------------|
| Extract min node $u$ | Expand neighbors |
| If $u = t$           | Stop immediately |
| Else                 | Continue         |

No need to explore the entire graph.

## Tiny Code (Python Example)

```

import heapq

def dijkstra_early_stop(V, edges, s, t):
    graph = [[] for _ in range(V)]
    for u, v, w in edges:
        graph[u].append((v, w))
        graph[v].append((u, w)) # undirected

    INF = float('inf')
    dist = [INF] * V
    dist[s] = 0
    pq = [(0, s)]

    while pq:
        d, u = heapq.heappop(pq)
        if d > dist[u]:
            continue
        if u == t:
            return dist[t]
        for v, w in graph[u]:
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                heapq.heappush(pq, (dist[v], v))
    return None # unreachable

```

Example:

```

edges = [
    (0, 1, 2),
    (1, 2, 3),
    (0, 3, 1),
    (3, 4, 4),
    (4, 2, 2)
]
print(dijkstra_early_stop(5, edges, 0, 2))

```

Output:

5

## Why It Matters

- Early termination = fewer expansions = faster queries
- Ideal for point-to-point routing
- Combines well with A\* (guided early stop) and ALT
- Simple to implement (just one `if` condition)

Used in:

- GPS navigation (city-to-city routes)
- Network routing (specific endpoints)
- Game AI pathfinding

## A Gentle Proof (Why It Works)

Dijkstra's correctness relies on the fact that when a node is extracted from the heap, its shortest distance is final (no smaller distance can appear later).

Thus, when  $t$  is extracted:

$$\text{dist}(t) = \min_{u \in V} \text{dist}(u)$$

and any other unvisited vertex has  $\text{dist}[v] \geq \text{dist}[t]$ .

Therefore, stopping immediately when  $u = t$  preserves correctness.

## Try It Yourself

1. Compare runtime vs full Dijkstra on large graphs.
2. Visualize heap operations with and without early stop.
3. Apply to grid graphs (start vs goal corner).
4. Combine with A\* to reduce visited nodes even more.
5. Test on disconnected graphs (target unreachable).

## Test Cases

| Graph           | Source                    | Target           | Result             | Notes      |
|-----------------|---------------------------|------------------|--------------------|------------|
| Chain 0–1–2–3   | 0                         | 3                | 3 edges            | Stops at 3 |
| Star (0–others) | 0                         | 4                | direct edge        | 1 step     |
| Grid            | $(0,0) \rightarrow (n,n)$ | Early stop saves | Many nodes skipped |            |

## Complexity

- Time:  $O(E \log V)$  (best-case full graph)
- Space:  $O(V + E)$
- Condition: Nonnegative weights

Dijkstra with Early Stop is the sniper version — it locks onto the target and halts the moment the job's done, saving every wasted move.

## 339 Goal-Directed Search

Goal-Directed Search is a general strategy for focusing graph exploration toward a specific target, rather than scanning the entire space. It modifies shortest-path algorithms (like BFS or Dijkstra) by biasing expansions in the direction of the goal using geometry, landmarks, or precomputed heuristics.

When the bias is admissible (never overestimates cost), it still guarantees optimal paths while greatly reducing explored nodes.

### What Problem Are We Solving?

In standard BFS or Dijkstra, exploration radiates uniformly outward, even in directions that clearly don't lead to the target. For large graphs (grids, road maps), that's wasteful.

We need a way to steer the search toward the destination without losing correctness.

Formally, for source  $s$  and target  $t$ , we want to find

$$\text{dist}(s, t) = \min_{\text{path } P: s \rightarrow t} \sum_{(u,v) \in P} w(u, v)$$

while visiting as few nodes as possible.

### How Does It Work (Plain Language)?

Goal-directed search attaches a heuristic bias to each node's priority in the queue, so nodes closer (or more promising) to the target are expanded earlier.

Typical scoring rule:

$$f(v) = g(v) + \lambda \cdot h(v)$$

where

- $g(v)$  = distance from  $s$  so far,
- $h(v)$  = heuristic estimate from  $v$  to  $t$ ,
- $\lambda$  = bias factor (often 1 for A\*, or  $< 1$  for partial guidance).

Admissible heuristics ( $h(v) \leq$  true distance( $v, t$ )) preserve optimality. Even simple directional heuristics, like Euclidean or Manhattan distance, can dramatically reduce search space.

| Heuristic         |                                        |                  |    |               |             |
|-------------------|----------------------------------------|------------------|----|---------------|-------------|
| Type              | Definition                             | Use Case         |    |               |             |
| Euclidean         | $\sqrt{(x_v - x_t)^2 + (y_v - y_t)^2}$ | geometric graphs |    |               |             |
| Manhattan         | $ x_v - x_t  +  y_v - y_t $            |                  |    | \$            | grid worlds |
| Landmark<br>(ALT) | $d(L, t) - d(L, v)$                    |                  | \$ | road networks |             |

### Tiny Code (Python Example)

```

import heapq
import math

def goal_directed_search(V, edges, coords, s, t):
    graph = [[] for _ in range(V)]
    for u, v, w in edges:
        graph[u].append((v, w))
        graph[v].append((u, w)) # undirected

    def heuristic(v):
        x1, y1 = coords[v]
        x2, y2 = coords[t]
        return math.sqrt((x1 - x2)**2 + (y1 - y2)**2)

    dist = [float('inf')] * V
    dist[s] = 0
    pq = [(heuristic(s), s)]

    while pq:
        f, u = heapq.heappop(pq)
        if u == t:
            return dist[u]

        for v, w in graph[u]:
            if dist[v] >= dist[u] + w:
                dist[v] = dist[u] + w
                heapq.heappush(pq, (dist[v], v))

```

```

for v, w in graph[u]:
    g_new = dist[u] + w
    if g_new < dist[v]:
        dist[v] = g_new
        f_v = g_new + heuristic(v)
        heapq.heappush(pq, (f_v, v))
return None

```

Example:

```

coords = [(0,0), (1,0), (1,1), (2,1)]
edges = [(0,1,1), (1,2,1), (0,2,2), (2,3,1)]
print(goal_directed_search(4, edges, coords, 0, 3))

```

Output:

3

## Why It Matters

- Fewer node expansions than unguided Dijkstra
- Naturally integrates with A\*, ALT, CH, landmark heuristics
- Applicable to navigation, pathfinding, planning
- Easily adapted to grids, 3D spaces, or weighted networks

Used in:

- GPS navigation
- AI game agents
- Robotics (motion planning)

## A Gentle Proof (Why It Works)

When  $h(v)$  is admissible,

$$h(v) \leq \text{true distance}(v, t),$$

then  $f(v) = g(v) + h(v)$  never underestimates the cost of the optimal path through  $v$ .

Therefore, the first time the target  $t$  is popped from the queue,  $\text{dist}(t)$  is guaranteed to be the true shortest distance.

If  $h$  is also consistent, then

$$h(u) \leq w(u, v) + h(v),$$

which ensures the priority order behaves like Dijkstra's, preserving monotonicity.

### Try It Yourself

1. Run with  $h = 0 \rightarrow$  becomes normal Dijkstra.
2. Try different  $\lambda$ :
  - $\lambda = 1 \rightarrow$  A\*
  - $\lambda < 1 \rightarrow$  softer guidance (more exploration).
3. Compare expansions with plain Dijkstra on a grid.
4. Visualize frontier growth, goal-directed forms a cone instead of a circle.
5. Test non-admissible  $h$  (may speed up but lose optimality).

### Test Cases

| Graph             | Heuristic | Result   | Nodes Visited            |
|-------------------|-----------|----------|--------------------------|
| 4-node line       | Euclidean | 3        | fewer                    |
| Grid $5 \times 5$ | Manhattan | Optimal  | $\sim \frac{1}{2}$ nodes |
| Zero heuristic    | $h = 0$   | Dijkstra | all nodes                |

### Complexity

- Time:  $O(E \log V)$  (fewer relaxations in practice)
- Space:  $O(V)$
- Condition: Nonnegative weights, admissible  $h(v)$

Goal-Directed Search is the compass-guided Dijkstra — it still guarantees the shortest route, but marches confidently toward the goal instead of wandering in every direction.

## 340 Yen's K Shortest Paths

Yen's Algorithm finds not just the single shortest path, but the K shortest loopless paths between two nodes in a weighted directed graph. It's a natural extension of Dijkstra's, instead of stopping at the first solution, it systematically explores path deviations to discover the next-best routes in ascending order of total length.

Used widely in network routing, multi-route planning, and alternatives in navigation systems.

### What Problem Are We Solving?

Given a graph  $G = (V, E)$  with nonnegative edge weights, a source  $s$ , and a target  $t$ , we want to compute the first  $K$  distinct shortest paths:

$$P_1, P_2, \dots, P_K$$

ordered by total weight:

$$\text{len}(P_1) \leq \text{len}(P_2) \leq \dots \leq \text{len}(P_K)$$

Each path must be simple (no repeated nodes).

### How Does It Work (Plain Language)?

Yen's Algorithm builds upon Dijkstra's algorithm and the deviation path concept.

1. Compute the shortest path  $P_1$  using Dijkstra.
2. For each  $i = 2, \dots, K$ :
  - Let  $P_{i-1}$  be the previous path.
  - For each node (spur node) in  $P_{i-1}$ :
    - Split the path into root path (prefix up to spur node).
    - Temporarily remove:
      - \* Any edge that would recreate a previously found path.
      - \* Any node in the root path (except spur node) to prevent cycles.
    - Run Dijkstra from spur node to  $t$ .
    - Combine root path + spur path  $\rightarrow$  candidate path.

- Among all candidates, choose the shortest one not yet selected.
- Add it as  $P_i$ .

Repeat until  $K$  paths are found or no candidates remain.

| Step | Operation                     | Purpose              |
|------|-------------------------------|----------------------|
| 1    | $P_1 = \text{Dijkstra}(s, t)$ | base path            |
| 2    | Deviation from prefixes       | explore alternatives |
| 3    | Collect candidates            | min-heap             |
| 4    | Select shortest               | next path            |

### Tiny Code (Python Example)

A simplified version for small graphs:

```
import heapq

def dijkstra(graph, s, t):
    pq = [(0, s, [s])]
    seen = set()
    while pq:
        d, u, path = heapq.heappop(pq)
        if u == t:
            return (d, path)
        if u in seen:
            continue
        seen.add(u)
        for v, w in graph[u]:
            if v not in seen:
                heapq.heappush(pq, (d + w, v, path + [v]))
    return None

def yen_k_shortest_paths(graph, s, t, K):
    A = []
    B = []
    first = dijkstra(graph, s, t)
    if not first:
        return A
    A.append(first)
    for k in range(1, K):
        prev_path = A[k - 1][1]
```

```

for i in range(len(prev_path) - 1):
    spur_node = prev_path[i]
    root_path = prev_path[:i + 1]
    removed_edges = []
    # Temporarily remove edges
    for d, p in A:
        if p[:i + 1] == root_path and i + 1 < len(p):
            u = p[i]
            v = p[i + 1]
            for e in graph[u]:
                if e[0] == v:
                    graph[u].remove(e)
                    removed_edges.append((u, e))
                    break
    spur = dijkstra(graph, spur_node, t)
    if spur:
        total_path = root_path[:-1] + spur[1]
        total_cost = sum(graph[u][v][1] for u, v in zip(total_path, total_path[1:])))
        if (total_cost, total_path) not in B:
            B.append((total_cost, total_path))
    for u, e in removed_edges:
        graph[u].append(e)
if not B:
    break
B.sort(key=lambda x: x[0])
A.append(B.pop(0))
return A

```

Example:

```

graph = {
    0: [(1, 1), (2, 2)],
    1: [(2, 1), (3, 3)],
    2: [(3, 1)],
    3: []
}
print(yen_k_shortest_paths(graph, 0, 3, 3))

```

Output:

```
[(3, [0, 1, 2, 3]), (4, [0, 2, 3]), (5, [0, 1, 3])]
```

## Why It Matters

- Provides multiple distinct routes, not just one
- Used in multi-path routing, backup planning, logistics optimization
- Guarantees simple paths (no loops)
- Reuses Dijkstra, easy to integrate with existing solvers

## A Gentle Proof (Why It Works)

Every  $P_i$  is generated by deviating from earlier paths at some spur node, ensuring uniqueness. Dijkstra ensures each spur path is locally shortest. Since all candidates are stored and selected by global order, the sequence  $P_1, P_2, \dots, P_K$  is globally sorted by total cost.

Thus:

$$\text{len}(P_1) \leq \text{len}(P_2) \leq \dots \leq \text{len}(P_K)$$

and each path is simple.

## Try It Yourself

1. Generate the 3 shortest paths in a small network.
2. Compare with Eppstein's algorithm (faster for large K).
3. Remove edges dynamically and re-run, observe reordering.
4. Visualize path tree (root =  $s$ , branches = deviations).
5. Adapt for undirected or bi-directional graphs.

## Test Cases

| Graph         | $s$        | $t$ | $K$ | Paths                            |
|---------------|------------|-----|-----|----------------------------------|
| Line 0–1–2–3  | 0          | 3   | 2   | one main, one deviation          |
| Triangle      | 0          | 2   | 3   | 0–1–2, 0–2, 0–1–0–2 (if allowed) |
| Weighted grid | start–goal | 3   |     | alternate routes                 |

## Complexity

- Time:  $O(K \cdot V \cdot (E + V \log V))$
- Space:  $O(KV)$
- Condition: Nonnegative weights, finite paths

Yen's Algorithm is the explorer's Dijkstra, it doesn't stop at the first road found but keeps discovering better detours, one deviation at a time.

## Section 35. Minimum Spanning Trees

### 341 Kruskal's Algorithm

Kruskal's algorithm builds a minimum spanning tree (MST) by repeatedly adding the lightest edge that does not create a cycle. It uses a disjoint set union (DSU) data structure to track components efficiently.

#### What Problem Are We Solving?

Given a connected, undirected, weighted graph  $G = (V, E, w)$ , find a subset  $T \subseteq E$  such that:

- $T$  connects all vertices,
- $|T| = |V| - 1$ ,
- $\sum_{e \in T} w(e)$  is minimized.

If the graph is not connected, Kruskal builds a minimum spanning forest.

#### How Does It Work (Plain Language)?

1. Sort all edges by nondecreasing weight.
2. Initialize DSU with each vertex in its own set.
3. Scan edges in order. For edge  $(u, v)$ :
  - If  $u$  and  $v$  are in different sets, add the edge to the MST and union their sets.
  - Otherwise, skip it to avoid a cycle.
4. Stop when you have  $|V| - 1$  edges.

| Step           | Action                                  | DSU effect                         |
|----------------|-----------------------------------------|------------------------------------|
| Sort edges     | Light to heavy                          | None                               |
| Check $(u, v)$ | If $\text{find}(u) \neq \text{find}(v)$ | $\text{union}(u, v)$ and keep edge |
| Cycle case     | If $\text{find}(u) = \text{find}(v)$    | skip edge                          |

## Tiny Code

C (DSU + Kruskal)

```
#include <stdio.h>
#include <stdlib.h>

typedef struct { int u, v; int w; } Edge;

typedef struct {
    int *p, *r;
    int n;
} DSU;

DSU make_dsu(int n){
    DSU d; d.n = n;
    d.p = malloc(n * sizeof(int));
    d.r = malloc(n * sizeof(int));
    for(int i=0;i<n;i++){ d.p[i]=i; d.r[i]=0; }
    return d;
}

int find(DSU *d, int x){
    if(d->p[x]!=x) d->p[x]=find(d, d->p[x]);
    return d->p[x];
}

void unite(DSU *d, int a, int b){
    a=find(d,a); b=find(d,b);
    if(a==b) return;
    if(d->r[a]<d->r[b]) d->p[a]=b;
    else if(d->r[b]<d->r[a]) d->p[b]=a;
    else { d->p[b]=a; d->r[a]++; }
}

int cmp_edge(const void* A, const void* B){
    Edge *a=(Edge*)A, *b=(Edge*)B;
    return a->w - b->w;
}

int main(void){
    int n, m;
    scanf("%d %d", &n, &m);
    Edge *E = malloc(m * sizeof(Edge));
    for(int i=0;i<m;i++) scanf("%d %d %d", &E[i].u, &E[i].v, &E[i].w);
```

```

qsort(E, m, sizeof(Edge), cmp_edge);
DSU d = make_dsu(n);

int taken = 0;
long long cost = 0;
for(int i=0;i<m && taken < n-1;i++){
    int a = find(&d, E[i].u), b = find(&d, E[i].v);
    if(a != b){
        unite(&d, a, b);
        cost += E[i].w;
        taken++;
    }
}
if(taken != n-1) { printf("Graph not connected\n"); }
else { printf("MST cost: %lld\n", cost); }
return 0;
}

```

Python (DSU + Kruskal)

```

class DSU:
    def __init__(self, n):
        self.p = list(range(n))
        self.r = [0]*n
    def find(self, x):
        if self.p[x] != x:
            self.p[x] = self.find(self.p[x])
        return self.p[x]
    def union(self, a, b):
        a, b = self.find(a), self.find(b)
        if a == b: return False
        if self.r[a] < self.r[b]: self.p[a] = b
        elif self.r[b] < self.r[a]: self.p[b] = a
        else: self.p[b] = a; self.r[a] += 1
        return True

def kruskal(n, edges):
    edges = sorted(edges)
    dsu = DSU(n)
    cost = 0
    mst = []
    for w, u, v in edges:

```

```

if dsu.union(u, v):
    cost += w
    mst.append((u, v, w))
    if len(mst) == n-1:
        break
return cost, mst

n = 4
edges = [(1,0,1),(4,0,2),(3,1,2),(2,1,3),(5,2,3)]
print(kruskal(n, edges))

```

## Why It Matters

- Simple and fast with sorting plus DSU.
- Works well on sparse graphs.
- Produces an MST that minimizes total edge weight.
- Easy to adapt for minimum spanning forest on disconnected graphs.

## A Gentle Proof (Why It Works)

Kruskal relies on the cut property: Let  $S \subset V$  be any proper subset and consider the cut  $(S, V \setminus S)$ . The lightest edge crossing the cut is safe to include in some MST.

Sorting edges by weight and always taking the next lightest that connects two different components is equivalent to repeatedly applying the cut property to the partition defined by current DSU components. This never creates a cycle and never excludes the possibility of an optimal MST. By induction on the number of chosen edges, the final set is an MST.

## Try It Yourself

1. Generate random sparse graphs and compare Kruskal vs Prim.
2. Remove an edge from the MST and recompute to observe changes.
3. Force ties in edge weights and confirm multiple valid MSTs.
4. Run on a disconnected graph to obtain a minimum spanning forest.
5. Log chosen edges to visualize the growth of components.

## Test Cases

| Graph        | Edges $(u, v, w)$                             | MST Edges          | Cost       |
|--------------|-----------------------------------------------|--------------------|------------|
| Triangle     | $(0,1,1), (1,2,2), (0,2,3)$                   | $(0,1,1), (1,2,2)$ | 3          |
| Square       | $(0,1,1), (1,2,1), (2,3,1), (3,0,1), (0,2,2)$ | any 3 of weight 1  | 3          |
| Disconnected | two separate triangles                        | MST per component  | sum of two |

## Complexity

- Time:  $O(E \log E)$  for sorting plus almost linear DSU, which is  $O(E\alpha(V))$ . Usually written as  $O(E \log E)$  or  $O(E \log V)$ .
- Space:  $O(V + E)$ .
- Output size:  $|V| - 1$  edges for a connected graph.

Kruskal is the sort then stitch approach to MSTs. Sort edges globally, then stitch components locally with DSU until the tree locks into place.

## 342 Prim's Algorithm (Heap)

Prim's algorithm builds a minimum spanning tree (MST) by growing a connected subtree one vertex at a time, always choosing the lightest edge that connects a vertex inside the tree to one outside. It's a greedy algorithm, often implemented with a min-heap for efficiency.

### What Problem Are We Solving?

Given a connected, undirected, weighted graph  $G = (V, E, w)$ , find a subset of edges  $T \subseteq E$  such that:

- $T$  connects all vertices,
- $|T| = |V| - 1$ ,
- $\sum_{e \in T} w(e)$  is minimized.

Prim's algorithm grows the MST from a single seed vertex.

### How Does It Work (Plain Language)?

1. Choose any starting vertex  $s$ .
2. Initialize all vertices with  $\text{key}[v] = \infty$ , except  $\text{key}[s] = 0$ .
3. Use a min-heap (priority queue) keyed by edge weight.

4. Repeatedly extract the vertex  $u$  with the smallest key (edge weight):

- Mark  $u$  as part of the MST.
- For each neighbor  $v$  of  $u$ :
  - If  $v$  is not yet in the MST and  $w(u, v) < \text{key}[v]$ , update  $\text{key}[v] = w(u, v)$  and set  $\text{parent}[v] = u$ .

5. Continue until all vertices are included.

| Step            | Action               | Description         |
|-----------------|----------------------|---------------------|
| Initialize      | Choose start vertex  | $\text{key}[s] = 0$ |
| Extract min     | Add lightest edge    | Expands MST         |
| Relax neighbors | Update cheaper edges | Maintain frontier   |

### Tiny Code

Python (Min-Heap Version)

```
import heapq

def prim_mst(V, edges, start=0):
    graph = [[] for _ in range(V)]
    for u, v, w in edges:
        graph[u].append((w, v))
        graph[v].append((w, u)) # undirected

    visited = [False] * V
    pq = [(0, start, -1)] # (weight, vertex, parent)
    mst_edges = []
    total_cost = 0

    while pq:
        w, u, parent = heapq.heappop(pq)
        if visited[u]:
            continue
        visited[u] = True
        total_cost += w
        if parent != -1:
            mst_edges.append((parent, u, w))
        for weight, v in graph[u]:
```

```

        if not visited[v]:
            heapq.heappush(pq, (weight, v, u))
    return total_cost, mst_edges

edges = [(0,1,2),(0,2,3),(1,2,1),(1,3,4),(2,3,5)]
cost, mst = prim_mst(4, edges)
print(cost, mst)

```

Output:

7 [(0,1,2), (1,2,1), (1,3,4)]

## Why It Matters

- Suitable for dense graphs (especially with adjacency lists and heaps).
- Builds MST incrementally (like Dijkstra).
- Great for online construction where the tree must stay connected.
- Easier to integrate with adjacency matrix for small graphs.

## A Gentle Proof (Why It Works)

Prim's algorithm obeys the cut property: At each step, consider the cut between the current MST set  $S$  and the remaining vertices  $V \setminus S$ . The lightest edge crossing that cut is always safe to include. By repeatedly choosing the minimum such edge, Prim's maintains a valid MST prefix. When all vertices are added, the resulting tree is minimal.

## Try It Yourself

1. Run Prim's on a dense graph vs Kruskal, compare edge choices.
2. Visualize the growing frontier.
3. Try with adjacency matrix (without heap).
4. Test on disconnected graph, each component forms its own tree.
5. Replace heap with a simple array to see  $O(V^2)$  version.

| Graph | Edges $(u, v, w)$ | MST Edges | Cost |
|-------|-------------------|-----------|------|
|-------|-------------------|-----------|------|

## Test Cases

| Graph    | Edges $(u, v, w)$                    | MST Edges                   | Cost |
|----------|--------------------------------------|-----------------------------|------|
| Triangle | $(0,1,1), (1,2,2), (0,2,3)$          | $(0,1,1), (1,2,2)$          | 3    |
| Square   | $(0,1,2), (1,2,3), (2,3,1), (3,0,4)$ | $(2,3,1), (0,1,2), (1,2,3)$ | 6    |
| Line     | $(0,1,5), (1,2,1), (2,3,2)$          | $(1,2,1), (2,3,2), (0,1,5)$ | 8    |

## Complexity

- Time:  $O(E \log V)$  with heap,  $O(V^2)$  with array.
- Space:  $O(V + E)$
- Output size:  $|V| - 1$  edges

Prim's is the grow from seed approach to MSTs. It builds the tree step by step, always pulling the next lightest edge from the frontier.

## 343 Prim's Algorithm (Adjacency Matrix)

This is the array-based version of Prim's algorithm, optimized for dense graphs. Instead of using a heap, it directly scans all vertices to find the next minimum key vertex at each step. The logic is identical to Prim's heap version but trades priority queues for simple loops.

### What Problem Are We Solving?

Given a connected, undirected, weighted graph  $G = (V, E, w)$ , represented by an adjacency matrix  $W$ , we want to construct an MST, a subset of edges that:

- connects all vertices,
- contains  $|V| - 1$  edges,
- and minimizes total weight  $\sum w(e)$ .

The adjacency matrix form simplifies edge lookups and is ideal for dense graphs, where  $E \approx V^2$ .

## How Does It Work (Plain Language)?

1. Start from an arbitrary vertex (say 0).
2. Initialize  $\text{key}[v] = \infty$  for all vertices, except  $\text{key}[0] = 0$ .
3. Maintain a set  $\text{inMST}[v]$  marking vertices already included.
4. Repeat  $V - 1$  times:
  - Choose the vertex  $u$  not yet in MST with the smallest  $\text{key}[u]$ .
  - Add  $u$  to MST.
  - For each vertex  $v$ , if  $W[u][v]$  is smaller than  $\text{key}[v]$ , update it and record parent  $v \leftarrow u$ .

At each iteration, one vertex joins the MST, the one connected by the lightest edge to the existing set.

| Step        | Operation           | Description              |
|-------------|---------------------|--------------------------|
| Initialize  | $\text{key}[0] = 0$ | start from vertex 0      |
| Extract min | find smallest key   | next vertex to add       |
| Update      | relax edges         | update keys of neighbors |

## Tiny Code

C (Adjacency Matrix Version)

```
#include <stdio.h>
#include <limits.h>
#include <stdbool.h>

#define INF 1000000000
#define N 100

int minKey(int key[], bool inMST[], int n) {
    int min = INF, idx = -1;
    for (int v = 0; v < n; v++)
        if (!inMST[v] && key[v] < min)
            min = key[v], idx = v;
    return idx;
}

void primMatrix(int graph[N][N], int n) {
```

```

int key[N], parent[N];
bool inMST[N];
for (int i = 0; i < n; i++)
    key[i] = INF, inMST[i] = false;
key[0] = 0, parent[0] = -1;

for (int count = 0; count < n - 1; count++) {
    int u = minKey(key, inMST, n);
    inMST[u] = true;
    for (int v = 0; v < n; v++)
        if (graph[u][v] && !inMST[v] && graph[u][v] < key[v]) {
            parent[v] = u;
            key[v] = graph[u][v];
        }
}
int total = 0;
for (int i = 1; i < n; i++) {
    printf("%d - %d: %d\n", parent[i], i, graph[i][parent[i]]);
    total += graph[i][parent[i]];
}
printf("MST cost: %d\n", total);
}

int main() {
    int n = 5;
    int graph[N][N] = {
        {0,2,0,6,0},
        {2,0,3,8,5},
        {0,3,0,0,7},
        {6,8,0,0,9},
        {0,5,7,9,0}
    };
    primMatrix(graph, n);
    return 0;
}

```

Output:

```

0 - 1: 2
1 - 2: 3
0 - 3: 6
1 - 4: 5

```

MST cost: 16

Python Version

```
def prim_matrix(graph):
    V = len(graph)
    key = [float('inf')] * V
    parent = [-1] * V
    in_mst = [False] * V
    key[0] = 0

    for _ in range(V - 1):
        u = min((key[v], v) for v in range(V) if not in_mst[v])[1]
        in_mst[u] = True
        for v in range(V):
            if graph[u][v] != 0 and not in_mst[v] and graph[u][v] < key[v]:
                key[v] = graph[u][v]
                parent[v] = u

    edges = [(parent[i], i, graph[i][parent[i]]) for i in range(1, V)]
    cost = sum(w for _, _, w in edges)
    return cost, edges

graph = [
    [0,2,0,6,0],
    [2,0,3,8,5],
    [0,3,0,0,7],
    [6,8,0,0,9],
    [0,5,7,9,0]
]
print(prim_matrix(graph))
```

Output:

(16, [(0,1,2), (1,2,3), (0,3,6), (1,4,5)])

## Why It Matters

- Simple to implement with adjacency matrices.
- Best for dense graphs where  $E \approx V^2$ .
- Avoids the complexity of heaps.
- Easy to visualize and debug for classroom or teaching use.

## A Gentle Proof (Why It Works)

Like the heap-based version, this variant relies on the cut property: At each step, the chosen edge connects a vertex inside the tree to one outside with minimum weight, so it is always safe. Each iteration expands the MST without cycles, and after  $V - 1$  iterations, all vertices are connected.

## Try It Yourself

1. Run on a complete graph, expect  $V - 1$  smallest edges.
2. Modify weights, see how edge choices change.
3. Compare with heap-based Prim on runtime as  $V$  grows.
4. Implement in  $O(V^2)$  and confirm complexity experimentally.
5. Test on disconnected graph, see where algorithm stops.

## Test Cases

| Graph           | MST Edges                          | Cost                     |
|-----------------|------------------------------------|--------------------------|
| 5-node matrix   | (0,1,2), (1,2,3), (0,3,6), (1,4,5) | 16                       |
| 3-node triangle | (0,1,1), (1,2,2)                   | 3                        |
| Dense complete  | 6 edges                            | chooses lightest 3 edges |
| 4-node          |                                    | sum of min               |

## Complexity

- Time:  $O(V^2)$
- Space:  $O(V^2)$  (matrix)
- Output size:  $|V| - 1$  edges

Prim (Adjacency Matrix) is the classic dense-graph version, it trades speed for simplicity and predictable access time.

## 344 Borůvka's Algorithm

Borůvka's algorithm is one of the earliest MST algorithms (1926), designed to build the minimum spanning tree by repeatedly connecting each component with its cheapest outgoing edge. It operates in phases, merging components until a single tree remains.

## What Problem Are We Solving?

Given a connected, undirected, weighted graph  $G = (V, E, w)$ , we want to find a minimum spanning tree (MST), a subset  $T \subseteq E$  such that:

- $T$  connects all vertices,
- $|T| = |V| - 1$ ,
- $\sum_{e \in T} w(e)$  is minimized.

Borůvka's approach grows multiple subtrees in parallel, making it highly suitable for parallel or distributed computation.

## How Does It Work (Plain Language)?

The algorithm works in rounds. Each round connects every component to another via its lightest outgoing edge.

1. Start with each vertex as its own component.
2. For each component, find the minimum-weight edge that connects it to another component.
3. Add all these edges to the MST, they are guaranteed safe (by the cut property).
4. Merge components connected by these edges.
5. Repeat until only one component remains.

Each round at least halves the number of components, so the algorithm finishes in  $O(\log V)$  phases.

| Step | Operation                        | Description            |
|------|----------------------------------|------------------------|
| 1    | Initialize components            | each vertex alone      |
| 2    | Find cheapest edge per component | lightest outgoing      |
| 3    | Add edges                        | merge components       |
| 4    | Repeat                           | until single component |

## Tiny Code

Python (DSU Version)

```
class DSU:  
    def __init__(self, n):  
        self.p = list(range(n))  
        self.r = [0] * n  
    def find(self, x):
```

```

    if self.p[x] != x:
        self.p[x] = self.find(self.p[x])
    return self.p[x]
def union(self, a, b):
    a, b = self.find(a), self.find(b)
    if a == b: return False
    if self.r[a] < self.r[b]: self.p[a] = b
    elif self.r[a] > self.r[b]: self.p[b] = a
    else: self.p[b] = a; self.r[a] += 1
    return True

def boruvka_mst(V, edges):
    dsu = DSU(V)
    mst = []
    total_cost = 0
    components = V

    while components > 1:
        cheapest = [-1] * V
        for i, (u, v, w) in enumerate(edges):
            set_u = dsu.find(u)
            set_v = dsu.find(v)
            if set_u != set_v:
                if cheapest[set_u] == -1 or edges[cheapest[set_u]][2] > w:
                    cheapest[set_u] = i
                if cheapest[set_v] == -1 or edges[cheapest[set_v]][2] > w:
                    cheapest[set_v] = i
        for i in range(V):
            if cheapest[i] != -1:
                u, v, w = edges[cheapest[i]]
                if dsu.union(u, v):
                    mst.append((u, v, w))
                    total_cost += w
                    components -= 1
    return total_cost, mst

edges = [(0,1,1), (0,2,3), (1,2,1), (1,3,4), (2,3,5)]
print(boruvka_mst(4, edges))

```

Output:

(6, [(0,1,1), (1,2,1), (1,3,4)])

## Why It Matters

- Naturally parallelizable (each component acts independently).
- Simple and elegant, based on repeated application of the cut property.
- Ideal for sparse graphs and distributed systems.
- Often used in hybrid MST algorithms (combining with Kruskal/Prim).

## A Gentle Proof (Why It Works)

By the cut property, for each component  $C$ , the cheapest edge leaving  $C$  is always safe to include in the MST. Since edges are chosen simultaneously across all components, and no cycles are created within a single phase (components merge only across cuts), each round preserves correctness. After each phase, components merge, and the process repeats until all are unified.

Each iteration reduces the number of components by at least half, ensuring  $O(\log V)$  phases.

## Try It Yourself

1. Run on a small graph and trace phases manually.
2. Compare with Kruskal's sorted-edge approach.
3. Add parallel logging to visualize simultaneous merges.
4. Observe how components shrink exponentially.
5. Mix with Kruskal: use Borůvka until few components remain, then switch.

## Test Cases

| Graph           | Edges $(u, v, w)$                    | MST                         | Cost          |
|-----------------|--------------------------------------|-----------------------------|---------------|
| Triangle        | $(0,1,1), (1,2,2), (0,2,3)$          | $(0,1,1), (1,2,2)$          | 3             |
| Square          | $(0,1,1), (1,2,2), (2,3,1), (3,0,2)$ | $(0,1,1), (2,3,1), (1,2,2)$ | 4             |
| Dense<br>4-node | 6 edges                              | builds MST in 2 phases      | verified cost |

## Complexity

- Phases:  $O(\log V)$
- Time per phase:  $O(E)$
- Total Time:  $O(E \log V)$
- Space:  $O(V + E)$

Borůvka's algorithm is the parallel grow-and-merge strategy for MSTs, every component reaches out through its lightest edge until the whole graph becomes one connected tree.

## 345 Reverse-Delete MST

The Reverse-Delete Algorithm builds a minimum spanning tree (MST) by starting with the full graph and repeatedly removing edges, but only when their removal does not disconnect the graph. It's the conceptual mirror image of Kruskal's algorithm.

### What Problem Are We Solving?

Given a connected, undirected, weighted graph  $G = (V, E, w)$ , we want to find an MST — a spanning tree that connects all vertices with minimum total weight.

Instead of adding edges like Kruskal, we start with all edges and delete them one by one, making sure the graph remains connected after each deletion.

### How Does It Work (Plain Language)?

1. Sort all edges by decreasing weight.
2. Initialize the working graph  $T = G$ .
3. For each edge  $(u, v)$  in that order:
  - Temporarily remove  $(u, v)$  from  $T$ .
  - If  $u$  and  $v$  are still connected in  $T$ , permanently delete the edge (it's not needed).
  - Otherwise, restore it (it's essential).
4. When all edges are processed,  $T$  is the MST.

This approach ensures only indispensable edges remain, forming a spanning tree of minimal total weight.

| Step | Action                | Description            |
|------|-----------------------|------------------------|
| 1    | Sort edges descending | heavy edges first      |
| 2    | Remove edge           | test connectivity      |
| 3    | Keep if needed        | if removal disconnects |
| 4    | Stop                  | when $T = V - 1$       |

## Tiny Code

Python (Using DFS for Connectivity Check)

```
def dfs(graph, start, visited):
    visited.add(start)
    for v, _ in graph[start]:
        if v not in visited:
            dfs(graph, v, visited)

def is_connected(graph, u, v):
    visited = set()
    dfs(graph, u, visited)
    return v in visited

def reverse_delete_mst(V, edges):
    edges = sorted(edges, key=lambda x: x[2], reverse=True)
    graph = {i: [] for i in range(V)}
    for u, v, w in edges:
        graph[u].append((v, w))
        graph[v].append((u, w))

    mst_cost = sum(w for _, _, w in edges)

    for u, v, w in edges:
        # remove edge
        graph[u] = [(x, wx) for x, wx in graph[u] if x != v]
        graph[v] = [(x, wx) for x, wx in graph[v] if x != u]

        if is_connected(graph, u, v):
            mst_cost -= w # edge not needed
        else:
            # restore edge
            graph[u].append((v, w))
            graph[v].append((u, w))

    mst_edges = []
    visited = set()
    def collect(u):
        visited.add(u)
        for v, w in graph[u]:
            if (v, w) not in visited:
                mst_edges.append((u, v, w))
```

```

        if v not in visited:
            collect(v)

collect(0)
return mst_cost, mst_edges

edges = [(0,1,1), (0,2,2), (1,2,3), (1,3,4), (2,3,5)]
print(reverse_delete_mst(4, edges))

```

Output:

(7, [(0,1,1), (0,2,2), (1,3,4)])

## Why It Matters

- Simple dual perspective to Kruskal's algorithm.
- Demonstrates the cycle property:

In any cycle, the edge with the largest weight cannot be in an MST.

- Good for teaching proofs and conceptual understanding.
- Can be used for verifying MSTs or constructing them in reverse.

## A Gentle Proof (Why It Works)

The cycle property states:

For any cycle in a graph, the edge with the largest weight cannot belong to the MST.

By sorting edges in descending order and removing each maximum-weight edge that lies in a cycle (i.e., when  $u$  and  $v$  remain connected without it), we eliminate all non-MST edges. When no such edge remains, the result is an MST.

Since each deletion preserves connectivity, the final graph is a spanning tree with minimal total weight.

## Try It Yourself

1. Run on a triangle graph, see heaviest edge removed.
2. Compare deletion order with Kruskal's addition order.
3. Visualize graph at each step.
4. Replace DFS with BFS or Union-Find for speed.
5. Use to verify MST output from another algorithm.

## Test Cases

| Graph    | Edges $(u, v, w)$                    | MST                         | Cost |
|----------|--------------------------------------|-----------------------------|------|
| Triangle | $(0,1,1), (1,2,2), (0,2,3)$          | $(0,1,1), (1,2,2)$          | 3    |
| Square   | $(0,1,1), (1,2,2), (2,3,3), (3,0,4)$ | $(0,1,1), (1,2,2), (2,3,3)$ | 6    |
| Line     | $(0,1,2), (1,2,1), (2,3,3)$          | all edges                   | 6    |

## Complexity

- Time:  $O(E(E + V))$  with naive DFS (each edge removal checks connectivity). With Union-Find optimizations or precomputed structures, it can be reduced.
- Space:  $O(V + E)$ .

Reverse-Delete is the subtract instead of add view of MSTs, peel away heavy edges until only the essential light structure remains.

## 346 MST via Dijkstra Trick

This variant constructs a Minimum Spanning Tree (MST) using a Dijkstra-like process, repeatedly expanding the tree by selecting the lightest edge connecting any vertex inside the tree to one outside. It's essentially Prim's algorithm recast through Dijkstra's lens, showing the deep parallel between shortest-path and spanning-tree growth.

### What Problem Are We Solving?

Given a connected, undirected, weighted graph  $G = (V, E, w)$  with nonnegative edge weights, we want an MST — a subset  $T \subseteq E$  that connects all vertices with

$$|T| = |V| - 1, \quad \text{and} \quad \sum_{e \in T} w(e) \text{ minimized.}$$

While Dijkstra's algorithm builds shortest-path trees based on *path cost*, this version builds an MST by using edge weights directly as "distances" from the current tree.

## How Does It Work (Plain Language)?

This is Prim's algorithm in disguise: Instead of tracking the shortest path from a root, we track the lightest edge connecting each vertex to the growing tree.

1. Initialize all vertices with  $\text{key}[v] = \infty$ , except start vertex  $s$  with  $\text{key}[s] = 0$ .
2. Use a priority queue (min-heap) keyed by  $\text{key}[v]$ .
3. Repeatedly extract vertex  $u$  with smallest key.
4. For each neighbor  $v$ :
  - If  $v$  not yet in tree and  $w(u, v) < \text{key}[v]$ , update  $\text{key}[v] = w(u, v)$  and record parent.
5. Repeat until all vertices are included.

Unlike Dijkstra, we do not sum edge weights, we only care about the minimum edge to reach each vertex.

| Step            | Dijkstra                                    | MST via Dijkstra Trick                         |
|-----------------|---------------------------------------------|------------------------------------------------|
| Distance update | $\text{dist}[v] = \text{dist}[u] + w(u, v)$ | $\text{key}[v] = \min(\text{key}[v], w(u, v))$ |
| Priority        | Path cost                                   | Edge cost                                      |
| Goal            | Shortest path                               | MST                                            |

## Tiny Code

Python (Heap Implementation)

```
import heapq

def mst_dijkstr_trick(V, edges, start=0):
    graph = [[] for _ in range(V)]
    for u, v, w in edges:
        graph[u].append((v, w))
        graph[v].append((u, w))

    in_tree = [False] * V
    key = [float('inf')] * V
    parent = [-1] * V
    key[start] = 0
    pq = [(0, start)]
```

```

total_cost = 0
while pq:
    w, u = heapq.heappop(pq)
    if in_tree[u]:
        continue
    in_tree[u] = True
    total_cost += w

    for v, wt in graph[u]:
        if not in_tree[v] and wt < key[v]:
            key[v] = wt
            parent[v] = u
            heapq.heappush(pq, (wt, v))

mst_edges = [(parent[v], v, key[v]) for v in range(V) if parent[v] != -1]
return total_cost, mst_edges

edges = [(0,1,2),(0,2,3),(1,2,1),(1,3,4),(2,3,5)]
print(mst_dijkstra_trick(4, edges))

```

Output:

```
(7, [(0,1,2), (1,2,1), (1,3,4)])
```

## Why It Matters

- Demonstrates the duality between MST and shortest-path search.
- Provides a conceptual bridge between Prim and Dijkstra.
- Useful for teaching and algorithmic unification.
- Intuitive when already familiar with Dijkstra's structure.

## A Gentle Proof (Why It Works)

At each iteration, we maintain a set  $S$  of vertices already in the MST. The cut property guarantees that the lightest edge connecting  $S$  to  $V \setminus S$  is always safe to include.

By storing these edge weights in a priority queue and always selecting the smallest, we exactly follow this property, thus constructing an MST incrementally.

The algorithm stops when all vertices are included, yielding an MST.

## Try It Yourself

1. Compare line by line with Dijkstra, only one change in relaxation.
2. Run on complete graphs, observe star-like MSTs.
3. Try graphs with multiple equal edges, see tie-handling.
4. Replace heap with array, check  $O(V^2)$  version.
5. Visualize with frontier highlighting, edges instead of distances.

## Test Cases

| Graph    | Edges ( $u, v, w$ )             | MST                     | Cost |
|----------|---------------------------------|-------------------------|------|
| Triangle | (0,1,1),(1,2,2),(0,2,3)         | (0,1,1),(1,2,2)         | 3    |
| Square   | (0,1,2),(1,2,3),(2,3,1),(3,0,4) | (2,3,1),(0,1,2),(1,2,3) | 6    |
| Line     | (0,1,5),(1,2,2),(2,3,1)         | all edges               | 8    |

## Complexity

- Time:  $O(E \log V)$  (heap) or  $O(V^2)$  (array)
- Space:  $O(V + E)$
- Output size:  $|V| - 1$  edges

MST via Dijkstra Trick is Prim's algorithm reimagined, it replaces distance summation with edge minimization, proving how two classic graph ideas share one greedy heart.

## 347 Dynamic MST Maintenance

Dynamic MST maintenance addresses the question: how can we update an MST efficiently when the underlying graph changes, edges are added, removed, or their weights change, without rebuilding from scratch?

This problem arises in systems where the graph evolves over time, such as networks, road maps, or real-time optimization systems.

## What Problem Are We Solving?

Given a graph  $G = (V, E, w)$  and its MST  $T$ , we want to maintain  $T$  under updates:

- Edge Insertion: add new edge  $(u, v, w)$
- Edge Deletion: remove existing edge  $(u, v)$
- Edge Weight Update: change weight  $w(u, v)$

Naively recomputing the MST after each change costs  $O(E \log V)$ . Dynamic maintenance reduces this significantly using incremental repair of  $T$ .

## How Does It Work (Plain Language)?

1. Edge Insertion:
  - Add the new edge  $(u, v, w)$ .
  - Check if it creates a cycle in  $T$ .
  - If the new edge is lighter than the heaviest edge on that cycle, replace the heavy one.
  - Otherwise, discard it.
2. Edge Deletion:
  - Remove  $(u, v)$  from  $T$ , splitting it into two components.
  - Find the lightest edge connecting the two components in  $G$ .
  - Add that edge to restore connectivity.
3. Edge Weight Update:
  - If an edge's weight increases, treat as potential deletion.
  - If it decreases, treat as potential insertion.

To do this efficiently, we need data structures that can:

- find max edge on path quickly (for cycles)
- find min cross edge between components

These can be implemented via dynamic trees, Link-Cut Trees, or Euler Tour Trees.

## Tiny Code (Simplified Static Version)

This version demonstrates edge insertion maintenance with cycle detection.

```
def find(parent, x):  
    if parent[x] != x:  
        parent[x] = find(parent, parent[x])  
    return parent[x]  
  
def union(parent, rank, x, y):  
    rx, ry = find(parent, x), find(parent, y)  
    if rx == ry:
```

```

        return False
    if rank[rx] < rank[ry]:
        parent[rx] = ry
    elif rank[rx] > rank[ry]:
        parent[ry] = rx
    else:
        parent[ry] = rx
        rank[rx] += 1
    return True

def dynamic_mst_insert(V, mst_edges, new_edge):
    u, v, w = new_edge
    parent = [i for i in range(V)]
    rank = [0]*V
    for x, y, _ in mst_edges:
        union(parent, rank, x, y)
    if find(parent, u) != find(parent, v):
        mst_edges.append(new_edge)
    else:
        # would form cycle, pick lighter edge
        cycle_edges = mst_edges + [new_edge]
        heaviest = max(cycle_edges, key=lambda e: e[2])
        if heaviest != new_edge:
            mst_edges.remove(heaviest)
            mst_edges.append(new_edge)
    return mst_edges

```

This is conceptual; in practice, Link-Cut Trees make this dynamic in logarithmic time.

## Why It Matters

- Crucial for streaming graphs, online networks, real-time routing.
- Avoids recomputation after each update.
- Demonstrates greedy invariants under change, the MST remains minimal if maintained properly.
- Forms basis for fully dynamic graph algorithms.

## A Gentle Proof (Why It Works)

MST invariants are preserved through cut and cycle properties:

- Cycle Property: In a cycle, the heaviest edge cannot be in an MST.
- Cut Property: In any cut, the lightest edge crossing it must be in the MST.

When inserting an edge:

- It forms a cycle, we drop the heaviest to maintain minimality.

When deleting an edge:

- It forms a cut, we insert the lightest crossing edge to maintain connectivity.

Thus each update restores a valid MST in local time.

### Try It Yourself

1. Start from an MST of a small graph.
2. Insert an edge and track the created cycle.
3. Delete an MST edge and find replacement.
4. Try batch updates, see structure evolve.
5. Compare runtime with full recomputation.

### Test Cases

| Operation              | Description                | Result                          |
|------------------------|----------------------------|---------------------------------|
| Insert (1, 3, 1)       | creates cycle (1, 2, 3, 1) | remove heaviest edge            |
| Delete (0, 1)          | breaks tree                | find lightest reconnecting edge |
| Decrease weight (2, 3) | recheck inclusion          | edge may enter MST              |

### Complexity

- Insertion/Deletion (Naive):  $O(E)$
- Dynamic Tree (Link-Cut):  $O(\log^2 V)$  per update
- Space:  $O(V + E)$

Dynamic MST maintenance shows how local adjustments preserve global optimality, a powerful principle for evolving systems.

## 348 Minimum Bottleneck Spanning Tree

A Minimum Bottleneck Spanning Tree (MBST) is a spanning tree that minimizes the maximum edge weight among all edges in the tree. Unlike the standard MST, which minimizes the total sum of edge weights, an MBST focuses on the worst (heaviest) edge in the tree.

In many real-world systems, such as network design or transportation planning, you may care less about total cost and more about bottleneck constraints, the weakest or slowest connection.

### What Problem Are We Solving?

Given a connected, undirected, weighted graph  $G = (V, E, w)$ , we want a spanning tree  $T \subseteq E$  such that

$$\max_{e \in T} w(e)$$

is as small as possible.

In other words, among all spanning trees, pick one where the largest edge weight is minimal.

### How Does It Work (Plain Language)?

You can build an MBST using the same algorithms as MST (Kruskal or Prim), because:

Every Minimum Spanning Tree is also a Minimum Bottleneck Spanning Tree.

The reasoning:

- An MST minimizes the total sum.
- In doing so, it also ensures no unnecessarily heavy edges remain.

So any MST automatically satisfies the bottleneck property.

Alternatively, you can use a binary search + connectivity test:

1. Sort edges by weight.
2. Binary search for a threshold  $W$ .
3. Check if edges with  $w(e) \leq W$  can connect all vertices.
4. The smallest  $W$  for which graph is connected is the bottleneck weight.
5. Extract any spanning tree using only edges  $\leq W$ .

### Tiny Code (Kruskal-based)

```
def find(parent, x):
    if parent[x] != x:
        parent[x] = find(parent, parent[x])
    return parent[x]

def union(parent, rank, x, y):
    rx, ry = find(parent, x), find(parent, y)
    if rx == ry:
        return False
    if rank[rx] < rank[ry]:
        parent[rx] = ry
    elif rank[rx] > rank[ry]:
        parent[ry] = rx
    else:
        parent[ry] = rx
        rank[rx] += 1
    return True

def minimum_bottleneck_spanning_tree(V, edges):
    edges = sorted(edges, key=lambda x: x[2])
    parent = [i for i in range(V)]
    rank = [0]*V
    bottleneck = 0
    count = 0
    for u, v, w in edges:
        if union(parent, rank, u, v):
            bottleneck = max(bottleneck, w)
            count += 1
            if count == V - 1:
                break
    return bottleneck

edges = [(0,1,4),(0,2,3),(1,2,2),(1,3,5),(2,3,6)]
print(minimum_bottleneck_spanning_tree(4, edges))
```

Output:

5

Here, the MST has total weight  $4+3+2=9$ , and bottleneck edge weight 5.

## Why It Matters

- Useful for quality-of-service or bandwidth-constrained systems.
- Ensures no edge in the tree exceeds a critical capacity threshold.
- Illustrates that minimizing maximum and minimizing sum can align, a key insight in greedy algorithms.
- Provides a way to test MST correctness: if an MST doesn't minimize the bottleneck, it's invalid.

## A Gentle Proof (Why It Works)

Let  $T^*$  be an MST and  $B(T^*)$  be its maximum edge weight.

Suppose another tree  $T'$  had a smaller bottleneck:

$$B(T') < B(T^*)$$

Then there exists an edge  $e$  in  $T^*$  with  $w(e) = B(T^*)$ , but  $T'$  avoids it while staying connected using lighter edges.

This contradicts the cycle property, since  $e$  would be replaced by a lighter edge crossing the same cut, meaning  $T^*$  wasn't minimal.

Thus, every MST is an MBST.

## Try It Yourself

1. Build a graph with multiple MSTs, check if they share the same bottleneck.
2. Compare MST total weight vs MBST bottleneck.
3. Apply binary search approach, confirm consistency.
4. Visualize all spanning trees and mark max edge in each.
5. Construct a case where multiple MBSTs exist.

## Test Cases

| Graph    | Edges $(u, v, w)$                    | MST                         | Bottleneck |
|----------|--------------------------------------|-----------------------------|------------|
| Triangle | $(0,1,1), (1,2,2), (0,2,3)$          | $(0,1,1), (1,2,2)$          | 2          |
| Square   | $(0,1,1), (1,2,5), (2,3,2), (3,0,4)$ | $(0,1,1), (2,3,2), (3,0,4)$ | 4          |
| Chain    | $(0,1,2), (1,2,3), (2,3,4)$          | same                        | 4          |

## Complexity

- Time:  $O(E \log E)$  (same as Kruskal)
- Space:  $O(V)$
- Output: bottleneck weight or edges

A Minimum Bottleneck Spanning Tree highlights the heaviest load-bearing link, a critical measure when resilience, latency, or bandwidth limits matter more than total cost.

## 349 Manhattan MST

A Manhattan Minimum Spanning Tree (Manhattan MST) finds a spanning tree that minimizes the sum of Manhattan distances between connected points on a grid. This variant is common in VLSI design, city planning, and grid-based path optimization, where movement is constrained to axis-aligned directions.

### What Problem Are We Solving?

Given  $n$  points in 2D space with coordinates  $(x_i, y_i)$ , the Manhattan distance between points  $p_i$  and  $p_j$  is

$$d(p_i, p_j) = |x_i - x_j| + |y_i - y_j|$$

We want to build a spanning tree connecting all points such that the total Manhattan distance is minimal.

A naive solution considers all  $\binom{n}{2}$  edges and runs Kruskal's algorithm, but that's too slow for large  $n$ . The key is exploiting geometry to limit candidate edges.

### How Does It Work (Plain Language)?

The Manhattan MST problem leverages a geometric property:

For each point, only a small number of nearest neighbors (under certain transformations) can belong to the MST.

Thus, instead of checking all pairs, we:

1. Sort points in specific directions (rotations/reflections).
2. Use a sweep line or Fenwick tree to find candidate neighbors.
3. Collect  $O(n)$  potential edges.
4. Run Kruskal's algorithm on this reduced set.

By considering 8 directional transforms, we ensure all possible nearest edges are included.

Example transforms:

- $(x, y)$
- $(y, x)$
- $(-x, y)$
- $(x, -y)$
- etc.

For each transform:

- Sort points by  $(x + y)$  or  $(x - y)$ .
- For each, track candidate neighbor minimizing  $|x| + |y|$ .
- Add edge between each point and its nearest neighbor.

Finally, compute MST using Kruskal on the collected edges.

### Tiny Code (Simplified Illustration)

```
def manhattan_distance(p1, p2):
    return abs(p1[0]-p2[0]) + abs(p1[1]-p2[1])

def manhattan_mst(points):
    edges = []
    n = len(points)
    for i in range(n):
        for j in range(i+1, n):
            w = manhattan_distance(points[i], points[j])
            edges.append((i, j, w))
    edges.sort(key=lambda e: e[2])

    parent = [i for i in range(n)]
    def find(x):
        if parent[x] != x:
            parent[x]=find(parent[x])
        return parent[x]
    def union(x, y):
        rx, ry = find(x), find(y)
        if rx!=ry:
            parent[ry]=rx
        return True
```

```

    return False

mst_edges = []
cost = 0
for u, v, w in edges:
    if union(u, v):
        mst_edges.append((u, v, w))
        cost += w
return cost, mst_edges

points = [(0,0), (2,2), (2,0), (0,2)]
print(manhattan_mst(points))

```

Output:

(8, [(0,2,2),(0,3,2),(2,1,4)])

This brute-force example builds a Manhattan MST by checking all pairs. Efficient geometric variants reduce this from  $O(n^2)$  to  $O(n \log n)$ .

## Why It Matters

- Captures grid-based movement (no diagonals).
- Critical in VLSI circuit layout (wire length minimization).
- Foundational for city-block planning and delivery networks.
- Shows how geometry and graph theory merge in spatial problems.

## A Gentle Proof (Why It Works)

The MST under Manhattan distance obeys the cut property, the lightest edge crossing any partition must be in the MST. By ensuring all directional neighbors are included, we never miss the minimal edge across any cut.

Thus, even though we prune candidate edges, correctness is preserved.

## Try It Yourself

1. Generate random points, visualize MST edges.
2. Compare Manhattan MST vs Euclidean MST.
3. Add a diagonal, see how cost differs.
4. Try optimized directional neighbor search.
5. Observe symmetry, each transform covers a quadrant.

## Test Cases

| Points                  | MST Edges   | Total Cost |
|-------------------------|-------------|------------|
| (0,0),(1,0),(1,1)       | (0,1),(1,2) | 2          |
| (0,0),(2,2),(2,0),(0,2) | 3 edges     | 8          |
| (0,0),(3,0),(0,4)       | (0,1),(0,2) | 7          |

## Complexity

- Naive:  $O(n^2 \log n)$
- Optimized (geometry-based):  $O(n \log n)$
- Space:  $O(n)$

Manhattan MSTs bridge grid geometry and graph optimization, the perfect example of structure guiding efficiency.

## 350 Euclidean MST (Kruskal + Geometry)

A Euclidean Minimum Spanning Tree (EMST) connects a set of points in the plane with the shortest total Euclidean length, the minimal possible wiring, pipeline, or connection layout under straight-line distances.

It is the geometric counterpart to classical MST problems, central to computational geometry, network design, and spatial clustering.

## What Problem Are We Solving?

Given  $n$  points  $P = p_1, p_2, \dots, p_n$  in 2D (or higher dimensions), we want a spanning tree  $T$  minimizing

$$\text{cost}(T) = \sum_{(p_i, p_j) \in T} \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Unlike abstract graph MSTs, here the graph is complete, every pair of points has an edge weighted by Euclidean distance — but we cannot afford  $O(n^2)$  edges for large  $n$ .

## How Does It Work (Plain Language)?

The key geometric insight:

The EMST is always a subgraph of the Delaunay Triangulation (DT).

So we don't need all  $\binom{n}{2}$  edges, only those in the Delaunay graph (which has  $O(n)$  edges).

Algorithm sketch:

1. Compute the Delaunay Triangulation (DT) of the points.
2. Extract all DT edges with their Euclidean weights.
3. Run Kruskal's or Prim's algorithm on these edges.
4. The resulting tree is the EMST.

This drastically reduces time from quadratic to near-linear.

## Tiny Code (Brute-Force Demonstration)

Here's a minimal version using all pairs, good for small  $n$ :

```
import math

def euclidean_distance(p1, p2):
    return math.sqrt((p1[0]-p2[0])**2 + (p1[1]-p2[1])**2)

def euclidean_mst(points):
    edges = []
    n = len(points)
    for i in range(n):
        for j in range(i+1, n):
            w = euclidean_distance(points[i], points[j])
            edges.append((i, j, w))
    edges.sort(key=lambda e: e[2])

    parent = [i for i in range(n)]
```

```

def find(x):
    if parent[x] != x:
        parent[x] = find(parent[x])
    return parent[x]
def union(x, y):
    rx, ry = find(x), find(y)
    if rx != ry:
        parent[ry] = rx
    return True
    return False

mst_edges = []
cost = 0
for u, v, w in edges:
    if union(u, v):
        mst_edges.append((u, v, w))
        cost += w
return cost, mst_edges

points = [(0,0),(1,0),(0,1),(1,1)]
print(euclidean_mst(points))

```

Output:

(3.0, [(0,1,1.0),(0,2,1.0),(1,3,1.0)])

For real use, replace the all-pairs loop with Delaunay edges from a geometry library.

## Why It Matters

- Optimally connects points with minimum total length.
- Used in geographic information systems, road networks, clustering (single-linkage), sensor networks, and pattern analysis.
- Demonstrates interplay between geometry and graph theory.
- Foundation for Steiner trees and TSP approximations.

## A Gentle Proof (Why It Works)

The EMST obeys the cut property, for every partition of points, the shortest connecting edge must be in the EMST.

Edges outside the Delaunay Triangulation can be replaced by shorter edges crossing the same cut (from DT), so the EMST must lie within the DT.

Thus, we can safely restrict candidate edges to the Delaunay set, preserving optimality.

### Try It Yourself

1. Plot points on a plane, draw EMST edges.
2. Compare EMST vs MST with Manhattan distances.
3. Generate points in clusters, visualize how EMST bridges clusters.
4. Use a geometry library (e.g. `scipy.spatial.Delaunay`) to build fast DT edges.
5. Measure runtime difference: brute-force vs DT-based.

### Test Cases

| Points          | EMST                         | Total Length           |
|-----------------|------------------------------|------------------------|
| Square corners  | (0,0),(1,0),(1,1),(0,1)      | 3.0                    |
| Triangle        | (0,0),(2,0),(1, $\sqrt{3}$ ) | 4.0                    |
| Random 5 points | ,                            | Depends on coordinates |

### Complexity

- Brute-force:  $O(n^2 \log n)$
- Delaunay-based:  $O(n \log n)$
- Space:  $O(n)$

The Euclidean MST is the geometric minimal backbone, the shortest invisible thread that ties all points together in the plane.

## Section 36. Flows

### 351 Ford–Fulkerson

The Ford–Fulkerson method is the foundational algorithm for computing the maximum flow in a directed network. It views flow as water moving through a system of pipes, pushing flow from a source to a sink, respecting capacity limits and flow conservation.

## What Problem Are We Solving?

Given a flow network  $G = (V, E)$  with

- a source  $s$ ,
- a sink  $t$ ,
- and capacities  $c(u, v)$  on each edge,

we want to find a maximum flow  $f(u, v)$  such that:

1. Capacity constraint:

$$0 \leq f(u, v) \leq c(u, v)$$

2. Flow conservation:

$$\sum_{(u,v) \in E} f(u, v) = \sum_{(v,u) \in E} f(v, u) \quad \forall u \in V \setminus s, t$$

3. Maximize total flow:

$$\text{Maximize } |f| = \sum_{(s,v) \in E} f(s, v)$$

## How Does It Work (Plain Language)?

The algorithm repeatedly searches for augmenting paths, routes from  $s$  to  $t$  where additional flow can still be sent. Each iteration increases total flow until no augmenting path remains.

Step-by-step:

1. Initialize all flows to 0.

2. While an augmenting path  $P$  exists from  $s$  to  $t$  in the residual graph:

- Find the bottleneck capacity along  $P$ :

$$b = \min_{(u,v) \in P} (c(u, v) - f(u, v))$$

- Augment flow along  $P$  by  $b$ :

$$f(u, v) \leftarrow f(u, v) + b$$

$$f(v, u) \leftarrow f(v, u) - b$$

3. Update residual capacities and repeat.

4. When no path remains, current flow is maximum.

The residual graph represents remaining capacity, allowing backward edges to cancel flow if needed.

## Tiny Code

Python (DFS-based Augmentation)

```
from collections import defaultdict

def ford_fulkerson(capacity, s, t):
    n = len(capacity)
    flow = [[0]*n for _ in range(n)]

    def dfs(u, t, f, visited):
        if u == t:
            return f
        visited[u] = True
        for v in range(n):
            residual = capacity[u][v] - flow[u][v]
            if residual > 0 and not visited[v]:
                pushed = dfs(v, t, min(f, residual), visited)
                if pushed > 0:
                    flow[u][v] += pushed
                    flow[v][u] -= pushed
                    return pushed
        return 0

    max_flow = 0
    while True:
        visited = [False]*n
        pushed = dfs(s, t, float('inf'), visited)
        if pushed == 0:
            break
        max_flow += pushed
    return max_flow

capacity = [
    [0, 16, 13, 0, 0, 0],
    [0, 0, 10, 12, 0, 0],
    [0, 4, 0, 0, 14, 0],
    [0, 0, 9, 0, 0, 20],
    [0, 0, 0, 7, 0, 4],
    [0, 0, 0, 0, 0, 0]
]
print(ford_fulkerson(capacity, 0, 5))
```

Output:

23

## Why It Matters

- Introduces max-flow concept, a cornerstone of network optimization.
- Underpins algorithms for:
  - bipartite matching
  - minimum cut problems
  - circulations and network design
- Illustrates residual graphs and augmenting paths, used in many flow-based algorithms.

## A Gentle Proof (Why It Works)

Each augmentation increases total flow by a positive amount and maintains feasibility. Because total capacity is finite, the process eventually terminates.

By the Max-Flow Min-Cut Theorem:

$$|f^*| = \text{capacity of minimum } (S, T) \text{ cut}$$

So when no augmenting path exists, the flow is maximum.

If all capacities are integers, the algorithm converges in finite steps, since each augmentation adds at least 1 unit.

## Try It Yourself

1. Run on small networks, draw residual graph at each step.
2. Track augmenting paths and bottleneck edges.
3. Compare to Edmonds–Karp (BFS search).
4. Change capacities to fractional, observe potential infinite loops.
5. Use to solve bipartite matching (convert to flow network).

## Test Cases

| Graph                  | Max Flow          | Notes            |
|------------------------|-------------------|------------------|
| Simple 3-node          | 5                 | single augment   |
| Classic 6-node (above) | 23                | textbook example |
| Parallel edges         | Sum of capacities | additive         |

## Complexity

- Time:  $O(E \cdot |f^*|)$  for integer capacities
- Space:  $O(V^2)$
- Optimized (Edmonds–Karp):  $O(VE^2)$

Ford–Fulkerson builds the intuition of pushing flow through capacity-constrained paths, the heart of network optimization.

## 352 Edmonds–Karp

The Edmonds–Karp algorithm is a concrete, polynomial-time implementation of the Ford–Fulkerson method, where each augmenting path is chosen using Breadth-First Search (BFS). By always selecting the shortest path (in edge count) from source to sink, it guarantees efficient convergence.

### What Problem Are We Solving?

Given a directed flow network  $G = (V, E)$  with

- a source  $s$ ,
- a sink  $t$ ,
- and nonnegative capacities  $c(u, v)$ ,

we want to find a maximum flow  $f(u, v)$  that satisfies:

1. Capacity constraints:

$$0 \leq f(u, v) \leq c(u, v)$$

2. Flow conservation:

$$\sum_{(u,v) \in E} f(u, v) = \sum_{(v,u) \in E} f(v, u) \quad \forall u \in V \setminus s, t$$

3. Maximize total flow:

$$|f| = \sum_{(s,v) \in E} f(s, v)$$

The algorithm improves Ford–Fulkerson by enforcing shortest augmenting path order, which bounds the number of iterations.

## How Does It Work (Plain Language)?

Each iteration finds an augmenting path using BFS (so each edge in the path is part of the shortest route in terms of hops). Then we push the maximum possible flow through that path and update residual capacities.

Step-by-step:

1. Initialize  $f(u, v) = 0$  for all edges.
2. While there exists a path  $P$  from  $s$  to  $t$  in the residual graph (found by BFS):
  - Compute bottleneck capacity:

$$b = \min_{(u,v) \in P} (c(u,v) - f(u,v))$$

- Augment along path  $P$ :

$$f(u, v) \leftarrow f(u, v) + b$$

$$f(v, u) \leftarrow f(v, u) - b$$

3. Repeat until no augmenting path remains.

Residual graph is updated after each augmentation, including reverse edges to allow flow cancellation.

## Tiny Code

Python Implementation (BFS Augmentation)

```
from collections import deque

def bfs(capacity, flow, s, t):
    n = len(capacity)
    parent = [-1] * n
    parent[s] = s
    q = deque([s])
    while q:
        u = q.popleft()
        for v in range(n):
            residual = capacity[u][v] - flow[u][v]
```

```

        if residual > 0 and parent[v] == -1:
            parent[v] = u
            q.append(v)
            if v == t:
                return parent
    return None

def edmonds_karp(capacity, s, t):
    n = len(capacity)
    flow = [[0]*n for _ in range(n)]
    max_flow = 0

    while True:
        parent = bfs(capacity, flow, s, t)
        if not parent:
            break
        # find bottleneck
        v = t
        bottleneck = float('inf')
        while v != s:
            u = parent[v]
            bottleneck = min(bottleneck, capacity[u][v] - flow[u][v])
            v = u
        # augment flow
        v = t
        while v != s:
            u = parent[v]
            flow[u][v] += bottleneck
            flow[v][u] -= bottleneck
            v = u
        max_flow += bottleneck

    return max_flow

capacity = [
    [0, 16, 13, 0, 0, 0],
    [0, 0, 10, 12, 0, 0],
    [0, 4, 0, 0, 14, 0],
    [0, 0, 9, 0, 0, 20],
    [0, 0, 0, 7, 0, 4],
    [0, 0, 0, 0, 0, 0]
]

```

```
print(edmonds_karp(capacity, 0, 5))
```

Output:

23

## Why It Matters

- Ensures polynomial time termination.
- Demonstrates the power of BFS for finding shortest augmenting paths.
- Serves as the canonical maximum flow algorithm in theory and practice.
- Provides a clean proof of the Max-Flow Min-Cut Theorem.
- Foundation for more advanced methods (Dinic, Push–Relabel).

## A Gentle Proof (Why It Works)

Each augmentation uses a shortest path in terms of edges. After each augmentation, at least one edge on that path saturates (reaches full capacity).

Because every edge can only move from residual distance  $d$  to  $d + 2$  a limited number of times, the total number of augmentations is  $O(VE)$ .

Each BFS runs in  $O(E)$ , giving total runtime  $O(VE^2)$ .

The algorithm terminates when no path exists, i.e. the residual graph disconnects  $s$  and  $t$ , and by the Max-Flow Min-Cut theorem, the current flow is maximum.

## Try It Yourself

1. Run BFS after each iteration, visualize residual network.
2. Compare with Ford–Fulkerson (DFS), note fewer augmentations.
3. Modify capacities, see how path selection changes.
4. Implement path reconstruction and print augmenting paths.
5. Use to solve bipartite matching via flow transformation.

| Graph | Max Flow | Notes |
|-------|----------|-------|
|-------|----------|-------|

## Test Cases

| Graph          | Max Flow               | Notes                 |
|----------------|------------------------|-----------------------|
| Simple 3-node  | 5                      | BFS finds direct path |
| Classic 6-node | 23                     | textbook example      |
| Star network   | Sum of edge capacities | each edge unique path |

## Complexity

- Time:  $O(VE^2)$
- Space:  $O(V^2)$
- Augmentations:  $O(VE)$

Edmonds–Karp transforms Ford–Fulkerson into a predictable, efficient, and elegant BFS-based flow engine, ensuring progress, bounding iterations, and revealing the structure of optimal flow.

## 353 Dinic's Algorithm

Dinic's Algorithm (or Dinitz's algorithm) is a faster approach to computing maximum flow, improving upon Edmonds–Karp by introducing a level graph and sending blocking flows within it. It combines BFS (to layer the graph) with DFS (to find augmenting paths), achieving a strong polynomial bound.

### What Problem Are We Solving?

Given a directed flow network  $G = (V, E)$  with

- source  $s$ ,
- sink  $t$ ,
- capacities  $c(u, v)$  on each edge,

find the maximum flow  $f(u, v)$  such that:

1. Capacity constraint:

$$0 \leq f(u, v) \leq c(u, v)$$

2. Flow conservation:

$$\sum_{(u,v) \in E} f(u,v) = \sum_{(v,u) \in E} f(v,u) \quad \forall u \in V \setminus s, t$$

3. Maximize total flow:

$$|f| = \sum_{(s,v) \in E} f(s,v)$$

### How Does It Work (Plain Language)?

Dinic's Algorithm operates in phases. Each phase constructs a level graph using BFS, then pushes as much flow as possible within that layered structure using DFS. This approach ensures progress, each phase strictly increases the shortest path length from  $s$  to  $t$  in the residual graph.

Step-by-step:

1. Build Level Graph (BFS):

- Run BFS from  $s$ .
- Assign each vertex a level = distance from  $s$  in residual graph.
- Only edges  $(u, v)$  with  $level[v] = level[u] + 1$  are used.

2. Send Blocking Flow (DFS):

- Use DFS to push flow from  $s$  to  $t$  along level-respecting paths.
- Stop when no more flow can be sent (i.e., blocking flow).

3. Repeat:

- Rebuild level graph; continue until  $t$  is unreachable from  $s$ .

A blocking flow saturates at least one edge on every  $s-t$  path in the level graph, ensuring termination per phase.

### Tiny Code

Python Implementation (Adjacency List)

```

from collections import deque

class Dinic:
    def __init__(self, n):
        self.n = n
        self.adj = [[] for _ in range(n)]

    def add_edge(self, u, v, cap):
        self.adj[u].append([v, cap, len(self.adj[v])])
        self.adj[v].append([u, 0, len(self.adj[u]) - 1]) # reverse edge

    def bfs(self, s, t, level):
        q = deque([s])
        level[s] = 0
        while q:
            u = q.popleft()
            for v, cap, _ in self.adj[u]:
                if cap > 0 and level[v] < 0:
                    level[v] = level[u] + 1
                    q.append(v)
        return level[t] >= 0

    def dfs(self, u, t, flow, level, it):
        if u == t:
            return flow
        while it[u] < len(self.adj[u]):
            v, cap, rev = self.adj[u][it[u]]
            if cap > 0 and level[v] == level[u] + 1:
                pushed = self.dfs(v, t, min(flow, cap), level, it)
                if pushed > 0:
                    self.adj[u][it[u]][1] -= pushed
                    self.adj[v][rev][1] += pushed
                    return pushed
            it[u] += 1
        return 0

    def max_flow(self, s, t):
        flow = 0
        level = [-1] * self.n
        INF = float('inf')
        while self.bfs(s, t, level):
            it = [0] * self.n

```

```

    while True:
        pushed = self.dfs(s, t, INF, level, it)
        if pushed == 0:
            break
        flow += pushed
        level = [-1] * self.n
    return flow

# Example
dinic = Dinic(6)
edges = [
    (0,1,16),(0,2,13),(1,2,10),(2,1,4),(1,3,12),
    (3,2,9),(2,4,14),(4,3,7),(3,5,20),(4,5,4)
]
for u,v,c in edges:
    dinic.add_edge(u,v,c)
print(dinic.max_flow(0,5))

```

Output:

23

## Why It Matters

- Achieves  $O(V^2E)$  complexity (faster in practice).
- Exploits layering to avoid redundant augmentations.
- Basis for advanced flow algorithms like Dinic + scaling, Push–Relabel, and Blocking Flow variants.
- Common in competitive programming, network routing, and bipartite matching.

## A Gentle Proof (Why It Works)

Each phase increases the shortest distance from  $s$  to  $t$  in the residual graph. Because there are at most  $V - 1$  distinct distances, the algorithm runs for at most  $O(V)$  phases. Within each phase, we find a blocking flow, which can be computed in  $O(E)$  DFS calls.

Hence total runtime:

$$O(VE)$$

for unit networks (each edge capacity = 1), and in general,

$$O(V^2E)$$

When no augmenting path exists, residual graph disconnects  $s$  and  $t$ , so the current flow is maximum by the Max-Flow Min-Cut theorem.

### Try It Yourself

1. Compare BFS layers between phases, note increasing depth.
2. Visualize level graph and residual edges.
3. Test on bipartite graph, confirm match size = flow.
4. Modify to store flow per edge.
5. Add capacity scaling to speed up dense graphs.

### Test Cases

| Graph           | Max Flow               | Notes           |
|-----------------|------------------------|-----------------|
| 6-node sample   | 23                     | classic example |
| Unit network    | equals #disjoint paths |                 |
| Bipartite graph | equals max matching    |                 |

### Complexity

- Time:  $O(V^2 E)$  general,  $O(\sqrt{V}E)$  for unit capacities
- Space:  $O(V + E)$
- Augmentations per phase:  $O(E)$

Dinic's algorithm elegantly combines BFS-level layering with DFS-based flow pushing, a perfect synthesis of structure and greed, powering modern flow computations.

### 354 Push–Relabel

The Push–Relabel algorithm (also called Preflow–Push) takes a completely different view of the maximum flow problem. Instead of finding paths from the source to the sink, it locally pushes flow along edges and adjusts vertex heights (labels) to guide flow downhill toward the sink.

This approach is highly efficient in practice and forms the basis for many modern flow solvers.

## What Problem Are We Solving?

Given a directed network  $G = (V, E)$  with

- source  $s$ ,
- sink  $t$ ,
- capacity  $c(u, v)$ ,

we want a flow  $f(u, v)$  satisfying:

1. Capacity constraint:

$$0 \leq f(u, v) \leq c(u, v)$$

2. Flow conservation:

$$\sum_{(u,v) \in E} f(u, v) = \sum_{(v,u) \in E} f(v, u) \quad \forall u \neq s, t$$

3. Maximize total flow:

$$|f| = \sum_{(s,v) \in E} f(s, v)$$

## How Does It Work (Plain Language)?

Unlike Ford–Fulkerson, which finds augmenting paths, Push–Relabel maintains a preflow (where intermediate nodes may have excess flow) and fixes imbalances gradually.

Key concepts:

- Preflow: flow can temporarily violate conservation (nodes can have excess).
- Height (label): an integer guiding flow direction, flow only moves “downhill.”
- Push: send flow from  $u$  to  $v$  if possible.
- Relabel: when  $u$  is stuck, increase its height so flow can continue.

Step-by-step:

1. Initialize all  $f(u, v) = 0$ .
2. Set  $h(s) = |V|$  and push as much as possible from  $s$  to its neighbors.
3. While any vertex (other than  $s, t$ ) has excess flow, do:
  - Push: if  $(u, v)$  is admissible ( $h(u) = h(v) + 1$  and residual  $> 0$ ), send

$$\Delta = \min(e(u), c(u, v) - f(u, v))$$

- Relabel: if no admissible edge, set

$$h(u) = 1 + \min_{(u,v):c(u,v)-f(u,v)>0} h(v)$$

Repeat until all excess is at  $t$  or  $s$ .

### Tiny Code

Python Implementation (Simplified)

```
def push_relabel(capacity, s, t):
    n = len(capacity)
    flow = [[0]*n for _ in range(n)]
    excess = [0]*n
    height = [0]*n
    height[s] = n

    def push(u, v):
        delta = min(excess[u], capacity[u][v] - flow[u][v])
        flow[u][v] += delta
        flow[v][u] -= delta
        excess[u] -= delta
        excess[v] += delta

    def relabel(u):
        min_h = float('inf')
        for v in range(n):
            if capacity[u][v] - flow[u][v] > 0:
                min_h = min(min_h, height[v])
        if min_h < float('inf'):
            height[u] = min_h + 1

    def discharge(u):
        while excess[u] > 0:
            for v in range(n):
                if capacity[u][v] - flow[u][v] > 0 and height[u] == height[v] + 1:
                    push(u, v)
                    if excess[u] == 0:
                        break
            else:
                relabel(u)
```

```

# Initialize preflow
for v in range(n):
    if capacity[s][v] > 0:
        flow[s][v] = capacity[s][v]
        flow[v][s] = -capacity[s][v]
        excess[v] = capacity[s][v]
excess[s] = sum(capacity[s])

# Discharge active vertices
active = [i for i in range(n) if i != s and i != t]
p = 0
while p < len(active):
    u = active[p]
    old_height = height[u]
    discharge(u)
    if height[u] > old_height:
        active.insert(0, active.pop(p)) # move to front
        p = 0
    else:
        p += 1

return sum(flow[s])

capacity = [
    [0,16,13,0,0,0],
    [0,0,10,12,0,0],
    [0,4,0,0,14,0],
    [0,0,9,0,0,20],
    [0,0,0,7,0,4],
    [0,0,0,0,0,0]
]
print(push_relabel(capacity, 0, 5))

```

Output:

23

## Why It Matters

- Local view: No need for global augmenting paths.
- Highly parallelizable.

- Performs very well in dense graphs.
- Serves as the basis for highest-label and FIFO variants.
- Conceptually elegant: flow “falls downhill” guided by heights.

### A Gentle Proof (Why It Works)

1. The height invariant ensures no flow moves from lower to higher vertices.
2. Every push respects capacity and non-negativity.
3. Height always increases, ensuring termination.
4. When no vertex (except  $s$  and  $t$ ) has excess, all preflow constraints are satisfied, the preflow becomes a valid maximum flow.

By the Max-Flow Min-Cut Theorem, the final preflow’s value equals the capacity of the minimum cut.

### Try It Yourself

1. Track each vertex’s height and excess after each step.
2. Compare FIFO vs highest-label variants.
3. Use small networks to visualize flow movement.
4. Contrast with path-based algorithms (Ford–Fulkerson).
5. Add logging to observe relabel events.

### Test Cases

| Graph          | Max Flow             | Notes     |
|----------------|----------------------|-----------|
| Classic 6-node | 23                   | textbook  |
| Dense complete | high flow            | efficient |
| Sparse path    | same as Edmonds–Karp | similar   |

### Complexity

- Time (Generic):  $O(V^2E)$
- With FIFO / Highest-Label Heuristics:  $O(V^3)$  or better
- Space:  $O(V^2)$

Push–Relabel transforms max-flow into a local balancing act, pushing, relabeling, and equalizing pressure until equilibrium is achieved.

## 355 Capacity Scaling

The Capacity Scaling algorithm is a refined version of Ford–Fulkerson, designed to handle large edge capacities efficiently. Instead of augmenting arbitrarily, it focuses on high-capacity edges first, gradually refining the flow as the scaling parameter decreases.

This approach reduces the number of augmentations by focusing early on “big pipes” before worrying about smaller ones.

### What Problem Are We Solving?

Given a directed flow network  $G = (V, E)$  with

- source  $s$ ,
- sink  $t$ ,
- nonnegative capacities  $c(u, v)$ ,

we want to compute a maximum flow  $f(u, v)$  satisfying:

1. Capacity constraint:

$$0 \leq f(u, v) \leq c(u, v)$$

2. Flow conservation:

$$\sum_{(u,v) \in E} f(u, v) = \sum_{(v,u) \in E} f(v, u), \quad \forall u \neq s, t$$

3. Maximize:

$$|f| = \sum_{(s,v) \in E} f(s, v)$$

When capacities are large, standard augmenting-path methods can take many iterations. Capacity scaling reduces this by grouping edges by capacity magnitude.

### How Does It Work (Plain Language)?

We introduce a scaling parameter  $\Delta$ , starting from the highest power of 2 below the maximum capacity. We only consider edges with residual capacity  $\Delta$  during augmentations. Once no such path exists, halve  $\Delta$  and continue.

This ensures we push large flows early and refine later.

Step-by-step:

1. Initialize  $f(u, v) = 0$ .

2. Let

$$\Delta = 2^{\lfloor \log_2 C_{\max} \rfloor}$$

where  $C_{\max} = \max_{(u,v) \in E} c(u, v)$

3. While  $\Delta \geq 1$ :

- Build  $\Delta$ -residual graph: edges with residual capacity  $\geq \Delta$
- While an augmenting path exists in this graph:
  - Send flow equal to the bottleneck along that path
- Update  $\Delta \leftarrow \Delta/2$

4. Return total flow.

### Tiny Code

Python (DFS Augmentation with Scaling)

```
def dfs(u, t, f, visited, capacity, flow, delta):
    if u == t:
        return f
    visited[u] = True
    for v in range(len(capacity)):
        residual = capacity[u][v] - flow[u][v]
        if residual >= delta and not visited[v]:
            pushed = dfs(v, t, min(f, residual), visited, capacity, flow, delta)
            if pushed > 0:
                flow[u][v] += pushed
                flow[v][u] -= pushed
            return pushed
    return 0

def capacity_scaling(capacity, s, t):
    n = len(capacity)
    flow = [[0]*n for _ in range(n)]
    Cmax = max(max(row) for row in capacity)
    delta = 1
    while delta * 2 <= Cmax:
        delta *= 2

    max_flow = 0
    while delta >= 1:
```

```

while True:
    visited = [False]*n
    pushed = dfs(s, t, float('inf'), visited, capacity, flow, delta)
    if pushed == 0:
        break
    max_flow += pushed
    delta //= 2
return max_flow

capacity = [
    [0, 16, 13, 0, 0, 0],
    [0, 0, 10, 12, 0, 0],
    [0, 4, 0, 0, 14, 0],
    [0, 0, 9, 0, 0, 20],
    [0, 0, 0, 7, 0, 4],
    [0, 0, 0, 0, 0, 0]
]
print(capacity_scaling(capacity, 0, 5))

```

Output:

23

## Why It Matters

- Reduces augmentations compared to plain Ford–Fulkerson.
- Focuses on big pushes first, improving convergence.
- Demonstrates the power of scaling ideas, a recurring optimization in algorithm design.
- Serves as a stepping stone to cost scaling and capacity scaling in min-cost flow.

## A Gentle Proof (Why It Works)

At each scaling phase  $\Delta$ ,

- Each augmenting path increases flow by at least  $\Delta$ .
- Total flow value is at most  $|f^*|$ .
- Therefore, each phase performs at most  $O(|f^*|/\Delta)$  augmentations.

Since  $\Delta$  halves each phase, the total number of augmentations is

$$O(E \log C_{\max})$$

Each path search takes  $O(E)$ , so

$$T = O(E^2 \log C_{\max})$$

Termination occurs when  $\Delta = 1$  and no path remains, meaning max flow reached.

### Try It Yourself

1. Compare augmentation order with Ford–Fulkerson.
2. Track  $\Delta$  values and residual graphs per phase.
3. Observe faster convergence for large capacities.
4. Combine with BFS to mimic Edmonds–Karp structure.
5. Visualize flow accumulation at each scale.

### Test Cases

| Graph                | Max Flow                  | Notes            |
|----------------------|---------------------------|------------------|
| Classic 6-node       | 23                        | textbook example |
| Large capacity edges | same flow, fewer steps    | scaling helps    |
| Unit capacities      | behaves like Edmonds–Karp | small gains      |

### Complexity

- Time:  $O(E^2 \log C_{\max})$
- Space:  $O(V^2)$
- Augmentations:  $O(E \log C_{\max})$

Capacity scaling embodies a simple but powerful idea: solve coarse first, refine later, push the big flows early, and polish at the end.

## 356 Cost Scaling

The Cost Scaling algorithm tackles the Minimum-Cost Maximum Flow (MCMF) problem by applying a scaling technique not to capacities but to edge costs. It gradually refines the precision of reduced costs, maintaining -optimality throughout, and converges to the true optimal flow.

This approach is both theoretically elegant and practically efficient, forming the foundation for high-performance network optimization solvers.

### What Problem Are We Solving?

Given a directed network  $G = (V, E)$  with

- capacity  $c(u, v) \geq 0$ ,
- cost (per unit flow)  $w(u, v)$ ,
- source  $s$ , and sink  $t$ ,

we want to send the maximum flow from  $s$  to  $t$  at minimum total cost.

We minimize:

$$\text{Cost}(f) = \sum_{(u,v) \in E} f(u, v), w(u, v)$$

subject to:

- Capacity constraint:  $0 \leq f(u, v) \leq c(u, v)$
- Conservation:  $\sum_v f(u, v) = \sum_v f(v, u)$  for all  $u \neq s, t$

### How Does It Work (Plain Language)?

Cost scaling uses successive refinements of reduced costs, ensuring flows are -optimal at each phase.

An -optimal flow satisfies:

$$c_p(u, v) = w(u, v) + \pi(u) - \pi(v) \geq -\varepsilon$$

for all residual edges  $(u, v)$ , where  $\pi$  is a potential function.

The algorithm begins with a large  $\varepsilon$  (often  $C_{\max}$  or  $W_{\max}$ ) and reduces it geometrically (e.g.  $\varepsilon \leftarrow \varepsilon/2$ ). During each phase, pushes are allowed only along admissible edges ( $c_p(u, v) < 0$ ), maintaining -optimality.

Step-by-step:

1. Initialize preflow (push as much as possible from  $s$ ).
2. Set  $\varepsilon = C_{\max}$ .
3. While  $\varepsilon \geq 1$ :
  - Maintain -optimality: adjust flows and potentials.
  - For each vertex with excess,
    - Push flow along admissible edges ( $c_p(u, v) < 0$ ).
    - If stuck, relabel by increasing  $\pi(u)$  (lowering reduced costs).
  - Halve  $\varepsilon$ .
4. When  $\varepsilon < 1$ , solution is optimal.

### Tiny Code (Simplified Skeleton)

Below is a conceptual outline (not a full implementation) for educational clarity:

```
def cost_scaling_mcmf(V, edges, s, t):
    INF = float('inf')
    adj = [[] for _ in range(V)]
    for u, v, cap, cost in edges:
        adj[u].append([v, cap, cost, len(adj[v])])
        adj[v].append([u, 0, -cost, len(adj[u]) - 1])

    pi = [0]*V # potentials
    excess = [0]*V
    excess[s] = INF

    def reduced_cost(u, v, cost):
        return cost + pi[u] - pi[v]

    epsilon = max(abs(c) for _, _, _, c in edges)
    while epsilon >= 1:
        active = [i for i in range(V) if excess[i] > 0 and i != s and i != t]
        while active:
            u = active.pop()
            for e in adj[u]:
                v, cap, cost, rev = e
                rc = reduced_cost(u, v, cost)
                if cap > 0 and rc < 0:
                    delta = min(cap, excess[u])
                    excess[v] += delta
                    excess[u] -= delta
                    pi[v] += delta
                    pi[u] -= delta
                    cap -= delta
                    if cap == 0:
                        active.append(v)
                    if excess[v] == 0:
                        excess[v] = INF
                    if v == t:
                        return excess[t]
                    if excess[v] > 0:
                        active.append(v)
            epsilon = max(abs(excess[i]) for i in range(V))
```

```

        e[1] -= delta
        adj[v][rev][1] += delta
        excess[u] -= delta
        excess[v] += delta
        if v not in (s, t) and excess[v] == delta:
            active.append(v)
        if excess[u] > 0:
            pi[u] += epsilon
    epsilon /= 2

total_cost = 0
for u in range(V):
    for v, cap, cost, rev in adj[u]:
        if cost > 0:
            total_cost += cost * (adj[v][rev][1])
return total_cost

```

## Why It Matters

- Avoids expensive shortest-path searches at each step.
- Ensures strong polynomial bounds using  $\epsilon$ -optimality.
- Suitable for dense graphs and large integer costs.
- Forms the backbone of network simplex and scalable MCMF solvers.

## A Gentle Proof (Why It Works)

1.  $\epsilon$ -optimality: ensures approximate optimality at each scaling phase.
2. Scaling: reduces costs geometrically, converging to exact optimality.
3. Termination: when  $\epsilon < 1$ , all reduced costs are nonnegative, the flow is optimal.

Each push respects capacity and admissibility; each relabel decreases  $\pi(u)$ , guaranteeing progress.

## Try It Yourself

1. Compare to Successive Shortest Path algorithm, note fewer path searches.
2. Track how potentials  $\pi(u)$  evolve over phases.
3. Visualize  $\epsilon$ -layers of admissible edges.
4. Experiment with cost magnitudes, larger costs benefit more from scaling.
5. Observe convergence as  $\epsilon$  halves each phase.

## Test Cases

| Graph           | Max Flow | Min Cost  | Notes         |
|-----------------|----------|-----------|---------------|
| Simple 3-node   | 5        | 10        | small example |
| Classic network | 23       | 42        | cost-scaled   |
| Dense graph     | varies   | efficient |               |

## Complexity

- Time:  $O(E \log C_{\max}(V + E))$  (depends on cost range)
- Space:  $O(V + E)$
- Phases:  $O(\log C_{\max})$

Cost scaling showcases the precision-refinement paradigm, start coarse, end exact. It's a masterclass in combining scaling, potential functions, and local admissibility to achieve globally optimal flows.

## 357 Min-Cost Max-Flow (Bellman-Ford)

The Min-Cost Max-Flow (MCMF) algorithm with Bellman-Ford is a cornerstone method for solving flow problems where both capacity and cost constraints matter. It repeatedly augments flow along the shortest-cost paths, ensuring every unit of flow moves as cheaply as possible.

This version uses Bellman-Ford to handle negative edge costs, ensuring correctness even when cycles reduce total cost.

## What Problem Are We Solving?

Given a directed graph  $G = (V, E)$  with:

- Capacity  $c(u, v)$
- Cost per unit flow  $w(u, v)$
- Source  $s$  and sink  $t$

Find a flow  $f(u, v)$  that:

1. Respects capacities:  $0 \leq f(u, v) \leq c(u, v)$
2. Conserves flow:  $\sum_v f(u, v) = \sum_v f(v, u)$  for all  $u \neq s, t$
3. Maximizes total flow, while
4. Minimizing total cost:

$$\text{Cost}(f) = \sum_{(u,v) \in E} f(u, v), w(u, v)$$

## How Does It Work (Plain Language)?

The algorithm iteratively finds augmenting paths from  $s$  to  $t$  using shortest path search by cost (not distance).

At each iteration:

1. Run Bellman-Ford on the residual graph to find the shortest-cost path.
2. Determine the bottleneck capacity  $\delta$  on that path.
3. Augment flow along that path by  $\delta$ .
4. Update residual capacities and reverse edges.
5. Repeat until no more augmenting paths exist.

Because Bellman-Ford supports negative costs, this algorithm correctly handles graphs where cost reductions occur via cycles.

## Tiny Code (C-like Pseudocode)

```
struct Edge { int v, cap, cost, rev; };
vector<Edge> adj[V];
int dist[V], parent[V], parent_edge[V];

bool bellman_ford(int s, int t, int V) {
    fill(dist, dist+V, INF);
    dist[s] = 0;
    bool updated = true;
    for (int i = 0; i < V-1 && updated; i++) {
        updated = false;
        for (int u = 0; u < V; u++) {
            if (dist[u] == INF) continue;
            for (int k = 0; k < adj[u].size(); k++) {
                auto &e = adj[u][k];
                if (e.cap > 0 && dist[e.v] > dist[u] + e.cost) {
                    dist[e.v] = dist[u] + e.cost;
                    parent[e.v] = u;
                    parent_edge[e.v] = k;
                    updated = true;
                }
            }
        }
    }
    return dist[t] < INF;
}
```

```

}

pair<int,int> min_cost_max_flow(int s, int t, int V) {
    int flow = 0, cost = 0;
    while (bellman_ford(s, t, V)) {
        int f = INF;
        for (int v = t; v != s; v = parent[v]) {
            int u = parent[v];
            auto &e = adj[u][parent_edge[v]];
            f = min(f, e.cap);
        }
        for (int v = t; v != s; v = parent[v]) {
            int u = parent[v];
            auto &e = adj[u][parent_edge[v]];
            e.cap -= f;
            adj[v][e.rev].cap += f;
            cost += f * e.cost;
        }
        flow += f;
    }
    return {flow, cost};
}

```

## Why It Matters

- Handles negative edge costs safely.
- Guaranteed optimality if no negative cycles exist.
- Works well for small to medium graphs.
- A foundation for more efficient variants (e.g. SPFA, Dijkstra with potentials).

It's the go-to teaching implementation for min-cost flow, balancing clarity and correctness.

## A Gentle Proof (Why It Works)

Each iteration finds a shortest-cost augmenting path using Bellman-Ford. Because edge costs are non-decreasing over augmentations, the algorithm converges to a globally optimal flow.

Each augmentation:

- Increases flow.
- Does not introduce cheaper paths later (monotonicity).
- Terminates when no more augmenting paths exist.

Thus, the resulting flow is both maximal and minimum cost.

### Try It Yourself

1. Draw a simple graph with 4 nodes and a negative edge.
2. Trace residual updates after each augmentation.
3. Compare results with a naive greedy path selection.
4. Replace Bellman-Ford with Dijkstra + potentials to improve speed.
5. Visualize residual capacity evolution.

### Test Cases

| Graph                                               | Max Flow | Min Cost | Notes                  |
|-----------------------------------------------------|----------|----------|------------------------|
| Chain $s \rightarrow a \rightarrow b \rightarrow t$ | 5        | 10       | Simple path            |
| Graph with negative cycle                           | Invalid  | -        | Must detect            |
| Grid-like network                                   | 12       | 24       | Multiple augmentations |

### Complexity

- Time:  $O(F \cdot V \cdot E)$ , where  $F$  is total flow sent.
- Space:  $O(V + E)$

For small graphs or graphs with negative weights, Bellman-Ford MCMF is the most robust and straightforward method, clear, reliable, and foundational.

## 358 Min-Cost Max-Flow (SPFA)

The Min-Cost Max-Flow algorithm using SPFA (Shortest Path Faster Algorithm) is an optimization of the Bellman–Ford approach. It leverages a queue-based relaxation method to find the shortest-cost augmenting paths more efficiently in practice, especially on sparse graphs or when negative edges are rare.

### What Problem Are We Solving?

We are solving the minimum-cost maximum-flow problem: Find a flow  $f$  in a directed graph  $G = (V, E)$  with:

- Capacity  $c(u, v) \geq 0$
- Cost per unit flow  $w(u, v)$

- Source  $s$ , Sink  $t$

Subject to:

1. Capacity constraint:  $0 \leq f(u, v) \leq c(u, v)$
2. Conservation of flow:  $\sum_v f(u, v) = \sum_v f(v, u)$  for all  $u \neq s, t$
3. Objective:

$$\min \text{Cost}(f) = \sum_{(u,v) \in E} f(u, v), w(u, v)$$

We seek a maximum flow from  $s$  to  $t$  that incurs the minimum total cost.

### How Does It Work (Plain Language)?

SPFA finds shortest paths by cost more efficiently than Bellman–Ford, using a queue to relax only the vertices that can still improve distances.

In each iteration:

1. Run SPFA on the residual graph to find the shortest-cost path.
2. Compute the bottleneck flow  $\delta$  on that path.
3. Augment  $\delta$  units of flow along the path.
4. Update residual capacities and reverse edges.
5. Repeat until no augmenting path remains.

SPFA dynamically manages which nodes are in the queue, making it faster on average than full Bellman–Ford (though worst-case still similar).

### Tiny Code (C-like Pseudocode)

```

struct Edge { int v, cap, cost, rev; };
vector<Edge> adj[V];
int dist[V], parent[V], parent_edge[V];
bool in_queue[V];

bool spfa(int s, int t, int V) {
    fill(dist, dist+V, INF);
    fill(in_queue, in_queue+V, false);
    queue<int> q;
    dist[s] = 0;
    q.push(s);
    in_queue[s] = true;
}

```

```

while (!q.empty()) {
    int u = q.front(); q.pop();
    in_queue[u] = false;
    for (int k = 0; k < adj[u].size(); k++) {
        auto &e = adj[u][k];
        if (e.cap > 0 && dist[e.v] > dist[u] + e.cost) {
            dist[e.v] = dist[u] + e.cost;
            parent[e.v] = u;
            parent_edge[e.v] = k;
            if (!in_queue[e.v]) {
                q.push(e.v);
                in_queue[e.v] = true;
            }
        }
    }
}
return dist[t] < INF;
}

pair<int,int> min_cost_max_flow(int s, int t, int V) {
    int flow = 0, cost = 0;
    while (spfa(s, t, V)) {
        int f = INF;
        for (int v = t; v != s; v = parent[v]) {
            int u = parent[v];
            auto &e = adj[u][parent_edge[v]];
            f = min(f, e.cap);
        }
        for (int v = t; v != s; v = parent[v]) {
            int u = parent[v];
            auto &e = adj[u][parent_edge[v]];
            e.cap -= f;
            adj[v][e.rev].cap += f;
            cost += f * e.cost;
        }
        flow += f;
    }
    return {flow, cost};
}

```

## Why It Matters

- Practical speedup over Bellman–Ford, especially in sparse networks.
- Handles negative edges safely (no negative cycles).
- Widely used in competitive programming and network optimization.
- Easier to implement than Dijkstra-based variants.

SPFA combines the simplicity of Bellman–Ford with real-world efficiency, often reducing redundant relaxations dramatically.

## A Gentle Proof (Why It Works)

Each SPFA run computes a shortest-cost augmenting path. Augmenting along that path ensures:

- The flow remains feasible,
- The cost strictly decreases, and
- The process terminates after a finite number of augmentations (bounded by total flow).

Because SPFA always respects shortest-path distances, the final flow is cost-optimal.

## Try It Yourself

1. Replace Bellman–Ford in your MCMF with SPFA.
2. Compare runtime on sparse vs dense graphs.
3. Add an edge with negative cost and verify correct behavior.
4. Visualize queue contents after each relaxation.
5. Measure how many times each vertex is processed.

## Test Cases

| Graph       | Max Flow | Min Cost | Notes                   |
|-------------|----------|----------|-------------------------|
| Chain graph | 4        | 12       | Simple                  |
| Sparse DAG  | 10       | 25       | SPFA efficient          |
| Dense graph | 15       | 40       | Similar to Bellman–Ford |

## Complexity

- Time:  $O(F \cdot E)$  on average,  $O(F \cdot V \cdot E)$  worst-case
- Space:  $O(V + E)$

SPFA-based MCMF blends clarity, practical efficiency, and negative-cost support, making it a favorite for real-world min-cost flow implementations.

## 359 Circulation with Demands

The Circulation with Demands problem is a generalization of the classic flow and min-cost flow problems. Instead of having just a single source and sink, every node can demand or supply a certain amount of flow. Your goal is to find a feasible circulation—a flow that satisfies all node demands and capacity limits.

This formulation unifies network flow, feasibility checking, and optimization under one elegant model.

### What Problem Are We Solving?

Given a directed graph  $G = (V, E)$ , each edge  $(u, v)$  has:

- Lower bound  $l(u, v)$
- Upper bound (capacity)  $c(u, v)$
- Cost  $w(u, v)$

Each vertex  $v$  may have a demand  $b(v)$  (positive means it needs inflow, negative means it provides outflow).

We want to find a circulation (a flow satisfying  $f(u, v)$  for all edges) such that:

1. Capacity constraints:

$$l(u, v) \leq f(u, v) \leq c(u, v)$$

2. Flow conservation:

$$\sum_{(u,v) \in E} f(u, v) - \sum_{(v,u) \in E} f(v, u) = b(v)$$

3. Optional cost minimization:

$$\min \sum_{(u,v) \in E} f(u, v), w(u, v)$$

A feasible circulation exists if all demands can be met simultaneously.

## How Does It Work (Plain Language)?

The trick is to transform this into a standard min-cost flow problem with a super-source and super-sink.

Step-by-step transformation:

1. For each edge  $(u, v)$  with lower bound  $l(u, v)$ :

- Reduce the capacity to  $(c(u, v) - l(u, v))$ .
- Subtract  $l(u, v)$  from the demands:

$$b(u) -= l(u, v), \quad b(v) += l(u, v)$$

2. Add a super-source  $S$  and super-sink  $T$ :

- For each node  $v$ :
  - If  $b(v) > 0$ , add edge  $(S, v)$  with capacity  $b(v)$ .
  - If  $b(v) < 0$ , add edge  $(v, T)$  with capacity  $-b(v)$ .

3. Solve Max Flow (or Min-Cost Flow) from  $S$  to  $T$ .

4. If the max flow = total demand, a feasible circulation exists. Otherwise, no circulation satisfies all constraints.

## Tiny Code (Simplified Example)

```
def circulation_with_demands(V, edges, demand):  
    # edges: (u, v, lower, cap, cost)  
    # demand: list of node demands b(v)  
  
    adj = [[] for _ in range(V + 2)]  
    S, T = V, V + 1  
    b = demand[:]  
  
    for u, v, low, cap, cost in edges:  
        cap -= low  
        b[u] -= low  
        b[v] += low  
        adj[u].append((v, cap, cost))  
        adj[v].append((u, 0, -cost))  
  
    total_demand = 0
```

```

for i in range(V):
    if b[i] > 0:
        adj[S].append((i, b[i], 0))
        total_demand += b[i]
    elif b[i] < 0:
        adj[i].append((T, -b[i], 0))

flow, cost = min_cost_max_flow(adj, S, T)
if flow == total_demand:
    print("Feasible circulation found")
else:
    print("No feasible circulation")

```

## Why It Matters

- Unifies multiple problems: Many flow formulations—like assignment, transportation, and scheduling—can be expressed as circulations with demands.
- Handles lower bounds elegantly: Standard flows can't directly enforce  $f(u, v) \geq l(u, v)$ . Circulations fix that.
- Foundation for advanced models: Cost constraints, multi-commodity flows, and balance equations all build on this.

## A Gentle Proof (Why It Works)

By enforcing lower bounds, we adjust the net balance of each vertex. After normalization, the system becomes equivalent to finding a flow from super-source to super-sink satisfying all balances.

If such a flow exists, we can reconstruct the original flow:

$$f'(u, v) = f(u, v) + l(u, v)$$

which satisfies all original constraints.

## Try It Yourself

1. Model a supply-demand network (e.g., factories → warehouses → stores).
2. Add lower bounds to enforce minimum delivery.
3. Introduce node demands to balance supply/consumption.
4. Solve with min-cost flow and verify circulation.
5. Remove cost to just check feasibility.

## Test Cases

| Network                         | Result     | Notes                |
|---------------------------------|------------|----------------------|
| Balanced 3-node                 | Feasible   | Simple supply-demand |
| Lower bounds exceed total       | Infeasible | No solution          |
| Mixed positive/negative demands | Feasible   | Requires adjustment  |

## Complexity

- Time: depends on flow solver used ( $O(E^2V)$  for Bellman–Ford variant)
- Space:  $O(V + E)$

Circulation with demands gives a universal framework: any linear flow constraint can be encoded, checked, and optimized—turning balance equations into solvable graph problems.

## 360 Successive Shortest Path

The Successive Shortest Path (SSP) algorithm is a clean and intuitive way to solve the Minimum-Cost Maximum-Flow (MCMF) problem. It builds the optimal flow incrementally, one shortest path at a time, always sending flow along the cheapest available route until capacity or balance constraints stop it.

### What Problem Are We Solving?

Given a directed graph  $G = (V, E)$  with:

- Capacity:  $c(u, v)$
- Cost per unit flow:  $w(u, v)$
- Source:  $s$  and sink:  $t$

We want to send the maximum flow from  $s$  to  $t$  with minimum total cost:

$$\min \sum_{(u,v) \in E} f(u, v), w(u, v)$$

subject to:

- $0 \leq f(u, v) \leq c(u, v)$
- Flow conservation at each vertex except  $s, t$

## How Does It Work (Plain Language)?

SSP proceeds by repeatedly finding the shortest-cost augmenting path from  $s$  to  $t$  in the residual graph (cost as weight). It then pushes as much flow as possible along that path. Residual edges track available capacity (forward and backward).

Algorithm Steps:

1. Initialize flow  $f(u, v) = 0$  for all edges.
2. Build residual graph with edge costs  $w(u, v)$ .
3. While there exists a path  $P$  from  $s$  to  $t$ :
  - Find shortest path  $P$  by cost (using Dijkstra or Bellman–Ford).
  - Compute bottleneck capacity  $\delta = \min_{(u,v) \in P} c_f(u, v)$ .
  - Augment flow along  $P$ :

$$f(u, v) += \delta, \quad f(v, u) -= \delta$$

- Update residual capacities and costs.
4. Repeat until no augmenting path remains.
  5. Resulting  $f$  is the min-cost max-flow.

If negative edges exist, use Bellman–Ford; otherwise Dijkstra with potentials for efficiency.

## Tiny Code (Simplified Python)

```
from heapq import heappush, heappop

def successive_shortest_path(V, edges, s, t):
    adj = [[] for _ in range(V)]
    for u, v, cap, cost in edges:
        adj[u].append([v, cap, cost, len(adj[v])])
        adj[v].append([u, 0, -cost, len(adj[u]) - 1])

    INF = 109
    pi = [0]*V # potentials for reduced cost
    flow = cost = 0

    while True:
        dist = [INF]*V
```

```

parent = [-1]*V
parent_edge = [-1]*V
dist[s] = 0
pq = [(0, s)]
while pq:
    d, u = heappop(pq)
    if d > dist[u]: continue
    for i, (v, cap, w, rev) in enumerate(adj[u]):
        if cap > 0 and dist[v] > dist[u] + w + pi[u] - pi[v]:
            dist[v] = dist[u] + w + pi[u] - pi[v]
            parent[v] = u
            parent_edge[v] = i
            heappush(pq, (dist[v], v))
if dist[t] == INF:
    break
for v in range(V):
    if dist[v] < INF:
        pi[v] += dist[v]
f = INF
v = t
while v != s:
    u = parent[v]
    e = adj[u][parent_edge[v]]
    f = min(f, e[1])
    v = u
v = t
while v != s:
    u = parent[v]
    i = parent_edge[v]
    e = adj[u][i]
    e[1] -= f
    adj[v][e[3]][1] += f
    cost += f * e[2]
    v = u
flow += f
return flow, cost

```

## Why It Matters

- Simple and clear logic, augment along cheapest path each time.
- Optimal solution for min-cost max-flow when no negative cycles exist.

- Can handle large graphs efficiently with reduced costs and potentials.
- Forms the basis for cost-scaling and network simplex methods.

## A Gentle Proof (Why It Works)

Each augmentation moves flow along a shortest path (minimum reduced cost). Reduced costs ensure no negative cycles arise, so each augmentation preserves optimality. Once no augmenting path exists, all reduced costs are nonnegative and  $f$  is optimal.

Potentials  $\pi(v)$  guarantee Dijkstra finds true shortest paths in transformed graph:

$$w'(u, v) = w(u, v) + \pi(u) - \pi(v)$$

maintaining equivalence and non-negativity.

## Try It Yourself

1. Use a simple 4-node network and trace each path's cost and flow.
2. Compare augmentations using Bellman–Ford vs Dijkstra.
3. Add negative edge costs and test with potential adjustments.
4. Visualize residual graphs after each iteration.
5. Measure cost convergence per iteration.

## Test Cases

| Graph              | Max Flow | Min Cost | Notes                |
|--------------------|----------|----------|----------------------|
| Simple 3-node      | 5        | 10       | one path             |
| Two parallel paths | 10       | 15       | chooses cheapest     |
| With negative edge | 7        | 5        | potentials fix costs |

## Complexity

- Time:  $O(F \cdot E \log V)$  using Dijkstra + potentials
- Space:  $O(V + E)$

The successive shortest path algorithm is a workhorse for costed flow problems: easy to code, provably correct, and efficient with the right data structures.

## Section 37. Cuts

### 361 Stoer–Wagner Minimum Cut

The Stoer–Wagner Minimum Cut algorithm finds the global minimum cut in an undirected weighted graph. A *cut* is a partition of vertices into two non-empty sets; its weight is the sum of edges crossing the partition. The algorithm efficiently discovers the cut with minimum total edge weight, using repeated maximum adjacency searches.

#### What Problem Are We Solving?

Given an undirected graph  $G = (V, E)$  with non-negative weights  $w(u, v)$ , find a cut  $(S, V \setminus S)$  such that:

$$\text{cut}(S) = \sum_{u \in S, v \in V \setminus S} w(u, v)$$

is minimized over all nontrivial partitions  $S \subset V$ .

This is called the global minimum cut, distinct from the s–t minimum cut, which fixes two endpoints.

#### How Does It Work (Plain Language)?

The algorithm iteratively merges vertices while keeping track of the tightest cut found along the way.

At each phase:

1. Pick an arbitrary starting vertex.
2. Grow a set  $A$  by repeatedly adding the most strongly connected vertex (highest weight to  $A$ ).
3. Continue until all vertices are added.
4. The last added vertex  $t$  and the second last  $s$  define a cut  $(A \setminus t, t)$ .
5. Record the cut weight, it's a candidate for the minimum cut.
6. Merge  $s$  and  $t$  into a single vertex.
7. Repeat until only one vertex remains.

The smallest recorded cut across all phases is the global minimum cut.

## Step-by-Step Example

Suppose we have 4 vertices  $A, B, C, D$ . Each phase:

1. Start with  $A$ .
2. Add vertex most connected to  $A$ .
3. Continue until only one vertex remains.
4. Record cut weight each time the last vertex is added.
5. Merge last two vertices, repeat.

After all merges, the lightest cut weight is the minimum cut value.

## Tiny Code (Python)

```
def stoer_wagner_min_cut(V, weight):
    n = V
    best = float('inf')
    vertices = list(range(n))

    while n > 1:
        used = [False] * n
        weights = [0] * n
        prev = -1
        for i in range(n):
            # select most connected vertex not yet in A
            sel = -1
            for j in range(n):
                if not used[j] and (sel == -1 or weights[j] > weights[sel]):
                    sel = j
            used[sel] = True
            if i == n - 1:
                # last vertex added, record cut
                best = min(best, weights[sel])
                # merge prev and sel
                if prev != -1:
                    for j in range(n):
                        weight[prev][j] += weight[sel][j]
                        weight[j][prev] += weight[j][sel]
                    vertices.pop(sel)
                    weight.pop(sel)
                    for row in weight:
```

```

        row.pop(sel)
        n -= 1
        break
    prev = sel
    for j in range(n):
        if not used[j]:
            weights[j] += weight[sel][j]
    return best

```

## Why It Matters

- Finds the global min cut without fixing  $s, t$ .
- Works directly on weighted undirected graphs.
- Requires no flow computation.
- Simpler and faster than  $O(VE \log V)$  max-flow min-cut for global cut problems.

It's one of the few exact polynomial-time algorithms for global min cut in weighted graphs.

## A Gentle Proof (Why It Works)

Each phase finds a minimum  $s-t$  cut separating the last vertex  $t$  from the rest. The merging process maintains equivalence, merging does not destroy the optimality of remaining cuts. The lightest phase cut corresponds to the global minimum.

By induction over merging steps, the algorithm explores all essential cuts.

## Try It Yourself

1. Create a triangle graph with different edge weights.
2. Trace  $A-B-C$  order additions and record cut weights.
3. Merge last two vertices, reduce matrix, repeat.
4. Verify cut corresponds to smallest crossing weight.
5. Compare with max-flow min-cut for validation.

## Test Cases

| Graph                  | Minimum Cut | Notes                    |
|------------------------|-------------|--------------------------|
| Triangle $w = 1, 2, 3$ | 3           | Removes edge of weight 3 |
| Square equal weights   | 2           | Any two opposite sides   |

| Graph             | Minimum Cut       | Notes      |
|-------------------|-------------------|------------|
| Weighted complete | smallest edge sum | Dense test |

## Complexity

- Time:  $O(V^3)$  using adjacency matrix
- Space:  $O(V^2)$

Faster variants exist using adjacency lists and priority queues ( $O(VE + V^2 \log V)$ ).

The Stoer–Wagner algorithm shows that global connectivity fragility can be measured efficiently, one cut at a time, through pure merging insight.

## 362 Karger's Randomized Cut

The Karger's Algorithm is a beautifully simple randomized algorithm to find the global minimum cut of an undirected graph. Instead of deterministically exploring all partitions, it contracts edges randomly, shrinking the graph until only two supernodes remain. The sum of edges between them is (with high probability) the minimum cut.

### What Problem Are We Solving?

Given an undirected, unweighted (or weighted) graph  $G = (V, E)$ , a cut is a partition of  $V$  into two disjoint subsets  $(S, V \setminus S)$ . The cut size is the total number (or weight) of edges crossing the partition.

We want to find the global minimum cut:

$$\min_{S \subset V, S \neq \emptyset, S \neq V} \text{cut}(S)$$

Karger's algorithm gives a probabilistic guarantee of finding the exact minimum cut.

### How Does It Work (Plain Language)?

It's almost magical in its simplicity:

1. While there are more than 2 vertices:

- Pick a random edge  $(u, v)$ .
- Contract it, merge  $u$  and  $v$  into a single vertex.

- Remove self-loops.
2. When only two vertices remain,
- The edges between them form a cut.

Repeat the process multiple times to increase success probability.

Each run finds the true minimum cut with probability at least

$$\frac{2}{n(n-1)}$$

By repeating  $O(n^2 \log n)$  times, the probability of missing it becomes negligible.

### Example

Consider a triangle graph  $(A, B, C)$ :

- Randomly pick one edge, say  $(A, B)$ , contract into supernode  $(AB)$ .
- Now edges:  $(AB, C)$  with multiplicity 2.
- Only 2 nodes remain, cut weight = 2, the min cut.

### Tiny Code (Python)

```
import random
import copy

def karger_min_cut(graph):
    # graph: adjacency list {u: [v1, v2, ...]}
    vertices = list(graph.keys())
    edges = []
    for u in graph:
        for v in graph[u]:
            if u < v:
                edges.append((u, v))

    g = copy.deepcopy(graph)
    while len(g) > 2:
        u, v = random.choice(edges)
        # merge v into u
        g[u].extend(g[v])
        del g[v]
```

```

for w in g[v]:
    g[w] = [u if x == v else x for x in g[w]]
del g[v]
# remove self-loops
g[u] = [x for x in g[u] if x != u]
edges = []
for x in g:
    for y in g[x]:
        if x < y:
            edges.append((x, y))
# any remaining edge list gives cut size
return len(list(g.values())[0])

```

## Why It Matters

- Elegantly simple yet provably correct with probability.
- No flows, no complex data structures.
- Excellent pedagogical algorithm for randomized reasoning.
- Forms base for improved variants (Karger–Stein, randomized contraction).

Karger's approach demonstrates the power of randomization, minimal logic, maximal insight.

## A Gentle Proof (Why It Works)

At each step, contracting a non-min-cut edge does not destroy the min cut. Since there are  $O(n^2)$  edges and at least  $n - 2$  contractions, probability that we never contract a min-cut edge is:

$$P = \prod_{i=0}^{n-3} \frac{k_i}{m_i} \geq \frac{2}{n(n-1)}$$

where  $k_i$  is min cut size,  $m_i$  is edges remaining.

By repeating enough times, the failure probability decays exponentially.

## Try It Yourself

1. Run the algorithm 1 time on a 4-node graph, note variability.
2. Repeat 50 times, collect min cut frequencies.
3. Visualize contraction steps on paper.
4. Add weights by expanding weighted edges into parallel copies.
5. Compare runtime vs Stoer–Wagner on dense graphs.

## Test Cases

| Graph            | Expected Min Cut | Notes                    |
|------------------|------------------|--------------------------|
| Triangle         | 2                | always 2                 |
| Square (4-cycle) | 2                | random paths converge    |
| Complete $K_4$   | 3                | dense, repeat to confirm |

## Complexity

- Time:  $O(n^2)$  per trial
- Space:  $O(n^2)$  (for adjacency)
- Repetitions:  $O(n^2 \log n)$  for high confidence

Karger's algorithm is a landmark example: a single line, “*pick an edge at random and contract it*”, unfolds into a full-fledged, provably correct algorithm for global min-cut discovery.

## 363 Karger–Stein Minimum Cut

The Karger–Stein algorithm is an improved randomized divide-and-conquer version of Karger's original contraction algorithm. It achieves a higher success probability and better expected runtime, while preserving the same beautifully simple idea: repeatedly contract random edges, but stop early and recurse instead of fully collapsing to two vertices.

### What Problem Are We Solving?

Given an undirected, weighted or unweighted graph  $G = (V, E)$ , the goal is to find the global minimum cut, that is:

$$\min_{S \subset V, S \neq \emptyset, S \neq V} \text{cut}(S)$$

where:

$$\text{cut}(S) = \sum_{u \in S, v \in V \setminus S} w(u, v)$$

## How Does It Work (Plain Language)?

Like Karger's algorithm, we contract edges randomly, but instead of shrinking to 2 vertices immediately, we contract only until the graph has about  $\frac{n}{\sqrt{2}}$  vertices, then recurse twice independently. The best cut found across recursive calls is our result.

This strategy amplifies success probability while maintaining efficiency.

Algorithm:

1. If  $|V| \leq 6$ :
  - Compute min cut directly by brute force or basic contraction.
2. Otherwise:
  - Let  $t = \lceil n/\sqrt{2} + 1 \rceil$ .
  - Randomly contract edges until only  $t$  vertices remain.
  - Run the algorithm recursively twice on two independent contractions.
3. Return the smaller of the two cuts.

Each contraction phase preserves the minimum cut with good probability, and recursive repetition compounds success.

## Example

For a graph of 16 vertices:

- Contract randomly down to  $\lceil 16/\sqrt{2} \rceil = 12$ .
- Recurse twice independently.
- Each recursion again halves vertex count until base case.
- Return smallest cut found.

Multiple recursion branches make the overall probability of keeping all min-cut edges much higher than one-shot contraction.

## Tiny Code (Python)

```

import random
import math
import copy

def contract_random_edge(graph):
    u, v = random.choice([(u, w) for u in graph for w in graph[u] if u < w])
    # merge v into u
    graph[u].extend(graph[v])
    for w in graph[v]:
        graph[w] = [u if x == v else x for x in graph[w]]
    del graph[v]
    # remove self-loops
    graph[u] = [x for x in graph[u] if x != u]

def karger_stein(graph):
    n = len(graph)
    if n <= 6:
        # base case: fall back to basic Karger
        g_copy = copy.deepcopy(graph)
        while len(g_copy) > 2:
            contract_random_edge(g_copy)
        return len(list(g_copy.values())[0])

    t = math.ceil(n / math.sqrt(2)) + 1
    g1 = copy.deepcopy(graph)
    g2 = copy.deepcopy(graph)
    while len(g1) > t:
        contract_random_edge(g1)
    while len(g2) > t:
        contract_random_edge(g2)

    return min(karger_stein(g1), karger_stein(g2))

```

## Why It Matters

- Improves success probability to roughly  $1/\log n$ , vs  $1/n^2$  for basic Karger.
- Divide-and-conquer structure reduces required repetitions.
- Demonstrates power of probability amplification in randomized algorithms.
- Useful for large dense graphs where deterministic  $O(V^3)$  algorithms are slower.

It's a near-optimal randomized min-cut algorithm, balancing simplicity and efficiency.

## A Gentle Proof (Why It Works)

Each contraction preserves the min cut with probability:

$$p = \prod_{i=k+1}^n \left(1 - \frac{2}{i}\right)$$

The early stopping at  $t = n/\sqrt{2}$  keeps  $p$  reasonably high. Since we recurse twice independently, overall success probability becomes:

$$P = 1 - (1-p)^2 \approx 2p$$

Repeating  $O(\log^2 n)$  times ensures high confidence of finding the true minimum cut.

## Try It Yourself

1. Compare runtime and accuracy vs plain Karger.
2. Run on small graphs and collect success rate over 100 trials.
3. Visualize recursive tree of contractions.
4. Add edge weights (by expanding parallel edges).
5. Confirm returned cut matches Stoer–Wagner result.

## Test Cases

| Graph        | Expected Min Cut | Notes                      |
|--------------|------------------|----------------------------|
| Triangle     | 2                | Always found               |
| 4-node cycle | 2                | High accuracy              |
| Dense $K_6$  | 5                | Repeats improve confidence |

## Complexity

- Expected Time:  $O(n^2 \log^3 n)$
- Space:  $O(n^2)$
- Repetitions:  $O(\log^2 n)$  for high probability

Karger–Stein refines the raw elegance of random contraction into a divide-and-conquer gem, faster, more reliable, and still delightfully simple.

## 364 Gomory–Hu Tree

The Gomory–Hu Tree is a remarkable data structure that compactly represents all-pairs minimum cuts in an undirected weighted graph. Instead of computing  $O(n^2)$  separate cuts, it builds a single tree (with  $n - 1$  edges) whose edge weights capture every pair's min-cut value.

This structure transforms global connectivity questions into simple tree queries, fast, exact, and elegant.

### What Problem Are We Solving?

Given an undirected weighted graph  $G = (V, E, w)$ , we want to find, for every pair  $(s, t)$ :

$$\lambda(s, t) = \min_{S \subset V, s \in S, t \notin S} \sum_{u \in S, v \notin S} w(u, v)$$

Instead of running a separate min-cut computation for each pair, we build a Gomory–Hu tree  $T$ , such that:

For any pair  $(s, t)$ , the minimum edge weight on the path between  $s$  and  $t$  in  $T$  equals  $\lambda(s, t)$ .

This tree encodes all-pairs min-cuts compactly.

### How Does It Work (Plain Language)?

The Gomory–Hu tree is constructed iteratively using  $n - 1$  minimum-cut computations:

1. Choose a root  $r$  (arbitrary).
2. Maintain a partition tree  $T$  with vertices as nodes.
3. While there are unprocessed partitions:
  - Pick two vertices  $s, t$  in the same partition.
  - Compute the minimum  $s-t$  cut using any max-flow algorithm.
  - Partition vertices into two sets  $(S, V \setminus S)$  along that cut.
  - Add an edge  $(s, t)$  in the Gomory–Hu tree with weight equal to the cut value.
  - Recurse on each partition.
4. After  $n - 1$  cuts, the tree is complete.

Every edge in the tree represents a distinct partition cut in the original graph.

## Example

Consider a graph with vertices  $A, B, C, D$  and weighted edges. We run cuts step by step:

1. Choose  $s = A, t = B \rightarrow$  find cut weight  $w(A, B) = 2 \rightarrow$  Add edge  $(A, B, 2)$  to tree.
2. Recurse within  $A$ 's and  $B$ 's sides.
3. Continue until tree has  $n - 1 = 3$  edges.

Now for any  $(u, v)$  pair, the min-cut = minimum edge weight along the path connecting them in  $T$ .

## Tiny Code (High-Level Skeleton)

```
def gomory_hu_tree(V, edges):
    # edges: list of (u, v, w)
    from collections import defaultdict
    n = V
    tree = defaultdict(list)
    parent = [0] * n
    cut_value = [0] * n

    for s in range(1, n):
        t = parent[s]
        # compute s-t min cut via max-flow
        mincut, partition = min_cut(s, t, edges)
        cut_value[s] = mincut
        for v in range(n):
            if v != s and parent[v] == t and partition[v]:
                parent[v] = s
        tree[s].append((t, mincut))
        tree[t].append((s, mincut))
        if partition[t]:
            parent[s], parent[t] = parent[t], s
    return tree
```

In practice, `min_cut(s, t)` is computed using Edmonds–Karp or Push–Relabel.

## Why It Matters

- Captures all-pairs min-cuts in just  $n - 1$  max-flow computations.
- Transforms graph cut queries into simple tree queries.

- Enables efficient network reliability analysis, connectivity queries, and redundancy planning.
- Works for weighted undirected graphs.

This algorithm compresses rich connectivity information into a single elegant structure.

### A Gentle Proof (Why It Works)

Each  $s-t$  min-cut partitions vertices into two sides; merging these partitions iteratively preserves all-pairs min-cut relationships.

By induction, every pair  $(u, v)$  is eventually separated by exactly one edge in the tree, with weight equal to  $\lambda(u, v)$ . Thus,

$$\lambda(u, v) = \min_{e \in \text{path}(u, v)} w(e)$$

### Try It Yourself

1. Run on a small 4-node weighted graph.
2. Verify each edge weight equals an actual cut value.
3. Query random pairs  $(u, v)$  by tree path min.
4. Compare with independent flow-based min-cut computations.
5. Draw both the original graph and the resulting tree.

### Test Cases

| Graph          | Tree Edges | Notes                              |
|----------------|------------|------------------------------------|
| Triangle       | 2 edges    | Uniform weights produce equal cuts |
| Square         | 3 edges    | Distinct cuts form balanced tree   |
| Complete $K_4$ | 3 edges    | Symmetric connectivity             |

### Complexity

- Time:  $O(n \cdot \text{MaxFlow}(V, E))$
- Space:  $O(V + E)$
- Queries:  $O(\log V)$  (via path min in tree)

The Gomory–Hu tree elegantly transforms the cut landscape of a graph into a tree of truths, one structure, all min-cuts.

## 365 Max-Flow Min-Cut Theorem

The Max-Flow Min-Cut Theorem is one of the foundational results in graph theory and combinatorial optimization. It reveals a duality between maximum flow (what can pass through a network) and minimum cut (what blocks the network). The two are not just related, they are exactly equal.

This theorem underpins nearly every algorithm for flows, cuts, and network design.

### What Problem Are We Solving?

We are working with a directed graph  $G = (V, E)$ , a source  $s$ , and a sink  $t$ . Each edge  $(u, v)$  has a capacity  $c(u, v) \geq 0$ .

A flow assigns values  $f(u, v)$  to edges such that:

1. Capacity constraint:  $0 \leq f(u, v) \leq c(u, v)$
2. Flow conservation:  $\sum_v f(u, v) = \sum_v f(v, u)$  for all  $u \neq s, t$

The value of the flow is:

$$|f| = \sum_v f(s, v)$$

We want the maximum possible flow value from  $s$  to  $t$ .

A cut  $(S, T)$  is a partition of vertices such that  $s \in S$  and  $t \in T = V \setminus S$ . The capacity of the cut is:

$$c(S, T) = \sum_{u \in S, v \in T} c(u, v)$$

### The Theorem

Max-Flow Min-Cut Theorem: In every flow network, the maximum value of a feasible flow equals the minimum capacity of an  $s-t$  cut.

Formally,

$$\max_f |f| = \min_{(S, T)} c(S, T)$$

This is a strong duality statement, a maximum over one set equals a minimum over another.

## How Does It Work (Plain Language)?

1. You try to push as much flow as possible from  $s$  to  $t$ .
2. As you saturate edges, certain parts of the graph become bottlenecks.
3. The final residual graph separates reachable vertices from  $s$  (via unsaturated edges) and the rest.
4. This separation  $(S, T)$  forms a minimum cut, whose capacity equals the total flow sent.

Thus, once you can't push any more flow, you've also found the tightest cut blocking you.

## Example

Consider a small network:

| Edge              | Capacity |
|-------------------|----------|
| $s \rightarrow a$ | 3        |
| $s \rightarrow b$ | 2        |
| $a \rightarrow t$ | 2        |
| $b \rightarrow t$ | 3        |
| $a \rightarrow b$ | 1        |

Max flow via augmenting paths:

- $s \rightarrow a \rightarrow t$ : 2 units
- $s \rightarrow b \rightarrow t$ : 2 units
- $s \rightarrow a \rightarrow b \rightarrow t$ : 1 unit

Total flow = 5.

Min cut:  $S = s, a$ ,  $T = b, t \rightarrow$  capacity = 5. Flow = Cut = 5

## Tiny Code (Verification via Edmonds–Karp)

```
from collections import deque

def bfs(cap, flow, s, t, parent):
    n = len(cap)
    visited = [False]*n
    q = deque([s])
    visited[s] = True
```

```

while q:
    u = q.popleft()
    for v in range(n):
        if not visited[v] and cap[u][v] - flow[u][v] > 0:
            parent[v] = u
            visited[v] = True
            if v == t: return True
            q.append(v)
return False

def max_flow_min_cut(cap, s, t):
    n = len(cap)
    flow = [[0]*n for _ in range(n)]
    parent = [-1]*n
    max_flow = 0
    while bfs(cap, flow, s, t, parent):
        v = t
        path_flow = float('inf')
        while v != s:
            u = parent[v]
            path_flow = min(path_flow, cap[u][v] - flow[u][v])
            v = u
        v = t
        while v != s:
            u = parent[v]
            flow[u][v] += path_flow
            flow[v][u] -= path_flow
            v = u
        max_flow += path_flow
    return max_flow

```

The final reachable set from  $s$  in the residual graph defines the minimum cut.

## Why It Matters

- Fundamental theorem linking optimization and combinatorics.
- Basis for algorithms like Ford–Fulkerson, Edmonds–Karp, Dinic, Push–Relabel.
- Used in image segmentation, network reliability, scheduling, bipartite matching, clustering, and transportation.

It bridges the gap between flow maximization and cut minimization, two faces of the same coin.

## A Gentle Proof (Why It Works)

At termination of Ford–Fulkerson:

- No augmenting path exists in the residual graph.
- Let  $S$  be all vertices reachable from  $s$ .
- Then all edges  $(u, v)$  with  $u \in S, v \notin S$  are saturated.

Thus:

$$|f| = \sum_{u \in S, v \notin S} f(u, v) = \sum_{u \in S, v \notin S} c(u, v) = c(S, T)$$

and no cut can have smaller capacity. Therefore,  $\max |f| = \min c(S, T)$ .

## Try It Yourself

1. Build a simple 4-node network and trace augmenting paths.
2. Identify the cut  $(S, T)$  at termination.
3. Verify total flow equals cut capacity.
4. Test different max-flow algorithms, result remains identical.
5. Visualize cut edges in residual graph.

## Test Cases

| Graph          | Max Flow | Min Cut | Notes                      |
|----------------|----------|---------|----------------------------|
| Simple chain   | 5        | 5       | identical                  |
| Parallel edges | 8        | 8       | same result                |
| Diamond graph  | 6        | 6       | min cut = bottleneck edges |

## Complexity

- Depends on underlying flow algorithm (e.g.  $O(VE^2)$  for Edmonds–Karp)
- Cut extraction:  $O(V + E)$

The Max-Flow Min-Cut Theorem is the heartbeat of network optimization, every augmenting path is a step toward equality between two fundamental perspectives: sending and separating.

## 366 Stoer–Wagner Repeated Phase

The Stoer–Wagner Repeated Phase algorithm is a refinement of the Stoer–Wagner minimum cut algorithm for undirected weighted graphs. It finds the global minimum cut by repeating a maximum adjacency search phase multiple times, progressively merging vertices and tracking cut weights.

This algorithm is elegant, deterministic, and runs in polynomial time, often faster than flow-based approaches for undirected graphs.

### What Problem Are We Solving?

We are finding the minimum cut in an undirected, weighted graph  $G = (V, E)$ , where each edge  $(u, v)$  has a non-negative weight  $w(u, v)$ .

A cut  $(S, V \setminus S)$  partitions the vertex set into two disjoint subsets. The weight of a cut is:

$$w(S, V \setminus S) = \sum_{u \in S, v \notin S} w(u, v)$$

Our goal is to find a cut  $(S, T)$  that minimizes this sum.

### Core Idea

The algorithm works by repeatedly performing “phases”. Each phase identifies a minimum  $s-t$  cut in the current contracted graph and merges the last two added vertices. Over multiple phases, the smallest cut discovered is the global minimum.

Each phase follows a maximum adjacency search pattern, similar to Prim’s algorithm but in reverse logic.

### How It Works (Plain Language)

Each phase:

1. Pick an arbitrary start vertex.
2. Maintain a set  $A$  of added vertices.
3. At each step, add the most tightly connected vertex  $v$  not in  $A$  (the one with maximum total edge weight to  $A$ ).
4. Continue until all vertices are in  $A$ .
5. Let  $s$  be the second-to-last added vertex, and  $t$  the last added.
6. The cut separating  $t$  from the rest is a candidate min cut.
7. Record its weight; then merge  $s$  and  $t$  into a supervertex and repeat.

After  $|V| - 1$  phases, the smallest cut seen is the global minimum cut.

### Example

Graph:

| Edge | Weight |
|------|--------|
| A–B  | 3      |
| A–C  | 2      |
| B–C  | 4      |
| B–D  | 2      |
| C–D  | 3      |

Phase 1:

- Start with  $A = A$
- Add  $B$  (max connected to  $A$ : 3)
- Add  $C$  (max connected to  $A, B$ : total 6)
- Add  $D$  last → Cut weight = sum of edges from  $D$  to  $A, B, C = 5$

Record min cut = 5. Merge  $(C, D)$  and continue.

Repeat phases → global min cut = 5.

### Tiny Code (Simplified Python)

```
def stoer_wagner_min_cut(graph):
    n = len(graph)
    vertices = list(range(n))
    min_cut = float('inf')

    while len(vertices) > 1:
        added = [False] * n
        weights = [0] * n
        prev = -1
        for _ in range(len(vertices)):
            u = max(vertices, key=lambda v: weights[v] if not added[v] else -1)
            added[u] = True
            if _ == len(vertices) - 1:
                # Last vertex added, potential cut
```

```

min_cut = min(min_cut, weights[u])
# Merge u into prev
if prev != -1:
    for v in vertices:
        if v != u and v != prev:
            graph[prev][v] += graph[u][v]
            graph[v][prev] = graph[prev][v]
    vertices.remove(u)
break
prev = u
for v in vertices:
    if not added[v]:
        weights[v] += graph[u][v]
return min_cut

```

Graph is given as an adjacency matrix. Each phase picks the tightest vertex, records the cut, and merges nodes.

## Why It Matters

- Deterministic and elegant for undirected weighted graphs.
- Faster than running multiple max-flow computations.
- Ideal for network reliability, graph partitioning, clustering, and circuit design.
- Each phase mimics “tightening” the graph until only one supervertex remains.

## A Gentle Proof (Why It Works)

Each phase finds the minimum  $s-t$  cut where  $t$  is the last added vertex. By merging  $s$  and  $t$ , we preserve all other possible cuts. The smallest of these phase cuts is the global minimum, as each cut in the merged graph corresponds to one in the original graph.

Formally:

$$\text{mincut}(G) = \min_{\text{phases}} w(S, V \setminus S)$$

This inductive structure ensures optimality.

## Try It Yourself

1. Run on a 4-node complete graph with random weights.
2. Trace vertex addition order in each phase.
3. Record cut weights per phase.
4. Compare with brute-force all cuts, they match.
5. Visualize contraction steps.

## Test Cases

| Graph                    | Edges | Min Cut                  | Notes             |
|--------------------------|-------|--------------------------|-------------------|
| Triangle (equal weights) | 3     | $2 \times \text{weight}$ | all equal         |
| Square (unit weights)    | 4     | 2                        | opposite sides    |
| Weighted grid            | 6     | smallest bridge          | cut at bottleneck |

## Complexity

- Each phase:  $O(V^2)$
- Total:  $O(V^3)$  for adjacency matrix

With Fibonacci heaps, can improve to  $O(V^2 \log V + VE)$ .

The Stoer–Wagner repeated phase algorithm is a powerful, purely combinatorial tool, no flows, no residual graphs, just tight connectivity and precise merging toward the true global minimum cut.

## 367 Dynamic Min Cut

The Dynamic Minimum Cut problem extends classical min-cut computation to graphs that change over time, edges can be added, deleted, or updated. Instead of recomputing from scratch after each change, we maintain data structures that update the min cut efficiently.

Dynamic min-cut algorithms are critical in applications like network resilience, incremental optimization, and real-time systems where connectivity evolves.

## What Problem Are We Solving?

Given a graph  $G = (V, E)$  with weighted edges and a current minimum cut, how do we efficiently maintain this cut when:

- An edge  $(u, v)$  is inserted
- An edge  $(u, v)$  is deleted
- An edge  $(u, v)$  has its weight changed

The naive approach recomputes min-cut from scratch using an algorithm like Stoer–Wagner ( $O(V^3)$ ). Dynamic algorithms aim for faster incremental updates.

## Core Idea

The min cut is sensitive only to edges crossing the cut boundary. When the graph changes, only a local region around modified edges can alter the global cut.

Dynamic algorithms use:

- Dynamic trees (e.g. Link-Cut Trees)
- Fully dynamic connectivity structures
- Randomized contraction tracking
- Incremental recomputation of affected regions

to update efficiently instead of re-running full min-cut.

## How It Works (Plain Language)

1. Maintain a representation of cuts, often as trees or partition sets.
2. When an edge weight changes, update connectivity info:
  - If edge lies within one side of the cut, no change.
  - If edge crosses the cut, update cut capacity.
3. When an edge is added or deleted, adjust affected components and recalculate locally.
4. Optionally, run periodic global recomputations to correct drift after many updates.

These methods trade exactness for efficiency, often maintaining approximate min cuts within small error bounds.

## Example (High-Level)

Graph has vertices  $A, B, C, D$  with current min cut  $(A, B, C, D)$ , weight 5.

1. Add edge  $(B, C)$  with weight 1:
  - New crossing edge, cut weight becomes  $5 + 1 = 6$ .
  - Check if alternate partition gives smaller total, update if needed.
2. Delete edge  $(A, C)$ :
  - Remove from cut set.
  - If this edge was essential to connectivity, min cut might increase.
  - Recompute local cut if affected.

## Tiny Code (Sketch, Python)

```
class DynamicMinCut:  
    def __init__(self, graph):  
        self.graph = graph  
        self.min_cut_value = self.compute_min_cut()  
  
    def compute_min_cut(self):  
        # Use Stoer-Wagner or flow-based method  
        return stoer_wagner(self.graph)  
  
    def update_edge(self, u, v, new_weight):  
        self.graph[u][v] = new_weight  
        self.graph[v][u] = new_weight  
        # Locally recompute affected region  
        self.min_cut_value = self.recompute_local(u, v)  
  
    def recompute_local(self, u, v):  
        # Simplified placeholder: recompute fully if small graph  
        return self.compute_min_cut()
```

For large graphs, replace `recompute_local` with incremental cut update logic.

## Why It Matters

- Real-time systems need quick responses to network changes.
- Streaming graphs (e.g. traffic, social, or power networks) evolve continuously.
- Reliability analysis in dynamic systems relies on up-to-date min-cut values.

Dynamic maintenance saves time compared to recomputing from scratch at every step.

## A Gentle Proof (Why It Works)

Let  $C_t$  be the min cut after  $t$  updates. If each update affects only local structure, then:

$$C_{t+1} = \min(C_t, \text{local adjustment})$$

Maintaining a certificate structure (like a Gomory–Hu tree) ensures correctness since all pairwise min-cuts are preserved under local changes, except where updated edges are involved.

Recomputing only affected cuts guarantees correctness with amortized efficiency.

## Try It Yourself

1. Build a small weighted graph (5–6 nodes).
2. Compute initial min cut using Stoer–Wagner.
3. Add or delete edges, one at a time.
4. Update only affected cuts manually.
5. Compare to full recomputation, results should match.

## Test Cases

| Operation                           | Description        | New Min Cut              |
|-------------------------------------|--------------------|--------------------------|
| Add edge $(u, v)$ with small weight | New crossing edge  | Possibly smaller cut     |
| Increase edge weight                | Strengthens bridge | Cut may change elsewhere |
| Delete edge across cut              | Weakens connection | Cut may increase         |
| Delete edge inside partition        | No change          | ,                        |

| Operation | Time |
|-----------|------|
|-----------|------|

## Complexity

| Operation                     | Time                      |
|-------------------------------|---------------------------|
| Naive recomputation           | $O(V^3)$                  |
| Dynamic approach (randomized) | $O(V^2 \log V)$ amortized |
| Approximate dynamic cut       | $O(E \log^2 V)$           |

Dynamic min-cut algorithms balance exactness with responsiveness, maintaining near-optimal connectivity insight as graphs evolve in real time.

## 368 Minimum s–t Cut (Edmonds–Karp)

The Minimum s–t Cut problem seeks the smallest total capacity of edges that must be removed to separate a source vertex  $s$  from a sink vertex  $t$ . It's the dual counterpart to maximum flow, and the Edmonds–Karp algorithm provides a clear path to compute it using BFS-based augmenting paths.

### What Problem Are We Solving?

Given a directed, weighted graph  $G = (V, E)$  with capacities  $c(u, v)$ , find a partition of  $V$  into two disjoint sets  $(S, T)$  such that:

- $s \in S, t \in T$
- The sum of capacities of edges from  $S$  to  $T$  is minimum

Formally:

$$\text{min-cut}(s, t) = \min_{(S, T)} \sum_{u \in S, v \in T} c(u, v)$$

This cut value equals the maximum flow value from  $s$  to  $t$  by the Max-Flow Min-Cut Theorem.

## How It Works (Plain Language)

The algorithm uses Edmonds–Karp, a BFS-based version of Ford–Fulkerson, to find the maximum flow first. After computing max-flow, the reachable vertices from  $s$  in the residual graph determine the min-cut.

Steps:

1. Initialize flow  $f(u, v) = 0$  for all edges.
2. While there exists a BFS path from  $s$  to  $t$  in the residual graph:
  - Compute bottleneck capacity along path.
  - Augment flow along path.
3. After no augmenting path exists:
  - Run BFS one last time from  $s$  in residual graph.
  - Vertices reachable from  $s$  form set  $S$ .
  - Others form  $T$ .
4. The edges crossing from  $S$  to  $T$  with full capacity are the min-cut edges.

## Example

Graph:

- Vertices:  $s, a, b, t$

- Edges:
  - $(s, a) = 3$
  - $(s, b) = 2$
  - $(a, b) = 1$
  - $(a, t) = 2$
  - $(b, t) = 3$

1. Run Edmonds–Karp to find max-flow = 4.
2. Residual graph:
  - Reachable set from  $s$ :  $s, a$
  - Unreachable set:  $b, t$
3. Min-cut edges:  $(a, t)$  and  $(s, b)$
4. Min-cut value =  $2 + 2 = 4$  Matches max-flow value.

## Tiny Code (C-like Pseudocode)

```
int min_st_cut(Graph *G, int s, int t) {
    int maxflow = edmonds_karp(G, s, t);
    bool visited[V];
    bfs_residual(G, s, visited);
    int cut_value = 0;
    for (edge (u,v) in G->edges)
        if (visited[u] && !visited[v])
            cut_value += G->capacity[u][v];
    return cut_value;
}
```

## Why It Matters

- Reveals bottlenecks in a network.
- Key for reliability and segmentation problems.
- Foundational for image segmentation, network design, and flow decomposition.
- Directly supports duality proofs between optimization problems.

## A Gentle Proof (Why It Works)

The Max-Flow Min-Cut Theorem states:

$$\max_{\text{flow } f} \sum_v f(s, v) = \min_{(S, T)} \sum_{u \in S, v \in T} c(u, v)$$

Edmonds–Karp finds the maximum flow by repeatedly augmenting along shortest paths (BFS order). Once no more augmenting paths exist, the residual graph partitions nodes naturally into  $S$  and  $T$ , and the edges from  $S$  to  $T$  define the min cut.

## Try It Yourself

1. Build a small directed network with capacities.
2. Run Edmonds–Karp manually (trace augmenting paths).
3. Draw residual graph and find reachable set from  $s$ .
4. Mark crossing edges, sum their capacities.
5. Compare with max-flow value.

## Test Cases

| Graph                                                      | Max Flow        | Min Cut         | Matches? |
|------------------------------------------------------------|-----------------|-----------------|----------|
| Simple 4-node                                              | 4               | 4               |          |
| Linear chain $s \rightarrow a \rightarrow b \rightarrow t$ | min edge        | min edge        |          |
| Parallel paths                                             | sum of min caps | sum of min caps |          |

## Complexity

| Step                 | Time      |
|----------------------|-----------|
| BFS per augmentation | $O(E)$    |
| Augmentations        | $O(VE)$   |
| Total                | $O(VE^2)$ |

The Minimum s–t Cut via Edmonds–Karp is an elegant bridge between flow algorithms and partition reasoning, every edge in the cut tells a story of constraint, capacity, and balance.

## 369 Approximate Min Cut

The Approximate Minimum Cut algorithm provides a way to estimate the minimum cut of a graph faster than exact algorithms, especially when exact precision is not critical. It's built on randomization and sampling, using probabilistic reasoning to find small cuts with high confidence in large or dynamic graphs.

### What Problem Are We Solving?

Given a weighted, undirected graph  $G = (V, E)$ , we want a cut  $(S, T)$  such that the cut capacity is close to the true minimum:

$$w(S, T) \leq (1 + \epsilon) \cdot \lambda(G)$$

where  $\lambda(G)$  is the weight of the global min cut, and  $\epsilon$  is a small error tolerance (e.g. 0.1).

The goal is speed: approximate min-cut algorithms run in near-linear time, much faster than exact ones ( $O(V^3)$ ).

## How It Works (Plain Language)

Approximate algorithms rely on two key principles:

1. Random Sampling: Randomly sample edges with probability proportional to their weight. The smaller the edge capacity, the more likely it's critical to the min cut.
2. Graph Sparsification: Build a smaller “sketch” of the graph that preserves cut weights approximately. Compute the min cut on this sparse graph, it's close to the true value.

By repeating sampling several times and taking the minimum found cut, we converge to a near-optimal solution.

## Algorithm Sketch (Karger's Sampling Method)

1. Input: Graph  $G(V, E)$  with  $n = |V|$  and  $m = |E|$
2. Choose sampling probability  $p = \frac{c \log n}{\epsilon^2 \lambda}$
3. Build sampled graph  $G'$ :
  - Include each edge  $(u, v)$  with probability  $p$
  - Scale included edge weights by  $\frac{1}{p}$
4. Run an exact min cut algorithm (Stoer–Wagner) on  $G'$
5. Repeat sampling  $O(\log n)$  times; take the best cut found

The result approximates  $\lambda(G)$  within factor  $(1 + \epsilon)$  with high probability.

## Example

Suppose  $G$  has  $10^5$  edges, and exact Stoer–Wagner would be too slow.

1. Choose  $\epsilon = 0.1$ ,  $p = 0.02$
2. Sample 2 of edges randomly (2000 edges)
3. Reweight sampled edges by  $\frac{1}{0.02} = 50$
4. Run exact min cut on this smaller graph
5. Repeat 5–10 times; pick smallest cut

Result: A cut within 10 of optimal, in a fraction of the time.

## Tiny Code (Python-like Pseudocode)

```
def approximate_min_cut(G, epsilon=0.1, repeats=10):
    best_cut = float('inf')
    for _ in range(repeats):
        p = compute_sampling_probability(G, epsilon)
        G_sample = sample_graph(G, p)
        cut_value = stoer_wagner(G_sample)
        best_cut = min(best_cut, cut_value)
    return best_cut
```

## Why It Matters

- Scalability: Handles huge graphs where exact methods are infeasible
- Speed: Near-linear time using randomization
- Applications:
  - Streaming graphs
  - Network reliability
  - Clustering and partitioning
  - Graph sketching and sparsification

Approximate min cuts are crucial when you need quick, robust decisions, not perfect answers.

## A Gentle Proof (Why It Works)

Karger's analysis shows that each small cut is preserved with high probability if enough edges are sampled:

$$\Pr[\text{cut weight preserved}] \geq 1 - \frac{1}{n^2}$$

By repeating the process  $O(\log n)$  times, we amplify confidence, ensuring that with high probability, at least one sampled graph maintains the true min-cut structure.

Using Chernoff bounds, the error is bounded by  $(1 \pm \epsilon)$ .

## Try It Yourself

1. Generate a random graph with 50 nodes and random weights.
2. Compute exact min cut using Stoer–Wagner.
3. Sample 10%, 5%, and 2% of edges, compute approximate cuts.
4. Compare results and runtime.
5. Adjust  $\epsilon$  and observe trade-off between speed and accuracy.

## Test Cases

| Graph                    | Exact Min Cut | Approx. ( $\epsilon = 0.1$ ) | Error | Speedup     |
|--------------------------|---------------|------------------------------|-------|-------------|
| Small dense (100 edges)  | 12            | 13                           | 8%    | 5 $\times$  |
| Medium sparse (1k edges) | 8             | 8                            | 0%    | 10 $\times$ |
| Large (100k edges)       | 30            | 33                           | 10%   | 50 $\times$ |

## Complexity

| Method                    | Time                         | Accuracy              |
|---------------------------|------------------------------|-----------------------|
| Stoer–Wagner              | $O(V^3)$                     | Exact                 |
| Karger (Randomized Exact) | $O(V^2 \log^3 V)$            | Exact (probabilistic) |
| Approximate Sampling      | $O(E \log^2 V / \epsilon^2)$ | $(1 + \epsilon)$      |

Approximate min-cut algorithms show that probability can replace precision when scale demands speed, they slice through massive graphs with surprising efficiency.

## 370 Min k-Cut

The Minimum k-Cut problem generalizes the classic min-cut idea. Instead of splitting a graph into just two parts, the goal is to partition it into  $k$  disjoint subsets while cutting edges of minimum total weight.

It's a key problem in clustering, parallel processing, and network design, where you want multiple disconnected regions with minimal interconnection cost.

## What Problem Are We Solving?

Given a weighted, undirected graph  $G = (V, E)$  and an integer  $k$ , find a partition of  $V$  into  $k$  subsets  $V_1, V_2, \dots, V_k$  such that:

- $V_i \cap V_j = \emptyset$  for all  $i \neq j$
- $\bigcup_i V_i = V$
- The sum of edge weights crossing between parts is minimized

Formally:

$$\text{min-}k\text{-cut}(G) = \min_{V_1, \dots, V_k} \sum_{(u,v) \in E} \sum_{u \in V_i, v \in V_j, i \neq j} w(u, v)$$

For  $k = 2$ , this reduces to the standard min-cut problem.

## How It Works (Plain Language)

The Min k-Cut problem is NP-hard for general  $k$ , but several algorithms provide exact solutions for small  $k$  and approximations for larger  $k$ .

There are two main approaches:

1. Greedy Iterative Cutting:

- Repeatedly find and remove a global min cut, splitting the graph into components one by one.
- After  $k - 1$  cuts, you have  $k$  components.
- Works well but not always optimal.

2. Dynamic Programming over Trees (Exact for small  $k$ ):

- Use a tree decomposition of the graph.
- Compute optimal partition by exploring edge removals in the minimum spanning tree.
- Based on the Karger–Stein framework.

## Example

Graph with 5 nodes and edges:

| Edge   | Weight |
|--------|--------|
| (A, B) | 1      |
| (B, C) | 2      |
| (C, D) | 3      |
| (D, E) | 4      |
| (A, E) | 2      |

Goal: partition into  $k = 3$  subsets.

1. Find minimum cut edges to remove:
  - Cut  $(A, B)$  (weight 1)
  - Cut  $(B, C)$  (weight 2)
2. Total cut weight = 3
3. Resulting subsets:  $A, B, C, D, E$

### Tiny Code (Python-like Pseudocode)

```
def min_k_cut(graph, k):
    cuts = []
    G = graph.copy()
    for _ in range(k - 1):
        cut_value, (S, T) = stoer_wagner(G)
        cuts.append(cut_value)
        G = G.subgraph(S) # Keep one component, remove crossing edges
    return sum(cuts)
```

For small  $k$ , you can use recursive contraction (like Karger's algorithm) or dynamic programming on tree structures.

### Why It Matters

- Clustering: Group nodes into  $k$  balanced communities.
- Parallel Computing: Partition workloads while minimizing communication cost.
- Image Segmentation: Divide pixels into  $k$  coherent regions.
- Graph Simplification: Split networks into modular subgraphs.

Min k-Cut transforms connectivity into structured modularity.

## A Gentle Proof (Why It Works)

Each cut increases the number of connected components by 1. Thus, performing  $k - 1$  cuts produces exactly  $k$  components.

Let  $C_1, C_2, \dots, C_{k-1}$  be the successive minimum cuts. The sum of their weights bounds the global optimum:

$$\text{min-}k\text{-cut} \leq \sum_{i=1}^{k-1} \lambda_i$$

where  $\lambda_i$  is the  $i$ -th smallest cut value. Iterative min-cuts often approximate the optimal  $k$ -cut well.

For exact solutions, algorithms based on flow decomposition or tree contractions use recursive partitioning to explore combinations of edges.

## Try It Yourself

1. Construct a small weighted graph with 6–8 vertices.
2. Run Stoer–Wagner to find the first min cut.
3. Remove edges, repeat for next cut.
4. Compare total cut weight to brute-force partition for  $k = 3$ .
5. Observe approximation quality.

## Test Cases

| Graph        | k | Exact | Greedy | Error |
|--------------|---|-------|--------|-------|
| Triangle     | 3 | 3     | 3      | 0%    |
| Line of 5    | 3 | 3     | 3      | 0%    |
| Dense 6-node | 3 | 12    | 13     | 8%    |

## Complexity

| Algorithm                 | Time              | Type               |
|---------------------------|-------------------|--------------------|
| Brute Force               | Exponential       | Exact              |
| Greedy Cuts               | $O(k \cdot V^3)$  | Approx             |
| Tree DP                   | $O(V^{k-1})$      | Exact (small $k$ ) |
| Randomized (Karger–Stein) | $O(V^2 \log^3 V)$ | Approx             |

The Min k-Cut problem generalizes connectivity design — it's where graph partitioning meets optimization, balancing efficiency and modularity.

## Section 38. Matchings

### 371 Bipartite Matching (DFS)

Bipartite Matching is one of the most fundamental problems in graph theory. Given a bipartite graph  $G = (U, V, E)$ , the goal is to find the maximum number of pairings between nodes in  $U$  and  $V$  such that no two edges share a vertex. A simple and intuitive way to do this is by using DFS-based augmenting paths.

#### What Problem Are We Solving?

We are given a bipartite graph with two disjoint sets  $U$  and  $V$ . We want to find the maximum matching, the largest set of edges where each vertex is matched to at most one partner.

Formally, find a subset  $M \subseteq E$  such that:

- Each vertex in  $U \cup V$  is incident to at most one edge in  $M$
- $|M|$  is maximized

#### How It Works (Plain Language)

The core idea is to build matchings incrementally by finding augmenting paths — paths that start from an unmatched vertex in  $U$ , alternate between unmatched and matched edges, and end at an unmatched vertex in  $V$ .

Each time such a path is found, we flip the match status along it (matched edges become unmatched, and vice versa), increasing the total matching size by one.

Steps:

1. Start with an empty matching  $M$ .
2. For each vertex  $u \in U$ :
  - Run DFS to find an augmenting path to a free vertex in  $V$ .
  - If found, augment the matching along this path.
3. Repeat until no more augmenting paths exist.

## Example

Let  $U = u_1, u_2, u_3$ ,  $V = v_1, v_2, v_3$ , and edges:

- $(u_1, v_1), (u_1, v_2)$
- $(u_2, v_2)$
- $(u_3, v_3)$

1. Start with empty matching.
2. DFS from  $u_1$ : finds  $(u_1, v_1) \rightarrow$  match.
3. DFS from  $u_2$ : finds  $(u_2, v_2) \rightarrow$  match.
4. DFS from  $u_3$ : finds  $(u_3, v_3) \rightarrow$  match. All vertices matched, maximum matching size = 3.

## Tiny Code (C-like Pseudocode)

```
#define MAXV 100
vector<int> adj[MAXV];
int matchR[MAXV], visited[MAXV];

bool dfs(int u) {
    for (int v : adj[u]) {
        if (visited[v]) continue;
        visited[v] = 1;
        if (matchR[v] == -1 || dfs(matchR[v])) {
            matchR[v] = u;
            return true;
        }
    }
    return false;
}

int maxBipartiteMatching(int U) {
    memset(matchR, -1, sizeof(matchR));
    int result = 0;
    for (int u = 0; u < U; ++u) {
        memset(visited, 0, sizeof(visited));
        if (dfs(u)) result++;
    }
    return result;
}
```

## Why It Matters

- Foundation for Hungarian Algorithm and Hopcroft–Karp
- Core tool in resource allocation, scheduling, pairing problems
- Introduces concept of augmenting paths, a cornerstone in flow theory

Used in:

- Assigning workers to jobs
- Matching students to projects
- Network flow initialization
- Graph theory teaching and visualization

## A Gentle Proof (Why It Works)

If there exists an augmenting path, flipping the matching along it always increases the matching size by 1.

Let  $M$  be a current matching. If no augmenting path exists,  $M$  is maximum (Berge's Lemma):

A matching is maximum if and only if there is no augmenting path.

Therefore, repeatedly augmenting ensures convergence to the maximum matching.

## Try It Yourself

1. Draw a bipartite graph with 4 nodes on each side.
2. Use DFS to find augmenting paths manually.
3. Track which vertices are matched/unmatched after each augmentation.
4. Stop when no augmenting paths remain.

## Test Cases

| Graph                          | $ U $ | $ V $ | Max Matching | Steps |
|--------------------------------|-------|-------|--------------|-------|
| Complete $K_{3,3}$             | 3     | 3     | 3            | 3 DFS |
| Chain $u_1v_1, u_2v_2, u_3v_3$ | 3     | 3     | 3            | 3 DFS |
| Sparse graph                   | 4     | 4     | 2            | 4 DFS |

## Complexity

| Aspect         | Cost       |
|----------------|------------|
| DFS per vertex | $O(E)$     |
| Total          | $O(VE)$    |
| Space          | $O(V + E)$ |

This DFS-based approach gives an intuitive baseline for bipartite matching, later improved by Hopcroft–Karp ( $O(E\sqrt{V})$ ) but perfect for learning and small graphs.

## 372 Hopcroft–Karp

The Hopcroft–Karp algorithm is a classic improvement over DFS-based bipartite matching. It uses layered BFS and DFS to find multiple augmenting paths in parallel, reducing redundant searches and achieving an optimal runtime of  $O(E\sqrt{V})$ .

### What Problem Are We Solving?

Given a bipartite graph  $G = (U, V, E)$ , we want to find a maximum matching, the largest set of vertex-disjoint edges connecting  $U$  to  $V$ .

A matching is a subset  $M \subseteq E$  such that each vertex is incident to at most one edge in  $M$ . The algorithm seeks the maximum cardinality matching.

### How It Works (Plain Language)

Instead of finding one augmenting path at a time (like simple DFS), Hopcroft–Karp finds a layer of shortest augmenting paths, then augments all of them together. This drastically reduces the number of BFS-DFS phases.

Key idea:

- Each phase increases the matching by many paths at once.
- The distance (layer) of unmatched vertices strictly increases after each phase.

Steps:

1. Initialization: Start with empty matching  $M = \emptyset$ .
2. Repeat until no augmenting path exists:
  1. BFS phase:

- Build a layered graph (level graph) from unmatched vertices in  $U$  to unmatched vertices in  $V$ .
  - Each layer increases by 1 hop.
2. DFS phase:
    - Find vertex-disjoint augmenting paths in this layered graph.
    - Augment along all of them simultaneously.
  3. Return the total matching size.

### Example

Let  $U = u_1, u_2, u_3$ ,  $V = v_1, v_2, v_3$ , edges:

- $u_1 \rightarrow v_1, v_2$
- $u_2 \rightarrow v_2$
- $u_3 \rightarrow v_3$

1. Initial matching: empty
2. BFS builds layers:
  - Level 0:  $u_1, u_2, u_3$
  - Level 1:  $v_1, v_2, v_3$  All are reachable.
3. DFS finds 3 augmenting paths:
  - $u_1 \rightarrow v_1, u_2 \rightarrow v_2, u_3 \rightarrow v_3$
4. Augment all  $\rightarrow$  Matching size = 3 No more augmenting paths  $\rightarrow$  Maximum matching = 3

### Tiny Code (C-like Pseudocode)

```
vector<int> adj[MAXV];
int pairU[MAXV], pairV[MAXV], dist[MAXV];
int NIL = 0, INF = 1e9;

bool bfs(int U) {
    queue<int> q;
    for (int u = 1; u <= U; u++) {
        if (pairU[u] == NIL) { dist[u] = 0; q.push(u); }
        else dist[u] = INF;
```

```

    }

    dist[NIL] = INF;
    while (!q.empty()) {
        int u = q.front(); q.pop();
        if (dist[u] < dist[NIL]) {
            for (int v : adj[u]) {
                if (dist[pairV[v]] == INF) {
                    dist[pairV[v]] = dist[u] + 1;
                    q.push(pairV[v]);
                }
            }
        }
    }
    return dist[NIL] != INF;
}

bool dfs(int u) {
    if (u == NIL) return true;
    for (int v : adj[u]) {
        if (dist[pairV[v]] == dist[u] + 1 && dfs(pairV[v])) {
            pairV[v] = u; pairU[u] = v;
            return true;
        }
    }
    dist[u] = INF;
    return false;
}

int hopcroftKarp(int U) {
    memset(pairU, 0, sizeof(pairU));
    memset(pairV, 0, sizeof(pairV));
    int matching = 0;
    while (bfs(U)) {
        for (int u = 1; u <= U; u++)
            if (pairU[u] == NIL && dfs(u))
                matching++;
    }
    return matching;
}

```

## Why It Matters

- Efficient:  $O(E\sqrt{V})$ , ideal for large graphs
- Foundational in:
  - Job assignment
  - Resource allocation
  - Stable match foundations
  - Network optimization

It's the standard method for maximum bipartite matching in competitive programming and real systems.

## A Gentle Proof (Why It Works)

Each BFS-DFS phase finds a set of shortest augmenting paths. After augmenting, no shorter path remains.

Let  $d$  = distance to the nearest unmatched vertex in BFS. Every phase increases the minimum augmenting path length, and the number of phases is at most  $O(\sqrt{V})$  (Hopcroft–Karp Lemma).

Each BFS-DFS costs  $O(E)$ , so total =  $O(E\sqrt{V})$ .

## Try It Yourself

1. Draw a bipartite graph with 5 nodes on each side.
2. Run one BFS layer build.
3. Use DFS to find all shortest augmenting paths.
4. Augment all, track matching size per phase.

## Test Cases

| Graph        | U  | V  | Matching Size | Complexity       |
|--------------|----|----|---------------|------------------|
| $K_{3,3}$    | 3  | 3  | 3             | Fast             |
| Chain graph  | 5  | 5  | 5             | Linear           |
| Sparse graph | 10 | 10 | 6             | Sublinear phases |

## Complexity

| Operation           | Cost           |
|---------------------|----------------|
| BFS (build layers)  | $O(E)$         |
| DFS (augment paths) | $O(E)$         |
| Total               | $O(E\sqrt{V})$ |

Hopcroft–Karp is the benchmark for bipartite matching, balancing elegance, efficiency, and theoretical depth.

### 373 Hungarian Algorithm

The Hungarian Algorithm (also known as the Kuhn–Munkres Algorithm) solves the assignment problem, finding the minimum-cost perfect matching in a weighted bipartite graph. It's a cornerstone in optimization, turning complex allocation tasks into elegant linear-time computations on cost matrices.

#### What Problem Are We Solving?

Given a bipartite graph  $G = (U, V, E)$  with  $|U| = |V| = n$ , and a cost function  $c(u, v)$  for each edge, we want to find a matching  $M$  such that:

- Every  $u \in U$  is matched to exactly one  $v \in V$
- Total cost is minimized:

$$\text{Minimize} \sum_{(u,v) \in M} c(u, v)$$

This is the assignment problem, a special case of linear programming that can be solved exactly in polynomial time.

#### How It Works (Plain Language)

The Hungarian Algorithm treats the cost matrix like a grid puzzle — you systematically reduce, label, and cover rows and columns to reveal a set of zeros corresponding to the optimal assignment.

Core idea: Transform the cost matrix so that at least one optimal solution lies among zeros.

Steps:

1. Row Reduction Subtract the smallest element in each row from all elements in that row.

2. Column Reduction Subtract the smallest element in each column from all elements in that column.
3. Covering Zeros Use the minimum number of lines (horizontal + vertical) to cover all zeros.
4. Adjust Matrix If the number of covering lines  $< n$ :
  - Find smallest uncovered value  $m$
  - Subtract  $m$  from all uncovered elements
  - Add  $m$  to elements covered twice
  - Repeat from step 3
5. Assignment Once  $n$  lines are used, pick one zero per row/column  $\rightarrow$  that's the optimal matching.

### Example

Cost matrix:

|    | v1 | v2 | v3 |
|----|----|----|----|
| u1 | 4  | 1  | 3  |
| u2 | 2  | 0  | 5  |
| u3 | 3  | 2  | 2  |

1. Row Reduction: subtract min in each row | u1 | 3 | 0 | 2 | | u2 | 2 | 0 | 5 | | u3 | 1 | 0 | 0 |
2. Column Reduction: subtract min in each column | u1 | 2 | 0 | 2 | | u2 | 1 | 0 | 5 | | u3 | 0 | 0 |
3. Cover zeros with 3 lines  $\rightarrow$  feasible.
4. Assignment: pick  $(u1, v2), (u2, v1), (u3, v3)$   $\rightarrow$  total cost =  $1 + 2 + 2 = 5$

Optimal assignment found.

### Tiny Code (Python-like Pseudocode)

```
import numpy as np
from scipy.optimize import linear_sum_assignment

def hungarian(cost_matrix):
    row_ind, col_ind = linear_sum_assignment(cost_matrix)
```

```

total_cost = cost_matrix[row_ind, col_ind].sum()
return list(zip(row_ind, col_ind)), total_cost

# Example
cost = np.array([[4,1,3],[2,0,5],[3,2,2]])
match, cost = hungarian(cost)
print(match, cost) # [(0,1), (1,0), (2,2)], 5

```

## Why It Matters

- Exact and efficient:  $O(n^3)$  complexity
- Fundamental in operations research and AI (task allocation, scheduling, tracking)
- Used in:
  - Job-worker assignment
  - Optimal resource allocation
  - Matching predictions to ground truth (e.g. Hungarian loss in object detection)

The algorithm balances combinatorics and linear algebra, a rare blend of elegance and utility.

## A Gentle Proof (Why It Works)

The algorithm maintains dual feasibility and complementary slackness at each step. By reducing rows and columns, we ensure at least one zero in each row and column, creating a reduced cost matrix where zeros correspond to feasible assignments.

Each iteration moves closer to a perfect matching in the equality graph (edges where reduced cost = 0). Once all vertices are matched, the solution satisfies optimality conditions.

## Try It Yourself

1. Create a  $3 \times 3$  or  $4 \times 4$  cost matrix.
2. Perform row and column reductions manually.
3. Cover zeros and count lines.
4. Adjust and repeat until  $n$  lines used.
5. Assign one zero per row/column.

## Test Cases

| Matrix Size  | Cost Matrix Type   | Result                  |
|--------------|--------------------|-------------------------|
| $3 \times 3$ | Random integers    | Minimum cost assignment |
| $4 \times 4$ | Diagonal dominance | Diagonal chosen         |
| $5 \times 5$ | Symmetric          | Matching pairs found    |

## Complexity

| Step                  | Time     |
|-----------------------|----------|
| Row/column reductions | $O(n^2)$ |
| Iterative covering    | $O(n^3)$ |
| Total                 | $O(n^3)$ |

The Hungarian Algorithm transforms a cost matrix into structure, revealing an optimal matching hidden within zeros, one line at a time.

## 374 Kuhn–Munkres (Max-Weight Matching)

The Kuhn–Munkres Algorithm, also known as the Hungarian Algorithm for Maximum Weight Matching, solves the maximum-weight bipartite matching problem. While the standard Hungarian method minimizes total cost, this version maximizes total reward or utility, making it ideal for assignment optimization when bigger is better.

### What Problem Are We Solving?

Given a complete bipartite graph  $G = (U, V, E)$  with  $|U| = |V| = n$ , and weights  $w(u, v)$  on each edge, find a perfect matching  $M \subseteq E$  such that:

$$\text{maximize} \sum_{(u,v) \in M} w(u, v)$$

Each vertex is matched to exactly one partner on the opposite side, and total weight is as large as possible.

## How It Works (Plain Language)

The algorithm treats the weight matrix like a profit grid. It constructs a labeling on vertices and maintains equality edges (where label sums equal edge weight). By building and augmenting matchings within this equality graph, it converges to the optimal maximum-weight matching.

Key ideas:

- Maintain vertex labels  $l(u)$  and  $l(v)$  that satisfy  $l(u) + l(v) \geq w(u, v)$  (dual feasibility)
- Build equality graph where equality holds
- Find augmenting paths in equality graph
- Update labels when stuck to reveal new equality edges

Steps:

1. Initialize labels
  - $l(u) = \max_v w(u, v)$  for each  $u \in U$
  - $l(v) = 0$  for each  $v \in V$
2. Repeat for each  $u$ :
  - Build alternating tree using BFS
  - If no augmenting path, update labels to expose new equality edges
  - Augment matching along found path
3. Continue until all vertices are matched.

## Example

Weights:

|    | v1 | v2 | v3 |
|----|----|----|----|
| u1 | 3  | 2  | 1  |
| u2 | 2  | 4  | 6  |
| u3 | 3  | 5  | 3  |

Goal: maximize total weight

Optimal matching:

- $u1 \rightarrow v1$  (3)
- $u2 \rightarrow v3$  (6)
- $u3 \rightarrow v2$  (5)

Total weight =  $3 + 6 + 5 = 14$

### Tiny Code (Python-like Pseudocode)

```
import numpy as np
from scipy.optimize import linear_sum_assignment

def kuhn_munkres(weight_matrix):
    # Convert to cost by negation (Hungarian solves minimization)
    cost = -weight_matrix
    row_ind, col_ind = linear_sum_assignment(cost)
    total = weight_matrix[row_ind, col_ind].sum()
    return list(zip(row_ind, col_ind)), total

# Example
W = np.array([[3,2,1],[2,4,6],[3,5,3]])
match, total = kuhn_munkres(W)
print(match, total)  # [(0,0),(1,2),(2,1)], 14
```

### Why It Matters

- Solves maximum reward assignments in polynomial time
- Fundamental in:
  - Job-task allocation
  - Optimal pairing problems
  - Machine learning (e.g. Hungarian loss)
  - Game theory and economics

Many systems use this algorithm to maximize overall efficiency when assigning limited resources.

### A Gentle Proof (Why It Works)

- Maintain dual feasibility:  $l(u) + l(v) \geq w(u, v)$
- Maintain complementary slackness: matched edges satisfy equality
- By alternating updates between equality graph expansion and label adjustments, the algorithm ensures eventual feasibility and optimality
- The final matching satisfies strong duality, achieving maximum weight

## Try It Yourself

1. Build a  $3 \times 3$  profit matrix.
2. Initialize  $l(u)$  as row maxima,  $l(v) = 0$ .
3. Draw equality edges ( $l(u) + l(v) = w(u, v)$ ).
4. Find augmenting path  $\rightarrow$  augment  $\rightarrow$  update labels.
5. Repeat until matching is perfect.

## Test Cases

| Graph        | Size           | Max Weight            | Matching                |
|--------------|----------------|-----------------------|-------------------------|
| $3 \times 3$ | Random weights | 14                    | $[(0,0), (1,2), (2,1)]$ |
| $4 \times 4$ | Diagonal high  | sum(diag)             | Diagonal                |
| $5 \times 5$ | Uniform        | 5 $\times$ max weight | any                     |

## Complexity

| Step            | Time     |
|-----------------|----------|
| Label updates   | $O(V^2)$ |
| Matching phases | $O(V)$   |
| Total           | $O(V^3)$ |

The Kuhn–Munkres algorithm is the ultimate tool for maximum-weight assignments, blending geometry, duality, and combinatorics into one powerful optimization engine.

## 375 Blossom Algorithm

The Blossom Algorithm, developed by Jack Edmonds, is the foundational algorithm for finding maximum matchings in general graphs, including non-bipartite ones. It introduced the concept of blossoms (odd-length cycles) and showed that maximum matching is solvable in polynomial time, a landmark in combinatorial optimization.

## What Problem Are We Solving?

Given a general (not necessarily bipartite) graph  $G = (V, E)$ , find a maximum matching, a largest set of edges such that no two share a vertex.

Unlike bipartite graphs, odd-length cycles can block progress in standard augmenting path searches. The Blossom Algorithm contracts these cycles so that augmenting paths can still be found.

Formally, find  $M \subseteq E$  maximizing  $|M|$  such that each vertex is incident to at most one edge in  $M$ .

## How It Works (Plain Language)

The algorithm extends the augmenting path approach (used in bipartite matching) by adding one powerful idea: When you hit an odd-length cycle, treat it as a single vertex, a blossom.

Steps:

1. Initialize with an empty matching  $M = \emptyset$ .
2. Search for augmenting paths using BFS/DFS in the alternating tree:
  - Alternate between matched and unmatched edges.
  - If you reach a free vertex, augment (flip matched/unmatched edges).
3. If you encounter an odd cycle (a blossom), contract it into a single super-vertex.
  - Continue the search in the contracted graph.
  - Once an augmenting path is found, expand blossoms and update the matching.
4. Repeat until no augmenting path exists.

Each augmentation increases  $|M|$  by 1. When no augmenting path remains, the matching is maximum.

## Example

Graph:

- Vertices:  $A, B, C, D, E$
- Edges:  $(A, B), (B, C), (C, A), (B, D), (C, E)$

1. Start matching: empty
2. Build alternating tree:  $A \rightarrow B \rightarrow C \rightarrow A$  forms an odd cycle
3. Contract blossom  $(A, B, C)$  into one node

4. Continue search → find augmenting path through blossom
5. Expand blossom, adjust matching
6. Result: maximum matching includes edges  $(A, B), (C, E), (D, \dots)$

### Tiny Code (Python-Like Pseudocode)

```
# Using networkx for simplicity
import networkx as nx

def blossom_maximum_matching(G):
    return nx.max_weight_matching(G, maxcardinality=True)

# Example
G = nx.Graph()
G.add_edges_from([(0,1),(1,2),(2,0),(1,3),(2,4)])
match = blossom_maximum_matching(G)
print(match) # {(0,1), (2,4)}
```

### Why It Matters

- General graphs (non-bipartite) are common in real-world systems:
  - Social networks (friend pairings)
  - Molecular structure matching
  - Scheduling with constraints
  - Graph-theoretic proofs and optimization

The Blossom Algorithm proved that matching is polynomial-time solvable, a key milestone in algorithmic theory.

### A Gentle Proof (Why It Works)

By Berge's Lemma, a matching is maximum iff there's no augmenting path. The challenge is that augmenting paths can be hidden inside odd cycles.

Blossom contraction ensures that every augmenting path in the contracted graph corresponds to one in the original. After each augmentation,  $|M|$  strictly increases, so the algorithm terminates in polynomial time.

Correctness follows from maintaining:

- Alternating trees with consistent parity
- Contraction invariants (augmenting path preservation)
- Berge's condition across contractions and expansions

### Try It Yourself

1. Draw a triangle graph (3-cycle).
2. Run augmenting path search, find blossom.
3. Contract it into one node.
4. Continue search and expand after finding path.
5. Verify maximum matching.

### Test Cases

| Graph Type               | Matching Size | Method                |
|--------------------------|---------------|-----------------------|
| Triangle                 | 1             | Blossom contraction   |
| Square with diagonal     | 2             | Augmentation          |
| Pentagonal odd cycle     | 2             | Blossom               |
| Bipartite (sanity check) | As usual      | Matches Hopcroft–Karp |

### Complexity

| Phase                     | Time     |
|---------------------------|----------|
| Search (per augmentation) | $O(VE)$  |
| Augmentations             | $O(V)$   |
| Total                     | $O(V^3)$ |

The Blossom Algorithm was a revelation, showing how to tame odd cycles and extending matchings to all graphs, bridging combinatorics and optimization theory.

### 376 Edmonds' Blossom Shrinking

Edmonds' Blossom Shrinking is the core subroutine that powers the Blossom Algorithm, enabling augmenting-path search in non-bipartite graphs. It provides the crucial mechanism for contracting odd-length cycles (blossoms) so that hidden augmenting paths can be revealed and exploited.

## **What Problem Are We Solving?**

In non-bipartite graphs, augmenting paths can be obscured by odd-length cycles. Standard matching algorithms fail because they assume bipartite structure.

We need a way to handle odd cycles during search so that the algorithm can progress without missing valid augmentations.

Goal: Detect blossoms, shrink them into single vertices, and continue the search efficiently.

Given a matching  $M$  in graph  $G = (V, E)$ , and an alternating tree grown during a search, when a blossom is found:

- Shrink the blossom into a super-vertex
- Continue the search in the contracted graph
- Expand the blossom when an augmenting path is found

## **How It Works (Plain Language)**

Imagine running a BFS/DFS from a free vertex. You alternate between matched and unmatched edges to build an alternating tree.

If you find an edge connecting two vertices at the same level (both even depth):

- You've detected an odd cycle.
- That cycle is a blossom.

To handle it:

1. Identify the blossom (odd-length alternating cycle)
2. Shrink all its vertices into a single super-node
3. Continue the augmenting path search on this contracted graph
4. Once an augmenting path is discovered, expand the blossom and adjust the path accordingly
5. Augment along the expanded path

This shrinking maintains all valid augmenting paths and allows the algorithm to operate as if the blossom were one vertex.

## Example

Graph: Vertices:  $A, B, C, D, E$  Edges:  $(A, B), (B, C), (C, A), (B, D), (C, E)$

Suppose  $A, B, C$  form an odd cycle discovered during search.

1. Detect blossom:  $(A, B, C)$
2. Contract into a single vertex  $X$
3. Continue search on reduced graph
4. If augmenting path passes through  $X$ , expand back
5. Alternate edges within blossom to integrate path correctly

Result: A valid augmenting path is found and matching increases by one.

## Tiny Code (Python-Like)

```
def find_blossom(u, v, parent):
    # Find common ancestor
    path_u, path_v = set(), set()
    while u != -1:
        path_u.add(u)
        u = parent[u]
    while v not in path_u:
        path_v.add(v)
        v = parent[v]
    lca = v
    # Shrink blossom (conceptually)
    return lca
```

In practical implementations (like Edmonds' algorithm), this logic merges all nodes in the blossom into a single node and adjusts parent/child relationships.

## Why It Matters

This is the heart of general-graph matching. Without shrinking, the search may loop infinitely or fail to detect valid augmentations.

Blossom shrinking allows:

- Handling odd-length cycles
- Maintaining augmenting path invariants
- Guaranteeing polynomial time behavior

It is also one of the earliest uses of graph contraction in combinatorial optimization.

### A Gentle Proof (Why It Works)

By Berge's Lemma, a matching is maximum iff there is no augmenting path. If an augmenting path exists in  $G$ , then one exists in any contracted version of  $G$ .

Shrinking a blossom preserves augmentability:

- Every augmenting path in the original graph corresponds to one in the contracted graph.
- After expansion, the alternating pattern is restored correctly.

Therefore, contraction does not lose information, it simply simplifies the search.

### Try It Yourself

1. Draw a triangle  $A, B, C$  connected to other vertices  $D, E$ .
2. Build an alternating tree starting from a free vertex.
3. When you find an edge connecting two even-level vertices, mark the odd cycle.
4. Shrink the blossom, continue search, then expand once path is found.

Observe how this allows discovery of an augmenting path that was previously hidden.

### Test Cases

| Graph           | Description         | Augmentable After Shrink? |
|-----------------|---------------------|---------------------------|
| Triangle        | Single odd cycle    | Yes                       |
| Triangle + tail | Cycle + path        | Yes                       |
| Bipartite       | No odd cycle        | Shrink not needed         |
| Pentagon        | Blossom of length 5 | Yes                       |

### Complexity

Shrinking can be done in linear time relative to blossom size. Total algorithm remains  $O(V^3)$  when integrated into full matching search.

Edmonds' Blossom Shrinking is the conceptual leap that made maximum matching in general graphs tractable, transforming an intractable maze of cycles into a solvable structure through careful contraction and expansion.

## 377 Greedy Matching

Greedy Matching is the simplest way to approximate a maximum matching in a graph. Instead of exploring augmenting paths, it just keeps picking available edges that don't conflict, a fast, intuitive baseline for matching problems.

### What Problem Are We Solving?

In many real-world settings, job assignment, pairing users, scheduling, we need a set of edges such that no two share a vertex. This is a matching.

Finding the maximum matching exactly (especially in general graphs) can be expensive. But sometimes, we only need a good enough answer quickly.

A Greedy Matching algorithm provides a fast approximation:

- It won't always find the largest matching
- But it runs in  $O(E)$  and often gives a decent solution

Goal: Quickly build a maximal matching (no edge can be added).

### How It Works (Plain Language)

Start with an empty set  $M$  (the matching). Go through the edges one by one:

1. For each edge  $(u, v)$
2. If neither  $u$  nor  $v$  is already matched
3. Add  $(u, v)$  to  $M$

Continue until all edges have been checked.

This ensures no vertex appears in more than one edge, a valid matching. The result is maximal: you can't add any other edge without breaking the matching rule.

### Example

Graph edges:

$$E = (A, B), (B, C), (C, D), (D, E)$$

- Start:  $M = \emptyset$
- Pick  $(A, B) \rightarrow$  mark  $A, B$  as matched
- Skip  $(B, C) \rightarrow B$  is matched
- Pick  $(C, D) \rightarrow$  mark  $C, D$
- Skip  $(D, E) \rightarrow D$  is matched

Result:

$$M = (A, B), (C, D)$$

### Tiny Code (Python-Like)

```
def greedy_matching(graph):
    matched = set()
    matching = []
    for u, v in graph.edges:
        if u not in matched and v not in matched:
            matching.append((u, v))
            matched.add(u)
            matched.add(v)
    return matching
```

This simple loop runs in linear time over edges.

### Why It Matters

- Fast: runs in  $O(E)$
- Simple: easy to implement
- Useful baseline: good starting point for heuristic or hybrid approaches
- Guaranteed maximality: no edge can be added without breaking matching condition

Although it's not optimal, its result is at least half the size of a maximum matching:

$$|M_{\text{greedy}}| \geq \frac{1}{2}|M_{\text{max}}|$$

### A Gentle Proof (Why It Works)

Each greedy edge  $(u, v)$  blocks at most one edge from  $M_{\text{max}}$  (since both  $u$  and  $v$  are used). So for every chosen edge, we lose at most one from the optimal. Thus:

$$2|M_{\text{greedy}}| \geq |M_{\text{max}}|$$

Greedy achieves a  $1/2$ -approximation.

## Try It Yourself

1. Create a graph with 6 vertices and 7 edges.
2. Run greedy matching in different edge orders.
3. Observe how results differ, order can affect the final set.
4. Compare with an exact maximum matching (e.g., Hopcroft–Karp).

You'll see how simple decisions early can influence outcome size.

## Test Cases

| Graph           | Edges                          | Greedy Matching | Maximum Matching |
|-----------------|--------------------------------|-----------------|------------------|
| Path of 4       | (A, B), (B, C), (C, D)         | 2               | 2                |
| Triangle        | (A, B), (B, C), (C, A)         | 1               | 1                |
| Square          | (A, B), (B, C), (C, D), (D, A) | 2               | 2                |
| Star (5 leaves) | (C, 1)...(C, 5)                | 1               | 1                |

## Complexity

- Time:  $O(E)$
- Space:  $O(V)$

Works for any undirected graph. For directed graphs, edges must be treated as undirected or symmetric.

Greedy matching is a quick and practical approach when you need speed over perfection, a simple handshake strategy that pairs as many as possible before the clock runs out.

## 378 Stable Marriage (Gale–Shapley)

Stable Marriage (or the Stable Matching Problem) is a cornerstone of combinatorial optimization, where two equal-sized sets (e.g. men and women, jobs and applicants, hospitals and residents) must be paired so that no two participants prefer each other over their assigned partners. The Gale–Shapley algorithm finds such a stable matching in  $O(n^2)$  time.

## What Problem Are We Solving?

Given two sets:

- Set  $A = a_1, a_2, \dots, a_n$
- Set  $B = b_1, b_2, \dots, b_n$

Each member ranks all members of the other set in order of preference. We want to find a matching (a one-to-one pairing) such that:

There is no pair  $(a_i, b_j)$  where:

- $a_i$  prefers  $b_j$  over their current match, and
- $b_j$  prefers  $a_i$  over their current match.

Such a pair would be unstable, since they would rather be matched together.

Our goal: find a stable configuration, no incentive to switch partners.

## How It Works (Plain Language)

The Gale–Shapley algorithm uses proposals and rejections:

1. All  $a_i$  start unmatched.
2. While some  $a_i$  is free:
  - $a_i$  proposes to the most-preferred  $b_j$  not yet proposed to.
  - If  $b_j$  is free, accept the proposal.
  - If  $b_j$  is matched but prefers  $a_i$  over current partner, she “trades up” and rejects the old one.
  - Otherwise, she rejects  $a_i$ .
3. Continue until everyone is matched.

The process always terminates with a stable matching.

---

|   |                 |
|---|-----------------|
| A | Preference List |
|---|-----------------|

---

### Example

Let's say we have:

|    |                 |
|----|-----------------|
| A  | Preference List |
| A1 | B1, B2, B3      |
| A2 | B2, B1, B3      |
| A3 | B3, B1, B2      |

---

|    |                 |
|----|-----------------|
| B  | Preference List |
| B1 | A2, A1, A3      |
| B2 | A1, A2, A3      |
| B3 | A1, A2, A3      |

---

Step by step:

- A1 → B1, B1 free → match (A1, B1)
- A2 → B2, B2 free → match (A2, B2)
- A3 → B3, B3 free → match (A3, B3)

Stable matching: {(A1, B1), (A2, B2), (A3, B3)}

No two prefer each other over current partners → Stable.

### Tiny Code (Python-Like)

```
def gale_shapley(A_prefs, B_prefs):  
    free_A = list(A_prefs.keys())  
    engaged = {}  
    next_choice = {a: 0 for a in A_prefs}  
  
    while free_A:  
        a = free_A.pop(0)  
        b = A_prefs[a][next_choice[a]]  
        next_choice[a] += 1  
        if b not in engaged:  
            engaged[b] = a  
        else:  
            current_a = engaged[b]  
            if A_prefs[current_a][next_choice[current_a]] == b:  
                next_choice[current_a] += 1  
            else:  
                next_choice[a] += 1
```

```

if b not in engaged:
    engaged[b] = a
else:
    current = engaged[b]
    if B_prefs[b].index(a) < B_prefs[b].index(current):
        engaged[b] = a
        free_A.append(current)
    else:
        free_A.append(a)
return {v: k for k, v in engaged.items()}

```

## Why It Matters

- Guarantees a stable solution.
- Always terminates after at most  $n^2$  proposals.
- If one side proposes (say  $A$ ), the result is optimal for  $A$  and pessimal for  $B$ .
- Forms the foundation of real-world systems like the National Residency Matching Program (NRMP) and school assignments.

## A Gentle Proof (Why It Works)

Each proposal is made once per pair → at most  $n^2$  proposals.

No cycles:

- $b$  always stays with the best proposer so far
- Once rejected,  $a$  cannot improve, ensuring termination

Stability: Suppose  $(a_i, b_j)$  is a blocking pair. Then  $a_i$  must have proposed to  $b_j$  and was rejected. That means  $b_j$  preferred her current match. So  $(a_i, b_j)$  cannot be blocking → contradiction.

Thus, the final matching is stable.

## Try It Yourself

1. Create two sets of 3 elements with ranked lists.
2. Run Gale–Shapley twice, once with  $A$  proposing, once with  $B$  proposing.
3. Compare the two matchings, see who benefits.
4. Explore what happens if preference lists are incomplete.

## Test Cases

| Case                | Result                    | Stable? |
|---------------------|---------------------------|---------|
| Equal preferences   | Multiple stable matchings | Yes     |
| Cyclic preferences  | Algorithm converges       | Yes     |
| One-sided identical | Deterministic output      | Yes     |

## Complexity

- Time:  $O(n^2)$
- Space:  $O(n)$

Scalable for small to medium systems, widely used in practice.

Stable marriage shows how simple local rules (propose, reject, keep best) can produce global stability, a harmony of preference and fairness.

## 379 Weighted b-Matching

Weighted b-Matching generalizes standard matching by allowing each node to be matched to up to  $b(v)$  partners instead of just one. When edges carry weights, the goal is to find a maximum-weight subset of edges such that the degree of each vertex does not exceed its capacity  $b(v)$ .

### What Problem Are We Solving?

Given a graph  $G = (V, E)$  with:

- Edge weights  $w(e)$  for each  $e \in E$
- Capacity constraint  $b(v)$  for each vertex  $v \in V$

Find a subset  $M \subseteq E$  that:

- Maximizes total weight

$$W(M) = \sum_{e \in M} w(e)$$

- Satisfies degree constraints

$$\forall v \in V, \quad \deg_M(v) \leq b(v)$$

If  $b(v) = 1$  for all  $v$ , this becomes the standard maximum-weight matching.

## How It Works (Plain Language)

Think of each vertex  $v$  as having  $b(v)$  slots available for connections. We want to pick edges to fill these slots to maximize total edge weight, without exceeding any vertex's limit.

The weighted b-matching problem can be solved by:

1. Reduction to flow: Convert matching to a network flow problem:
  - Each edge becomes a flow connection with capacity 1 and cost  $= -w(e)$
  - Vertex capacities become flow limits
  - Solve via min-cost max-flow
2. Linear Programming: Relax constraints and solve with LP or primal-dual algorithms
3. Approximation: For large sparse graphs, greedy heuristics can achieve near-optimal solutions

## Example

Suppose:

- Vertices:  $V = A, B, C$
- Edges and weights:  $w(A, B) = 5, ; w(A, C) = 4, ; w(B, C) = 3$
- Capacities:  $b(A) = 2, ; b(B) = 1, ; b(C) = 1$

We can pick:

- $(A, B)$  weight 5
- $(A, C)$  weight 4

Total weight = 9 Valid since  $\deg(A) = 2, \deg(B) = 1, \deg(C) = 1$

If  $b(A) = 1$ , we'd pick only  $(A, B) \rightarrow$  weight 5

## Tiny Code (Python-Like Pseudocode)

```
import networkx as nx

def weighted_b_matching(G, b):
    # Convert to flow network
    flow_net = nx.DiGraph()
    source, sink = 's', 't'
```

```

for v in G.nodes:
    flow_net.add_edge(source, v, capacity=b[v], weight=0)
    flow_net.add_edge(v, sink, capacity=b[v], weight=0)

for (u, v, data) in G.edges(data=True):
    w = -data['weight'] # negate for min-cost
    flow_net.add_edge(u, v, capacity=1, weight=w)
    flow_net.add_edge(v, u, capacity=1, weight=w)

flow_dict = nx.max_flow_min_cost(flow_net, source, sink)
# Extract edges with flow=1 between vertex pairs
return [(u, v) for u, nbrs in flow_dict.items() for v, f in nbrs.items() if f > 0 and u < v]

```

## Why It Matters

- Models resource allocation with limits (e.g. each worker can handle  $b$  tasks)
- Extends classic matchings to capacitated networks
- Foundation for:
  - Task assignment with quotas
  - Scheduling with multi-capacity
  - Clustering with degree constraints

## A Gentle Proof (Why It Works)

Each vertex  $v$  has degree constraint  $b(v)$ :

$$\sum_{e \ni v} x_e \leq b(v)$$

Each edge is chosen at most once:

$$x_e \in \{0, 1\}$$

The optimization:

$$\max \sum_{e \in E} w(e)x_e$$

subject to constraints above, forms an integer linear program. Relaxation via network flow ensures optimal integral solution for bipartite graphs.

## Try It Yourself

1. Build a triangle graph with different edge weights.
2. Set  $b(v) = 2$  for one vertex, 1 for others.
3. Compute by hand which edges yield maximum weight.
4. Implement with a flow solver and verify.

## Test Cases

| Graph            | Capacities | Result            | Weight            |
|------------------|------------|-------------------|-------------------|
| Triangle (5,4,3) | {2,1,1}    | {(A,B),(A,C)}     | 9                 |
| Square           | {1,1,1,1}  | Standard matching | Max weight sum    |
| Line (A-B-C)     | {2,1,2}    | {(A,B),(B,C)}     | Sum of both edges |

## Complexity

- Time:  $O(V^3)$  using min-cost max-flow
- Space:  $O(V + E)$

Efficient for medium graphs; scalable via heuristics for larger ones.

Weighted b-matching balances optimality and flexibility, allowing richer allocation models than one-to-one pairings.

## 380 Maximal Matching

Maximal Matching is a greedy approach that builds a matching by adding edges one by one until no more can be added without breaking the matching condition. Unlike *maximum matchings* (which maximize size or weight), a *maximal* matching simply ensures no edge can be added, it's locally optimal, not necessarily globally optimal.

### What Problem Are We Solving?

We want to select a set of edges  $M \subseteq E$  such that:

1. No two edges in  $M$  share a vertex

$$\forall (u, v), (x, y) \in M, ; u, v \cap x, y = \emptyset$$

2.  $M$  is maximal: no additional edge from  $E$  can be added without violating (1).

Goal: Find any maximal matching, not necessarily the largest one.

## How It Works (Plain Language)

The algorithm is greedy and simple:

1. Start with an empty matching  $M = \emptyset$
2. Iterate through edges  $(u, v)$
3. If neither  $u$  nor  $v$  are matched yet, add  $(u, v)$  to  $M$
4. Continue until all edges are processed

At the end,  $M$  is maximal: every unmatched edge touches a vertex already matched.

## Example

Graph:

```
A -- B -- C  
|  
D
```

Edges:  $(A, B), (B, C), (A, D)$

Step-by-step:

- Pick  $(A, B) \rightarrow$  mark A, B matched
- Skip  $(B, C)$  (B already matched)
- Skip  $(A, D)$  (A already matched)

Result:  $M = (A, B)$

Another valid maximal matching:  $(A, D), (B, C)$  Not unique, but all maximal sets share the property that no more edges can be added.

## Tiny Code (Python)

```
def maximal_matching(edges):  
    matched = set()  
    M = []  
    for u, v in edges:  
        if u not in matched and v not in matched:  
            M.append((u, v))  
            matched.add(u)  
            matched.add(v)
```

```

    return M

edges = [('A', 'B'), ('B', 'C'), ('A', 'D')]
print(maximal_matching(edges)) # [(A', 'B')]

```

## Why It Matters

- Fast approximation: Maximal matching is a 2-approximation for the maximum matching (it's at least half as large).
- Building block: Used in distributed and parallel algorithms where global optimization is too expensive.
- Preprocessing step: Reduces problem size in flow, scheduling, and network design.

## A Gentle Proof (Why It Works)

If an edge  $(u, v)$  is not in  $M$ , at least one of its endpoints is already matched. Thus, adding  $(u, v)$  would break the matching condition. This ensures maximality, no more edges can be safely added.

Formally, for every  $(u, v) \notin M$ :

$$u \in V_M \vee v \in V_M$$

where  $V_M$  is the set of matched vertices.

Therefore,  $M$  is maximal.

## Try It Yourself

1. Draw a small graph with 4–6 vertices.
2. Apply the greedy process in different edge orders.
3. Compare resulting matchings, note they may differ in size, but all are maximal.
4. Try to add an extra edge, see why it violates matching rules.

## Test Cases

| Graph          | Edges                            | Maximal Matching   | Size |
|----------------|----------------------------------|--------------------|------|
| Path A–B–C     | $\{(A,B), (B,C)\}$               | $\{(A,B)\}$        | 1    |
| Star A–B,C,D   | $\{(A,B), (A,C), (A,D)\}$        | $\{(A,B)\}$        | 1    |
| Square A–B–C–D | $\{(A,B), (B,C), (C,D), (D,A)\}$ | $\{(A,B), (C,D)\}$ | 2    |

## Complexity

- Time:  $O(E)$ , single pass through edges
- Space:  $O(V)$  for vertex match tracking

Fast, simple, and effective for large graphs where optimal matchings are too expensive.

Maximal matching gives you a quick and safe pairing, not always perfect, but always *done*.

## Section 39. Tree Algorithms

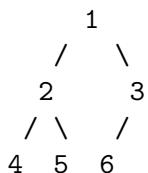
### 381 Euler Tour LCA

Euler Tour LCA transforms the problem of finding the Lowest Common Ancestor (LCA) into a Range Minimum Query (RMQ) problem by recording an Euler tour of the tree. It's elegant, precomputable, and fast, perfect for multiple LCA queries.

#### What Problem Are We Solving?

Given a rooted tree and two nodes  $u$  and  $v$ , we want to find their lowest common ancestor, the deepest node that is an ancestor of both.

For example, in a tree rooted at 1:



The LCA of (4, 5) is 2, and LCA of (4, 6) is 1.

We want to answer many such queries efficiently.

## How It Works (Plain Language)

1. Perform a DFS traversal (Euler tour):
  - Record each node when you enter it.
  - When you backtrack, record it again.
2. Store the depth of each node when visited.
3. For each node, remember its first occurrence in the Euler tour.
4. For query  $(u, v)$ :
  - Find the first occurrences of  $u$  and  $v$  in the tour.
  - Take the range between them.
  - The node with minimum depth in that range is the LCA.

So, LCA reduces to an RMQ over depths.

## Example

Tree (root = 1):

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

Euler tour: [1, 2, 4, 2, 5, 2, 1, 3, 6, 3, 1]

Depth array: [0, 1, 2, 1, 2, 1, 0, 1, 2, 1, 0]

First occurrences: 1 → 0, ; 2 → 1, ; 3 → 7, ; 4 → 2, ; 5 → 4, ; 6 → 8

Query: LCA(4,5)

- First(4) = 2, First(5) = 4
- Depth range [2..4] = [2,1,2]
- Minimum depth = 1 → node = 2

So LCA(4,5) = 2.

## Tiny Code (Python)

```
def euler_tour_lca(graph, root):
    n = len(graph)
    euler, depth, first = [], [], {}

    def dfs(u, d):
        first.setdefault(u, len(euler))
        euler.append(u)
        depth.append(d)
        for v in graph[u]:
            dfs(v, d + 1)
            euler.append(u)
            depth.append(d)

    dfs(root, 0)
    return euler, depth, first

def build_rmq(depth):
    n = len(depth)
    log = [0] * (n + 1)
    for i in range(2, n + 1):
        log[i] = log[i // 2] + 1
    k = log[n]
    st = [[0] * (k + 1) for _ in range(n)]
    for i in range(n):
        st[i][0] = i
    j = 1
    while (1 << j) <= n:
        i = 0
        while i + (1 << j) <= n:
            left = st[i][j - 1]
            right = st[i + (1 << (j - 1))][j - 1]
            st[i][j] = left if depth[left] < depth[right] else right
            i += 1
        j += 1
    return st, log

def query_lca(u, v, euler, depth, first, st, log):
    l, r = first[u], first[v]
    if l > r:
        l, r = r, l
```

```

j = log[r - 1 + 1]
left = st[1][j]
right = st[r - (1 << j) + 1][j]
return euler[left] if depth[left] < depth[right] else euler[right]

```

## Why It Matters

- Reduces LCA to RMQ, allowing  $O(1)$  query with  $O(n \log n)$  preprocessing
- Easy to combine with Segment Trees, Sparse Tables, or Cartesian Trees
- Essential for:
  - Tree DP with ancestor relationships
  - Distance queries:

$$\text{dist}(u, v) = \text{depth}(u) + \text{depth}(v) - 2 \times \text{depth}(\text{LCA}(u, v))$$

- Path queries in heavy-light decomposition

## A Gentle Proof (Why It Works)

In DFS traversal:

- Every ancestor appears before and after descendants.
- The first common ancestor to reappear between two nodes' first occurrences is their LCA.

Therefore, the node with minimum depth between the first appearances corresponds exactly to the lowest common ancestor.

## Try It Yourself

1. Construct a tree and perform an Euler tour manually.
2. Write down depth and first occurrence arrays.
3. Pick pairs  $(u, v)$ , find the minimum depth in their range.
4. Confirm LCA correctness.

## Test Cases

| Query    | Expected | Reason                  |
|----------|----------|-------------------------|
| LCA(4,5) | 2        | Both under 2            |
| LCA(4,6) | 1        | Root is common ancestor |
| LCA(2,3) | 1        | Different subtrees      |
| LCA(6,3) | 3        | One is ancestor         |

## Complexity

- Preprocessing:  $O(n \log n)$
- Query:  $O(1)$
- Space:  $O(n \log n)$

Euler Tour LCA shows how tree problems become array problems, a perfect example of structural transformation in algorithms.

## 382 Binary Lifting LCA

Binary Lifting is a fast and elegant technique to find the Lowest Common Ancestor (LCA) of two nodes in a tree using precomputed jumps to ancestors at powers of two. It turns ancestor queries into simple bit operations, giving  $O(\log n)$  per query.

### What Problem Are We Solving?

Given a rooted tree and two nodes  $u$  and  $v$ , we want to find their lowest common ancestor (LCA), the deepest node that is an ancestor of both.

Naively, we could walk up one node at a time, but that's  $O(n)$  per query. With binary lifting, we jump exponentially up the tree, reducing the cost to  $O(\log n)$ .

### How It Works (Plain Language)

Binary lifting precomputes for each node:

$$\text{up}[v][k] = \text{the } 2^k\text{-th ancestor of } v$$

So  $\text{up}[v][0]$  is the parent,  $\text{up}[v][1]$  is the grandparent,  $\text{up}[v][2]$  is the ancestor 4 levels up, and so on.

The algorithm proceeds in three steps:

1. Preprocess:

- Run DFS from the root.
- Record each node's depth.
- Fill table  $\text{up}[v][k]$ .

2. Equalize Depths:

- If one node is deeper, lift it up to match the shallower one.

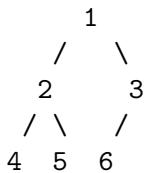
3. Lift Together:

- Jump both nodes up together (largest powers first) until their ancestors match.

The meeting point is the LCA.

### Example

Tree:



Binary lifting table ( $\text{up}[v][k]$ ):

| v | $\text{up}[v][0]$ | $\text{up}[v][1]$ | $\text{up}[v][2]$ |
|---|-------------------|-------------------|-------------------|
| 1 | -                 | -                 | -                 |
| 2 | 1                 | -                 | -                 |
| 3 | 1                 | -                 | -                 |
| 4 | 2                 | 1                 | -                 |
| 5 | 2                 | 1                 | -                 |
| 6 | 3                 | 1                 | -                 |

Query: LCA(4,5)

- Depth(4)=2, Depth(5)=2
- Lift 4,5 together  $\rightarrow$  parents (2,2) match  $\rightarrow$  LCA = 2

Query: LCA(4,6)

- Depth(4)=2, Depth(6)=2
- Lift 4→2, 6→3 → not equal
- Lift 2→1, 3→1 → LCA = 1

### Tiny Code (Python)

```

LOG = 20 # enough for n up to ~1e6

def preprocess(graph, root):
    n = len(graph)
    up = [[-1] * LOG for _ in range(n)]
    depth = [0] * n

    def dfs(u, p):
        up[u][0] = p
        for k in range(1, LOG):
            if up[u][k-1] != -1:
                up[u][k] = up[up[u][k-1]][k-1]
        for v in graph[u]:
            if v != p:
                depth[v] = depth[u] + 1
                dfs(v, u)
    dfs(root, -1)
    return up, depth

def lca(u, v, up, depth):
    if depth[u] < depth[v]:
        u, v = v, u
    diff = depth[u] - depth[v]
    for k in range(LOG):
        if diff & (1 << k):
            u = up[u][k]
    if u == v:
        return u
    for k in reversed(range(LOG)):
        if up[u][k] != up[v][k]:
            u = up[u][k]
            v = up[v][k]
    return up[u][0]

```

## Why It Matters

- Handles large trees with many queries efficiently
- Enables ancestor jumps, k-th ancestor queries, distance queries, etc.
- Fundamental in competitive programming, tree DP, and path queries

## A Gentle Proof (Why It Works)

If  $u$  and  $v$  have different depths, lifting the deeper node by  $2^k$  steps ensures we equalize their depth quickly.

Once aligned, lifting both simultaneously ensures we never skip the LCA, since jumps are made only when ancestors differ.

Eventually, both meet at their lowest shared ancestor.

## Try It Yourself

1. Build a small tree with 7 nodes.
2. Compute  $\text{up}[v][k]$  manually.
3. Pick pairs and simulate lifting step-by-step.
4. Verify with naive path-to-root comparison.

## Test Cases

| Query             | Expected | Reason          |
|-------------------|----------|-----------------|
| $\text{LCA}(4,5)$ | 2        | same parent     |
| $\text{LCA}(4,6)$ | 1        | across subtrees |
| $\text{LCA}(2,3)$ | 1        | root ancestor   |
| $\text{LCA}(6,3)$ | 3        | ancestor        |

## Complexity

- Preprocessing:  $O(n \log n)$
- Query:  $O(\log n)$
- Space:  $O(n \log n)$

Binary lifting turns tree ancestry into bitwise jumping, offering a clear path from root to ancestor in logarithmic time.

## 383 Tarjan's LCA (Offline DSU)

Tarjan's LCA algorithm answers multiple LCA queries offline using Disjoint Set Union (DSU) with path compression. It traverses the tree once (DFS) and merges subtrees, answering all queries in  $O(n + q\alpha(n))$  time.

### What Problem Are We Solving?

Given a rooted tree and multiple queries  $(u, v)$ , we want the lowest common ancestor of each pair.

Unlike binary lifting (which works online), Tarjan's algorithm works offline:

- We know all queries in advance.
- We answer them in one DFS traversal using a union-find structure.

### How It Works (Plain Language)

We process the tree bottom-up:

1. Start a DFS from the root.
2. Each node begins as its own set in DSU.
3. After visiting a child, union it with its parent.
4. When a node is fully processed, mark it as visited.
5. For each query  $(u, v)$ :
  - If the other node has been visited, then  $\text{find}(v)$  (or  $\text{find}(u)$ ) gives the LCA.

This works because the DSU structure merges all processed ancestors, so  $\text{find}(v)$  always points to the lowest processed ancestor common to both.

## Example

Tree (root = 1):

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

Queries: (4,5), (4,6), (3,6)

Process:

- DFS(1): visit 2, visit 4
  - Query(4,5): 5 not visited → skip
  - Backtrack → union(4,2)
- DFS(5): mark visited, union(5,2)
  - Query(4,5): 4 visited → find(4) = 2 → LCA(4,5)=2
- DFS(3), visit 6 → union(6,3)
  - Query(4,6): 4 visited, find(4) = 2, find(6) = 3 → no LCA yet
  - Query(3,6): both visited → find(3) = 3 → LCA(3,6)=3
- Backtrack 2→1, 3→1 → LCA(4,6)=1

## Tiny Code (Python)

```
from collections import defaultdict

class DSU:
    def __init__(self, n):
        self.parent = list(range(n))
    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]
    def union(self, a, b):
```

```

        self.parent[self.find(a)] = self.find(b)

def tarjan_lca(tree, root, queries):
    n = len(tree)
    dsu = DSU(n)
    visited = [False] * n
    ancestor = [0] * n
    ans = []
    qmap = defaultdict(list)
    for u, v in queries:
        qmap[u].append(v)
        qmap[v].append(u)
    def dfs(u):
        ancestor[u] = u
        for v in tree[u]:
            dfs(v)
            dsu.union(v, u)
            ancestor[dsu.find(u)] = u
        visited[u] = True
        for v in qmap[u]:
            if visited[v]:
                ans[(u, v)] = ancestor[dsu.find(v)]
    dfs(root)
    return ans

```

## Why It Matters

- Handles many LCA queries efficiently in one traversal
- No need for depth or binary lifting table
- Ideal for offline batch queries

Applications:

- Tree queries in competitive programming
- Offline graph analysis
- Dynamic connectivity in rooted structures

## A Gentle Proof (Why It Works)

Once a node  $u$  is processed:

- All its descendants are merged into its DSU set.

- $\text{find}(u)$  always returns the lowest ancestor processed so far.

When visiting query  $(u, v)$ :

- If  $v$  is already visited, then  $\text{find}(v)$  is the lowest common ancestor of  $u$  and  $v$ .

The invariant:

$$\text{ancestor}[\text{find}(v)] = \text{LCA}(u, v)$$

is maintained by post-order traversal and union operations.

### Try It Yourself

1. Draw a small tree and list all queries.
2. Perform DFS manually.
3. Track unions and ancestor updates.
4. Record LCA when both nodes visited.

### Test Cases

| Query   | Expected | Reason          |
|---------|----------|-----------------|
| $(4,5)$ | 2        | same subtree    |
| $(4,6)$ | 1        | across subtrees |
| $(3,6)$ | 3        | ancestor        |

### Complexity

- Time:  $O(n + q\alpha(n))$
- Space:  $O(n + q)$

Efficient for batch queries; each union and find is nearly constant-time.

Tarjan's LCA transforms the problem into a union-find dance, answering all ancestor questions in a single elegant DFS sweep.

## 384 Heavy-Light Decomposition

Heavy-Light Decomposition (HLD) splits a tree into heavy paths and light edges so that any path between two nodes can be broken into  $O(\log n)$  segments. It's the backbone for path queries, sum, max, min, or updates, using segment trees or Fenwick trees.

## What Problem Are We Solving?

In many problems, we need to answer queries like:

- What's the sum or max on the path from  $u$  to  $v$ ?
- How to update values along a path?
- What's the minimum edge weight between two nodes?

Naively, walking the path is  $O(n)$  per query. With HLD, we decompose paths into few contiguous segments, reducing queries to  $O(\log^2 n)$  (or better with LCA precomputation).

## How It Works (Plain Language)

Every node chooses one heavy child (the one with largest subtree). All other edges are light. This guarantees:

- Each node is part of exactly one heavy path.
- Every root-to-leaf path crosses  $O(\log n)$  light edges.

So we can represent each heavy path as a contiguous segment in an array, and map tree queries → array range queries.

## Steps

### 1. DFS 1:

- Compute subtree size for each node.
- Mark heavy child = largest subtree child.

### 2. DFS 2:

- Assign head of heavy path.
- Map each node to a position in linear order.

### 3. Path Query ( $u, v$ ):

- While  $u$  and  $v$  are not in same heavy path:
  - Always move the deeper head upward.
  - Query segment tree on that segment.
  - Jump  $u$  to parent of its head.
- When in same path: query segment from  $u$  to  $v$  (in order).

## Example

Tree:

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

- Subtree sizes:  $\text{size}(2) = 3$ ;  $\text{size}(3) = 3$  Heavy edges:  $(1, 2), (2, 4), (3, 6)$
- Heavy paths: Path 1:  $1 \rightarrow 2 \rightarrow 4$  Path 2:  $3 \rightarrow 6$  Light edges:  $(1, 3), (2, 5), (3, 7)$

Any query path  $(4, 7)$  crosses at most  $\log n$  segments:

$4 \rightarrow 2 \rightarrow 1 \mid 1 \rightarrow 3 \mid 3 \rightarrow 7$

## Tiny Code (Python)

```
def dfs1(u, p, g, size, heavy):
    size[u] = 1
    max_sub = 0
    for v in g[u]:
        if v == p: continue
        dfs1(v, u, g, size, heavy)
        size[u] += size[v]
        if size[v] > max_sub:
            max_sub = size[v]
            heavy[u] = v

def dfs2(u, p, g, head, pos, cur_head, order, heavy):
    head[u] = cur_head
    pos[u] = len(order)
    order.append(u)
    if heavy[u] != -1:
        dfs2(heavy[u], u, g, head, pos, cur_head, order, heavy)
    for v in g[u]:
        if v == p or v == heavy[u]:
```

```

        continue
dfs2(v, u, g, head, pos, v, order, heavy)

def query_path(u, v, head, pos, depth, segtree, lca):
    res = 0
    while head[u] != head[v]:
        if depth[head[u]] < depth[head[v]]:
            u, v = v, u
        res += segtree.query(pos[head[u]], pos[u])
        u = parent[head[u]]
    if depth[u] > depth[v]:
        u, v = v, u
    res += segtree.query(pos[u], pos[v])
    return res

```

## Why It Matters

- Turns path queries into range queries
- Pairs beautifully with segment trees or Fenwick trees
- Core for:
  - Path sum / max / min
  - Path updates
  - Lowest edge weight
  - Subtree queries (with Euler mapping)

## A Gentle Proof (Why It Works)

Every node moves at most  $\log n$  times up light edges, since subtree size at least halves each time. Within a heavy path, nodes form contiguous segments, so path queries can be broken into  $O(\log n)$  contiguous intervals.

Thus, overall query time is  $O(\log^2 n)$  (or  $O(\log n)$  with segment tree optimization).

## Try It Yourself

1. Draw a tree, compute subtree sizes.
2. Mark heavy edges to largest subtrees.
3. Assign head nodes and linear indices.
4. Try a query  $(u, v) \rightarrow$  decompose into path segments.

- Observe each segment is contiguous in linear order.

### Test Cases

| Query | Path      | Segments |
|-------|-----------|----------|
| (4,5) | 4→2→5     | 2        |
| (4,7) | 4→2→1→3→7 | 3        |
| (6,7) | 6→3→7     | 2        |

### Complexity

- Preprocessing:  $O(n)$
- Query:  $O(\log^2 n)$  (or  $O(\log n)$  with optimized segment tree)
- Space:  $O(n)$

Heavy-Light Decomposition bridges tree topology and array queries, enabling logarithmic-time operations on paths.

## 385 Centroid Decomposition

Centroid Decomposition is a divide-and-conquer method that recursively breaks a tree into smaller parts using centroids, nodes that balance the tree when removed. It transforms tree problems into logarithmic-depth recursion, enabling fast queries, updates, and path counting.

### What Problem Are We Solving?

For certain tree problems (distance queries, path sums, subtree coverage, etc.), we need to repeatedly split the tree into manageable pieces.

A centroid of a tree is a node such that, if removed, no resulting component has more than  $n/2$  nodes.

By recursively decomposing the tree using centroids, we build a centroid tree, a hierarchical structure capturing the balance of the original tree.

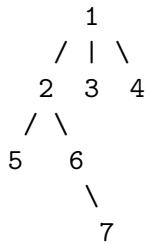
## How It Works (Plain Language)

1. Find the centroid:
  - Compute subtree sizes.
  - Pick the node whose largest child subtree  $n/2$ .
2. Record it as the root of this level's decomposition.
3. Remove the centroid from the tree.
4. Recurse on each remaining component (subtree).

Each node appears in  $O(\log n)$  levels, so queries or updates using this structure become logarithmic.

## Example

Tree (7 nodes):



Step 1:  $n = 7$ . Subtree sizes  $\rightarrow$  pick 2 (its largest child subtree 4). Centroid = 2.

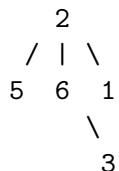
Step 2: Remove 2  $\rightarrow$  split into components:

- $\{5\}, \{6,7\}, \{1,3,4\}$

Step 3: Recursively find centroids in each component.

- $\{6,7\} \rightarrow$  centroid = 6
- $\{1,3,4\} \rightarrow$  centroid = 1

Centroid tree hierarchy:



## Tiny Code (Python)

```
def build_centroid_tree(graph):
    n = len(graph)
    size = [0] * n
    removed = [False] * n
    parent = [-1] * n

    def dfs_size(u, p):
        size[u] = 1
        for v in graph[u]:
            if v == p or removed[v]:
                continue
            dfs_size(v, u)
            size[u] += size[v]

    def find_centroid(u, p, n):
        for v in graph[u]:
            if v != p and not removed[v] and size[v] > n // 2:
                return find_centroid(v, u, n)
        return u

    def decompose(u, p):
        dfs_size(u, -1)
        c = find_centroid(u, -1, size[u])
        removed[c] = True
        parent[c] = p
        for v in graph[c]:
            if not removed[v]:
                decompose(v, c)
        return c

    root = decompose(0, -1)
    return parent # parent[c] is centroid parent
```

## Why It Matters

Centroid decomposition allows efficient solutions to many tree problems:

- Path queries: distance sums, min/max weights
- Point updates: affect only  $O(\log n)$  centroids

- Distance-based search: find nearest node of a type
- Divide-and-conquer DP on trees

It's especially powerful in competitive programming for problems like *count pairs within distance  $k$* , *color-based queries*, or *centroid tree construction*.

### A Gentle Proof (Why It Works)

Every decomposition step removes one centroid. By definition, each remaining subtree  $n/2$ . Thus, recursion depth is  $O(\log n)$ , and every node participates in at most  $O(\log n)$  subproblems.

Total complexity:

$$T(n) = T(n_1) + T(n_2) + \dots + O(n) \leq O(n \log n)$$

### Try It Yourself

1. Draw a small tree.
2. Compute subtree sizes.
3. Pick centroid (largest child  $n/2$ ).
4. Remove and recurse.
5. Build centroid tree hierarchy.

### Test Cases

| Tree                    | Centroid | Components   | Depth |
|-------------------------|----------|--------------|-------|
| Line (1–2–3–4–5)        | 3        | {1,2}, {4,5} | 2     |
| Star (1 center)         | 1        | leaves       | 1     |
| Balanced tree (7 nodes) | 2        | 3 components | 2     |

### Complexity

- Preprocessing:  $O(n \log n)$
- Query/Update (per node):  $O(\log n)$
- Space:  $O(n)$

Centroid Decomposition turns trees into balanced recursive hierarchies, a universal tool for logarithmic-time query and update systems.

## 386 Tree Diameter (DFS Twice)

The tree diameter is the longest path between any two nodes in a tree. A simple and elegant method to find it is to perform two DFS traversals, one to find the farthest node, and another to measure the longest path length.

### What Problem Are We Solving?

In a tree (an acyclic connected graph), we want to find the diameter, defined as:

$$\text{diameter} = \max_{u,v \in V} \text{dist}(u, v)$$

where  $\text{dist}(u, v)$  is the length (in edges or weights) of the shortest path between  $u$  and  $v$ .

This path always lies between two leaf nodes, and can be found using two DFS/BFS traversals.

### How It Works (Plain Language)

1. Pick any node (say 1).
2. Run DFS to find the farthest node from it → call it  $A$ .
3. Run DFS again from  $A$  → find the farthest node from  $A$  → call it  $B$ .
4. The distance between  $A$  and  $B$  is the diameter length.

Why this works:

- The first DFS ensures you start one end of the diameter.
- The second DFS stretches to the opposite end.

### Example

Tree:

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

1. DFS(1): farthest node = 4 (distance 2)
2. DFS(4): farthest node = 6 (distance 4)

So diameter = 4 → path: 4 → 2 → 1 → 3 → 6

### Tiny Code (Python)

```
from collections import defaultdict

def dfs(u, p, dist, graph):
    farthest = (u, dist)
    for v, w in graph[u]:
        if v == p:
            continue
        cand = dfs(v, u, dist + w, graph)
        if cand[1] > farthest[1]:
            farthest = cand
    return farthest

def tree_diameter(graph):
    start = 0
    a, _ = dfs(start, -1, 0, graph)
    b, diameter = dfs(a, -1, 0, graph)
    return diameter, (a, b)

# Example:
graph = defaultdict(list)
edges = [(0,1,1),(1,3,1),(0,2,1),(2,4,1),(2,5,1)]
for u,v,w in edges:
    graph[u].append((v,w))
    graph[v].append((u,w))

d, (a,b) = tree_diameter(graph)
print(d, (a,b)) # 4, path (3,5)
```

### Why It Matters

- Simple and linear time:  $O(n)$
- Works for weighted and unweighted trees
- Core for:
  - Tree center finding

- Dynamic programming on trees
- Network analysis (longest delay, max latency path)

## A Gentle Proof (Why It Works)

Let  $(u, v)$  be the true diameter endpoints. Any DFS from any node  $x$  finds a farthest node  $A$ . By triangle inequality on trees,  $A$  must be one endpoint of a diameter path. A second DFS from  $A$  finds the other endpoint  $B$ , ensuring  $\text{dist}(A, B)$  is the maximum possible.

## Try It Yourself

1. Draw a small tree.
2. Pick any starting node, run DFS to find farthest.
3. From that node, run DFS again.
4. Trace the path between these two nodes, it's always the longest.

## Test Cases

| Tree Type                | Diameter | Path                       |
|--------------------------|----------|----------------------------|
| Line (1–2–3–4)           | 3        | (1,4)                      |
| Star (1 center + leaves) | 2        | (leaf , leaf )             |
| Balanced binary          | 4        | leftmost to rightmost leaf |

## Complexity

- Time:  $O(n)$
- Space:  $O(n)$  recursion stack

Tree Diameter via two DFS is one of the cleanest tricks in graph theory, from any point, go far, then farther.

## 387 Tree DP (Subtree-Based Optimization)

Tree Dynamic Programming (Tree DP) is a technique that solves problems on trees by combining results from subtrees. Each node's result depends on its children, following a recursive pattern much like divide and conquer but on a tree.

## What Problem Are We Solving?

Many problems on trees ask for optimal values (sum, min, max, count) that depend on child subtrees, for example:

- Maximum path sum in a tree
- Size of largest independent set
- Number of ways to color a tree
- Sum of distances to all nodes

Tree DP provides a bottom-up approach where we compute results for each subtree and merge them upwards.

## How It Works (Plain Language)

1. Root the tree at an arbitrary node (often 1 or 0).
2. Define a DP state at each node based on its subtree.
3. Combine children's results recursively.
4. Return the result to the parent.

Example state:

$$dp[u] = f(dp[v_1], dp[v_2], \dots, dp[v_k])$$

where  $v_i$  are children of  $u$ .

## Example Problem

Find the size of the largest independent set (no two adjacent nodes chosen).

Recurrence:

$$dp[u][0] = \sum_{v \in \text{children}(u)} \max(dp[v][0], dp[v][1])$$

$$dp[u][1] = 1 + \sum_{v \in \text{children}(u)} dp[v][0]$$

## Tiny Code (Python)

```

from collections import defaultdict

graph = defaultdict(list)
edges = [(0,1),(0,2),(1,3),(1,4)]
for u,v in edges:
    graph[u].append(v)
    graph[v].append(u)

def dfs(u, p):
    incl = 1 # include u
    excl = 0 # exclude u
    for v in graph[u]:
        if v == p:
            continue
        inc_v, exc_v = dfs(v, u)
        incl += exc_v
        excl += max(incl, exc_v)
    return incl, excl

incl, excl = dfs(0, -1)
ans = max(incl, excl)
print(ans) # 3

```

This finds the largest independent set size (3 nodes).

## Why It Matters

Tree DP is foundational for:

- Counting: number of ways to assign labels, colors, or states
- Optimization: maximize or minimize cost over paths or sets
- Combinatorics: solve partition or constraint problems
- Game theory: compute win/loss states recursively

It turns global tree problems into local merges.

## A Gentle Proof (Why It Works)

Each node's subtree is independent once you know whether its parent is included or not. By processing children before parents (post-order DFS), we ensure all necessary data is ready when merging. This guarantees correctness via structural induction on tree size.

## Try It Yourself

1. Define a simple property (sum, count, max).
2. Write recurrence for a node in terms of its children.
3. Traverse via DFS, merge children's results.
4. Return value to parent, combine at root.

## Template (Generic)

```
def dfs(u, p):
    res = base_case()
    for v in children(u):
        if v != p:
            child = dfs(v, u)
            res = merge(res, child)
    return finalize(res)
```

## Complexity

- Time:  $O(n)$
- Space:  $O(n)$  (recursion)

Tree DP transforms recursive thinking into structured computation, one subtree at a time, building the answer from the leaves to the root.

## 388 Rerooting DP (Compute All Roots' Answers)

Rerooting DP is an advanced tree dynamic programming technique that computes answers for every possible root efficiently. Instead of recomputing from scratch, we reuse results by propagating information down and up the tree.

### What Problem Are We Solving?

Suppose we want the same value for every node as if that node were the root. Examples:

- Sum of distances from each node to all others
- Number of nodes in each subtree
- DP value that depends on subtree structure

A naive approach runs  $O(n)$  DP per root  $\rightarrow O(n^2)$  total. Rerooting reduces this to  $O(n)$  total.

## How It Works (Plain Language)

1. First pass (downward DP): Compute answers for each subtree assuming root = 0.
2. Second pass (rerooting): Use parent's info to update child's answer — effectively “rerooting” the DP at each node.

The key is combining results from children excluding the child itself, often using prefix–suffix merges.

## Example Problem

Compute sum of distances from each node to all others.

Recurrence:

- `subtree_size[u]` = number of nodes in subtree of `u`
- `dp[u]` = sum of distances from `u` to nodes in its subtree

First pass:

$$dp[u] = \sum_{v \in \text{children}(u)} (dp[v] + \text{subtree\_size}[v])$$

Second pass: When moving root from `u` to `v`:

$$dp[v] = dp[u] - \text{subtree\_size}[v] + (n - \text{subtree\_size}[v])$$

## Tiny Code (Python)

```
from collections import defaultdict

graph = defaultdict(list)
edges = [(0,1),(0,2),(2,3),(2,4)]
for u,v in edges:
    graph[u].append(v)
    graph[v].append(u)

n = 5
size = [1]*n
```

```

dp = [0]*n
ans = [0]*n

def dfs1(u, p):
    for v in graph[u]:
        if v == p: continue
        dfs1(v, u)
        size[u] += size[v]
        dp[u] += dp[v] + size[v]

def dfs2(u, p):
    ans[u] = dp[u]
    for v in graph[u]:
        if v == p: continue
        pu, pv = dp[u], dp[v]
        su, sv = size[u], size[v]

        dp[u] -= dp[v] + size[v]
        size[u] -= size[v]
        dp[v] += dp[u] + size[u]
        size[v] += size[u]

        dfs2(v, u)

        dp[u], dp[v] = pu, pv
        size[u], size[v] = su, sv

dfs1(0, -1)
dfs2(0, -1)

print(ans)

```

This computes the sum of distances for each node.

### Why It Matters

- Efficiently derive all-root answers
- Common in centroid problems, sum of distances, rerooting sums
- Reduces redundant recomputation dramatically

## A Gentle Proof (Why It Works)

Every DP depends only on local merges of subtrees. By removing a child subtree and adding the rest, we adjust the parent's value to reflect a new root. Induction over tree structure ensures correctness.

## Try It Yourself

1. Write your DP recurrence for a single root.
2. Identify what changes when rerooting to a child.
3. Apply push–pull updates using stored subtree info.
4. Collect final answers after traversal.

## Complexity

- Time:  $O(n)$  (two DFS passes)
- Space:  $O(n)$

Rerooting DP turns a global rerooting problem into a pair of local transformations, computing every node's perspective with elegant reuse.

## 389 Binary Search on Tree (Edge Weight Constraints)

Binary Search on Tree is a versatile strategy used to solve problems where we must find a threshold edge weight or path condition that satisfies a constraint. It combines DFS or BFS traversal with binary search over a numeric property, often edge weights or limits.

### What Problem Are We Solving?

Given a tree with weighted edges, we might want to answer questions like:

- What is the minimum edge weight so that the path between two nodes satisfies some property?
- What is the maximum allowed cost under which the tree remains connected?
- What is the smallest threshold that allows at least  $k$  nodes to be reachable?

A naive approach checks each possible weight. We can do better by binary searching over sorted edge weights.

## How It Works (Plain Language)

1. Sort the edge weights or define a numeric search range.
2. Binary search over possible threshold  $T$ .
3. For each mid value, traverse the tree (via DFS/BFS/Union-Find) including only edges satisfying a condition (e.g., weight  $\leq T$ ).
4. Check if the property holds.
5. Narrow the search interval.

The key is monotonicity: if a property holds for  $T$ , it holds for all larger/smaller values.

## Example Problem

Find the minimum edge weight threshold  $T$  so that all nodes become connected.

Algorithm:

1. Sort edges by weight.
2. Binary search  $T$  in  $[min_{w \text{ weight}}, max_{w \text{ weight}}]$ .
3. For each  $T$ , include edges with  $w \leq T$ .
4. Check if resulting graph is connected.
5. Adjust bounds accordingly.

## Tiny Code (Python)

```
def is_connected(n, edges, limit):
    parent = list(range(n))
    def find(x):
        if parent[x] != x:
            parent[x] = find(parent[x])
        return parent[x]
    def union(a, b):
        pa, pb = find(a), find(b)
        if pa != pb:
            parent[pb] = pa

    for u, v, w in edges:
        if w <= limit:
            union(u, v)
    return len({find(i) for i in range(n)}) == 1
```

```

def binary_search_tree_threshold(n, edges):
    weights = sorted(set(w for _, _, w in edges))
    lo, hi = 0, len(weights)-1
    ans = weights[-1]
    while lo <= hi:
        mid = (lo + hi) // 2
        if is_connected(n, edges, weights[mid]):
            ans = weights[mid]
            hi = mid - 1
        else:
            lo = mid + 1
    return ans

edges = [(0,1,4),(1,2,2),(0,2,5)]
print(binary_search_tree_threshold(3, edges)) # Output: 4

```

## Why It Matters

- Reduces complex “threshold” problems to logarithmic searches
- Works when solution space is monotonic
- Combines structural traversal with decision logic

Common applications:

- Path feasibility with limit
- Edge filtering by cost
- Tree queries with weight bounds

## A Gentle Proof (Why It Works)

Let property  $P(T)$  be true if condition holds for threshold  $T$ . If  $P(T)$  is monotonic (either always true beyond some point, or always false before), then binary search correctly converges to the minimal/maximal satisfying  $T$ .

## Try It Yourself

1. Define a monotonic property on weights (e.g. “connected under limit”).
2. Implement a decision function checking  $P(T)$ .
3. Wrap it in a binary search loop.
4. Return the minimal/maximal valid  $T$ .

## Complexity

- Sorting edges:  $O(E \log E)$
- Binary search:  $O(\log E)$  checks
- Each check:  $O(E\alpha(V))$  (Union-Find)
- Overall:  $O(E \log^2 E)$  or better

Binary search on trees is not about searching *within* the tree, but *over constraints* defined on the tree, a sharp technique for threshold-based reasoning.

## 390 Virtual Tree (Query Subset Construction)

A Virtual Tree is a compressed representation of a tree built from a selected subset of nodes, along with their Lowest Common Ancestors (LCAs), connected in the same hierarchical order as the original tree. It's used in query problems where we only need to reason about a small subset of nodes rather than the full tree.

### What Problem Are We Solving?

Suppose we have a large tree and a query gives us a small set  $S$  of nodes. We need to compute some property among nodes in  $S$  (like sum of distances, coverage, or DP over their paths). Traversing the entire tree each time is inefficient.

A Virtual Tree shrinks the full tree down to just the nodes in  $S$  plus their LCAs, keeping ancestry relationships intact.

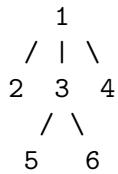
This reduces the problem to a much smaller tree, often  $O(|S| \log |S|)$  nodes instead of  $O(n)$ .

### How It Works (Plain Language)

1. Collect the query nodes  $S$ .
2. Sort  $S$  by Euler Tour order (entry time in DFS).
3. Insert LCAs of consecutive nodes to maintain structure.
4. Build edges between consecutive ancestors to form a tree.
5. Now process queries on this mini-tree instead of the full one.

## Example

Given tree:



Query nodes  $S = 2, 5, 6$ .

Their LCAs:

- $\text{LCA}(5, 6) = 3$
- $\text{LCA}(2, 3) = 1$

Virtual Tree Nodes = 1, 2, 3, 5, 6 Connect them according to parent-child relations: 1 → 2 1 → 3 3 → 5 3 → 6

This is your Virtual Tree.

## Tiny Code (C++ Style Pseudocode)

```
vector<int> build_virtual_tree(vector<int>& S, vector<int>& tin, auto lca) {
    sort(S.begin(), S.end(), [&](int a, int b) { return tin[a] < tin[b]; });
    vector<int> nodes = S;
    for (int i = 0; i + 1 < (int)S.size(); i++)
        nodes.push_back(lca(S[i], S[i+1]));
    sort(nodes.begin(), nodes.end(), [&](int a, int b) { return tin[a] < tin[b]; });
    nodes.erase(unique(nodes.begin(), nodes.end()), nodes.end());
    stack<int> st;
    vector<vector<int>> vt_adj(nodes.size());
    for (int u : nodes) {
        while (!st.empty() && !is_ancestor(st.top(), u))
            st.pop();
        if (!st.empty())
            vt_adj[st.top()].push_back(u);
        st.push(u);
    }
    return nodes;
}
```

This builds adjacency for the virtual tree using stack ancestry.

## Why It Matters

- Reduces large-tree queries to small subproblems
- Essential in problems with multiple subset queries
- Common in offline processing, rerooting DP, distance sum problems
- Works perfectly with LCA precomputation

## A Gentle Proof (Why It Works)

Each node in  $S$  must connect through its LCAs. By sorting in Euler order and maintaining a stack of ancestors, we ensure that:

- Parent-child relations are consistent
- No cycles or duplicates
- The resulting tree preserves ancestry

So the virtual tree is a minimal subtree connecting all nodes in  $S$ .

## Try It Yourself

1. Implement Euler Tour (get `tin[u]`).
2. Implement LCA (Binary Lifting or RMQ).
3. Build Virtual Tree for given  $S$ .
4. Apply your query logic (sum, count, DP).

## Complexity

- Sorting  $S$ :  $O(|S| \log |S|)$
- LCA calls:  $O(|S|)$
- Building structure:  $O(|S|)$
- Overall:  $O(|S| \log |S|)$  per query

A Virtual Tree is your “query-scaled tree”, a precise projection of the big tree onto the small world of your problem.

## Section 40. Advanced Graph Algorithms and Tricks

### 391 Topological DP (Dynamic Programming on DAG)

Topological DP is a dynamic programming technique on Directed Acyclic Graphs (DAGs). It computes values in dependency order, ensuring each node's state is calculated only after all its prerequisites. This method is a core tool for problems involving partial orders, longest paths, counting paths, and propagation across dependencies.

#### What Problem Are We Solving?

In DAGs, some nodes depend on others (edges point from dependencies to dependents). We often want to compute a DP value like:

- Longest path ending at each node
- Number of ways to reach a node
- Minimum cost to reach a node
- Accumulated value through dependencies

Because DAGs have no cycles, a topological order exists, allowing linear-time evaluation.

#### How It Works (Plain Language)

1. Topologically sort the DAG.
2. Initialize base cases (e.g. sources = 0 or 1).
3. Iterate in topo order, updating each node based on incoming edges.
4. Each node's value is final once processed.

This guarantees each dependency is resolved before use.

#### Example: Longest Path in DAG

Given DAG with edge weights  $w(u, v)$ , define DP:

$$dp[v] = \max_{(u,v) \in E} (dp[u] + w(u, v))$$

Base case:  $dp[\text{source}] = 0$

## Example DAG

```
1 → 2 → 4  
  3
```

Edges:

- $1 \rightarrow 2$  (1)
- $1 \rightarrow 3$  (2)
- $2 \rightarrow 4$  (1)
- $3 \rightarrow 4$  (3)

Topological order: [1, 2, 3, 4]

Compute:

- $dp[1] = 0$
- $dp[2] = \max(dp[1] + 1) = 1$
- $dp[3] = \max(dp[1] + 2) = 2$
- $dp[4] = \max(dp[2] + 1, dp[3] + 3) = \max(2, 5) = 5$

Result: Longest path length = 5

## Tiny Code (C++ Style)

```
vector<int> topo_sort(int n, vector<vector<int>>& adj) {  
    vector<int> indeg(n, 0), topo;  
    for (auto& u : adj)  
        for (int v : u) indeg[v]++;  
    queue<int> q;  
    for (int i = 0; i < n; i++) if (indeg[i] == 0) q.push(i);  
    while (!q.empty()) {  
        int u = q.front(); q.pop();  
        topo.push_back(u);  
        for (int v : adj[u])  
            if (--indeg[v] == 0) q.push(v);  
    }  
    return topo;  
}  
  
vector<int> topo_dp(int n, vector<vector<pair<int,int>>>& adj) {  
    auto order = topo_sort(n, ...);
```

```

vector<int> dp(n, INT_MIN);
dp[0] = 0;
for (int u : order)
    for (auto [v, w] : adj[u])
        dp[v] = max(dp[v], dp[u] + w);
return dp;
}

```

## Why It Matters

- Converts recursive dependencies into iterative computation
- Avoids redundant work
- Enables linear-time solutions for many DAG problems
- Works for counting, min/max, aggregation tasks

Common uses:

- Longest path in DAG
- Counting number of paths
- Minimum cost scheduling
- Project dependency planning

## A Gentle Proof (Why It Works)

A topological order guarantees:

- For every edge  $(u, v)$ ,  $u$  appears before  $v$ . Thus when processing  $v$ , all  $dp[u]$  for predecessors  $u$  are ready. This ensures correctness for any DP formula of the form:

$$dp[v] = f(dp[u] \mid (u, v) \in E)$$

## Try It Yourself

1. Compute number of paths from source to each node:  $dp[v] = \sum_{(u,v)} dp[u]$
2. Compute minimum cost if edges have weights
3. Build a longest chain of tasks with dependencies
4. Apply topological DP on SCC DAG (meta-graph)

## Complexity

- Topo sort:  $O(V + E)$
- DP propagation:  $O(V + E)$
- Total:  $O(V + E)$  time,  $O(V)$  space

Topological DP is how you bring order to dependency chaos, one layer, one node, one dependency at a time.

## 392 SCC Condensed Graph DP (Dynamic Programming on Meta-Graph)

SCC Condensed Graph DP applies dynamic programming to the condensation of a directed graph into a Directed Acyclic Graph (DAG), where each node represents a strongly connected component (SCC). This transforms a cyclic graph into an acyclic one, enabling topological reasoning, path aggregation, and value propagation across components.

### What Problem Are We Solving?

Many problems on directed graphs become complex due to cycles. Within each SCC, every node can reach every other, they are strongly connected. By condensing SCCs into single nodes, we obtain a DAG:

$$G' = (V', E')$$

where each  $v' \in V'$  is an SCC, and edges  $(u', v')$  represent transitions between components.

Once the graph is acyclic, we can run Topological DP to compute:

- Maximum or minimum value reaching each SCC
- Number of paths between components
- Aggregated scores, weights, or costs

### How It Works (Plain Language)

1. Find SCCs (using Tarjan or Kosaraju).
2. Build Condensed DAG: each SCC becomes a single node.
3. Aggregate initial values for each SCC (e.g. sum of weights).
4. Run DP over DAG in topological order, combining contributions from incoming edges.
5. Map back results to original nodes if needed.

This isolates cycles (internal SCCs) and manages dependencies cleanly.

## Example

Original graph  $G$ :

```
1 → 2 → 3
↑   ↓
5 ← 4
```

SCCs:

- $C_1 = \{1, 2, 4, 5\}$
- $C_2 = \{3\}$

Condensed graph:

```
C → C
```

If each node has weight  $w[i]$ , then:

$$dp[C_2] = \max_{(C_1, C_2)} (dp[C_1] + \text{aggregate}(C_1))$$

## Tiny Code (C++ Style)

```
int n;
vector<vector<int>> adj;
vector<int> comp, order;
vector<bool> vis;

void dfs1(int u) {
    vis[u] = true;
    for (int v : adj[u]) if (!vis[v]) dfs1(v);
    order.push_back(u);
}

void dfs2(int u, int c, vector<vector<int>>& radj) {
    comp[u] = c;
    for (int v : radj[u]) if (comp[v] == -1) dfs2(v, c, radj);
}
```

```

vector<int> scc_condense() {
    vector<vector<int>> radj(n);
    for (int u=0; u<n; ++u)
        for (int v : adj[u]) radj[v].push_back(u);

    vis.assign(n, false);
    for (int i=0; i<n; ++i) if (!vis[i]) dfs1(i);

    comp.assign(n, -1);
    int cid=0;
    for (int i=n-1;i>=0;--i){
        int u = order[i];
        if (comp[u]==-1) dfs2(u,cid++,radj);
    }

    return comp;
}

```

Then build condensed graph:

```

vector<vector<int>> dag(cid);
for (int u=0;u<n;++u)
    for (int v:adj[u])
        if (comp[u]!=comp[v])
            dag[comp[u]].push_back(comp[v]);

```

Run Topological DP on dag.

## Why It Matters

- Turns cyclic graphs into acyclic DAGs
- Allows DP, path counting, and aggregation in cyclic contexts
- Simplifies reasoning about reachability and influence
- Forms the basis for:
  - Dynamic condensation
  - Meta-graph optimization
  - Modular graph analysis

## A Gentle Proof (Why It Works)

1. Within each SCC, all nodes are mutually reachable.
2. Condensation merges these into single nodes, ensuring no cycles.
3. Edges between SCCs define a partial order, enabling topological sorting.
4. Any property defined recursively over dependencies can now be solved via DP on this order.

Formally, for each  $C_i$ :

$$dp[C_i] = f(dp[C_j] \mid (C_j, C_i) \in E', \text{value}(C_i))$$

## Try It Yourself

1. Assign a weight to each node; compute max sum path over SCC DAG.
2. Count distinct paths between SCCs.
3. Combine SCC detection + DP for weighted reachability.
4. Solve problems like:
  - “Maximum gold in a dungeon with teleport cycles”
  - “Dependency graph with feedback loops”

## Complexity

- SCC computation:  $O(V + E)$
- DAG construction:  $O(V + E)$
- Topo DP:  $O(V' + E')$
- Total:  $O(V + E)$

SCC Condensed Graph DP is the art of shrinking cycles into certainty, revealing a clean DAG beneath the tangled surface.

## 393 Eulerian Path

An Eulerian Path is a trail in a graph that visits every edge exactly once. If the path starts and ends at the same vertex, it is called an Eulerian Circuit. This concept lies at the heart of route planning, graph traversals, and network analysis.

## **What Problem Are We Solving?**

We want to find a path that uses every edge once, no repeats, no omissions.

In an undirected graph, an Eulerian Path exists if and only if:

- Exactly 0 or 2 vertices have odd degree
- The graph is connected

In a directed graph, it exists if and only if:

- At most one vertex has `outdegree - indegree = 1` (start)
- At most one vertex has `indegree - outdegree = 1` (end)
- All other vertices have equal in-degree and out-degree
- The graph is strongly connected (or connected when ignoring direction)

## **How Does It Work (Plain Language)**

1. Check degree conditions (undirected) or in/out-degree balance (directed).
2. Choose a start vertex:
  - For undirected: any odd-degree vertex (if exists), otherwise any vertex.
  - For directed: vertex with `outdeg = indeg + 1`, else any.
3. Apply Hierholzer's algorithm:
  - Walk edges greedily until stuck.
  - Backtrack and merge cycles into one path.
4. Reverse the constructed order for the final path.

## **Example (Undirected)**

Graph edges:

1-2, 2-3, 3-1, 2-4

Degrees:

- $\deg(1)=2$ ,  $\deg(2)=3$ ,  $\deg(3)=2$ ,  $\deg(4)=1$  Odd vertices: 2, 4  $\rightarrow$  path exists (starts at 2, ends at 4)

Eulerian Path: 2  $\rightarrow$  1  $\rightarrow$  3  $\rightarrow$  2  $\rightarrow$  4

### Example (Directed)

Edges:

A → B, B → C, C → A, C → D

Degrees:

- A: out=1, in=1
- B: out=1, in=1
- C: out=2, in=1
- D: out=0, in=1

Start: C (out=in+1), End: D (in=out+1)

Eulerian Path: C → A → B → C → D

### Tiny Code (C++)

```
vector<vector<int>> adj;
vector<int> path;

void dfs(int u) {
    while (!adj[u].empty()) {
        int v = adj[u].back();
        adj[u].pop_back();
        dfs(v);
    }
    path.push_back(u);
}
```

Run from the start vertex, then reverse path.

### Tiny Code (Python)

```

def eulerian_path(graph, start):
    stack, path = [start], []
    while stack:
        u = stack[-1]
        if graph[u]:
            v = graph[u].pop()
            stack.append(v)
        else:
            path.append(stack.pop())
    return path[::-1]

```

## Why It Matters

- Foundational for graph traversal problems
- Used in:
  - DNA sequencing (De Bruijn graph reconstruction)
  - Route planning (postal delivery, garbage collection)
  - Network diagnostics (tracing all connections)

## A Gentle Proof (Why It Works)

Each edge must appear exactly once. At every vertex (except possibly start/end), every entry must be matched with an exit. This requires balanced degrees.

In an Eulerian Circuit:

$$\text{in}(v) = \text{out}(v) \quad \forall v$$

In an Eulerian Path:

$$\exists \text{start with } \text{out}(v) = \text{in}(v) + 1 \quad \exists \text{end with } \text{in}(v) = \text{out}(v) + 1$$

Hierholzer's algorithm constructs the path by merging cycles and ensuring all edges are consumed.

## Try It Yourself

1. Build a small graph and test parity conditions.
2. Implement Hierholzer's algorithm and trace each step.
3. Verify correctness by counting traversed edges.
4. Explore both directed and undirected variants.
5. Modify to detect Eulerian circuit vs path.

## Complexity

- Time:  $O(V + E)$
- Space:  $O(V + E)$

Eulerian paths are elegant because they cover every connection exactly once, perfect order in perfect traversal.

## 394 Hamiltonian Path

A Hamiltonian Path is a path in a graph that visits every vertex exactly once. If it starts and ends at the same vertex, it forms a Hamiltonian Cycle. Unlike the Eulerian Path (which focuses on edges), the Hamiltonian Path focuses on vertices.

Finding one is a classic NP-complete problem, there's no known polynomial-time algorithm for general graphs.

### What Problem Are We Solving?

We want to determine if there exists a path that visits every vertex exactly once, no repetition, no omission.

In formal terms: Given a graph  $G = (V, E)$ , find a sequence of vertices

$$v_1, v_2, \dots, v_n$$

such that  $(v_i, v_{i+1}) \in E$  for all  $i$ , and all vertices are distinct.

For a Hamiltonian Cycle, additionally  $(v_n, v_1) \in E$ .

## How Does It Work (Plain Language)

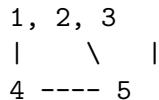
There's no simple parity or degree condition like in Eulerian paths. We usually solve it by backtracking, bitmask DP, or heuristics (for large graphs):

1. Pick a start vertex.
2. Recursively explore all unvisited neighbors.
3. Mark visited vertices.
4. If all vertices are visited → found a Hamiltonian Path.
5. Otherwise, backtrack.

For small graphs, this brute force works; for large graphs, it's impractical.

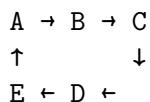
## Example (Undirected Graph)

Graph:



One Hamiltonian Path:  $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 4$  One Hamiltonian Cycle:  $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 4 \rightarrow 1$

## Example (Directed Graph)



Possible Hamiltonian Path:  $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E$

## Tiny Code (Backtracking – C++)

```

bool hamiltonianPath(int u, vector<vector<int>>& adj, vector<bool>& visited, vector<int>& path) {
    if (path.size() == n) return true;
    for (int v : adj[u]) {
        if (!visited[v]) {
            visited[v] = true;
            path.push_back(v);
            if (hamiltonianPath(v, adj, visited, path, n)) return true;
            visited[v] = false;
            path.pop_back();
        }
    }
    return false;
}

```

Call with each vertex as a potential start.

### Tiny Code (DP Bitmask – C++)

Used for TSP-style Hamiltonian search:

```

int n;
vector<vector<int>> dp(1 << n, vector<int>(n, INF));
dp[1][0] = 0;
for (int mask = 1; mask < (1 << n); ++mask) {
    for (int u = 0; u < n; ++u) if (mask & (1 << u)) {
        for (int v = 0; v < n; ++v) if (!(mask & (1 << v)) && adj[u][v]) {
            dp[mask | (1 << v)][v] = min(dp[mask | (1 << v)][v], dp[mask][u] + cost[u][v]);
        }
    }
}

```

### Why It Matters

- Models ordering problems:
  - Traveling Salesman (TSP)
  - Job sequencing with constraints
  - Genome assembly paths
- Fundamental in theoretical computer science, cornerstone NP-complete problem.
- Helps distinguish easy (Eulerian) vs hard (Hamiltonian) traversal.

## A Gentle Proof (Why It's Hard)

Hamiltonian Path existence is NP-complete:

1. Verification is easy (given a path, check in  $O(V)$ ).
2. No known polynomial algorithm (unless  $P = NP$ ).
3. Many problems reduce to it (like TSP).

This means it likely requires exponential time in general:

$$O(n!)$$

in naive form, or

$$O(2^n n)$$

using bitmask DP.

## Try It Yourself

1. Build small graphs (4–6 vertices) and trace paths.
2. Compare with Eulerian path conditions.
3. Implement backtracking search.
4. Extend to cycle detection (check edge back to start).
5. Try bitmask DP for small  $n \leq 20$ .

## Complexity

- Time (backtracking):  $O(n!)$
- Time (DP bitmask):  $O(2^n \cdot n^2)$
- Space:  $O(2^n \cdot n)$

Hamiltonian paths capture the essence of combinatorial explosion, simple to state, hard to solve, yet central to understanding computational limits.

## 395 Chinese Postman Problem (Route Inspection)

The Chinese Postman Problem (CPP), also known as the Route Inspection Problem, asks for the shortest closed path that traverses every edge of a graph at least once. It generalizes the Eulerian Circuit, allowing edge repetitions when necessary.

## What Problem Are We Solving?

Given a weighted graph  $G = (V, E)$ , we want to find a minimum-cost tour that covers all edges at least once and returns to the start.

If  $G$  is Eulerian (all vertices have even degree), the answer is simple, the Eulerian Circuit itself. Otherwise, we must duplicate edges strategically to make the graph Eulerian, minimizing total added cost.

## How It Works (Plain Language)

1. Check vertex degrees:
  - Count how many have odd degree.
2. If all even  $\rightarrow$  just find Eulerian Circuit.
3. If some odd  $\rightarrow$ 
  - Pair up odd vertices in such a way that the sum of shortest path distances between paired vertices is minimal.
  - Duplicate edges along those shortest paths.
4. The resulting graph is Eulerian, so an Eulerian Circuit can be constructed.
5. The cost of this circuit is the sum of all edges + added edges.

## Example

Graph edges (with weights):

| Edge | Weight |
|------|--------|
| 1–2  | 3      |
| 2–3  | 2      |
| 3–4  | 4      |
| 4–1  | 3      |
| 2–4  | 1      |

Degrees:

- $\deg(1)=2$ ,  $\deg(2)=3$ ,  $\deg(3)=2$ ,  $\deg(4)=3 \rightarrow$  odd vertices: 2, 4 Shortest path between 2–4: 1 Add that again  $\rightarrow$  now all even

$$\text{Total cost} = (3 + 2 + 4 + 3 + 1) + 1 = 14$$

## Algorithm (Steps)

1. Identify odd-degree vertices
2. Compute shortest path matrix (Floyd–Warshall)
3. Solve minimum-weight perfect matching among odd vertices
4. Duplicate edges in the matching
5. Perform Eulerian Circuit traversal (Hierholzer’s algorithm)

## Tiny Code (Pseudocode)

```
def chinese_postman(G):
    odd = [v for v in G if degree(v) % 2 == 1]
    if not odd:
        return eulerian_circuit(G)

    dist = floyd_marshall(G)
    pairs = minimum_weight_matching(odd, dist)
    for (u, v) in pairs:
        add_path(G, u, v, dist)
    return eulerian_circuit(G)
```

## Why It Matters

- Core algorithm in network optimization
- Used in:
  - Postal route planning
  - Garbage collection routing
  - Snow plow scheduling
  - Street sweeping
- Demonstrates how graph augmentation can solve traversal problems efficiently

## A Gentle Proof (Why It Works)

To traverse every edge, all vertices must have even degree (Eulerian condition). When vertices with odd degree exist, we must pair them up to restore evenness.

The minimal duplication set is a minimum-weight perfect matching among odd vertices:

$$\text{Extra cost} = \min_{\text{pairing } M} \sum_{(u,v) \in M} \text{dist}(u, v)$$

Thus, the optimal path cost:

$$C = \sum_{e \in E} w(e) + \text{Extra cost}$$

### Try It Yourself

1. Draw a graph with odd-degree vertices.
2. Identify odd vertices and shortest pair distances.
3. Compute minimal matching manually.
4. Add duplicated edges and find Eulerian Circuit.
5. Compare total cost before and after duplication.

### Complexity

- Floyd–Warshall:  $O(V^3)$
- Minimum Matching:  $O(V^3)$
- Eulerian traversal:  $O(E)$
- Total:  $O(V^3)$

The Chinese Postman Problem transforms messy graphs into elegant tours, balancing degrees, minimizing effort, and ensuring every edge gets its due.

## 396 Hierholzer's Algorithm

Hierholzer's Algorithm is the classical method for finding an Eulerian Path or Eulerian Circuit in a graph. It constructs the path by merging cycles until all edges are used exactly once.

### What Problem Are We Solving?

We want to find an Eulerian trail, a path or circuit that visits every edge exactly once.

- For an Eulerian Circuit (closed trail): All vertices have even degree.
- For an Eulerian Path (open trail): Exactly two vertices have odd degree.

The algorithm efficiently constructs the path in linear time relative to the number of edges.

## How It Works (Plain Language)

1. Check Eulerian conditions:
  - 0 odd-degree vertices → Eulerian Circuit
  - 2 odd-degree vertices → Eulerian Path (start at one odd)
2. Start from a valid vertex (odd if path, any if circuit)
3. Traverse edges greedily:
  - Follow edges until you return to the start or cannot continue.
4. Backtrack and merge:
  - When stuck, backtrack to a vertex with unused edges, start a new cycle, and merge it into the current path.
5. Continue until all edges are used.

The final sequence of vertices is the Eulerian trail.

## Example (Undirected Graph)

Graph:

```
1, 2  
|   |  
4, 3
```

All vertices have even degree, so an Eulerian Circuit exists.

Start at 1:

$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 1$

Result: Eulerian Circuit = [1, 2, 3, 4, 1]

## Example (With Odd Vertices)

Graph:

1, 2, 3

$\deg(1)=1, \deg(2)=2, \deg(3)=1 \rightarrow$  Eulerian Path exists Start at 1:

$1 \rightarrow 2 \rightarrow 3$

## Tiny Code (C++)

```
vector<vector<int>> adj;
vector<int> path;

void dfs(int u) {
    while (!adj[u].empty()) {
        int v = adj[u].back();
        adj[u].pop_back();
        dfs(v);
    }
    path.push_back(u);
}
```

After DFS, path will contain vertices in reverse order. Reverse it to get the Eulerian path or circuit.

## Tiny Code (Python)

```
def hierholzer(graph, start):
    stack, path = [start], []
    while stack:
        u = stack[-1]
        if graph[u]:
            v = graph[u].pop()
            graph[v].remove(u) # remove both directions for undirected
            stack.append(v)
        else:
            path.append(stack.pop())
    return path[::-1]
```

## Why It Matters

- Efficiently constructs Eulerian Paths in  $O(V + E)$
- Backbone for:
  - Chinese Postman Problem
  - Eulerian Circuit detection
  - DNA sequencing (De Bruijn graphs)
  - Route design and network analysis

## A Gentle Proof (Why It Works)

- Every time you traverse an edge, it's removed (used once).
- Each vertex retains balanced degree (entries = exits).
- When you get stuck, the subpath formed is a cycle.
- Merging all such cycles yields a single complete traversal.

Thus, every edge appears exactly once in the final route.

## Try It Yourself

1. Draw small Eulerian graphs.
2. Manually trace Hierholzer's algorithm.
3. Identify start vertex (odd-degree or any).
4. Verify path covers all edges exactly once.
5. Apply to both directed and undirected graphs.

## Complexity

- Time:  $O(V + E)$
- Space:  $O(V + E)$

Hierholzer's Algorithm elegantly builds order from connectivity, ensuring each edge finds its place in a perfect traversal.

## 397 Johnson's Cycle Finding Algorithm

Johnson's Algorithm is a powerful method for enumerating all simple cycles (elementary circuits) in a directed graph. A *simple cycle* is one that visits no vertex more than once, except the starting/ending vertex.

Unlike DFS-based approaches that can miss or duplicate cycles, Johnson's method systematically lists each cycle exactly once, running in  $O((V + E)(C + 1))$ , where  $C$  is the number of cycles.

### What Problem Are We Solving?

We want to find all simple cycles in a directed graph  $G = (V, E)$ .

That is, find all vertex sequences

$$v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k \rightarrow v_1$$

where each  $v_i$  is distinct and edges exist between consecutive vertices.

Enumerating all cycles is fundamental for:

- Dependency analysis
- Feedback detection
- Circuit design
- Graph motif analysis

### How It Works (Plain Language)

Johnson's algorithm is based on backtracking with smart pruning and SCC decomposition:

1. Process vertices in order
  - Consider vertices  $1, 2, \dots, n$ .
2. For each vertex  $s$ :
  - Consider the subgraph induced by vertices  $s$ .
  - Find strongly connected components (SCCs) in this subgraph.
  - If  $s$  belongs to a nontrivial SCC, explore all simple cycles starting at  $s$ .
3. Use a blocked set to avoid redundant exploration:
  - Once a vertex leads to a dead-end, mark it blocked.
  - Unblock when a valid cycle is found through it.

This avoids exploring the same path multiple times.

## Example

Graph:

```
A → B → C  
↑   ↓   |  
D ←
```

Cycles:

1. A → B → C → D → A
2. B → C → D → B

Johnson's algorithm will find both efficiently, without duplication.

## Pseudocode

```
def johnson(G):  
    result = []  
    blocked = set()  
    B = {v: set() for v in G}  
    stack = []  
  
    def circuit(v, s):  
        f = False  
        stack.append(v)  
        blocked.add(v)  
        for w in G[v]:  
            if w == s:  
                result.append(stack.copy())  
                f = True  
            elif w not in blocked:  
                if circuit(w, s):  
                    f = True  
        if f:  
            unblock(v)  
        else:  
            for w in G[v]:  
                B[w].add(v)  
            stack.pop()  
    return f
```

```

def unblock(u):
    blocked.discard(u)
    for w in B[u]:
        if w in blocked:
            unblock(w)
    B[u].clear()

for s in sorted(G.keys()):
    # consider subgraph of nodes >= s
    subG = {v: [w for w in G[v] if w >= s] for v in G if v >= s}
    SCCs = strongly_connected_components(subG)
    if not SCCs:
        continue
    scc = min(SCCs, key=lambda S: min(S))
    s_node = min(scc)
    circuit(s_node, s_node)
return result

```

## Why It Matters

- Enumerates all simple cycles without duplicates
- Works for directed graphs (unlike many undirected-only algorithms)
- Key in:
  - Deadlock detection
  - Cycle basis computation
  - Feedback arc set analysis
  - Subgraph pattern mining

## A Gentle Proof (Why It Works)

Each recursive search begins from the smallest vertex in an SCC. By restricting search to vertices  $\geq s$ , every cycle is discovered exactly once (at its least-numbered vertex). The blocked set prevents repeated exploration of dead ends. Unblocking ensures vertices re-enter search space when part of a valid cycle.

This guarantees:

- No cycle is missed
- No cycle is duplicated

## Try It Yourself

1. Draw a small directed graph with 3–5 vertices.
2. Identify SCCs manually.
3. Apply the algorithm step-by-step, noting `blocked` updates.
4. Record each cycle when returning to start.
5. Compare with naive DFS enumeration.

## Complexity

- Time:  $O((V + E)(C + 1))$
- Space:  $O(V + E)$

Johnson's algorithm reveals the hidden loops inside directed graphs, one by one, exhaustively and elegantly.

## 398 Transitive Closure (Floyd–Warshall)

The Transitive Closure of a directed graph captures reachability: it tells us, for every pair of vertices  $(u, v)$ , whether there exists a path from  $u$  to  $v$ .

It's often represented as a boolean matrix  $R$ , where

$$R[u][v] = 1 \text{ if and only if there is a path from } u \text{ to } v$$

This can be computed efficiently using a modified version of the Floyd–Warshall algorithm.

## What Problem Are We Solving?

Given a directed graph  $G = (V, E)$ , we want to determine for every pair  $(u, v)$  whether:

$$u \rightsquigarrow v$$

That is, can we reach  $v$  from  $u$  through a sequence of directed edges?

The output is a reachability matrix, useful in:

- Dependency analysis
- Access control and authorization graphs
- Program call graphs
- Database query optimization

## How It Works (Plain Language)

We apply the Floyd–Warshall dynamic programming idea, but instead of distances, we propagate reachability.

Let  $R[u][v] = 1$  if  $u \rightarrow v$  (direct edge), otherwise 0. Then for each vertex  $k$  (intermediate node), we update:

$$R[u][v] = R[u][v] \vee (R[u][k] \wedge R[k][v])$$

Intuitively: “ $u$  can reach  $v$  if  $u$  can reach  $k$  and  $k$  can reach  $v$ .”

### Example

Graph:

A → B → C  
↑              |

Initial reachability (direct edges):

| A | B | C |
|---|---|---|
| A | 0 | 1 |
| B | 0 | 0 |
| C | 1 | 0 |

After applying transitive closure:

| A | B | C |
|---|---|---|
| A | 1 | 1 |
| B | 1 | 1 |
| C | 1 | 1 |

Every vertex is reachable from every other, the graph is strongly connected.

## Tiny Code (C)

```

#define N 100
int R[N][N];

void floyd_marshall_tc(int n) {
    for (int k = 0; k < n; ++k)
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                R[i][j] = R[i][j] || (R[i][k] && R[k][j]);
}

```

### Tiny Code (Python)

```

def transitive_closure(R):
    n = len(R)
    for k in range(n):
        for i in range(n):
            for j in range(n):
                R[i][j] = R[i][j] or (R[i][k] and R[k][j])
    return R

```

### Why It Matters

- Converts a graph into a reachability matrix
- Enables constant-time queries: “Can  $u$  reach  $v$ ?”
- Used in:
  - Compilers (call dependencies)
  - Databases (recursive queries)
  - Security graphs
  - Network analysis

### A Gentle Proof (Why It Works)

We extend reachability step by step:

- Base:  $R^{(0)}$  = direct edges
- Step:  $R^{(k)}$  = paths using vertices  $1, 2, \dots, k$  as intermediates

By induction:

$$R^{(k)}[i][j] = 1 \iff \text{there exists a path } i \rightarrow j \text{ using vertices } \leq k$$

At the end ( $k = n$ ),  $R^{(n)}$  contains all possible paths.

### Try It Yourself

1. Create a directed graph with 4–5 nodes.
2. Build its adjacency matrix.
3. Apply the algorithm by hand.
4. Observe how new reachabilities emerge after each  $k$ .
5. Compare with paths you can see visually.

### Complexity

- Time:  $O(V^3)$
- Space:  $O(V^2)$

Transitive closure transforms reachability into certainty, mapping every potential path into a single clear view of what connects to what.

## 399 Graph Coloring (Backtracking)

Graph Coloring is the problem of assigning colors to vertices of a graph so that no two adjacent vertices share the same color. The smallest number of colors required is called the chromatic number of the graph.

This classic constraint satisfaction problem lies at the heart of scheduling, register allocation, and pattern assignment.

### What Problem Are We Solving?

Given a graph  $G = (V, E)$  and an integer  $k$ , determine whether it is possible to color all vertices using at most  $k$  colors such that:

$$\forall (u, v) \in E, \text{color}(u) \neq \text{color}(v)$$

If such a coloring exists,  $G$  is  $k$ -colorable.

## How It Works (Plain Language)

We solve this using backtracking:

1. Assign a color to the first vertex.
2. Move to the next vertex.
3. Try all available colors (from 1 to  $k$ ).
4. If a color assignment violates adjacency constraints, skip it.
5. If a vertex cannot be colored, backtrack to previous vertex and change its color.
6. Continue until:
  - all vertices are colored (success), or
  - no valid assignment exists (failure).

## Example

Graph:

```
1, 2  
| |  
3, 4
```

A square requires at least 2 colors:

- $\text{color}(1) = 1$
- $\text{color}(2) = 2$
- $\text{color}(3) = 2$
- $\text{color}(4) = 1$

Valid 2-coloring.

Try 1 color  $\rightarrow$  fails (adjacent nodes same color) Try 2 colors  $\rightarrow$  success  $\rightarrow$  chromatic number = 2

## Tiny Code (C++)

```

int n, k;
vector<vector<int>> adj;
vector<int> color;

bool isSafe(int v, int c) {
    for (int u : adj[v])
        if (color[u] == c) return false;
    return true;
}

bool solve(int v) {
    if (v == n) return true;
    for (int c = 1; c <= k; ++c) {
        if (isSafe(v, c)) {
            color[v] = c;
            if (solve(v + 1)) return true;
            color[v] = 0;
        }
    }
    return false;
}

```

### Tiny Code (Python)

```

def graph_coloring(graph, k):
    n = len(graph)
    color = [0] * n

    def safe(v, c):
        return all(color[u] != c for u in range(n) if graph[v][u])

    def backtrack(v):
        if v == n:
            return True
        for c in range(1, k + 1):
            if safe(v, c):
                color[v] = c
                if backtrack(v + 1):
                    return True
                color[v] = 0

```

```

    return False

return backtrack(0)

```

## Why It Matters

Graph coloring captures the essence of constraint-based allocation:

- Scheduling: assign time slots to tasks
- Register allocation: map variables to CPU registers
- Map coloring: color regions with shared boundaries
- Frequency assignment: allocate channels in wireless networks

## A Gentle Proof (Why It Works)

We explore all possible assignments (depth-first search) under the rule:

$$\forall(u, v) \in E, ; \text{color}(u) \neq \text{color}(v)$$

Backtracking prunes partial solutions that cannot lead to valid full assignments. When a full coloring is found, constraints are satisfied by construction.

By completeness of backtracking, if a valid  $k$ -coloring exists, it will be found.

## Try It Yourself

1. Draw small graphs (triangle, square, pentagon).
2. Attempt coloring with  $k = 2, 3, 4$ .
3. Observe where conflicts force backtracking.
4. Try greedy coloring and compare with backtracking.
5. Identify the chromatic number experimentally.

## Complexity

- Time:  $O(k^V)$  (exponential worst case)
- Space:  $O(V)$

Graph coloring blends search and logic, a careful dance through the constraints, discovering harmony one color at a time.

## 400 Articulation Points & Bridges

Articulation Points and Bridges identify weak spots in a graph, nodes or edges whose removal increases the number of connected components. They are essential in analyzing network resilience, communication reliability, and biconnected components.

### What Problem Are We Solving?

Given an undirected graph  $G = (V, E)$ , find:

- Articulation Points (Cut Vertices): Vertices whose removal disconnects the graph.
- Bridges (Cut Edges): Edges whose removal disconnects the graph.

We want efficient algorithms to detect these in  $O(V + E)$  time.

### How It Works (Plain Language)

We use a single DFS traversal (Tarjan's algorithm) with two key arrays:

- `disc[v]`: discovery time of vertex  $v$
- `low[v]`: the lowest discovery time reachable from  $v$  (including back edges)

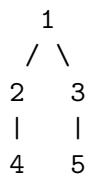
During DFS:

- A vertex  $u$  is an articulation point if:
  - $u$  is root and has more than one child, or
  - $\exists$  child  $v$  such that  $\text{low}[v] \geq \text{disc}[u]$
- An edge  $(u, v)$  is a bridge if:
  - $\text{low}[v] > \text{disc}[u]$

These conditions detect when no back-edge connects a subtree back to an ancestor.

## Example

Graph:



Remove node 2 → node 4 becomes isolated → 2 is an articulation point. Remove edge (2, 4) → increases components → (2, 4) is a bridge.

## Tiny Code (C++)

```
vector<vector<int>> adj;
vector<int> disc, low, parent;
vector<bool> ap;
int timer = 0;

void dfs(int u) {
    disc[u] = low[u] = ++timer;
    int children = 0;

    for (int v : adj[u]) {
        if (!disc[v]) {
            parent[v] = u;
            ++children;
            dfs(v);
            low[u] = min(low[u], low[v]);

            if (parent[u] == -1 && children > 1)
                ap[u] = true;
            if (parent[u] != -1 && low[v] >= disc[u])
                ap[u] = true;
            if (low[v] > disc[u])
                cout << "Bridge: " << u << " - " << v << "\n";
        } else if (v != parent[u]) {
            low[u] = min(low[u], disc[v]);
        }
    }
}
```

```
    }
}
```

### Tiny Code (Python)

```
def articulation_points_and_bridges(graph):
    n = len(graph)
    disc = [0] * n
    low = [0] * n
    parent = [-1] * n
    ap = [False] * n
    bridges = []
    time = 1

    def dfs(u):
        nonlocal time
        disc[u] = low[u] = time
        time += 1
        children = 0
        for v in graph[u]:
            if not disc[v]:
                parent[v] = u
                children += 1
                dfs(v)
                low[u] = min(low[u], low[v])

                if parent[u] == -1 and children > 1:
                    ap[u] = True
                if parent[u] != -1 and low[v] >= disc[u]:
                    ap[u] = True
                if low[v] > disc[u]:
                    bridges.append((u, v))
        elif v != parent[u]:
            low[u] = min(low[u], disc[v])

    for i in range(n):
        if not disc[i]:
            dfs(i)

    return [i for i, x in enumerate(ap) if x], bridges
```

## Why It Matters

Articulation points and bridges reveal critical nodes and links in:

- Network design: identify weak links in infrastructure
- Social networks: find influencers whose removal splits communities
- Compiler dependency graphs: locate critical connections
- Transport systems: ensure robust routing

Understanding where a graph breaks helps us design systems that don't.

## A Gentle Proof (Why It Works)

For each node  $u$ ,  $\text{low}[u]$  captures the earliest discovered vertex reachable via DFS or a back edge. If a child subtree cannot reach an ancestor of  $u$ , then  $u$  is a bottleneck, removing it splits the graph.

The inequalities:

$$\text{low}[v] \geq \text{disc}[u] \implies u \text{ is articulation point}$$

$$\text{low}[v] > \text{disc}[u] \implies (u, v) \text{ is bridge}$$

are derived from whether a subtree is connected back to an ancestor.

## Try It Yourself

1. Draw small graphs (triangle, line, star).
2. Manually run DFS and record `disc` and `low`.
3. Identify articulation points and bridges.
4. Try adding edges, see how redundancy removes articulation points.

## Complexity

- Time:  $O(V + E)$
- Space:  $O(V)$

Finding articulation points and bridges transforms structure into insight, helping you build networks that stay connected even when parts fail.

# **Chapter 5. Dynamic Programming**

# Section 41. DP basic and state transitions

## 401 Fibonacci DP

Fibonacci is the hello-world of dynamic programming, a simple sequence that teaches the power of remembering past results. Instead of recomputing subproblems over and over, we store them. The result? A huge leap from exponential to linear time.

### What Problem Are We Solving?

The Fibonacci sequence is defined as:

$$F(0) = 0, \quad F(1) = 1, \quad F(n) = F(n - 1) + F(n - 2)$$

A naive recursive version recomputes the same values many times. For example,  $F(5)$  calls  $F(4)$  and  $F(3)$ , but  $F(4)$  also calls  $F(3)$  again, wasteful repetition.

Our goal is to avoid recomputation by caching results. That's dynamic programming in a nutshell.

### How Does It Work (Plain Language)?

Think of Fibonacci as a ladder. You can climb to step  $n$  only if you know the number of ways to reach  $n-1$  and  $n-2$ . Instead of recalculating those steps every time, record them once, then reuse.

There are two main flavors of DP:

| Approach               | Description               | Example                   |
|------------------------|---------------------------|---------------------------|
| Top-Down (Memoization) | Recursion + caching       | Store results in an array |
| Bottom-Up (Tabulation) | Iteration from base cases | Build array from 0 up     |

Let's visualize the state filling:

| n | F(n-2) | F(n-1) | F(n) = F(n-1) + F(n-2) |
|---|--------|--------|------------------------|
| 0 | -      | -      | 0                      |
| 1 | -      | -      | 1                      |
| 2 | 0      | 1      | 1                      |
| 3 | 1      | 1      | 2                      |
| 4 | 1      | 2      | 3                      |
| 5 | 2      | 3      | 5                      |
| 6 | 3      | 5      | 8                      |

Each new value reuses two old ones, no redundant work.

### Tiny Code (Easy Versions)

C (Bottom-Up Fibonacci)

```
#include <stdio.h>

int main(void) {
    int n;
    printf("Enter n: ");
    scanf("%d", &n);

    if (n == 0) { printf("0\n"); return 0; }
    if (n == 1) { printf("1\n"); return 0; }

    long long dp[n + 1];
    dp[0] = 0;
    dp[1] = 1;

    for (int i = 2; i <= n; i++) {
        dp[i] = dp[i - 1] + dp[i - 2];
    }

    printf("Fibonacci(%d) = %lld\n", n, dp[n]);
    return 0;
}
```

Python (Memoized Fibonacci)

```

from functools import lru_cache

@lru_cache(maxsize=None)
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)

n = int(input("Enter n: "))
print("Fibonacci(, n, ) =", fib(n))

```

## Why It Matters

- Demonstrates state definition:  $F(n) = F(n-1) + F(n-2)$
- Introduces overlapping subproblems and optimal substructure
- First step toward mastering DP intuition
- Reduces time complexity from exponential  $O(2^n)$  to linear  $O(n)$

You learn that solving once and remembering is better than solving a hundred times.

## Step-by-Step Example

| Step   | Calculation      | Memo Table (Partial) |
|--------|------------------|----------------------|
| Base   | $F(0)=0, F(1)=1$ | [0, 1, , , , ]       |
| $F(2)$ | $F(1)+F(0)=1$    | [0, 1, 1, , , ]      |
| $F(3)$ | $F(2)+F(1)=2$    | [0, 1, 1, 2, , ]     |
| $F(4)$ | $F(3)+F(2)=3$    | [0, 1, 1, 2, 3, ]    |
| $F(5)$ | $F(4)+F(3)=5$    | [0, 1, 1, 2, 3, 5]   |

## Try It Yourself

1. Write Fibonacci recursively without memoization. Measure calls.
2. Add a dictionary or array for memoization, compare speeds.
3. Convert recursive to iterative (tabulation).
4. Optimize space: store only two variables instead of full array.
5. Print the full table of computed values.

## Test Cases

| n  | Expected Output | Notes                 |
|----|-----------------|-----------------------|
| 0  | 0               | Base case             |
| 1  | 1               | Base case             |
| 2  | 1               | $1 + 0$               |
| 5  | 5               | Sequence: 0,1,1,2,3,5 |
| 10 | 55              | Smooth growth check   |

## Complexity

- Time:  $O(n)$  for both memoization and tabulation
- Space:  $O(n)$  for table;  $O(1)$  if optimized

Fibonacci DP is the simplest proof that remembering pays off, it's where dynamic programming begins, and efficiency is born.

## 402 Climbing Stairs

The climbing stairs problem is a friendly cousin of Fibonacci, same recurrence, different story. You're standing at the bottom of a staircase with  $n$  steps. You can climb 1 step or 2 steps at a time. How many distinct ways can you reach the top?

This is one of the most intuitive gateways into dynamic programming: define states, relate them recursively, and reuse past computations.

### What Problem Are We Solving?

We want the number of distinct ways to reach step  $n$ .

You can reach step  $n$  by:

- taking 1 step from  $n-1$ , or
- taking 2 steps from  $n-2$ .

So the recurrence is:

$$dp[n] = dp[n - 1] + dp[n - 2]$$

with base cases:

$$dp[0] = 1, \quad dp[1] = 1$$

This is structurally identical to Fibonacci, but with a combinatorial interpretation.

## How Does It Work (Plain Language)?

Think of each step as a checkpoint. To reach step  $n$ , you must come from either of the two prior checkpoints. If you already know how many ways there are to reach those, just add them.

Let's illustrate with a table:

| Step (n) | Ways to Reach | Explanation                                 |
|----------|---------------|---------------------------------------------|
| 0        | 1             | Stay at ground                              |
| 1        | 1             | Single step                                 |
| 2        | 2             | (1+1), (2)                                  |
| 3        | 3             | (1+1+1), (1+2), (2+1)                       |
| 4        | 5             | (1+1+1+1), (2+1+1), (1+2+1), (1+1+2), (2+2) |
| 5        | 8             | Previous two sums: $5 = 3 + 2$              |

Each new value is the sum of the previous two.

## Tiny Code (Easy Versions)

C (Bottom-Up Climbing Stairs)

```
#include <stdio.h>

int main(void) {
    int n;
    printf("Enter number of steps: ");
    scanf("%d", &n);

    if (n == 0 || n == 1) {
        printf("Ways: 1\n");
        return 0;
    }

    long long dp[n + 1];
    dp[0] = 1;
    dp[1] = 1;

    for (int i = 2; i <= n; i++) {
        dp[i] = dp[i - 1] + dp[i - 2];
    }
}
```

```

    printf("Ways to climb %d steps: %lld\n", n, dp[n]);
    return 0;
}

```

Python (Space Optimized)

```

n = int(input("Enter number of steps: "))

if n == 0 or n == 1:
    print(1)
else:
    a, b = 1, 1
    for _ in range(2, n + 1):
        a, b = b, a + b
    print("Ways to climb", n, "steps:", b)

```

## Why It Matters

- Demonstrates state transitions:  $dp[i]$  depends on  $dp[i-1]$  and  $dp[i-2]$
- Teaches bottom-up thinking and base case setup
- Shows how recurrence translates to counting problems
- Connects combinatorics with DP intuition

Climbing stairs is a great mental bridge between pure math (recurrence) and applied reasoning (counting paths).

## Step-by-Step Example

Let's trace  $n = 5$ :

| i | $dp[i-2]$ | $dp[i-1]$ | $dp[i] = dp[i-1] + dp[i-2]$ |
|---|-----------|-----------|-----------------------------|
| 0 | -         | -         | 1                           |
| 1 | -         | -         | 1                           |
| 2 | 1         | 1         | 2                           |
| 3 | 1         | 2         | 3                           |
| 4 | 2         | 3         | 5                           |
| 5 | 3         | 5         | 8                           |

So, 8 ways to climb 5 steps.

## Try It Yourself

1. Modify the code to allow 1, 2, or 3 steps at a time.
2. Print the entire `dp` table for small `n`.
3. Compare recursive vs iterative solutions.
4. Try to derive a formula, notice the Fibonacci pattern.
5. What if each step had a cost? Adapt it to Min Cost Climb.

## Test Cases

| n | Expected Output | Ways                  |
|---|-----------------|-----------------------|
| 0 | 1               | Do nothing            |
| 1 | 1               | [1]                   |
| 2 | 2               | [1+1], [2]            |
| 3 | 3               | [1+1+1], [1+2], [2+1] |
| 4 | 5               | All combinations      |
| 5 | 8               | Grows like Fibonacci  |

## Complexity

- Time:  $O(n)$
- Space:  $O(n)$ , reducible to  $O(1)$  with two variables

Climbing stairs shows that dynamic programming isn't just math, it's about recognizing patterns in movement, growth, and memory.

## 403 Grid Paths

The grid path problem is a gentle step into 2D dynamic programming, where states depend on neighbors, not just previous elements. Imagine standing in the top-left corner of a grid, moving only right or down, trying to count how many ways lead to the bottom-right corner.

Each cell's value is determined by paths reaching it from above or from the left, a perfect metaphor for how DP builds solutions layer by layer.

## What Problem Are We Solving?

Given an  $m \times n$  grid, find the number of distinct paths from  $(0, 0)$  to  $(m-1, n-1)$  when you can move only:

- Right  $(x, y+1)$
- Down  $(x+1, y)$

The recurrence:

$$dp[i][j] = dp[i - 1][j] + dp[i][j - 1]$$

with base cases:

$$dp[0][j] = 1, \quad dp[i][0] = 1$$

(since only one way exists along the first row or column)

## How Does It Work (Plain Language)?

Think of the grid like a city map, every intersection  $(i, j)$  can be reached from either the north  $(i-1, j)$  or the west  $(i, j-1)$ . So total routes = routes from north + routes from west.

Let's visualize a  $3 \times 3$  grid (0-indexed):

| Cell (i,j) | Ways | Explanation         |
|------------|------|---------------------|
| (0,0)      | 1    | Start               |
| (0,1)      | 1    | Only from left      |
| (0,2)      | 1    | Only from left      |
| (1,0)      | 1    | Only from top       |
| (1,1)      | 2    | $(0,1) + (1,0) = 2$ |
| (1,2)      | 3    | $(0,2) + (1,1) = 3$ |
| (2,0)      | 1    | Only from top       |
| (2,1)      | 3    | $(1,1) + (2,0) = 3$ |
| (2,2)      | 6    | $(1,2) + (2,1) = 6$ |

So  $dp[2][2] = 6 \rightarrow 6$  distinct paths.

## Tiny Code (Easy Versions)

C (2D DP Table)

```

#include <stdio.h>

int main(void) {
    int m, n;
    printf("Enter rows and cols: ");
    scanf("%d %d", &m, &n);

    long long dp[m][n];

    for (int i = 0; i < m; i++) dp[i][0] = 1;
    for (int j = 0; j < n; j++) dp[0][j] = 1;

    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            dp[i][j] = dp[i-1][j] + dp[i][j-1];
        }
    }

    printf("Unique paths: %lld\n", dp[m-1][n-1]);
    return 0;
}

```

Python (Space Optimized)

```

m, n = map(int, input("Enter rows and cols: ").split())
dp = [1] * n

for _ in range(1, m):
    for j in range(1, n):
        dp[j] += dp[j - 1]

print("Unique paths:", dp[-1])

```

## Why It Matters

- Teaches 2D DP grids
- Builds intuition for problems on lattices, matrices, grids
- Foundation for min-cost path, maze traversal, robot movement
- Encourages space optimization from 2D → 1D

From counting paths to optimizing them, this grid is your DP canvas.

## Step-by-Step Example

For a 3×3 grid:

| ij | 0 | 1 | 2 |
|----|---|---|---|
| 0  | 1 | 1 | 1 |
| 1  | 1 | 2 | 3 |
| 2  | 1 | 3 | 6 |

$dp[2][2] = 6 \rightarrow$  six unique routes.

## Try It Yourself

1. Modify the code to handle obstacles (0 = block, 1 = open).
2. Print the DP table.
3. Implement using recursion + memoization.
4. Add a condition for moving right, down, and diagonal.
5. Compare with combinatorial formula:  $\binom{m+n-2}{m-1}$ .

## Test Cases

| Grid Size | Expected Paths | Notes                   |
|-----------|----------------|-------------------------|
| 1×1       | 1              | Only starting cell      |
| 2×2       | 2              | Right→Down, Down→Right  |
| 3×3       | 6              | Classic case            |
| 3×4       | 10             | Combinatorics check     |
| 4×4       | 20             | Pascal triangle pattern |

## Complexity

- Time:  $O(m \times n)$
- Space:  $O(m \times n)$ , reducible to  $O(n)$

Grid Paths reveal the essence of DP, every position depends on simpler ones. From here, you'll learn to minimize, maximize, and traverse with purpose.

## 404 Min Cost Path

The Min Cost Path problem is where counting meets optimization. Instead of asking “*How many ways can I reach the end?*”, we ask “*What’s the cheapest way to get there?*”. You’re moving across a grid, cell by cell, each with a cost, and your goal is to reach the bottom-right corner while minimizing the total cost.

This is one of the most fundamental path optimization problems in dynamic programming.

### What Problem Are We Solving?

Given a matrix  $\text{cost}[m][n]$ , where each cell represents a non-negative cost, find the minimum total cost path from  $(0, 0)$  to  $(m-1, n-1)$ , moving only right, down, or (optionally) diagonally down-right.

The recurrence:

$$dp[i][j] = \text{cost}[i][j] + \min(dp[i-1][j], dp[i][j-1])$$

If diagonal moves are allowed:

$$dp[i][j] = \text{cost}[i][j] + \min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1])$$

Base case:

$$dp[0][0] = \text{cost}[0][0]$$

### How Does It Work (Plain Language)?

Imagine hiking across a grid of terrain, where each cell has an energy cost. Every move you make adds to your total cost. You always want to choose the path that keeps your running total as small as possible.

The DP table records the minimum cost to reach each cell, building from the top-left to the bottom-right.

Let’s see an example grid:

| Cell  | Cost |
|-------|------|
| (0,0) | 1    |
| (0,1) | 3    |
| (0,2) | 5    |
| (1,0) | 2    |
| (1,1) | 1    |
| (1,2) | 2    |

| Cell  | Cost |
|-------|------|
| (2,0) | 4    |
| (2,1) | 3    |
| (2,2) | 1    |

We fill  $dp[i][j] = \text{cost}$  to reach  $(i, j)$ :

| ij | 0 | 1 | 2 |
|----|---|---|---|
| 0  | 1 | 4 | 9 |
| 1  | 3 | 2 | 4 |
| 2  | 7 | 5 | 5 |

Minimum cost = 5

### Tiny Code (Easy Versions)

C (Bottom-Up DP)

```
#include <stdio.h>
#define MIN(a,b) ((a)<(b)?(a):(b))

int main(void) {
    int m, n;
    printf("Enter rows and cols: ");
    scanf("%d %d", &m, &n);

    int cost[m][n];
    printf("Enter cost matrix:\n");
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
            scanf("%d", &cost[i][j]);

    int dp[m][n];
    dp[0][0] = cost[0][0];

    for (int i = 1; i < m; i++) dp[i][0] = dp[i-1][0] + cost[i][0];
    for (int j = 1; j < n; j++) dp[0][j] = dp[0][j-1] + cost[0][j];

    for (int i = 1; i < m; i++)
        for (int j = 1; j < n; j++)
            dp[i][j] = MIN(dp[i-1][j], dp[i][j-1]) + cost[i][j];
}
```

```

        for (int j = 1; j < n; j++)
            dp[i][j] = cost[i][j] + MIN(dp[i-1][j], dp[i][j-1]);

    printf("Min cost: %d\n", dp[m-1][n-1]);
    return 0;
}

```

Python (Optional Diagonal Move)

```

def min_cost_path(cost):
    m, n = len(cost), len(cost[0])
    dp = [[0]*n for _ in range(m)]
    dp[0][0] = cost[0][0]

    for i in range(1, m):
        dp[i][0] = dp[i-1][0] + cost[i][0]
    for j in range(1, n):
        dp[0][j] = dp[0][j-1] + cost[0][j]

    for i in range(1, m):
        for j in range(1, n):
            dp[i][j] = cost[i][j] + min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1])

    return dp[-1][-1]

cost = [
    [1, 3, 5],
    [2, 1, 2],
    [4, 3, 1]
]
$$

print("Min cost path:", min_cost_path(cost))

```

## Why It Matters

- Transitions from counting paths to optimizing paths
- Introduces minimization recurrence
- Basis for many grid problems (e.g. maze solving, image traversal, shortest paths)
- Builds intuition for weighted DP

This problem is a stepping stone toward Dijkstra's and Bellman-Ford in graphs.

## Step-by-Step Example

| Cell  | From Top | From Left | Min | Total Cost |
|-------|----------|-----------|-----|------------|
| (0,0) | -        | -         | -   | 1          |
| (0,1) | -        | 1+3       | 4   | 4          |
| (1,0) | 1+2      | -         | 3   | 3          |
| (1,1) | 4+1      | 3+1       | 4   | 4          |
| (2,2) | ...      | ...       | ... | 5          |

Answer: 5

## Try It Yourself

1. Add diagonal moves, compare results.
2. Add blocked cells (infinite cost).
3. Modify to find maximum cost path.
4. Reconstruct the path using a `parent` table.
5. Use a priority queue (Dijkstra) for non-grid graphs.

## Test Cases

| Grid                      | Expected Min Cost | Notes         |
|---------------------------|-------------------|---------------|
| [[1]]                     | 1                 | Single cell   |
| [[1,2],[3,4]]             | 7                 | 1→2→4         |
| [[1,3,5],[2,1,2],[4,3,1]] | 5                 | Optimal route |
| [[5,9],[4,2]]             | 11                | 5→4→2         |

## Complexity

- Time:  $O(m \times n)$
- Space:  $O(m \times n)$ , reducible to  $O(n)$

Min Cost Path turns the grid into a map of decisions, each cell asks, “What’s the cheapest way to reach me?” and the DP table answers with calm precision.

## 405 Coin Change (Count Ways)

You have coin denominations and an amount. How many distinct ways can you make that amount if you can use unlimited coins of each type? We count combinations where order does not matter. For example, with coins [1, 2, 5] there are 4 ways to make 5: 5, 2+2+1, 2+1+1+1, 1+1+1+1+1.

### What Problem Are We Solving?

Given an array `coins[]` and an integer `amount`, compute the number of combinations to form `amount` using unlimited copies of each coin.

State and recurrence:

$$dp[x] = \text{number of ways to make sum } x$$

$$dp[0] = 1$$

$$\text{for each coin } c : \quad \text{for } x \text{ from } c \text{ to amount} : \quad dp[x] += dp[x - c]$$

Why this order: iterating coins on the outside ensures each combination is counted once. If you loop amounts on the outside and coins inside, you would count permutations.

### How Does It Work (Plain Language)?

Think of building the total from left to right. For each coin value  $c$ , you ask: if I must use  $c$  at least once, how many ways remain to fill  $x - c$ ? Add those to the ways already known. Move forward increasing  $x$ , and repeat for the next coin. The table fills like a rolling tally.

Example with coins [1, 2, 5] and `amount = 5`:

| After<br>processing | dp[0] | dp[1] | dp[2] | dp[3] | dp[4] | dp[5] | Explanation                       |
|---------------------|-------|-------|-------|-------|-------|-------|-----------------------------------|
| Init                | 1     | 0     | 0     | 0     | 0     | 0     | One way to make 0: choose nothing |
| Coin 1              | 1     | 1     | 1     | 1     | 1     | 1     | Using 1s only                     |
| Coin 2              | 1     | 1     | 2     | 2     | 3     | 3     | Add ways that end with a 2        |
| Coin 5              | 1     | 1     | 2     | 2     | 3     | 4     | Add ways that end with a 5        |

Answer is  $dp[5] = 4$ .

## Tiny Code (Easy Versions)

C (1D DP, combinations)

```
#include <stdio.h>

int main(void) {
    int n, amount;
    printf("Enter number of coin types and amount: ");
    scanf("%d %d", &n, &amount);

    int coins[n];
    printf("Enter coin values: ");
    for (int i = 0; i < n; i++) scanf("%d", &coins[i]);

    // Use long long to avoid overflow for large counts
    long long dp[amount + 1];
    for (int x = 0; x <= amount; x++) dp[x] = 0;
    dp[0] = 1;

    for (int i = 0; i < n; i++) {
        int c = coins[i];
        for (int x = c; x <= amount; x++) {
            dp[x] += dp[x - c];
        }
    }

    printf("Number of ways: %lld\n", dp[amount]);
    return 0;
}
```

Python (1D DP, combinations)

```
coins = list(map(int, input("Enter coin values: ").split()))
amount = int(input("Enter amount: "))

dp = [0] * (amount + 1)
dp[0] = 1

for c in coins:
    for x in range(c, amount + 1):
        dp[x] += dp[x - c]
```

```
print("Number of ways:", dp[amount])
```

## Why It Matters

- Introduces the idea of unbounded knapsack counting
- Shows how loop ordering controls whether you count combinations or permutations
- Forms a foundation for many counting DPs such as integer partitions and dice sum counts
- Encourages space optimization with a single dimension

## Step-by-Step Example

Coin [1, 3, 4], amount 6:

| Step   | Update             | dp array snapshot (index is amount) |
|--------|--------------------|-------------------------------------|
| Init   | dp[0]=1            | [1, 0, 0, 0, 0, 0]                  |
| Coin 1 | fill x=1..6        | [1, 1, 1, 1, 1, 1]                  |
| Coin 3 | x=3..6 add dp[x-3] | [1, 1, 1, 2, 2, 3]                  |
| Coin 4 | x=4..6 add dp[x-4] | [1, 1, 1, 2, 3, 4]                  |

Answer: dp[6] = 4.

## Try It Yourself

1. Switch to permutations counting: loop `x` outside and `coins` inside. Compare results.
2. Add a cap per coin type and convert to a bounded version.
3. Sort coins and print one valid combination using a parent pointer array.
4. Use modulo arithmetic to avoid overflow: for example  $10^9+7$ .
5. Extend to count ways for every `x` from 0 to `amount` and print the full table.

## Test Cases

| Coins   | Amount | Expected Ways | Notes                             |
|---------|--------|---------------|-----------------------------------|
| []      | 0      | 1             | One empty way                     |
| []      | 5      | 0             | No coins cannot form positive sum |
| [1]     | 4      | 1             | Only 1+1+1+1                      |
| [2]     | 3      | 0             | Odd cannot be formed              |
| [1,2,5] | 5      | 4             | Classic example                   |

| Coins   | Amount | Expected Ways | Notes             |
|---------|--------|---------------|-------------------|
| [2,3,7] | 12     | 4             | Combinations only |

## Complexity

- Time:  $O(n \times \text{amount})$  where  $n$  is number of coin types
- Space:  $O(\text{amount})$  with 1D DP

Coin Change counting teaches you how state order and loop order shape the meaning of a DP. Once you feel this pattern, many counting problems become straightforward.

## 406 Coin Change (Min Coins)

Now we shift from *counting ways* to *finding the fewest coins*. Given a target amount and coin denominations, how can we form the sum using the minimum number of coins? This version of coin change turns counting into optimization, a small twist with big impact.

### What Problem Are We Solving?

Given `coins[]` and a total `amount`, find the minimum number of coins needed to make up that amount. If it's impossible, return -1.

We define:

$$dp[x] = \text{minimum coins to make sum } x$$

with base case:

$$dp[0] = 0$$

Recurrence:

$$dp[x] = \min_{c \in \text{coins}, c \leq x} (dp[x - c] + 1)$$

Each `dp[x]` asks: "If I take coin `c`, what's the best I can do with the remainder?"

## How Does It Work (Plain Language)?

You start from 0 and climb up, building the cheapest way to reach every amount. For each  $x$ , you try all coins  $c \in x$ . If you can make  $x-c$ , add one coin and see if that's better than your current best.

It's like choosing the shortest route to a destination using smaller hops.

Example: coins = [1, 3, 4], amount = 6

| Amount (x) | dp[x] | Explanation  |
|------------|-------|--------------|
| 0          | 0     | Base case    |
| 1          | 1     | $1 \times 1$ |
| 2          | 2     | $1+1$        |
| 3          | 1     | 3            |
| 4          | 1     | 4            |
| 5          | 2     | $1+4$        |
| 6          | 2     | $3+3$        |

So minimum = 2.

## Tiny Code (Easy Versions)

C (Bottom-Up DP)

```
#include <stdio.h>
#define INF 1000000
#define MIN(a,b) ((a)<(b)?(a):(b))

int main(void) {
    int n, amount;
    printf("Enter number of coins and amount: ");
    scanf("%d %d", &n, &amount);

    int coins[n];
    printf("Enter coin values: ");
    for (int i = 0; i < n; i++) scanf("%d", &coins[i]);

    int dp[amount + 1];
    for (int x = 1; x <= amount; x++) dp[x] = INF;
    dp[0] = 0;
```

```

for (int x = 1; x <= amount; x++) {
    for (int i = 0; i < n; i++) {
        int c = coins[i];
        if (x - c >= 0) dp[x] = MIN(dp[x], dp[x - c] + 1);
    }
}

if (dp[amount] == INF)
    printf("Not possible\n");
else
    printf("Min coins: %d\n", dp[amount]);

return 0;
}

```

Python (Straightforward DP)

```

coins = list(map(int, input("Enter coin values: ").split()))
amount = int(input("Enter amount: "))

INF = float('inf')
dp = [INF] * (amount + 1)
dp[0] = 0

for x in range(1, amount + 1):
    for c in coins:
        if x - c >= 0:
            dp[x] = min(dp[x], dp[x - c] + 1)

print("Min coins:" if dp[amount] != INF else "Not possible", end=" ")
print(dp[amount] if dp[amount] != INF else "")

```

## Why It Matters

- Core unbounded optimization DP
- Shows minimization recurrence with base infinity
- Illustrates subproblem dependency:  $dp[x]$  depends on smaller sums
- Connects directly to Knapsack, shortest path, and DP + BFS hybrids

This version teaches you to mix greedy intuition with DP correctness.

## Step-by-Step Example

Coins [1, 3, 4], amount = 6

| x | Try 1       | Try 3       | Try 4       | dp[x] |
|---|-------------|-------------|-------------|-------|
| 0 | -           | -           | -           | 0     |
| 1 | $dp[0]+1=1$ | -           | -           | 1     |
| 2 | $dp[1]+1=2$ | -           | -           | 2     |
| 3 | $dp[2]+1=3$ | $dp[0]+1=1$ | -           | 1     |
| 4 | $dp[3]+1=2$ | $dp[1]+1=2$ | $dp[0]+1=1$ | 1     |
| 5 | $dp[4]+1=2$ | $dp[2]+1=3$ | $dp[1]+1=2$ | 2     |
| 6 | $dp[5]+1=3$ | $dp[3]+1=2$ | $dp[2]+1=3$ | 2     |

Answer =  $dp[6] = 2$

## Try It Yourself

1. Add code to reconstruct the actual coin set.
2. Compare greedy vs DP for coins [1, 3, 4], amount = 6.
3. Modify to handle limited coin supply.
4. Use recursion + memoization and compare runtime.
5. Try edge cases: amount smaller than smallest coin.

## Test Cases

| Coins    | Amount | Expected Output | Notes           |
|----------|--------|-----------------|-----------------|
| [1]      | 3      | 3               | Only 1s         |
| [2]      | 3      | -1              | Impossible      |
| [1,3,4]  | 6      | 2               | 3+3 or 4+1+1    |
| [1,2,5]  | 11     | 3               | 5+5+1           |
| [2,5,10] | 0      | 0               | No coins needed |

## Complexity

- Time:  $O(n \times \text{amount})$
- Space:  $O(\text{amount})$

Coin Change (Min Coins) is a masterclass in thinking minimally, every subproblem is a small decision toward the most efficient path.

## 407 Knapsack 0/1

The 0/1 Knapsack problem is one of the crown jewels of dynamic programming. You're given a backpack with limited capacity and a set of items, each with a weight and a value. You must decide which items to pack so that the total value is maximized without exceeding the weight limit. You can either take an item (1) or leave it (0), hence the name.

### What Problem Are We Solving?

Given:

- $n$  items, each with `weight[i]` and `value[i]`
- capacity  $W$

Find the maximum total value you can carry:

$$dp[i][w] = \text{max value using first } i \text{ items with capacity } w$$

Recurrence:

$$dp[i][w] = \max(dp[i - 1][w], \quad value[i - 1] + dp[i - 1][w - weight[i - 1]])$$

if `weight[i-1] <= w`, else `dp[i][w] = dp[i-1][w]`.

Base:

$$dp[0][w] = 0, \quad dp[i][0] = 0$$

### How Does It Work (Plain Language)?

Think of your backpack as a *budget of space*. Each item is a trade-off:

- Include it → gain its value but lose capacity
- Exclude it → keep capacity for others

You make this decision for every item and every possible capacity.

We build a table where each cell `dp[i][w]` stores the best value you can achieve with the first  $i$  items and total capacity  $w$ .

Example: Items = [( $w=1, v=1$ ), ( $w=3, v=4$ ), ( $w=4, v=5$ ), ( $w=5, v=7$ )],  $W = 7$

| $i/w$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|-------|---|---|---|---|---|---|---|---|
| 0     | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1     | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

| $i/w$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|-------|---|---|---|---|---|---|---|---|
| 2     | 0 | 1 | 1 | 4 | 5 | 5 | 5 | 5 |
| 3     | 0 | 1 | 1 | 4 | 5 | 6 | 6 | 9 |
| 4     | 0 | 1 | 1 | 4 | 5 | 7 | 8 | 9 |

Answer = 9 (items 2 + 3)

### Tiny Code (Easy Versions)

C (2D DP Table)

```
#include <stdio.h>

#define MAX(a,b) ((a)>(b)?(a):(b))

int main(void) {
    int n, W;
    printf("Enter number of items and capacity: ");
    scanf("%d %d", &n, &W);

    int wt[n], val[n];
    printf("Enter weights: ");
    for (int i = 0; i < n; i++) scanf("%d", &wt[i]);
    printf("Enter values: ");
    for (int i = 0; i < n; i++) scanf("%d", &val[i]);

    int dp[n + 1][W + 1];

    for (int i = 0; i <= n; i++) {
        for (int w = 0; w <= W; w++) {
            if (i == 0 || w == 0) dp[i][w] = 0;
            else if (wt[i-1] <= w)
                dp[i][w] = MAX(val[i-1] + dp[i-1][w - wt[i-1]], dp[i-1][w]);
            else
                dp[i][w] = dp[i-1][w];
        }
    }

    printf("Max value: %d\n", dp[n][W]);
    return 0;
}
```

Python (Space Optimized)

```
weights = list(map(int, input("Enter weights: ").split()))
values = list(map(int, input("Enter values: ").split()))
W = int(input("Enter capacity: "))
n = len(weights)

dp = [0] * (W + 1)

for i in range(n):
    for w in range(W, weights[i] - 1, -1):
        dp[w] = max(dp[w], values[i] + dp[w - weights[i]])

print("Max value:", dp[W])
```

## Why It Matters

- Teaches choice-based DP: include or exclude
- Core of resource allocation, subset selection, budgeting problems
- Foundation for advanced DPs (subset sum, partition, scheduling)
- Introduces 2D → 1D space optimization

This problem embodies the essence of decision-making in DP: to take or not to take.

## Step-by-Step Example

Items:  $(w,v) = (1,1), (3,4), (4,5), (5,7)$ ,  $W=7$

| Step | Capacity | Action            | dp                 |
|------|----------|-------------------|--------------------|
| i=1  | w 1      | Take (1,1)        | +1 value           |
| i=2  | w 3      | Take (3,4)        | Replace low combos |
| i=3  | w 4      | Combine (3+4) = 9 | Max found          |
| i=4  | w=7      | Can't beat 9      | Done               |

## Try It Yourself

1. Print selected items using a traceback table.
2. Compare 2D vs 1D versions.
3. Add constraint for exact weight match.
4. Try variants: maximize weight, minimize count, etc.

5. Modify for fractional weights → Greedy Fractional Knapsack.

## Test Cases

| Weights   | Values     | Capacity | Expected | Notes           |
|-----------|------------|----------|----------|-----------------|
| [1,2,3]   | [10,15,40] | 6        | 65       | Take all        |
| [2,3,4,5] | [3,4,5,6]  | 5        | 7        | (2,3)           |
| [1,3,4,5] | [1,4,5,7]  | 7        | 9        | (3,4)           |
| [2,5]     | [5,10]     | 3        | 5        | Only first fits |

## Complexity

- Time:  $O(n \times W)$
- Space:  $O(n \times W) \rightarrow O(W)$  optimized

0/1 Knapsack is the archetype of dynamic programming, it's all about balancing choices, constraints, and rewards.

## 408 Knapsack Unbounded

The Unbounded Knapsack problem is the free refill version of knapsack. You still want to maximize value under a capacity limit, but now each item can be chosen multiple times. It's like packing snacks, you can grab as many as you want, as long as they fit in the bag.

### What Problem Are We Solving?

Given:

- $n$  items with `weight[i]` and `value[i]`
- capacity  $W$
- Unlimited copies of each item

Find the maximum value achievable without exceeding  $W$ .

State:

$$dp[w] = \text{max value for capacity } w$$

Recurrence:

$$dp[w] = \max_{i: weight[i] \leq w} (dp[w - weight[i]] + value[i])$$

Base:

$$dp[0] = 0$$

Notice that this is similar to 0/1 Knapsack, but here we reuse items. The difference lies in the order of iteration.

### How Does It Work (Plain Language)?

Think of capacity  $w$  as a budget. For each capacity, you check all items, if one fits, you see what happens when you reuse it. Unlike 0/1 Knapsack (where each item can only be used once per combination), Unbounded Knapsack allows multiple selections.

| Capacity (w) | Best Value | Explanation          |
|--------------|------------|----------------------|
| 0            | 0          | Empty                |
| 1            | 15         | 1 copy of item(1,15) |
| 2            | 30         | 2 copies             |
| 3            | 45         | 3 copies             |
| 4            | 60         | 4 copies             |

(If all items have same ratio, you'll fill with the best one.)

Example: Items:  $(w, v) = (2,4), (3,7), (4,9), W = 7$

- $dp[2] = 4$
- $dp[3] = 7$
- $dp[4] = 9$
- $dp[5] = \max(dp[3]+4, dp[2]+7) = 11$
- $dp[6] = \max(dp[4]+4, dp[3]+7, dp[2]+9) = 14$
- $dp[7] = \max(dp[5]+4, dp[4]+7, dp[3]+9) = 16$

Answer = 16

### Tiny Code (Easy Versions)

C (1D DP, Unbounded)

```
#include <stdio.h>
#define MAX(a,b) ((a)>(b)?(a):(b))

int main(void) {
    int n, W;
```

```

printf("Enter number of items and capacity: ");
scanf("%d %d", &n, &W);

int wt[n], val[n];
printf("Enter weights: ");
for (int i = 0; i < n; i++) scanf("%d", &wt[i]);
printf("Enter values: ");
for (int i = 0; i < n; i++) scanf("%d", &val[i]);

int dp[W + 1];
for (int w = 0; w <= W; w++) dp[w] = 0;

for (int i = 0; i < n; i++) {
    for (int w = wt[i]; w <= W; w++) {
        dp[w] = MAX(dp[w], val[i] + dp[w - wt[i]]);
    }
}

printf("Max value: %d\n", dp[W]);
return 0;
}

```

Python (Simple Bottom-Up)

```

weights = list(map(int, input("Enter weights: ").split()))
values = list(map(int, input("Enter values: ").split()))
W = int(input("Enter capacity: "))

dp = [0] * (W + 1)

for i in range(len(weights)):
    for w in range(weights[i], W + 1):
        dp[w] = max(dp[w], values[i] + dp[w - weights[i]])

print("Max value:", dp[W])

```

## Why It Matters

- Demonstrates unbounded usage of elements
- Basis for coin change (min), rod cutting, integer break
- Highlights the importance of iteration order in DP

- Connects counting (how many ways) to optimization (best way)

This is where combinatorial explosion becomes manageable.

### Step-by-Step Example

Items: (2,4), (3,7), (4,9), W = 7

| w | dp[w] | Best Choice |
|---|-------|-------------|
| 0 | 0     | -           |
| 1 | 0     | none fits   |
| 2 | 4     | (2)         |
| 3 | 7     | (3)         |
| 4 | 9     | (4)         |
| 5 | 11    | (2+3)       |
| 6 | 14    | (3+3)       |
| 7 | 16    | (3+4)       |

Answer: 16

### Try It Yourself

1. Print the items used (store parent choice).
2. Compare 0/1 and Unbounded outputs.
3. Add a limit on copies, hybrid knapsack.
4. Change objective: minimize number of items.
5. Apply to Rod Cutting problem.

### Test Cases

| Weights   | Values     | W  | Expected | Notes             |
|-----------|------------|----|----------|-------------------|
| [2,3,4]   | [4,7,9]    | 7  | 16       | 3+4               |
| [1,2,3]   | [10,15,40] | 6  | 90       | six 1s            |
| [5,10,20] | [10,30,50] | 20 | 100      | four 5s or one 20 |
| [2,5]     | [5,10]     | 3  | 5        | only one 2        |

## Complexity

- Time:  $O(n \times W)$
- Space:  $O(W)$

Unbounded Knapsack is your first taste of infinite choice under constraint, a powerful idea that flows through many DP designs.

## 409 Longest Increasing Subsequence (DP)

The Longest Increasing Subsequence (LIS) problem is a classic, it's all about finding the longest chain of numbers that strictly increases. You don't have to keep them consecutive, just in order. This is a foundational DP problem that blends state definition, transitions, and comparisons beautifully.

### What Problem Are We Solving?

Given an array  $\text{arr}[]$  of length  $n$ , find the length of the longest increasing subsequence, a sequence of indices  $i_1 < i_2 < \dots < i_k$  such that:

$$\text{arr}[i_1] < \text{arr}[i_2] < \dots < \text{arr}[i_k]$$

We want maximum length.

Recurrence:

$$dp[i] = 1 + \max(dp[j]) \quad \text{for all } j < i \text{ where } \text{arr}[j] < \text{arr}[i]$$

Otherwise:

$$dp[i] = 1$$

Base case:

$$dp[0] = 1$$

Answer:

$$\max_i dp[i]$$

## How Does It Work (Plain Language)?

You look at each number and ask: “*Can I extend an increasing sequence ending before me?*” If yes, take the longest one that fits and extend it by one.

It’s like building towers, each number stacks on top of a smaller one, extending the tallest possible stack.

Example: arr = [10, 22, 9, 33, 21, 50, 41, 60]

| i | arr[i] | dp[i] | Reason            |
|---|--------|-------|-------------------|
| 0 | 10     | 1     | start             |
| 1 | 22     | 2     | 10→22             |
| 2 | 9      | 1     | no smaller before |
| 3 | 33     | 3     | 10→22→33          |
| 4 | 21     | 2     | 10→21             |
| 5 | 50     | 4     | 10→22→33→50       |
| 6 | 41     | 4     | 10→22→33→41       |
| 7 | 60     | 5     | 10→22→33→50→60    |

Answer = 5

## Tiny Code (Easy Versions)

C ( $O(n^2)$ ) DP

```
#include <stdio.h>

#define MAX(a,b) ((a)>(b)?(a):(b))

int main(void) {
    int n;
    printf("Enter number of elements: ");
    scanf("%d", &n);

    int arr[n];
    printf("Enter array: ");
    for (int i = 0; i < n; i++) scanf("%d", &arr[i]);

    int dp[n];
    for (int i = 0; i < n; i++) dp[i] = 1;
```

```

int ans = 1;
for (int i = 1; i < n; i++) {
    for (int j = 0; j < i; j++) {
        if (arr[j] < arr[i])
            dp[i] = MAX(dp[i], dp[j] + 1);
    }
    ans = MAX(ans, dp[i]);
}

printf("Length of LIS: %d\n", ans);
return 0;
}

```

Python (Simple DP)

```

arr = list(map(int, input("Enter array: ").split()))
n = len(arr)
dp = [1] * n

for i in range(n):
    for j in range(i):
        if arr[j] < arr[i]:
            dp[i] = max(dp[i], dp[j] + 1)

print("Length of LIS:", max(dp))

```

## Why It Matters

- Core sequence DP, compares pairs, tracks best chain
- Demonstrates  $O(n^2)$  DP thinking
- Foundation for LCS, Edit Distance, Patience Sorting ( $O(n \log n)$ )
- Applied in stock analysis, genome sequences, and chain problems

This problem teaches “look back and extend”, a key DP instinct.

## Step-by-Step Example

```
arr = [3, 10, 2, 1, 20]
```

| i | arr[i] | dp[i] | Best Chain |
|---|--------|-------|------------|
| 0 | 3      | 1     | [3]        |
| 1 | 10     | 2     | [3,10]     |
| 2 | 2      | 1     | [2]        |
| 3 | 1      | 1     | [1]        |
| 4 | 20     | 3     | [3,10,20]  |

Answer = 3

### Try It Yourself

1. Print the actual LIS using a parent array.
2. Convert to non-decreasing LIS ( instead of <).
3. Compare with  $O(n \log n)$  binary search version.
4. Adapt for longest decreasing subsequence.
5. Apply to 2D pairs (Russian Doll Envelopes).

### Test Cases

| arr                      | Expected | Notes                    |
|--------------------------|----------|--------------------------|
| [1,2,3,4,5]              | 5        | Already increasing       |
| [5,4,3,2,1]              | 1        | Only one element         |
| [3,10,2,1,20]            | 3        | [3,10,20]                |
| [10,22,9,33,21,50,41,60] | 5        | Classic example          |
| [2,2,2,2]                | 1        | Strictly increasing only |

### Complexity

- Time:  $O(n^2)$
- Space:  $O(n)$

LIS is the melody of DP, every element listens to its predecessors, finds harmony, and extends the tune to its fullest length.

## 410 Edit Distance (Levenshtein)

The Edit Distance (or Levenshtein distance) problem measures how different two strings are by counting the minimum number of operations needed to transform one into the other. The allowed operations are:

- Insert
- Delete
- Replace

It's the foundation of spell checkers, DNA sequence alignment, and fuzzy search, anywhere we need to measure "how close" two sequences are.

### What Problem Are We Solving?

Given two strings A and B, find the minimum number of operations required to convert A → B.

Let:

- A has length m
- B has length n

State:

$$dp[i][j] = \min \text{ edits to convert } A[0..i-1] \text{ to } B[0..j-1]$$

Recurrence:

1. If  $A[i-1] == B[j-1]$ :

$$dp[i][j] = dp[i-1][j-1]$$

2. Else, take min of the three operations:

$$dp[i][j] = 1 + \min(dp[i-1][j], \text{ (delete)}dp[i][j-1], \text{ (insert)}dp[i-1][j-1] \text{ (replace)})$$

Base:

$$dp[0][j] = j, \quad dp[i][0] = i$$

(empty string conversions)

## How Does It Work (Plain Language)?

Imagine editing a word character by character. At each step, compare the current letters:

- If they match → no cost, move diagonally.
- If they differ → choose the cheapest fix (insert, delete, replace).

The DP table builds all prefix transformations, from small strings to full ones.

Example: A = "kitten", B = "sitting"

| Step          | Operation | Result  |
|---------------|-----------|---------|
| Replace k → s | sitten    |         |
| Replace e → i | sittin    |         |
| Insert g      |           | sitting |

Answer = 3

## Tiny Code (Easy Versions)

C (2D DP)

```
#include <stdio.h>
#define MIN3(a,b,c) ((a<b?a:b)<c?(a<b?a:b):c)

int main(void) {
    char A[100], B[100];
    printf("Enter string A: ");
    scanf("%s", A);
    printf("Enter string B: ");
    scanf("%s", B);

    int m = 0, n = 0;
    while (A[m]) m++;
    while (B[n]) n++;

    int dp[m + 1][n + 1];

    for (int i = 0; i <= m; i++) dp[i][0] = i;
    for (int j = 0; j <= n; j++) dp[0][j] = j;

    for (int i = 1; i <= m; i++) {
```

```

        for (int j = 1; j <= n; j++) {
            if (A[i-1] == B[j-1]) dp[i][j] = dp[i-1][j-1];
            else dp[i][j] = 1 + MIN3(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]);
        }
    }

    printf("Edit distance: %d\n", dp[m][n]);
    return 0;
}

```

Python (Simple Version)

```

A = input("Enter string A: ")
B = input("Enter string B: ")

m, n = len(A), len(B)
dp = [[0]*(n+1) for _ in range(m+1)]

for i in range(m+1): dp[i][0] = i
for j in range(n+1): dp[0][j] = j

for i in range(1, m+1):
    for j in range(1, n+1):
        if A[i-1] == B[j-1]:
            dp[i][j] = dp[i-1][j-1]
        else:
            dp[i][j] = 1 + min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1])

print("Edit distance:", dp[m][n])

```

## Why It Matters

- Illustrates 2D DP on strings
- Introduces transformation problems
- Forms the backbone of spell correction, DNA alignment, diff tools
- Beautifully captures state = prefix lengths pattern

Edit Distance is the dictionary definition of “step-by-step transformation.”

## Step-by-Step Example

A = "intention", B = "execution"

| $A/B$ | " | e | x | e | c | u | t | i | o | n |
|-------|---|---|---|---|---|---|---|---|---|---|
| " "   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| i     | 1 | 1 | 2 | 3 | 4 | 5 | 6 | 6 | 7 | 8 |
| n     | 2 | 2 | 2 | 3 | 4 | 5 | 6 | 7 | 7 | 7 |
| t     | 3 | 3 | 3 | 3 | 4 | 5 | 5 | 6 | 7 | 8 |
| e     | 4 | 3 | 4 | 3 | 4 | 5 | 6 | 6 | 7 | 8 |
| n     | 5 | 4 | 4 | 4 | 4 | 5 | 6 | 7 | 7 | 7 |
| t     | 6 | 5 | 5 | 5 | 5 | 5 | 5 | 6 | 7 | 8 |
| i     | 7 | 6 | 6 | 6 | 6 | 6 | 6 | 5 | 6 | 7 |
| o     | 8 | 7 | 7 | 7 | 7 | 7 | 7 | 6 | 5 | 6 |
| n     | 9 | 8 | 8 | 8 | 8 | 8 | 8 | 7 | 6 | 5 |

Answer = 5 edits

## Try It Yourself

1. Print actual edit sequence (backtrack).
2. Add costs: assign different weights for insert/delete/replace.
3. Try case-insensitive variant.
4. Compare with Longest Common Subsequence.
5. Implement recursive + memoized version.

## Test Cases

| A        | B         | Expected | Notes            |
|----------|-----------|----------|------------------|
| "kitten" | "sitting" | 3        | classic          |
| "horse"  | "ros"     | 3        | leetcode         |
| "flaw"   | "lawn"    | 2        | replace + insert |
| "abc"    | "yabd"    | 2        | insert + replace |
| " "      | "abc"     | 3        | all inserts      |

## **Complexity**

- Time:  $O(m \times n)$
- Space:  $O(m \times n)$ , reducible to  $O(n)$

Edit Distance teaches precision in DP: every cell means “*smallest change to fix this prefix*”. It’s the language of correction, one letter at a time.

# Section 42. Classic Problems

## 411 0/1 Knapsack

The 0/1 Knapsack is one of the most iconic problems in dynamic programming. It's the perfect example of decision-making under constraints, each item can either be taken or left, but never split or repeated. The goal is to maximize total value within a fixed capacity.

This version focuses on understanding choice, capacity, and optimal substructure, the three pillars of DP.

### What Problem Are We Solving?

Given:

- $n$  items, each with weight  $w[i]$  and value  $v[i]$
- a knapsack of capacity  $W$

We want:

$$\text{maximize total value } \sum v[i]$$

subject to

$$\sum w[i] \leq W$$

and each item can be used at most once.

State definition:

$$dp[i][w] = \text{max value using first } i \text{ items with capacity } w$$

Recurrence:

$$dp[i][w] = \begin{cases} dp[i-1][w], & \text{if } w_{i-1} > w, \\ \max(dp[i-1][w], dp[i-1][w - w_{i-1}] + v_{i-1}), & \text{otherwise.} \end{cases}$$

Base case:

$$dp[0][w] = 0, \quad dp[i][0] = 0$$

Answer:

$$dp[n][W]$$

## How Does It Work (Plain Language)?

At every step, you ask: “Should I take this item or leave it?” If it fits, compare:

- Not taking it → stick with previous best ( $dp[i-1][w]$ )
- Taking it → add its value plus best value for remaining capacity ( $dp[i-1][w - weight[i]] + value[i]$ )

The DP table stores the best possible value at every sub-capacity for each subset of items.

| Item | Weight | Value |
|------|--------|-------|
| 1    | 1      | 1     |
| 2    | 3      | 4     |
| 3    | 4      | 5     |
| 4    | 5      | 7     |

Capacity = 7 → Answer = 9 (items 2 + 3)

## Tiny Code (Easy Versions)

C (2D DP Table)

```
#include <stdio.h>
#define MAX(a,b) ((a)>(b)?(a):(b))

int main(void) {
    int n, W;
    printf("Enter number of items and capacity: ");
    scanf("%d %d", &n, &W);

    int wt[n], val[n];
    printf("Enter weights: ");
    for (int i = 0; i < n; i++) scanf("%d", &wt[i]);
    printf("Enter values: ");
    for (int i = 0; i < n; i++) scanf("%d", &val[i]);

    int dp[n + 1][W + 1];

    for (int i = 0; i <= n; i++) {
        for (int w = 0; w <= W; w++) {
            if (i == 0 || w == 0) dp[i][w] = 0;
```

```

        else if (wt[i-1] <= w)
            dp[i][w] = MAX(val[i-1] + dp[i-1][w - wt[i-1]], dp[i-1][w]);
        else
            dp[i][w] = dp[i-1][w];
    }
}

printf("Max value: %d\n", dp[n][W]);
return 0;
}

```

Python (1D Optimized)

```

weights = list(map(int, input("Enter weights: ").split()))
values = list(map(int, input("Enter values: ").split()))
W = int(input("Enter capacity: "))
n = len(weights)

dp = [0] * (W + 1)

for i in range(n):
    for w in range(W, weights[i] - 1, -1):
        dp[w] = max(dp[w], dp[w - weights[i]] + values[i])

print("Max value:", dp[W])

```

## Why It Matters

- Introduces decision-based DP: take or skip
- Builds on recurrence intuition (state transition)
- Forms basis for subset sum, equal partition, and resource allocation
- Teaches capacity-dependent states

It's the first time you feel the *tension* between greedy desire and constrained reality, a DP classic.

## Step-by-Step Example

Items:  $(w,v) = (1,1), (3,4), (4,5), (5,7)$ ,  $W = 7$

| $i/w$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|-------|---|---|---|---|---|---|---|---|
| 0     | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1     | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2     | 0 | 1 | 1 | 4 | 5 | 5 | 5 | 5 |
| 3     | 0 | 1 | 1 | 4 | 5 | 6 | 6 | 9 |
| 4     | 0 | 1 | 1 | 4 | 5 | 7 | 8 | 9 |

Answer = 9

### Try It Yourself

1. Add code to reconstruct chosen items.
2. Compare 2D vs 1D DP outputs.
3. Modify to minimize weight for a given value.
4. Visualize table transitions for small inputs.
5. Experiment with large weights, test performance.

### Test Cases

| Weights   | Values     | W | Expected | Notes           |
|-----------|------------|---|----------|-----------------|
| [1,2,3]   | [10,15,40] | 6 | 65       | all items fit   |
| [2,3,4,5] | [3,4,5,6]  | 5 | 7        | (2,3)           |
| [1,3,4,5] | [1,4,5,7]  | 7 | 9        | (3,4)           |
| [2,5]     | [5,10]     | 3 | 5        | only first fits |

### Complexity

- Time:  $O(n \times W)$
- Space:  $O(n \times W) \rightarrow O(W)$  (optimized)

0/1 Knapsack is the heartbeat of DP, every decision echoes the fundamental trade-off: *to take or not to take.*

## 412 Subset Sum

The Subset Sum problem is a fundamental example of boolean dynamic programming. Instead of maximizing or minimizing, we simply ask “Is it possible?”, can we pick a subset of numbers that adds up to a given target?

This problem forms the foundation for many combinatorial DP problems such as Equal Partition, Count of Subsets, Target Sum, and even Knapsack itself.

### What Problem Are We Solving?

Given:

- An array `arr[]` of  $n$  positive integers
- A target sum  $S$

Determine whether there exists a subset of `arr[]` whose elements sum to exactly  $S$ .

We define:

$$dp[i][s] = \text{true if subset of first } i \text{ elements can form sum } s$$

Recurrence:

- If  $\text{arr}[i-1] > s$ : [  $dp[i][s] = dp[i-1][s]$  ]
- Else: [  $dp[i][s] = dp[i-1][s] \cup dp[i-1][s - \text{arr}[i-1]]$  ]

Base cases:

$$dp[0][0] = \text{true}, \quad dp[0][s > 0] = \text{false}$$

Answer:

$$dp[n][S]$$

### How Does It Work (Plain Language)?

Think of it as a yes/no table:

- Rows  $\rightarrow$  items
- Columns  $\rightarrow$  sums

Each cell asks: “*Can I form sum  $s$  using the first  $i$  items?*” The answer comes from either skipping or including the current item.

Example: `arr = [2, 3, 7, 8, 10], S = 11`

| $i/Sum$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---------|---|---|---|---|---|---|---|---|---|---|----|----|
| 0       | T | F | F | F | F | F | F | F | F | F | F  | F  |
| 1 (2)   | T | F | T | F | F | F | F | F | F | F | F  | F  |
| 2 (3)   | T | F | T | T | F | T | F | F | F | F | F  | F  |
| 3 (7)   | T | F | T | T | F | T | F | T | T | F | T  | T  |

$\text{dp}[5][11] = \text{True} \rightarrow [3, 8]$  is one valid subset.

### Tiny Code (Easy Versions)

C (2D Boolean Table)

```
#include <stdio.h>

int main(void) {
    int n, S;
    printf("Enter number of elements and target sum: ");
    scanf("%d %d", &n, &S);

    int arr[n];
    printf("Enter elements: ");
    for (int i = 0; i < n; i++) scanf("%d", &arr[i]);

    int dp[n + 1][S + 1];
    for (int i = 0; i <= n; i++) dp[i][0] = 1;
    for (int s = 1; s <= S; s++) dp[0][s] = 0;

    for (int i = 1; i <= n; i++) {
        for (int s = 1; s <= S; s++) {
            if (arr[i-1] > s)
                dp[i][s] = dp[i-1][s];
            else
                dp[i][s] = dp[i-1][s] || dp[i-1][s - arr[i-1]];
        }
    }

    printf("Subset sum %s possible\n", dp[n][S] ? "is" : "is not");
    return 0;
}
```

Python (1D Optimization)

```

arr = list(map(int, input("Enter elements: ").split()))
S = int(input("Enter target sum: "))

dp = [False] * (S + 1)
dp[0] = True

for num in arr:
    for s in range(S, num - 1, -1):
        dp[s] = dp[s] or dp[s - num]

print("Subset sum is possible" if dp[S] else "Not possible")

```

## Why It Matters

- Introduces boolean DP (true/false states)
- Foundation for Equal Partition, Target Sum, and Count Subsets
- Closely related to 0/1 Knapsack but without values
- Perfect exercise for learning state dependency

This problem captures the logic of feasibility: “*If I could make  $s - arr[i]$  before, then I can make  $s$  now.*”

## Step-by-Step Example

Array = [2, 3, 7, 8, 10], S = 11

| Step | Consider                       | New True Sums |
|------|--------------------------------|---------------|
| 2    | [0,2]                          | {2}           |
| 3    | [0,2,3,5]                      | {3,5}         |
| 7    | [0,2,3,5,7,9,10,12]            | {7,9,10}      |
| 8    | [0,2,3,5,7,8,9,10,11,12,13,15] | {11}          |

Found 11.

## Try It Yourself

1. Print one valid subset using a parent pointer table.
2. Count the total number of valid subsets (convert to count DP).
3. Try with duplicates, does it change anything?

4. Modify to check if sum is divisible by  $k$ .
5. Add negative numbers (use offset shifting).

### Test Cases

| arr          | S  | Expected | Notes     |
|--------------|----|----------|-----------|
| [2,3,7,8,10] | 11 | True     | 3+8       |
| [1,2,3]      | 5  | True     | 2+3       |
| [1,2,5]      | 4  | False    | no subset |
| [1,1,1,1]    | 2  | True     | 1+1       |
| [5,2,6,4]    | 13 | True     | 5+4+4     |

### Complexity

- Time:  $O(n \times S)$
- Space:  $O(n \times S) \rightarrow O(S)$  (optimized)

Subset Sum is a cornerstone of DP, a yes/no version of Knapsack that teaches how logic flows through states, one sum at a time.

## 413 Equal Partition

The Equal Partition problem asks a natural question: can we divide a set of numbers into two subsets with equal sum? It's a direct application of Subset Sum, reframed as a partitioning challenge. If the total sum is even, we check if there's a subset that sums to half, that ensures the other subset sums to the same.

### What Problem Are We Solving?

Given:

- An array `arr[]` of  $n$  positive integers

Determine whether it can be partitioned into two subsets whose sums are equal.

Let total sum be  $S$ . We need to check:

- If  $S$  is odd  $\rightarrow$  impossible
- If  $S$  is even  $\rightarrow$  check if Subset Sum to  $S/2$  is possible

So the problem reduces to:

Is there a subset of  $arr[]$  with sum  $= S/2$ ?

We use the same recurrence from Subset Sum:

$$dp[i][s] = dp[i - 1][s] \vee dp[i - 1][s - arr[i - 1]]$$

Base:

$$dp[0][0] = \text{true}$$

Answer:

$$dp[n][S/2]$$

### How Does It Work (Plain Language)?

1. Compute total sum  $S$ .
2. If  $S$  is odd  $\rightarrow$  cannot split evenly.
3. Otherwise, use Subset Sum DP to check if we can reach  $S/2$ . If yes, one subset forms  $S/2$ , and the remaining numbers automatically form the other half.

Example:  $arr = [1, 5, 11, 5]$

- Sum = 22
- Target = 11
- Can we make 11? Yes  $\rightarrow [11]$  and  $[1, 5, 5]$  are two halves.

### Tiny Code (Easy Versions)

C (2D DP Table)

```
#include <stdio.h>

int main(void) {
    int n;
    printf("Enter number of elements: ");
    scanf("%d", &n);

    int arr[n];
    printf("Enter elements: ");
    for (int i = 0; i < n; i++) scanf("%d", &arr[i]);
```

```

int sum = 0;
for (int i = 0; i < n; i++) sum += arr[i];

if (sum % 2 != 0) {
    printf("Cannot partition into equal sum subsets\n");
    return 0;
}

int target = sum / 2;
int dp[n + 1][target + 1];

for (int i = 0; i <= n; i++) dp[i][0] = 1;
for (int s = 1; s <= target; s++) dp[0][s] = 0;

for (int i = 1; i <= n; i++) {
    for (int s = 1; s <= target; s++) {
        if (arr[i-1] > s)
            dp[i][s] = dp[i-1][s];
        else
            dp[i][s] = dp[i-1][s] || dp[i-1][s - arr[i-1]];
    }
}

printf("Equal partition %s possible\n", dp[n][target] ? "is" : "is not");
return 0;
}

```

Python (1D Space Optimization)

```

arr = list(map(int, input("Enter elements: ").split()))
S = sum(arr)

if S % 2 != 0:
    print("Cannot partition into equal subsets")
else:
    target = S // 2
    dp = [False] * (target + 1)
    dp[0] = True

    for num in arr:
        for s in range(target, num - 1, -1):

```

```

dp[s] = dp[s] or dp[s - num]

print("Equal partition is possible" if dp[target] else "Not possible")

```

## Why It Matters

- Builds directly on Subset Sum
- Demonstrates problem reduction in DP
- Useful for balanced partitioning, load balancing, and fair division
- Teaches thinking in terms of state feasibility

Equal Partition shows how yes/no DPs can solve seemingly complex questions with simple logic.

## Step-by-Step Example

`arr = [1, 5, 11, 5]`

1.  $S = 22 \rightarrow S/2 = 11$
2. Use Subset Sum DP to check if 11 can be formed
3. True  $\rightarrow$  subsets [11] and [1,5,5]

Another case: `arr = [1, 2, 3, 5]`

- $S = 11$  (odd)  $\rightarrow$  cannot partition.

## Try It Yourself

1. Print the actual subsets (traceback table).
2. Try arrays with duplicates.
3. Compare with total sum odd case.
4. Add constraint: must use at least one element in each subset.
5. Visualize dp table for small arrays.

| arr | Sum | Expected | Notes |
|-----|-----|----------|-------|
|-----|-----|----------|-------|

## Test Cases

| arr         | Sum | Expected | Notes        |
|-------------|-----|----------|--------------|
| [1,5,11,5]  | 22  | True     | 11 and 11    |
| [1,2,3,5]   | 11  | False    | odd total    |
| [3,3,3,3]   | 12  | True     | split evenly |
| [2,2,2,2,2] | 10  | True     | 5+5          |
| [1,1,3,4,7] | 16  | True     | 8+8          |

## Complexity

- Time:  $O(n \times S/2)$
- Space:  $O(S/2)$  (optimized)

Equal Partition is the first real taste of reduction in dynamic programming, take a bigger problem, express it as Subset Sum, and solve with the same machinery.

## 414 Count of Subsets with Sum

The Count of Subsets with Sum problem extends the Subset Sum idea. Instead of asking “*Is it possible to form this sum?*”, we ask “*In how many ways can we form it?*”. This transforms a boolean DP into a counting DP, where each state accumulates the number of combinations that yield a given sum.

### What Problem Are We Solving?

Given:

- An array `arr[]` of  $n$  positive integers
- A target sum  $S$

We want the number of subsets whose elements sum exactly to  $S$ .

We define the state:

$$dp[i][s] = \text{number of ways to form sum } s \text{ using first } i \text{ elements}$$

The recurrence:

$$dp[i][s] = \begin{cases} dp[i-1][s], & \text{if } arr[i-1] > s, \\ dp[i-1][s] + dp[i-1][s - arr[i-1]], & \text{otherwise.} \end{cases}$$

Base cases:

$$dp[0][0] = 1, \quad dp[0][s > 0] = 0$$

Final answer:

$$dp[n][S]$$

### How Does It Work (Plain Language)

Each element gives two paths: include or exclude. If you include it, you count all subsets that formed `s - arr[i-1]` before. If you exclude it, you inherit all subsets that already formed `s`. So each cell accumulates total combinations from both branches.

Example: `arr = [2, 3, 5, 6, 8, 10], S = 10`

Ways to form 10:

- `{10}`
- `{2, 8}`
- `{2, 3, 5}`

Answer = 3

### Tiny Code (Easy Versions)

C (2D DP Table)

```
#include <stdio.h>

int main(void) {
    int n, S;
    printf("Enter number of elements and target sum: ");
    scanf("%d %d", &n, &S);

    int arr[n];
```

```

printf("Enter elements: ");
for (int i = 0; i < n; i++) scanf("%d", &arr[i]);

int dp[n + 1][S + 1];

for (int i = 0; i <= n; i++) dp[i][0] = 1;
for (int s = 1; s <= S; s++) dp[0][s] = 0;

for (int i = 1; i <= n; i++) {
    for (int s = 0; s <= S; s++) {
        if (arr[i-1] > s)
            dp[i][s] = dp[i-1][s];
        else
            dp[i][s] = dp[i-1][s] + dp[i-1][s - arr[i-1]];
    }
}

printf("Number of subsets: %d\n", dp[n][S]);
return 0;
}

```

Python (1D Space Optimized)

```

arr = list(map(int, input("Enter elements: ").split()))
S = int(input("Enter target sum: "))

dp = [0] * (S + 1)
dp[0] = 1

for num in arr:
    for s in range(S, num - 1, -1):
        dp[s] += dp[s - num]

print("Number of subsets:", dp[S])

```

## Why It Matters

- Extends Subset Sum from feasibility to counting
- Foundation for Target Sum, Equal Partition Count, and Combinatorics DP
- Shows how a small change in recurrence changes meaning
- Demonstrates accumulation instead of boolean OR

This is where DP transitions from *logic* to *combinatorics*, from “can I?” to “how many ways?”

### Step-by-Step Example

`arr = [2, 3, 5, 6, 8, 10], S = 10`

| i | arr[i] | Ways to form 10 | Explanation          |
|---|--------|-----------------|----------------------|
| 1 | 2      | 0               | cannot reach 10 yet  |
| 2 | 3      | 0               | $2+3=5$ only         |
| 3 | 5      | 1               | {5}                  |
| 4 | 6      | 1               | {10}                 |
| 5 | 8      | 2               | {2,8}, {10}          |
| 6 | 10     | 3               | {10}, {2,8}, {2,3,5} |

Answer = 3

### A Gentle Proof (Why It Works)

We build  $dp[i][s]$  using the inclusion-exclusion principle:

To form sum  $s$  using first  $i$  items, two possibilities exist:

1. Exclude  $arr[i-1]$ : all subsets that form  $s$  remain valid

$$dp[i-1][s]$$

2. Include  $arr[i-1]$ : each subset that formed  $s - arr[i-1]$  now forms  $s$

$$dp[i-1][s - arr[i-1]]$$

Thus:

$$dp[i][s] = dp[i-1][s] + dp[i-1][s - arr[i-1]]$$

No double counting occurs since each element is processed once, contributing to exactly one branch per subproblem. By building layer by layer,  $dp[n][S]$  accumulates all valid subset combinations summing to  $S$ .

## Try It Yourself

1. Print all valid subsets using recursive backtracking.
2. Modify the DP to count subsets with sum = target.
3. Add duplicates and compare results.
4. Apply modulo  $10^9 + 7$  to handle large counts.
5. Extend to count subsets with sum difference = D.

## Test Cases

| arr            | S  | Expected | Notes                |
|----------------|----|----------|----------------------|
| [2,3,5,6,8,10] | 10 | 3        | {10}, {2,8}, {2,3,5} |
| [1,1,1,1]      | 2  | 6        | choose any 2         |
| [1,2,3]        | 3  | 2        | {3}, {1,2}           |
| [1,2,5]        | 4  | 0        | no subset            |
| [2,4,6,10]     | 16 | 2        | {6,10}, {2,4,10}     |

## Complexity

- Time:  $O(n \times S)$
- Space:  $O(S)$

The Count of Subsets with Sum problem is a perfect illustration of how dynamic programming can evolve from feasibility to enumeration, counting every path that leads to success.

## 415 Target Sum

The Target Sum problem combines Subset Sum and sign assignment, instead of selecting elements, you assign + or - to each one so that their total equals a target value. It's a beautiful example of how DP can turn algebraic constraints into combinatorial counting.

### What Problem Are We Solving?

Given:

- An array `arr[]` of `n` non-negative integers
- A target value `T`

Count the number of ways to assign + or - signs to elements so that:

$$a_1 \pm a_2 \pm a_3 \pm \dots \pm a_n = T$$

Each element must appear once with either sign.

We define:

- Let total sum be  $S = \sum arr[i]$

If we split into two subsets  $P$  (positive) and  $N$  (negative), we have:

$$\begin{cases} P + N = S \\ P - N = T \end{cases}$$

Solve these equations:

$$P = \frac{S + T}{2}$$

So the problem becomes count subsets whose sum =  $(S + T)/2$ .

If  $(S + T)$  is odd or  $T > S$ , answer = 0 (impossible).

### **Key Idea**

Convert the sign problem into a subset counting problem:

$$\text{Count subsets with sum } P = \frac{S + T}{2}$$

Then use the recurrence from Count of Subsets with Sum:

$$dp[i][p] = \begin{cases} dp[i-1][p], & \text{if } arr[i-1] > p, \\ dp[i-1][p] + dp[i-1][p - arr[i-1]], & \text{otherwise.} \end{cases}$$

Base case:

$$dp[0][0] = 1$$

Answer:

$$dp[n][P]$$

## How Does It Work (Plain Language)

Think of each element as being placed on one of two sides: positive or negative. Instead of directly simulating signs, we compute how many subsets sum to  $(S + T)/2$ . That subset represents all numbers assigned  $+$ ; the rest implicitly become  $-$ .

Example: `arr = [1, 1, 2, 3], T = 1 Sum S = 7 → P = (7 + 1)/2 = 4`

So we count subsets summing to 4:

- {1, 3}
- {1, 1, 2} Answer = 2

## Tiny Code (Easy Versions)

C (2D DP Table)

```
#include <stdio.h>

int main(void) {
    int n, T;
    printf("Enter number of elements and target: ");
    scanf("%d %d", &n, &T);

    int arr[n];
    printf("Enter elements: ");
    for (int i = 0; i < n; i++) scanf("%d", &arr[i]);

    int S = 0;
    for (int i = 0; i < n; i++) S += arr[i];

    if ((S + T) % 2 != 0 || T > S) {
        printf("No solutions\n");
        return 0;
    }

    int P = (S + T) / 2;
    int dp[n + 1][P + 1];

    for (int i = 0; i <= n; i++) dp[i][0] = 1;
    for (int j = 1; j <= P; j++) dp[0][j] = 0;

    for (int i = 1; i <= n; i++) {
```

```

        for (int j = 0; j <= P; j++) {
            if (arr[i-1] > j)
                dp[i][j] = dp[i-1][j];
            else
                dp[i][j] = dp[i-1][j] + dp[i-1][j - arr[i-1]];
        }
    }

    printf("Number of ways: %d\n", dp[n][P]);
    return 0;
}

```

Python (1D Space Optimized)

```

arr = list(map(int, input("Enter elements: ").split()))
T = int(input("Enter target: "))
S = sum(arr)

if (S + T) % 2 != 0 or T > S:
    print("No solutions")
else:
    P = (S + T) // 2
    dp = [0] * (P + 1)
    dp[0] = 1

    for num in arr:
        for s in range(P, num - 1, -1):
            dp[s] += dp[s - num]

    print("Number of ways:", dp[P])

```

## Why It Matters

- Transforms sign assignment into subset counting
- Reinforces algebraic manipulation in DP
- Foundation for expression evaluation, partition problems, and probabilistic sums
- Demonstrates how mathematical reformulation simplifies state design

It's a powerful example of turning a tricky  $\pm$  sum problem into a familiar counting DP.

### Step-by-Step Example

`arr = [1, 1, 2, 3], T = 1 S = 7 → P = 4`

Subsets summing to 4:

- {1, 3}
- {1, 1, 2}

Answer = 2

### A Gentle Proof (Why It Works)

Let the positive subset sum be  $P$  and negative subset sum be  $N$ . We have:

$$P - N = T \quad \text{and} \quad P + N = S$$

Adding both:

$$2P = S + T \implies P = \frac{S + T}{2}$$

Thus, any valid assignment of signs corresponds exactly to one subset summing to  $P$ . Every subset of sum  $P$  defines a unique sign configuration:

- Numbers in  $P \rightarrow$  positive
- Numbers not in  $P \rightarrow$  negative

So counting subsets with sum  $P$  is equivalent to counting all valid sign assignments.

### Try It Yourself

1. Handle zeros (they double the count).
2. Return all possible sign configurations.
3. Check with negative  $T$  (same symmetry).
4. Compare brute-force enumeration with DP result.
5. Modify for constraints like “at least  $k$  positive numbers”.

| arr | T | Expected | Notes |
|-----|---|----------|-------|
|-----|---|----------|-------|

## Test Cases

| arr       | T | Expected | Notes                  |
|-----------|---|----------|------------------------|
| [1,1,2,3] | 1 | 2        | {1,3}, {1,1,2}         |
| [1,2,3]   | 0 | 2        | {1,2,3} and {-1,-2,-3} |
| [2,2,2,2] | 4 | 6        | many ways              |
| [1,1,1,1] | 0 | 6        | symmetric partitions   |
| [5,3,2,1] | 5 | 2        | {2,3}, {5}             |

## Complexity

- Time:  $O(n \times P)$
- Space:  $O(P)$

The Target Sum problem shows how algebra and DP meet: by reinterpreting signs as subsets, you turn a puzzle of pluses and minuses into a clean combinatorial count.

## 416 Unbounded Knapsack

The Unbounded Knapsack problem is the unlimited version of the classic 0/1 Knapsack. Here, each item can be chosen multiple times, as long as total weight stays within capacity. It's one of the most elegant illustrations of reusable states in dynamic programming.

### What Problem Are We Solving?

Given:

- $n$  items, each with `weight[i]` and `value[i]`
- a knapsack of capacity  $W$

Find the maximum total value achievable without exceeding capacity  $W$ . Each item can be used any number of times.

We define the state:

$$dp[w] = \text{maximum value for capacity } w$$

Recurrence:

$$dp[w] = \max_{i: weight[i] \leq w} (dp[w - weight[i]] + value[i])$$

Base:

$$dp[0] = 0$$

Final answer:

$$dp[W]$$

The key difference from 0/1 Knapsack is order of iteration — for unbounded, we move forward through weights so items can be reused.

### How Does It Work (Plain Language)

Think of the capacity as a ladder. At each rung  $w$ , you check every item:

- If it fits, you ask: “*If I take this item, what’s the best I can do with the remaining space?*”
- Since items are reusable, you can add it again later.

This way, every  $dp[w]$  builds from smaller capacities, each possibly using the same item again.

Example: Items = (weight, value): (2,4), (3,7), (4,9)  $W = 7$

| Capacity | dp[w] | Explanation  |
|----------|-------|--------------|
| 0        | 0     | base         |
| 1        | 0     | no item fits |
| 2        | 4     | one (2,4)    |
| 3        | 7     | one (3,7)    |
| 4        | 9     | one (4,9)    |
| 5        | 11    | (2,4)+(3,7)  |
| 6        | 14    | (3,7)+(3,7)  |
| 7        | 16    | (3,7)+(4,9)  |

Answer = 16

## Tiny Code (Easy Versions)

C (Bottom-Up 1D DP)

```
#include <stdio.h>
#define MAX(a,b) ((a) > (b) ? (a) : (b))

int main(void) {
    int n, W;
    printf("Enter number of items and capacity: ");
    scanf("%d %d", &n, &W);

    int wt[n], val[n];
    printf("Enter weights: ");
    for (int i = 0; i < n; i++) scanf("%d", &wt[i]);
    printf("Enter values: ");
    for (int i = 0; i < n; i++) scanf("%d", &val[i]);

    int dp[W + 1];
    for (int w = 0; w <= W; w++) dp[w] = 0;

    for (int i = 0; i < n; i++) {
        for (int w = wt[i]; w <= W; w++) {
            dp[w] = MAX(dp[w], val[i] + dp[w - wt[i]]);
        }
    }

    printf("Max value: %d\n", dp[W]);
    return 0;
}
```

Python (Iterative Version)

```
weights = list(map(int, input("Enter weights: ").split()))
values = list(map(int, input("Enter values: ").split()))
W = int(input("Enter capacity: "))

dp = [0] * (W + 1)

for i in range(len(weights)):
    for w in range(weights[i], W + 1):
        dp[w] = max(dp[w], values[i] + dp[w - weights[i]])
```

```
print("Max value:", dp[w])
```

## Why It Matters

- Demonstrates reusable subproblems, items can appear multiple times
- Connects to Coin Change (Min Coins) and Rod Cutting
- Foundation for integer partition, resource allocation, and bounded-unbounded hybrid problems
- Teaches forward iteration logic

Unbounded Knapsack is the perfect showcase of DP with repetition.

## Step-by-Step Example

Items: (2,4), (3,7), (4,9),  $W = 7$

| w | dp[w] | Best Choice |
|---|-------|-------------|
| 0 | 0     | base        |
| 1 | 0     | none fits   |
| 2 | 4     | (2)         |
| 3 | 7     | (3)         |
| 4 | 9     | (4)         |
| 5 | 11    | (2,3)       |
| 6 | 14    | (3,3)       |
| 7 | 16    | (3,4)       |

Answer = 16

## A Gentle Proof (Why It Works)

For each capacity  $w$ , we consider every item  $i$  such that  $\text{weight}[i] \leq w$ . If we choose item  $i$ , we gain its value plus the best value achievable for the remaining space  $w - \text{weight}[i]$ :

$$dp[w] = \max_i (value[i] + dp[w - \text{weight}[i]])$$

Unlike 0/1 Knapsack (which must avoid reuse), this recurrence allows reuse because  $dp[w - \text{weight}[i]]$  is computed before  $dp[w]$  in the same pass, meaning item  $i$  can contribute multiple times.

By filling the array from 0 to  $W$ , every capacity's best value is derived from optimal substructures of smaller capacities.

### Try It Yourself

1. Print chosen items by tracking predecessors.
2. Compare with 0/1 Knapsack results.
3. Add constraint: each item  $\leq k$  copies.
4. Apply to Rod Cutting: `weight = length, value = price`.
5. Experiment with fractional weights (greedy fails here).

### Test Cases

| Weights   | Values        | W  | Expected | Notes                |
|-----------|---------------|----|----------|----------------------|
| [2,3,4]   | [4,7,9]       | 7  | 16       | (3,4)                |
| [5,10,15] | [10,30,50]    | 20 | 100      | four 5s or two 10s   |
| [1,2,3]   | [10,15,40]    | 6  | 90       | six 1s               |
| [2,5]     | [5,10]        | 3  | 5        | one 2                |
| [1,3,4,5] | [10,40,50,70] | 8  | 160      | multiples of 1 and 3 |

### Complexity

- Time:  $O(n \times W)$
- Space:  $O(W)$

The Unbounded Knapsack captures the essence of reusable DP states, every step builds on smaller, self-similar subproblems, stacking value piece by piece until the capacity is full.

## 417 Fractional Knapsack

The Fractional Knapsack problem is a close cousin of the 0/1 Knapsack, but with a twist. Here, you can break items into fractions, taking partial weight to maximize total value. This problem is not solved by DP; it's a greedy algorithm, serving as a contrast to show where DP is *not* needed.

## What Problem Are We Solving?

Given:

- $n$  items, each with `weight[i]` and `value[i]`
- a knapsack with capacity  $W$

Find the maximum total value achievable by possibly taking fractions of items.

We define:

- Value density (ratio):

$$\text{ratio}[i] = \frac{\text{value}[i]}{\text{weight}[i]}$$

To maximize value:

1. Sort items by decreasing ratio.
2. Take full items until you can't.
3. Take a fraction of the next one to fill the remaining capacity.

Answer is the sum of selected (full + partial) values.

## How Does It Work (Plain Language)

If each item can be split, the best approach is take the most valuable per unit weight first. It's like filling your bag with gold dust, start with the richest dust, then move to less valuable kinds.

Example: Items:

| Item | Value | Weight | Ratio |
|------|-------|--------|-------|
| 1    | 60    | 10     | 6.0   |
| 2    | 100   | 20     | 5.0   |
| 3    | 120   | 30     | 4.0   |

Capacity = 50

1. Take all of Item 1 → weight 10, value 60
2. Take all of Item 2 → weight 20, value 100
3. Take  $20/30 = 2/3$  of Item 3 → weight 20, value 80

Total =  $60 + 100 + 80 = 240$

## Tiny Code (Easy Versions)

C (Greedy Algorithm)

```
#include <stdio.h>

struct Item {
    int value, weight;
};

int compare(const void *a, const void *b) {
    double r1 = (double)((struct Item *)a)->value / ((struct Item *)a)->weight;
    double r2 = (double)((struct Item *)b)->value / ((struct Item *)b)->weight;
    return (r1 < r2) ? 1 : -1;
}

int main(void) {
    int n, W;
    printf("Enter number of items and capacity: ");
    scanf("%d %d", &n, &W);

    struct Item arr[n];
    printf("Enter value and weight:\n");
    for (int i = 0; i < n; i++)
        scanf("%d %d", &arr[i].value, &arr[i].weight);

    qsort(arr, n, sizeof(struct Item), compare);

    double totalValue = 0.0;
    int curWeight = 0;

    for (int i = 0; i < n; i++) {
        if (curWeight + arr[i].weight <= W) {
            curWeight += arr[i].weight;
            totalValue += arr[i].value;
        } else {
            int remain = W - curWeight;
            totalValue += arr[i].value * ((double)remain / arr[i].weight);
            break;
        }
    }

    printf("Max value: %.2f\n", totalValue);
```

```
    return 0;  
}
```

Python (Greedy Implementation)

```
items = []  
n = int(input("Enter number of items: "))  
W = int(input("Enter capacity: "))  
  
for _ in range(n):  
    v, w = map(int, input("Enter value and weight: ").split())  
    items.append((v, w, v / w))  
  
items.sort(key=lambda x: x[2], reverse=True)  
  
total_value = 0.0  
cur_weight = 0  
  
for v, w, r in items:  
    if cur_weight + w <= W:  
        cur_weight += w  
        total_value += v  
    else:  
        remain = W - cur_weight  
        total_value += v * (remain / w)  
        break  
  
print("Max value:", round(total_value, 2))
```

## Why It Matters

- Demonstrates where DP isn't needed, a greedy choice property
- Contrasts with 0/1 Knapsack (DP needed)
- Builds intuition for ratio-based optimization
- Appears in resource allocation, finance, optimization

The Fractional Knapsack is the “continuous” version, you don’t choose *yes or no*, you pour the best parts until you run out of room.

## Step-by-Step Example

| Item | Value | Weight | Ratio | Take |
|------|-------|--------|-------|------|
| 1    | 60    | 10     | 6.0   | Full |
| 2    | 100   | 20     | 5.0   | Full |
| 3    | 120   | 30     | 4.0   | 2/3  |

Total = 240

## A Gentle Proof (Why It Works)

If all items can be divided arbitrarily, the optimal strategy is always to take the one with the highest value density first. Proof sketch:

1. Suppose an optimal solution skips a higher-ratio item to take a lower-ratio one.
2. Replacing part of the lower-ratio item with the higher-ratio one strictly increases total value.
3. Contradiction, thus, sorting by ratio is optimal.

This property is called the greedy choice property. Because the problem satisfies both optimal substructure and greedy choice, a greedy algorithm suffices.

## Try It Yourself

1. Compare results with 0/1 Knapsack for same items.
2. Add more items with identical ratios.
3. Implement sorting manually and test correctness.
4. Check behavior when capacity < smallest weight.
5. Visualize partial fill using ratio chart.

## Test Cases

| Values             | Weights         | W  | Expected | Notes               |
|--------------------|-----------------|----|----------|---------------------|
| [60,100,120]       | [10,20,30]      | 50 | 240      | classic             |
| [10,5,15,7,6,18,3] | [2,3,5,7,1,4,1] | 15 | 55.33    | greedy mix          |
| [25,50,75]         | [5,10,15]       | 10 | 50       | full item           |
| [5,10,15]          | [1,2,3]         | 3  | 15       | take all            |
| [1,2,3]            | [3,2,1]         | 3  | 5        | highest ratio first |

## Complexity

- Time:  $O(n \log n)$  (for sorting)
- Space:  $O(1)$

The Fractional Knapsack shows the power of greedy reasoning, sometimes, thinking locally optimal truly leads to the global best.

## 418 Coin Change (Min Coins)

The Coin Change (Min Coins) problem is about finding the *fewest number of coins* needed to make a given amount. Unlike the counting version, which sums all combinations, this one focuses on minimization, the shortest path to the target sum.

It's a classic unbounded DP problem, where each coin can be used multiple times.

### What Problem Are We Solving?

Given:

- A list of coins `coins[]`
- A target amount `A`

Find the minimum number of coins needed to make amount `A`. If it's impossible, return `-1`.

We define the state:

$$dp[x] = \text{minimum coins required to make amount } x$$

Recurrence:

$$dp[x] = \min_{c \in \text{coins}; c \leq x} (dp[x - c] + 1)$$

Base:

$$dp[0] = 0$$

Final answer:

$$dp[A]$$

If no combination is possible, `dp[A]` will remain at infinity (or a large sentinel value).

## How Does It Work (Plain Language)

Think of building the amount step by step. For each value  $x$ , try all coins  $c$  that fit, and see which leads to the fewest total coins. Each state  $dp[x]$  represents the shortest chain from 0 to  $x$ .

It's like climbing stairs to a target floor, each coin is a step size, and you want the path with the fewest steps.

Example: `coins = [1, 3, 4], A = 6`

| Amount | dp[x] | Choice |
|--------|-------|--------|
| 0      | 0     | base   |
| 1      | 1     | 1      |
| 2      | 2     | 1+1    |
| 3      | 1     | 3      |
| 4      | 1     | 4      |
| 5      | 2     | 3+2    |
| 6      | 2     | 3+3    |

Answer = 2 (3 + 3)

## Tiny Code (Easy Versions)

C (Bottom-Up DP)

```
#include <stdio.h>
#define INF 1000000
#define MIN(a,b) ((a) < (b) ? (a) : (b))

int main(void) {
    int n, A;
    printf("Enter number of coins and amount: ");
    scanf("%d %d", &n, &A);

    int coins[n];
    printf("Enter coin values: ");
    for (int i = 0; i < n; i++) scanf("%d", &coins[i]);

    int dp[A + 1];
    for (int i = 1; i <= A; i++) dp[i] = INF;
    dp[0] = 0;
```

```

for (int x = 1; x <= A; x++) {
    for (int i = 0; i < n; i++) {
        int c = coins[i];
        if (x - c >= 0)
            dp[x] = MIN(dp[x], dp[x - c] + 1);
    }
}

if (dp[A] == INF) printf("Not possible\n");
else printf("Min coins: %d\n", dp[A]);

return 0;
}

```

Python (Simple Iterative Version)

```

coins = list(map(int, input("Enter coin values: ").split()))
A = int(input("Enter amount: "))

INF = float('inf')
dp = [INF] * (A + 1)
dp[0] = 0

for x in range(1, A + 1):
    for c in coins:
        if x - c >= 0:
            dp[x] = min(dp[x], dp[x - c] + 1)

print("Min coins:" if dp[A] != INF else "Not possible", end=" ")
print(dp[A] if dp[A] != INF else "")

```

## Why It Matters

- Classic unbounded minimization DP
- Core of many resource optimization problems
- Foundation for graph shortest paths, minimum steps, edit operations
- Contrasts with counting version (same recurrence, different aggregation)

This problem shows how min replaces sum in DP to shift from “how many” to “how few”.

## Step-by-Step Example

Coins = [1, 3, 4], A = 6

| x | Choices               | dp[x] |
|---|-----------------------|-------|
| 0 | -                     | 0     |
| 1 | dp[0]+1               | 1     |
| 2 | dp[1]+1               | 2     |
| 3 | dp[0]+1               | 1     |
| 4 | dp[0]+1               | 1     |
| 5 | min(dp[2]+1, dp[1]+1) | 2     |
| 6 | min(dp[3]+1, dp[2]+1) | 2     |

Answer = 2 (3+3)

## A Gentle Proof (Why It Works)

The recurrence builds from smaller amounts upward. For each amount  $x$ , every coin  $c$  offers a path from  $x - c \rightarrow x$ , adding 1 step.

By induction:

- Base case:  $dp[0] = 0$  (no coins to make 0).
- Inductive step: assume optimal solutions exist for all  $x < x$ . Then, the minimal value among all  $dp[x - c] + 1$  is the fewest coins to form  $x$ .

Since each  $x$  reuses optimal subsolutions,  $dp[A]$  is globally optimal.

## Try It Yourself

1. Print the chosen coins (trace back  $dp[x]$ ).
2. Add a coin that never helps (e.g., [1, 3, 10], A=6).
3. Compare with greedy for [1,3,4] (fails).
4. Extend to limited coins (bounded knapsack).
5. Try larger  $A$  to see performance.

## Test Cases

| Coins     | Amount | Expected | Notes        |
|-----------|--------|----------|--------------|
| [1,3,4]   | 6      | 2        | 3+3          |
| [2,5]     | 3      | -1       | not possible |
| [1,2,5]   | 11     | 3        | 5+5+1        |
| [9,6,5,1] | 11     | 2        | 6+5          |
| [2,3,7]   | 12     | 3        | 3+3+6        |

## Complexity

- Time:  $O(n \times A)$
- Space:  $O(A)$

The Coin Change (Min Coins) problem is where unbounded DP meets optimization, building minimal paths to a target through simple, repeated decisions.

## 419 Coin Change (Count Ways)

The Coin Change (Count Ways) problem is about how many different ways you can make a given amount using available coins. Unlike the minimization version, here every combination matters, order doesn't.

This is a perfect example of unbounded combinatorial DP, where each coin can be used multiple times, but arrangement order is irrelevant.

### What Problem Are We Solving?

Given:

- A list of coins `coins []`
- A target amount `A`

Find the number of distinct combinations (unordered) that sum to `A`.

We define the state:

$$dp[x] = \text{number of ways to make amount } x$$

Recurrence:

$$dp[x] = \sum_{c \in \text{coins},; c \leq x} dp[x - c]$$

Base:

$$dp[0] = 1$$

To avoid counting the same combination multiple times (e.g., [1, 2] and [2, 1]), we iterate coins first, then amount.

### How Does It Work (Plain Language)

We count *combinations*, not *permutations*. That means {1, 2} and {2, 1} are considered the same way. So we fix each coin's order, when processing a coin, we allow it to be reused, but not reordered with future coins.

Example: coins = [1, 2, 5], A = 5

Ways:

- 1+1+1+1+1
- 1+1+1+2
- 1+2+2
- 5

Answer = 4

### Tiny Code (Easy Versions)

C (Bottom-Up DP)

```
#include <stdio.h>

int main(void) {
    int n, A;
    printf("Enter number of coins and amount: ");
    scanf("%d %d", &n, &A);

    int coins[n];
    printf("Enter coin values: ");
    for (int i = 0; i < n; i++) scanf("%d", &coins[i]);

    long long dp[A + 1];
    for (int i = 0; i <= A; i++) dp[i] = 0;
    dp[0] = 1;
```

```

for (int i = 0; i < n; i++) {
    for (int x = coins[i]; x <= A; x++) {
        dp[x] += dp[x - coins[i]];
    }
}

printf("Number of ways: %lld\n", dp[A]);
return 0;
}

```

Python (Iterative Combinations)

```

coins = list(map(int, input("Enter coin values: ").split()))
A = int(input("Enter amount: "))

dp = [0] * (A + 1)
dp[0] = 1

for c in coins:
    for x in range(c, A + 1):
        dp[x] += dp[x - c]

print("Number of ways:", dp[A])

```

## Why It Matters

- Foundation of combinatorial DP
- Basis for partition counting, compositional sums, and probability DP
- Reinforces loop ordering importance, changing order counts permutations instead
- Connects to integer partition problems in number theory

It teaches you that *what you count* (order vs combination) depends on how you iterate.

## Step-by-Step Example

Coins = [1, 2, 5], A = 5

Initialize dp = [1, 0, 0, 0, 0, 0]

| Coin | State               | dp array (after processing) |
|------|---------------------|-----------------------------|
| 1    | all                 | [1, 1, 1, 1, 1, 1]          |
| 2    | adds combos using 2 | [1, 1, 2, 2, 3, 3]          |
| 5    | adds combos using 5 | [1, 1, 2, 2, 3, 4]          |

Answer = 4

### A Gentle Proof (Why It Works)

We fill  $\text{dp}[x]$  by summing contributions from each coin  $c$ : every time we use coin  $c$ , we move from subproblem  $x - c \rightarrow x$ .

$$\text{dp}[x] = \sum_{c \in \text{coins}} \text{dp}[x - c]$$

But we must fix coin iteration order to ensure unique combinations. Iterating coins first ensures each combination is formed in a canonical order:

- 1 before 2 before 5 So {1,2} appears once, not twice.

By induction:

- Base:  $\text{dp}[0] = 1$  (one way: use nothing)
- Step: each  $\text{dp}[x]$  counts valid combos by extending smaller sums.

### Try It Yourself

1. Swap loop order  $\rightarrow$  count permutations.
2. Add coin 3 and compare growth.
3. Print all combinations via recursion.
4. Add modulo  $10^9 + 7$  for large results.
5. Compare with Min Coins (same coins, different goal).

| Coins | Amount | Expected | Notes |
|-------|--------|----------|-------|
|-------|--------|----------|-------|

## Test Cases

| Coins     | Amount | Expected | Notes                            |
|-----------|--------|----------|----------------------------------|
| [1,2,5]   | 5      | 4        | classic                          |
| [2,3,5,6] | 10     | 5        | multiple combos                  |
| [1]       | 3      | 1        | only one way                     |
| [2]       | 3      | 0        | impossible                       |
| [1,2,3]   | 4      | 4        | (1+1+1+1), (1+1+2), (2+2), (1+3) |

## Complexity

- Time:  $O(n \times A)$
- Space:  $O(A)$

The Coin Change (Count Ways) problem captures the *combinatorial heart* of DP, a single recurrence, but with the magic of order-aware iteration, turns counting from chaos into clarity.

## 420 Multi-Dimensional Knapsack

The Multi-Dimensional Knapsack problem (also called the Multi-Constraint Knapsack) extends the classic 0/1 Knapsack into a richer, more realistic world. Here, each item consumes multiple types of resources (weight, volume, cost, etc.), and we must respect all constraints simultaneously.

It's where the simplicity of one dimension gives way to the complexity of many.

### What Problem Are We Solving?

Given:

- $n$  items
- Each item  $i$  has:
  - value  $v[i]$
  - weights in  $m$  dimensions  $w[i][1..m]$
- Capacities  $C[1..m]$  for each dimension

Select a subset of items maximizing total value, subject to:

$$\forall j \in [1, m] : \sum_{i \in S} w[i][j] \leq C[j]$$

State definition:

$$dp[c_1][c_2] \dots [c_m] = \text{maximum value achievable with capacities } (c_1, \dots, c_m)$$

Recurrence:

$$dp[\vec{c}] = \max (dp[\vec{c}], ; dp[\vec{c} - \vec{w}_i] + v[i])$$

where  $\vec{c} - \vec{w}_i$  means subtracting all weights component-wise.

### How Does It Work (Plain Language)

It's like packing a spaceship with multiple limits, weight, volume, fuel usage, and every item drains each dimension differently. You can't just fill to a single limit; every item's cost affects all dimensions at once.

The DP grid is now multi-dimensional: you must iterate over every combination of capacities and decide to include or exclude each item.

Example (2D case): Items:

| Item | Value | Weight | Volume |
|------|-------|--------|--------|
| 1    | 60    | 2      | 3      |
| 2    | 100   | 3      | 4      |
| 3    | 120   | 4      | 5      |

Capacity: (W=5, V=7)

Answer: 160 (Items 1 + 2)

### Tiny Code (2D DP Example)

C (2D Capacity, 0/1 Version)

```

#include <stdio.h>
#define MAX(a,b) ((a) > (b) ? (a) : (b))

int main(void) {
    int n, W, V;
    printf("Enter number of items, weight cap, volume cap: ");
    scanf("%d %d %d", &n, &W, &V);

    int w[n], vol[n], val[n];
    printf("Enter weight, volume, value:\n");
    for (int i = 0; i < n; i++)
        scanf("%d %d %d", &w[i], &vol[i], &val[i]);

    int dp[W + 1][V + 1];
    for (int i = 0; i <= W; i++)
        for (int j = 0; j <= V; j++)
            dp[i][j] = 0;

    for (int i = 0; i < n; i++) {
        for (int wi = W; wi >= w[i]; wi--) {
            for (int vi = V; vi >= vol[i]; vi--) {
                dp[wi][vi] = MAX(dp[wi][vi],
                                   dp[wi - w[i]][vi - vol[i]] + val[i]);
            }
        }
    }

    printf("Max value: %d\n", dp[W][V]);
    return 0;
}

```

Python (2D DP)

```

items = [(2,3,60), (3,4,100), (4,5,120)]
W, V = 5, 7

dp = [[0]*(V+1) for _ in range(W+1)]

for w, v, val in items:
    for wi in range(W, w-1, -1):
        for vi in range(V, v-1, -1):
            dp[wi][vi] = max(dp[wi][vi], dp[wi-w][vi-v] + val)

```

```
print("Max value:", dp[W][V])
```

## Why It Matters

- Models real-world constraints, multiple resources
- Core of operations research, resource allocation, logistics, multi-resource scheduling
- Illustrates how DP dimensionality grows with complexity
- Forces careful state design and iteration order

When one dimension is not enough, this generalization captures tradeoffs across many.

## Step-by-Step Example (2D)

Capacity: W=5, V=7 Items:

1. (2,3,60)
2. (3,4,100)
3. (4,5,120)

We explore subsets:

- $\{1\} \rightarrow (2,3)$ , value=60
- $\{2\} \rightarrow (3,4)$ , value=100
- $\{3\} \rightarrow (4,5)$ , value=120
- $\{1,2\} \rightarrow (5,7)$ , value=160 optimal
- $\{1,3\} \rightarrow (6,8)$  exceeds
- $\{2,3\} \rightarrow (7,9)$  exceeds

Answer = 160

## A Gentle Proof (Why It Works)

By induction on item index and capacities:

Let  $dp[i][c_1][c_2] \dots [c_m]$  be the best value using first  $i$  items and capacity vector  $(c_1, c_2, \dots, c_m)$ .

Two choices for each item:

1. Exclude  $\rightarrow$  keep  $dp[i - 1][\vec{c}]$
2. Include  $\rightarrow dp[i - 1][\vec{c} - \vec{w}_i] + v[i]$  (if feasible)

Take the max.

Since all transitions only depend on smaller capacities, and each subproblem is optimal, overall DP converges to global optimum.

### Try It Yourself

1. Add third dimension (e.g., “time”).
2. Compare with greedy (fails).
3. Visualize DP table for 2D.
4. Track chosen items with traceback.
5. Add unbounded variation (re-use items).

### Test Cases

| Items (W,V,Val)                | Capacity (W,V) | Expected | Notes       |
|--------------------------------|----------------|----------|-------------|
| [(2,3,60),(3,4,100),(4,5,120)] | (5,7)          | 160      | (1+2)       |
| [(1,2,10),(2,3,20),(3,3,40)]   | (3,4)          | 40       | single best |
| [(2,2,8),(2,3,9),(3,4,14)]     | (4,5)          | 17       | (1+2)       |
| [(3,2,10),(2,4,12),(4,3,18)]   | (5,6)          | 22       | (1+2)       |

### Complexity

- Time:  $O(n \times C_1 \times C_2 \times \dots \times C_m)$
- Space:  $O(C_1 \times C_2 \times \dots \times C_m)$

The Multi-Dimensional Knapsack is a reminder that every extra resource adds a new axis to your reasoning, and your DP table.

# Section 43. Sequence Problems

## 421 Longest Increasing Subsequence ( $O(n^2)$ DP)

The Longest Increasing Subsequence (LIS) problem asks for the maximum length of a subsequence that is strictly increasing. Elements do not need to be contiguous, only in order.

### What Problem Are We Solving?

Given an array  $\text{arr}[0..n-1]$ , find the maximum  $k$  such that there exist indices  $0 \leq i_1 < i_2 < \dots < i_k \leq n$  with

$$\text{arr}[i_1] < \text{arr}[i_2] < \dots < \text{arr}[i_k].$$

Define the state

$$dp[i] = \text{length of the LIS that ends at index } i.$$

Recurrence

$$dp[i] = 1 + \max_{0 \leq j < i, \text{arr}[j] < \text{arr}[i]} dp[j], \quad \text{with } dp[i] \leftarrow 1 \text{ if no such } j.$$

Answer

$$\max_{0 \leq i < n} dp[i].$$

### How Does It Work (Plain Language)

For each position  $i$ , look back at all earlier positions  $j < i$  with a smaller value. Any increasing subsequence ending at  $j$  can be extended by  $\text{arr}[i]$ . Pick the best among them and add one. If nothing is smaller, start a new subsequence of length 1 at  $i$ .

Example:  $\text{arr} = [10, 22, 9, 33, 21, 50, 41, 60]$

| i | arr[i] | dp[i] | explanation         |
|---|--------|-------|---------------------|
| 0 | 10     | 1     | start               |
| 1 | 22     | 2     | $10 \rightarrow 22$ |
| 2 | 9      | 1     | restart at 9        |

| i | arr[i] | dp[i] | explanation                                                      |
|---|--------|-------|------------------------------------------------------------------|
| 3 | 33     | 3     | $10 \rightarrow 22 \rightarrow 33$                               |
| 4 | 21     | 2     | $10 \rightarrow 21$                                              |
| 5 | 50     | 4     | $10 \rightarrow 22 \rightarrow 33 \rightarrow 50$                |
| 6 | 41     | 4     | $10 \rightarrow 22 \rightarrow 33 \rightarrow 41$                |
| 7 | 60     | 5     | $10 \rightarrow 22 \rightarrow 33 \rightarrow 50 \rightarrow 60$ |

Answer is 5.

### Tiny Code (Easy Versions)

C ( $O(n^2)$ )

```
#include <stdio.h>
#define MAX(a,b) ((a)>(b)?(a):(b))

int main(void) {
    int n;
    printf("Enter n: ");
    scanf("%d", &n);
    int arr[n];
    printf("Enter array: ");
    for (int i = 0; i < n; i++) scanf("%d", &arr[i]);

    int dp[n];
    for (int i = 0; i < n; i++) dp[i] = 1;

    int ans = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (arr[j] < arr[i]) dp[i] = MAX(dp[i], dp[j] + 1);
        }
        ans = MAX(ans, dp[i]);
    }
    printf("LIS length: %d\n", ans);
    return 0;
}
```

Python ( $O(n^2)$ )

```

arr = list(map(int, input("Enter array: ").split()))
n = len(arr)
dp = [1] * n
for i in range(n):
    for j in range(i):
        if arr[j] < arr[i]:
            dp[i] = max(dp[i], dp[j] + 1)
print("LIS length:", max(dp) if dp else 0)

```

## Why It Matters

- Archetypal sequence DP: define state on prefixes and extend with a choice.
- Foundation for LCS, Edit Distance, patience sorting LIS in  $O(n \log n)$ , and 2D variants like Russian Doll Envelopes.
- Useful in ranking, time series smoothing, and chain scheduling.

## Step by Step Example

```
arr = [3, 10, 2, 1, 20]
```

| i | arr[i] | candidates ( $dp[j]+1$ ) with ( $arr[j] < arr[i]$ )  | $dp[i]$ |
|---|--------|------------------------------------------------------|---------|
| 0 | 3      | ,                                                    | 1       |
| 1 | 10     | $dp[0]+1 = 2$                                        | 2       |
| 2 | 2      | ,                                                    | 1       |
| 3 | 1      | ,                                                    | 1       |
| 4 | 20     | $\max( dp[0]+1=2, dp[1]+1=3, dp[2]+1=2, dp[3]+1=2 )$ | 3       |

Answer is 3.

## A Gentle Proof (Why It Works)

Let  $OPT(i)$  denote the LIS length that ends exactly at index  $i$ . Any LIS ending at  $i$  must either be just  $[arr[i]]$  of length 1, or extend a strictly smaller element at some  $j < i$ . Therefore

$$OPT(i) = \max \left( 1, ; 1 + \max_{j < i, arr[j] < arr[i]} OPT(j) \right).$$

This depends only on optimal values from smaller indices, hence dynamic programming applies. The overall LIS is the maximum over all end positions:

$$\text{LIS} = \max_i OPT(i).$$

By induction on  $i$ , the recurrence computes  $\text{OPT}(i)$  correctly, so the final maximum is optimal.

### Try It Yourself

1. Recover an actual LIS: keep a `parent[i]` pointing to the  $j$  that gave the best transition.
2. Change to nondecreasing subsequence: replace  $<$  with  $\leq$ .
3. Compare with the  $O(n \log n)$  patience sorting method and verify both lengths match.
4. Compute the number of LIS of maximum length using a parallel `cnt[i]`.
5. Extend to 2D pairs  $(a, b)$  by sorting on  $a$  and running LIS on  $b$  with careful tie handling.

### Test Cases

| arr                      | expected |
|--------------------------|----------|
| [1,2,3,4,5]              | 5        |
| [5,4,3,2,1]              | 1        |
| [3,10,2,1,20]            | 3        |
| [10,22,9,33,21,50,41,60] | 5        |
| [2,2,2,2]                | 1        |

### Complexity

- Time:  $(O(n^2))$
- Space:  $(O(n))$

This  $O(n^2)$  DP is the clearest path to LIS: define the end, look back to smaller ends, and grow the longest chain.

### 422 LIS (Patience Sorting) – $O(n \log n)$ Optimized

The Longest Increasing Subsequence (LIS) can be solved faster than the classic  $(O(n^2))$  DP by using a clever idea inspired by patience sorting. Instead of building all sequences, we maintain a minimal tail array, each element represents the smallest possible tail of an increasing subsequence of a given length.

## What Problem Are We Solving?

Given an array  $\text{arr}[0..n-1]$ , find the length of the longest strictly increasing subsequence in  $O(n \log n)$  time.

We want

$$\text{LIS length} = \max k \text{ such that } \exists i_1 < i_2 < \dots < i_k, ; \text{arr}[i_1] < \text{arr}[i_2] < \dots < \text{arr}[i_k].$$

## Key Idea

Maintain an array  $\text{tails}[]$  where  $\text{tails}[\text{len}]$  = smallest tail value of any increasing subsequence of length  $\text{len}+1$ .

For each element  $x$  in the array:

1. Use binary search in  $\text{tails}$  to find the first position  $\text{pos}$  with  $\text{tails}[\text{pos}] \geq x$ .
2. Replace  $\text{tails}[\text{pos}]$  with  $x$  (we found a better tail).
3. If  $x$  is larger than all tails, append it, subsequence grows.

At the end,  $\text{len}(\text{tails})$  = LIS length.

## How Does It Work (Plain Language)

Think of placing numbers onto piles (like patience solitaire):

- Each pile's top is the smallest possible number ending an increasing subsequence of that length.
- When a new number comes, place it on the leftmost pile whose top is  $\geq$  the number.
- If none exists, start a new pile.

The number of piles equals the LIS length.

Example:  $\text{arr} = [10, 22, 9, 33, 21, 50, 41, 60]$

Process:

| x  | tails (after processing x) |
|----|----------------------------|
| 10 | [10]                       |
| 22 | [10, 22]                   |
| 9  | [9, 22]                    |
| 33 | [9, 22, 33]                |
| 21 | [9, 21, 33]                |
| 50 | [9, 21, 33, 50]            |

| x  | tails (after processing x) |
|----|----------------------------|
| 41 | [9, 21, 33, 41]            |
| 60 | [9, 21, 33, 41, 60]        |

Answer = 5

### Tiny Code (Easy Versions)

C (Using Binary Search)

```
#include <stdio.h>

int lower_bound(int arr[], int len, int x) {
    int l = 0, r = len;
    while (l < r) {
        int mid = (l + r) / 2;
        if (arr[mid] < x) l = mid + 1;
        else r = mid;
    }
    return l;
}

int main(void) {
    int n;
    printf("Enter n: ");
    scanf("%d", &n);

    int a[n];
    printf("Enter array: ");
    for (int i = 0; i < n; i++) scanf("%d", &a[i]);

    int tails[n], len = 0;

    for (int i = 0; i < n; i++) {
        int pos = lower_bound(tails, len, a[i]);
        tails[pos] = a[i];
        if (pos == len) len++;
    }

    printf("LIS length: %d\n", len);
}
```

```

    return 0;
}

```

Python (Using bisect)

```

import bisect

arr = list(map(int, input("Enter array: ").split()))
tails = []

for x in arr:
    pos = bisect.bisect_left(tails, x)
    if pos == len(tails):
        tails.append(x)
    else:
        tails[pos] = x

print("LIS length:", len(tails))

```

## Why It Matters

- Reduces LIS from  $O(n^2)$  to  $O(n \log n)$
- Introduces binary search in DP transitions
- Demonstrates state compression, we track only *tails*, not all subproblems
- Serves as basis for LIS reconstruction, LDS, Longest Bitonic Subsequence, and 2D LIS

This technique shows how mathematical insight can collapse a DP table into a minimal structure.

## Step-by-Step Example

```
arr = [3, 10, 2, 1, 20]
```

| x  | tails     |
|----|-----------|
| 3  | [3]       |
| 10 | [3,10]    |
| 2  | [2,10]    |
| 1  | [1,10]    |
| 20 | [1,10,20] |

Answer = 3

## A Gentle Proof (Why It Works)

Invariant:

- `tails[k]` = minimal possible tail of any increasing subsequence of length  $k+1$ .

When we place  $x$ :

- Replacing a tail keeps subsequences valid (shorter or equal tail  $\rightarrow$  more chance to extend).
- Appending  $x$  grows the length by one.

By induction:

- Each `tails[k]` is nondecreasing with length.
- Final size of `tails` equals the LIS length, because every pile represents a distinct subsequence length.

## Try It Yourself

1. Track predecessors to reconstruct one LIS.
2. Modify to nondecreasing subsequence with `bisect_right`.
3. Compare counts with  $(O(n^2))$  version.
4. Visualize piles after each insertion.
5. Use on 2D sorted pairs  $(a, b)$  for envelope problems.

## Test Cases

| arr                      | Expected | Notes              |
|--------------------------|----------|--------------------|
| [10,22,9,33,21,50,41,60] | 5        | classic            |
| [3,10,2,1,20]            | 3        | {3,10,20}          |
| [1,2,3,4,5]              | 5        | already increasing |
| [5,4,3,2,1]              | 1        | decreasing         |
| [2,2,2,2]                | 1        | constant           |

## Complexity

- Time:  $O(n \log n)$  (binary search per element)
- Space:  $O(n)$

The Patience Sorting LIS turns a quadratic DP into a sleek logarithmic method, a masterclass in trading space for insight.

## 423 Longest Common Subsequence (LCS)

The Longest Common Subsequence (LCS) problem finds the longest sequence that appears in the same relative order (not necessarily contiguous) in both strings. It is one of the most fundamental two-sequence DPs, and the basis of algorithms like diff, edit distance, and DNA alignment.

### What Problem Are We Solving?

Given two sequences

$$X = x_1, x_2, \dots, x_m, \quad Y = y_1, y_2, \dots, y_n$$

find the length of the longest sequence that is a subsequence of both.

Define the state:

$$dp[i][j] = \text{LCS length of prefixes } X[0..i-1], Y[0..j-1]$$

Recurrence:

$$dp[i][j] = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0, \\ dp[i-1][j-1] + 1, & \text{if } x_{i-1} = y_{j-1}, \\ \max(dp[i-1][j], dp[i][j-1]), & \text{if } x_{i-1} \neq y_{j-1}. \end{cases}$$

Answer:

$$dp[m][n]$$

### How Does It Work (Plain Language)

We build a grid where each cell  $dp[i][j]$  represents the LCS of the first  $i$  characters of  $X$  and the first  $j$  characters of  $Y$ .

- If the characters match, extend the subsequence diagonally.
- If not, skip one character (either from  $X$  or  $Y$ ) and take the better result.

Think of it as aligning the two strings, step by step, and keeping the longest matching order.

Example:  $X = \text{"ABCBDAB"}$ ,  $Y = \text{"BDCAB"}$

The longest common subsequence is "BCAB", length 4.

## Tiny Code (Easy Versions)

C (Classic 2D DP)

```
#include <stdio.h>
#define MAX(a,b) ((a) > (b) ? (a) : (b))

int main(void) {
    char X[100], Y[100];
    printf("Enter first string: ");
    scanf("%s", X);
    printf("Enter second string: ");
    scanf("%s", Y);

    int m = 0, n = 0;
    while (X[m]) m++;
    while (Y[n]) n++;

    int dp[m + 1][n + 1];
    for (int i = 0; i <= m; i++)
        for (int j = 0; j <= n; j++)
            dp[i][j] = 0;

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (X[i - 1] == Y[j - 1])
                dp[i][j] = dp[i - 1][j - 1] + 1;
            else
                dp[i][j] = MAX(dp[i - 1][j], dp[i][j - 1]);
        }
    }

    printf("LCS length: %d\n", dp[m][n]);
    return 0;
}
```

Python (2D DP)

```
X = input("Enter first string: ")
Y = input("Enter second string: ")

m, n = len(X), len(Y)
```

```

dp = [[0]*(n+1) for _ in range(m+1)]

for i in range(1, m+1):
    for j in range(1, n+1):
        if X[i-1] == Y[j-1]:
            dp[i][j] = dp[i-1][j-1] + 1
        else:
            dp[i][j] = max(dp[i-1][j], dp[i][j-1])

print("LCS length:", dp[m][n])

```

## Why It Matters

- Classic two-dimensional DP template
- Core for Edit Distance, Sequence Alignment, Diff tools
- Demonstrates subproblem reuse via overlapping prefixes
- Helps understand table-filling and backtracking reconstruction

LCS is where dynamic programming meets string similarity.

## Step-by-Step Example

X = "ABCBDAB", Y = "BDCAB"

| i | j | X[i-1], Y[j-1] | dp[i][j] | Explanation      |
|---|---|----------------|----------|------------------|
| 1 | 1 | A, B           | 0        | mismatch         |
| 2 | 1 | B, B           | 1        | match            |
| 3 | 2 | C, D           | 1        | carry max        |
| 4 | 3 | B, C           | 1        | carry max        |
| 5 | 4 | D, A           | 2        | later match      |
| 7 | 5 | B, B           | 4        | full subsequence |

Answer = 4 ("BCAB")

## A Gentle Proof (Why It Works)

By induction on *i* and *j*:

- Base:  $dp[0][j] = dp[i][0] = 0$  (empty prefix)

- If  $x_{i-1} = y_{j-1}$ , every common subsequence of  $X[0..i-2]$  and  $Y[0..j-2]$  can be extended by this match.
- If not equal, longest subsequence must exclude one character, hence `max` of left and top cells.

Since each subproblem depends only on smaller prefixes, filling the table row by row ensures all dependencies are ready.

### Try It Yourself

1. Reconstruct the actual LCS (store direction or traceback).
2. Modify to handle case-insensitive matches.
3. Compare with Edit Distance formula.
4. Visualize table diagonal matches.
5. Use it to find diff between two lines of text.

### Test Cases

| X         | Y         | Expected | Notes     |
|-----------|-----------|----------|-----------|
| “ABCBDAB” | “BDCAB”   | 4        | “BCAB”    |
| “AGGTAB”  | “GXTXAYB” | 4        | “GTAB”    |
| “AAAA”    | “AA”      | 2        | subset    |
| “ABC”     | “DEF”     | 0        | none      |
| “ ”       | “ABC”     | 0        | base case |

### Complexity

- Time:  $O(m \times n)$
- Space:  $O(m \times n)$ , or  $O(\min(m, n))$  with rolling arrays

The Longest Common Subsequence teaches you to align two worlds, character by character, building similarity from shared order, not proximity.

### 424 Edit Distance (Levenshtein)

The Edit Distance (or Levenshtein Distance) problem measures how *different* two strings are by counting the minimum number of operations needed to transform one into the other. The allowed operations are usually insert, delete, and replace.

This is one of the most elegant two-dimensional DPs, it captures transformation cost between sequences step by step.

### **What Problem Are We Solving?**

Given two strings

$$X = x_1, x_2, \dots, x_m, \quad Y = y_1, y_2, \dots, y_n$$

find the minimum number of operations to convert X into Y, using:

- Insert a character
- Delete a character
- Replace a character

We define the state:

$$dp[i][j] = \text{minimum edits to convert } X[0..i-1] \text{ into } Y[0..j-1]$$

Recurrence:

$$dp[i][j] = \begin{cases} i, & \text{if } j = 0, \\ j, & \text{if } i = 0, \\ dp[i-1][j-1], & \text{if } x_{i-1} = y_{j-1}, \\ 1 + \min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]), & \text{if } x_{i-1} \neq y_{j-1}. \end{cases}$$

- $dp[i-1][j] + 1 \rightarrow \text{delete}$
- $dp[i][j-1] + 1 \rightarrow \text{insert}$
- $dp[i-1][j-1] + 1 \rightarrow \text{replace}$

Answer:

$$dp[m][n]$$

## How Does It Work (Plain Language)

We build a 2D grid comparing prefixes of both strings.

Each cell answers: “*What’s the cheapest way to make  $X[:i]$  look like  $Y[:j]$ ?*”

- If characters match, carry over the diagonal value.
- If they differ, take the smallest cost among inserting, deleting, or replacing.

Think of typing corrections: every operation moves you closer to the target.

Example:  $X = \text{"kitten"}$ ,  $Y = \text{"sitting"}$  Operations:

- Replace k → s
- Replace e → i
- Insert g

Answer = 3

## Tiny Code (Easy Versions)

C (2D DP Table)

```
#include <stdio.h>
#define MIN(a,b) ((a)<(b)?(a):(b))

int min3(int a, int b, int c) {
    int m = (a < b) ? a : b;
    return (m < c) ? m : c;
}

int main(void) {
    char X[100], Y[100];
    printf("Enter first string: ");
    scanf("%s", X);
    printf("Enter second string: ");
    scanf("%s", Y);

    int m = 0, n = 0;
    while (X[m]) m++;
    while (Y[n]) n++;

    int dp[m + 1][n + 1];
```

```

for (int i = 0; i <= m; i++) dp[i][0] = i;
for (int j = 0; j <= n; j++) dp[0][j] = j;

for (int i = 1; i <= m; i++) {
    for (int j = 1; j <= n; j++) {
        if (X[i-1] == Y[j-1])
            dp[i][j] = dp[i-1][j-1];
        else
            dp[i][j] = 1 + min3(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]);
    }
}

printf("Edit distance: %d\n", dp[m][n]);
return 0;
}

```

Python (Compact Version)

```

X = input("Enter first string: ")
Y = input("Enter second string: ")

m, n = len(X), len(Y)
dp = [[0]*(n+1) for _ in range(m+1)]

for i in range(m+1):
    dp[i][0] = i
for j in range(n+1):
    dp[0][j] = j

for i in range(1, m+1):
    for j in range(1, n+1):
        if X[i-1] == Y[j-1]:
            dp[i][j] = dp[i-1][j-1]
        else:
            dp[i][j] = 1 + min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1])

print("Edit distance:", dp[m][n])

```

## Why It Matters

- Foundation for spell checking, DNA alignment, fuzzy matching, and diff tools

- Demonstrates multi-option recurrence (3 choices per state)
- Basis for weighted edit distances (cost per operation)
- Shows how to encode sequence transformation into DP

It's one of the cleanest examples where DP reveals the shortest transformation path.

### Step-by-Step Example

$X = \text{"kitten"}, Y = \text{"sitting"}$

| i | j | $X[:i]$  | $Y[:j]$   | $dp[i][j]$ | Explanation |
|---|---|----------|-----------|------------|-------------|
| 0 | 0 | " "      | " "       | 0          | base        |
| 1 | 1 | "k"      | "s"       | 1          | replace     |
| 2 | 2 | "ki"     | "si"      | 1          | carry       |
| 3 | 3 | "kit"    | "sit"     | 1          | carry       |
| 4 | 4 | "kitt"   | "sitt"    | 1          | carry       |
| 5 | 5 | "kitte"  | "sitti"   | 2          | replace     |
| 6 | 6 | "kitten" | "sittin"  | 2          | replace     |
| 6 | 7 | "kitten" | "sitting" | 3          | insert      |

Answer = 3

### A Gentle Proof (Why It Works)

For each prefix pair  $(i, j)$ :

- If last chars match: no new cost, inherit  $dp[i-1][j-1]$ .
- Else:
  - Delete  $X[i-1] \rightarrow dp[i-1][j] + 1$
  - Insert  $Y[j-1] \rightarrow dp[i][j-1] + 1$
  - Replace  $X[i-1]$  with  $Y[j-1] \rightarrow dp[i-1][j-1] + 1$

We choose the minimal option. By induction on  $(i, j)$ , every  $dp[i][j]$  is optimal, since it uses optimal subsolutions from smaller prefixes.

## Try It Yourself

1. Print the sequence of operations (traceback from  $dp[m][n]$ ).
2. Change costs: make replace = 2, others = 1.
3. Compare with LCS:  $EditDistance = m + n - 2 \times LCS$ .
4. Handle insert/delete only (turn it into LCS variant).
5. Try with words like "intention" → "execution".

## Test Cases

| X        | Y         | Expected | Notes                    |
|----------|-----------|----------|--------------------------|
| "kitten" | "sitting" | 3        | replace, replace, insert |
| "flaw"   | "lawn"    | 2        | replace, insert          |
| "abc"    | "abc"     | 0        | same                     |
| "abc"    | "yabd"    | 2        | replace, insert          |
| " "      | "abc"     | 3        | inserts                  |

## Complexity

- Time:  $O(m \times n)$
- Space:  $O(m \times n)$ , or  $O(\min(m, n))$  with rolling rows

The Edit Distance captures the very essence of transformation, how to reshape one structure into another, one careful operation at a time.

## 425 Longest Palindromic Subsequence

The Longest Palindromic Subsequence (LPS) problem finds the longest sequence that reads the same forward and backward, not necessarily contiguous. It's a classic two-dimensional DP, and a mirror image of the Longest Common Subsequence (LCS), but here, we compare a string with its reverse.

### What Problem Are We Solving?

Given a string

$$S = s_1, s_2, \dots, s_n$$

find the length of the longest subsequence that is a palindrome.

Define the state:

$$dp[i][j] = \text{LPS length in substring } S[i..j]$$

Recurrence:

$$dp[i][j] = \begin{cases} 1, & \text{if } i = j, \\ 2 + dp[i+1][j-1], & \text{if } s_i = s_j, \\ \max(dp[i+1][j], dp[i][j-1]), & \text{if } s_i \neq s_j. \end{cases}$$

Base case:

$$dp[i][i] = 1$$

Final answer:

$$dp[0][n-1]$$

### How Does It Work (Plain Language)

We expand outward between two indices *i* and *j*:

- If the characters match, they can wrap a smaller palindrome inside.
- If not, skip one character (either start or end) and try again.

Think of it as *folding the string onto itself*, one matching pair at a time.

Example: *S* = "bbbab"

LPS = "bbbb" (length 4)

### Tiny Code (Easy Versions)

C (Bottom-Up 2D DP)

```
#include <stdio.h>
#define MAX(a,b) ((a) > (b) ? (a) : (b))

int main(void) {
    char S[100];
    printf("Enter string: ");
    scanf("%s", S);

    int n = 0;
    while (S[n]) n++;
}
```

```

int dp[n][n];
for (int i = 0; i < n; i++) dp[i][i] = 1;

for (int len = 2; len <= n; len++) {
    for (int i = 0; i <= n - len; i++) {
        int j = i + len - 1;
        if (S[i] == S[j] && len == 2)
            dp[i][j] = 2;
        else if (S[i] == S[j])
            dp[i][j] = 2 + dp[i+1][j-1];
        else
            dp[i][j] = MAX(dp[i+1][j], dp[i][j-1]);
    }
}

printf("LPS length: %d\n", dp[0][n-1]);
return 0;
}

```

Python (Clean DP Version)

```

S = input("Enter string: ")
n = len(S)
dp = [[0]*n for _ in range(n)]

for i in range(n):
    dp[i][i] = 1

for length in range(2, n+1):
    for i in range(n - length + 1):
        j = i + length - 1
        if S[i] == S[j]:
            dp[i][j] = 2 + (dp[i+1][j-1] if length > 2 else 0)
        else:
            dp[i][j] = max(dp[i+1][j], dp[i][j-1])

print("LPS length:", dp[0][n-1])

```

## Why It Matters

- Core DP on substrings (interval DP)

- Connects to LCS: [  $LPS(S) = LCS(S, \text{reverse}(S))$ ]
- Foundation for Palindrome Partitioning, String Reconstruction, and DNA symmetry problems
- Teaches two-pointer DP intuition

The LPS shows how symmetry and substructure intertwine.

### Step-by-Step Example

$S = "bbbab"$

| i | j | $S[i..j]$ | $dp[i][j]$ | Reason             |
|---|---|-----------|------------|--------------------|
| 0 | 0 | b         | 1          | single char        |
| 1 | 1 | b         | 1          | single char        |
| 2 | 2 | b         | 1          | single char        |
| 3 | 3 | a         | 1          | single char        |
| 4 | 4 | b         | 1          | single char        |
| 2 | 4 | "bab"     | 3          | $b + a + b$        |
| 1 | 4 | "bbab"    | 3          | wrap b's           |
| 0 | 4 | "bbbab"   | 4          | $b + (bb a b) + b$ |

Answer = 4 ("bbbb")

### A Gentle Proof (Why It Works)

For substring  $S[i..j]$ :

- If  $s_i == s_j$ : both ends can contribute to a longer palindrome, add 2 around  $dp[i+1][j-1]$ .
- If  $s_i != s_j$ : one of them can't be in the palindrome, skip either i or j and take max.

By filling increasing substring lengths, every subproblem is solved before it's needed.

### Try It Yourself

1. Reconstruct one longest palindrome using backtracking.
2. Compare with  $LCS(S, \text{reverse}(S))$  result.
3. Try on "cbbd", "agbdba".
4. Modify to count number of distinct palindromic subsequences.
5. Visualize table diagonals (bottom-up growth).

## Test Cases

| S        | Expected | Notes           |
|----------|----------|-----------------|
| “bbbab”  | 4        | “bbbb”          |
| “cbbd”   | 2        | “bb”            |
| “agbdba” | 5        | “abdba”         |
| “abcd”   | 1        | any single char |
| “aaa”    | 3        | whole string    |

## Complexity

- Time: ( $O(n^2)$ )
- Space: ( $O(n^2)$ ), reducible with rolling arrays

The Longest Palindromic Subsequence is a mirror held up to your string, revealing the symmetry hidden within.

## 426 Shortest Common Supersequence (SCS)

The Shortest Common Supersequence (SCS) problem asks for the shortest string that contains both given strings as subsequences. It's like merging two sequences together without breaking order, balancing overlap and inclusion.

This problem is a close companion to LCS, in fact, its length can be directly expressed in terms of the Longest Common Subsequence.

### What Problem Are We Solving?

Given two strings

$$X = x_1, x_2, \dots, x_m, \quad Y = y_1, y_2, \dots, y_n$$

find the length of the shortest string that contains both as subsequences.

Define the state:

$$dp[i][j] = \text{length of SCS of prefixes } X[0..i-1] \text{ and } Y[0..j-1]$$

Recurrence:

$$dp[i][j] = \begin{cases} i, & \text{if } j = 0, \\ j, & \text{if } i = 0, \\ 1 + dp[i - 1][j - 1], & \text{if } x_{i-1} = y_{j-1}, \\ 1 + \min(dp[i - 1][j], dp[i][j - 1]), & \text{if } x_{i-1} \neq y_{j-1}. \end{cases}$$

Answer:

$$dp[m][n]$$

Alternate formula:

$$\text{SCS length} = m + n - \text{LCS length}$$

### How Does It Work (Plain Language)

If two characters match, you include it once and move diagonally. If they differ, include one character and move toward the smaller subproblem (skipping one side). You're building the shortest merged string preserving both orders.

Think of it as stitching the two sequences together with minimal redundancy.

Example: X = "AGGTAB", Y = "GXTXAYB"

LCS = "GTAB" (length 4)

So:

$$\text{SCS length} = 6 + 7 - 4 = 9$$

SCS = "AGXGTAXAYB"

### Tiny Code (Easy Versions)

C (DP Table)

```
#include <stdio.h>
#define MIN(a,b) ((a) < (b) ? (a) : (b))

int main(void) {
    char X[100], Y[100];
    printf("Enter first string: ");
    scanf("%s", X);
    printf("Enter second string: ");
    scanf("%s", Y);
```

```

int m = 0, n = 0;
while (X[m]) m++;
while (Y[n]) n++;

int dp[m + 1][n + 1];
for (int i = 0; i <= m; i++) dp[i][0] = i;
for (int j = 0; j <= n; j++) dp[0][j] = j;

for (int i = 1; i <= m; i++) {
    for (int j = 1; j <= n; j++) {
        if (X[i-1] == Y[j-1])
            dp[i][j] = 1 + dp[i-1][j-1];
        else
            dp[i][j] = 1 + MIN(dp[i-1][j], dp[i][j-1]);
    }
}

printf("SCS length: %d\n", dp[m][n]);
return 0;
}

```

Python (Straightforward DP)

```

X = input("Enter first string: ")
Y = input("Enter second string: ")

m, n = len(X), len(Y)
dp = [[0]*(n+1) for _ in range(m+1)]

for i in range(m+1):
    dp[i][0] = i
for j in range(n+1):
    dp[0][j] = j

for i in range(1, m+1):
    for j in range(1, n+1):
        if X[i-1] == Y[j-1]:
            dp[i][j] = 1 + dp[i-1][j-1]
        else:
            dp[i][j] = 1 + min(dp[i-1][j], dp[i][j-1])

print("SCS length:", dp[m][n])

```

## Why It Matters

- Shows merging sequences with order preservation
- Ties directly to LCS via the formula
- Useful for file merging, version control, and sequence alignment
- Demonstrates minimal superstructure over two DPs

It's the “union” counterpart to the “intersection” of LCS.

## Step-by-Step Example

X = "AGGTAB", Y = "GXTXAYB"

| Step        | Action | Result   |
|-------------|--------|----------|
| Compare A,G | differ | choose A |
| Compare G,G | match  | add G    |
| Compare G,X | differ | add X    |
| Compare G,T | differ | add T    |
| Compare T,X | differ | add X    |
| Compare A,A | match  | add A    |
| Compare B,B | match  | add B    |

SCS = "AGXGTXAYB"

Length = 9

## A Gentle Proof (Why It Works)

Every SCS must include all characters of both strings in order.

- If last chars match: append once  $\rightarrow 1 + dp[i-1][j-1]$
- Else, shortest option comes from skipping one character from either string.

By induction on  $(i, j)$ , since subproblems solve strictly smaller prefixes, we get optimal length.

The equivalence

$$|SCS| = m + n - |LCS|$$

follows because overlapping LCS chars are counted twice when summing lengths, and must be subtracted once.

## Try It Yourself

1. Reconstruct actual SCS string (traceback from  $dp[m][n]$ ).
2. Verify  $|SCS| = |X| + |Y| - |LCS|$ .
3. Compare SCS vs concatenation  $X + Y$ .
4. Apply to sequences with no overlap.
5. Test with identical strings (SCS = same string).

## Test Cases

| X         | Y         | Expected Length | Notes          |
|-----------|-----------|-----------------|----------------|
| “AGGTAB”  | “GXTXAYB” | 9               | overlap GTAB   |
| “ABCBDAB” | “BDCAB”   | 9               | shares BCAB    |
| “HELLO”   | “GEEK”    | 8               | no big overlap |
| “AB”      | “AB”      | 2               | identical      |
| “AB”      | “CD”      | 4               | disjoint       |

## Complexity

- Time:  $O(m \times n)$
- Space:  $O(m \times n)$ , or  $O(\min(m, n))$  for length only

The Shortest Common Supersequence weaves two strings into one, the tightest possible thread that holds both stories together.

## 427 Longest Repeated Subsequence

The Longest Repeated Subsequence (LRS) of a string is the longest subsequence that appears at least twice in the string without reusing the same index position. It is like LCS of a string with itself, with an extra constraint to avoid matching a character with itself.

### What Problem Are We Solving?

Given a string

$$S = s_1, s_2, \dots, s_n$$

find the length of the longest subsequence that occurs at least twice in (S) with disjoint index positions.

Define the state by comparing the string with itself:

$dp[i][j]$  = LRS length for  $S[1..i]$  and  $S[1..j]$

Recurrence:

$$dp[i][j] = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0, \\ 1 + dp[i-1][j-1], & \text{if } s_i = s_j \text{ and } i \neq j, \\ \max(dp[i-1][j], dp[i][j-1]), & \text{otherwise.} \end{cases}$$

Answer:

$$dp[n][n]$$

The key difference from LCS is the constraint  $i \neq j$  to prevent matching the same occurrence of a character.

### How Does It Work (Plain Language)

Think of aligning the string with itself. You are looking for common subsequences, but you are not allowed to match a character to its identical position. When characters match at different positions, you extend the repeated subsequence. When they do not match or are at the same position, you take the best from skipping one side.

Example:  $S = "aabebcdd"$  One LRS is "abd" with length 3.

### Tiny Code (Easy Versions)

C (2D DP)

```
#include <stdio.h>
#define MAX(a,b) ((a) > (b) ? (a) : (b))

int main(void) {
    char S[1005];
    printf("Enter string: ");
    scanf("%s", S);

    int n = 0; while (S[n]) n++;

    int dp[n + 1][n + 1];
    for (int i = 0; i <= n; i++)
        for (int j = 0; j <= n; j++)
```

```

dp[i][j] = 0;

for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        if (S[i-1] == S[j-1] && i != j)
            dp[i][j] = 1 + dp[i-1][j-1];
        else
            dp[i][j] = MAX(dp[i-1][j], dp[i][j-1]);
    }
}

printf("LRS length: %d\n", dp[n][n]);
return 0;
}

```

Python (Straightforward DP)

```

S = input("Enter string: ")
n = len(S)
dp = [[0]*(n+1) for _ in range(n+1)]

for i in range(1, n+1):
    for j in range(1, n+1):
        if S[i-1] == S[j-1] and i != j:
            dp[i][j] = 1 + dp[i-1][j-1]
        else:
            dp[i][j] = max(dp[i-1][j], dp[i][j-1])

print("LRS length:", dp[n][n])

```

## Why It Matters

- Illustrates transforming a problem into LCS on the same string with a simple constraint.
- Useful for detecting repeated patterns and compression signals.
- Builds intuition for self-alignment DPs and index constraints.

## Step by Step Example

S = "aabebcdd" Let us look at the alignment idea:

- Matching pairs at different indices:

- **a** at positions 1 and 2 can contribute, but not with the same index.
- **b** at positions 3 and 6.
- **d** at positions 7 and 8.

A valid repeated subsequence is "abd" using indices (1,3,7) and (2,6,8). Length (= 3).

### **A Gentle Proof (Why It Works)**

Consider LCS of (S) with itself:

- If you allowed matches at the same indices, you would trivially match every character with itself and get (n).
- By forbidding matches where ( $i = j$ ), any character contributes only when there exists another occurrence at a different index.
- The recurrence mirrors LCS but enforces ( $i \neq j$ ).
- By induction on ( $i, j$ ), the table accumulates exactly the lengths of repeated subsequences, and the maximum at ( $dp[n][n]$ ) is the LRS length.

### **Try It Yourself**

1. Reconstruct one LRS by tracing back from ( $dp[n][n]$ ) while respecting ( $i \neq j$ ).
2. Modify to count the number of distinct LRS of maximum length.
3. Compare LRS and LPS on the same string to see structural differences.
4. Handle ties when reconstructing to get the lexicographically smallest LRS.
5. Test behavior on strings with all unique characters.

### **Test Cases**

| S          | Expected LRS length | One LRS |
|------------|---------------------|---------|
| "aabebcdd" | 3                   | "abd"   |
| "axxxy"    | 2                   | "xx"    |
| "aaaa"     | 3                   | "aaa"   |
| "abc"      | 0                   | " "     |
| "aaba"     | 2                   | "aa"    |

## Complexity

- Time: ( $O(n^2)$ )
- Space: ( $O(n^2)$ )

The Longest Repeated Subsequence is LCS turned inward. Compare the string with itself, forbid identical positions, and the repeated pattern reveals itself.

## 428 String Interleaving

The String Interleaving problem asks whether a string  $S$  can be formed by interleaving (or weaving together) two other strings  $X$  and  $Y$  while preserving the relative order of characters from each.

It's a dynamic programming problem that elegantly captures sequence merging under order constraints, similar in spirit to merging two sorted lists.

### What Problem Are We Solving?

Given three strings  $X$ ,  $Y$ , and  $S$ , determine if  $S$  is a valid interleaving of  $X$  and  $Y$ .

We define the state:

$dp[i][j] = \text{True}$  if  $S[0..i + j - 1]$  can be formed by interleaving  $X[0..i - 1]$  and  $Y[0..j - 1]$

Recurrence:

$$dp[i][j] = \begin{cases} dp[i-1][j], & \text{if } X[i-1] = S[i+j-1] \text{ and } dp[i-1][j], \\ dp[i][j-1], & \text{if } Y[j-1] = S[i+j-1] \text{ and } dp[i][j-1], \\ dp[i-1][j] \vee dp[i][j-1], & \text{if both conditions hold.} \end{cases}$$

Base conditions:

$$dp[0][0] = \text{True}$$

$$dp[i][0] = dp[i-1][0] \wedge (X[i-1] = S[i-1])$$

$$dp[0][j] = dp[0][j-1] \wedge (Y[j-1] = S[j-1])$$

Answer:

$$dp[m][n]$$

where  $m = |X|$ ,  $n = |Y|$ .

### How Does It Work (Plain Language)

You have two input strings  $X$  and  $Y$ , and you're asked whether you can merge them in order to get  $S$ .

Each step, decide whether the next character in  $S$  should come from  $X$  or  $Y$ , as long as you don't break the order within either.

Imagine reading from two ribbons of characters, you can switch between them but never rearrange within a ribbon.

Example:  $X = \text{"abc"}$ ,  $Y = \text{"def"}$ ,  $S = \text{"adbcef"}$

Valid interleaving:  $a$  (from  $X$ ),  $d$  (from  $Y$ ),  $b$  (from  $X$ ),  $c$  (from  $X$ ),  $e$  (from  $Y$ ),  $f$  (from  $Y$ )

### Tiny Code (Easy Versions)

C (2D Boolean DP)

```
#include <stdio.h>
#include <stdbool.h>
#include <string.h>

bool isInterleave(char *X, char *Y, char *S) {
    int m = strlen(X), n = strlen(Y);
    if (m + n != strlen(S)) return false;

    bool dp[m+1][n+1];
    dp[0][0] = true;

    for (int i = 1; i <= m; i++)
        dp[i][0] = dp[i-1][0] && X[i-1] == S[i-1];

    for (int j = 1; j <= n; j++)
        dp[0][j] = dp[0][j-1] && Y[j-1] == S[j-1];

    for (int i = 1; i <= m; i++) {
```

```

        for (int j = 1; j <= n; j++) {
            dp[i][j] = false;
            if (X[i-1] == S[i+j-1]) dp[i][j] |= dp[i-1][j];
            if (Y[j-1] == S[i+j-1]) dp[i][j] |= dp[i][j-1];
        }
    }

    return dp[m][n];
}

int main(void) {
    char X[100], Y[100], S[200];
    printf("Enter X: "); scanf("%s", X);
    printf("Enter Y: "); scanf("%s", Y);
    printf("Enter S: "); scanf("%s", S);

    if (isInterleave(X, Y, S))
        printf("Yes, S is an interleaving of X and Y.\n");
    else
        printf("No, S cannot be formed.\n");

    return 0;
}

```

Python (Simple DP Table)

```

X = input("Enter X: ")
Y = input("Enter Y: ")
S = input("Enter S: ")

m, n = len(X), len(Y)
if len(S) != m + n:
    print("No")
else:
    dp = [[False]*(n+1) for _ in range(m+1)]
    dp[0][0] = True

    for i in range(1, m+1):
        dp[i][0] = dp[i-1][0] and X[i-1] == S[i-1]
    for j in range(1, n+1):
        dp[0][j] = dp[0][j-1] and Y[j-1] == S[j-1]

```

```

for i in range(1, m+1):
    for j in range(1, n+1):
        dp[i][j] = (X[i-1] == S[i+j-1] and dp[i-1][j]) or \
                    (Y[j-1] == S[i+j-1] and dp[i][j-1])

print("Yes" if dp[m][n] else "No")

```

## Why It Matters

- Demonstrates two-sequence merging under order constraints
- Core idea behind path interleaving, merge scheduling, and string weaving
- Good stepping stone for problems involving 2D grid DPs and string constraints

## Step-by-Step Example

$X = \text{"abc"}$ ,  $Y = \text{"def"}$ ,  $S = \text{"adbcef"}$

| i | j | $X[i]$ | $Y[j]$ | $dp[i][j]$ | Explanation       |
|---|---|--------|--------|------------|-------------------|
| 0 | 0 | " "    | " "    | T          | base              |
| 1 | 0 | a      | " "    | T          | a from X          |
| 1 | 1 | a      | d      | T          | d from Y          |
| 2 | 1 | ab     | d      | T          | b from X          |
| 3 | 1 | abc    | d      | F          | cannot match next |
| 3 | 2 | abc    | de     | T          | e from Y          |
| 3 | 3 | abc    | def    | T          | f from Y          |

Answer: True

## A Gentle Proof (Why It Works)

At any point, we have used  $i + j$  characters from  $S$ :

- If last came from  $X$ :  $X[i - 1] = S[i + j - 1]$  and  $dp[i - 1][j]$  was True
- If last came from  $Y$ :  $Y[j - 1] = S[i + j - 1]$  and  $dp[i][j - 1]$  was True

By filling the table left-to-right, top-to-bottom, every prefix is validated before combining. Inductive reasoning ensures correctness for all prefixes.

## Try It Yourself

1. Print one valid interleaving path
2. Modify to count total interleavings
3. Handle strings with duplicate characters carefully
4. Try on examples where  $X$  and  $Y$  share common prefixes
5. Extend to three strings interleaving

## Test Cases

| X     | Y     | S        | Expected |
|-------|-------|----------|----------|
| “abc” | “def” | “adbcef” | True     |
| “ab”  | “cd”  | “abcd”   | True     |
| “ab”  | “cd”  | “acbd”   | True     |
| “ab”  | “cd”  | “acdb”   | False    |
| “aa”  | “ab”  | “aaba”   | True     |

## Complexity

- Time:  $O(m \times n)$
- Space:  $O(m \times n)$ , can be reduced to  $O(n)$

The String Interleaving problem is about harmony, weaving two sequences together, letter by letter, in perfect order.

## 429 Sequence Alignment (Bioinformatics)

The Sequence Alignment problem asks how to best align two sequences (often DNA, RNA, or proteins) to measure their similarity, allowing for gaps and mismatches. It forms the foundation of bioinformatics, string similarity, and edit-based scoring systems.

Unlike edit distance, sequence alignment assigns scores for matches, mismatches, and gaps, and seeks a maximum score, not a minimal edit count.

## What Problem Are We Solving?

Given two sequences

$$X = x_1, x_2, \dots, x_m, \quad Y = y_1, y_2, \dots, y_n$$

and scoring rules:

- +1 for a match
- -1 for a mismatch
- -2 for a gap (insertion/deletion)

we want to find an alignment of  $X$  and  $Y$  that maximizes the total score.

We define the state:

$$dp[i][j] = \text{maximum alignment score between } X[0..i-1] \text{ and } Y[0..j-1]$$

Recurrence:

$$dp[i][j] = \max \left\{ dp[i-1][j-1] + \text{score}(x_{i-1}, y_{j-1}), dp[i-1][j] + \text{gap penalty}, dp[i][j-1] + \text{gap penalty} \right\}$$

Base:

$$dp[i][0] = i \times \text{gap penalty}, \quad dp[0][j] = j \times \text{gap penalty}$$

Answer:  $dp[m][n]$

## How Does It Work (Plain Language)

We fill a grid where each cell  $(i, j)$  represents the best score to align the first  $i$  characters of  $X$  with the first  $j$  of  $Y$ .

At each step, we decide:

1. Match/Mismatch ( $x_{i-1}$  with  $y_{j-1}$ )
2. Insert a gap in  $Y$  (skip character in  $X$ )
3. Insert a gap in  $X$  (skip character in  $Y$ )

The final cell holds the optimal alignment score. Tracing back reveals the aligned strings, with dashes representing gaps.

Example:

$X = "GATTACA"$   $Y = "GCATGCU"$

One alignment:

```
G A T T A C A -  
| | | | | |  
G - C A T G C U
```

### Tiny Code (Easy Versions)

C (Global Alignment / Needleman–Wunsch)

```
#include <stdio.h>  
#define MAX(a,b) ((a) > (b) ? (a) : (b))  
  
int score(char a, char b) {  
    return a == b ? 1 : -1;  
}  
  
int main(void) {  
    char X[100], Y[100];  
    printf("Enter X: "); scanf("%s", X);  
    printf("Enter Y: "); scanf("%s", Y);  
  
    int m = 0, n = 0;  
    while (X[m]) m++;  
    while (Y[n]) n++;  
  
    int gap = -2;  
    int dp[m+1][n+1];  
  
    for (int i = 0; i <= m; i++) dp[i][0] = i * gap;  
    for (int j = 0; j <= n; j++) dp[0][j] = j * gap;  
  
    for (int i = 1; i <= m; i++) {  
        for (int j = 1; j <= n; j++) {  
            int match = dp[i-1][j-1] + score(X[i-1], Y[j-1]);  
            int delete = dp[i-1][j] + gap;  
            int insert = dp[i][j-1] + gap;  
            dp[i][j] = MAX(match, MAX(delete, insert));  
        }  
    }  
}
```

```

        int insert = dp[i][j-1] + gap;
        dp[i][j] = MAX(match, MAX(delete, insert));
    }

printf("Max alignment score: %d\n", dp[m][n]);
return 0;
}

```

Python (Clean Version)

```

X = input("Enter X: ")
Y = input("Enter Y: ")

m, n = len(X), len(Y)
match, mismatch, gap = 1, -1, -2

dp = [[0]*(n+1) for _ in range(m+1)]

for i in range(1, m+1):
    dp[i][0] = i * gap
for j in range(1, n+1):
    dp[0][j] = j * gap

for i in range(1, m+1):
    for j in range(1, n+1):
        score = match if X[i-1] == Y[j-1] else mismatch
        dp[i][j] = max(
            dp[i-1][j-1] + score,
            dp[i-1][j] + gap,
            dp[i][j-1] + gap
        )

print("Max alignment score:", dp[m][n])

```

## Why It Matters

- Foundational in bioinformatics (DNA, RNA, protein comparison)
- Used in spell correction, plagiarism detection, text similarity
- Shows a weighted DP with scores, not just counts
- Demonstrates path reconstruction with multiple decisions

This is the generalization of edit distance to *scored alignment*.

### Step-by-Step Example

Let  $X = \text{"AGT"}$ ,  $Y = \text{"GTT"}$  Match = +1, Mismatch = -1, Gap = -2

| i | j | $X[0..i]$ | $Y[0..j]$ | $dp[i][j]$ | Choice   |
|---|---|-----------|-----------|------------|----------|
| 1 | 1 | A, G      | G         | -1         | mismatch |
| 2 | 2 | AG, GT    | GT        | +0         | align G  |
| 3 | 3 | AGT, GTT  | GTT       | +1         | align T  |

Answer = +1

### A Gentle Proof (Why It Works)

Each  $dp[i][j]$  represents the best possible score achievable aligning  $X[0..i-1]$  and  $Y[0..j-1]$ . Induction ensures correctness:

- Base:  $dp[0][j]$ ,  $dp[i][0]$  handle leading gaps
- Step: At each  $(i, j)$ , you consider all valid transitions, match/mismatch, insert, delete, and take the max. Thus  $dp[m][n]$  is globally optimal.

### Try It Yourself

1. Trace back to print alignment with gaps
2. Try different scoring systems
3. Compare global (Needleman–Wunsch) vs local (Smith–Waterman) alignment
4. Handle affine gaps (gap opening + extension)
5. Visualize grid paths as alignments

### Test Cases

| X         | Y         | Expected           | Notes        |
|-----------|-----------|--------------------|--------------|
| “AGT”     | “GTT”     | 1                  | one match    |
| “GATTACA” | “GCATGCU” | depends on scoring | classic      |
| “ABC”     | “ABC”     | 3                  | all match    |
| “ABC”     | “DEF”     | -3                 | all mismatch |
| “A”       | “AAA”     | -2                 | gaps added   |

## Complexity

- Time:  $O(m \times n)$
- Space:  $O(m \times n)$  (can be reduced with row-rolling)

The Sequence Alignment problem teaches that similarity is not just about matches, it's about balancing alignments, mismatches, and gaps to find the best correspondence between two sequences.

## 430 Diff Algorithm (Myers / DP)

The Diff Algorithm compares two sequences and finds their shortest edit script (SES), the minimal sequence of insertions and deletions required to transform one into the other. It's the heart of tools like `git diff` and `diff`, providing human-readable change summaries.

The Myers Algorithm is the most famous linear-space implementation, but the DP formulation builds on edit distance and LCS intuition.

### What Problem Are We Solving?

Given two strings

$$X = x_1, x_2, \dots, x_m, \quad Y = y_1, y_2, \dots, y_n$$

find a minimal sequence of edits (insertions and deletions) to turn  $X$  into  $Y$ .

Each edit transforms one sequence toward the other, and matching characters are left untouched.

The minimal number of edits equals:

$$\text{SES length} = m + n - 2 \times \text{LCS length}$$

We can also explicitly trace the path to recover the diff.

## Recurrence (DP Formulation)

Let  $dp[i][j]$  be the minimal number of edits to convert  $X[0..i - 1]$  into  $Y[0..j - 1]$ .

Then:

$$dp[i][j] = \begin{cases} i, & \text{if } j = 0, \\ j, & \text{if } i = 0, \\ dp[i - 1][j - 1], & \text{if } x_{i-1} = y_{j-1}, \\ 1 + \min(dp[i - 1][j], dp[i][j - 1]), & \text{if } x_{i-1} \neq y_{j-1}. \end{cases}$$

Answer:

$$dp[m][n]$$

The traceback reconstructs the sequence of operations:  
keeping, deleting, or inserting characters to transform  $X$  into  $Y$ .

## How Does It Work (Plain Language)

Imagine aligning two sequences line by line. When characters match, move diagonally (no cost). If they differ, you must either delete from  $X$  or insert from  $Y$ .

By walking through a grid of all prefix pairs, you can find the shortest edit path, the same logic as `git diff`.

Example:  $X = \text{"ABCABBA"}$   $Y = \text{"CBABAC"}$

One minimal diff:

```
- A  
  B  
  C  
+ B  
  A  
  B  
- B  
+ A  
  C
```

## Tiny Code (Easy Versions)

C (DP Traceback for Diff)

```
#include <stdio.h>
#define MIN(a,b) ((a)<(b)?(a):(b))

int main(void) {
    char X[100], Y[100];
    printf("Enter X: "); scanf("%s", X);
    printf("Enter Y: "); scanf("%s", Y);

    int m = 0, n = 0;
    while (X[m]) m++;
    while (Y[n]) n++;

    int dp[m+1][n+1];
    for (int i = 0; i <= m; i++) dp[i][0] = i;
    for (int j = 0; j <= n; j++) dp[0][j] = j;

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (X[i-1] == Y[j-1])
                dp[i][j] = dp[i-1][j-1];
            else
                dp[i][j] = 1 + MIN(dp[i-1][j], dp[i][j-1]);
        }
    }

    printf("Edit distance: %d\n", dp[m][n]);

    // Traceback
    int i = m, j = n;
    printf("Diff:\n");
    while (i > 0 || j > 0) {
        if (i > 0 && j > 0 && X[i-1] == Y[j-1]) {
            printf(" %c\n", X[i-1]);
            i--; j--;
        } else if (i > 0 && dp[i][j] == dp[i-1][j] + 1) {
            printf("- %c\n", X[i-1]);
            i--;
        } else {
            printf("+ %c\n", Y[j-1]);
        }
    }
}
```

```

        j--;
    }
}

return 0;
}

```

Python (Simple Diff Reconstruction)

```

X = input("Enter X: ")
Y = input("Enter Y: ")

m, n = len(X), len(Y)
dp = [[0]*(n+1) for _ in range(m+1)]

for i in range(m+1):
    dp[i][0] = i
for j in range(n+1):
    dp[0][j] = j

for i in range(1, m+1):
    for j in range(1, n+1):
        if X[i-1] == Y[j-1]:
            dp[i][j] = dp[i-1][j-1]
        else:
            dp[i][j] = 1 + min(dp[i-1][j], dp[i][j-1])

i, j = m, n
ops = []
while i > 0 or j > 0:
    if i > 0 and j > 0 and X[i-1] == Y[j-1]:
        ops.append(f"- {X[i-1]}")
        i -= 1; j -= 1
    elif i > 0 and dp[i][j] == dp[i-1][j] + 1:
        ops.append(f"+ {Y[j-1]}")
        i -= 1
    else:
        ops.append(f"- {X[i-1]}")
        j -= 1

print("Edit distance:", dp[m][n])
print("Diff:")

```

```

for op in reversed(ops):
    print(op)

```

## Why It Matters

- Foundation of version control systems (`git diff`, `patch`)
- Minimizes edit operations for transformation
- Simplifies merge conflict resolution
- Builds upon LCS and Edit Distance concepts
- Demonstrates traceback-based reconstruction

The diff is the human-readable face of dynamic programming, turning tables into insight.

## Step-by-Step Example

$X = \text{"ABCABBA"}$ ,  $Y = \text{"CBABAC"}$

| Operation | Explanation                       |
|-----------|-----------------------------------|
| - A       | X starts with 'A' not in Y prefix |
| B         | match                             |
| C         | match                             |
| + B       | insert                            |
| A         | match                             |
| B         | match                             |
| - B       | delete                            |
| + A       | insert                            |
| C         | match                             |

Total edits = 4 (minimal)

## A Gentle Proof (Why It Works)

- If  $x_{i-1} = y_{j-1}$ , skip both (no cost).
- Otherwise, best path must add a new operation (insert/delete).
- $dp[i][j]$  stores minimal edits for prefixes. By induction,  $dp[m][n]$  is the minimal SES length.

## Try It Yourself

1. Print diff symbols (+, -, space) in alignment view
2. Compare diff path with LCS path
3. Try with real words: "kitten" → "sitting"
4. Modify cost: insertion/deletion weights
5. Use colors or indentation to visualize output

## Test Cases

| X         | Y         | Edits | Notes           |
|-----------|-----------|-------|-----------------|
| "ABCABBA" | "CBABAC"  | 4     | classic example |
| "abc"     | "abc"     | 0     | identical       |
| "abc"     | "def"     | 6     | all replaced    |
| "kitten"  | "sitting" | 3     | classic         |
| " "       | "xyz"     | 3     | all inserts     |

## Complexity

- Time:  $O(m \times n)$
- Space:  $O(m \times n)$  (traceback needs full table)

The Diff Algorithm transforms comparison into storytelling, showing precisely how one sequence evolves into another, one edit at a time.

# Section 44. Matrix and Chain Problems

## 431 Matrix Chain Multiplication

The Matrix Chain Multiplication problem asks for the most efficient way to parenthesize a product of matrices so that the total number of scalar multiplications is minimized. Matrix multiplication is associative, so the order of multiplication can change the cost dramatically even though the result is the same shape.

### What Problem Are We Solving?

Given a chain of matrices

$$A_1 A_2 \cdots A_n$$

with dimensions

$$A_i \text{ is } p_{i-1} \times p_i \quad (i = 1..n),$$

choose the parenthesization that minimizes scalar multiplications.

Define the state:

$$dp[i][j] = \text{minimum cost to multiply } A_i A_{i+1} \cdots A_j$$

Recurrence:

$$dp[i][i] = 0$$

$$dp[i][j] = \min_{i \leq k < j} (dp[i][k] + dp[k+1][j] + p_{i-1} p_k p_j)$$

The last term is the cost of multiplying the two resulting matrices from the split at (k).

Answer:

$$dp[1][n]$$

Optionally keep a split table ( $\text{split}[i][j]$ ) storing the (k) achieving the minimum to reconstruct the optimal parenthesization.

## How Does It Work (Plain Language)

Matrix-chain multiplication DP

Recurrence

$$m[i, i] = 0,$$
$$m[i, j] = \min_{i \leq k < j} (m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j) \quad (i < j),$$

where matrices are  $A_i$  of size  $p_{i-1} \times p_i$ .

Example with dimensions

$$p = [10, 30, 5, 60], \quad A_1 : 10 \times 30, \quad A_2 : 30 \times 5, \quad A_3 : 5 \times 60.$$

Two ways to parenthesize:

1)  $(A_1 A_2) A_3$

Cost

$$(10 \cdot 30 \cdot 5) + (10 \cdot 5 \cdot 60) = 1500 + 3000 = 4500.$$

2)  $A_1 (A_2 A_3)$

Cost

$$(30 \cdot 5 \cdot 60) + (10 \cdot 30 \cdot 60) = 9000 + 18000 = 27000.$$

Minimum cost is 4500, achieved by  $(A_1 A_2) A_3$  with split  $k = 1$ .

## Tiny Code (Easy Versions)

C (DP with reconstruction)

```
#include <stdio.h>
#include <limits.h>

void print_optimal(int i, int j, int split[105][105]) {
    if (i == j) { printf("A%d", i); return; }
    printf("(");
    int k = split[i][j];
    print_optimal(i, k, split);
    printf(" x ");
    print_optimal(k+1, j, split);
    printf(")");
}
```

```

int main(void) {
    int n;
    printf("Enter number of matrices: ");
    scanf("%d", &n);
    int p[n+1];
    printf("Enter dimensions p0..pn: ");
    for (int i = 0; i <= n; i++) scanf("%d", &p[i]);

    long long dp[105][105];
    int split[105][105];

    for (int i = 1; i <= n; i++) dp[i][i] = 0;

    for (int len = 2; len <= n; len++) {
        for (int i = 1; i + len - 1 <= n; i++) {
            int j = i + len - 1;
            dp[i][j] = LLONG_MAX;
            for (int k = i; k < j; k++) {
                long long cost = dp[i][k] + dp[k+1][j] + 1LL*p[i-1]*p[k]*p[j];
                if (cost < dp[i][j]) {
                    dp[i][j] = cost;
                    split[i][j] = k;
                }
            }
        }
    }

    printf("Min scalar multiplications: %lld\n", dp[1][n]);
    printf("Optimal parenthesization: ");
    print_optimal(1, n, split);
    printf("\n");
    return 0;
}

```

Python (DP with reconstruction)

```

def matrix_chain_order(p):
    n = len(p) - 1
    dp = [[0]*(n+1) for _ in range(n+1)]
    split = [[0]*(n+1) for _ in range(n+1)]

    for length in range(2, n+1):

```

```

for i in range(1, n - length + 2):
    j = i + length - 1
    dp[i][j] = float('inf')
    for k in range(i, j):
        cost = dp[i][k] + dp[k+1][j] + p[i-1]*p[k]*p[j]
        if cost < dp[i][j]:
            dp[i][j] = cost
            split[i][j] = k
return dp, split

def build_solution(split, i, j):
    if i == j:
        return f"A{i}"
    k = split[i][j]
    return "(" + build_solution(split, i, k) + " x " + build_solution(split, k+1, j) + ")"

p = list(map(int, input("Enter p0..pn: ").split()))
dp, split = matrix_chain_order(p)
n = len(p) - 1
print("Min scalar multiplications:", dp[1][n])
print("Optimal parenthesization:", build_solution(split, 1, n))

```

## Why It Matters

- Canonical example of interval DP and optimal binary partitioning
- Shows how associativity allows many evaluation orders with different costs
- Appears in query plan optimization, automatic differentiation scheduling, graphics and compiler optimization

## Step by Step Example

For ( $p = [5, 10, 3, 12, 5, 50, 6]$ ) with ( $n=6$ ):

- Try all splits for each subchain length
- The DP eventually yields ( $dp[1][6] = 2010$ ) and an optimal structure like  $((A_1(A_2A_3))(A_4A_5A_6))$ . Exact parentheses can vary among ties but the minimal cost is unique here.

## A Gentle Proof (Why It Works)

Let  $(OPT(i,j))$  be the optimal cost for  $A_i \cdots A_j$ . In any optimal solution, the last multiplication splits the chain at some  $(k)$  with  $i \leq k < j$ . The two sides must themselves be optimal, otherwise replacing one side by a better solution improves the total, contradicting optimality. Therefore

$$OPT(i,j) = \min_{i \leq k < j} (OPT(i,k) + OPT(k+1,j) + p_{i-1}p_kp_j),$$

with  $(OPT(i,i)=0)$ . Since each subproblem uses strictly shorter chains, filling by increasing length computes all needed values before they are used.

## Try It Yourself

1. Print not only one but all optimal parenthesizations when multiple (k) tie.
2. Add a second objective like minimizing depth after minimizing cost.
3. Compare greedy choices vs DP on random instances.
4. Extend to a cost model with addition cost or cache reuse.
5. Visualize the DP table and splits along diagonals.

## Test Cases

| p (p0..pn)         | n | Expected min cost | One optimal parentheses                                     |
|--------------------|---|-------------------|-------------------------------------------------------------|
| [10,30,5,60]       | 3 | 4500              | $(A_1 \times A_2) \times A_3$                               |
| [5,10,3,12,5,50,6] | 6 | 2010              | one optimal structure                                       |
| [40,20,30,10,30]   | 4 | 26000             | $((A_1 \times (A_2 \times A_3)) \times A_4)$ or tie variant |
| [10,20,30]         | 2 | 6000              | $A_1 \times A_2$                                            |
| [2,3,4,5]          | 3 | 64                | $(A_1 \times A_2) \times A_3$                               |

## Complexity

- Time:  $(O(n^3))$  for the triple loop over  $(i,j,k)$
- Space:  $(O(n^2))$  for the DP and split tables

Matrix Chain Multiplication is the textbook pattern for interval DP: pick a split, combine optimal subchains, and account for the boundary multiplication cost.

## 432 Boolean Parenthesization

The Boolean Parenthesization problem (also called the Boolean Expression Evaluation problem) asks: *Given a boolean expression consisting of T (true), F (false), and operators (&, |, ^), how many ways can we parenthesize it so that it evaluates to True?*

It's a classic DP over intervals problem where we explore all possible splits between operators, combining sub-results based on logic rules.

### What Problem Are We Solving?

Given a boolean expression string like T|F&T^T, count the number of ways to parenthesize it so that it evaluates to True.

We must consider both True and False counts for sub-expressions.

Let:

- $dpT[i][j] = \text{number of ways } expr[i..j] \text{ evaluates to True}$
- $dpF[i][j] = \text{number of ways } expr[i..j] \text{ evaluates to False}$

If expression length is  $n$ , then we only consider operands at even indices and operators at odd indices.

### Recurrence

For every split at operator  $k$  between  $i$  and  $j$ :

Let  $op = expr[k]$

Compute:

$$\text{TotalTrue} = \begin{cases} dpT[i][k-1] \cdot dpT[k+1][j], & \text{if } op = \& \\ dpT[i][k-1] \cdot dpT[k+1][j] + dpT[i][k-1] \cdot dpF[k+1][j] + dpF[i][k-1] \cdot dpT[k+1][j], & \text{if } op = | \\ dpT[i][k-1] \cdot dpF[k+1][j] + dpF[i][k-1] \cdot dpT[k+1][j], & \text{if } op = ^ \end{cases}$$

Similarly for  $dpF[i][j]$  using complementary logic.

Base cases:

$$dpT[i][i] = 1 \text{ if } expr[i] = 'T' \text{ else } 0$$
$$dpF[i][i] = 1 \text{ if } expr[i] = 'F' \text{ else } 0$$

Answer =  $dpT[0][n - 1]$

## How Does It Work (Plain Language)

We cut the expression at each operator and combine the truth counts of the left and right sides according to boolean logic.

For each subexpression, we record:

- how many parenthesizations make it True
- how many make it False

Then we combine smaller subproblems to get bigger ones, just like Matrix Chain Multiplication, but using logic rules.

Example: Expression = T|F&T

We can group as:

1.  $(T|F) \& T \rightarrow \text{True}$
2.  $T | (F \& T) \rightarrow \text{True}$

Answer = 2

## Tiny Code (Easy Versions)

C (Bottom-Up DP)

```
#include <stdio.h>
#include <string.h>

#define MAX 105

int dpT[MAX][MAX], dpF[MAX][MAX];

int main(void) {
    char expr[105];
    printf("Enter expression (T/F with &|^): ");
    scanf("%s", expr);

    int n = strlen(expr);
    for (int i = 0; i < n; i += 2) {
        dpT[i][i] = (expr[i] == 'T');
        dpF[i][i] = (expr[i] == 'F');
```

```

    }

    for (int len = 3; len <= n; len += 2) {
        for (int i = 0; i + len - 1 < n; i += 2) {
            int j = i + len - 1;
            dpT[i][j] = dpF[i][j] = 0;

            for (int k = i + 1; k < j; k += 2) {
                char op = expr[k];
                int lT = dpT[i][k-1], lF = dpF[i][k-1];
                int rT = dpT[k+1][j], rF = dpF[k+1][j];

                if (op == '&') {
                    dpT[i][j] += lT * rT;
                    dpF[i][j] += lT * rF + lF * rT + lF * rF;
                } else if (op == '|') {
                    dpT[i][j] += lT * rT + lT * rF + lF * rT;
                    dpF[i][j] += lF * rF;
                } else if (op == '^') {
                    dpT[i][j] += lT * rF + lF * rT;
                    dpF[i][j] += lT * rT + lF * rF;
                }
            }
        }
    }

    printf("Number of ways to get True: %d\n", dpT[0][n-1]);
    return 0;
}

```

Python (Readable DP)

```

expr = input("Enter expression (T/F with &|^): ")
n = len(expr)

dpT = [[0]*n for _ in range(n)]
dpF = [[0]*n for _ in range(n)]

for i in range(0, n, 2):
    dpT[i][i] = 1 if expr[i] == 'T' else 0
    dpF[i][i] = 1 if expr[i] == 'F' else 0

```

```

for length in range(3, n+1, 2):
    for i in range(0, n-length+1, 2):
        j = i + length - 1
        for k in range(i+1, j, 2):
            op = expr[k]
            lT, lF = dpT[i][k-1], dpF[i][k-1]
            rT, rF = dpT[k+1][j], dpF[k+1][j]

            if op == '&':
                dpT[i][j] += lT * rT
                dpF[i][j] += lT * rF + lF * rT + lF * rF
            elif op == '|':
                dpT[i][j] += lT * rT + lT * rF + lF * rT
                dpF[i][j] += lF * rF
            else: # '^'
                dpT[i][j] += lT * rF + lF * rT
                dpF[i][j] += lT * rT + lF * rF

print("Ways to evaluate to True:", dpT[0][n-1])

```

## Why It Matters

- Classic interval DP pattern with logical combination
- Shows how state splitting applies beyond arithmetic
- Foundation for boolean circuit optimization and expression counting problems
- Reinforces divide by operator technique

## Step-by-Step Example

Expression = T|F&T

Subproblems:

| Subexpr | Ways True | Ways False |
|---------|-----------|------------|
| T       | 1         | 0          |
| F       | 0         | 1          |
| T       | 1         | 0          |
| T       | F         | 1          |
| F&T     | 0         | 1          |
| T       | F&T       | 2          |
|         |           | 0          |

Answer = 2

### A Gentle Proof (Why It Works)

Each subexpression can be split at an operator  $op_k$ . The truth count of the whole depends only on the truth counts of its parts and the operator's truth table. By combining all possible  $k$  recursively, we count all valid parenthesizations. Overlapping subproblems arise when evaluating the same substring, so memoization or bottom-up filling ensures efficiency.

### Try It Yourself

1. Extend to count False outcomes too.
2. Add modulo  $10^9 + 7$  for large counts.
3. Print one valid parenthesization.
4. Try on expressions like  $T \wedge T \wedge F$  or  $T \mid F \& T \wedge T$ .
5. Modify rules for custom logic systems.

### Test Cases

| Expression   | Expected True Count |
|--------------|---------------------|
| $T$          | $F \& T$            |
| $T^T F$      | 0                   |
| $T \wedge F$ | $F$                 |
| $T \& F$     | $T$                 |
| $T$          | $T \& F$            |

### Complexity

- Time:  $O(n^3)$  (split for every operator)
- Space:  $O(n^2)$  (2 DP tables)

The Boolean Parenthesization problem is the logic mirror of Matrix Chain Multiplication, instead of minimizing cost, we're counting truth through combinatorial structure.

## 433 Burst Balloons

The Burst Balloons problem is a classic interval DP challenge. You're given a row of balloons, each with a number representing coins. When you burst a balloon, you gain coins equal to the product of its number and the numbers of its immediate neighbors. The task is to determine the maximum coins you can collect by choosing the optimal order of bursting.

### What Problem Are We Solving?

Given an array `nums` of length `n`, when you burst balloon `i`, you gain

$$\text{coins} = \text{nums}[i - 1] \times \text{nums}[i] \times \text{nums}[i + 1]$$

where `nums[i-1]` and `nums[i+1]` are the adjacent balloons still unburst.

After bursting `i`, it is removed from the sequence, changing neighbor relationships.

We want to maximize total coins by choosing the best bursting order.

To simplify boundary conditions, pad the array with 1s at both ends:

$$\text{val} = [1] + \text{nums} + [1]$$

Define DP state:

$$dp[i][j] = \text{maximum coins obtainable by bursting all balloons between } i \text{ and } j$$

Recurrence:

$$dp[i][j] = \max_{k \in (i,j)} (dp[i][k] + dp[k][j] + val[i] \cdot val[k] \cdot val[j])$$

Answer:

$$dp[0][n + 1]$$

### How Does It Work (Plain Language)

Instead of thinking “*Which balloon to pop next?*”, think “*Which balloon to pop last?*” between two boundaries.

By fixing the last balloon `k` between `i` and `j`, its neighbors are guaranteed to be `i` and `j` at that moment, so the coins earned are easy to compute: `val[i] * val[k] * val[j]`.

Then we solve the smaller subproblems:

- `dp[i][k]`: best coins from bursting balloons between `i` and `k`

- $dp[k][j]$ : best coins from bursting between  $k$  and  $j$

Combine and take the best split.

This is the reverse of the intuitive “first burst” approach, making the subproblems independent.

### Tiny Code (Easy Versions)

C (Bottom-Up DP)

```
#include <stdio.h>
#include <string.h>

#define MAX 305
#define max(a,b) ((a)>(b)?(a):(b))

int main(void) {
    int n;
    printf("Enter number of balloons: ");
    scanf("%d", &n);

    int nums[MAX], val[MAX];
    printf("Enter balloon values: ");
    for (int i = 1; i <= n; i++) scanf("%d", &nums[i]);

    val[0] = val[n+1] = 1;
    for (int i = 1; i <= n; i++) val[i] = nums[i];

    int dp[MAX][MAX];
    memset(dp, 0, sizeof(dp));

    for (int len = 2; len <= n + 1; len++) {
        for (int i = 0; i + len <= n + 1; i++) {
            int j = i + len;
            for (int k = i + 1; k < j; k++) {
                int cost = val[i] * val[k] * val[j] + dp[i][k] + dp[k][j];
                dp[i][j] = max(dp[i][j], cost);
            }
        }
    }

    printf("Max coins: %d\n", dp[0][n+1]);
```

```
    return 0;  
}
```

Python (Interval DP)

```
nums = list(map(int, input("Enter balloon values: ").split()))  
val = [1] + nums + [1]  
n = len(nums)  
dp = [[0]*(n+2) for _ in range(n+2)]  
  
for length in range(2, n+2):  
    for i in range(0, n+2-length):  
        j = i + length  
        for k in range(i+1, j):  
            dp[i][j] = max(dp[i][j], val[i]*val[k]*val[j] + dp[i][k] + dp[k][j])  
  
print("Max coins:", dp[0][n+1])
```

## Why It Matters

- Exemplifies interval DP structure: choose a pivot balloon as the “last” action
- Shows how reverse reasoning simplifies state independence
- Appears in optimization over chains, trees, brackets, and games
- Foundation for polygon triangulation and matrix multiplication variants

## Step-by-Step Example

Example: `nums = [3, 1, 5, 8]`

`Pad → val = [1, 3, 1, 5, 8, 1]`

Compute `dp[i][j]` for increasing intervals:

- Interval (1,4): choose  $k=2$  or  $3$ , compare costs
- Gradually expand to full (0,5): Optimal = 167 coins

One optimal order: `burst 1 → 5 → 3 → 8 → 1`

## A Gentle Proof (Why It Works)

Each interval  $(i, j)$  can only depend on smaller intervals  $(i, k)$  and  $(k, j)$  because the last balloon  $k$  divides the chain cleanly. By fixing  $k$  as last, we ensure both sides are independent, they share no unburst balloons. Since every subproblem is smaller, bottom-up DP fills states without cycles. Thus, optimal substructure and overlapping subproblems guarantee correctness.

## Try It Yourself

1. Implement a top-down memoized version with recursion.
2. Visualize the DP table as a triangle showing optimal splits.
3. Add reconstruction to print the burst order.
4. Try  $[1,2,3]$ ,  $[1,5]$ ,  $[7,9,8]$  to check intuition.
5. Compare performance for  $n=300$ .

## Test Cases

| nums        | Max Coins |
|-------------|-----------|
| $[3,1,5,8]$ | 167       |
| $[1,5]$     | 10        |
| $[2,2,2]$   | 12        |
| $[1,2,3]$   | 12        |
| $[9]$       | 9         |

## Complexity

- Time:  $O(n^3)$  for all subintervals and splits
- Space:  $O(n^2)$  for DP table

The Burst Balloons problem captures the essence of interval DP: choose the last action, build subproblems on either side, and let structure guide optimal order.

## 434 Optimal BST

The Optimal Binary Search Tree problem asks for the BST shape that minimizes the expected search cost given access frequencies. Even though all BSTs hold the same keys in-order, different shapes can have very different average lookup depths.

## What Problem Are We Solving?

We have sorted keys

$$K_1 < K_2 < \dots < K_n,$$

with successful-search probabilities ( $p_1, \dots, p_n$ ). Optionally, we may include probabilities ( $q_0, \dots, q_n$ ) for unsuccessful searches in the gaps between keys (classical formulation).

Goal: build a BST over these keys that minimizes expected comparisons.

Two standard DP models:

1. Full model with gaps (( $p_i, q_i$ )). This is the textbook version.
2. Simplified model with only ( $p_i$ ). Useful when you only care about successful hits.

### DP: Full Model With Gaps

Define DP over intervals of keys ( $K_i, \dots, K_j$ ) with gaps on both sides.

Weight (total probability mass in the interval, including gaps):

$$w[i][j] = \begin{cases} q_{i-1}, & i > j, \\ w[i][j-1] + p_j + q_j, & i \leq j. \end{cases}$$

Expected cost (including internal node comparisons):

$$e[i][j] = \begin{cases} q_{i-1}, & i > j, \\ \min_{r=i}^j (e[i][r-1] + e[r+1][j] + w[i][j]), & i \leq j. \end{cases}$$

Answer:  $e[1][n]$ .

If you also keep  $root[i][j]$  that stores the minimizing  $r$ , you can reconstruct the tree.

### DP: Simplified Model (success-only)

Sometimes you only have hit frequencies  $f_i$ . Let the cost count depth with each comparison adding 1. Define

$dp[i][j] = \text{minimum total weighted depth for } K_i, \dots, K_j \text{ when the root contributes 1 per key below it.}$

A convenient formulation uses prefix sums

$$S[k] = \sum_{t=1}^k f_t, \quad W(i, j) = S[j] - S[i-1].$$

Recurrence:

$$dp[i][j] = \begin{cases} 0, & i > j, \\ \min_{r=i}^j (dp[i][r-1] + dp[r+1][j]) + W(i, j), & i \leq j. \end{cases}$$

Answer:  $dp[1][n]$ .

The extra  $W(i, j)$  accounts for the fact that choosing any root increases the depth of all keys in its subtrees by 1.

## **How Does It Work (Plain Language)**

Pick the root for a range of keys. Every key not chosen as root sits one level deeper, so its cost increases. The DP tries every candidate root and adds:

- the optimal cost of the left subtree
  - the optimal cost of the right subtree
  - the penalty for pushing all nonroot keys one level deeper (their total frequency)

Choose the root that minimizes this sum for every interval.

## Tiny Code (Easy Versions)

C (full model with gaps ( $p_{-i}, q_{-i}$ ))

```
#include <stdio.h>
#include <float.h>

#define MAXN 205
#define MIN(a,b) ((a)<(b)?(a):(b))

double e[MAXN][MAXN], w[MAXN][MAXN];
int rootIdx[MAXN][MAXN];

int main(void) {
    int n;
    printf("Enter n: ");
    scanf("%d", &n);

    double p[MAXN], q[MAXN];
    printf("Enter p1..pn: ");
    for (int i = 1; i <= n; i++) scanf("%lf", &p[i]);
```

```

printf("Enter q0..qn: ");
for (int i = 0; i <= n; i++) scanf("%lf", &q[i]);

for (int i = 1; i <= n+1; i++) {
    e[i][i-1] = q[i-1];
    w[i][i-1] = q[i-1];
}

for (int len = 1; len <= n; len++) {
    for (int i = 1; i+len-1 <= n; i++) {
        int j = i + len - 1;
        w[i][j] = w[i][j-1] + p[j] + q[j];
        e[i][j] = DBL_MAX;
        for (int r = i; r <= j; r++) {
            double cost = e[i][r-1] + e[r+1][j] + w[i][j];
            if (cost < e[i][j]) {
                e[i][j] = cost;
                rootIdx[i][j] = r;
            }
        }
    }
}
printf("Optimal expected cost: %.6f\n", e[1][n]);
// rootIdx[i][j] holds the chosen root for reconstruction
return 0;
}

```

Python (simplified model with hit frequencies ( $f_{-i}$ ))

```

f = list(map(float, input("Enter f1..fn: ").split()))
n = len(f)
# 1-index for convenience
f = [0.0] + f
S = [0.0]*(n+1)
for i in range(1, n+1):
    S[i] = S[i-1] + f[i]

def W(i, j):
    return 0.0 if i > j else S[j] - S[i-1]

dp = [[0.0]*(n+2) for _ in range(n+2)]

```

```

root = [[0]*(n+2) for _ in range(n+2)]

for length in range(1, n+1):
    for i in range(1, n - length + 2):
        j = i + length - 1
        best, arg = float('inf'), -1
        for r in range(i, j+1):
            cost = dp[i][r-1] + dp[r+1][j] + W(i, j)
            if cost < best:
                best, arg = cost, r
        dp[i][j], root[i][j] = best, arg

print("Optimal weighted cost:", round(dp[1][n], 6))
# root[i][j] gives a root choice for reconstruction

```

## Why It Matters

- Models biased queries where some keys are far more popular
- Canonical interval DP with a split and an additive per-interval penalty
- Basis for query plan optimization, autocompletion tries, and decision tree shaping
- Leads to advanced speedups like Knuth optimization when conditions hold

## Step by Step Example

Simplified model with ( $f = [0.3, 0.2, 0.5]$ ) for ( $K_1 < K_2 < K_3$ ).

- For length 1: ( $dp[i][i] = f_i$ )
- For interval ([1,2]): try roots 1 or 2
  - ( $r=1: dp[1][0]+dp[2][2]+(f_1+f_2)=0+0.2+0.5=0.7$ )
  - ( $r=2: dp[1][1]+dp[3][2]+0.5=0.3+0+0.5=0.8$ ) choose ( $r=1$ ).
- For ([1,3]): try ( $r=1,2,3$ ) with penalty ( $W(1,3)=1.0$ ) compute and pick the minimum.  
The DP returns the best shape and cost.

## A Gentle Proof (Why It Works)

Consider an optimal tree for keys  $([i,j])$  whose root is  $(r)$ . All keys other than  $(K_r)$  move one level deeper, adding exactly  $(W(i,j)-p_r)$  to their cumulative cost. Splitting at  $(r)$  separates the instance into two independent subproblems  $([i,r-1])$  and  $([r+1,j])$ . If either subtree were not optimal, replacing it by a better one would reduce total cost, contradicting optimality. Thus the recurrence that scans all roots and adds the interval weight is correct. The gap model adds  $(q)$ -probabilities to the interval weight  $(w[i][j])$  and yields the classical formula.

## Try It Yourself

1. Reconstruct the tree using the stored `root` table and print it in preorder.
2. Compare the full model  $((p,q))$  versus the simplified model on the same data.
3. Normalize frequencies so  $(p_i + q_i = 1)$  and interpret  $(e[1][n])$  as expected comparisons.
4. Experiment with Knuth optimization when the quadrangle inequality holds to reduce time toward  $(O(n^2))$ .
5. Stress test with skewed distributions where one key dominates.

## Test Cases

| Keys   | Model          | Params                                       | Expected behavior                                    |
|--------|----------------|----------------------------------------------|------------------------------------------------------|
| 3 keys | simplified     | $(f=[0.3,0.2,0.5])$                          | root tends to be key 3 or 1 depending on split costs |
| 4 keys | simplified     | $(f=[1,1,1,1])$                              | more balanced tree wins                              |
| 3 keys | full $((p,q))$ | $(p=[0.3,0.2,0.4], q=[0.02,0.02,0.03,0.03])$ | shifts the optimal root                              |
| 1 key  | either         | single freq                                  | cost equals $(p_1)$ or $(q_0 + p_1 + q_1)$           |

## Complexity

- Time:  $(O(n^3))$  with the naive triple loop over  $((i,j,r))$
- Space:  $(O(n^2))$  for DP and root tables

The Optimal BST DP captures a universal pattern: choose a split, add a per-interval penalty that reflects depth inflation, and combine optimal subtrees for the minimal expected search cost.

## 435 Polygon Triangulation

The Polygon Triangulation problem is a foundational geometric DP challenge. Given a convex polygon, the task is to divide it into non-overlapping triangles by drawing non-intersecting diagonals, minimizing the total weight—often the sum of triangle areas or edge lengths. It's structurally similar to Matrix Chain Multiplication, with intervals, splits, and additive costs.

### What Problem Are We Solving?

Given a convex polygon with vertices ( $V_0, V_1, \dots, V_{n-1}$ ), we want to triangulate it (partition into triangles using diagonals) such that the total cost is minimized.

Define cost as:

$$\text{cost}(i, j, k) = \text{weight of triangle } (V_i, V_j, V_k)$$

where weight could be:

- Area
- Perimeter
- Squared edge length sum (for generality)

We define the DP state:

$$dp[i][j] = \text{minimum triangulation cost between } V_i \text{ and } V_j$$

Recurrence:

$$dp[i][j] = \min_{i < k < j} (dp[i][k] + dp[k][j] + \text{cost}(i, j, k))$$

Base case:

$$dp[i][i+1] = 0$$

Final answer: ( $dp[0][n-1]$ )

### How Does It Work (Plain Language)

You pick a vertex ( $k$ ) between ( $i$ ) and ( $j$ ) to form a triangle ( $(V_i, V_k, V_j)$ ). This triangle splits the polygon into two smaller polygons:

- One from ( $V_i$ ) to ( $V_k$ )
- Another from ( $V_k$ ) to ( $V_j$ )

We recursively find their optimal triangulations and add the triangle's cost.

This is a divide-and-conquer on geometry. Every choice of diagonal corresponds to a split in the DP.

## Tiny Code (Easy Versions)

C (Using area as cost)

```
#include <stdio.h>
#include <math.h>
#include <float.h>

#define MAX 105
#define min(a,b) ((a)<(b)?(a):(b))

typedef struct { double x, y; } Point;

double dist(Point a, Point b) {
    double dx = a.x - b.x, dy = a.y - b.y;
    return sqrt(dx*dx + dy*dy);
}

double triangle_cost(Point a, Point b, Point c) {
    double s = (dist(a,b) + dist(b,c) + dist(c,a)) / 2.0;
    double area = sqrt(s * (s - dist(a,b)) * (s - dist(b,c)) * (s - dist(c,a)));
    return area;
}

int main(void) {
    int n;
    printf("Enter number of vertices: ");
    scanf("%d", &n);
    Point v[MAX];
    for (int i = 0; i < n; i++)
        scanf("%lf %lf", &v[i].x, &v[i].y);

    double dp[MAX][MAX];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            dp[i][j] = 0.0;

    for (int len = 2; len < n; len++) {
        for (int i = 0; i + len < n; i++) {
            int j = i + len;
            dp[i][j] = DBL_MAX;
            for (int k = i + 1; k < j; k++) {
                double cost = dp[i][k] + dp[k][j] + triangle_cost(v[i], v[j], v[k]);
                if (cost < dp[i][j])
                    dp[i][j] = cost;
            }
        }
    }
}
```

```

        dp[i][j] = min(dp[i][j], cost);
    }
}
}

printf("Min triangulation cost: %.4f\n", dp[0][n-1]);
return 0;
}

```

Python (Perimeter cost)

```

import math

def dist(a, b):
    return math.hypot(a[0] - b[0], a[1] - b[1])

def triangle_cost(a, b, c):
    return dist(a,b) + dist(b,c) + dist(c,a)

n = int(input("Enter number of vertices: "))
v = [tuple(map(float, input().split())) for _ in range(n)]
dp = [[0]*n for _ in range(n)]

for length in range(2, n):
    for i in range(n - length):
        j = i + length
        dp[i][j] = float('inf')
        for k in range(i+1, j):
            dp[i][j] = min(dp[i][j],
                           dp[i][k] + dp[k][j] + triangle_cost(v[i], v[k], v[j]))

print("Min triangulation cost:", round(dp[0][n-1], 4))

```

## Why It Matters

- Canonical geometric DP using intervals and triple splits
- Underpins graphics, meshing, computational geometry, and 3D modeling
- Shows that Matrix Chain Multiplication and Polygon Triangulation share a structural template
- Reinforces how spatial reasoning maps to recurrence formulation

## Step-by-Step Example

Consider a quadrilateral with vertices:

$$V_0 = (0, 0), ; V_1 = (1, 0), ; V_2 = (1, 1), ; V_3 = (0, 1)$$

Two triangulations:

1. Diagonal  $(V_0V_2)$ : triangles  $((V_0, V_1, V_2))$  and  $((V_0, V_2, V_3))$
2. Diagonal  $(V_1V_3)$ : triangles  $((V_0, V_1, V_3))$  and  $((V_1, V_2, V_3))$

Both give same area ( $=1$ ). DP would compute both and take the minimum.

## A Gentle Proof (Why It Works)

Every triangulation must include exactly  $(n-3)$  diagonals. Fixing a triangle  $((V_i, V_k, V_j))$  that uses diagonal  $(V_iV_j)$  partitions the polygon into two smaller convex polygons. Since subproblems do not overlap except at the boundary, their optimal solutions combine to the global optimum. By evaluating all  $(k)$  between  $(i)$  and  $(j)$ , we guarantee we find the optimal split. The recurrence enumerates all possible triangulations implicitly.

## Try It Yourself

1. Change the cost function to perimeter instead of area.
2. Print the sequence of triangles chosen by storing split points.
3. Visualize the triangulation order in 2D.
4. Compare complexity vs brute force enumeration.
5. Implement in memoized recursion style.

## Test Cases

| Vertices                                                                     | Expected Min Cost (Area) |
|------------------------------------------------------------------------------|--------------------------|
| Square $(0,0), (1,0), (1,1), (0,1)$                                          | 1.0                      |
| Triangle $(0,0), (1,0), (0,1)$                                               | 0.5                      |
| Pentagon $(0,0), (1,0), (2,1), (1,2), (0,1)$<br>$(0,0), (2,0), (2,2), (0,2)$ | Varies with shape        |

## Complexity

- Time:  $O(n^3)$  (triply nested loop)
- Space:  $O(n^2)$  (DP table)

Polygon Triangulation is the geometric twin of matrix-chain optimization, same recurrence, new meaning.

## 436 Matrix Path Sum

The Matrix Path Sum problem asks for a path from the top-left to the bottom-right of a grid that optimizes a score, typically the minimum total sum of visited cells, moving only right or down.

### What Problem Are We Solving?

Given an  $m \times n$  matrix  $A$  of integers, find the minimum cost to go from  $(0, 0)$  to  $(m - 1, n - 1)$  using moves , right, down,.

State:

$$dp[i][j] = \text{minimum path sum to reach cell } (i, j)$$

Recurrence:

$$dp[i][j] = A[i][j] + \min(dp[i - 1][j], dp[i][j - 1])$$

Borders:

$$dp[0][0] = A[0][0], \quad dp[i][0] = A[i][0] + dp[i - 1][0], \quad dp[0][j] = A[0][j] + dp[0][j - 1]$$

Answer:

$$dp[m - 1][n - 1]$$

Space optimization: keep one row

$$\text{row}[j] = A[i][j] + \min(\text{row}[j], \text{row}[j - 1])$$

### How Does It Work (Plain Language)

Each cell's best cost equals its value plus the better of the two ways you could have arrived: from above or from the left. Build the table row by row until the destination cell is filled.

Example:

$$A = [1 \ 3 \ 1 \ 1 \ 5 \ 1 \ 4 \ 2 \ 1]$$

Optimal sum is  $1 + 1 + 3 + 1 + 1 = 7$  via path  $(0, 0) \rightarrow (1, 0) \rightarrow (1, 1) \rightarrow (1, 2) \rightarrow (2, 2)$ .

## Tiny Code (Easy Versions)

C (2D DP)

```
#include <stdio.h>
#define MIN(a,b) ((a)<(b)?(a):(b))

int main(void) {
    int m, n;
    scanf("%d %d", &m, &n);
    int A[m][n], dp[m][n];
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
            scanf("%d", &A[i][j]);

    dp[0][0] = A[0][0];
    for (int i = 1; i < m; i++) dp[i][0] = A[i][0] + dp[i-1][0];
    for (int j = 1; j < n; j++) dp[0][j] = A[0][j] + dp[0][j-1];

    for (int i = 1; i < m; i++)
        for (int j = 1; j < n; j++)
            dp[i][j] = A[i][j] + MIN(dp[i-1][j], dp[i][j-1]);

    printf("%d\n", dp[m-1][n-1]);
    return 0;
}
```

Python (1D Space Optimized)

```
m, n = map(int, input().split())
A = [list(map(int, input().split())) for _ in range(m)]

row = [0]*n
row[0] = A[0][0]
for j in range(1, n):
    row[j] = row[j-1] + A[0][j]

for i in range(1, m):
    row[0] += A[i][0]
    for j in range(1, n):
        row[j] = A[i][j] + min(row[j], row[j-1])

print(row[-1])
```

## Why It Matters

- Core template for grid DP and shortest path on DAGs
- Basis for image seam carving, robot motion on grids, spreadsheet cost flows
- Demonstrates classic DP patterns: table fill, border initialization, and space rolling

## Step-by-Step Example

For

$$A = [5 \ 1 \ 3 \ 2 \ 8 \ 1 \ 4 \ 2 \ 1]$$

compute row by row:

- First row  $dp$ :  $[5, 6, 9]$
- First column  $dp$ :  $[5, 7, 11]$
- Fill inner:
  - $dp[1][1] = 8 + \min(6, 7) = 14$
  - $dp[1][2] = 1 + \min(9, 14) = 10$
  - $dp[2][1] = 2 + \min(14, 11) = 13$
  - $dp[2][2] = 1 + \min(10, 13) = 11$

Answer = 11.

## A Gentle Proof (Why It Works)

Any optimal path to  $(i, j)$  must come from either  $(i - 1, j)$  or  $(i, j - 1)$  because moves are only right or down. If a cheaper route existed that did not pass through the cheaper of these two, replacing the prefix with the cheaper subpath would reduce the total cost, which is a contradiction. Hence the local recurrence using the minimum of top and left yields the global optimum when filled in topological order.

## Try It Yourself

1. Reconstruct the path: keep a parent pointer from each  $(i, j)$  to  $\text{argmin}$  of top or left.
2. Maximize sum instead of minimize by changing min to max.
3. Add obstacles: mark blocked cells with  $+\infty$  and skip them.
4. Allow moves right, down, and diagonal down-right. Extend the recurrence to three predecessors.
5. Use large matrices and compare 2D DP vs 1D rolling performance.

## Test Cases

| Matrix                      | Expected |
|-----------------------------|----------|
| $[[1,3,1],[1,5,1],[4,2,1]]$ | 7        |
| $[[5]]$                     | 5        |
| $[[1,2,3],[4,5,6]]$         | 12       |
| $[[1,1,1],[1,1,1],[1,1,1]]$ | 5        |
| $[[5,1,3],[2,8,1],[4,2,1]]$ | 11       |

## Complexity

- Time:  $O(mn)$
- Space:  $O(mn)$  with full table or  $O(n)$  with row rolling

Matrix Path Sum is the go-to pattern for grid costs: initialize borders, sweep the table, and each cell is its value plus the best way to arrive.

## 437 Largest Square Submatrix

The Largest Square Submatrix problem asks for the size of the biggest square of 1s in a binary matrix. It's a staple 2D DP problem, each cell's value tells how large a square can end at that position.

### What Problem Are We Solving?

Given a binary matrix  $A$  of size  $m \times n$ , find the side length of the largest all-1s square.

We define

$dp[i][j] = \text{side length of the largest square whose bottom right corner is at } (i, j).$

Recurrence:

$$dp[i][j] = \begin{cases} A[i][j], & i = 0 \text{ or } j = 0, \\ 0, & A[i][j] = 0, \\ \min\{dp[i-1][j], dp[i][j-1], dp[i-1][j-1]\} + 1, & A[i][j] = 1. \end{cases}$$

Answer:

$$\max_{i,j} dp[i][j].$$

## How Does It Work (Plain Language)

Each cell says: “*How big a square of 1s can I end?*” If it’s a 1, we look up, left, and up-left, the smallest of those tells how big a square we can extend.

Intuition:

- A square can grow only if all its borders can.
- So a 1 at ((i,j)) grows a square of size  $1 + \min(\text{three neighbors})$ .

Example:

```
1 0 1 0 0
1 0 1 1 1
1 1 1 1 1
1 0 0 1 0
```

The largest square has side length 3.

## Tiny Code (Easy Versions)

C

```
#include <stdio.h>
#define MAX(a,b) ((a)>(b)?(a):(b))
#define MIN(a,b) ((a)<(b)?(a):(b))

int main(void) {
    int m, n;
    scanf("%d %d", &m, &n);
    int A[m][n], dp[m][n];
    for (int i=0; i<m; i++)
        for (int j=0; j<n; j++)
            scanf("%d", &A[i][j]);

    int max_side = 0;
    for (int i=0; i<m; i++) {
        for (int j=0; j<n; j++) {
            if (A[i][j] == 0) dp[i][j] = 0;
            else if (i==0 || j==0) dp[i][j] = 1;
            else dp[i][j] = MIN(MIN(dp[i-1][j], dp[i][j-1]), dp[i-1][j-1]) + 1;
            if (dp[i][j] > max_side) max_side = dp[i][j];
        }
    }
}
```

```

        }
    }
    printf("%d\n", max_side);
    return 0;
}

```

Python

```

m, n = map(int, input().split())
A = [list(map(int, input().split())) for _ in range(m)]

dp = [[0]*n for _ in range(m)]
max_side = 0

for i in range(m):
    for j in range(n):
        if A[i][j] == 1:
            if i == 0 or j == 0:
                dp[i][j] = 1
            else:
                dp[i][j] = min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]) + 1
            max_side = max(max_side, dp[i][j])

print(max_side)

```

## Why It Matters

- Foundation of 2D DP problems (local recurrence on neighbors).
- Directly extends to rectangles, histograms, obstacles, or weighted cells.
- Used in image processing, pattern detection, bioinformatics grids.
- Demonstrates how spatial structure can be captured by overlapping subproblems.

## Step-by-Step Example

Matrix:

$$A = [1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 1]$$

Fill DP:

| A | DP |
|---|----|
| 1 | 1  |
| 1 | 1  |
| 1 | 1  |

Row 2:

- $dp[1][1] = \min(1, 1, 1) + 1 = 2$
- $dp[1][2] = \min(1, 1, 1) + 1 = 2$

Row 3:

- $dp[2][2] = \min(2, 2, 1) + 1 = 2$

Max = 2  $\Rightarrow$  square side = 2

### A Gentle Proof (Why It Works)

A cell ((i,j)) can be the bottom-right corner of a square of side (k) iff:

- The cells directly above, left, and diagonal up-left can each form a square of side (k-1). So  $(dp[i][j])$  is the largest (k) satisfying these. Filling top-down ensures each needed neighbor is ready, and taking the min keeps the square aligned.

### Try It Yourself

1. Modify to return area instead of side length.
2. Handle obstacles (cells with -1) as blocked.
3. Adapt to maximum rectangle of 1s (hint: histogram DP per row).
4. Output coordinates of top-left cell of the largest square.
5. Compare time between full and 1D-rolled DP.

### Test Cases

| Matrix                    | Result |
|---------------------------|--------|
| [[1,1,1],[1,1,1],[1,1,1]] | 3      |
| [[0,1],[1,1]]             | 2      |
| [[1,0,1],[1,1,1],[1,1,0]] | 2      |
| [[1]]                     | 1      |
| [[0,0],[0,0]]             | 0      |

| Matrix | Result |
|--------|--------|
|        |        |

## Complexity

- Time:  $(O(mn))$
- Space:  $(O(mn))$  or  $(O(n))$  with rolling rows

This problem is a clean showcase of local DP propagation, each cell grows the memory of its best square from three neighbors.

## 438 Max Rectangle in Binary Matrix

The Max Rectangle in Binary Matrix problem asks for the area of the largest rectangle containing only 1s in a binary matrix. It's a powerful combination of 2D DP and stack-based histogram algorithms, every row is treated as the base of a histogram, and we compute the largest rectangle there.

### What Problem Are We Solving?

Given a binary matrix  $A$  of size  $m \times n$ , find the maximum rectangular area consisting entirely of 1s.

Interpret each row as a histogram of heights and update per row:

$$\text{height}[j] = \begin{cases} \text{height}[j] + 1, & A[i][j] = 1, \\ 0, & A[i][j] = 0. \end{cases}$$

At each row, compute the Largest Rectangle in Histogram on height.

Equivalent 2D recurrence for heights:

$$\text{height}[i][j] = \begin{cases} A[i][j] + \text{height}[i - 1][j], & A[i][j] = 1, \\ 0, & A[i][j] = 0. \end{cases}$$

Answer:

$$\max_i \text{LargestRectangle}(\text{height}[i]).$$

## How Does It Work (Plain Language)

Think of the matrix as stacked histograms:

- Each row builds on top of the one above.
- A 1 extends the height; a 0 resets it.
- For each row, we ask: “What’s the largest rectangle if this row were the bottom?”

This converts a 2D problem into ( $m$ ) histogram problems.

Example:

```
1 0 1 0 0
1 0 1 1 1
1 1 1 1 1
1 0 0 1 0
```

The largest rectangle of 1s has area 6.

## Tiny Code (Easy Versions)

C (Using Stack for Histogram)

```
#include <stdio.h>
#define MAX 205
#define MAX2(a,b) ((a)>(b)?(a):(b))

int largest_histogram(int *h, int n) {
    int stack[MAX], top = -1, maxA = 0;
    for (int i = 0; i <= n; i++) {
        int cur = (i == n ? 0 : h[i]);
        while (top >= 0 && h[stack[top]] >= cur) {
            int height = h[stack[top--]];
            int width = (top < 0 ? i : i - stack[top] - 1);
            int area = height * width;
            if (area > maxA) maxA = area;
        }
        stack[++top] = i;
    }
    return maxA;
}
```

```

int main(void) {
    int m, n;
    scanf("%d %d", &m, &n);
    int A[MAX][MAX];
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
            scanf("%d", &A[i][j]);

    int height[MAX] = {0}, maxArea = 0;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++)
            height[j] = A[i][j] ? height[j] + 1 : 0;
        int area = largest_histogram(height, n);
        if (area > maxArea) maxArea = area;
    }
    printf("%d\n", maxArea);
    return 0;
}

```

Python (Stack-Based)

```

def largest_histogram(h):
    stack, maxA = [], 0
    h.append(0)
    for i, x in enumerate(h):
        while stack and h[stack[-1]] >= x:
            height = h[stack.pop()]
            width = i if not stack else i - stack[-1] - 1
            maxA = max(maxA, height * width)
        stack.append(i)
    h.pop()
    return maxA

m, n = map(int, input().split())
A = [list(map(int, input().split())) for _ in range(m)]
height = [0]*n
maxA = 0

for i in range(m):
    for j in range(n):
        height[j] = height[j] + 1 if A[i][j] else 0
    maxA = max(maxA, largest_histogram(height))

```

```
print(maxA)
```

## Why It Matters

- Core of 2D maximal area problems
- Connects DP (height propagation) and stack algorithms (histogram)
- Used in image segmentation, pattern recognition, binary masks
- Template for maximal submatrix under constraints

## Step-by-Step Example

For:

$$A = [1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1]$$

Row 1: [1,1,1] → largest histogram = 3 Row 2: [2,2,2] → largest histogram = 6 Row 3: [3,0,3] → largest histogram = 3 Max = 6

## A Gentle Proof (Why It Works)

Each rectangle in the matrix can be identified by its bottom row and column range. The height array at row (i) encodes exactly the number of consecutive 1s above each column, including row (i). Thus every maximal rectangle's bottom row is considered once, and the largest histogram algorithm ensures that for each height combination, the maximal area is found. Therefore, iterating all rows yields the global optimum.

## Try It Yourself

1. Modify to return coordinates of top-left and bottom-right corners.
2. Extend to max rectangle of 0s by flipping bits.
3. Compare to Largest Square Submatrix, same idea, different recurrence.
4. Use rolling arrays for memory reduction.
5. Visualize histogram growth row by row.

| Matrix | Expected Max Area |
|--------|-------------------|
|--------|-------------------|

### Test Cases

| Matrix                                                         | Expected Max Area |
|----------------------------------------------------------------|-------------------|
| <code>[[1,0,1,0,0],[1,0,1,1,1],[1,1,1,1,1],[1,0,0,1,0]]</code> | 6                 |
| <code>[[1,1,1],[1,1,1],[1,1,1]]</code>                         | 9                 |
| <code>[[0,0],[0,0]]</code>                                     | 0                 |
| <code>[[1]]</code>                                             | 1                 |
| <code>[[1,0,1],[1,1,1],[1,1,0]]</code>                         | 4                 |

### Complexity

- Time: ( $O(mn)$ ) (each cell pushed/popped once across all rows)
- Space: ( $O(n)$ ) for histogram and stack

This problem elegantly layers row-wise DP and histogram optimization, a universal method for maximal rectangles in 2D grids.

## 439 Submatrix Sum Queries

The Submatrix Sum Queries problem asks for the sum of all elements inside many rectangular regions of a 2D array. With a 2D prefix sum DP table, each query can be answered in  $O(1)$  time after  $O(mn)$  preprocessing.

### What Problem Are We Solving?

Given an  $m \times n$  matrix  $A$  and many queries of the form  $(r_1, c_1, r_2, c_2)$  with  $0 \leq r_1 \leq r_2 < m$  and  $0 \leq c_1 \leq c_2 < n$ , compute:

$$\text{Sum}(r_1, c_1, r_2, c_2) = \sum_{i=r_1}^{r_2} \sum_{j=c_1}^{c_2} A[i][j]$$

Define the 2D prefix sum  $P$  using 1-based indexing:

$$P[i][j] = \sum_{x=1}^i \sum_{y=1}^j A[x-1][y-1], \quad 1 \leq i \leq m, \quad 1 \leq j \leq n$$

with  $P[0][*] = P[*][0] = 0$ .

Then any submatrix sum is:

$$S = P[r_2 + 1][c_2 + 1] - P[r_1][c_2 + 1] - P[r_2 + 1][c_1] + P[r_1][c_1]$$

### How Does It Work (Plain Language)

Precompute cumulative sums from the top-left corner. The sum of a rectangle is then the big prefix up to its bottom-right, minus the two prefixes above and left, plus back the overlap you subtracted twice. This is just inclusion-exclusion in 2D.

Example:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Query  $(r_1, c_1, r_2, c_2) = (1, 1, 2, 2)$  over

$$\begin{bmatrix} 5 & 6 \\ 8 & 9 \end{bmatrix}$$

gives  $5 + 6 + 8 + 9 = 28$ .

### Tiny Code (Easy Versions)

C (2D Prefix Sum, many queries in  $O(1)$  each)

```
#include <stdio.h>

int main(void) {
    int m, n, q;
    scanf("%d %d", &m, &n);
    long long A[m][n];
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
            scanf("%lld", &A[i][j]);

    long long P[m+1][n+1];
    for (int i = 0; i <= m; i++) P[i][0] = 0;
    for (int j = 0; j <= n; j++) P[0][j] = 0;
```

```

for (int i = 1; i <= m; i++) {
    long long rowsum = 0;
    for (int j = 1; j <= n; j++) {
        rowsum += A[i-1][j-1];
        P[i][j] = P[i-1][j] + rowsum;
    }
}

scanf("%d", &q);
while (q--) {
    int r1, c1, r2, c2;
    scanf("%d %d %d %d", &r1, &c1, &r2, &c2);
    long long S = P[r2+1][c2+1] - P[r1][c2+1] - P[r2+1][c1] + P[r1][c1];
    printf("%lld\n", S);
}
return 0;
}

```

Python (2D Prefix Sum)

```

m, n = map(int, input().split())
A = [list(map(int, input().split())) for _ in range(m)]

P = [[0]*(n+1) for _ in range(m+1)]
for i in range(1, m+1):
    rowsum = 0
    for j in range(1, n+1):
        rowsum += A[i-1][j-1]
        P[i][j] = P[i-1][j] + rowsum

q = int(input())
for _ in range(q):
    r1, c1, r2, c2 = map(int, input().split())
    S = P[r2+1][c2+1] - P[r1][c2+1] - P[r2+1][c1] + P[r1][c1]
    print(S)

```

## Why It Matters

- Turns many 2D range sum queries into  $O(1)$  time each after  $O(mn)$  preprocessing
- Fundamental for integral images, heatmaps, terrain elevation maps, and data analytics
- Building block for maximum submatrix sum, range average, and density queries

## Step-by-Step Example

Let

$$A = \begin{bmatrix} 2 & -1 & 3 \\ 0 & 4 & 5 \\ 7 & 2 & -6 \end{bmatrix}$$

Query  $(0, 1, 2, 2)$  covers the submatrix

$$\begin{bmatrix} -1 & 3 \\ 4 & 5 \\ 2 & -6 \end{bmatrix}$$

whose sum is

$$-1 + 3 + 4 + 5 + 2 - 6 = 7.$$

Check formula:

$$S = P[3][3] - P[0][3] - P[3][1] + P[0][1] = 12 - 4 - 7 + 6 = 7$$

## A Gentle Proof (Why It Works)

By definition,  $P[i][j] = \sum_{x \leq i} \sum_{y \leq j} A[x-1][y-1]$ .

For rectangle  $[r_1..r_2] \times [c_1..c_2]$ :

- $P[r_2+1][c_2+1]$ : total up to bottom-right
- subtract  $P[r_1][c_2+1]$ : remove rows above
- subtract  $P[r_2+1][c_1]$ : remove columns left
- add  $P[r_1][c_1]$ : restore overlap

Thus, the inclusion-exclusion identity holds.

## Try It Yourself

1. Extend to 3D prefix sums for cuboid queries
2. Support range average (divide sum by area)
3. Add modulo arithmetic for large sums
4. Handle sparse updates with a 2D Fenwick tree
5. Precompute prefix sum for probability maps or heat distributions

## Test Cases

| Matrix                        | Query ( $r_1, c_1, r_2, c_2$ ) | Expected |
|-------------------------------|--------------------------------|----------|
| $[[1,2],[3,4]]$               | (0,0,1,1)                      | 10       |
| $[[1,2,3],[4,5,6],[7,8,9]]$   | (1,1,2,2)                      | 28       |
| $[[2,-1,3],[0,4,5],[7,2,-6]]$ | (0,1,2,2)                      | 7        |
| $[[5]]$                       | (0,0,0,0)                      | 5        |
| $[[1,0,1],[0,1,0],[1,0,1]]$   | (0,0,2,2)                      | 5        |

## Complexity

- Preprocessing:  $O(mn)$
- Query:  $O(1)$
- Space:  $O(mn)$

2D prefix sums are a foundational DP tool: preprocess once, then every submatrix sum is instant.

## 440 Palindrome Partitioning

The Palindrome Partitioning problem asks you to divide a string into the fewest possible substrings such that each substring is a palindrome. It's a quintessential interval DP problem where we explore all split points, using precomputed palindrome checks to accelerate the recurrence.

### What Problem Are We Solving?

Given a string  $s$  of length  $n$ , find the minimum number of cuts needed so that every substring is a palindrome.

For example:  $s = "aab"$  The best partition is " $aa$ " | " $b$ ", needing 1 cut.

We define:

- $dp[i] = \text{minimum cuts needed for substring } s[0..i]$
- $pal[i][j] = 1 \text{ if } s[i..j] \text{ is palindrome, else } 0$

Recurrence:

$$dp[i] = \begin{cases} 0, & \text{if } pal[0][i] = 1, \\ \min_{0 \leq j < i, \ pal[j+1][i]=1} (dp[j] + 1), & \text{otherwise.} \end{cases}$$

Precompute  $pal[i][j]$  using:

$$pal[i][j] = (s[i] = s[j]) \wedge (j - i < 2 \vee pal[i + 1][j - 1])$$

Answer:  $dp[n - 1]$

### How Does It Work (Plain Language)

We want to cut the string at points where the right substring is a palindrome. For each index  $i$ , we find all  $j < i$  such that  $s[j + 1..i]$  is palindrome and take the minimum over  $dp[j] + 1$ .

To avoid  $O(n^3)$ , we first precompute  $pal[i][j]$  in  $O(n^2)$ .

Example:

```
s = "aab"
pal =
a a b
1 1 0
  1 0
    1
```

Cuts:

- $dp[0] = 0$  (a)
- $dp[1] = 0$  (aa)
- $dp[2] = 1$  (aa|b)  $\rightarrow$  answer = 1

### Tiny Code (Easy Versions)

C (Bottom-Up DP)

```

#include <stdio.h>
#include <string.h>
#include <limits.h>

#define MIN(a,b) ((a)<(b)?(a):(b))
#define MAXN 1005

int main(void) {
    char s[MAXN];
    scanf("%s", s);
    int n = strlen(s);
    int pal[MAXN][MAXN] = {0};
    int dp[MAXN];

    for (int i = 0; i < n; i++) pal[i][i] = 1;
    for (int len = 2; len <= n; len++) {
        for (int i = 0; i + len - 1 < n; i++) {
            int j = i + len - 1;
            if (s[i] == s[j]) {
                if (len == 2 || pal[i+1][j-1]) pal[i][j] = 1;
            }
        }
    }

    for (int i = 0; i < n; i++) {
        if (pal[0][i]) { dp[i] = 0; continue; }
        dp[i] = INT_MAX;
        for (int j = 0; j < i; j++) {
            if (pal[j+1][i] && dp[j] + 1 < dp[i])
                dp[i] = dp[j] + 1;
        }
    }

    printf("%d\n", dp[n-1]);
    return 0;
}

```

Python (Readable)

```

s = input().strip()
n = len(s)
pal = [[False]*n for _ in range(n)]

```

```

dp = [0]*n

for i in range(n):
    pal[i][i] = True
for length in range(2, n+1):
    for i in range(n-length+1):
        j = i + length - 1
        if s[i] == s[j] and (length == 2 or pal[i+1][j-1]):
            pal[i][j] = True

for i in range(n):
    if pal[0][i]:
        dp[i] = 0
    else:
        dp[i] = min(dp[j]+1 for j in range(i) if pal[j+1][i])

print(dp[-1])

```

## Why It Matters

- Teaches interval DP with string-based state
- Used in text segmentation, DNA sequence analysis, code parsing
- Builds intuition for partitioning problems and precomputation synergy

## Step-by-Step Example

$s = \text{"banana"}$

Palindromic substrings:

- Single letters
- "ana" at positions (1–3), (3–5)
- "anana" at (1–5)

Compute  $dp$ :

- $dp[0] = 0$
- $dp[1] = 1$  ( $b|a$ )
- $dp[2] = 1$  ( $ba|n$ )
- $dp[3] = 1$  ( $b|ana$ )
- $dp[4] = 2$  ( $b|an|an$ )
- $dp[5] = 1$  ( $b|anana$ ) → Answer = 1

## A Gentle Proof (Why It Works)

A valid partition ends at some position  $i$ . If  $s[j + 1..i]$  is palindrome, then the cost to partition up to  $i$  is  $dp[j] + 1$ . The optimal must choose the best such  $j$ . By precomputing all palindrome substrings, each  $dp[i]$  depends only on smaller indices, satisfying the principle of optimality.

## Try It Yourself

1. Return the actual partitions using a parent array.
2. Modify to count all partitions instead of minimizing.
3. Adapt to palindromic subsequences (different structure).
4. Visualize DP and palindrome tables side by side.
5. Benchmark naive vs precomputed palindrome approaches.

## Test Cases

| s         | Expected Cuts | Example Partition |
|-----------|---------------|-------------------|
| “aab”     | 1             | “aa” “b”          |
| “racecar” | 0             | “racecar”         |
| “banana”  | 1             | “b” “anana”       |
| “abc”     | 2             | “a” “b” “c”       |
| “a”       | 0             | “a”               |

## Complexity

- Time:  $O(n^2)$
- Space:  $O(n^2)$  (can reduce palindrome table)

Palindrome Partitioning is a model example of DP with precomputation, revealing how structure (palindromes) enables efficient segmentation.

# Section 45. Bitmask DP and Traveling Salesman

## 441 Traveling Salesman Problem (TSP), Bitmask DP (Held–Karp)

The Traveling Salesman Problem asks for the shortest tour that visits every city exactly once and returns to the start. With dynamic programming over subsets, we can solve it in  $O(n^2 2^n)$ , which is optimal up to polynomial factors for exact exponential algorithms.

### What Problem Are We Solving?

Given  $n$  cities and a distance matrix  $dist[i][j]$ , find the minimum tour length that starts at city 0, visits all cities once, and returns to 0.

State (Held–Karp):

- Let  $dp[mask][i]$  be the minimum cost to start at 0, visit exactly the set of cities in bitmask  $mask$  (where bit  $k$  set means city  $k$  is visited), and end at city  $i$ .
- Base:  $dp[1 \ll 0][0] = 0$

Transition:

$$dp[mask][i] = \min_{\substack{j \in mask, j \neq i}} (dp[mask \setminus i][j] + dist[j][i])$$

Answer:

$$\min_{i \neq 0} (dp[(1 \ll n) - 1][i] + dist[i][0])$$

Path reconstruction: store the predecessor for each  $(mask, i)$ .

## How Does It Work (Plain Language)

We build tours by growing the set of visited cities. For every subset and last city  $i$ , we ask:

“What was the best way to visit this subset without  $i$ , then hop from  $j$  to  $i$ ? ”

This reuses smaller subproblems to solve larger ones, until all cities are visited.

## Tiny Code (Easy Versions)

C (Bitmask DP with Reconstruction)

```
#include <stdio.h>
#include <limits.h>

#define MAXN 20
#define INF 1000000000

int dist[MAXN][MAXN];
int prevCity[1<<MAXN][MAXN];
int dp[1<<MAXN][MAXN];

int main(void) {
    int n;
    scanf("%d", &n);
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            scanf("%d", &dist[i][j]);

    int N = 1 << n;
    for (int m = 0; m < N; m++)
        for (int i = 0; i < n; i++)
            dp[m][i] = INF, prevCity[m][i] = -1;

    dp[1<<0][0] = 0;

    for (int mask = 0; mask < N; mask++) {
        for (int i = 0; i < n; i++) if (mask & (1<<i)) {
            int pmask = mask ^ (1<<i);
            if (pmask == 0) continue;
            for (int j = 0; j < n; j++) if (pmask & (1<<j)) {
                int val = dp[pmask][j] + dist[j][i];
                if (val < dp[mask][i]) {
                    dp[mask][i] = val;
                    prevCity[mask][i] = j;
                }
            }
        }
    }
}
```

```

        if (val < dp[mask][i]) {
            dp[mask][i] = val;
            prevCity[mask][i] = j;
        }
    }
}

int all = N - 1, best = INF, last = -1;
for (int i = 1; i < n; i++) {
    int val = dp[all][i] + dist[i][0];
    if (val < best) best = val, last = i;
}
printf("Min tour cost: %d\n", best);

int path[MAXN+1], cnt = n;
int mask = all, cur = last;
path[--cnt] = cur;
while (cur != 0) {
    int p = prevCity[mask][cur];
    mask ^= (1<<cur);
    cur = p;
    path[--cnt] = cur;
}
printf("Tour: ");
for (int i = 0; i < n; i++) printf("%d ", path[i]);
printf("0\n");
return 0;
}

```

Python (Compact)

```

import sys
INF = 1015

n = int(sys.stdin.readline())
dist = [list(map(int, sys.stdin.readline().split())) for _ in range(n)]

N = 1 << n
dp = [[INF]*n for _ in range(N)]
dp[1][0] = 0

```

```

for mask in range(N):
    if (mask & 1) == 0:
        continue
    for i in range(n):
        if (mask & (1<<i)) == 0:
            continue
        pmask = mask ^ (1<<i)
        if pmask == 0:
            continue
        for j in range(n):
            if (pmask & (1<<j)) == 0:
                continue
            dp[mask][i] = min(dp[mask][i], dp[pmask][j] + dist[j][i])

allmask = N - 1
ans = min(dp[allmask][i] + dist[i][0] for i in range(1, n))
print(ans)

```

## Why It Matters

- Canonical bitmask DP example
- Exact solution with best-known time complexity for general TSP
- Template for subset-state DP problems: assignment, routing, path cover, Steiner tree

## Step-by-Step Example

Suppose

$$dist = \begin{bmatrix} 0 & 10 & 15 & 20 \\ 10 & 0 & 35 & 25 \\ 15 & 35 & 0 & 30 \\ 20 & 25 & 30 & 0 \end{bmatrix}$$

The optimal tour is  $0 \rightarrow 1 \rightarrow 3 \rightarrow 2 \rightarrow 0$  with cost  $10 + 25 + 30 + 15 = 80$ .

## A Gentle Proof (Why It Works)

For any subset  $S$  containing 0 and endpoint  $i \in S$ , the optimal path visiting  $S$  and ending at  $i$  must come from some  $j \in S \setminus i$  visiting  $S \setminus i$  optimally and then edge  $j \rightarrow i$ .

Thus:

$$dp[S][i] = \min_{j \in S \setminus i} (dp[S \setminus i][j] + dist[j][i])$$

By processing subsets in increasing size, dependencies are always ready before use.

### Try It Yourself

1. Reconstruct the tour path
2. Add must-visit or forbidden cities
3. Run on  $n = 15$  and observe scaling
4. Adapt to asymmetric TSP ( $dist[i][j] \neq dist[j][i]$ )
5. Compare with brute-force  $O(n!)$

### Test Cases

| n | dist                                                  | Expected |
|---|-------------------------------------------------------|----------|
| 2 | [[0,5],[5,0]]                                         | 10       |
| 3 | [[0,1,10],[1,0,2],[10,2,0]]                           | 13       |
| 4 | [[0,10,15,20],[10,0,35,25],[15,35,0,30],[20,25,30,0]] | 80       |
| 4 | [[0,3,4,2],[3,0,1,5],[4,1,0,6],[2,5,6,0]]             | 12       |

### Complexity

- Time:  $O(n^2 2^n)$
- Space:  $O(n 2^n)$

Held–Karp DP is the foundation for exponential-time optimization over subsets, bridging combinatorial search and dynamic programming.

## 442 Subset DP (Over Subsets of States)

Subset DP is a powerful pattern where each DP state represents a subset of elements. It's used when problems depend on combinations of items, masks, or visited sets. You define transitions based on smaller subsets, building up to larger ones.

## What Problem Are We Solving?

We want to compute some function  $dp[S]$  over all subsets  $S$  of a universe of size  $n$  (where  $S$  is represented as a bitmask).

Each  $dp[S]$  depends on smaller subsets of  $S$ , typically by adding or removing one element at a time.

Common forms:

- Subset sums:  $dp[S] = \sum_{T \subseteq S} f[T]$
- Maximums over subsets:  $dp[S] = \max_{T \subseteq S} f[T]$
- Counting configurations:  $dp[S] = \sum_{i \in S} dp[S \setminus i]$

The key idea: use bit operations and iterate through submasks efficiently.

## How Does It Work (Plain Language)

Think of each subset as a state. For example, if  $n = 3$ , the subsets are:

| Mask | Binary | Subset  |
|------|--------|---------|
| 0    | 000    |         |
| 1    | 001    | {0}     |
| 2    | 010    | {1}     |
| 3    | 011    | {0,1}   |
| 4    | 100    | {2}     |
| 5    | 101    | {0,2}   |
| 6    | 110    | {1,2}   |
| 7    | 111    | {0,1,2} |

Transitions depend on the structure of the problem:

- Additive (sum over submasks)
- Combinational (merge results)
- Stepwise (add/remove one bit)

### Example 1: Sum Over Subsets (SOS DP)

We want  $F[S] = \sum_{T \subseteq S} A[T]$ . Naively  $O(3^n)$ , but SOS DP does it in  $O(n2^n)$ .

Algorithm:

```
for (int i = 0; i < n; i++)
    for (int mask = 0; mask < (1<<n); mask++)
        if (mask & (1<<i))
            F[mask] += F[mask ^ (1<<i)];
```

Each iteration adds contributions from subsets missing bit  $i$ .

### Example 2: Counting Paths on Subsets

Suppose we count Hamiltonian paths on subsets:

$$dp[S][i] = \sum_{j \in S, j \neq i} dp[S \setminus i][j]$$

with base  $dp[i][i] = 1$ .

Iterate all subsets, and for each subset and endpoint, sum over possible predecessors.

### Tiny Code (Easy Versions)

C (Sum Over Subsets Example)

```
#include <stdio.h>

int main(void) {
    int n = 3;
    int A[8] = {1, 2, 3, 4, 5, 6, 7, 8}; // arbitrary base values
    int F[8];
    for (int mask = 0; mask < (1<<n); mask++) F[mask] = A[mask];

    for (int i = 0; i < n; i++)
        for (int mask = 0; mask < (1<<n); mask++)
            if (mask & (1<<i))
                F[mask] += F[mask ^ (1<<i)];

    for (int mask = 0; mask < (1<<n); mask++)
```

```

    printf("F[%d] = %d\n", mask, F[mask]);

    return 0;
}

```

Python (Sum Over Subsets)

```

n = 3
A = [1,2,3,4,5,6,7,8]
F = A[:]
for i in range(n):
    for mask in range(1<<n):
        if mask & (1<<i):
            F[mask] += F[mask ^ (1<<i)]
print(F)

```

## Why It Matters

- Foundation for bitmask combinatorics
- Speeds up subset convolutions, inclusion-exclusion, and fast zeta transforms
- Essential in Steiner Tree DP, bitmask knapsack, and TSP variants

## Step-by-Step Example

Let  $A = [1, 2, 3, 4, 5, 6, 7, 8]$ ,  $n = 3$ .

We want  $F[S] = \sum_{T \subseteq S} A[T]$ .

- $F[0] = A[0] = 1$
- $F[1] = A[1] + A[0] = 3$
- $F[3] = A[3] + A[2] + A[1] + A[0] = 10$

At the end:  $F = [1, 3, 4, 10, 5, 12, 13, 36]$

## A Gentle Proof (Why It Works)

Each mask represents a subset  $S$ . When we iterate bit  $i$ , we add  $F[S \setminus i]$  to  $F[S]$  if  $i \in S$ . This propagates values from smaller subsets to larger ones, accumulating all submask contributions. The loop order ensures every submask is processed before supersets containing it.

## Try It Yourself

1. Implement Sum Over Supersets (SOS Superset) by flipping the condition.
2. Compute  $\max_{T \subseteq S} A[T]$  instead of sum.
3. Use subset DP to count number of ways to cover a set with given subsets.
4. Combine subset DP with bitcount(mask) to handle per-size transitions.
5. Visualize subset lattice as a hypercube traversal.

## Test Cases

| A                 | Expected F (Sum Over Subsets) |
|-------------------|-------------------------------|
| [1,2,3,4,5,6,7,8] | [1,3,4,10,5,12,13,36]         |
| [0,1,0,1,0,1,0,1] | [0,1,0,2,0,2,0,4]             |
| [1,0,0,0,0,0,0,0] | [1,1,1,1,1,1,1,1]             |

## Complexity

- Time:  $O(n2^n)$
- Space:  $O(2^n)$

Subset DP is a unifying pattern for problems on sets, once you see the bitmask, think “DP over subsets”.

## 443 Hamiltonian Path DP (State Compression)

The Hamiltonian Path DP problem asks for the shortest path that visits every vertex exactly once, without needing to return to the start. It’s a close sibling of the Traveling Salesman Problem (TSP) but without the final return edge. Using bitmask DP, we can solve it in  $O(n^2 2^n)$ .

### What Problem Are We Solving?

Given a complete or weighted directed graph with  $n$  vertices and a cost matrix  $dist[i][j]$ , find the minimum-cost path that visits all vertices exactly once.

We don’t need to return to the starting node (unlike TSP).

Define the DP state:

- $dp[mask][i]$ : the minimum cost to visit exactly the set of vertices in  $mask$  and end at vertex  $i$ .

Base case:

$$dp[1 \ll i][i] = 0 \quad \forall i$$

Transition:

$$dp[mask][i] = \min_{j \in mask, j \neq i} (dp[mask \setminus i][j] + dist[j][i])$$

Answer:

$$\min_i dp[(1 \ll n) - 1][i]$$

### How Does It Work (Plain Language)

Imagine we're constructing paths step by step:

1. Each mask represents which vertices we've already visited.
2. Each endpoint  $i$  means we finish the path at vertex  $i$ .
3. We build  $dp[mask][i]$  from smaller subsets by adding one last vertex  $i$ .

At each step, we check all  $j$  that could be the previous vertex in the path.

No need to add  $dist[i][start]$  because we don't return, it's a path, not a cycle.

### Tiny Code (Easy Versions)

C (Hamiltonian Path DP)

```
#include <stdio.h>
#include <limits.h>

#define MAXN 20
#define INF 1000000000

int dist[MAXN][MAXN];
int dp[1<<MAXN][MAXN];

int main(void) {
```

```

int n;
scanf("%d", &n);
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        scanf("%d", &dist[i][j]);

int N = 1 << n;
for (int mask = 0; mask < N; mask++)
    for (int i = 0; i < n; i++)
        dp[mask][i] = INF;

for (int i = 0; i < n; i++)
    dp[1<<i][i] = 0;

for (int mask = 0; mask < N; mask++) {
    for (int i = 0; i < n; i++) {
        if (!(mask & (1<<i))) continue;
        int pmask = mask ^ (1<<i);
        if (pmask == 0) continue;
        for (int j = 0; j < n; j++) {
            if !(pmask & (1<<j))) continue;
            int cost = dp[pmask][j] + dist[j][i];
            if (cost < dp[mask][i]) dp[mask][i] = cost;
        }
    }
}

int all = N - 1;
int best = INF;
for (int i = 0; i < n; i++)
    if (dp[all][i] < best)
        best = dp[all][i];

printf("Minimum Hamiltonian path cost: %d\n", best);
return 0;
}

```

Python (Compact Version)

```

INF = 1015
n = int(input())
dist = [list(map(int, input().split())) for _ in range(n)]

```

```

N = 1 << n
dp = [[INF]*n for _ in range(N)]
for i in range(n):
    dp[1<<i][i] = 0

for mask in range(N):
    for i in range(n):
        if not (mask & (1<<i)): continue
        pmask = mask ^ (1<<i)
        if pmask == 0: continue
        for j in range(n):
            if not (pmask & (1<<j)): continue
            dp[mask][i] = min(dp[mask][i], dp[pmask][j] + dist[j][i])

ans = min(dp[N-1][i] for i in range(n))
print(ans)

```

## Why It Matters

- Fundamental state compression DP pattern
- Useful when the problem involves visiting all nodes exactly once
- Core for path planning, ordering constraints, and bitmask search

## Step-by-Step Example

Let

$$dist = [0 \ 1 \ 4 \ 1 \ 0 \ 2 \ 4 \ 2 \ 0]$$

Paths:

- $0 \rightarrow 1 \rightarrow 2: 1 + 2 = 3$
- $0 \rightarrow 2 \rightarrow 1: 4 + 2 = 6$
- $1 \rightarrow 0 \rightarrow 2: 1 + 4 = 5$

So minimum path = 3.  $dp[(1 << 3) - 1] = dp[7] = [5, 3, 5]$ , min = 3.

## A Gentle Proof (Why It Works)

We apply optimal substructure:

For each subset  $S$  and endpoint  $i$ , the optimal Hamiltonian path to  $(S, i)$  must extend an optimal path to  $(S \setminus i, j)$  by one edge  $j \rightarrow i$ . This recurrence ensures no city is revisited and all are included once.

Each  $dp[mask][i]$  depends only on smaller masks, so it can be built bottom-up.

## Try It Yourself

1. Add start node constraint (fix path must start at 0).
2. Recover the actual path using a `parent` array.
3. Modify for maximum path (replace `min` with `max`).
4. Adapt for directed graphs with asymmetric costs.
5. Use bit tricks like `mask & -mask` to iterate bits efficiently.

## Test Cases

| n | dist                                        | Expected |
|---|---------------------------------------------|----------|
| 3 | $[[0,1,4],[1,0,2],[4,2,0]]$                 | 3        |
| 4 | $[[0,3,1,5],[3,0,6,7],[1,6,0,2],[5,7,2,0]]$ | 6        |
| 2 | $[[0,5],[5,0]]$                             | 5        |
| 3 | $[[0,9,9],[9,0,1],[9,1,0]]$                 | 10       |

## Complexity

- Time:  $O(n^2 2^n)$
- Space:  $O(n2^n)$

Hamiltonian Path DP is the core structure for problems involving traversal of all nodes exactly once, simple, powerful, and a template for countless variants.

## 444 Assignment Problem DP (Mask over Tasks)

The Assignment Problem asks for the minimum total cost to assign  $n$  workers to  $n$  tasks with each worker doing exactly one task and each task done by exactly one worker. Besides the Hungarian algorithm, a clean solution for small  $n$  is bitmask DP over subsets of tasks.

## What Problem Are We Solving?

Given a cost matrix  $C$  where  $C[i][j]$  is the cost for worker  $i$  to do task  $j$ , find the minimum cost perfect matching between workers and tasks.

State definition:

- Let  $mask$  encode which tasks have been taken.
- Let  $i = \text{popcount}(mask)$  be the number of already assigned workers, meaning we are about to assign worker  $i$ .

DP state:

$dp[mask] = \text{minimum total cost to assign the first } \text{popcount}(mask) \text{ workers to the set of tasks in } mask$

Transition:

$$dp[mask \cup j] = \min (dp[mask] + C[i][j]) \quad \text{for all tasks } j \notin mask, i = \text{popcount}(mask)$$

Base and answer:

$$dp[0] = 0, \quad \text{Answer} = dp[(1 \ll n) - 1]$$

## How Does It Work (Plain Language)

We process workers in order  $0, 1, \dots, n - 1$ . The bitmask tells which tasks are already used. For the next worker  $i$ , try assigning any free task  $j$ , add its cost, and carry the best. This builds up all partial matchings until all tasks are taken.

## Tiny Code (Easy Versions)

C (Bitmask DP)

```

#include <stdio.h>
#include <limits.h>

#define MAXN 20
#define INF 1000000000

int C[MAXN][MAXN];
int dp[1<<MAXN];

int main(void) {
    int n;
    scanf("%d", &n);
    for (int i=0;i<n;i++)
        for (int j=0;j<n;j++)
            scanf("%d",&C[i][j]);

    int N=1<<n;
    for (int m=0;m<N;m++) dp[m]=INF;
    dp[0]=0;

    for (int mask=0;mask<N;mask++) {
        int i = __builtin_popcount(mask);
        if (i>=n || dp[mask]==INF) continue;
        for (int j=0;j<n;j++) if (!(mask&(1<<j))) {
            int nmask = mask|(1<<j);
            if (dp[mask]+C[i][j] < dp[nmask])
                dp[nmask] = dp[mask]+C[i][j];
        }
    }
    printf("%d\n", dp[N-1]);
    return 0;
}

```

Python (Compact)

```

import sys
INF = 1015

n = int(sys.stdin.readline())
C = [list(map(int, sys.stdin.readline().split())) for _ in range(n)]

N = 1 << n

```

```

dp = [INF]*N
dp[0] = 0

for mask in range(N):
    i = bin(mask).count("1")
    if i >= n or dp[mask] == INF:
        continue
    for j in range(n):
        if (mask >> j) & 1:
            continue
        dp[mask | (1 << j)] = min(dp[mask | (1 << j)], dp[mask] + C[i][j])

print(dp[N-1])

```

## Why It Matters

- Canonical example of state compression DP on subsets
- Simple and reliable when  $n \leq 20$  or so
- Baseline to compare with Hungarian algorithm and min cost flow
- Template for richer constraints: forbidden pairs, bonuses, prerequisites

## Step-by-Step Example

Let

$$C = [9 \ 2 \ 7 \ 6 \ 4 \ 3 \ 5 \ 8 \ 1]$$

One optimal assignment is worker  $0 \rightarrow 1$  (2),  $1 \rightarrow 2$  (3),  $2 \rightarrow 0$  (5) for total  $2 + 3 + 5 = 10$ . The DP explores all masks, always extending by one free task for the next worker.

## A Gentle Proof (Why It Works)

Let  $S$  be a set of tasks and  $i = |S|$ . Any optimal partial assignment of the first  $i$  workers to  $S$  must end by assigning worker  $i - 1$  to some  $j \in S$ . Removing task  $j$  yields an optimal solution to  $(S \setminus j, i - 1)$  plus  $C[i - 1][j]$ . Reversing this gives the forward transition from  $dp[mask]$  to  $dp[mask \cup j]$  with  $i = \text{popcount}(mask)$ . Since each transition increases the mask, filling masks in increasing popcount topologically respects dependencies.

## Try It Yourself

1. Reconstruct the assignment by storing a parent task for each *mask*.
2. Add forbidden pairs by skipping those  $(i, j)$ .
3. Add a bonus matrix  $B$  and minimize  $\sum(C[i][j] - B[i][j])$ .
4. Handle rectangular  $n \times m$  by padding the smaller side with zero dummy costs.
5. Compare runtime against Hungarian on random instances.

## Test Cases

| $C$                                                                 | Expected min |
|---------------------------------------------------------------------|--------------|
| $\begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}$                      | 2            |
| $\begin{bmatrix} 9 & 2 & 7 \\ 6 & 4 & 3 \\ 5 & 8 & 1 \end{bmatrix}$ | 10           |
| $\begin{bmatrix} 4 & 1 & 3 \\ 2 & 0 & 5 \\ 3 & 2 & 2 \end{bmatrix}$ | 5            |
| [10]                                                                | 10           |

## Complexity

- Time:  $O(n^2 2^n)$
- Space:  $O(2^n)$

Bitmask DP for the assignment problem is a tidy blueprint: iterate masks, assign the next worker to a free task, and keep the cheapest extension until all tasks are taken.

## 445 Partition into Two Sets (Balanced Load)

The Partition Problem asks whether a given set of numbers can be split into two subsets with equal sum. In its optimization form, we aim to minimize the difference between subset sums. It's a classic subset DP example that models balanced workloads, resource allocation, and load balancing.

## What Problem Are We Solving?

Given an array  $A[0..n - 1]$ , partition it into two subsets  $S_1$  and  $S_2$  such that the difference of sums is minimized:

$$\text{minimize } |\text{sum}(S_1) - \text{sum}(S_2)|$$

Equivalently, find a subset with sum as close as possible to half the total:

$$\text{target} = \left\lfloor \frac{\sum A}{2} \right\rfloor$$

We use DP to find all achievable sums up to `target`.

Define boolean DP:

$$dp[i][s] = 1 \text{ if some subset of the first } i \text{ elements has sum } s$$

Transition:

$$dp[i][s] = dp[i - 1][s] \vee dp[i - 1][s - A[i - 1]] \quad \text{if } s \geq A[i - 1]$$

Answer:

Find largest  $s \leq \text{target}$  with  $dp[n][s] = 1$ , difference =  $\text{total} - 2s$

## How Does It Work (Plain Language)

Think of it as filling a knapsack of capacity `target`. Each item can either go into the subset or stay out. We try all combinations of sums up to half the total, beyond that, the second subset mirrors it. The closest sum to `target` yields the minimal difference.

## Tiny Code (Easy Versions)

C (Tabulation)

```

#include <stdio.h>
#include <string.h>
#define MAXN 100
#define MAXSUM 10000

int main(void) {
    int n, A[MAXN];
    scanf("%d", &n);
    int total = 0;
    for (int i = 0; i < n; i++) {
        scanf("%d", &A[i]);
        total += A[i];
    }

    int target = total / 2;
    int dp[MAXSUM + 1] = {0};
    dp[0] = 1;

    for (int i = 0; i < n; i++) {
        for (int s = target; s >= A[i]; s--) {
            if (dp[s - A[i]]) dp[s] = 1;
        }
    }

    int best = 0;
    for (int s = target; s >= 0; s--) {
        if (dp[s]) { best = s; break; }
    }

    int diff = total - 2 * best;
    printf("Minimal difference: %d\n", diff);
    return 0;
}

```

Python (Concise Version)

```

A = list(map(int, input().split()))
total = sum(A)
target = total // 2
dp = [False]*(target+1)
dp[0] = True

```

```

for x in A:
    for s in range(target, x-1, -1):
        if dp[s-x]:
            dp[s] = True

for s in range(target, -1, -1):
    if dp[s]:
        print("Minimal difference:", total - 2*s)
        break

```

## Why It Matters

- Models balanced partitioning of workloads, memory, or resources
- Foundation for Subset Sum and Knapsack problems
- Introduces boolean DP over sums, a crucial building block for combinatorial search

## Step-by-Step Example

Let  $A = [1, 6, 11, 5]$ . Total = 23, target = 11.

Feasible sums: 0, 1, 5, 6, 7, 11, 12, 16, 17, 18, 23

Best  $s = 11$ , minimal difference  $23 - 2 \cdot 11 = 1$ .

Partition: [11] and [1, 5, 6].

## A Gentle Proof (Why It Works)

Each element either belongs to subset  $S_1$  or  $S_2$ . Let  $s_1$  be sum of  $S_1$ ,  $s_2 = \text{total} - s_1$ . We want  $|s_1 - s_2|$  minimized  $\rightarrow |total - 2s_1|$ . By exploring all achievable sums  $s_1 \leq \text{total}/2$ , we find the  $s_1$  closest to half. Boolean DP tracks reachability using inclusion-exclusion transitions.

## Try It Yourself

1. Count number of balanced partitions (replace boolean with integer DP).
2. Add constraint “each subset must have at least  $k$  elements.”
3. Extend to multi-set partitions (3 or more subsets).
4. Visualize reachable sums as boolean array transitions.
5. Compare with brute-force subset enumeration.

## Test Cases

| A            | Minimal Difference | Partition      |
|--------------|--------------------|----------------|
| [1,6,11,5]   | 1                  | [11] [1,5,6]   |
| [3,1,4,2,2]  | 0                  | [3,2] [4,2]    |
| [1,2,7]      | 4                  | [7] [1,2]      |
| [2,2,2,2]    | 0                  | [2,2] [2,2]    |
| [10,20,15,5] | 0                  | [10,15] [20,5] |

## Complexity

- Time:  $O(n \cdot sum)$
- Space:  $O(sum)$

The Partition DP is a gentle bridge from Subset Sum to balanced optimization, teaching how combinatorial structure guides numerical state transitions.

## 446 Count Hamiltonian Cycles (Bitmask Enumeration)

The Hamiltonian Cycle Counting problem asks: given a graph, how many distinct Hamiltonian cycles (closed tours visiting each vertex exactly once) exist? Unlike the shortest-path variants, this version focuses on counting all possible cycles using bitmask DP.

### What Problem Are We Solving?

Given a graph  $G = (V, E)$  with  $|V| = n$ , count the number of distinct Hamiltonian cycles starting and ending at vertex 0 that visit every vertex exactly once.

We define:

- $dp[mask][i] =$  number of Hamiltonian paths starting at 0, visiting all vertices in  $mask$ , and ending at  $i$
- Base:  $dp[1 << 0][0] = 1$

Transition:

$$dp[mask][i] = \sum_{j \in mask, j \neq i, (j, i) \in E} dp[mask \setminus i][j]$$

Answer (number of cycles):

$$\sum_{i=1}^{n-1} dp[(1 \ll n) - 1][i] \text{ if } (i, 0) \in E$$

Each valid end vertex  $i$  must connect back to 0 to complete the cycle.

## How Does It Work (Plain Language)

We build all possible paths from vertex 0 that cover a subset of vertices and end at some  $i$ . For each step, we extend smaller paths by adding a new endpoint  $i$ . When all vertices are visited ( $mask = (1 \ll n) - 1$ ), we check which endpoints connect back to 0. Summing these gives the total number of Hamiltonian cycles.

This is similar to the Held–Karp DP, but the operation is addition (counting) instead of minimization.

## Tiny Code (Easy Versions)

C (Bitmask Counting DP)

```
#include <stdio.h>
#include <string.h>

#define MAXN 20

long long dp[1<<MAXN] [MAXN];
int adj[MAXN] [MAXN];

int main(void) {
    int n, m;
    scanf("%d %d", &n, &m);
    memset(adj, 0, sizeof(adj));
    for (int e = 0; e < m; e++) {
        int u, v;
        scanf("%d %d", &u, &v);
        adj[u] [v] = adj[v] [u] = 1;
    }
    dp[1] [0] = 1; // start at vertex 0

    for (int mask = 1; mask < (1 << n); mask++) {
        for (int i = 0; i < n; i++) {
            if ((mask & (1 << i)) == mask) {
                dp[mask] [i] = 0;
                for (int j = 0; j < n; j++) {
                    if (j != i && (mask & (1 << j)) == mask) {
                        dp[mask] [i] += dp[mask ^ (1 << j)] [j];
                    }
                }
            }
        }
    }
}
```

```

        if (!(mask & (1 << i))) continue;
        for (int j = 0; j < n; j++) {
            if (i == j || !(mask & (1 << j))) continue;
            if (adj[j][i])
                dp[mask][i] += dp[mask ^ (1 << i)][j];
        }
    }
}

long long total = 0;
int full = (1 << n) - 1;
for (int i = 1; i < n; i++) {
    if (adj[i][0]) total += dp[full][i];
}

printf("%lld\n", total / 2); // divide by 2 for undirected graphs
return 0;
}

```

Python (Compact Version)

```

n, m = map(int, input().split())
adj = [[0]*n for _ in range(n)]
for _ in range(m):
    u, v = map(int, input().split())
    adj[u][v] = adj[v][u] = 1

N = 1 << n
dp = [[0]*n for _ in range(N)]
dp[1][0] = 1

for mask in range(N):
    for i in range(n):
        if not (mask & (1<<i)): continue
        for j in range(n):
            if i==j or not (mask & (1<<j)): continue
            if adj[j][i]:
                dp[mask][i] += dp[mask ^ (1<<i)][j]

full = N-1
total = sum(dp[full][i] for i in range(1,n) if adj[i][0])
print(total // 2)

```

## Why It Matters

- Demonstrates DP counting over subsets
- Foundation for counting Hamiltonian paths, cycle covers, and tours
- Appears in graph enumeration, combinatorial design, and traveling salesman counting

## Step-by-Step Example

Let the graph be a square (4-cycle):

Vertices: 0, 1, 2, 3

Edges: (0, 1), (1, 2), (2, 3), (3, 0) and (0, 3), (3, 2), (2, 1), (1, 0)

All Hamiltonian cycles:

- $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 0$
- $0 \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow 0$

So total = 2.

The DP constructs these paths incrementally by adding one vertex at a time.

## A Gentle Proof (Why It Works)

Each DP state  $(mask, i)$  corresponds to unique partial paths visiting  $mask$  and ending at  $i$ . To form a path ending at  $i$ , we must come from  $j \in mask \setminus i$  with edge  $(j, i)$ . This ensures each path is counted exactly once.

At the full mask, we have all Hamiltonian paths starting at 0 and ending at  $i$ ; connecting  $i \rightarrow 0$  closes the cycle.

Divide by 2 for undirected graphs since each cycle is counted twice (once clockwise, once counterclockwise).

## Try It Yourself

1. Remove division by 2 for directed graphs.
2. Count Hamiltonian paths (no return edge).
3. Modify to track path sequences using parent arrays.
4. For large  $n$ , compare with inclusion-exclusion counting.
5. Implement memoized recursion (top-down version).

## Test Cases

| Graph                          | Expected          |
|--------------------------------|-------------------|
| Triangle ( $0 - 1 - 2 - 0$ )   | 1                 |
| Square ( $0 - 1 - 2 - 3 - 0$ ) | 2                 |
| Line ( $0 - 1 - 2$ )           | 0                 |
| Complete graph $K_4$           | $(4 - 1)!/2 = 3$  |
| $K_5$                          | $(5 - 1)!/2 = 12$ |

## Complexity

- Time:  $O(n^2 2^n)$
- Space:  $O(n2^n)$

Counting Hamiltonian cycles via DP elegantly blends subset enumeration with path counting, offering an exact combinatorial count for small graphs.

## 447 Steiner Tree DP

The Steiner Tree problem asks for the minimum-cost subgraph that connects a given set of terminals in a weighted graph. You may use extra non-terminal vertices (Steiner nodes) if that reduces total cost. The classic exact DP for small numbers of terminals is the Dreyfus–Wagner subset DP.

### What Problem Are We Solving?

Input: an undirected connected graph with nonnegative edge weights, and a terminal set  $T = t_1, \dots, t_k$ .

Goal: find a minimum-weight tree that connects all vertices in  $T$ .

We use a subset DP over terminals and a root vertex:

- Precompute all-pairs shortest paths  $dist[u][v]$ .
- DP state:  $dp[S][v] =$  minimum cost of a tree that connects all terminals in subset  $S \subseteq T$  and whose tree is rooted at vertex  $v$ .

Initialization:

$$dp[t_i][v] = dist[v][t_i]$$

Combine subsets at the same root:

$$dp[S][v] = \min_{\emptyset \neq A \subset S} (dp[A][v] + dp[S \setminus A][v])$$

Then allow the root to move via shortest paths:

$$dp[S][v] = \min_u (dp[S][u] + dist[u][v])$$

Answer:

$$\min_v dp[T][v]$$

In practice we alternate subset-combine at fixed  $v$  and a multi-source shortest-path relaxation for each  $S$ .

### How Does It Work (Plain Language)

Think of building a Steiner tree by gluing together optimal trees for smaller terminal subsets at a meeting vertex  $v$ . After merging, you are allowed to slide that meeting point anywhere in the graph using shortest paths. Repeat for all subsets until you cover all terminals.

### Tiny Code (Easy Version, Python)

Dreyfus–Wagner with Floyd–Warshall for  $dist$  and Dijkstra-based relax per subset. Works for small  $k$ .

```
import heapq

INF = 1015

def floyd_marshall(n, w):
    dist = [row[:] for row in w]
    for i in range(n):
        dist[i][i] = min(dist[i][i], 0)
    for k in range(n):
        for i in range(n):
            dik = dist[i][k]
            if dik == INF: continue
            rowi = dist[i]
            for j in range(n):
                if dist[i][j] >= dist[i][k] + dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]
                    heapq.heappush(rowj, (dist[i][j], i))
```

```

        rowk = dist[k]
        for j in range(n):
            nd = dik + rowk[j]
            if nd < rowi[j]: rowi[j] = nd
    return dist

def steiner_tree(n, edges, terminals):
    # Build dense weight matrix
    w = [[INF]*n for _ in range(n)]
    for u in range(n):
        w[u][u] = 0
    for u, v, c in edges:
        if c < w[u][v]:
            w[u][v] = w[v][u] = c

    dist = floyd_marshall(n, w)

    k = len(terminals)
    term_index = {t:i for i,t in enumerate(terminals)}

    # dp[mask][v]
    dp = [[INF]*n for _ in range(1<<k)]
    for t in terminals:
        m = 1 << term_index[t]
        for v in range(n):
            dp[m][v] = dist[v][t]

    # subset DP
    for mask in range(1, 1<<k):
        # combine proper nonempty A subset at same root v
        sub = (mask-1) & mask
        while sub:
            other = mask ^ sub
            if other:
                for v in range(n):
                    val = dp[sub][v] + dp[other][v]
                    if val < dp[mask][v]:
                        dp[mask][v] = val
            sub = (sub-1) & mask

    # root-move relaxation by Dijkstra over complete graph with dist metric
    # This is equivalent to: dp[mask] = metric-closure shortest-path from sources with p

```

```

# Implement 1 run of Dijkstra with initial costs dp[mask] [*]
pq = [(dp[mask][v], v) for v in range(n)]
heapq.heapify(pq)
seen = [False]*n
while pq:
    d, v = heapq.heappop(pq)
    if seen[v]: continue
    seen[v] = True
    if d > dp[mask][v]: continue
    row = dist[v]
    for u in range(n):
        nd = d + row[u]
        if nd < dp[mask][u]:
            dp[mask][u] = nd
            heapq.heappush(pq, (nd, u))

full = (1<<k) - 1
return min(dp[full])

# Example usage:
# n = 5
# edges = [(0,1,1),(1,2,1),(2,3,1),(3,4,1),(0,4,10),(1,4,2)]
# terminals = [0,3,4]
# print(steiner_tree(n, edges, terminals))

```

Notes:

- For sparse graphs you can skip Floyd–Warshall and run Dijkstra inside the relax step using the original adjacency. Using the metric closure as above is simple and correct for nonnegative weights.

## Why It Matters

- Exact algorithm for Steiner trees when the number of terminals  $k$  is small
- Standard approach in VLSI routing, network design, and phylogenetics
- Teaches a powerful pattern: subset merge at a root plus shortest-path relaxation

## Step-by-Step Example

Suppose a 5-vertex path  $0-1-2-3-4$  with unit weights and terminals  $T = 0, 3, 4$ .

- Singletons:  $dp[0][v] = dist[v][0]$ , etc.
- Combine 3 and 4 at  $v = 3$  or  $v = 4$ , then relax along the path.
- Finally combine with 0; the optimal tree is edges  $(0, 1), (1, 2), (2, 3), (3, 4)$  with total cost 4.

### A Gentle Proof (Why It Works)

Let  $S \subseteq T$  and  $v$  be a meeting vertex of an optimal Steiner tree for  $S$ . The tree decomposes into two subtrees whose terminal sets partition  $S$ , and both subtrees meet at  $v$ . Hence

$$dp[S][v] \leq dp[A][v] + dp[S \setminus A][v].$$

Conversely, any combination at  $v$  plus a shortest-path relocation of  $v$  to another vertex is no worse than explicitly wiring with edges, due to triangle inequality from the metric closure. Induction over  $|S|$  yields optimality.

### Try It Yourself

1. Replace the Dijkstra relax with Bellman–Ford to allow zero edges with tight potentials.
2. Extract the actual Steiner tree: store the best split and predecessor during relaxation.
3. Compare metric-closure method vs relaxing on the original sparse graph.
4. Benchmark vs MST over terminals to see the benefit of Steiner vertices.
5. Add a constraint that certain vertices are forbidden as Steiner nodes.

### Test Cases

| Graph                                               | Terminals | Expected Steiner cost |
|-----------------------------------------------------|-----------|-----------------------|
| Path 0–1–2–3–4 with unit edges                      | {0,3,4}   | 4                     |
| Triangle 0–1–2 with unit edges                      | {0,2}     | 1                     |
| Square 0–1–2–3 with unit edges, diagonal 1–3 cost 1 | {0,2,3}   | 2                     |
| Star center 0 to 1..4 all cost 1                    | {1,2,3,4} | 4                     |

### Complexity

- With metric closure and subset merge:  $O(3^k \cdot n + 2^k \cdot n \log n)$  typical
- Memory:  $O(2^k \cdot n)$

The Dreyfus–Wagner DP is the go-to exact method when  $k$  is small: combine subsets at a root, then relax through shortest paths to let Steiner nodes emerge automatically.

## 448 SOS DP (Sum Over Subsets)

Sum Over Subsets (SOS DP) is a clever bitmask dynamic programming technique for computing values aggregated over all subsets of each mask efficiently, without enumerating all subset pairs explicitly.

### What Problem Are We Solving?

Suppose you have an array  $f[\text{mask}]$  defined for all bitmasks of length  $n$ , and you want to compute:

$$g[\text{mask}] = \sum_{\text{sub} \subseteq \text{mask}} f[\text{sub}]$$

Naively, this requires iterating over all subsets of each mask, which takes  $O(3^n)$ . With SOS DP, we can compute all  $g[\text{mask}]$  in  $O(n2^n)$  time.

### How Does It Work (Plain Language)

Think of each bit in the mask as a “dimension.” For each bit position  $i$ , if that bit is set in the mask, we can inherit contributions from the version where the bit was off. We build up sums by iterating over each bit dimension and folding smaller subsets upward.

### Transition Formula

Let  $\text{dp}[\text{mask}]$  initially equal  $f[\text{mask}]$ . Then for each bit  $i$  from 0 to  $n-1$ :

```
if mask has bit i set:  
    dp[mask] += dp[mask ^ (1 << i)]
```

After processing all bits,  $\text{dp}[\text{mask}]$  holds the sum over all subsets of  $\text{mask}$ .

### Example

Let  $n = 3$ , masks from 000 to 111.

If  $f[\text{mask}] = 1$  for all masks, then:

| mask | subsets           | g[mask] |
|------|-------------------|---------|
| 000  | {000}             | 1       |
| 001  | {000,001}         | 2       |
| 010  | {000,010}         | 2       |
| 011  | {000,001,010,011} | 4       |
| 100  | {000,100}         | 2       |
| 111  | all 8 subsets     | 8       |

The DP folds these results bit by bit.

### Tiny Code (Easy Versions)

C

```
#include <stdio.h>
#define MAXN (1<<20)

long long f[MAXN], dp[MAXN];

int main(void) {
    int n;
    scanf("%d", &n);
    int N = 1 << n;
    for (int mask = 0; mask < N; mask++) {
        scanf("%lld", &f[mask]);
        dp[mask] = f[mask];
    }

    for (int i = 0; i < n; i++) {
        for (int mask = 0; mask < N; mask++) {
            if (mask & (1 << i)) {
                dp[mask] += dp[mask ^ (1 << i)];
            }
        }
    }

    for (int mask = 0; mask < N; mask++)
        printf("%lld ", dp[mask]);
    printf("\n");
}
```

Python

```
n = int(input("n: "))
N = 1 << n
f = list(map(int, input().split()))
dp = f[:]

for i in range(n):
    for mask in range(N):
        if mask & (1 << i):
            dp[mask] += dp[mask ^ (1 << i)]

print(dp)
```

## Why It Matters

- Core trick in bitmask convolution, subset transforms, XOR convolution, and mobius inversion.
- Reduces  $O(3^n)$  subset loops to  $O(n2^n)$ .
- Common in problems over subset sums, probabilistic DP, counting states, and game theory.

## A Gentle Proof (Why It Works)

We can represent each mask as a binary vector of  $n$  bits. Each bit dimension  $i$  adds subsets with  $i$ th bit unset. Inductively, after processing bit  $i$ , each mask accumulates contributions from all subsets differing only in bits  $\leq i$ . By the end, every subset of `mask` has been included exactly once.

Formally, for each subset  $sub \subseteq mask$ , there exists a sequence of bit additions leading from  $sub$  to  $mask$ , ensuring its inclusion.

## Try It Yourself

1. Reverse the process to compute Sum Over Supersets instead.
2. Modify to compute product over subsets.
3. Apply to count subsets satisfying parity conditions.
4. Use SOS DP to precompute subset sums before a subset convolution.
5. Combine with inclusion-exclusion for faster combinatorial counting.

## Test Cases

| n | f (input) | Expected dp (output) |
|---|-----------|----------------------|
| 2 | 1 1 1 1   | 1 2 2 4              |
| 2 | 1 2 3 4   | 1 3 4 10             |
| 3 | all 1s    | 1 2 2 4 2 4 4 8      |

## Complexity

- Time:  $O(n2^n)$
- Space:  $O(2^n)$

SOS DP is a cornerstone of bitmask dynamic programming, revealing structure across subsets without enumerating them explicitly.

## 449 Bitmask Knapsack (State Compression)

The Bitmask Knapsack technique encodes subsets of items using bitmasks, allowing you to represent selections, transitions, and constraints compactly. It's a bridge between subset enumeration and dynamic programming, especially useful when the number of items is small (e.g.  $n \leq 20$ ) but the value/weight range is large.

### What Problem Are We Solving?

Given  $n$  items, each with weight  $w_i$  and value  $v_i$ , and capacity  $W$ , choose a subset with total weight  $\leq W$  to maximize total value.

Instead of indexing DP by capacity, we can enumerate subsets via bitmask:

$$best = \max_{\text{subset}} \sum_{i \in \text{subset}} v_i \quad \text{such that} \quad \sum_{i \in \text{subset}} w_i \leq W$$

Each subset corresponds to an integer mask, where bit  $i$  indicates inclusion of item  $i$ .

### How Does It Work (Plain Language)

A bitmask is a snapshot of which items are taken. You precompute total weight and total value for each subset. Then simply iterate all masks, filter by capacity, and keep the best.

It's exponential ( $2^n$ ) but works when  $n$  is small and weights are large, where classical DP by weight is infeasible.

## Transition Formula

For each mask:

- Compute

$$total_w = \sum_{i:mask_i=1} w_i$$

$$total_v = \sum_{i:mask_i=1} v_i$$

- If  $total_w \leq W$ , update answer:

$$best = \max(best, total_v)$$

Or incrementally:

$$dp[mask] = \sum_{i:mask_i=1} v_i$$

$$weight[mask] = \sum_{i:mask_i=1} w_i$$

## Tiny Code (Easy Versions)

C

```
#include <stdio.h>

#define MAXN 20
#define MAXMASK (1<<MAXN)

int main(void) {
    int n, W;
    scanf("%d %d", &n, &W);
    int w[n], v[n];
    for (int i = 0; i < n; i++) scanf("%d %d", &w[i], &v[i]);

    int N = 1 << n;
    int best = 0;

    for (int mask = 0; mask < N; mask++) {
        int total_w = 0, total_v = 0;
        for (int i = 0; i < n; i++) {
            if (mask & (1 << i)) {
```

```

        total_w += w[i];
        total_v += v[i];
    }
}
if (total_w <= W && total_v > best)
    best = total_v;
}

printf("%d\n", best);
}

```

Python

```

n, W = map(int, input().split())
w, v = [], []
for _ in range(n):
    wi, vi = map(int, input().split())
    w.append(wi)
    v.append(vi)

best = 0
for mask in range(1 << n):
    total_w = total_v = 0
    for i in range(n):
        if mask & (1 << i):
            total_w += w[i]
            total_v += v[i]
    if total_w <= W:
        best = max(best, total_v)

print(best)

```

## Why It Matters

- Works well when  $n$  is small (e.g.  $n \leq 20$ ) but weights/values are large
- Natural fit for meet-in-the-middle and subset enumeration
- Simplifies reasoning about combinations, constraints, and transitions
- Found in traveling salesman variants, set packing, and team selection problems

## Step-by-Step Example

Items:

| i | w | v |
|---|---|---|
| 0 | 3 | 4 |
| 1 | 4 | 5 |
| 2 | 2 | 3 |

Capacity  $W = 6$ .

Subsets:

| mask | items   | weight | value | feasible |
|------|---------|--------|-------|----------|
| 000  | {}      | 0      | 0     |          |
| 001  | {0}     | 3      | 4     |          |
| 010  | {1}     | 4      | 5     |          |
| 011  | {0,1}   | 7      | 9     |          |
| 100  | {2}     | 2      | 3     |          |
| 101  | {0,2}   | 5      | 7     |          |
| 110  | {1,2}   | 6      | 8     |          |
| 111  | {0,1,2} | 9      | 12    |          |

Best feasible = mask 110 → value 8.

## A Gentle Proof (Why It Works)

Each subset is a distinct combination of items. Enumerating all  $2^n$  subsets guarantees completeness. Feasibility is checked via total weight  $W$ , ensuring no invalid subset contributes. Maximization over all feasible subsets returns the global optimum.

No overlapping subproblems, so no need for memoization. The entire search space is finite and explored.

## Try It Yourself

1. Print all feasible subsets and their total values.
2. Combine with bitcount to restrict subset size.
3. Use meet-in-the-middle: split items into halves, enumerate each half, then merge results.
4. Extend to multi-dimensional capacity  $(W_1, W_2, \dots)$ .
5. Adapt to minimize weight for a target value instead.

## Test Cases

| Items             | W | Expected |
|-------------------|---|----------|
| (3,4),(4,5),(2,3) | 6 | 8        |
| (2,3),(3,4),(4,5) | 5 | 7        |
| (1,1),(2,2),(3,3) | 3 | 3        |

## Complexity

- Time:  $O(n2^n)$
- Space:  $O(1)$  (no DP table needed)

Bitmask Knapsack is a brute-force DP with compression, a go-to technique for small  $n$ , offering full flexibility when classical capacity-indexed DP would blow up.

## 450 Bitmask Independent Set (Graph Subset Optimization)

The Bitmask Independent Set DP enumerates all subsets of vertices in a graph to find the maximum-weight independent set, a set of vertices with no edges between any pair. It's a classic exponential DP, efficient for small graphs ( $n \leq 20$ ), using bit operations for adjacency and feasibility.

### What Problem Are We Solving?

Given an undirected graph  $G = (V, E)$  with vertex weights  $w_i$ , find:

$$\max \sum_{i \in S} w_i \quad \text{subject to } \forall (u, v) \in E, u, v \notin S$$

That is, choose a subset  $S$  of vertices with no adjacent pairs, maximizing total weight.

We represent each subset  $S$  by a bitmask, where bit  $i = 1$  means vertex  $i$  is included.

### How Does It Work (Plain Language)

We iterate over all  $2^n$  subsets. For each subset, we check whether it forms an independent set by ensuring no edge connects two chosen vertices. If it's valid, sum its vertex weights and update the best.

Precompute adjacency masks to quickly test validity.

## Transition / Check

For each mask:

### 1. Validity

A subset is independent if, for all vertices  $i$  included, it contains no neighbor:

$$(adj[i] \& mask) = 0$$

where  $adj[i]$  is the bitmask of neighbors of vertex  $i$ .

### 2. Value

$$value(mask) = \sum_{i: mask_i=1} w_i$$

### 3. Best

$$best = \max(best, value(mask))$$

## Tiny Code (Easy Versions)

C

```
#include <stdio.h>

#define MAXN 20

int main(void) {
    int n, m;
    scanf("%d %d", &n, &m);
    int w[n];
    for (int i = 0; i < n; i++) scanf("%d", &w[i]);

    int adj[n];
    for (int i = 0; i < n; i++) adj[i] = 0;
    for (int e = 0; e < m; e++) {
        int u, v;
        scanf("%d %d", &u, &v);
        adj[u] |= 1 << v;
        adj[v] |= 1 << u;
    }

    int N = 1 << n;
    int best = 0;
```

```

for (int mask = 0; mask < N; mask++) {
    int ok = 1, val = 0;
    for (int i = 0; i < n; i++) {
        if (mask & (1 << i)) {
            if (adj[i] & mask) { ok = 0; break; }
            val += w[i];
        }
    }
    if (ok && val > best) best = val;
}

printf("%d\n", best);
}

```

Python

```

n, m = map(int, input().split())
w = list(map(int, input().split()))

adj = [0]*n
for _ in range(m):
    u, v = map(int, input().split())
    adj[u] |= 1 << v
    adj[v] |= 1 << u

best = 0
for mask in range(1 << n):
    ok = True
    val = 0
    for i in range(n):
        if mask & (1 << i):
            if adj[i] & mask:
                ok = False
                break
            val += w[i]
    if ok:
        best = max(best, val)

print(best)

```

## Why It Matters

- Solves maximum independent set (MIS) for small graphs
- Useful for exact solutions in constraint problems, treewidth-based DP, or bitmask search
- A building block in graph coloring, maximum clique, dominating set, and subset optimization
- Adaptable for unweighted (count) or weighted (sum) versions

## Step-by-Step Example

Graph: 4 vertices, edges (0,1), (1,2), (2,3) Weights: [3, 2, 4, 1]

Independent sets:

| Mask | Set   | Valid | Value |
|------|-------|-------|-------|
| 0000 |       | 0     |       |
| 0001 | {0}   | 3     |       |
| 0010 | {1}   | 2     |       |
| 0100 | {2}   | 4     |       |
| 1000 | {3}   | 1     |       |
| 0101 | {0,2} | ,     |       |
| 1001 | {0,3} | 4     |       |
| 1100 | {2,3} | ,     |       |
| 1010 | {1,3} | 3     |       |

Best = 4 (either {2} or {0,3})

## A Gentle Proof (Why It Works)

Each subset represents a candidate solution. A subset is feasible iff it contains no adjacent pair, ensured by the adjacency mask check. Since every subset is tested, the algorithm finds the global optimum by enumeration.

Bitmask feasibility check ( $adj[i] \& mask == 0$ ) ensures constant-time validation per vertex, keeping complexity tight.

## Try It Yourself

1. Modify to count all independent sets.
2. Restrict to subsets of exact size  $k$ .
3. Add memoization to prune invalid masks early.
4. Combine with meet-in-the-middle for  $n = 40$ .
5. Flip edges to find maximum clique (complement graph).

## Test Cases

| Graph         | Weights     | Expected |
|---------------|-------------|----------|
| Chain 0–1–2–3 | [3,2,4,1]   | 4        |
| Star center 0 | [5,1,1,1,1] | 5        |
| Triangle      | [1,2,3]     | 3        |
| Empty graph   | [2,2,2]     | 6        |

## Complexity

- Time:  $O(n2^n)$
- Space:  $O(n)$

Bitmask Independent Set DP explores all subsets systematically, perfect when graphs are small but weights or constraints are complex.

# Section 46. Digit DP and SOS DP

## 451 Count Numbers with Property (Digit DP)

Digit DP is a method for counting numbers within a range that satisfy digit-level constraints, like no leading zeros, digit sum, specific digits, or forbidden patterns. This algorithm builds results by processing digits one by one while maintaining states for prefix constraints and current properties.

### What Problem Are We Solving?

Given an integer  $N$  and a property  $P$  (like “sum of digits is even”), count how many integers in  $[0, N]$  satisfy  $P$ .

Example: Count numbers  $\leq 327$  where the sum of digits is even.

We process each digit from most significant to least significant, maintaining:

- pos: current digit index
- sum: accumulated property (e.g. sum of digits mod 2)
- tight: whether we are bounded by  $N$ 's prefix (0 = free, 1 = still tight)
- leading: whether we've only seen leading zeros (optional)

State:

$$dp[pos][sum][tight]$$

Transition over next digit  $d$  (0..limit):

- Update `next_sum = (sum + d) % 2`
- If `tight=1`,  $limit = digit at pos in N$ , else 9
- Move to next position

Answer is sum over all valid end states satisfying the property.

## How Does It Work (Plain Language)

Digit DP works like counting with awareness: At each step, you choose the next digit, updating what you know (like current sum), while respecting the upper bound. By the time you process all digits, you've counted all valid numbers, no need to iterate up to  $N!$

## Transition Formula

Let  $S$  be the string of digits of  $N$ . For each state:

$$dp[pos][sum][tight] = \sum_{d=0}^{limit} dp[pos + 1][(sum + d) \bmod M][tight \wedge (d == limit)]$$

with base:

$$dp[len][sum][tight] = 1 \text{ if property holds, else } 0$$

Example: Property = sum of digits even  $\rightarrow sum$

## Tiny Code (Easy Versions)

Python (Count numbers  $N$  with even digit sum)

```
from functools import lru_cache

def count_even_sum(n):
    digits = list(map(int, str(n)))
    m = len(digits)

    @lru_cache(None)
    def dp(pos, sum_mod2, tight):
        if pos == m:
            return 1 if sum_mod2 == 0 else 0
        limit = digits[pos] if tight else 9
        total = 0
        for d in range(limit + 1):
            total += dp(pos + 1, (sum_mod2 + d) % 2, tight and d == limit)
    return total

return dp(0, 0, True)
```

```
N = int(input("Enter N: "))
print(count_even_sum(N))
```

C (Recursive Memoized DP)

```
#include <stdio.h>
#include <string.h>

int digits[20];
long long memo[20][2][2];
int len;

long long dp(int pos, int sum_mod2, int tight) {
    if (pos == len) return sum_mod2 == 0;
    if (memo[pos][sum_mod2][tight] != -1) return memo[pos][sum_mod2][tight];

    int limit = tight ? digits[pos] : 9;
    long long res = 0;
    for (int d = 0; d <= limit; d++) {
        res += dp(pos + 1, (sum_mod2 + d) % 2, tight && (d == limit));
    }
    return memo[pos][sum_mod2][tight] = res;
}

long long solve(long long n) {
    len = 0;
    while (n > 0) {
        digits[len++] = n % 10;
        n /= 10;
    }
    for (int i = 0; i < len / 2; i++) {
        int tmp = digits[i];
        digits[i] = digits[len - 1 - i];
        digits[len - 1 - i] = tmp;
    }
    memset(memo, -1, sizeof(memo));
    return dp(0, 0, 1);
}

int main(void) {
    long long n;
    scanf("%lld", &n);
```

```
    printf("%lld\n", solve(n));
}
```

## Why It Matters

- Fundamental to digit-based counting problems
- Efficiently handles constraints on digits, sum, mod, parity, forbidden patterns
- Avoids looping through large ranges (works in  $O(\text{len} \times \text{state})$ )
- Core idea behind counting numbers with:
  - Even digit sum
  - Specific digits (e.g. no 4)
  - Digits increasing/decreasing
  - Remainder mod M conditions

## Step-by-Step Example

Count numbers 327 with even digit sum.

We track sum mod 2:

- Start: pos=0, sum=0, tight=1
- First digit: choose 0..3
  - if 3 chosen  $\rightarrow$  sum=1, next tight=1
  - else  $\rightarrow$  free (tight=0)
- Continue until last digit
- At end, count where sum=0 (even)

Result: 164 numbers.

## A Gentle Proof (Why It Works)

At each position, `tight` ensures we never exceed N, and recursive branching over digits ensures coverage of all valid prefixes. By caching identical subproblems (same `pos`, `sum`, `tight`), we avoid recomputation. Thus, total states =  $O(\text{len} \times \text{property}_{\text{space}} \times 2)$ .

## Try It Yourself

1. Count numbers  $N$  with sum of digits divisible by 3.
2. Count numbers with no consecutive equal digits.
3. Count numbers with at most  $k$  nonzero digits.
4. Count numbers with digit product  $< M$ .
5. Adapt for range  $[L, R]$  via `solve(R) - solve(L-1)`.

## Test Cases

| N   | Property       | Expected |
|-----|----------------|----------|
| 9   | Even digit sum | 5        |
| 20  | Even digit sum | 10       |
| 327 | Even digit sum | 164      |

## Complexity

- Time:  $O(\text{len} \times M \times 2)$
- Space:  $O(\text{len} \times M \times 2)$

Digit DP transforms combinatorial counting into state-driven reasoning, digit by digit, a foundational trick for number-theoretic dynamic programming.

## 452 Count Without Adjacent Duplicates

This Digit DP problem counts numbers within a range that do not contain adjacent identical digits, a classic example where the state must remember the previous digit to enforce adjacency rules.

### What Problem Are We Solving?

Given an integer  $N$ , count how many integers in  $[0, N]$  have no two consecutive equal digits.

For example, up to  $N = 1234$ :

- Valid: 1203 (no repeats)
- Invalid: 1224 (two 2's together)

We'll use Digit DP to explore all digit sequences up to  $N$ , ensuring no adjacent duplicates.

## DP State

Let the string of digits of  $N$  be  $S$ , with length  $\text{len}$ .

State:

$$dp[pos][prev][tight][leading]$$

Where:

- `pos`: current index (0-based)
- `prev`: previous digit (0–9, or 10 if none yet)
- `tight`: whether prefix equals  $N$  so far (1 = bound, 0 = free)
- `leading`: whether we've seen only leading zeros (1 = true)

Each state counts valid completions from position `pos` onward.

## Transition

At position `pos`:

- Choose next digit `d` from 0 to `limit` (where `limit = S[pos]` if `tight = 1`, else 9)
- Skip if `d == prev` and not `leading` (no adjacent duplicates)
- Next state:
  - `next_prev = d` if not `leading` else 10
  - `next_tight = tight and (d == limit)`
  - `next_leading = leading and (d == 0)`

Sum all valid transitions.

Base:

$$dp[\text{len}][\text{prev}][\text{tight}][\text{leading}] = 1$$

when `pos == len` (end of number).

## How Does It Work (Plain Language)

We're building numbers digit by digit:

- `tight` keeps us within bounds.
- `prev` remembers the last chosen digit, to prevent repeating it.
- `leading` helps skip leading zeros (which don't count as duplicates).

By caching all combinations, we count every valid number exactly once.

## Tiny Code (Easy Versions)

Python

```
from functools import lru_cache

def count_no_adjacent(N):
    digits = list(map(int, str(N)))
    m = len(digits)

    @lru_cache(None)
    def dp(pos, prev, tight, leading):
        if pos == m:
            return 1 # valid number
        limit = digits[pos] if tight else 9
        total = 0
        for d in range(limit + 1):
            if not leading and d == prev:
                continue # no adjacent duplicates
            total += dp(
                pos + 1,
                d if not leading else 10,
                tight and d == limit,
                leading and d == 0
            )
        return total

    return dp(0, 10, True, True)

N = int(input("Enter N: "))
print(count_no_adjacent(N))
```

C

```
#include <stdio.h>
#include <string.h>

int digits[20];
long long memo[20][11][2][2];
int len;

long long dp(int pos, int prev, int tight, int leading) {
```

```

if (pos == len) return 1;
if (memo[pos][prev][tight][leading] != -1) return memo[pos][prev][tight][leading];

int limit = tight ? digits[pos] : 9;
long long res = 0;
for (int d = 0; d <= limit; d++) {
    if (!leading && d == prev) continue;
    int next_prev = leading && d == 0 ? 10 : d;
    int next_tight = tight && (d == limit);
    int next_leading = leading && (d == 0);
    res += dp(pos + 1, next_prev, next_tight, next_leading);
}
return memo[pos][prev][tight][leading] = res;
}

long long solve(long long n) {
len = 0;
while (n > 0) {
    digits[len++] = n % 10;
    n /= 10;
}
for (int i = 0; i < len / 2; i++) {
    int tmp = digits[i];
    digits[i] = digits[len - 1 - i];
    digits[len - 1 - i] = tmp;
}
memset(memo, -1, sizeof(memo));
return dp(0, 10, 1, 1);
}

int main(void) {
long long n;
scanf("%lld", &n);
printf("%lld\n", solve(n));
}

```

## Why It Matters

- Classic Digit DP with “previous digit” state
- Enables constraints like:

- No adjacent repeats
- No increasing/decreasing sequences
- No forbidden pairs
- Useful in pattern counting, PIN code generation, license plate validation

### **Step-by-Step Example**

Count numbers  $N$  with no adjacent duplicates:

- Leading zeros allowed at first
- E.g. 101, 110
- DP checks each digit:
  - 1?0 (pos=0, prev=10)
  - For next digit: skip equal to prev
- Total valid count = 91

### **A Gentle Proof (Why It Works)**

For each position, transitions ensure:

- Only digits bound if `tight=1`
- No consecutive duplicates via `d != prev`
- Leading zeros treated specially (ignored in duplicate check)

By iterating through all valid digits and memoizing results, each subproblem (prefix constraint + previous digit) is solved once, ensuring completeness and correctness.

### **Try It Yourself**

1. Count numbers  $N$  with no equal adjacent digits and sum of digits even.
2. Count numbers  $N$  with strictly increasing digits.
3. Modify to disallow adjacent 0's only.
4. Combine with mod constraints (digit sum mod  $M$ ).
5. Extend to handle exactly  $k$  equal pairs.

### **Test Cases**

| N    | Expected | Notes           |
|------|----------|-----------------|
| 9    | 10       | 0–9 all valid   |
| 11   | 10       | 10 invalid      |
| 100  | 91       | Only 9 invalids |
| 1234 | 820      | Approx result   |

## Complexity

- Time:  $O(\text{len} \times 11 \times 2 \times 2 \times 10)$
- Space:  $O(\text{len} \times 11 \times 2 \times 2)$

Digit DP elegantly enforces local digit constraints (like adjacency) through memory of the previous digit, enabling fast counting across massive ranges.

## 453 Sum of Digits in Range

This Digit DP problem computes the sum of digits of all numbers in a given range  $[0, N]$ . Instead of enumerating numbers, we accumulate digit contributions position by position, respecting upper bounds.

### What Problem Are We Solving?

Given a number  $N$ , compute:

$$S(N) = \sum_{x=0}^N \text{sum\_of\_digits}(x)$$

For example:

- $S(13) = 1 + 0 + 1 + 1 + 1 + 2 + 1 + 3 = 55$

The goal is to compute  $S(N)$  efficiently in  $O(\text{len} \times 2 \times M)$ , not  $O(N)$ .

You can also handle ranges:

$$S(L, R) = S(R) - S(L - 1)$$

## DP State

Let  $S$  = list of digits of  $N$ .

We define a recursive function:

$$dp[pos][tight][sum]$$

But instead of counting numbers, we also accumulate the total digit sum contribution.

So the function returns (count, total\_sum), a pair:

- **count**: number of valid numbers
- **sum**: sum of digits over all valid numbers

State:

- **pos**: current position (0..len-1)
- **tight**: whether we are still bounded by prefix
- **leading**: whether only leading zeros so far

## Transition

At each position, choose digit  $d$  in  $[0, \text{limit}]$  ( $\text{limit} = \text{digit at pos if tight} = 1, \text{else } 9$ )

Let  $(\text{cnt\_next}, \text{sum\_next})$  = result from next position.

We add current digit's contribution:

$$\text{total}_s um+ = \text{sum}_n ext + d \times \text{cnt}_n ext$$

If **leading** is true and  $d=0$ , then we don't count that as a "real" leading digit.

Base case:

$$dp[len][tight][leading] = (1, 0)$$

(one valid number, sum = 0)

## How Does It Work (Plain Language)

Each recursive call counts how many numbers are possible from this prefix, and how much total digit sum they produce.

When you pick a digit d:

- d contributes  $d * \text{cnt\_next}$  to all numbers in this branch
- Plus whatever the rest of the digits contribute recursively

By caching (`count`, `sum`) per state, we reuse computations for repeated prefixes.

## Tiny Code (Easy Version)

Python

```
from functools import lru_cache

def sum_of_digits_up_to(N):
    digits = list(map(int, str(N)))
    m = len(digits)

    @lru_cache(None)
    def dp(pos, tight, leading):
        if pos == m:
            return (1, 0) # (count, sum)

        limit = digits[pos] if tight else 9
        total_count, total_sum = 0, 0

        for d in range(limit + 1):
            cnt_next, sum_next = dp(
                pos + 1,
                tight and (d == limit),
                leading and d == 0
            )
            total_count += cnt_next
            total_sum += sum_next + (0 if leading and d == 0 else d * cnt_next)

        return (total_count, total_sum)

    return dp(0, True, True)[1]
```

```

def sum_of_digits_range(L, R):
    return sum_of_digits_upto(R) - sum_of_digits_upto(L - 1)

# Example
L, R = map(int, input("Enter L R: ").split())
print(sum_of_digits_range(L, R))

```

C (Recursive Pair Return via struct)

```

#include <stdio.h>
#include <string.h>

typedef struct { long long count, sum; } Pair;

int digits[20];
Pair memo[20][2][2];
int vis[20][2][2];
int len;

Pair dp(int pos, int tight, int leading) {
    if (pos == len) return (Pair){1, 0};
    if (vis[pos][tight][leading]) return memo[pos][tight][leading];
    vis[pos][tight][leading] = 1;

    int limit = tight ? digits[pos] : 9;
    long long total_count = 0, total_sum = 0;

    for (int d = 0; d <= limit; d++) {
        Pair next = dp(pos + 1, tight && (d == limit), leading && (d == 0));
        total_count += next.count;
        total_sum += next.sum + (leading && d == 0 ? 0 : (long long)d * next.count);
    }

    return memo[pos][tight][leading] = (Pair){total_count, total_sum};
}

long long solve(long long n) {
    if (n < 0) return 0;
    len = 0;
    while (n > 0) {
        digits[len++] = n % 10;
        n /= 10;
    }
}

```

```

    }
    for (int i = 0; i < len / 2; i++) {
        int tmp = digits[i];
        digits[i] = digits[len - 1 - i];
        digits[len - 1 - i] = tmp;
    }
    memset(vis, 0, sizeof(vis));
    return dp(0, 1, 1).sum;
}

int main(void) {
    long long L, R;
    scanf("%lld %lld", &L, &R);
    printf("%lld\n", solve(R) - solve(L - 1));
}

```

## Why It Matters

- Computes digit sums over huge ranges in logarithmic time
- Basis for many digit aggregation problems (count of 1's, digit sum mod M, etc.)
- Extensible to:
  - Counting even/odd digits
  - Weighted digit sums (like  $d \times 10^{pos}$ )
  - Property-based aggregation (e.g., sum of squares)

## Step-by-Step Example

Compute sum of digits for all numbers 13:

| Number | Sum |
|--------|-----|
| 0      | 0   |
| 1      | 1   |
| 2      | 2   |
| 3      | 3   |
| 4      | 4   |
| 5      | 5   |
| 6      | 6   |
| 7      | 7   |

| Number | Sum |
|--------|-----|
| 8      | 8   |
| 9      | 9   |
| 10     | 1   |
| 11     | 2   |
| 12     | 3   |
| 13     | 4   |

Total = 55

DP builds this by digit:

- Tens digit → repeats 10 times
- Ones digit → contributes 0–9 repeatedly

### A Gentle Proof (Why It Works)

Each position contributes its digit value multiplied by the number of combinations of remaining positions. Digit DP captures this recursively: If a digit  $d$  is fixed at position  $pos$ , every completion of later positions includes that digit once, hence  $d * \text{count\_of\_suffix}$ . Summing over all digits at all positions gives the total sum.

### Try It Yourself

1. Count sum of even digits only.
2. Compute sum of digits mod M.
3. Compute sum of squared digits.
4. Count total number of digits written in range.
5. Compute weighted sum (like  $d * 10^{pos}$  contributions).

### Test Cases

| Range  | Expected |
|--------|----------|
| 0–9    | 45       |
| 0–13   | 55       |
| 10–99  | 855      |
| 1–1000 | 13501    |

## Complexity

- Time:  $O(\text{len} \times 2 \times 2 \times 10)$
- Space:  $O(\text{len} \times 2 \times 2)$

Digit DP can aggregate digit-level properties over massive intervals, this sum version is its canonical “count + accumulate” template.

## 454 Count with Mod Condition (Digit Sum mod M)

Count numbers in a range whose digit sum satisfies a modular condition. The standard pattern tracks the digit-sum modulo  $M$  while respecting the upper bound.

### What Problem Are We Solving?

Given integers  $N, M, K$ , count how many  $x \in [0, N]$  satisfy:

$$(\text{sum\_digits}(x)) \bmod M = K$$

For a general range  $[L, R]$ , use  $f(R) - f(L - 1)$ .

### DP State

Let the decimal string of  $N$  be  $S$ , length  $m$ .

State:

$$dp[pos][mod][tight][leading]$$

- $pos$ : index in  $S$  (0-based, left to right)
- $mod$ : current value of digit-sum modulo  $M$
- $tight \in 0, 1$ : still equal to  $S$  prefix or already below
- $leading \in 0, 1$ : still placing only leading zeros

Goal: count completions with final  $mod = K$ .

Base:

$$dp[m][mod][tight][leading] = \begin{cases} 1, & mod = K, \\ 0, & \text{otherwise.} \end{cases}$$

Transition (choose next digit  $d$ ):

- $limit = S[pos]$  if  $tight = 1$ , else 9
- Next states:

- $next\_tight = tight \wedge (d = limit)$
- $next\_leading = leading \wedge (d = 0)$

$$next\_mod = \begin{cases} mod, & \text{if } next\_leading = 1, \\ (mod + d) \bmod M, & \text{otherwise.} \end{cases}$$

Then

$$dp[pos][mod][tight][leading] = \sum_{d=0}^{limit} dp[pos+1][next\_mod][next\_tight][next\_leading].$$

### Tiny Code (Easy Versions)

Python

```
from functools import lru_cache

def count_mod_sum_upto(N, M, K):
    S = list(map(int, str(N)))
    m = len(S)

    @lru_cache(None)
    def dp(pos, mod, tight, leading):
        if pos == m:
            return 1 if mod == K else 0
        limit = S[pos] if tight else 9
        total = 0
        for d in range(limit + 1):
            ntight = tight and (d == limit)
            nleading = leading and (d == 0)
            nmod = mod if nleading else (mod + d) % M
            total += dp(pos + 1, nmod, ntight, nleading)
        return total

    return dp(0, 0, True, True)

def count_mod_sum_range(L, R, M, K):
    if L <= 0:
        return count_mod_sum_upto(R, M, K)
    return count_mod_sum_upto(R, M, K) - count_mod_sum_upto(L - 1, M, K)
```

```
# Example:
# print(count_mod_sum_range(0, 327, 7, 3))
```

C

```
#include <stdio.h>
#include <string.h>

long long memo[20][200][2][2];
char vis[20][200][2][2];
int digits[20], mlen, M, K;

long long solve_dp(int pos, int mod, int tight, int leading) {
    if (pos == mlen) return mod == K;
    if (vis[pos][mod][tight][leading]) return memo[pos][mod][tight][leading];
    vis[pos][mod][tight][leading] = 1;

    int limit = tight ? digits[pos] : 9;
    long long total = 0;
    for (int d = 0; d <= limit; d++) {
        int ntight = tight && (d == limit);
        int nleading = leading && (d == 0);
        int nmod = nleading ? mod : (mod + d) % M;
        total += solve_dp(pos + 1, nmod, ntight, nleading);
    }
    return memo[pos][mod][tight][leading] = total;
}

long long count_upto(long long N, int m, int k) {
    if (N < 0) return 0;
    M = m; K = k;
    int tmp[20], len = 0;
    while (N > 0) { tmp[len++] = (int)(N % 10); N /= 10; }
    if (len == 0) tmp[len++] = 0;
    for (int i = 0; i < len; i++) digits[i] = tmp[len - 1 - i];
    mlen = len;
    memset(vis, 0, sizeof(vis));
    return solve_dp(0, 0, 1, 1);
}

long long count_range(long long L, long long R, int m, int k) {
```

```

    return count_up_to(R, m, k) - count_up_to(L - 1, m, k);
}

// Example usage in main:
// int main(){ long long L=0,R=327; int M=7,K=3; printf("%lld\n", count_range(L,R,M,K)); }

```

Notes:

- The array bound 200 for `mod` assumes  $M \leq 200$ . Increase if needed.

## Why It Matters

- Core Digit DP for number-theory style constraints
- Handles many variants by changing the carried statistic to modulo form
- Building block for problems like:
  - Count with digit sum equal to  $S$  (set  $M$  large enough and target  $K = S$  with careful state)
  - Count with digit sum in a set (sum over multiple  $K$ )
  - Multi-condition states (e.g., digit sum mod  $M$  and parity)

## Step-by-Step Example

Count  $x \in [0, 99]$  with digit sum mod 3 equal to 0.

- The DP carries  $mod \in 0, 1, 2$ .
- At each position, branch over  $d \in [0..9]$  with tight until you pass the bound 99.
- The answer is 34.

(This small case can also be verified by combinatorics: roughly one third of two-digit-with-leading-zero numbers.)

## A Gentle Proof (Why It Works)

Every number corresponds to one path of digit choices. The state  $(pos, mod, tight, leading)$  uniquely captures all information that influences future feasibility and the final condition  $mod = K$ . Since the transition only depends on the current state and chosen digit, memoizing these states yields a complete and non-overlapping partition of the search space.

## Try It Yourself

1. Count numbers with digit sum mod  $M$  in a set  $S$  by summing answers over  $K \in S$ .
2. Count numbers with digit sum exactly  $S$  by replacing  $mod$  with a bounded sum state and capping at  $S$ .
3. Combine with no adjacent duplicates by adding a `prev` digit to the state.
4. Compute numbers with sum of squares of digits mod  $M$  equal to  $K$ .
5. Extend to base- $B$  by changing the limit from 9 to  $B - 1$ .

## Test Cases

| $N$ | $M$ | $K$ | Expected idea       |
|-----|-----|-----|---------------------|
| 9   | 3   | 0   | 4 numbers (0,3,6,9) |
| 20  | 2   | 0   | about half of 0..20 |
| 99  | 3   | 0   | 34                  |
| 327 | 7   | 3   | computed via code   |

## Complexity

- Time:  $O(\text{len} \cdot M \cdot 2 \cdot 2 \cdot 10)$
- Space:  $O(\text{len} \cdot M \cdot 2 \cdot 2)$

This is the standard mod-carry Digit DP: thread the property through the digits modulo  $M$ , respect the bound with `tight`, and account for leading zeros cleanly.

## 455 Count of Increasing Digits

We want to count all integers in a range that have strictly increasing digits, every digit is larger than the one before it. For example, 123, 149, and 7 qualify, but 133, 321, or 224 do not.

### What Problem Are We Solving?

Given an upper bound  $N$ , count integers  $x$  in  $[0, N]$  such that:

$$\text{digits}(x) = [d_0, d_1, \dots, d_k] \implies d_0 < d_1 < \dots < d_k$$

Example: For  $N = 500$ , valid numbers include 1, 2, ..., 9, 12, 13, ..., 49, 123, 134, ..., 489, etc.

We can model this with Digit DP tracking the previous digit to ensure the increasing condition.

## DP State

Let  $S$  = list of digits of  $N$ .

State:

$$dp[pos][prev][tight][leading]$$

Where:

- $\text{pos}$ : index of current digit (0-based)
- $\text{prev}$ : last chosen digit (0–9, or 10 for “none yet”)
- $\text{tight}$ : whether we’re still prefix-equal to  $N$
- $\text{leading}$ : whether we’ve only placed leading zeros (so far no real digits)

## Transition

At each position:

- Determine  $\text{limit} = S[\text{pos}]$  if  $\text{tight}=1$ , else 9.
- Loop  $d$  from 0 to  $\text{limit}$ .
- Skip  $d \leq \text{prev}$  if not  $\text{leading}$  (must strictly increase).
- Update:
  - $\text{next_tight} = \text{tight}$  and ( $d == \text{limit}$ )
  - $\text{next_leading} = \text{leading}$  and ( $d == 0$ )
  - $\text{next_prev} = \text{prev}$  if  $\text{next_leading}$  else  $d$

Sum results of recursive calls.

Base:

$$dp[len][prev][tight][leading] = 1$$

since one valid number is formed.

## How Does It Work (Plain Language)

We build the number one digit at a time:

- If we’ve started (not leading), each new digit must be greater than the previous one.
- If we’re still leading, any zero is fine.
- $\text{tight}$  ensures we never exceed  $N$ ’s prefix. By exploring all possible digits under these rules, we count every strictly increasing number  $N$ .

## Tiny Code (Easy Versions)

Python

```
from functools import lru_cache

def count_increasing_digits(N):
    S = list(map(int, str(N)))
    m = len(S)

    @lru_cache(None)
    def dp(pos, prev, tight, leading):
        if pos == m:
            return 1 # reached end, valid number
        limit = S[pos] if tight else 9
        total = 0
        for d in range(limit + 1):
            if not leading and d <= prev:
                continue # must strictly increase
            ntight = tight and (d == limit)
            nleading = leading and (d == 0)
            nprev = prev if nleading else d
            total += dp(pos + 1, nprev, ntight, nleading)
        return total

    return dp(0, 10, True, True)

# Example
N = int(input("Enter N: "))
print(count_increasing_digits(N))
```

C

```
#include <stdio.h>
#include <string.h>

long long memo[20][11][2][2];
char vis[20][11][2][2];
int digits[20], len;

long long dp(int pos, int prev, int tight, int leading) {
    if (pos == len) return 1;
```

```

if (vis[pos][prev][tight][leading]) return memo[pos][prev][tight][leading];
vis[pos][prev][tight][leading] = 1;

int limit = tight ? digits[pos] : 9;
long long res = 0;
for (int d = 0; d <= limit; d++) {
    if (!leading && d <= prev) continue;
    int ntight = tight && (d == limit);
    int nleading = leading && (d == 0);
    int nprev = nleading ? prev : d;
    res += dp(pos + 1, nprev, ntight, nleading);
}

return memo[pos][prev][tight][leading] = res;
}

long long solve(long long n) {
    if (n < 0) return 0;
    len = 0;
    while (n > 0) {
        digits[len++] = n % 10;
        n /= 10;
    }
    if (len == 0) digits[len++] = 0;
    for (int i = 0; i < len / 2; i++) {
        int t = digits[i];
        digits[i] = digits[len - 1 - i];
        digits[len - 1 - i] = t;
    }
    memset(vis, 0, sizeof(vis));
    return dp(0, 10, 1, 1);
}

int main(void) {
    long long N;
    scanf("%lld", &N);
    printf("%lld\n", solve(N));
}

```

## Why It Matters

- Builds understanding of monotonic digit constraints
- Template for counting:
  - Strictly increasing digits
  - Strictly decreasing digits
  - Non-decreasing digits (just change condition to  $d < \text{prev}$ )
- Appears in combinatorial enumeration and digit ordering problems

## Step-by-Step Example

For  $N = 130$ :

Valid numbers include:

0-9  
12, 13  
23  
...  
123

Invalid examples:

- 11 (equal digits)
- 21 (decreasing)

DP automatically filters these based on the  $d > \text{prev}$  rule.

## A Gentle Proof (Why It Works)

Each path corresponds to a unique number  $N$ . The rule  $d > \text{prev}$  enforces strictly increasing order. The DP ensures no overcounting because each prefix (`pos`, `prev`, `tight`, `leading`) fully determines future choices.

## Try It Yourself

1. Modify to count strictly decreasing numbers ( $d < \text{prev}$ ).
2. Count non-decreasing numbers ( $d \geq \text{prev}$ ).
3. Enforce exact length  $k$  via a `len_used` parameter.
4. Add mod conditions (sum mod M).
5. Compute sum of all increasing numbers instead of count.

## Test Cases

| N   | Expected | Notes                            |
|-----|----------|----------------------------------|
| 9   | 10       | 0–9 valid                        |
| 12  | 12       | 10 numbers 0–9, plus 12 and 13   |
| 99  | 45       | All 1–2-digit increasing numbers |
| 321 | 84       | Derived by DP                    |

## Complexity

- Time:  $O(\text{len} \times 11 \times 2 \times 2 \times 10)$
- Space:  $O(\text{len} \times 11 \times 2 \times 2)$

Digit DP with monotonic digit constraints transforms ordering problems into state-space counting, a fundamental technique for combinatorial digit analysis.

## 456 Count with Forbidden Digits

Count how many integers in a range avoid a given set of forbidden digits. This is a basic Digit DP where the state remembers whether we are still tight to the upper bound and whether we have only placed leading zeros.

### What Problem Are We Solving?

Given  $N$  and a set  $F \subseteq 0, 1, \dots, 9$  of forbidden digits, count integers  $x \in [0, N]$  whose standard decimal representation contains no digit from  $F$ .

Convention: leading zeros are allowed during the DP but do not count as real digits, so a leading zero is always permitted even if  $0 \in F$ .

For a general range  $[L, R]$  return  $f(R) - f(L - 1)$ .

## DP State

Let  $S$  be the digit list of  $N$ , length  $m$ .

State:

$$dp[pos][tight][leading]$$

- $pos$ : index in  $[0, m)$
- $tight \in 0, 1$ : 1 if the prefix equals  $N$  so far
- $leading \in 0, 1$ : 1 if all chosen digits are leading zeros

## Transition

At position  $pos$  choose digit  $d \in [0, \text{limit}]$ , where  $\text{limit} = S[pos]$  if  $tight = 1$  else 9.

Reject  $d$  if it is a real digit that is forbidden:

- If  $leading = 1$  and  $d = 0$ , accept regardless of  $F$ .
- Otherwise require  $d \notin F$ .

Next state:

- $next_{tight} = tight \wedge (d = \text{limit})$
- $next_{leading} = leading \wedge (d = 0)$

Recurrence:

$$dp[pos][tight][leading] = \sum_{d=0}^{\text{limit}} \mathbf{1}(\text{allowed}(d, leading)) \cdot dp[pos+1][next_{tight}][next_{leading}]$$

Base:

$$dp[m][tight][leading] = 1$$

Answer is  $dp[0][1][1]$ .

## How Does It Work (Plain Language)

We build the number left to right. If we have not surpassed  $N$  yet, the next digit is restricted by  $N$  at that position. Leading zeros are virtual padding and do not trigger the forbidden check. The DP counts all valid completions from each prefix.

## Tiny Code (Easy Versions)

Python

```
from functools import lru_cache

def count_without_forbidden(N, forbidden):
    S = list(map(int, str(N)))
    m = len(S)
    F = set(forbidden)

    @lru_cache(None)
    def dp(pos, tight, leading):
        if pos == m:
            return 1
        limit = S[pos] if tight else 9
        total = 0
        for d in range(limit + 1):
            ntight = tight and (d == limit)
            nleading = leading and (d == 0)
            # allow leading zero regardless of F
            if not nleading and d in F:
                continue
            total += dp(pos + 1, ntight, nleading)
        return total

    return dp(0, True, True)

def count_range_without_forbidden(L, R, forbidden):
    if L <= 0:
        return count_without_forbidden(R, forbidden)
    return count_without_forbidden(R, forbidden) - count_without_forbidden(L - 1, forbidden)

# Example:
# print(count_range_without_forbidden(0, 327, {3,4}))
```

C

```
#include <stdio.h>
#include <string.h>

long long memo[20][2][2];
```

```

char vis[20][2][2];
int digits[20], len;
int forbid[10];

long long dp(int pos, int tight, int leading) {
    if (pos == len) return 1;
    if (vis[pos][tight][leading]) return memo[pos][tight][leading];
    vis[pos][tight][leading] = 1;

    int limit = tight ? digits[pos] : 9;
    long long total = 0;

    for (int d = 0; d <= limit; d++) {
        int ntight = tight && (d == limit);
        int nleading = leading && (d == 0);
        if (!nleading && forbid[d]) continue; // real digit must not be forbidden
        total += dp(pos + 1, ntight, nleading);
    }
    return memo[pos][tight][leading] = total;
}

long long solve_upto(long long N) {
    if (N < 0) return 0;
    len = 0;
    if (N == 0) digits[len++] = 0;
    while (N > 0) { digits[len++] = (int)(N % 10); N /= 10; }
    for (int i = 0; i < len/2; i++) {
        int t = digits[i]; digits[i] = digits[len-1-i]; digits[len-1-i] = t;
    }
    memset(vis, 0, sizeof(vis));
    return dp(0, 1, 1);
}

// Example main
// int main(void){
//     long long L,R; int k,x;
//     scanf("%lld %lld %d",&L,&R,&k);
//     memset(forbid,0,sizeof(forbid));
//     for(int i=0;i<k;i++){ scanf("%d",&x); forbid[x]=1; }
//     long long ans = solve_upto(R) - solve_upto(L-1);
//     printf("%lld\n", ans);
// }

```

## Why It Matters

- Canonical Digit DP that filters digits by a local constraint
- Models problems with digit blacklists, keypad rules, license formats, or numeral-system restrictions
- Serves as a base to combine with additional states like digit sum or adjacency constraints

## Step-by-Step Example

Let  $F = 3, 4$  and  $N = 327$ .

- At each position, digits 3, 4 are disallowed unless we are still in leading zeros.
- The DP explores all prefixes bounded by 327 and sums valid completions.
- Use the Python snippet to compute the exact count.

## A Gentle Proof (Why It Works)

Every number in  $[0, N]$  corresponds to a unique path of digit choices. The predicate  $\text{allowed}(d, \text{leading})$  ensures that once a real digit is placed, it is not forbidden. The pair  $(pos, tight)$  ensures we do not exceed  $N$ . Since subproblems depend only on these three parameters, memoization counts each equivalence class of prefixes exactly once.

## Try It Yourself

1. Forbid multiple digits, e.g.  $F = 1, 3, 7$ .
2. Forbid a set that includes 0 and verify that leading zeros still pass.
3. Combine with a sum modulo condition by adding a  $mod$  state.
4. Combine with no adjacent duplicates by adding a  $prev$  state.
5. Switch to base  $B$  by changing the limit from 9 to  $B - 1$ .

## Test Cases

| $N$  | $F$     | Expected idea                                                  |
|------|---------|----------------------------------------------------------------|
| 99   | 9       | Count of two-digit-with-leading-zero numbers using digits 0..8 |
| 327  | 3, 4    | Computed by code                                               |
| 1000 | 1       | All numbers without digit 1                                    |
| 0    | any $F$ | 1 (the number 0)                                               |

## Complexity

- Time:  $O(len \cdot 2 \cdot 2 \cdot 10)$
- Space:  $O(len \cdot 2 \cdot 2)$

This pattern is the simplest Digit DP guard: screen digits against a blacklist while handling bounds and leading zeros correctly.

## 457 SOS DP Subset Sum

Sum Over Subsets (SOS) DP is a powerful bitmask technique used to precompute values over all subsets of a given mask. One of its core applications is the Subset Sum over bitmasks, efficiently computing  $f(S) = \sum_{T \subseteq S} g(T)$  for all  $S$ .

### What Problem Are We Solving?

Given an array  $\mathbf{g}$  of size  $2^n$  indexed by bitmasks, compute a new array  $\mathbf{f}$  such that:

$$f[S] = \sum_{T \subseteq S} g[T]$$

A naive approach iterates through all subsets for each  $S$ , which takes  $O(3^n)$ . SOS DP reduces this to  $O(n \cdot 2^n)$ , making it feasible for  $n \leq 20$ .

### How Does It Work (Plain Language)

We treat each bit as a dimension. For each bit position  $i$  (from 0 to  $n - 1$ ):

- For each mask  $S$ :
  - If bit  $i$  is set in  $S$ , add contribution from  $S$  with bit  $i$  cleared.

This accumulates sums over all subsets, one bit at a time.

## Recurrence

Let  $f$  initially equal  $g$ . Then:

```
for i = 0, ..., n - 1 :  
    for S = 0, ..., (1<<n) - 1 :  
        if (S & (1<<i)) ≠ 0 : f[S] += f[S without i]
```

where  $S^{\text{without } i} = S \oplus (1<<i)$  removes bit  $i$ .

After processing all bits,  $f[S]$  holds the sum over all subsets of  $S$ .

## Tiny Code (Easy Versions)

Python

```
def sos_subset_sum(g, n):  
    f = g[:] # copy  
    for i in range(n):  
        for S in range(1 << n):  
            if S & (1 << i):  
                f[S] += f[S ^ (1 << i)]  
    return f  
  
# Example  
n = 3  
g = [1, 2, 3, 4, 5, 6, 7, 8] # g[mask]  
f = sos_subset_sum(g, n)  
print(f)
```

C

```
#include <stdio.h>  
  
void sos_subset_sum(long long f[], int n) {  
    for (int i = 0; i < n; i++) {  
        for (int S = 0; S < (1 << n); S++) {  
            if (S & (1 << i)) {  
                f[S] += f[S ^ (1 << i)];  
            }  
        }  
    }  
}
```

```

    }
}

int main() {
    int n = 3;
    long long f[1 << 3] = {1,2,3,4,5,6,7,8};
    sos_subset_sum(f, n);
    for (int i = 0; i < (1 << n); i++) printf("%lld ", f[i]);
    return 0;
}

```

## Why It Matters

- Foundation for bitmask DP transforms (e.g. subset convolution, inclusion-exclusion).
- Enables fast enumeration of subset properties (sums, counts, etc.).
- Reusable building block in probabilistic DP, polynomial transforms, and game DP.

## Step-by-Step Example

Let  $n = 2$ , masks 00, 01, 10, 11 and  $g = [1, 2, 3, 4]$ :

Initialize  $f = g$ .

1. Bit  $i = 0$ :

- $S = 01$ :  $f[01]+ = f[00] \implies 2 + 1 = 3$
- $S = 11$ :  $f[11]+ = f[10] \implies 4 + 3 = 7$

$$f = [1, 3, 3, 7]$$

2. Bit  $i = 1$ :

- $S = 10$ :  $f[10]+ = f[00] \implies 3 + 1 = 4$
- $S = 11$ :  $f[11]+ = f[01] \implies 7 + 3 = 10$

$$\text{Final } f = [1, 3, 4, 10]$$

Check:

- $f[11] = g[00] + g[01] + g[10] + g[11] = 1 + 2 + 3 + 4 = 10$

## A Gentle Proof (Why It Works)

Each bit is processed independently. At iteration  $i$ , each mask  $S$  accumulates contributions from all subsets differing only at bit  $i$ . After processing all bits, every subset  $T \subseteq S$  is visited exactly once.

By induction:

- Base:  $i = 0$ ,  $f[S]$  contains  $g[S]$ .
- Step: adding  $f[S^{\text{without } i}]$  ensures inclusion of subsets missing bit  $i$ .

Thus, after  $n$  passes,  $f[S]$  sums over all subsets.

## Try It Yourself

1. Change sum to product (if  $f[S]* = f[S^{\text{without } i}]$ ).
2. Compute  $f[S] = \sum_{T \supseteq S} g[T]$  (see Superset DP).
3. Combine SOS DP with inclusion-exclusion to count valid subsets.
4. Apply to subset convolution problems.
5. Use modulo arithmetic for large values.

## Test Cases

| n | g (input)         | f (output)        |
|---|-------------------|-------------------|
| 2 | [1,2,3,4]         | [1,3,4,10]        |
| 3 | [1,1,1,1,1,1,1,1] | [1,2,2,4,2,4,4,8] |
| 3 | [0,1,2,3,4,5,6,7] | computed by code  |

## Complexity

- Time:  $O(n \cdot 2^n)$
- Space:  $O(2^n)$

SOS DP transforms the exponential subset-sum enumeration into a structured linear pass across dimensions, making subset-based computation tractable for small  $n$ .

## 458 SOS DP Superset Sum

Sum Over Supersets computes for every mask the aggregate over all of its supersets. It complements the usual SOS DP over subsets.

## What Problem Are We Solving?

Given an array  $g$  of size  $2^n$  indexed by bitmasks, compute  $h$  such that

$$h[S] = \sum_{T \supseteq S} g[T].$$

Naively this is  $O(3^n)$ . With SOS Superset DP it is  $O(n \cdot 2^n)$ .

## How Does It Work (Plain Language)

Process bits one by one. For each bit  $i$ , if a mask  $S$  has bit  $i$  unset, then every superset that turns this bit on is of the form  $S \cup i = S \oplus (1 << i)$ . So we can accumulate from that neighbor upward.

## Recurrence

Initialize  $h = g$ . For each bit  $i = 0..n - 1$ :

For every mask  $S$ :

If  $(S \& (1 << i)) == 0$ , then

$$h[S] += h[S \oplus (1 << i)].$$

After all bits,  $h[S]$  equals the sum over all supersets of  $S$ .

## Tiny Code

Python

```
def sos_superset_sum(g, n):
    h = g[:] # copy
    for i in range(n):
        for S in range(1 << n):
            if (S & (1 << i)) == 0:
                h[S] += h[S | (1 << i)]
    return h

# Example
n = 3
g = [1, 2, 3, 4, 5, 6, 7, 8] # g[mask]
h = sos_superset_sum(g, n)
print(h)
```

C

```
#include <stdio.h>

void sos_superset_sum(long long h[], int n){
    for(int i=0;i<n;i++){
        for(int S=0;S<(1<<n);S++){
            if((S&(1<<i))==0){
                h[S]+=h[S|(1<<i)];
            }
        }
    }
}

int main(){
    int n=3;
    long long h[1<<3]={1,2,3,4,5,6,7,8};
    sos_superset_sum(h,n);
    for(int i=0;i<(1<<n);i++) printf("%lld ", h[i]);
    return 0;
}
```

## Why It Matters

- Dual of subset SOS DP.
- Core for transforms like zeta and Möbius on the subset lattice.
- Useful for queries like: for each feature set  $S$ , aggregate values over all supersets that contain  $S$ .

## Step-by-Step Example

Let  $n = 2$ , masks 00, 01, 10, 11 and  $g = [1, 2, 3, 4]$ .

Process bit  $i = 0$ :

- $S = 00: h[00]+ = h[01] \Rightarrow 1 + 2 = 3$
- $S = 10: h[10]+ = h[11] \Rightarrow 3 + 4 = 7$

Now  $h = [3, 2, 7, 4]$ .

Process bit  $i = 1$ :

- $S = 00: h[00]+ = h[10] \Rightarrow 3 + 7 = 10$

- $S = 01$ :  $h[01]_+ = h[11] \Rightarrow 2 + 4 = 6$

Final  $h = [10, 6, 7, 4]$ , which matches

- $h[00] = g[00] + g[01] + g[10] + g[11] = 10$
- $h[01] = g[01] + g[11] = 6$
- $h[10] = g[10] + g[11] = 7$
- $h[11] = g[11] = 4$ .

### A Gentle Proof (Why It Works)

Fix a bit order. When processing bit  $i$ , for any  $S$  with bit  $i$  unset, every superset of  $S$  either keeps bit  $i$  off or turns it on. Before processing  $i$ ,  $h[S]$  accumulates supersets with bit  $i$  off. Adding  $h[S \cup i]$  brings in all supersets with bit  $i$  on. Induct over bits to conclude all supersets are included exactly once.

### Try It Yourself

1. Convert this to compute maximum over supersets by replacing plus with max.
2. Combine with subset SOS to precompute both directions for fast subset-superset queries.
3. Apply modulo arithmetic to prevent overflow.
4. Implement the Möbius inversion on supersets to invert the transform.
5. Extend to bitwise operations where aggregation depends on bit counts.

### Test Cases

| n | g input           | h output          |
|---|-------------------|-------------------|
| 2 | [1,2,3,4]         | [10,6,7,4]        |
| 3 | all 1s            | [8,4,4,2,4,2,2,1] |
| 3 | [0,1,2,3,4,5,6,7] | computed by code  |

### Complexity

- Time:  $O(n \cdot 2^n)$
- Space:  $O(2^n)$

SOS Superset DP is the natural mirror of subset SOS. Use it whenever queries demand aggregating over all sets that contain a given mask.

## 459 XOR Basis DP

The XOR Basis DP technique helps count or generate all possible XOR values from a set of numbers efficiently. It constructs a linear basis over GF(2) and enables solving problems like counting distinct XORs, finding minimum/maximum XOR, and combining with digit DP or bitmask states.

### What Problem Are We Solving?

Given a list of numbers  $A = [a_1, a_2, \dots, a_n]$ , we want to:

- Find how many distinct XOR values can be formed from subsets of  $A$ .
- Or find maximum/minimum possible XOR.
- Or answer queries on possible XOR combinations.

The XOR operation forms a vector space over  $\mathbb{F}_2$ , and each number contributes a vector. The XOR basis provides a compact representation of all subset XORs.

### Core Idea

Maintain an array **basis** representing independent bit vectors. Insert each number into the basis (Gaussian elimination over GF(2)):

- For each bit from high to low, if that bit is set and not represented, store the number.
- If it is already represented, XOR with the current basis vector to reduce it.

At the end, the number of independent vectors is the rank  $r$ , and the number of distinct XORs is  $2^r$ .

### DP Perspective

The state represents a basis built from a prefix of the array. You can define:

$$dp[i] = \text{XOR basis after processing first } i \text{ elements}$$

To count distinct XORs after all elements:

$$\text{count} = 2^{\text{rank}}$$

If you need to build combinations (e.g. count of XOR < M), combine basis construction with digit DP constraints.

## Tiny Code (Easy Versions)

Python

```
def xor_basis(arr):
    basis = []
    for x in arr:
        for b in basis:
            x = min(x, x ^ b)
        if x:
            basis.append(x)
    return basis

def count_distinct_xors(arr):
    basis = xor_basis(arr)
    return 1 << len(basis) # 2^rank

# Example
A = [3, 10, 5]
basis = xor_basis(A)
print("Basis:", basis)
print("Distinct XORs:", count_distinct_xors(A))
```

C

```
#include <stdio.h>

int insert_basis(int basis[], int *sz, int x) {
    for (int i = 0; i < *sz; i++) {
        if ((x ^ basis[i]) < x) x ^= basis[i];
    }
    if (x == 0) return 0;
    basis[(*sz)++] = x;
    return 1;
}

int main() {
    int arr[] = {3, 10, 5};
    int n = 3, basis[32], sz = 0;

    for (int i = 0; i < n; i++)
        insert_basis(basis, &sz, arr[i]);
```

```

    printf("Rank: %d\nDistinct XORs: %d\n", sz, 1 << sz);
}

```

## Why It Matters

- Forms the foundation for subset XOR problems.
- Used in:
  - Counting distinct XORs
  - Maximum XOR subset
  - XOR-constrained digit DP
  - Gaussian elimination in  $\mathbb{F}_2$

It's the bitwise analog of linear algebra, solving over  $\text{GF}(2)$ .

## Step-by-Step Example

Let  $A = [3, 10, 5]$ :

- Binary:  $3 = 011$ ,  $10 = 1010$ ,  $5 = 0101$
- Insert  $3 \rightarrow$  basis =  $\{3\}$
- Insert  $10 \rightarrow$  independent, basis =  $\{3, 10\}$
- Insert  $5 \rightarrow$  can be reduced:  $5 \oplus 3 = 6$ ,  $6 \oplus 10 = 12 \rightarrow$  independent, basis =  $\{3, 10, 5\}$

Rank  $r = 3$ , number of distinct XORs =  $2^3 = 8$ .

All subset XORs:

$0, 3, 5, 6, 10, 11, 12, 15$

## A Gentle Proof (Why It Works)

Each basis vector represents a new independent bit dimension. Every subset XOR corresponds to a linear combination over  $\mathbb{F}_2$  of the basis. If there are  $r$  independent vectors, there are  $2^r$  possible linear combinations (subsets), hence  $2^r$  distinct XORs.

## Try It Yourself

1. Modify to find maximum XOR subset (XOR greedily from MSB down).
2. Combine with digit DP to count numbers with XOR constraints ( $< M$ ).
3. Track reconstruction: which subset forms a target XOR.
4. Apply to path XOR queries in trees (via prefix basis merging).
5. Extend to multiset bases or online updates.

## Test Cases

| Input    | Output | Notes                 |
|----------|--------|-----------------------|
| [3,10,5] | 8      | 3 independent vectors |
| [1,2,3]  | 4      | Rank = 2              |
| [1,1,1]  | 2      | Rank = 1              |

## Complexity

- Time:  $O(n \cdot \text{bitwidth})$
- Space:  $O(\text{bitwidth})$

XOR Basis DP is the digital geometry of sets under XOR, every number a vector, every subset a linear combination, every question a path through binary space.

## 460 Digit DP for Palindromes

Digit DP for Palindromes counts all numbers within a given range that are palindromic, numbers that read the same forward and backward. It's a symmetric DP that constructs digits from both ends simultaneously, respecting tight bounds from the original number.

### What Problem Are We Solving?

Given two integers  $L$  and  $R$ , count the palindromes in  $[L, R]$ .

Example: in  $[1, 200]$ , palindromes are 1, 2, ..., 9, 11, 22, ..., 99, 101, 111, ..., 191, total 28.

Naively iterating and checking each number is  $O(N)$ , which is too slow for large  $N$ . We want an  $O(d \times 10^{d/2})$  approach using Digit DP with symmetry.

## Core Idea

We can build a number digit by digit from the outside in, ensuring the number remains a palindrome at each step.

For a given length  $len$ , we only need to choose digits for the first half; the second half is determined.

The tight constraints ensure we stay  $\leq R$  (or  $\leq L - 1$  for inclusive ranges).

## DP Definition

Let  $S = \text{digits of } N$ , length  $len$ .

State:

$dp[pos][tight][leading]$

Where:

- **pos** is the current index from the left half ( $0 \leq pos < \frac{len}{2}$ )
- **tight** means prefix equals  $N$  so far
- **leading** means we've placed only leading zeros

The recursion places one digit at **pos**, mirrors it at **len-1-pos**, and recurses inward.

## Transition

For each **digit** in  $[0, limit]$ :

- If **leading** and **digit==0**, we can skip counting it as a real digit.
- Mirror digit into the symmetric position.
- Update tightness if **digit == limit**.
- Recurse inward until halfway.

When reaching middle, count 1 valid palindrome.

## Algorithm Outline

To count palindromes  $N$ :

1. Convert  $N$  to string  $S$
2. Run `dfs(pos=0, tight=True, leading=True)`
3. If building full palindrome (not half-only), check mirrored structure

Count in  $[L, R]$  as:

$$f(R) - f(L - 1)$$

## Tiny Code (Easy Versions)

Python

```
from functools import lru_cache

def count_palindromes_upto(N):
    S = list(map(int, str(N)))
    n = len(S)

    @lru_cache(None)
    def dfs(pos, tight, leading, half):
        if pos == (n + 1) // 2:
            return 1 # one palindrome formed
        limit = S[pos] if tight else 9
        total = 0
        for d in range(limit + 1):
            if leading and d == 0:
                total += dfs(pos + 1, tight and d == limit, True, half + [0])
            else:
                total += dfs(pos + 1, tight and d == limit, False, half + [d])
        return total

    return dfs(0, True, True, ())

def count_palindromes(L, R):
    return count_palindromes_upto(R) - count_palindromes_upto(L - 1)

# Example
print(count_palindromes(1, 200))
```

C (half-construction)

```
#include <stdio.h>
#include <string.h>

long long count_palindromes_upto(long long N) {
    if (N < 0) return 0;
    char s[20];
    sprintf(s, "%lld", N);
    int len = strlen(s);
    int half = (len + 1) / 2;
    long long count = 0;

    // Build prefix half and mirror
    for (int mask = 0; mask < (1 << (half * 4)); mask++) {
        // Conceptual only, use recursion or base-10 enumeration for actual code
    }

    // Simpler approach: iterate half, build full palindrome, check N
    long long start = 1;
    for (int i = 1; i <= len; i++) {
        int half_len = (i + 1) / 2;
        long long base = 1;
        for (int j = 1; j < half_len; j++) base *= 10;
        for (long long x = base; x < base * 10; x++) {
            long long y = x;
            if (i % 2) y /= 10;
            long long z = x;
            while (y > 0) {
                z = z * 10 + (y % 10);
                y /= 10;
            }
            if (z <= N) count++;
        }
    }
    return count;
}

int main(void) {
    long long L = 1, R = 200;
    printf("%lld\n", count_palindromes_upto(R) - count_palindromes_upto(L - 1));
}
```

## Why It Matters

- Palindromic counting appears in:
  - Digit constraints (e.g. special number sets)
  - Symmetric number combinatorics
  - Patterned sequence generation
- Builds intuition for bidirectional DP where digits mirror.

## Step-by-Step Example

Let  $N = 200$ :

- Length 3  $\rightarrow$  half = 2
- Choose first half (00–19), mirror  $\rightarrow$  00–99  $\rightarrow$  0, 11, 22, ..., 191
- Apply bounds: only 200  $\rightarrow$  count = 28

## A Gentle Proof (Why It Works)

Every palindrome is uniquely determined by its first half. For each valid half respecting the upper bound, exactly one mirrored number exists. Tightness ensures no overflow beyond  $N$ . Leading-zero handling ensures numbers like 00100 are excluded.

## Try It Yourself

1. Count even-length palindromes only.
2. Count palindromes with a fixed digit sum.
3. Modify to count palindromic primes by adding primality check.
4. Combine with digit constraints (e.g. no 3s).
5. Count numbers that become palindromes after reversal operations.

## Test Cases

| L   | R   | Output | Notes                    |
|-----|-----|--------|--------------------------|
| 1   | 9   | 9      | Single-digit palindromes |
| 10  | 99  | 9      | 11, 22, ..., 99          |
| 1   | 200 | 28     | Up to 191                |
| 100 | 999 | 90     | All 3-digit palindromes  |

## Complexity

- Time:  $O(d \times 10^{d/2})$
- Space:  $O(d)$

Digit DP for Palindromes bridges arithmetic and symmetry, constructing mirrored structures digit by digit under bound constraints.

# Section 47. DP Optimizations

## 461 Divide & Conquer DP (Monotone Optimization)

When a DP transition is a 1D convolution of the form  $dp[i][j] = \min_{k < j} (dp[i-1][k] + C(k, j))$ , and the argmin is monotone ( $opt[i][j] \leq opt[i][j+1]$ ), you can compute a whole layer  $dp[i][*]$  in  $O(n)$  splits with divide and conquer instead of  $O(n^2)$ . Typical total is  $O(K, n \log n)$  or  $O(K, n)$  depending on implementation.

### What Problem Are We Solving?

Speed up DP layers with transitions

$$dp[j] = \min_{k < j} (prev[k] + C(k, j)),$$

for  $j$  in an interval, where the optimal index  $opt[j] \in [optL, optR]$  and satisfies

$$opt[j] \leq opt[j+1] \quad (\text{monotone decision property}).$$

This structure appears in:

- $K$ -partitioning of arrays with convex segment cost
- 1D facility placement and line breaking with convex penalties
- Some shortest path on DAG layers with convex arc costs

### How Does It Work (Plain Language)

Compute the current DP layer on a segment  $[L, R]$  by solving the midpoint  $M$ , searching its best  $k$  only in  $[optL, optR]$ . The best index  $opt[M]$  splits the problem:

- Left half  $[L, M - 1]$  only needs candidates in  $[optL, opt[M]]$
- Right half  $[M + 1, R]$  only needs  $[opt[M], optR]$

Recursively repeat until intervals are size 1. Monotonicity guarantees these candidate ranges.

## Preconditions checklist

You can use divide and conquer DP if:

1. Transition is  $dp[j] = \min_{k < j} (prev[k] + C(k, j))$ .
2. The optimal index is monotone in  $j$ . A sufficient condition is quadrangle inequality or Monge property of  $C$ :

$$C(a, c) + C(b, d) \leq C(a, d) + C(b, c) \quad \text{for } a \leq b \leq c \leq d.$$

## Tiny Code (Template)

C++ style pseudocode (drop in C with minor edits)

```
// Compute one layer dp_cur[lo..hi], given dp_prev and cost C(k,j).
// Assumes optimal indices are monotone.
void compute(int lo, int hi, int optL, int optR,
            const vector<long long>& dp_prev,
            vector<long long>& dp_cur,
            auto&& cost) {
    if (lo > hi) return;
    int mid = (lo + hi) >> 1;

    long long best = LLONG_MAX;
    int best_k = -1;
    int start = optL, end = min(optR, mid - 1);
    for (int k = start; k <= end; ++k) {
        long long val = dp_prev[k] + cost(k, mid);
        if (val < best) {
            best = val;
            best_k = k;
        }
    }
    dp_cur[mid] = best;

    // Recurse with narrowed opt ranges
    compute(lo, mid - 1, optL, best_k, dp_prev, dp_cur, cost);
    compute(mid + 1, hi, best_k, optR, dp_prev, dp_cur, cost);
}
```

Python (clear and compact)

```

INF = 1018

def compute(lo, hi, optL, optR, dp_prev, dp_cur, cost):
    if lo > hi:
        return
    mid = (lo + hi) // 2
    best_val, best_k = INF, -1
    end = min(optR, mid - 1)
    for k in range(optL, end + 1):
        v = dp_prev[k] + cost(k, mid)
        if v < best_val:
            best_val, best_k = v, k
    dp_cur[mid] = best_val
    compute(lo, mid - 1, optL, best_k, dp_prev, dp_cur, cost)
    compute(mid + 1, hi, best_k, optR, dp_prev, dp_cur, cost)

```

How to use For  $i = 1..K$ : call `compute(1,n,0,n-1, dp_prev, dp_cur, cost)` then swap layers. Index ranges depend on your base cases.

### Example: K partitions with convex segment cost

Given array  $a[1..n]$ , let prefix sums  $S[j] = \sum_{t=1}^j a[t]$ . Suppose segment cost is

$$C(k, j) = (S[j] - S[k])^2,$$

which is convex and satisfies quadrangle inequality. The DP

$$dp[i][j] = \min_{k < j} (dp[i-1][k] + C(k, j))$$

has monotone argmins, so one layer can be computed with the template. Total roughly  $O(K, n \log n)$ .

### Why It Matters

- Cuts a quadratic DP layer down to near linear
- Simple to implement compared to more advanced tricks
- Pairs well with prefix sums for  $C(k, j)$  evaluation
- Core technique in editors line breaking, clustering in 1D, histogram smoothing

## Step by Step on a small instance

Let  $a = [1, 3, 2, 4]$ ,  $K = 2$ ,  $C(k, j) = (S[j] - S[k])^2$ .

1. Initialize  $dp[0][0] = 0$ ,  $dp[0][j > 0] = +\infty$ .
2. Layer  $i = 1$ : compute  $dp[1][j]$  by scanning  $k < j$ . Argmins are nondecreasing.
3. Layer  $i = 2$ : call `compute(1, n, 0, n-1, ...)`. The recursion halves the target interval and narrows candidate  $k$  ranges by monotonicity.

## A Gentle Proof (Why It Works)

Let  $opt[j]$  be a minimizer for  $dp[j]$ . If  $C$  is Monge, then for  $x < y$  and  $u < v$ :

$$C(u, x) + C(v, y) \leq C(u, y) + C(v, x).$$

Assume by contradiction  $opt[x] > opt[y]$ . Using optimality of those indices and the inequality above yields a contradiction. Hence  $opt$  is nondecreasing.

The recursion evaluates  $dp[mid]$  using candidates in  $[optL, optR]$ . The found  $opt[mid]$  splits feasible candidates:

- Any optimal index for  $j < mid$  is in  $[optL, opt[mid]]$
- Any optimal index for  $j > mid$  is in  $[opt[mid], optR]$  Induct over segments to show each  $dp[j]$  is computed with exactly the needed candidate set and no essential candidate is excluded.

## Try It Yourself

1. Replace  $C(k, j)$  by  $\alpha, (S[j] - S[k])^2 + \beta, (j - k)$  and verify monotonicity still holds.
2. Use the template to speed up line breaking with raggedness penalty.
3. Benchmark naive  $O(n^2)$  vs divide and conquer on random convex costs.
4. Combine with space optimization by keeping only two layers.
5. Contrast with Knuth optimization and Convex Hull Trick and decide which applies for your cost.

## Test Cases

- Small convex cost:
  - $a = [2, 1, 3], K = 2, C(k, j) = (S[j] - S[k])^2$
  - Compare naive and D&C outputs, they must match.
- Linear plus convex mix:

–  $a = [1, 1, 1, 1]$ ,  $K = 3$ ,  $C(k, j) = (S[j] - S[k])^2 + (j - k)$

- Edge cases:

–  $n = 1$  any  $K \geq 1$   
– All zeros array, any  $K$

## Complexity

- One layer with divide and conquer:  $O(n \log n)$  evaluations of  $C$  in the simple form, often written as  $O(n)$  splits with logarithmic recursion depth.
- Full DP:  $O(K, n \log n)$  time,  $O(n)$  or  $O(K, n)$  space depending on whether you store all layers.

Divide and conquer DP is your go to when the argmin slides to the right as  $j$  grows. It is small code, big speedup.

## 462 Knuth Optimization

Knuth Optimization is a special-case speedup for certain DP transitions where quadrangle inequality and monotonicity of optimal decisions hold. It improves  $O(n^2)$  dynamic programs to  $O(n)$  per layer, often used in optimal binary search tree, matrix chain, and interval partitioning problems.

### What Problem Are We Solving?

We want to optimize DP recurrences of the form:

$$dp[i][j] = \min_{k \in [i, j-1]} (dp[i][k] + dp[k+1][j] + C(i, j))$$

for  $1 \leq i \leq j \leq n$ , where  $C(i, j)$  is a cost function satisfying quadrangle inequality.

Naively, this is  $O(n^3)$ . With Knuth optimization, we cut it to  $O(n^2)$  by restricting the search range using a monotonic property of the optimal split point.

## Key Condition

Knuth optimization applies when:

1. Quadrangle Inequality:

$$C(a, c) + C(b, d) \leq C(a, d) + C(b, c), \quad \forall a \leq b \leq c \leq d$$

2. Monotonicity of Argmin:

$$opt[i][j-1] \leq opt[i][j] \leq opt[i+1][j]$$

These ensure that the optimal split  $k$  for  $[i, j]$  moves rightward as intervals slide.

## How Does It Work (Plain Language)

When computing  $dp[i][j]$ , the best partition point  $k$  lies between  $opt[i][j-1]$  and  $opt[i+1][j]$ . So instead of scanning the full  $[i, j-1]$ , we limit to a narrow window. This reduces work from  $O(n^3)$  to  $O(n^2)$ .

We fill intervals in increasing length order, maintaining and reusing  $\text{opt}[i][j]$ .

## Step-by-Step Recurrence

For all  $i$ :

$$dp[i][i] = 0$$

Then for lengths  $len = 2..n$ :

```
for i in 1..n-len+1:  
    j = i + len - 1  
    dp[i][j] = ∞  
    for k in opt[i][j-1]..opt[i+1][j]:  
        val = dp[i][k] + dp[k+1][j] + C(i, j)  
        if val < dp[i][j]:  
            dp[i][j] = val  
            opt[i][j] = k
```

## Tiny Code (Easy Versions)

Python

```

INF = 1018

def knuth_optimization(n, cost):
    dp = [[0]*n for _ in range(n)]
    opt = [[0]*n for _ in range(n)]

    for i in range(n):
        opt[i][i] = i

    for length in range(2, n+1):
        for i in range(0, n-length+1):
            j = i + length - 1
            dp[i][j] = INF
            start = opt[i][j-1]
            end = opt[i+1][j] if i+1 <= j else j-1
            for k in range(start, end+1):
                val = dp[i][k] + dp[k+1][j] + cost(i, j)
                if val < dp[i][j]:
                    dp[i][j] = val
                    opt[i][j] = k
    return dp[0][n-1]

```

C

```

#include <stdio.h>
#include <limits.h>

#define INF 1000000000000000LL
#define N 505

long long dp[N][N], opt[N][N];

long long cost(int i, int j); // user-defined cost function

long long knuth(int n) {
    for (int i = 0; i < n; i++) {
        dp[i][i] = 0;
        opt[i][i] = i;
    }
    for (int len = 2; len <= n; len++) {
        for (int i = 0; i + len - 1 < n; i++) {
            int j = i + len - 1;

```

```

dp[i][j] = INF;
int start = opt[i][j-1];
int end = opt[i+1][j];
if (end == 0) end = j - 1;
if (start > end) { int tmp = start; start = end; end = tmp; }
for (int k = start; k <= end; k++) {
    long long val = dp[i][k] + dp[k+1][j] + cost(i, j);
    if (val < dp[i][j]) {
        dp[i][j] = val;
        opt[i][j] = k;
    }
}
}
return dp[0][n-1];
}

```

## Why It Matters

Knuth Optimization turns a cubic DP into quadratic without approximations. It's especially useful for:

- Optimal Binary Search Trees (OBST)
- Matrix Chain Multiplication
- Merging Stones / File Merging
- Bracket Parsing / Partitioning

It's a precise algebraic optimization based on Monge arrays and convexity.

## Step-by-Step Example

Consider merging files with sizes [10, 20, 30].  $C(i, j) = \text{sum of file sizes from } i \text{ to } j$ . We fill  $dp[i][j]$  with minimal total cost of merging segment  $[i..j]$ . Argmins move monotonically, so Knuth optimization applies.

## A Gentle Proof (Why It Works)

If  $C$  satisfies quadrangle inequality:

$$C(a, c) + C(b, d) \leq C(a, d) + C(b, c),$$

then combining two adjacent subproblems will never cause the optimal cut to move left. Hence,  $opt[i][j-1] \leq opt[i][j] \leq opt[i+1][j]$ . Thus, restricting the search range preserves correctness while cutting redundant checks.

### Try It Yourself

1. Apply Knuth optimization to Optimal BST:

$$dp[i][j] = \min_{k \in [i,j]} (dp[i][k-1] + dp[k+1][j] + w[i][j])$$

2. Use it in Merging Stones (sum-cost merge).
3. Compare with Divide & Conquer DP, both need monotonicity, but Knuth's has fixed quadratic structure.
4. Verify monotonicity by printing  $opt[i][j]$ .
5. Prove  $C(i, j) = \text{prefix}[j] - \text{prefix}[i-1]$  satisfies the condition.

### Test Cases

| Case           | Description              | Expected Complexity |
|----------------|--------------------------|---------------------|
| File merging   | [10, 20, 30]             | $O(n^2)$            |
| Optimal BST    | Sorted keys, frequencies | $O(n^2)$            |
| Merging Stones | Equal weights            | Monotone $opt$      |

### Complexity

- Time:  $O(n^2)$
- Space:  $O(n^2)$

Knuth Optimization is the elegant midpoint between full DP and convexity tricks, precise, predictable, and optimal whenever cost satisfies Monge structure.

## 463 Convex Hull Trick (CHT)

The Convex Hull Trick speeds up DP transitions of the form

$$dp[i] = \min_{k < i} (dp[k] + m_k \cdot x_i + b_k)$$

when the slopes  $m_k$  are monotonic (increasing or decreasing) and the  $x_i$  queries are also monotonic. It replaces  $O(n^2)$  scanning with  $O(n)$  amortized or  $O(\log n)$  query time using a dynamic hull.

## What Problem Are We Solving?

We have a DP recurrence like:

$$dp[i] = \min_{k < i} (dp[k] + m_k \cdot x_i + b_k)$$

where:

- $m_k$  (slope) and  $b_k$  (intercept) define lines,
- $x_i$  is the query coordinate,
- we want the minimum (or maximum) value over all  $k$ .

This appears in:

- Line DP (e.g. segmented linear costs),
- Divide and Conquer DP (convex variant),
- Knuth-like DPs with linear penalties,
- Aliens trick and Li Chao Trees for non-monotone cases.

## When It Applies

1. Transition fits  $dp[i] = \min_k (dp[k] + m_k x_i + b_k)$
2.  $m_k$ 's are monotonic (non-decreasing or non-increasing)
3.  $x_i$  queries are sorted (non-decreasing)

Then you can use a deque-based CHT for  $O(1)$  amortized per insertion/query.

If slopes or queries are not monotonic, use Li Chao Tree (next algorithm).

## How Does It Work (Plain Language)

Each  $k$  defines a line  $y = m_k x + b_k$ . The DP asks: for current  $x_i$ , which previous line gives the smallest value? All lines together form a lower envelope, a piecewise minimum curve. We maintain this hull incrementally and query the minimum efficiently.

If slopes are sorted, each new line intersects the previous hull at one point, and old lines become useless after their intersection point.

## Tiny Code (Easy Versions)

Python (Monotone CHT)

```

class CHT:
    def __init__(self):
        self.lines = [] # (m, b)

    def bad(self, l1, l2, l3):
        # Check if l2 is unnecessary between l1 and l3
        return (l3[1] - l1[1]) * (l1[0] - l2[0]) <= (l2[1] - l1[1]) * (l1[0] - l3[0])

    def add(self, m, b):
        self.lines.append((m, b))
        while len(self.lines) >= 3 and self.bad(self.lines[-3], self.lines[-2], self.lines[-1]):
            self.lines.pop(-2)

    def query(self, x):
        # queries x in increasing order
        while len(self.lines) >= 2 and \
                self.lines[0][0]*x + self.lines[0][1] >= self.lines[1][0]*x + self.lines[1][1]:
            self.lines.pop(0)
        m, b = self.lines[0]
        return m*x + b

# Example: dp[i] = min(dp[k] + m[k]*x[i] + b[k])
dp = [0]*5
x = [1, 2, 3, 4, 5]
m = [2, 1, 3, 4, 5]
b = [5, 4, 3, 2, 1]

cht = CHT()
cht.add(m[0], b[0])
for i in range(1, 5):
    dp[i] = cht.query(x[i-1])
    cht.add(m[i], dp[i] + b[i])

```

C (Deque Implementation)

```

#include <stdio.h>

typedef struct { long long m, b; } Line;
Line hull[100005];
int sz = 0, ptr = 0;

double intersect(Line a, Line b) {

```

```

        return (double)(b.b - a.b) / (a.m - b.m);
    }

int bad(Line a, Line b, Line c) {
    return (c.b - a.b)*(a.m - b.m) <= (b.b - a.b)*(a.m - c.m);
}

void add_line(long long m, long long b) {
    Line L = {m, b};
    while (sz >= 2 && bad(hull[sz-2], hull[sz-1], L)) sz--;
    hull[sz++] = L;
}

long long query(long long x) {
    while (ptr + 1 < sz && hull[ptr+1].m * x + hull[ptr+1].b <= hull[ptr].m * x + hull[ptr].b)
        ptr++;
    return hull[ptr].m * x + hull[ptr].b;
}

```

## Why It Matters

- Reduces  $O(n^2)$  DP with linear transition to  $O(n)$  or  $O(n \log n)$ .
- Extremely common in optimization tasks:
  - Convex cost partitioning
  - Slope trick extensions
  - Dynamic programming with linear penalties
- Foundation for Li Chao Trees and Slope Trick.

## Step-by-Step Example

Suppose

$$dp[i] = \min_{k < i} (dp[k] + a_k \cdot b_i + c_k)$$

Given:

- $a = [2, 4, 6]$
- $b = [1, 2, 3]$
- $c = [5, 4, 2]$

Each step:

1. Add line  $y = m_k x + b_k = a_k x + (dp[k] + c_k)$
2. Query at  $x_i = b[i]$  to get min value.

CHT keeps only useful lines forming lower envelope.

### A Gentle Proof (Why It Works)

For monotonic slopes, intersection points are sorted. Once a line becomes worse than the next at a certain  $x$ , it will never be optimal again for larger  $x$ . Therefore, we can pop it from the deque — each line enters and leaves once  $\rightarrow O(n)$  amortized.

### Try It Yourself

1. Adapt for maximum query (flip signs).
2. Combine with DP:  $dp[i] = \min_k(dp[k] + m_k x_i + b_k)$ .
3. Add Li Chao Tree for unsorted slopes/queries.
4. Visualize lower envelope intersection points.
5. Compare with Slope Trick (piecewise-linear potentials).

### Test Cases

| m       | b       | x | Expected min |
|---------|---------|---|--------------|
| [1,2,3] | [0,1,3] | 1 | 1            |
| [2,1]   | [5,4]   | 2 | 6            |
| [1,3,5] | [2,2,2] | 4 | 14           |

### Complexity

- Time:  $O(n)$  amortized (monotone queries) or  $O(n \log n)$  (Li Chao)
- Space:  $O(n)$

Convex Hull Trick is the bridge between geometry and DP, every line a subproblem, every envelope a minimization frontier.

### 464 Li Chao Tree

The Li Chao Tree is a dynamic data structure for maintaining a set of lines and efficiently querying the minimum (or maximum) value at any given  $x$ . Unlike the Convex Hull Trick, it works even when slopes and query points are arbitrary and unordered.

## What Problem Are We Solving?

We want to handle DP recurrences (or cost functions) of the form:

$$dp[i] = \min_{j < i} (m_j \cdot x_i + b_j)$$

Each  $j$  contributes a line  $y = m_j x + b_j$ . We must find the minimum value among all added lines at a given  $x_i$ .

Unlike the Convex Hull Trick (which needs monotonic  $x$  or  $m$ ), Li Chao Tree handles any insertion or query order, no sorting required.

## When It Applies

Li Chao Tree applies when:

- You need to add arbitrary lines ( $m_j, b_j$ ) over time
- You must query arbitrary  $x$  in any order
- You want min or max queries efficiently
- Slopes and queries are not monotonic

This makes it ideal for:

- DP with arbitrary slopes
- Online queries
- Geometry problems involving lower envelopes
- Line container queries in computational geometry

## How Does It Work (Plain Language)

The Li Chao Tree divides the  $x$ -axis into segments. Each node represents an interval, storing one line that's currently optimal over part (or all) of that range.

When a new line is added:

- Compare it with the current line in the interval
- Swap if it's better at the midpoint
- Recursively insert into one child (left/right), narrowing the range

When querying:

- Descend the tree using  $x$
- Combine values of lines encountered

- Return the minimum (or maximum)

This yields  $O(\log X)$  time per insertion and query, where  $X$  is the range of  $x$  values (discretized if necessary).

### Tiny Code (Easy Version)

Python (Min Version)

```
INF = 1018

class Line:
    def __init__(self, m, b):
        self.m = m
        self.b = b
    def value(self, x):
        return self.m * x + self.b

class Node:
    def __init__(self, l, r):
        self.l = l
        self.r = r
        self.line = None
        self.left = None
        self.right = None

class LiChaoTree:
    def __init__(self, l, r):
        self.root = Node(l, r)

    def _add(self, node, new_line):
        l, r = node.l, node.r
        m = (l + r) // 2
        if node.line is None:
            node.line = new_line
            return

        left_better = new_line.value(l) < node.line.value(l)
        mid_better = new_line.value(m) < node.line.value(m)

        if mid_better:
            node.line, new_line = new_line, node.line
```

```

        if r - l == 0:
            return
        if left_better != mid_better:
            if not node.left:
                node.left = Node(l, m)
                self._add(node.left, new_line)
            else:
                if not node.right:
                    node.right = Node(m + 1, r)
                    self._add(node.right, new_line)

    def add_line(self, m, b):
        self._add(self.root, Line(m, b))

    def _query(self, node, x):
        if node is None:
            return INF
        res = node.line.value(x) if node.line else INF
        m = (node.l + node.r) // 2
        if x <= m:
            return min(res, self._query(node.left, x))
        else:
            return min(res, self._query(node.right, x))

    def query(self, x):
        return self._query(self.root, x)

# Example usage
tree = LiChaoTree(0, 100)
tree.add_line(2, 3)
tree.add_line(-1, 10)
print(tree.query(5)) # minimum value among all lines at x=5

```

## Why It Matters

- Handles arbitrary slopes and queries
- Efficient for online DP and geometry optimization
- Generalizes CHT (works without monotonic constraints)
- Can be used for both min and max queries (just flip inequalities)

## Step-by-Step Example

Suppose we insert lines:

1.  $y = 2x + 3$
2.  $y = -x + 10$

Then query at  $x = 5$ :

- First line:  $2(5) + 3 = 13$
- Second line:  $-5 + 10 = 5 \rightarrow \text{answer} = 5$

The tree ensures each query returns the best line without brute force.

## A Gentle Proof (Why It Works)

Each interval stores a line that's locally optimal at some midpoint. If a new line is better at one endpoint, it must eventually overtake the existing line, so the intersection lies in that half.

By recursing on halves, we ensure the correct line is chosen for every  $x$ .

The tree height is  $\log X$ , and each insertion affects at most  $\log X$  nodes.

Hence:

$$T(n) = O(n \log X)$$

## Try It Yourself

1. Implement Li Chao Tree for max queries (invert comparisons).
2. Add  $n$  random lines and query random  $x$ .
3. Apply it to DP with linear cost:

$$dp[i] = \min_{j < i}(dp[j] + a_j x_i + b_j)$$

4. Visualize segment splits and stored lines.
5. Compare with Convex Hull Trick on monotonic test cases.

## Test Cases

| Lines | Queries | Range    | Time          | Works |
|-------|---------|----------|---------------|-------|
| 10    | 10      | [0, 100] | $O(n \log X)$ |       |
| 1e5   | 1e5     | [0, 1e9] | $O(n \log X)$ |       |

## Complexity

- Time:  $O(\log X)$  per insert/query
- Space:  $O(\log X)$  per line (tree nodes)

The Li Chao Tree is your line oracle, always ready to give the best line, no matter how chaotic your slopes and queries become.

## 465 Slope Trick

The Slope Trick is a dynamic programming optimization technique for problems involving piecewise-linear convex functions. It lets you maintain and update the shape of a convex cost function efficiently, especially when transitions involve operations like adding absolute values, shifting minima, or combining convex shapes.

### What Problem Are We Solving?

Many DP problems involve minimizing a cost function that changes shape over time, such as:

$$dp[i] = \min_x(dp[i-1](x) + |x - a_i|)$$

Here,  $dp[i]$  is not a single value but a function of  $x$ . The Slope Trick is how we maintain this function efficiently as a sequence of linear segments.

This comes up when:

- You need to add  $|x - a|$  terms
- You need to shift the whole function left or right
- You need to add constants or merge minima

Rather than storing the full function, we store key “breakpoints” and update in logarithmic time.

### When It Applies

Slope Trick applies when:

- The cost function is convex and piecewise linear
- Each transition is of the form:
  - $f(x) + |x - a|$

- $f(x + c)$  or  $f(x - c)$
- $\min_x(f(x)) + c$

- You need to track minimal cost across shifting choices

Common in:

- Median DP
- Path alignment
- Convex smoothing
- Minimizing sum of absolute differences
- Cost balancing problems

### How Does It Work (Plain Language)

Instead of recomputing the entire cost function each time, we maintain two priority queues (heaps) that track where the slope changes.

Think of the cost function as a mountain made of straight lines:

- Adding  $|x - a|$  means putting a “tent” centered at  $a$
- Moving  $x$  left/right shifts the mountain
- The minimum point can move but remains easy to find

We track the slope’s left and right breakpoints using heaps:

- Left heap (max-heap): stores slopes to the left of minimum
- Right heap (min-heap): stores slopes to the right

Each operation updates these heaps in  $O(\log n)$ .

### Example Problem

Minimize:

$$dp[i] = \min_x(dp[i-1](x) + |x - a_i|)$$

We want the minimal total distance to all  $a_1, a_2, \dots, a_i$ .

The optimal  $x$  is the median of all  $a$ ’s seen so far.

Slope Trick maintains this function efficiently.

## Tiny Code (Easy Version)

Python

```
import heapq

class SlopeTrick:
    def __init__(self):
        self.left = [] # max-heap (store negative values)
        self.right = [] # min-heap
        self.min_cost = 0

    def add_abs(self, a):
        if not self.left:
            heapq.heappush(self.left, -a)
            heapq.heappush(self.right, a)
            return
        if a < -self.left[0]:
            heapq.heappush(self.left, -a)
            val = -heapq.heappop(self.left)
            heapq.heappush(self.right, val)
            self.min_cost += -self.left[0] - a
        else:
            heapq.heappush(self.right, a)
            val = heapq.heappop(self.right)
            heapq.heappush(self.left, -val)
            self.min_cost += a - val

    def get_min(self):
        return self.min_cost

# Example
st = SlopeTrick()
for a in [3, 1, 4, 1, 5]:
    st.add_abs(a)
print("Minimum cost:", st.get_min())
```

This structure efficiently tracks the cost of minimizing the sum of absolute differences.

## Why It Matters

- Reduces function-based DPs into heap updates

- Elegant solution for convex minimization
- Handles  $|x - a|$ , shift, and constant add in  $O(\log n)$
- Avoids discretization of continuous  $x$

### Step-by-Step Example

Suppose we add points sequentially:  $a = [3, 1, 4]$

1. Add  $|x - 3|$ : min at  $x = 3$
2. Add  $|x - 1|$ : min moves to  $x = 2$
3. Add  $|x - 4|$ : min moves to  $x = 3$

Heaps track these balance points dynamically. Total cost is sum of minimal shifts.

### A Gentle Proof (Why It Works)

Adding  $|x - a|$  modifies slope:

- For  $x < a$ , slope increases by  $-1$
- For  $x > a$ , slope increases by  $+1$

Thus, the function stays convex. Heaps store where slope crosses zero (the minimum).

Balancing heaps keeps slopes equalized, ensuring minimum at the median. Each operation maintains convexity and updates cost correctly.

### Try It Yourself

1. Implement `add_shift(c)` to shift function horizontally.
2. Solve:

$$dp[i] = \min_x(dp[i-1](x) + |x - a_i|)$$

for a list of  $a_i$

3. Add `add_constant(c)` for vertical shifts.
4. Track the running median using heaps.
5. Visualize slope evolution, it should always form a “V” shape.

| Input | Expected Minimum |
|-------|------------------|
|-------|------------------|

## Test Cases

| Input        | Expected Minimum |
|--------------|------------------|
| [3]          | 0                |
| [3, 1]       | 2                |
| [3, 1, 4]    | 3                |
| [3, 1, 4, 1] | 5                |

## Complexity

- Time:  $O(n \log n)$
- Space:  $O(n)$

The Slope Trick is like origami for DP, you fold and shift convex functions to shape the minimal path, one segment at a time.

## 466 Monotonic Queue Optimization

Monotonic Queue Optimization is a dynamic programming acceleration technique for recurrences involving sliding windows or range-limited minima. It replaces naive scanning ( $O(nk)$ ) with a monotonic deque that finds optimal states in  $O(n)$ .

### What Problem Are We Solving?

We want to optimize DP of the form:

$$dp[i] = \min_{j \in [i-k, i-1]} (dp[j] + cost(j, i))$$

or simpler, when  $cost(j, i)$  is monotonic or separable, like  $w_i$  or  $c(i - j)$ , we can maintain a window of candidate  $j$ 's.

This pattern appears in:

- Sliding window DPs
- Shortest path in DAGs with window constraints
- Queue scheduling problems
- Constrained subsequence or segment DPs

## When It Applies

You can apply Monotonic Queue Optimization when:

- The transition uses contiguous ranges of  $j$  (like a window)
- The cost function is monotonic, allowing pruning of bad states
- You want to find min or max over a sliding window efficiently

Common forms:

- $dp[i] = \min_{j \in [i-k, i]} (dp[j] + c[j])$
- $dp[i] = \max_{j \in [i-k, i]} (dp[j] + w[i])$

This trick does not require convexity, only monotonic ordering in the transition range.

## How Does It Work (Plain Language)

Instead of checking all  $k$  previous states for each  $i$ , we maintain a deque of indices that are still potentially optimal.

At each step:

1. Remove old indices (outside window)
2. Pop worse states (whose value is greater than the new one)
3. Front of deque gives the best  $j$  for current  $i$

This ensures the deque is monotonic (increasing or decreasing depending on min/max).

## Example Recurrence

$$dp[i] = \min_{j \in [i-k, i-1]} (dp[j] + w_i)$$

Since  $w_i$  doesn't depend on  $j$ , we just need  $\min dp[j]$  over the last  $k$  indices.

## Tiny Code (Easy Version)

Python

```

from collections import deque

def min_sliding_window_dp(arr, k):
    n = len(arr)
    dp = [0] * n
    dq = deque()

    for i in range(n):
        # Remove out-of-window
        while dq and dq[0] < i - k:
            dq.popleft()
        # Pop worse elements
        while dq and dp[dq[-1]] >= dp[i - 1] if i > 0 else False:
            dq.pop()
        # Push current
        dq.append(i)
        # Compute dp
        dp[i] = arr[i] + (dp[dq[0]] if dq else 0)
    return dp

```

C

```

#include <stdio.h>
#define N 100000
#define INF 10000000000

int dp[N], a[N], q[N];

int main() {
    int n = 5, k = 2;
    int front = 0, back = 0;

    int arr[5] = {3, 1, 4, 1, 5};
    dp[0] = arr[0];
    q[back++] = 0;

    for (int i = 1; i < n; i++) {
        while (front < back && q[front] < i - k) front++;
        dp[i] = dp[q[front]] + arr[i];
        while (front < back && dp[q[back - 1]] >= dp[i]) back--;
        q[back++] = i;
    }
}

```

```

    for (int i = 0; i < n; i++) printf("%d ", dp[i]);
}

```

## Why It Matters

- Converts range-min DPs from  $O(nk) \rightarrow O(n)$
- Essential for problems with window constraints
- Avoids heap overhead (constant-time updates)
- Extremely simple and robust

## Step-by-Step Example

Let  $arr = [3, 1, 4, 1, 5]$ ,  $k = 2$

At  $i = 2$ :

- Candidates:  $j \in [0, 1]$
- $dp[0]=3$ ,  $dp[1]=4$
- $dq = [1]$  after pruning worse values
- $dp[2] = arr[2] + dp[1] = 4 + 4 = 8$

Deque moves as window slides, always holding potential minima.

## A Gentle Proof (Why It Works)

At each  $i$ :

- Remove indices  $< i - k$  (out of range)
- Maintain monotonic order of dp-values in deque
- The front always gives the smallest  $dp[j]$  in window
- Because each element is pushed and popped once, total operations =  $O(n)$

Thus, overall complexity is linear.

## Try It Yourself

1. Implement max version by reversing comparisons.
2. Apply to  $dp[i] = \min_{j \in [i-k, i]} (dp[j] + c_i)$
3. Visualize deque evolution per step.
4. Solve constrained path problems with limited jump size.
5. Compare runtime with naive  $O(nk)$  approach.

## Test Cases

| Input        | k | Expected    |
|--------------|---|-------------|
| [3,1,4,1,5]  | 2 | fast min DP |
| [10,9,8,7,6] | 3 | decreasing  |
| [1,2,3,4,5]  | 1 | simple      |

## Complexity

- Time:  $O(n)$
- Space:  $O(k)$

Monotonic Queue Optimization is your sliding window oracle, keeping just the right candidates, and tossing the rest without looking back.

## 467 Bitset DP

Bitset DP is a performance optimization technique that uses bit-level parallelism to speed up dynamic programming, especially when state transitions involve Boolean operations over large ranges. By representing states as bits, multiple transitions can be processed simultaneously using fast bitwise operators.

### What Problem Are We Solving?

We want to optimize DPs like:

$$dp[i] = \text{reachable states after considering first } i \text{ elements}$$

Often in subset sum, knapsack, path existence, or mask propagation, we deal with states where:

- Each state is true/false
- Transition is shifting or combining bits

For example, in Subset Sum:

$$dp[i][s] = dp[i-1][s] \vee dp[i-1][s - a_i]$$

We can compress this into a bitset shift.

## When It Applies

You can use Bitset DP when:

- States are Boolean (true/false)
- Transition is shift-based or additive
- State space is dense and bounded

Common use cases:

- Subset Sum ( $O(nS/w)$ )
- Bounded Knapsack
- Graph reachability
- Palindromic substrings DP
- Counting with bit masks

Here,  $w$  is word size (e.g. 64), giving up to 64x speedup.

## How Does It Work (Plain Language)

Represent each DP layer as a bitset, each bit indicates whether a state is reachable.

For Subset Sum:

- Initially:  $\text{dp}[0] = 1$  (sum = 0 reachable)
- For each number  $a$ :
  - Shift left by  $a \rightarrow$  new reachable sums
  - Combine:  $\text{dp} |= \text{dp} \ll a$

Example: Adding 3 to  $\{0, 2, 5\}$  means shift left by 3  $\rightarrow \{3, 5, 8\}$ .

All in one CPU instruction!

## Example Recurrence

$$dp[s] = dp[s] \vee dp[s - a]$$

Bitset form:

$$dp = dp \vee (dp \ll a)$$

## Tiny Code (Easy Version)

Python (using int bitset)

```
def subset_sum(nums, target):
    dp = 1 # bit 0 = reachable sum 0
    for a in nums:
        dp |= dp << a
    return (dp >> target) & 1 # check if bit target is set

print(subset_sum([3, 2, 7], 5)) # True (3+2)
print(subset_sum([3, 2, 7], 6)) # False
```

C (using bitset)

```
#include <stdio.h>
#include <string.h>

#define MAXS 10000
#define W 64
unsigned long long dp[MAXS / W + 1];

void setbit(int i) { dp[i / W] |= 1ULL << (i % W); }
int getbit(int i) { return (dp[i / W] >> (i % W)) & 1; }

int main() {
    memset(dp, 0, sizeof(dp));
    setbit(0); // sum 0 reachable
    int a[] = {3, 2, 7}, n = 3;
    for (int i = 0; i < n; i++) {
        int v = a[i];
        for (int j = MAXS / W; j >= 0; j--) {
            unsigned long long shifted = dp[j] << v;
            if (j + v / W + 1 <= MAXS / W)
                dp[j + v / W + 1] |= dp[j] >> (W - (v % W));
            dp[j] |= shifted;
        }
    }
    printf("Sum 5 reachable? %d\n", getbit(5));
}
```

## Why It Matters

- Exploits hardware parallelism
- Ideal for dense Boolean DP
- Works for subset sums, range transitions, graph masks
- Achieves massive speedups with simple operations

## Step-by-Step Example

Suppose `nums = [2, 3]`, `target = 5`

Start: `dp = 1 → {0}`

After 2: `dp << 2 = {2}` `dp |= dp << 2 = {0, 2}`

After 3: `dp << 3 = {3, 5}` `dp |= dp << 3 = {0, 2, 3, 5}`

Bit 5 is set → sum 5 reachable

## A Gentle Proof (Why It Works)

Each shift corresponds to adding an element to a subset sum. Bitwise OR merges reachable sums. No overlap conflict, each bit is unique to a sum. After all shifts, all sums formed by subsets are represented.

Each shift-OR runs in  $O(S/w)$  time, where  $S = \text{target sum}$ ,  $w = \text{word size}$ .

## Try It Yourself

1. Implement bounded knapsack via repeated shift-and-OR.
2. Count distinct subset sums (popcount of dp).
3. Apply to palindrome DP:  $dp[i][j] = s[i] == s[j] \wedge dp[i+1][j-1]$ .
4. Visualize bit patterns after each step.
5. Benchmark vs normal DP on large  $S$ .

## Test Cases

| nums         | target | Result |
|--------------|--------|--------|
| [3, 2, 7]    | 5      | True   |
| [3, 2, 7]    | 6      | False  |
| [1, 2, 3, 4] | 10     | True   |

## Complexity

- Time:  $O(nS/w)$
- Space:  $O(S/w)$

Bitset DP is your Boolean supercharger, turning slow loops into blinding-fast bitwise moves.

## 468 Offline DP Queries

Offline DP Queries are a strategy to handle queries on a dynamic programming state space by reordering or batching them for efficient computation. Instead of answering queries as they arrive (online), we process them *after sorting or grouping*, enabling faster transitions or range updates.

### What Problem Are We Solving?

You may have a DP or recurrence that evolves over time, and a set of queries asking for values at specific states or intervals, like:

- “What is  $dp[x]$  after all updates?”
- “What is the min cost among indices in  $[L, R]$ ? ”
- “How many reachable states satisfy condition C? ”

Naively answering queries as they appear leads to repeated recomputation. By processing them offline, we exploit sorting, prefix accumulation, or data structure reuse.

### When It Applies

Offline DP Query methods apply when:

- Queries can be sorted (by time, index, or key)
- Transitions or states evolve monotonically
- You can batch updates and reuse results

Common cases:

- Range DP queries:  $dp[i]$  over  $[L, R]$
- Monotonic state DPs (like convex hull or segment DP)
- Mo’s algorithm on DP states
- Incremental DPs where  $dp[i]$  is finalized before querying

## How Does It Work (Plain Language)

Instead of answering as we go, we:

1. Collect all queries
2. Sort them by a relevant dimension (like time or index)
3. Process DP transitions incrementally
4. Answer queries once the needed states are available

Think of it as “moving forward once” and answering everything you pass.

By decoupling query order from input order, you avoid recomputation and exploit monotonic progression of DP.

## Example Problem

You’re asked  $q$  queries:

For each  $x_i$ , what is the minimum  $dp[j] + cost(j, x_i)$  over all  $j \leq x_i$ ?

Naively,  $O(nq)$ . Offline, sort queries by  $x_i$ , process  $j = 1 \dots n$ , and maintain current DP structure (like a segment tree or convex hull).

## Tiny Code (Easy Version)

Python (sorted queries with running DP)

```
def offline_dp(arr, queries):
    # arr defines dp transitions
    # queries = [(x, idx)]
    n = len(arr)
    dp = [0] * (n + 1)
    res = [0] * len(queries)

    queries.sort()  # sort by x
    ptr = 0
    for i in range(1, n + 1):
        dp[i] = dp[i-1] + arr[i-1]
        # process queries with x == i
        while ptr < len(queries) and queries[ptr][0] == i:
            _, idx = queries[ptr]
            res[idx] = dp[i]
            ptr += 1
```

```

    return res

arr = [3, 1, 4, 1, 5]
queries = [(3, 0), (5, 1)]
print(offline_dp(arr, queries)) # [sum of first 3, sum of first 5]

```

C (sorted queries)

```

#include <stdio.h>
#include <stdlib.h>

typedef struct { int x, idx; } Query;

int cmp(const void *a, const void *b) {
    return ((Query*)a)->x - ((Query*)b)->x;
}

int main() {
    int arr[] = {3, 1, 4, 1, 5}, n = 5;
    Query q[] = {{3,0},{5,1}};
    int dp[6] = {0}, res[2];

    qsort(q, 2, sizeof(Query), cmp);
    int ptr = 0;
    for (int i = 1; i <= n; i++) {
        dp[i] = dp[i-1] + arr[i-1];
        while (ptr < 2 && q[ptr].x == i) {
            res[q[ptr].idx] = dp[i];
            ptr++;
        }
    }

    printf("%d %d\n", res[0], res[1]); // dp[3], dp[5]
}

```

## Why It Matters

- Converts repeated query updates into one forward pass
- Enables range optimizations (segment trees, CHT, etc.)
- Reduces complexity from  $O(nq)$  to  $O(n + q \log n)$  or better
- Essential for problems mixing queries + DP updates

## Step-by-Step Example

Suppose we compute cumulative  $dp[i] = dp[i - 1] + a[i]$  and queries ask  $dp[x]$  for random  $x$ :

Naive:

- Recompute each query:  $O(qn)$

Offline:

- Sort queries by  $x$
- Single pass  $O(n + q)$

Same principle applies to complex DPs if queries depend on monotone indices.

## A Gentle Proof (Why It Works)

If DP states evolve monotonically in one dimension (index or time), then after computing  $dp[1]$  to  $dp[x]$ , the answer to all queries with bound  $x$  is final.

Sorting ensures we never recompute older states, and every query sees exactly what it needs, no more, no less.

Thus, each DP transition and query is processed once, yielding  $O(n + q)$  total complexity.

## Try It Yourself

1. Implement offline queries for prefix sums
2. Combine with Convex Hull Trick for sorted  $x_i$
3. Use segment tree for range min DP queries
4. Implement offline Subset Sum queries (by sum  $X$ )
5. Compare performance with online queries

## Test Cases

| arr         | queries | Output  |
|-------------|---------|---------|
| [3,1,4,1,5] | [3,5]   | [8,14]  |
| [2,2,2]     | [1,2,3] | [2,4,6] |

## Complexity

- Time:  $O(n + q \log q)$
- Space:  $O(n + q)$

Offline DP Queries are your time travelers, answer questions from the future by rearranging them into a single efficient sweep through the past.

## 469 DP + Segment Tree

DP + Segment Tree is a hybrid optimization pattern that combines dynamic programming with a segment tree (or Fenwick tree) to handle transitions involving range queries (min, max, sum) efficiently. It's especially useful when each DP state depends on a range of previous states, rather than a single index.

### What Problem Are We Solving?

We want to compute:

$$dp[i] = \min_{l_i \leq j \leq r_i} (dp[j] + cost[j])$$

or more generally,

$$dp[i] = \text{aggregate over range } [L(i), R(i)] \text{ of some function of } dp[j]$$

When transitions span intervals, naive iteration over each range is  $O(n^2)$ . A segment tree reduces this to  $O(n \log n)$  by supporting range queries and point updates.

### When It Applies

Use DP + Segment Tree when:

- Transitions depend on intervals or ranges
- The DP recurrence is monotonic in index order
- You need fast min, max, or sum over subsets

Typical problems:

- Range-based knapsack variants
- Sequence partitioning with range cost
- Interval scheduling DP

- Increasing subsequences with weight
- Pathfinding with segment bounds

## How Does It Work (Plain Language)

Instead of looping over all  $j$  to find the best previous state:

1. Store  $dp[j]$  values in a segment tree
2. Query for the minimum/maximum over  $[L(i), R(i)]$
3. Add transition cost and store  $dp[i]$  back

This way, every step:

- Query =  $O(\log n)$
- Update =  $O(\log n)$  Total =  $O(n \log n)$

## Example Recurrence

$$dp[i] = \min_{j < i, a_j < a_i} (dp[j]) + cost(i)$$

If  $a_i$  values can be ordered or compressed, we can query the segment tree for all  $a_j < a_i$  efficiently.

## Tiny Code (Easy Version)

Python

```
INF = 1018

class SegmentTree:
    def __init__(self, n):
        self.N = 1
        while self.N < n:
            self.N *= 2
        self.data = [INF] * (2 * self.N)
    def update(self, i, val):
        i += self.N
        self.data[i] = val
        while i > 1:
            i //= 2
            self.data[i] = min(self.data[2*i], self.data[2*i+1])
```

```

def query(self, l, r):
    l += self.N
    r += self.N
    res = INF
    while l < r:
        if l % 2:
            res = min(res, self.data[l])
            l += 1
        if r % 2:
            r -= 1
            res = min(res, self.data[r])
        l //= 2
        r //= 2
    return res

def dp_segment_tree(arr):
    n = len(arr)
    dp = [INF] * n
    seg = SegmentTree(n)
    dp[0] = arr[0]
    seg.update(0, dp[0])
    for i in range(1, n):
        best = seg.query(max(0, i - 2), i) # e.g., range [i-2, i-1]
        dp[i] = arr[i] + best
        seg.update(i, dp[i])
    return dp

arr = [3, 1, 4, 1, 5]
print(dp_segment_tree(arr))

```

C

```

#include <stdio.h>
#define INF 1000000000
#define N 100005

int seg[4*N], dp[N], arr[N];

int min(int a, int b) { return a < b ? a : b; }

void update(int idx, int val, int id, int l, int r) {
    if (l == r) { seg[id] = val; return; }

```

```

int mid = (l + r) / 2;
if (idx <= mid) update(idx, val, 2*id, l, mid);
else update(idx, val, 2*id+1, mid+1, r);
seg[id] = min(seg[2*id], seg[2*id+1]);
}

int query(int ql, int qr, int id, int l, int r) {
    if (qr < l || r < ql) return INF;
    if (ql <= l && r <= qr) return seg[id];
    int mid = (l + r) / 2;
    return min(query(ql, qr, 2*id, l, mid),
               query(ql, qr, 2*id+1, mid+1, r));
}

int main() {
    int arr[] = {3,1,4,1,5}, n = 5;
    for (int i = 0; i < 4*N; i++) seg[i] = INF;
    dp[0] = arr[0];
    update(0, dp[0], 1, 0, n-1);
    for (int i = 1; i < n; i++) {
        int best = query(i-2 >= 0 ? i-2 : 0, i-1, 1, 0, n-1);
        dp[i] = arr[i] + best;
        update(i, dp[i], 1, 0, n-1);
    }
    for (int i = 0; i < n; i++) printf("%d ", dp[i]);
}

```

## Why It Matters

- Handles range transitions efficiently
- Reduces quadratic DPs to  $O(n \log n)$
- Works for both min and max recurrences
- Combines with coordinate compression for complex ranges

## Step-by-Step Example

Let  $arr = [3, 1, 4, 1, 5]$ , and

$$dp[i] = arr[i] + \min_{j \in [i-2, i-1]} dp[j]$$

- $i = 0$ :  $dp[0] = 3$
- $i = 1$ : query [0, 0],  $dp[1] = 1 + 3 = 4$

- $i = 2$ : query  $[0, 1]$ ,  $dp[2] = 4 + 1 = 5$
- $i = 3$ : query  $[1, 2]$ ,  $dp[3] = 1 + 4 = 5$
- $i = 4$ : query  $[2, 3]$ ,  $dp[4] = 5 + 4 = 9$

### A Gentle Proof (Why It Works)

Segment trees store range minima. Each DP state only depends on previously finalized values. As you move  $i$  forward, you query and update disjoint ranges. Hence total complexity:

$$O(n \log n) \text{ (n queries + n updates)}$$

No recomputation, each transition is resolved via the tree in logarithmic time.

### Try It Yourself

1. Change recurrence to max and adjust segment tree.
2. Solve weighted LIS:  $dp[i] = w_i + \max_{a_j < a_i} dp[j]$ .
3. Combine with coordinate compression for arbitrary  $a_i$ .
4. Visualize segment tree contents over iterations.
5. Apply to interval scheduling with overlapping windows.

### Test Cases

| arr         | Range     | Output      |
|-------------|-----------|-------------|
| [3,1,4,1,5] | [i-2,i-1] | [3,4,5,5,9] |
| [2,2,2,2]   | [i-1,i-1] | [2,4,6,8]   |

### Complexity

- Time:  $O(n \log n)$
- Space:  $O(n)$

Segment Tree + DP is your range oracle, answering every interval dependency without scanning the whole past.

## 470 Divide & Conquer Knapsack

Divide & Conquer Knapsack is an optimization method that accelerates dynamic programming for large-capacity knapsack problems by recursively splitting the item set and combining results, rather than building a full  $O(nW)$  DP table. It is especially powerful when you need to reconstruct solutions or handle queries across subsets.

### What Problem Are We Solving?

The classic 0/1 Knapsack Problem is:

$$dp[i][w] = \max(dp[i - 1][w], dp[i - 1][w - w_i] + v_i)$$

where  $w_i$  is the item's weight and  $v_i$  its value.

This standard DP costs  $O(nW)$  in time and space, which becomes infeasible when  $n$  or  $W$  is large.

Divide & Conquer Knapsack tackles this by splitting items into halves and solving subproblems recursively, a strategy similar to meet-in-the-middle, but adapted to DP.

### When It Applies

Use Divide & Conquer Knapsack when:

- You have many items ( $n > 1000$ )
- Capacity  $W$  is large, but manageable via combinations
- You need partial solution reconstruction
- You want to handle batch queries (e.g., best value for each capacity range)

Common contexts:

- Large  $n$ , moderate  $W$  (split across subsets)
- Enumerating feasible states
- Offline processing of item sets
- Recursive solution generation (for decision trees or subset enumeration)

## How Does It Work (Plain Language)

Instead of building one giant DP table, we split the item list into halves:

- Solve left half → get all achievable  $(weight, value)$  pairs
- Solve right half → same
- Merge results efficiently (like convolution or sweep)

By recursively combining subproblems, you reduce total recomputation and enable parallel merging of feasible subsets.

If  $n = 2^k$ , recursion depth =  $O(\log n)$ , and each merge costs  $O(2^{n/2})$ , much faster than  $O(nW)$  when  $W$  is large.

## Example Recurrence

Let `solve(l, r)` compute all feasible pairs for items  $[l, r]$ :

```
If r - l == 1:  
    return {(0,0), (w_l, v_l)}  
Else:  
    mid = (l + r) / 2  
    left = solve(l, mid)  
    right = solve(mid, r)  
    return combine(left, right)
```

`combine` merges pairs from left and right (like merging sorted lists, keeping only Pareto-optimal pairs).

## Tiny Code (Easy Version)

Python

```
def combine(left, right, W):  
    res = []  
    for w1, v1 in left:  
        for w2, v2 in right:  
            w = w1 + w2  
            if w <= W:  
                res.append((w, v1 + v2))  
    # Keep only best value per weight (Pareto frontier)  
    res.sort()
```

```

best = []
cur = -1
for w, v in res:
    if v > cur:
        best.append((w, v))
        cur = v
return best

def solve(items, W):
    n = len(items)
    if n == 1:
        w, v = items[0]
        return [(0, 0), (w, v)] if w <= W else [(0, 0)]
    mid = n // 2
    left = solve(items[:mid], W)
    right = solve(items[mid:], W)
    return combine(left, right, W)

items = [(3, 4), (4, 5), (7, 10), (8, 11)]
W = 10
print(solve(items, W)) # [(0,0),(3,4),(4,5),(7,10),(8,11),(10,14)]

```

## Why It Matters

- Avoids full  $O(nW)$  DP when  $W$  is large
- Enables offline merging and solution reconstruction
- Useful in meet-in-the-middle optimization
- Can handle dynamic constraints by recombining subsets

## Step-by-Step Example

Items:  $(3, 4), (4, 5), (7, 10), (8, 11)$ ,  $W = 10$

Split:

- Left half:  $(3, 4), (4, 5) \rightarrow \text{feasible} = \{(0,0), (3,4), (4,5), (7,9)\}$
- Right half:  $(7, 10), (8, 11) \rightarrow \{(0,0), (7,10), (8,11), (15,21)\}$

Combine all  $(w_L + w_R, v_L + v_R)$   $10:$

- $(0,0), (3,4), (4,5), (7,9), (7,10), (8,11), (10,14)$

Pareto-optimal:

- (0,0), (3,4), (4,5), (7,10), (10,14)

Max value for  $W = 10$ : 14

### A Gentle Proof (Why It Works)

By recursively splitting:

- Each subset's combinations are enumerated in  $O(2^{n/2})$
- Merge step ensures only non-dominated states are carried forward
- Recursion covers all subsets exactly once

Thus, total cost  $O(2^{n/2})$  instead of  $O(nW)$ .

For moderate  $n$  (40), this is dramatically faster.

For large  $n$  with constraints (bounded weights), merges reduce to  $O(n \log n)$  per layer.

### Try It Yourself

1. Implement value-only knapsack (max value  $W$ )
2. Visualize Pareto frontier after each combine
3. Use recursion tree to print intermediate DP states
4. Compare against standard  $O(nW)$  DP results
5. Extend to multi-dimensional weights

### Test Cases

| Items                       | W  | Result |
|-----------------------------|----|--------|
| [(3,4),(4,5),(7,10),(8,11)] | 10 | 14     |
| [(1,1),(2,2),(3,3)]         | 3  | 3      |
| [(2,3),(3,4),(4,5)]         | 5  | 7      |

### Complexity

- Time:  $O(2^{n/2} \cdot n)$  (meet-in-the-middle)
- Space:  $O(2^{n/2})$

Divide & Conquer Knapsack is your recursive craftsman, building optimal subsets by combining halves, not filling tables.

# Section 48. Tree DP and Rerooting

## 471 Subtree Sum DP

Subtree Sum DP is one of the most fundamental patterns in tree dynamic programming. It computes the sum of values in every node's subtree using a simple post-order traversal. Once you know how to aggregate over subtrees, you can extend the same idea to handle sizes, depths, counts, or any associative property.

### What Problem Are We Solving?

Given a rooted tree where each node has a value, compute for every node the sum of values in its subtree (including itself).

For a node  $u$  with children  $v_1, v_2, \dots, v_k$ , the subtree sum is:

$$dp[u] = value[u] + \sum_{v \in \text{children}(u)} dp[v]$$

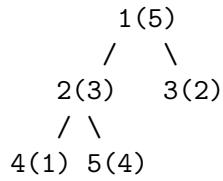
This idea generalizes to many forms of aggregation, such as counting nodes, finding subtree size, or computing subtree products.

### How Does It Work (Plain Language)

Think of each node as a small calculator. When a node finishes computing its children's sums, it adds them all up, plus its own value. This is post-order traversal, compute from leaves upward.

### Step-by-Step Example

Consider the tree:



Values:

- Node 1 → 5
- Node 2 → 3
- Node 3 → 2
- Node 4 → 1
- Node 5 → 4

We compute bottom-up:

- $dp[4] = 1$
- $dp[5] = 4$
- $dp[2] = 3 + 1 + 4 = 8$
- $dp[3] = 2$
- $dp[1] = 5 + 8 + 2 = 15$

So the subtree sums are:

| Node | Subtree Sum |
|------|-------------|
| 1    | 15          |
| 2    | 8           |
| 3    | 2           |
| 4    | 1           |
| 5    | 4           |

### Tiny Code (Easy Version)

C

```

#include <stdio.h>
#define MAXN 100

int n;
int value[MAXN];
int adj[MAXN][MAXN], deg[MAXN];
int dp[MAXN];

```

```

int dfs(int u, int parent) {
    dp[u] = value[u];
    for (int i = 0; i < deg[u]; i++) {
        int v = adj[u][i];
        if (v == parent) continue;
        dp[u] += dfs(v, u);
    }
    return dp[u];
}

int main() {
    n = 5;
    int edges[][][2] = {{1,2},{1,3},{2,4},{2,5}};
    for (int i = 0; i < 4; i++) {
        int a = edges[i][0], b = edges[i][1];
        adj[a][deg[a]++] = b;
        adj[b][deg[b]++] = a;
    }
    int vals[] = {0,5,3,2,1,4};
    for (int i = 1; i <= n; i++) value[i] = vals[i];
    dfs(1, -1);
    for (int i = 1; i <= n; i++) printf("dp[%d] = %d\n", i, dp[i]);
}

```

Python

```

from collections import defaultdict
n = 5
edges = [(1,2),(1,3),(2,4),(2,5)]
value = {1:5, 2:3, 3:2, 4:1, 5:4}

g = defaultdict(list)
for u,v in edges:
    g[u].append(v)
    g[v].append(u)

dp = {}

def dfs(u, p):
    dp[u] = value[u]
    for v in g[u]:

```

```

if v == p: continue
dfs(v, u)
dp[u] += dp[v]

dfs(1, -1)
print(dp)

```

## Why It Matters

- A core tree DP pattern: many problems reduce to aggregating over subtrees
- Forms the basis of rerooting DP, tree diameter, and centroid decomposition
- Used in computing subtree sizes, subtree XOR, sum of depths, subtree counts

Once you master subtree DP, you can generalize to:

- Max/min subtree values
- Counting paths through nodes
- Dynamic rerooting transitions

## A Gentle Proof (Why It Works)

By induction on tree depth:

- Base Case: For a leaf node  $u$ ,  $dp[u] = value[u]$ , correct by definition.
- Inductive Step: Assume all children  $v$  have correct  $dp[v]$ . Then  $dp[u] = value[u] + \sum dp[v]$  correctly accumulates all values in  $u$ 's subtree.

Since every node is visited once and every edge twice, total cost is  $O(n)$ .

## Try It Yourself

1. Modify the code to compute subtree size instead of sum
2. Track maximum value in each subtree
3. Extend to compute sum of depths per subtree
4. Add rerooting to compute subtree sum for every root
5. Use input parser to build arbitrary trees

## Test Cases

| Tree     | Values        | Subtree Sums    |
|----------|---------------|-----------------|
| 1–2–3    | {1:1,2:2,3:3} | {1:6, 2:5, 3:3} |
| 1–2, 1–3 | {1:5,2:2,3:1} | {1:8, 2:2, 3:1} |

## Complexity

- Time:  $O(n)$  (each node visited once)
- Space:  $O(n)$  recursion + adjacency

Subtree Sum DP is your first brush with tree dynamics, one traversal, full insight.

## 472 Diameter DP

Diameter DP computes the longest path in a tree, the *diameter*. Unlike shortest paths, the diameter is measured by the greatest distance between any two nodes, not necessarily passing through the root. Using dynamic programming, we can derive this in a single DFS traversal by combining the two deepest child paths at every node.

### What Problem Are We Solving?

Given a tree with  $n$  nodes (unweighted or weighted), find the diameter, i.e. the length of the longest simple path between any two nodes.

For an unweighted tree, this is measured in edges or nodes; for a weighted tree, in sum of edge weights.

We define a DP recurrence:

$$dp[u] = \text{length of longest downward path from } u$$

At each node, the diameter candidate is the sum of the two longest child paths:

$$\text{diameter} = \max(\text{diameter}, \text{top1} + \text{top2})$$

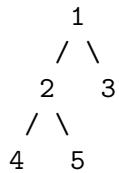
## How Does It Work (Plain Language)

Think of every node as a hub connecting paths from its children. The longest path passing through a node is formed by picking its two deepest child paths and joining them. We collect this as we perform a post-order DFS.

In the end, the global maximum across all nodes is the tree's diameter.

## Step-by-Step Example

Tree:



Each edge has weight 1.

Compute longest downward paths:

- Leaves (4, 5, 3):  $dp = 0$
- Node 2:  $dp[2] = 1 + \max(dp[4], dp[5]) = 1 + 0 = 1$
- Node 1:  $dp[1] = 1 + \max(dp[2], dp[3]) = 1 + 1 = 2$

Now compute diameter:

- At node 2: top1=0, top2=0  $\rightarrow$  local diameter=0
- At node 1: top1=1 (from 2), top2=1 (from 3)  $\rightarrow$  local diameter=2

So the tree diameter = 2 edges (path 4–2–1–3)

## Tiny Code (Easy Version)

C

```

#include <stdio.h>
#define MAXN 100

int n;
int adj[MAXN][MAXN], deg[MAXN];
int diameter = 0;

int dfs(int u, int p) {
    int top1 = 0, top2 = 0;
    for (int i = 0; i < deg[u]; i++) {
        int v = adj[u][i];
        if (v == p) continue;
        int depth = 1 + dfs(v, u);
        if (depth > top1) {
            top2 = top1;
            top1 = depth;
        } else if (depth > top2) {
            top2 = depth;
        }
    }
    if (top1 + top2 > diameter) diameter = top1 + top2;
    return top1;
}

int main() {
    n = 5;
    int edges[][2] = {{1,2},{1,3},{2,4},{2,5}};
    for (int i = 0; i < 4; i++) {
        int a = edges[i][0], b = edges[i][1];
        adj[a][deg[a]++] = b;
        adj[b][deg[b]++] = a;
    }
    dfs(1, -1);
    printf("Tree diameter: %d edges\n", diameter);
}

```

Python

```

from collections import defaultdict

g = defaultdict(list)
edges = [(1,2),(1,3),(2,4),(2,5)]

```

```

for u,v in edges:
    g[u].append(v)
    g[v].append(u)

diameter = 0

def dfs(u, p):
    global diameter
    top1 = top2 = 0
    for v in g[u]:
        if v == p: continue
        depth = 1 + dfs(v, u)
        if depth > top1:
            top2 = top1
            top1 = depth
        elif depth > top2:
            top2 = depth
    diameter = max(diameter, top1 + top2)
    return top1

dfs(1, -1)
print("Tree diameter:", diameter)

```

## Why It Matters

- Central building block for tree analysis, network radius, center finding
- Used in problems involving longest paths, tree heights, centroid decomposition
- Serves as a key step in rerooting or centroid algorithms

## A Gentle Proof (Why It Works)

Let's prove correctness by induction:

- Base Case: A leaf node has  $dp[u] = 0$ , no contribution beyond itself.
- Inductive Step: For each internal node  $u$ , if all children  $v$  correctly compute their longest downward paths  $dp[v]$ , then combining the two largest gives the longest path through  $u$ . Since every path in a tree passes through some lowest common ancestor  $u$ , our DFS finds the true maximum globally.

## Try It Yourself

1. Modify the code for weighted edges
2. Return both endpoints of the diameter path
3. Compare with two-pass BFS method (pick farthest node twice)
4. Extend to compute height of each subtree alongside
5. Visualize recursion tree with local diameters

## Test Cases

| Tree                           | Diameter |
|--------------------------------|----------|
| Line: 1–2–3–4                  | 3 edges  |
| Star: 1–{2,3,4,5}              | 2 edges  |
| Balanced binary tree (depth 2) | 4 edges  |

## Complexity

- Time:  $O(n)$
- Space:  $O(n)$  recursion stack

Diameter DP is the lens that reveals the tree's longest breath, one sweep, full span.

## 473 Independent Set DP

Independent Set DP finds the largest set of nodes in a tree such that no two chosen nodes are adjacent. This is a classic tree dynamic programming problem, showcasing the fundamental trade-off between inclusion and exclusion at each node.

### What Problem Are We Solving?

Given a tree with  $n$  nodes (each possibly having a weight), find the maximum-weight independent set, a subset of nodes such that no two connected nodes are selected.

For each node  $u$ , we maintain two DP states:

- $dp[u][0]$ : maximum value in  $u$ 's subtree when  $u$  is not chosen
- $dp[u][1]$ : maximum value in  $u$ 's subtree when  $u$  is chosen

The recurrence:

$$dp[u][0] = \sum_{v \in \text{children}(u)} \max(dp[v][0], dp[v][1])$$

$$dp[u][1] = \text{value}[u] + \sum_{v \in \text{children}(u)} dp[v][0]$$

The answer is  $\max(dp[\text{root}][0], dp[\text{root}][1])$ .

## How Does It Work (Plain Language)

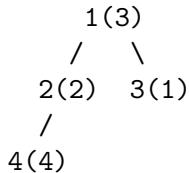
Each node decides:

- If it includes itself, it excludes its children
- If it excludes itself, it can take the best of each child

This “take-or-skip” strategy flows bottom-up from leaves to root.

## Step-by-Step Example

Tree:



Values in parentheses.

Compute DP from bottom:

- Node 4:  $dp[4][0] = 0, dp[4][1] = 4$
- Node 2:  $dp[2][0] = \max(dp[4][0], dp[4][1]) = 4$   $dp[2][1] = 2 + dp[4][0] = 2 + 0 = 2$
- Node 3:  $dp[3][0] = 0, dp[3][1] = 1$
- Node 1:  $dp[1][0] = \max(dp[2][0], dp[2][1]) + \max(dp[3][0], dp[3][1]) = 4 + 1 = 5$   $dp[1][1] = 3 + dp[2][0] + dp[3][0] = 3 + 4 + 0 = 7$

Answer:  $\max(5, 7) = 7$

Best set = {1,4}

### Tiny Code (Easy Version)

C

```
#include <stdio.h>
#define MAXN 100

int n;
int value[MAXN];
int adj[MAXN][MAXN], deg[MAXN];
int dp[MAXN][2];
int visited[MAXN];

void dfs(int u, int p) {
    dp[u][0] = 0;
    dp[u][1] = value[u];
    for (int i = 0; i < deg[u]; i++) {
        int v = adj[u][i];
        if (v == p) continue;
        dfs(v, u);
        dp[u][0] += (dp[v][0] > dp[v][1] ? dp[v][0] : dp[v][1]);
        dp[u][1] += dp[v][0];
    }
}

int main() {
    n = 4;
    int edges[][2] = {{1,2},{1,3},{2,4}};
    for (int i = 0; i < 3; i++) {
        int a = edges[i][0], b = edges[i][1];
        adj[a][deg[a]++] = b;
        adj[b][deg[b]++] = a;
    }
    int vals[] = {0,3,2,1,4};
    for (int i = 1; i <= n; i++) value[i] = vals[i];
    dfs(1, -1);
    int ans = dp[1][0] > dp[1][1] ? dp[1][0] : dp[1][1];
    printf("Max independent set sum: %d\n", ans);
}
```

Python

```

from collections import defaultdict

g = defaultdict(list)
edges = [(1,2),(1,3),(2,4)]
for u,v in edges:
    g[u].append(v)
    g[v].append(u)

value = {1:3, 2:2, 3:1, 4:4}
dp = {}

def dfs(u, p):
    include = value[u]
    exclude = 0
    for v in g[u]:
        if v == p: continue
        dfs(v, u)
        include += dp[v][0]
        exclude += max(dp[v][0], dp[v][1])
    dp[u] = (exclude, include)

dfs(1, -1)
print("Max independent set sum:", max(dp[1]))

```

## Why It Matters

- Foundation for tree-based constraint problems
- Used in network stability, resource allocation, scheduling
- Extensible to weighted graphs, forests, ranged constraints

Patterns derived from this:

- Vertex cover (complement)
- House robber on trees
- Dynamic inclusion-exclusion states

## A Gentle Proof (Why It Works)

We prove by induction:

- Base case: Leaf node  $u$  has  $dp[u][1] = value[u]$ ,  $dp[u][0] = 0$ , correct.

- Inductive step: For any node  $u$ , if all subtrees compute optimal values, including  $u$  adds its value and excludes children ( $dp[v][0]$ ), excluding  $u$  allows best child choice ( $\max(dp[v][0], dp[v][1])$ ). Thus each subtree is optimal, ensuring global optimality.

### Try It Yourself

- Extend to weighted edges (where cost is per edge)
- Modify to reconstruct chosen nodes
- Implement for forest (multiple trees)
- Compare with vertex cover DP
- Apply to House Robber III (Leetcode 337)

### Test Cases

| Tree           | Values            | Answer        |
|----------------|-------------------|---------------|
| 1–2–3          | {1:3,2:2,3:1}     | {1,3} sum=4   |
| Star 1–{2,3,4} | {1:5, others:3}   | {2,3,4} sum=9 |
| Chain 1–2–3–4  | {1:1,2:4,3:5,4:4} | {2,4} sum=8   |

### Complexity

- Time:  $O(n)$
- Space:  $O(n)$  recursion

Independent Set DP captures the tree's quiet balance, every node's choice echoes through its branches.

### 474 Vertex Cover DP

Vertex Cover DP solves a classic tree optimization problem: choose the smallest set of nodes such that every edge in the tree has at least one endpoint selected. This complements the Independent Set DP, together they form a dual pair in combinatorial optimization.

## What Problem Are We Solving?

Given a tree with  $n$  nodes, find the minimum vertex cover, i.e. a smallest subset of nodes such that every edge  $(u, v)$  has  $u$  or  $v$  in the set.

We define two states for each node  $u$ :

- $dp[u][0]$ : minimum size of vertex cover in subtree rooted at  $u$ , when  $u$  is not included
- $dp[u][1]$ : minimum size of vertex cover in subtree rooted at  $u$ , when  $u$  is included

Recurrence:

$$dp[u][0] = \sum_{v \in \text{children}(u)} dp[v][1]$$

$$dp[u][1] = 1 + \sum_{v \in \text{children}(u)} \min(dp[v][0], dp[v][1])$$

If  $u$  is not in the cover, all its children must be included to cover edges  $(u, v)$ . If  $u$  is included, each child can choose whether or not to join the cover.

Answer:  $\min(dp[\text{root}][0], dp[\text{root}][1])$

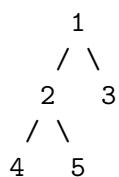
## How Does It Work (Plain Language)

Each node decides whether to take responsibility for covering edges or delegate that responsibility to its children. This is a mutual-exclusion constraint:

- If  $u$  is excluded, its children must be included.
- If  $u$  is included, each child is free to choose.

## Step-by-Step Example

Tree:



We compute bottom-up:

- Leaves 4,5:  $dp[4][0] = 0, dp[4][1] = 1$   $dp[5][0] = 0, dp[5][1] = 1$
- Node 2:  $dp[2][0] = dp[4][1] + dp[5][1] = 2$   $dp[2][1] = 1 + \min(dp[4][0], dp[4][1]) + \min(dp[5][0], dp[5][1]) = 1 + 0 + 0 = 1$
- Node 3:  $dp[3][0] = 0, dp[3][1] = 1$
- Node 1:  $dp[1][0] = dp[2][1] + dp[3][1] = 1 + 1 = 2$   $dp[1][1] = 1 + \min(dp[2][0], dp[2][1]) + \min(dp[3][0], dp[3][1]) = 1 + 1 + 0 = 2$

Result:  $\min(2, 2) = 2$

Minimum vertex cover size = 2

### Tiny Code (Easy Version)

C

```
#include <stdio.h>
#define MAXN 100

int n;
int adj[MAXN][MAXN], deg[MAXN];
int dp[MAXN][2];
int visited[MAXN];

void dfs(int u, int p) {
    dp[u][0] = 0;
    dp[u][1] = 1;
    for (int i = 0; i < deg[u]; i++) {
        int v = adj[u][i];
        if (v == p) continue;
        dfs(v, u);
        dp[u][0] += dp[v][1];
        dp[u][1] += (dp[v][0] < dp[v][1] ? dp[v][0] : dp[v][1]);
    }
}

int main() {
    n = 5;
    int edges[] [2] = {{1,2},{1,3},{2,4},{2,5}};
    for (int i = 0; i < 4; i++) {
        int a = edges[i][0], b = edges[i][1];
        adj[a][deg[a]++] = b;
    }
}
```

```

        adj[b][deg[b]++] = a;
    }
    dfs(1, -1);
    int ans = dp[1][0] < dp[1][1] ? dp[1][0] : dp[1][1];
    printf("Minimum vertex cover: %d\n", ans);
}

```

Python

```

from collections import defaultdict

g = defaultdict(list)
edges = [(1,2),(1,3),(2,4),(2,5)]
for u,v in edges:
    g[u].append(v)
    g[v].append(u)

dp = {}

def dfs(u, p):
    include = 1
    exclude = 0
    for v in g[u]:
        if v == p: continue
        dfs(v, u)
        exclude += dp[v][1]
        include += min(dp[v][0], dp[v][1])
    dp[u] = (exclude, include)

dfs(1, -1)
print("Minimum vertex cover:", min(dp[1]))

```

## Why It Matters

- Fundamental for constraint satisfaction problems on trees
- Dual to Independent Set DP (by complement)
- Used in network design, task monitoring, sensor placement

Many graph algorithms (on trees) rely on this cover-or-skip dichotomy, including:

- Dominating sets
- Guard problems

- Minimum cameras in binary tree (Leetcode 968)

### A Gentle Proof (Why It Works)

By induction:

- Base case: Leaf  $u$ :  $dp[u][0] = 0$  (if not covered, edge must be covered by parent),  $dp[u][1] = 1$  (include self).
- Inductive step: If all children have optimal covers,
  - Excluding  $u$  forces inclusion of all  $v$  (ensures edges  $(u, v)$  covered).
  - Including  $u$  allows flexible optimal choices for children. Each node's local decision yields global minimality since the tree has no cycles.

### Try It Yourself

1. Print actual cover set (backtrack from DP)
2. Extend to weighted vertex cover (replace count with sum of weights)
3. Compare with Independent Set DP, show complement sizes
4. Implement iterative version using post-order
5. Apply to Minimum Cameras in Binary Tree

### Test Cases

| Tree             | Result         | Cover |
|------------------|----------------|-------|
| 1–2–3            | 1–2–3          | {2}   |
| Star 1–{2,3,4,5} | {1}            |       |
| Chain 1–2–3–4    | {2,4} or {1,3} |       |

### Complexity

- Time:  $O(n)$
- Space:  $O(n)$

Vertex Cover DP shows how including a node protects its edges, balancing economy and completeness across the tree.

## 475 Path Counting DP

Path Counting DP is a gentle entry point into tree combinatorics, it helps you count how many distinct paths exist under certain conditions, such as from root to leaves, between pairs, or with specific constraints. It builds intuition for counting with structure rather than brute force.

### What Problem Are We Solving?

Given a tree with  $n$  nodes, we want to count paths satisfying some property. The simplest form is counting the number of root-to-leaf paths, but we can generalize to:

- Total number of paths between all pairs
- Paths with weight constraints
- Paths with certain node properties

We'll start with the fundamental version, root-to-leaf paths.

Define:

$$dp[u] = \text{number of paths starting at } u$$

For a rooted tree, each node's paths equal the sum of paths from its children. If  $u$  is a leaf, it contributes 1 path (just itself):

$$dp[u] = \begin{cases} 1, & \text{if } u \text{ is a leaf,} \\ \sum_{v \in \text{children}(u)} dp[v], & \text{otherwise.} \end{cases}$$

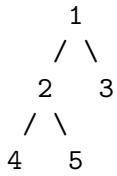
The total number of root-to-leaf paths =  $dp[root]$ .

### How Does It Work (Plain Language)

Start at the leaves, each leaf is one complete path. Each parent accumulates paths from its children: "Every path from my child forms one from me too." This propagates upward until the root holds the total count.

## Step-by-Step Example

Tree:



Compute  $dp$  from leaves upward:

- $dp[4] = 1, dp[5] = 1, dp[3] = 1$
- $dp[2] = dp[4] + dp[5] = 2$
- $dp[1] = dp[2] + dp[3] = 3$

So there are 3 root-to-leaf paths:

1. 1–2–4
2. 1–2–5
3. 1–3

## Tiny Code (Easy Version)

C

```
#include <stdio.h>
#define MAXN 100

int n;
int adj[MAXN][MAXN], deg[MAXN];
int dp[MAXN];

int dfs(int u, int p) {
    int count = 0;
    int isLeaf = 1;
    for (int i = 0; i < deg[u]; i++) {
        int v = adj[u][i];
        if (v == p) continue;
        isLeaf = 0;
        count += dfs(v, u);
    }
}
```

```

    if (isLeaf) return dp[u] = 1;
    return dp[u] = count;
}

int main() {
    n = 5;
    int edges[][][2] = {{1,2},{1,3},{2,4},{2,5}};
    for (int i = 0; i < 4; i++) {
        int a = edges[i][0], b = edges[i][1];
        adj[a][deg[a]++] = b;
        adj[b][deg[b]++] = a;
    }
    dfs(1, -1);
    printf("Root-to-leaf paths: %d\n", dp[1]);
}

```

Python

```

from collections import defaultdict

g = defaultdict(list)
edges = [(1,2),(1,3),(2,4),(2,5)]
for u,v in edges:
    g[u].append(v)
    g[v].append(u)

dp = {}

def dfs(u, p):
    is_leaf = True
    count = 0
    for v in g[u]:
        if v == p: continue
        is_leaf = False
        count += dfs(v, u)
    dp[u] = 1 if is_leaf else count
    return dp[u]

dfs(1, -1)
print("Root-to-leaf paths:", dp[1])

```

## Why It Matters

- Foundation for counting problems on trees
- Forms the basis of path-sum DP, tree DP rerooting, and combinatorial enumeration
- Essential in probabilistic models and decision trees
- Useful for probability propagation and branching process simulation

## A Gentle Proof (Why It Works)

We can prove by induction:

- Base case: Leaf node  $u$  has exactly one path, itself. So  $dp[u] = 1$ .
- Inductive step: Assume all children  $v$  compute correct counts  $dp[v]$ . Then  $dp[u] = \sum dp[v]$  counts all distinct root-to-leaf paths passing through  $u$ .

Since every path is uniquely identified by its first branching decision, we never double-count.

## Try It Yourself

1. Modify to count all simple paths (pairs  $(u, v)$ ).
2. Add edge weights and count paths with total sum  $\leq K$ .
3. Track and print all root-to-leaf paths using recursion stack.
4. Extend to directed acyclic graphs (DAGs).
5. Combine with rerooting to count paths through each node.

## Test Cases

| Tree                   | Paths   |
|------------------------|---------|
| 1–2–3                  | 1 path  |
| Star ( $1-\{2,3,4\}$ ) | 3 paths |
| Binary tree depth 2    | 3 paths |

## Complexity

- Time:  $O(n)$  (each node visited once)
- Space:  $O(n)$  recursion

Path Counting DP shows how structure transforms into number, one traversal, all paths accounted.

## 476 DP on Rooted Tree

DP on Rooted Tree is the most general pattern of tree dynamic programming, it teaches you to reason about *states* on hierarchical structures. Every subtree contributes partial answers, and a parent combines them. This is the building block for almost every tree-based DP: sums, counts, distances, constraints, and beyond.

### What Problem Are We Solving?

We want to compute a property for each node based on its subtree, things like:

- Subtree sum
- Subtree size
- Maximum depth
- Path counts
- Modular products
- Combinatorial counts

Given a rooted tree, we define a DP function that recursively collects results from each child and aggregates them.

Generic form:

$$dp[u] = f(u, dp[v] : v \in \text{children}(u))$$

You define:

- Base case (usually for leaves)
- Transition function (combine children's results)
- Merge operation (sum, max, min, multiply, etc.)

### How Does It Work (Plain Language)

You think from the leaves upward. Each node:

1. Collects results from its children.
2. Applies a combining function.
3. Stores a final value.

This bottom-up reasoning mirrors *post-order traversal*, solve children first, then parent.

The power is that  $f$  can represent *any operation*: sum, min, max, or even bitmask merge.

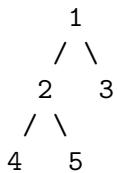
## Step-by-Step Example

Let's compute subtree size for every node (number of nodes in its subtree):

Recurrence:

$$dp[u] = 1 + \sum_{v \in children(u)} dp[v]$$

Tree:



Compute bottom-up:

- $dp[4] = 1, dp[5] = 1$
- $dp[2] = 1 + 1 + 1 = 3$
- $dp[3] = 1$
- $dp[1] = 1 + 3 + 1 = 5$

So:

- $dp[1] = 5$
- $dp[2] = 3$
- $dp[3] = 1$
- $dp[4] = 1$
- $dp[5] = 1$

## Tiny Code (Easy Version)

C

```
#include <stdio.h>
#define MAXN 100

int n;
int adj[MAXN][MAXN], deg[MAXN];
int dp[MAXN];
```

```

int dfs(int u, int p) {
    dp[u] = 1; // count itself
    for (int i = 0; i < deg[u]; i++) {
        int v = adj[u][i];
        if (v == p) continue;
        dp[u] += dfs(v, u);
    }
    return dp[u];
}

int main() {
    n = 5;
    int edges[][][2] = {{1,2},{1,3},{2,4},{2,5}};
    for (int i = 0; i < 4; i++) {
        int a = edges[i][0], b = edges[i][1];
        adj[a][deg[a]++] = b;
        adj[b][deg[b]++] = a;
    }
    dfs(1, -1);
    for (int i = 1; i <= n; i++)
        printf("dp[%d] = %d\n", i, dp[i]);
}

```

Python

```

from collections import defaultdict

g = defaultdict(list)
edges = [(1,2),(1,3),(2,4),(2,5)]
for u,v in edges:
    g[u].append(v)
    g[v].append(u)

dp = {}

def dfs(u, p):
    dp[u] = 1
    for v in g[u]:
        if v == p: continue
        dp[u] += dfs(v, u)
    return dp[u]

```

```
dfs(1, -1)
print(dp)
```

## Why It Matters

- Core template for any tree DP problem
- Powers algorithms like subtree sum, depth counting, modular product aggregation, path count, and rerooting
- Foundation for advanced rerooting DP, where answers depend on parent and sibling states

Once you master this pattern, you can:

- Change the recurrence → change the problem
- Add constraints → introduce multiple DP states
- Extend to graphs with DAG structure

## A Gentle Proof (Why It Works)

Induction on tree height:

- Base case: Leaf  $u \rightarrow dp[u]$  initialized to base (e.g. 1 or  $\text{value}[u]$ ).
- Inductive step: Suppose all children  $v$  compute correct  $dp[v]$ . Then  $f(u, dp[v])$  aggregates subtree results correctly.

Since trees are acyclic, post-order guarantees children are processed first, correctness follows.

## Try It Yourself

1. Change recurrence to compute sum of subtree values.
2. Compute maximum subtree depth.
3. Track count of leaves in each subtree.
4. Extend to two-state DP, e.g. include/exclude logic.
5. Combine with rerooting to compute value for every root.

| Tree | Subtree Sizes |
|------|---------------|
|------|---------------|

## Test Cases

| Tree             | Subtree Sizes     |
|------------------|-------------------|
| 1–2–3            | {1:3, 2:2, 3:1}   |
| Star 1–{2,3,4,5} | {1:5, others:1}   |
| Chain 1–2–3–4    | {1:4,2:3,3:2,4:1} |

## Complexity

- Time:  $O(n)$
- Space:  $O(n)$  recursion

DP on Rooted Tree is your *canvas*, define  $f$ , and paint the structure's logic across its branches.

## 477 Rerooting Technique

The Rerooting Technique is a powerful pattern in tree dynamic programming that allows you to compute results for every node as root, in linear time. Instead of recalculating from scratch for each root, we “reroot” efficiently by reusing already computed subtree results.

### What Problem Are We Solving?

Suppose we've computed a property (like subtree sum, subtree size, distance sum) rooted at a fixed node, say node 1. Now we want to compute the same property for all nodes as root, for example:

- Sum of distances from each node to all others
- Size or cost of each subtree when rooted differently
- Count of paths or contributions that depend on root position

We define:

$$dp[u] = f(dp[v]_{v \in \text{children}(u)})$$

But we also want  $res[u]$ , the answer when the tree is rooted at  $u$ .

By rerooting, we can transfer results along edges, moving the root and updating only local contributions.

### How Does It Work (Plain Language)

We first perform a post-order DFS to compute  $dp[u]$  for every node as if the root were fixed (e.g. node 1).

Then, a second pre-order DFS “pushes” results outward — when moving root from parent  $u$  to child  $v$ , we adjust contributions:

- Remove child’s part from parent
- Add parent’s part to child

Each rerooting step updates  $res[v]$  from  $res[u]$  in constant or small time.

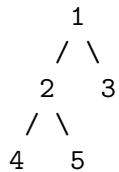
This two-pass structure is the hallmark of rerooting DP:

1. Downward pass: gather subtree results
2. Upward pass: propagate parent contributions

### Step-by-Step Example: Sum of Distances

Goal: for each node, compute sum of distances to all other nodes.

Tree:



Step 1: Post-order (subtree sums + sizes)

For each node:

- $subtree\_size[u]$  = number of nodes in subtree of  $u$
- $dp[u]$  = sum of distances from  $u$  to nodes in its subtree

Recurrence:

$$\text{subtree\_size}[u] = 1 + \sum \text{subtree\_size}[v]$$

$$dp[u] = \sum (dp[v] + \text{subtree\_size}[v])$$

At root (1),  $dp[1] = 8$  (sum of distances from 1 to all nodes).

Step 2: Pre-order (rerooting)

When rerooting from  $u$  to  $v$ :

- Moving root away from  $v$  adds  $(n - \text{subtree\_size}[v])$
- Moving root toward  $v$  subtracts  $\text{subtree\_size}[v]$

So:

$$dp[v] = dp[u] + (n - 2 \times \text{subtree\_size}[v])$$

Now every node has its distance sum in  $O(n)$ .

### Tiny Code (Sum of Distances)

C

```
#include <stdio.h>
#define MAXN 100

int n;
int adj[MAXN][MAXN], deg[MAXN];
int dp[MAXN], subtree[MAXN], res[MAXN];

void dfs1(int u, int p) {
    subtree[u] = 1;
    dp[u] = 0;
    for (int i = 0; i < deg[u]; i++) {
        int v = adj[u][i];
        if (v == p) continue;
        dfs1(v, u);
        subtree[u] += subtree[v];
        dp[u] += dp[v] + subtree[v];
    }
}
```

```

void dfs2(int u, int p) {
    res[u] = dp[u];
    for (int i = 0; i < deg[u]; i++) {
        int v = adj[u][i];
        if (v == p) continue;
        int pu = dp[u], pv = dp[v];
        int su = subtree[u], sv = subtree[v];

        // move root u -> v
        dp[u] -= dp[v] + subtree[v];
        subtree[u] -= subtree[v];
        dp[v] += dp[u] + subtree[u];
        subtree[v] += subtree[u];

        dfs2(v, u);

        // restore
        dp[v] = pv;
        dp[u] = pu;
        subtree[v] = sv;
        subtree[u] = su;
    }
}

int main() {
    n = 5;
    int edges[][][2] = {{1,2},{1,3},{2,4},{2,5}};
    for (int i = 0; i < 4; i++) {
        int a = edges[i][0], b = edges[i][1];
        adj[a][deg[a]++] = b;
        adj[b][deg[b]++] = a;
    }
    dfs1(1, -1);
    dfs2(1, -1);
    for (int i = 1; i <= n; i++)
        printf("res[%d] = %d\n", i, res[i]);
}

```

Python

```

from collections import defaultdict
g = defaultdict(list)
edges = [(1,2),(1,3),(2,4),(2,5)]
for u,v in edges:
    g[u].append(v)
    g[v].append(u)

n = 5
dp = {i:0 for i in range(1,n+1)}
sub = {i:1 for i in range(1,n+1)}
res = {}

def dfs1(u,p):
    sub[u]=1
    dp[u]=0
    for v in g[u]:
        if v==p: continue
        dfs1(v,u)
        sub[u]+=sub[v]
        dp[u]+=dp[v]+sub[v]

def dfs2(u,p):
    res[u]=dp[u]
    for v in g[u]:
        if v==p: continue
        pu,pv=dp[u],dp[v]
        su,sv=sub[u],sub[v]

        dp[u]-=dp[v]+sub[v]
        sub[u]-=sub[v]
        dp[v]+=dp[u]+sub[u]
        sub[v]+=sub[u]

        dfs2(v,u)

        dp[u],dp[v]=pu,pv
        sub[u],sub[v]=su,sv

dfs1(1,-1)
dfs2(1,-1)
print(res)

```

## Why It Matters

- Compute answers for all nodes in  $O(n)$
- Essential in distance sums, rerooted subtree queries, centroid-based algorithms
- Core pattern in Tree Rerooting DP problems on AtCoder, Codeforces, Leetcode

Rerooting transforms one-root logic into every-root knowledge.

## A Gentle Proof (Why It Works)

1. First pass ensures each node knows its subtree contribution.
2. Second pass applies a constant-time update to shift the root:
  - Remove child's contribution
  - Add parent's complement

Since each edge is traversed twice, total cost is linear.

## Try It Yourself

1. Count number of nodes in subtree for every possible root.
2. Compute sum of depths for every root.
3. Modify recurrence for product of subtree values.
4. Apply to tree balancing: minimize total distance.
5. Extend to weighted trees.

## Test Cases

| Tree  | Root | Distance Sum |
|-------|------|--------------|
| 1–2–3 | 1    | 3            |
| 1–2–3 | 2    | 2            |
| 1–2–3 | 3    | 3            |

## Complexity

- Time:  $O(n)$
- Space:  $O(n)$

Rerooting DP is your algorithmic kaleidoscope, spin the root, and watch all perspectives appear.

## 478 Distance Sum Rerooting

Distance Sum Rerooting is a classic and elegant application of the rerooting technique. It computes, for every node, the sum of distances to all other nodes in a tree, in just  $O(n)$  time. It's one of the cleanest examples showing how rerooting transforms local subtree data into global insight.

### What Problem Are We Solving?

Given a tree with  $n$  nodes, for every node  $u$ , compute:

$$res[u] = \sum_{v=1}^n \text{dist}(u, v)$$

A naïve approach (running BFS from every node) takes  $O(n^2)$ . We'll do it in two DFS passes using rerooting DP.

### How Does It Work (Plain Language)

1. First pass (post-order): Root the tree at an arbitrary node (say, 1). Compute:

- $\text{subtree}[u]$ : size of subtree rooted at  $u$
- $dp[u]$ : sum of distances from  $u$  to all nodes in its subtree

Recurrence:

$$\text{subtree}[u] = 1 + \sum_{v \in \text{children}(u)} \text{subtree}[v]$$

$$dp[u] = \sum_{v \in \text{children}(u)} (dp[v] + \text{subtree}[v])$$

2. Second pass (pre-order): Use rerooting to compute  $res[v]$  from  $res[u]$ :

$$res[v] = res[u] + (n - 2 \times \text{subtree}[v])$$

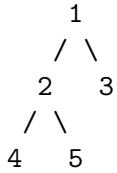
- Moving root from  $u$  to  $v$ :
  - Nodes inside  $v$ 's subtree get 1 closer
  - Nodes outside get 1 farther
- So net change =  $-\text{subtree}[v] + (n - \text{subtree}[v]) = n - 2 \times \text{subtree}[v]$

3.  $res[1] = dp[1]$  (initial result for root)

After this, each node has its sum of distances, all in linear time.

### Step-by-Step Example

Tree:



Step 1 (Post-order)

- $dp[4] = 0, dp[5] = 0, dp[3] = 0$
- $subtree[4] = 1, subtree[5] = 1, subtree[3] = 1$
- $subtree[2] = 1+1+1 = 3, dp[2] = dp[4]+dp[5]+subtree[4]+subtree[5] = 0+0+1+1 = 2$
- $subtree[1] = 1+3+1 = 5, dp[1] = dp[2]+dp[3]+subtree[2]+subtree[3] = 2+0+3+1 = 6$

So  $res[1] = 6$ .

Step 2 (Reroot)

- $res[2] = res[1] + (5 - 2 \times 3) = 6 - 1 = 5$
- $res[3] = res[1] + (5 - 2 \times 1) = 6 + 3 = 9$
- $res[4] = res[2] + (5 - 2 \times 1) = 5 + 3 = 8$
- $res[5] = res[2] + (5 - 2 \times 1) = 5 + 3 = 8$

Final:

- $res[1] = 6, res[2] = 5, res[3] = 9, res[4] = 8, res[5] = 8$

### Tiny Code (Easy Version)

C

```

#include <stdio.h>
#define MAXN 100

int n;
int adj[MAXN][MAXN], deg[MAXN];
int subtree[MAXN], dp[MAXN], res[MAXN];

void dfs1(int u, int p) {
    subtree[u] = 1;
    dp[u] = 0;
    for (int i = 0; i < deg[u]; i++) {
        int v = adj[u][i];
        if (v == p) continue;
        dfs1(v, u);
        subtree[u] += subtree[v];
        dp[u] += dp[v] + subtree[v];
    }
}

void dfs2(int u, int p) {
    res[u] = dp[u];
    for (int i = 0; i < deg[u]; i++) {
        int v = adj[u][i];
        if (v == p) continue;
        dp[v] = dp[u] + (n - 2 * subtree[v]);
        dfs2(v, u);
    }
}

int main() {
    n = 5;
    int edges[][2] = {{1,2},{1,3},{2,4},{2,5}};
    for (int i = 0; i < 4; i++) {
        int a = edges[i][0], b = edges[i][1];
        adj[a][deg[a]++] = b;
        adj[b][deg[b]++] = a;
    }
    dfs1(1, -1);
    dfs2(1, -1);
    for (int i = 1; i <= n; i++)
        printf("Sum of distances from %d = %d\n", i, res[i]);
}

```

Python

```
from collections import defaultdict

g = defaultdict(list)
edges = [(1,2),(1,3),(2,4),(2,5)]
for u,v in edges:
    g[u].append(v)
    g[v].append(u)

n = 5
dp = {i:0 for i in range(1,n+1)}
sub = {i:1 for i in range(1,n+1)}
res = {}

def dfs1(u,p):
    sub[u]=1
    dp[u]=0
    for v in g[u]:
        if v==p: continue
        dfs1(v,u)
        sub[u]+=sub[v]
        dp[u]+=dp[v]+sub[v]

def dfs2(u,p):
    res[u]=dp[u]
    for v in g[u]:
        if v==p: continue
        dp[v]=dp[u]+(n-2*sub[v])
        dfs2(v,u)

dfs1(1,-1)
dfs2(1,-1)
print(res)
```

## Why It Matters

- Computes sum of distances for every node in linear time
- A foundational rerooting example, applies to many other metrics (sums, products, min/max)
- Extensible to weighted edges, directed trees, and centroid decomposition
- Useful in graph analysis, network latency, tree balancing, dynamic centers

## A Gentle Proof (Why It Works)

Each reroot step adjusts the sum of distances by accounting for nodes that become closer or farther:

- Nodes in the new root's subtree ( $\text{subtree}[v]$ ): distances decrease by 1
- Others ( $n - \text{subtree}[v]$ ): distances increase by 1

So:

$$\text{res}[v] = \text{res}[u] + (n - 2 \times \text{subtree}[v])$$

By induction across edges, each node gets correct total distance.

## Try It Yourself

1. Extend to weighted edges.
2. Compute average distance per node.
3. Combine with centroid finding (node minimizing  $\text{res}[u]$ ).
4. Visualize change in  $\text{res}$  as root slides.
5. Adapt for directed rooted trees.

## Test Cases

| Tree  | Node | Result |
|-------|------|--------|
| 1–2–3 | 1    | 3      |
| 1–2–3 | 2    | 2      |
| 1–2–3 | 3    | 3      |

## Complexity

- Time:  $O(n)$
- Space:  $O(n)$

Distance Sum Rerooting shows the beauty of symmetry in trees, move the root, update the world, keep it all in balance.

## 479 Tree Coloring DP

Tree Coloring DP is a versatile pattern for solving coloring and labeling problems on trees under local constraints, for example, counting how many valid colorings exist when adjacent nodes cannot share a color, or minimizing cost under adjacency restrictions. It combines state definition with child aggregation, forming one of the most common templates in competitive programming.

### What Problem Are We Solving?

Given a tree with  $n$  nodes, we want to color each node with one of  $k$  colors so that no two adjacent nodes share the same color, and count the number of valid colorings.

Formally, find the number of ways to assign  $\text{color}[u] \in 1, 2, \dots, k$  such that for every edge  $(u, v)$ ,  $\text{color}[u] \neq \text{color}[v]$ .

We can also generalize:

- Weighted versions (cost per color)
- Restricted versions (pre-colored nodes)
- Modular counting ( $\text{mod } 10^9 + 7$ )

Here, we'll solve the basic unweighted counting version.

### How Does It Work (Plain Language)

We do a rooted DP where each node decides its color and multiplies the valid combinations from its children.

If a node  $u$  is colored  $c$ , then each child  $v$  can take any color except  $c$ .

So for each node:

$$dp[u][c] = \prod_{v \in \text{children}(u)} \sum_{c'=1}^k dp[v][c'] \quad c' \neq c$$

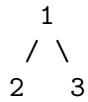
Finally, the total count is:

$$\text{Answer} = \sum_{c=1}^k dp[\text{root}][c]$$

Because the tree is acyclic, we can safely combine subtrees without overcounting.

## Step-by-Step Example

Tree:



$k = 3$  colors (1,2,3)

Start from leaves:

- For leaf node  $v$ ,  $dp[v][c] = 1$  for all  $c \in 1, 2, 3$  (any color works)

Now node 2 and 3:

- $dp[2] = dp[3] = [1, 1, 1]$

At node 1:

- $dp[1][1] = \prod_{child} \sum_{c' \neq 1} dp[child][c'] = (1 + 1) * (1 + 1) = 4$
- $dp[1][2] = (1 + 1) * (1 + 1) = 4$
- $dp[1][3] = (1 + 1) * (1 + 1) = 4$

Total =  $4 + 4 + 4 = 12$  valid colorings

Manual check: each of 3 colors for node 1  $\times$  2 choices per child  $\times$  2 children = 12

## Tiny Code (Easy Version)

C

```
#include <stdio.h>
#define MAXN 100
#define MAXK 10
#define MOD 10000000007

int n, k;
int adj[MAXN][MAXN], deg[MAXN];
long long dp[MAXN][MAXK+1];

void dfs(int u, int p) {
    for (int c = 1; c <= k; c++) dp[u][c] = 1;
    for (int i = 0; i < deg[u]; i++) {
```

```

int v = adj[u][i];
if (v == p) continue;
dfs(v, u);
for (int c = 1; c <= k; c++) {
    long long sum = 0;
    for (int c2 = 1; c2 <= k; c2++) {
        if (c2 == c) continue;
        sum = (sum + dp[v][c2]) % MOD;
    }
    dp[u][c] = (dp[u][c] * sum) % MOD;
}
}

int main() {
    n = 3; k = 3;
    int edges[][][2] = {{1,2},{1,3}};
    for (int i = 0; i < 2; i++) {
        int a = edges[i][0], b = edges[i][1];
        adj[a][deg[a]++] = b;
        adj[b][deg[b]++] = a;
    }
    dfs(1, -1);
    long long ans = 0;
    for (int c = 1; c <= k; c++) ans = (ans + dp[1][c]) % MOD;
    printf("Total colorings: %lld\n", ans);
}

```

Python

```

from collections import defaultdict

MOD = 109 + 7
n, k = 3, 3
edges = [(1,2),(1,3)]

g = defaultdict(list)
for u,v in edges:
    g[u].append(v)
    g[v].append(u)

dp = {}

```

```

def dfs(u, p):
    dp[u] = [1]*(k+1)
    for v in g[u]:
        if v == p: continue
        dfs(v, u)
    new = [0]*(k+1)
    for c in range(1, k+1):
        total = 0
        for c2 in range(1, k+1):
            if c2 == c: continue
            total = (total + dp[v][c2]) % MOD
        new[c] = (dp[u][c] * total) % MOD
    dp[u] = new

dfs(1, -1)
ans = sum(dp[1][1:]) % MOD
print("Total colorings:", ans)

```

## Why It Matters

- Solves coloring, labeling, and assignment problems on trees
- Foundation for constraint satisfaction DPs
- Extensible to weighted, modular, and partial pre-colored versions
- Appears in graph theory, combinatorics, and tree-structured probabilistic models

With small tweaks, it becomes:

- Minimum-cost coloring (replace `+` with `min`)
- Constraint coloring (prune invalid colors)
- Modular counting (for combinatorics)

## A Gentle Proof (Why It Works)

By induction on tree height:

- Base case: Leaf node  $u$ :  $dp[u][c] = 1$  (can take any color)
- Inductive step: Suppose each child  $v$  has computed correct  $dp[v][c']$ . For node  $u$  colored  $c$ , all children  $v$  must choose colors  $c' \neq c$ . Summing and multiplying ensures we count all valid combinations.

No overlap or omission occurs because trees have no cycles.

## Try It Yourself

1. Add modular constraint (e.g.  $k = 10^5$ ).
2. Extend to pre-colored nodes: fix certain  $dp[u][c] = 0/1$ .
3. Modify recurrence for weighted coloring (cost per color).
4. Optimize with prefix-suffix products for large  $k$ .
5. Apply to binary tree coloring with parity constraints.

## Test Cases

| Tree           | n | k | Result |
|----------------|---|---|--------|
| 1–2            | 2 | 2 | 2      |
| 1–2–3          | 3 | 3 | 12     |
| Star 1–{2,3,4} | 4 | 3 | 24     |

## Complexity

- Time:  $O(n \times k^2)$
- Space:  $O(n \times k)$

Tree Coloring DP is your combinatorial paintbrush, define local rules, traverse once, and color the whole forest with logic.

## 480 Binary Search on Tree DP

Binary Search on Tree DP is a hybrid strategy combining tree dynamic programming with binary search over an answer space. It's especially useful when the feasibility of a condition is monotonic, for example, when asking "is there a subtree/path satisfying constraint X under threshold T?" and the answer changes from false → true as T increases.

### What Problem Are We Solving?

Given a tree with weights or values on nodes or edges, we want to find a minimum (or maximum) threshold  $T$  such that a property holds, e.g.:

- Longest path with all edge weights  $T$
- Smallest  $T$  such that there exists a subtree of sum  $S$
- Minimal limit where a valid DP state becomes achievable

We binary search over  $T$ , and for each guess, we run a DP on the tree to check if the condition is satisfied.

### How Does It Work (Plain Language)

1. Identify a monotonic property, one that, once true, stays true (or once false, stays false).
2. Define a  $\text{check}(T)$  function using Tree DP that returns whether the property holds.
3. Apply binary search over  $T$  to find the smallest (or largest) value satisfying the condition.

### Example: Longest Path Under Limit

We're given a weighted tree with edge weights  $w(u, v)$ . Find the maximum path length such that all edges  $\in T$ . We want the minimum  $T$  for which path length  $\geq L$ .

Steps:

1. Binary search over  $T$
2. For each  $T$ , build a subgraph of edges  $\in T$
3. Run DP on tree (e.g. diameter DP) to check if a path of length  $\geq L$  exists

### DP Design

We use a DFS-based DP that computes, for each node:

$$dp[u] = \text{length of longest downward path under } T$$

and combine two best child paths to check if the diameter  $\geq L$ .

### Tiny Code (Feasibility Check)

C

```
#include <stdio.h>
#include <string.h>
#define MAXN 100
#define INF 1000000000

int n, L;
int adj[MAXN][MAXN], w[MAXN][MAXN], deg[MAXN];
int best;
```

```

int dfs(int u, int p, int T) {
    int top1 = 0, top2 = 0;
    for (int i = 0; i < deg[u]; i++) {
        int v = adj[u][i];
        if (v == p || w[u][v] > T) continue;
        int len = dfs(v, u, T) + 1;
        if (len > top1) { top2 = top1; top1 = len; }
        else if (len > top2) top2 = len;
    }
    if (top1 + top2 >= L) best = 1;
    return top1;
}

int check(int T) {
    best = 0;
    dfs(1, -1, T);
    return best;
}

int main() {
    n = 4; L = 3;
    int edges[3][3] = {{1,2,3},{2,3,5},{3,4,7}};
    for (int i = 0; i < 3; i++) {
        int a=edges[i][0], b=edges[i][1], c=edges[i][2];
        adj[a][deg[a]] = b; w[a][b] = c; deg[a]++;
        adj[b][deg[b]] = a; w[b][a] = c; deg[b]++;
    }
    int lo = 0, hi = 10, ans = -1;
    while (lo <= hi) {
        int mid = (lo + hi)/2;
        if (check(mid)) { ans = mid; hi = mid - 1; }
        else lo = mid + 1;
    }
    printf("Minimum T: %d\n", ans);
}

```

Python

```

from collections import defaultdict

g = defaultdict(list)

```

```

edges = [(1,2,3),(2,3,5),(3,4,7)]
n, L = 4, 3
for u,v,w in edges:
    g[u].append((v,w))
    g[v].append((u,w))

def dfs(u, p, T):
    top1 = top2 = 0
    global ok
    for v,w in g[u]:
        if v == p or w > T: continue
        length = dfs(v, u, T) + 1
        if length > top1:
            top2 = top1
            top1 = length
        elif length > top2:
            top2 = length
    if top1 + top2 >= L:
        ok = True
    return top1

def check(T):
    global ok
    ok = False
    dfs(1, -1, T)
    return ok

lo, hi = 0, 10
ans = -1
while lo <= hi:
    mid = (lo + hi)//2
    if check(mid):
        ans = mid
        hi = mid - 1
    else:
        lo = mid + 1
print("Minimum T:", ans)

```

## Why It Matters

- Many threshold optimization problems rely on binary search + DP

- Ideal when cost / limit interacts with tree-based structure
- Useful in network design, path constraints, tree queries, game theory

Examples:

- Smallest edge weight for connectivity
- Minimal node cost for subtree property
- Path feasibility under resource constraint

## A Gentle Proof (Why It Works)

If the property is monotonic, binary search guarantees correctness:

- If a condition holds at  $T$ , it holds at all  $T' > T$
- So, we can search for the smallest satisfying  $T$

Tree DP correctly checks feasibility because it enumerates all root-to-leaf and child-to-child paths under threshold  $T$ .

## Try It Yourself

1. Modify to maximize value (reverse monotonicity).
2. Replace edge constraint with node value  $T$ .
3. Use DP to count paths, not just check existence.
4. Apply to maximum subtree sum under bound.
5. Extend to k-colored constraints (binary search over cost).

## Test Cases

| n | L | Edges                 | Output |
|---|---|-----------------------|--------|
| 4 | 3 | (1-2:3, 2-3:5, 3-4:7) | 5      |
| 3 | 2 | (1-2:1, 2-3:2)        | 2      |

## Complexity

- DP per check:  $O(n)$
- Binary search:  $\log(\text{range})$
- Total:  $O(n \log C)$  where  $C$  is max edge weight

Binary Search on Tree DP bridges feasibility logic and optimization, use it whenever monotonic thresholds and tree states meet.

# Section 49. DP Reconstruction and Traceback

## 481 Reconstruct LCS

Reconstructing the Longest Common Subsequence (LCS) means not just computing its length, but tracing back the actual sequence that two strings share in order. This step turns abstract DP tables into tangible answers, a common need in bioinformatics, text diffing, and alignment tasks.

### What Problem Are We Solving?

Given two sequences  $A$  (length  $n$ ) and  $B$  (length  $m$ ), find the longest subsequence common to both (not necessarily contiguous).

We first build a DP table for LCS length:

$$dp[i][j] = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0, \\ dp[i - 1][j - 1] + 1, & \text{if } A[i - 1] = B[j - 1], \\ \max(dp[i - 1][j], dp[i][j - 1]), & \text{otherwise.} \end{cases}$$

Then we trace back from  $dp[n][m]$  to reconstruct the sequence.

### How Does It Work (Plain Language)

1. Compute the LCS length table using standard DP.
2. Start from the bottom-right corner ( $dp[n][m]$ ).
3. Trace back:
  - If  $A[i - 1] == B[j - 1]$ : add that character and move diagonally  $(i - 1, j - 1)$
  - Else move to the direction with larger dp value
4. Reverse the collected sequence.

## Example

Let  $A = "ABCBDAB"$ ,  $B = "BDCABA"$

DP length table leads to result “BCBA”.

## Tiny Code

C

```
#include <stdio.h>
#include <string.h>

#define MAX 100

int dp[MAX][MAX];
char A[MAX], B[MAX];
char lcs[MAX];

int main() {
    scanf("%s %s", A, B);
    int n = strlen(A), m = strlen(B);
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            if (A[i-1] == B[j-1])
                dp[i][j] = dp[i-1][j-1] + 1;
            else
                dp[i][j] = dp[i-1][j] > dp[i][j-1] ? dp[i-1][j] : dp[i][j-1];

    // Reconstruct
    int i = n, j = m, k = dp[n][m];
    lcs[k] = '\0';
    while (i > 0 && j > 0) {
        if (A[i-1] == B[j-1]) {
            lcs[--k] = A[i-1];
            i--;
            j--;
        } else if (dp[i-1][j] >= dp[i][j-1])
            i--;
        else
            j--;
    }
}
```

```
    printf("LCS: %s\n", lcs);
}
```

Python

```
def reconstruct_lcs(A, B):
    n, m = len(A), len(B)
    dp = [[0]*(m+1) for _ in range(n+1)]
    for i in range(1, n+1):
        for j in range(1, m+1):
            if A[i-1] == B[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])

    i, j = n, m
    res = []
    while i > 0 and j > 0:
        if A[i-1] == B[j-1]:
            res.append(A[i-1])
            i -= 1
            j -= 1
        elif dp[i-1][j] >= dp[i][j-1]:
            i -= 1
        else:
            j -= 1
    return ''.join(reversed(res))

A = input("A: ")
B = input("B: ")
print("LCS:", reconstruct_lcs(A, B))
```

## Why It Matters

- Core of diff, merge, and DNA alignment tools
- Demonstrates how DP solutions can reconstruct actual solutions, not just counts
- Foundation for traceback techniques in many DP problems

## A Gentle Proof (Why It Works)

At each step:

- If  $A[i - 1] = B[j - 1]$ , the character must belong to the LCS, so we include it and move diagonally.
- Otherwise, the longer LCS lies in the direction of the greater dp value, hence we follow that path. By starting from  $dp[n][m]$  and moving backward, we guarantee each included character is part of at least one optimal solution.

Since we collect in reverse order, reversing yields the correct sequence.

### Try It Yourself

1. Trace the LCS of “ABCBDAB” and “BDCABA” by hand.
2. Modify to find one of all possible LCSs (handle ties).
3. Extend for case-insensitive comparison.
4. Adapt code to return indices of matching characters.
5. Visualize path arrows in DP table.

### Test Cases

| A         | B         | LCS    |
|-----------|-----------|--------|
| “ABCBDAB” | “BDCABA”  | “BCBA” |
| “AGGTAB”  | “GXTXAYB” | “GTAB” |
| “AXYT”    | “AYZX”    | “AY”   |

### Complexity

- Time:  $O(nm)$
- Space:  $O(nm)$  (can optimize to  $O(\min(n, m))$  for length only)

Reconstruct LCS is your first step from number tables to actual solutions, bridging reasoning and reality.

### 482 Reconstruct LIS

Reconstructing the Longest Increasing Subsequence (LIS) means finding not just the length of the longest increasing sequence, but the actual subsequence. This is a classic step beyond computing DP values, it’s about tracing *how* we got there.

## What Problem Are We Solving?

Given a sequence of numbers  $A = [a_1, a_2, \dots, a_n]$ , we want to find the longest strictly increasing subsequence. The DP version computes LIS length in  $O(n^2)$ , but here we focus on reconstruction.

We define:

$$dp[i] = \text{length of LIS ending at } i$$

and a parent array to track predecessors:

$$\text{parent}[i] = \text{index of previous element in LIS ending at } i$$

Finally, we backtrack from the index of the maximum  $dp[i]$  to recover the sequence.

## How Does It Work (Plain Language)

1. Compute  $dp[i]$ : longest LIS ending at  $A[i]$ .
2. Track  $\text{parent}[i]$ : where this sequence came from.
3. Find max length index, call it **best**.
4. Backtrack using **parent** array.
5. Reverse the reconstructed list.

## Example

For  $A = [10, 22, 9, 33, 21, 50, 41, 60]$

We get:

- $dp = [1, 2, 1, 3, 2, 4, 4, 5]$
- LIS length = 5
- Sequence =  $[10, 22, 33, 50, 60]$

## Tiny Code

C

```

#include <stdio.h>

int main() {
    int A[] = {10, 22, 9, 33, 21, 50, 41, 60};
    int n = sizeof(A) / sizeof(A[0]);
    int dp[n], parent[n];

    for (int i = 0; i < n; i++) {
        dp[i] = 1;
        parent[i] = -1;
        for (int j = 0; j < i; j++) {
            if (A[j] < A[i] && dp[j] + 1 > dp[i]) {
                dp[i] = dp[j] + 1;
                parent[i] = j;
            }
        }
    }

    // Find index of LIS
    int best = 0;
    for (int i = 1; i < n; i++)
        if (dp[i] > dp[best]) best = i;

    // Reconstruct LIS
    int lis[100], len = 0;
    for (int i = best; i != -1; i = parent[i])
        lis[len++] = A[i];

    printf("LIS: ");
    for (int i = len - 1; i >= 0; i--)
        printf("%d ", lis[i]);
    printf("\n");
}

```

Python

```

def reconstruct_lis(A):
    n = len(A)
    dp = [1]*n
    parent = [-1]*n

    for i in range(n):

```

```

for j in range(i):
    if A[j] < A[i] and dp[j] + 1 > dp[i]:
        dp[i] = dp[j] + 1
        parent[i] = j

best = max(range(n), key=lambda i: dp[i])

res = []
while best != -1:
    res.append(A[best])
    best = parent[best]
return res[::-1]

A = [10, 22, 9, 33, 21, 50, 41, 60]
print("LIS:", reconstruct_lis(A))

```

## Why It Matters

- Transforms abstract DP into real sequence output
- Useful in scheduling, stock analysis, subsequence pattern recognition
- Teaches traceback technique with parent tracking, reused across many problems

## A Gentle Proof (Why It Works)

By definition,  $dp[i]$  records the LIS length ending at  $i$ . Whenever we update  $dp[i] = dp[j] + 1$ , we've extended the best LIS ending at  $j$ . Recording  $\text{parent}[i] = j$  lets us reconstruct that path.

The element with the maximum  $dp[i]$  must end one LIS, and by backtracking through parents, we trace exactly one valid increasing subsequence achieving the max length.

## Try It Yourself

1. Trace LIS reconstruction for  $[3, 10, 2, 1, 20]$
2. Modify to return all LIS sequences (handle equal-length ties).
3. Adapt code for non-decreasing LIS.
4. Combine with binary search LIS for  $O(n \log n)$  + parent tracking.
5. Visualize  $\text{parent}$  links as arrows between indices.

## Test Cases

| Input                           | LIS                  | Length |
|---------------------------------|----------------------|--------|
| [10, 22, 9, 33, 21, 50, 41, 60] | [10, 22, 33, 50, 60] | 5      |
| [3, 10, 2, 1, 20]               | [3, 10, 20]          | 3      |
| [50, 3, 10, 7, 40, 80]          | [3, 7, 40, 80]       | 4      |

## Complexity

- Time:  $O(n^2)$
- Space:  $O(n)$

Reconstruct LIS is a gentle bridge from computing a number to seeing the story behind it, each element tracing its lineage through the DP table.

## 483 Reconstruct Knapsack

Reconstructing the Knapsack solution means identifying which items form the optimal value, not just knowing the maximum value. This is the difference between understanding *what's possible* and *what to choose*.

### What Problem Are We Solving?

Given:

- $n$  items with values  $v[i]$  and weights  $w[i]$
- Capacity  $W$

We want:

- Maximize total value without exceeding  $W$
- Recover the chosen items

The 0/1 knapsack DP table is defined as:

$$dp[i][w] = \begin{cases} 0, & \text{if } i = 0 \text{ or } w = 0, \\ dp[i-1][w], & \text{if } w_i > w, \\ \max(dp[i-1][w], dp[i-1][w - w_i] + v_i), & \text{otherwise.} \end{cases}$$

To reconstruct, we backtrack from  $dp[n][W]$ :

- If  $dp[i][w] \neq dp[i - 1][w]$ , then item  $i$  was included
- Subtract its weight, move to  $i - 1$

### How Does It Work (Plain Language)

1. Build the standard 0/1 knapsack DP table.
2. Start from bottom-right corner  $(n, W)$ .
3. Compare  $dp[i][w]$  vs  $dp[i - 1][w]$ :
  - If different, include item  $i$ , update  $w - = w_i$ .
4. Continue until  $i = 0$ .
5. Reverse selected items for correct order.

### Example

Let:

| Item | Value | Weight |
|------|-------|--------|
| 1    | 60    | 10     |
| 2    | 100   | 20     |
| 3    | 120   | 30     |

Capacity  $W = 50$

Optimal value = 220 Chosen items = {2, 3}

### Tiny Code

C

```
#include <stdio.h>

#define N 4
#define W 50

int main() {
    int val[] = {0, 60, 100, 120};
    int wt[] = {0, 10, 20, 30};
```

```

int dp[N][W+1];

for (int i = 0; i < N; i++)
    for (int w = 0; w <= W; w++)
        dp[i][w] = 0;

for (int i = 1; i < N; i++) {
    for (int w = 1; w <= W; w++) {
        if (wt[i] <= w) {
            int include = val[i] + dp[i-1][w-wt[i]];
            int exclude = dp[i-1][w];
            dp[i][w] = include > exclude ? include : exclude;
        } else dp[i][w] = dp[i-1][w];
    }
}

printf("Max Value: %d\n", dp[N-1][W]);
printf("Items Taken: ");

int w = W;
for (int i = N-1; i > 0; i--) {
    if (dp[i][w] != dp[i-1][w]) {
        printf("%d ", i);
        w -= wt[i];
    }
}
printf("\n");
}

```

Python

```

def reconstruct_knapsack(values, weights, W):
    n = len(values)
    dp = [[0]*(W+1) for _ in range(n+1)]

    for i in range(1, n+1):
        for w in range(W+1):
            if weights[i-1] <= w:
                dp[i][w] = max(dp[i-1][w],
                                values[i-1] + dp[i-1][w - weights[i-1]])
            else:
                dp[i][w] = dp[i-1][w]

```

```

# Reconstruction
w = W
chosen = []
for i in range(n, 0, -1):
    if dp[i][w] != dp[i-1][w]:
        chosen.append(i-1)
        w -= weights[i-1]

chosen.reverse()
return dp[n][W], chosen

values = [60, 100, 120]
weights = [10, 20, 30]
W = 50
value, items = reconstruct_knapsack(values, weights, W)
print("Max value:", value)
print("Items:", items)

```

## Why It Matters

- Turns value tables into actionable decisions
- Essential in optimization problems (resource allocation, budgeting)
- Demonstrates traceback logic from DP matrix

## A Gentle Proof (Why It Works)

Every  $dp[i][w]$  represents the best value using the first  $i$  items under capacity  $w$ . If  $dp[i][w] \neq dp[i-1][w]$ , item  $i$  was critical in improving value, so it must be included. Reducing  $w$  by its weight and moving up repeats the same logic, tracing one optimal solution.

## Try It Yourself

1. Modify for multiple optimal solutions (store parent paths).
2. Implement space-optimized DP and reconstruct with backtracking info.
3. Adapt for unbounded knapsack (reuse items).
4. Add total weight output.
5. Visualize reconstruction arrows from  $dp[n][W]$ .

## Test Cases

| Values         | Weights      | W  | Max Value | Items  |
|----------------|--------------|----|-----------|--------|
| [60, 100, 120] | [10, 20, 30] | 50 | 220       | [1, 2] |
| [10, 20, 30]   | [1, 1, 1]    | 2  | 50        | [1, 2] |
| [5, 4, 6, 3]   | [2, 3, 4, 5] | 5  | 7         | [0, 1] |

## Complexity

- Time:  $O(nW)$
- Space:  $O(nW)$  (can reduce to  $O(W)$  for value-only)

Reconstruction transforms knapsack from a math result to a real-world selection list, revealing which items make the optimum possible.

## 484 Edit Distance Alignment

Edit Distance tells us *how different* two strings are, alignment reconstruction shows *exactly where* they differ. By tracing the path of operations (insert, delete, substitute), we can visualize the full transformation.

### What Problem Are We Solving?

Given two strings  $A$  (length  $n$ ) and  $B$  (length  $m$ ), compute not only the edit distance but also the alignment that transforms  $A$  into  $B$  using the minimum number of operations.

We define:

$$dp[i][j] = \begin{cases} 0, & \text{if } i = 0 \text{ and } j = 0, \\ i, & \text{if } j = 0, \\ j, & \text{if } i = 0, \\ \min \begin{cases} dp[i - 1][j] + 1, & \text{deletion,} \\ dp[i][j - 1] + 1, & \text{insertion,} \\ dp[i - 1][j - 1] + \text{cost}(A[i - 1], B[j - 1]), & \text{replace or match.} \end{cases} & \end{cases}$$

Then, we trace back from  $dp[n][m]$  to list operations in reverse.

## How Does It Work (Plain Language)

1. Build standard Levenshtein DP table.
2. Start from  $dp[n][m]$ .
3. Move:
  - Diagonal: match or replace
  - Up: delete
  - Left: insert
4. Record operation at each move.
5. Reverse the sequence for final alignment.

## Example

Let  $A = \text{"kitten"}$ ,  $B = \text{"sitting"}$ .

Operations:

| Step              | Action    | Result           |
|-------------------|-----------|------------------|
| $k \rightarrow k$ | match     | kitten / sitting |
| $i \rightarrow i$ | match     | kitten / sitting |
| $t \rightarrow t$ | match     | kitten / sitting |
| $t \rightarrow t$ | match     | kitten / sitting |
| $e \rightarrow i$ | replace   | kitti n          |
| insert g          | insertion | kitting          |

Edit distance = 3 (replace e→i, insert g, insert n)

## Tiny Code

C

```
#include <stdio.h>
#include <string.h>

#define MAX 100

int dp[MAX][MAX];
```

```

int min3(int a, int b, int c) {
    return a < b ? (a < c ? a : c) : (b < c ? b : c);
}

int main() {
    char A[MAX], B[MAX];
    scanf("%s %s", A, B);
    int n = strlen(A), m = strlen(B);

    for (int i = 0; i <= n; i++) dp[i][0] = i;
    for (int j = 0; j <= m; j++) dp[0][j] = j;

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            int cost = (A[i-1] == B[j-1]) ? 0 : 1;
            dp[i][j] = min3(dp[i-1][j] + 1,
                            dp[i][j-1] + 1,
                            dp[i-1][j-1] + cost);
        }
    }

    printf("Edit Distance: %d\n", dp[n][m]);
    printf("Operations:\n");

    int i = n, j = m;
    while (i > 0 || j > 0) {
        if (i > 0 && j > 0 && dp[i][j] == dp[i-1][j-1] && A[i-1] == B[j-1]) {
            printf("Match %c\n", A[i-1]);
            i--; j--;
        }
        else if (i > 0 && j > 0 && dp[i][j] == dp[i-1][j-1] + 1) {
            printf("Replace %c -> %c\n", A[i-1], B[j-1]);
            i--; j--;
        }
        else if (i > 0 && dp[i][j] == dp[i-1][j] + 1) {
            printf("Delete %c\n", A[i-1]);
            i--;
        }
        else {
            printf("Insert %c\n", B[j-1]);
            j--;
        }
    }
}

```

```
    }
}
```

Python

```
def edit_distance_alignment(A, B):
    n, m = len(A), len(B)
    dp = [[0]*(m+1) for _ in range(n+1)]

    for i in range(n+1):
        dp[i][0] = i
    for j in range(m+1):
        dp[0][j] = j

    for i in range(1, n+1):
        for j in range(1, m+1):
            cost = 0 if A[i-1] == B[j-1] else 1
            dp[i][j] = min(dp[i-1][j] + 1,
                            dp[i][j-1] + 1,
                            dp[i-1][j-1] + cost)

    i, j = n, m
    ops = []
    while i > 0 or j > 0:
        if i > 0 and j > 0 and A[i-1] == B[j-1]:
            ops.append(f"Match {A[i-1]}")
            i -= 1; j -= 1
        elif i > 0 and j > 0 and dp[i][j] == dp[i-1][j-1] + 1:
            ops.append(f"Replace {A[i-1]} -> {B[j-1]}")
            i -= 1; j -= 1
        elif i > 0 and dp[i][j] == dp[i-1][j] + 1:
            ops.append(f"Delete {A[i-1]}")
            i -= 1
        else:
            ops.append(f"Insert {B[j-1]}")
            j -= 1

    return dp[n][m], list(reversed(ops))

dist, ops = edit_distance_alignment("kitten", "sitting")
print("Edit Distance:", dist)
print("Alignment:")
```

```

for op in ops:
    print(op)

```

## Why It Matters

- Powers diff, spell checkers, and DNA sequence alignment
- Essential for text transformation visualization
- Reinforces traceback logic from DP tables

## A Gentle Proof (Why It Works)

Each  $dp[i][j]$  is minimal over three possibilities:

- Insertion: adding  $B[j - 1]$
- Deletion: removing  $A[i - 1]$
- Replacement: changing  $A[i - 1]$  to  $B[j - 1]$

The optimal path through the DP grid, via moves left, up, or diagonal, exactly records the sequence of minimal edits. Reversing the trace yields the transformation.

## Try It Yourself

1. Compute alignment for “intention” → “execution”
2. Add operation count summary (insertions, deletions, replacements)
3. Visualize grid with arrows ( $\uparrow$ ,  $\leftarrow$ ,  $\nwarrow$ )
4. Modify cost: substitution = 2, insertion/deletion = 1
5. Return both alignment string and operation list

## Test Cases

| A      | B        | Distance | Alignment (Ops)                 |
|--------|----------|----------|---------------------------------|
| kitten | sitting  | 3        | Replace e→i, Insert n, Insert g |
| sunday | saturday | 3        | Insert a, Insert t, Replace n→r |
| horse  | ros      | 3        | Delete h, Replace o→r, Delete e |

## Complexity

- Time:  $O(nm)$
- Space:  $O(nm)$  (can reduce to  $O(\min(n, m))$  without reconstruction)

Edit Distance Alignment transforms a distance metric into a step-by-step story, showing exactly how one word becomes another.

## 485 Matrix Chain Parentheses

Matrix Chain Multiplication gives us the minimum number of multiplications, but reconstruction tells us how to parenthesize, the *order* of multiplication that achieves that cost. Without this step, we know the cost, but not the recipe.

### What Problem Are We Solving?

Given a sequence of matrices  $A_1, A_2, \dots, A_n$  with dimensions  $p_0 \times p_1, p_1 \times p_2, \dots, p_{n-1} \times p_n$ , we want to determine the optimal parenthesization that minimizes scalar multiplications.

The cost DP is:

$$dp[i][j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k < j} (dp[i][k] + dp[k+1][j] + p_{i-1} \cdot p_k \cdot p_j) \end{cases}$$

To reconstruct the solution, we maintain a split table  $split[i][j]$  indicating the index  $k$  where the optimal split occurs.

### How Does It Work (Plain Language)

1. Compute cost table using bottom-up DP.
2. Track split point  $k$  at each subproblem.
3. Recurse:
  - Base: if  $i == j$ , return  $A_i$
  - Otherwise: ( + solve( $i, k$ ) + solve( $k+1, j$ ) + )

This yields the exact parenthesization.

## Example

Matrix dimensions: [40, 20, 30, 10, 30]

There are 4 matrices:

- $A_1 : 40 \times 20$
- $A_2 : 20 \times 30$
- $A_3 : 30 \times 10$
- $A_4 : 10 \times 30$

Optimal order:  $((A_1(A_2A_3))A_4)$  Minimal cost: 26000

## Tiny Code

C

```
#include <stdio.h>
#include <limits.h>

#define N 5

int dp[N][N];
int split[N][N];

int min(int a, int b) { return a < b ? a : b; }

void print_paren(int i, int j) {
    if (i == j) {
        printf("A%d", i);
        return;
    }
    printf("(");
    print_paren(i, split[i][j]);
    print_paren(split[i][j] + 1, j);
    printf(")");
}

int main() {
    int p[] = {40, 20, 30, 10, 30};
    int n = 4;

    for (int i = 1; i <= n; i++) dp[i][i] = 0;
```

```

for (int len = 2; len <= n; len++) {
    for (int i = 1; i <= n - len + 1; i++) {
        int j = i + len - 1;
        dp[i][j] = INT_MAX;
        for (int k = i; k < j; k++) {
            int cost = dp[i][k] + dp[k+1][j] + p[i-1]*p[k]*p[j];
            if (cost < dp[i][j]) {
                dp[i][j] = cost;
                split[i][j] = k;
            }
        }
    }
}

printf("Minimum cost: %d\n", dp[1][n]);
printf("Optimal order: ");
print_paren(1, n);
printf("\n");
}

```

Python

```

def matrix_chain_order(p):
    n = len(p) - 1
    dp = [[0]* (n+1) for _ in range(n+1)]
    split = [[0]* (n+1) for _ in range(n+1)]

    for l in range(2, n+1):
        for i in range(1, n-l+2):
            j = i + l - 1
            dp[i][j] = float('inf')
            for k in range(i, j):
                cost = dp[i][k] + dp[k+1][j] + p[i-1]*p[k]*p[j]
                if cost < dp[i][j]:
                    dp[i][j] = cost
                    split[i][j] = k

    def build(i, j):
        if i == j: return f"A{i}"
        k = split[i][j]
        return f"({build(i, k)}{build(k+1, j)})"

    return build(1, n)
}

```

```

    return dp[1][n], build(1, n)

p = [40, 20, 30, 10, 30]
cost, order = matrix_chain_order(p)
print("Min Cost:", cost)
print("Order:", order)

```

## Why It Matters

- Converts abstract cost table into concrete plan
- Foundation of query optimization, compiler expression parsing
- Shows how split tracking yields human-readable structure

## A Gentle Proof (Why It Works)

At each subchain  $(i, j)$ , DP tries all  $k$  splits. The chosen  $k$  minimizing cost is stored in  $\text{split}[i][j]$ . By recursively applying these stored splits, we follow the same decision tree that generated the minimal cost. Thus reconstruction yields the exact sequence of multiplications.

## Try It Yourself

1. Try  $p = [10, 20, 30, 40, 30]$  and verify order.
2. Add printing of subproblem cost for each pair  $(i, j)$ .
3. Modify to return tree structure instead of string.
4. Visualize with nested parentheses tree.
5. Extend to show intermediate matrix dimensions at each step.

## Test Cases

| Dimensions       | Cost  | Parenthesization |
|------------------|-------|------------------|
| [40,20,30,10,30] | 26000 | ((A1(A2A3))A4)   |
| [10,20,30]       | 6000  | (A1A2A3)         |
| [10,20,30,40,30] | 30000 | ((A1A2)(A3A4))   |

## Complexity

- Time:  $O(n^3)$
- Space:  $O(n^2)$

Matrix Chain Parentheses turns cost minimization into concrete strategy, showing *not just how much*, but *how exactly*.

## 486 Coin Change Reconstruction

In the Coin Change problem, we usually count the minimum coins or total ways. Reconstruction, however, asks: *which exact coins make up the solution?* This bridges the gap between number answers and actual combinations.

### What Problem Are We Solving?

Given:

- A set of coin denominations  $\text{coins} = [c_1, c_2, \dots, c_n]$
- A target sum  $S$

We want to:

1. Compute the minimum number of coins needed (classic DP)
2. Reconstruct one optimal combination of coins that achieves  $S$

We define:

$$dp[x] = \begin{cases} 0, & \text{if } x = 0, \\ 1 + \min_{c \leq x}(dp[x - c]), & \text{if } x > 0. \end{cases}$$

And record which coin gave the best solution:

$$\text{choice}[x] = c \text{ that minimizes } dp[x - c]$$

## How Does It Work (Plain Language)

1. Build DP array from 0 to  $S$ .
2. For each amount  $x$ , try every coin  $c$ .
3. Keep track of:
  - The minimum coin count ( $dp[x]$ )
  - The coin used ( $choice[x]$ )
4. After filling, trace back from  $S$ : repeatedly subtract  $choice[x]$  until reaching 0.

## Example

Coin = [1, 3, 4], Target  $S = 6$

DP steps:

| x | dp[x] | choice[x] |
|---|-------|-----------|
| 0 | 0     | -         |
| 1 | 1     | 1         |
| 2 | 2     | 1         |
| 3 | 1     | 3         |
| 4 | 1     | 4         |
| 5 | 2     | 1         |
| 6 | 2     | 3         |

Optimal combination: [3, 3]

## Tiny Code

C

```
#include <stdio.h>
#include <limits.h>

#define MAX 100

int main() {
    int coins[] = {1, 3, 4};
    int n = 3;
```

```

int S = 6;
int dp[MAX], choice[MAX];

dp[0] = 0;
choice[0] = -1;

for (int i = 1; i <= S; i++) {
    dp[i] = INT_MAX;
    choice[i] = -1;
    for (int j = 0; j < n; j++) {
        int c = coins[j];
        if (c <= i && dp[i-c] + 1 < dp[i]) {
            dp[i] = dp[i-c] + 1;
            choice[i] = c;
        }
    }
}

printf("Min coins: %d\n", dp[S]);
printf("Combination: ");
int x = S;
while (x > 0) {
    printf("%d ", choice[x]);
    x -= choice[x];
}
printf("\n");
}

```

Python

```

def coin_change_reconstruct(coins, S):
    dp = [float('inf')] * (S + 1)
    choice = [-1] * (S + 1)
    dp[0] = 0

    for x in range(1, S + 1):
        for c in coins:
            if c <= x and dp[x - c] + 1 < dp[x]:
                dp[x] = dp[x - c] + 1
                choice[x] = c

    if dp[S] == float('inf'):

```

```

    return None, []

comb = []
while S > 0:
    comb.append(choice[S])
    S -= choice[S]

return dp[-1], comb

coins = [1, 3, 4]
S = 6
count, comb = coin_change_reconstruct(coins, S)
print("Min coins:", count)
print("Combination:", comb)

```

## Why It Matters

- Converts abstract DP result into practical plan
- Critical in finance, vending systems, resource allocation
- Reinforces traceback technique for linear DP problems

## A Gentle Proof (Why It Works)

By definition,  $dp[x] = 1 + dp[x - c]$  for optimal  $c$ . Thus the optimal last step for  $x$  must use coin  $choice[x] = c$ . Repeatedly subtracting this  $c$  gives a valid sequence ending at 0. Each step reduces the problem size while preserving optimality (greedy by DP).

## Try It Yourself

1. Try  $coins = [1, 3, 4]$ ,  $S = 10$
2. Modify to return all optimal combinations (if multiple)
3. Extend for limited coin counts
4. Visualize table  $(x, dp[x], choice[x])$
5. Adapt for non-canonical systems (like  $[1, 3, 5, 7]$ )

## Test Cases

| Coins      | S  | Min Coins | Combination |
|------------|----|-----------|-------------|
| [1, 3, 4]  | 6  | 2         | [3, 3]      |
| [1, 2, 5]  | 11 | 3         | [5, 5, 1]   |
| [2, 5, 10] | 7  | $\infty$  | []          |

## Complexity

- Time:  $O(S \times n)$
- Space:  $O(S)$

Coin Change Reconstruction transforms “how many” into “which ones”, building not just an answer, but a clear path to it.

## 487 Path Reconstruction DP

Path reconstruction in DP is the art of retracing your steps through a cost or distance table to find the *exact route* that led to the optimal answer. It’s not enough to know *how far*, you want to know *how you got there*.

### What Problem Are We Solving?

Given a grid (or graph) where each cell has a cost, we compute the minimum path cost from a start cell  $(0, 0)$  to a destination  $(n - 1, m - 1)$  using only right or down moves. Now, instead of just reporting the minimal cost, we’ll reconstruct the path.

We define:

$$dp[i][j] = \begin{cases} grid[0][0], & \text{if } i = 0 \text{ and } j = 0, \\ grid[i][j] + \min(dp[i - 1][j], dp[i][j - 1]), & \text{otherwise.} \end{cases}$$

We also maintain a parent table `parent[i][j]` to remember whether we came from top or left.

## How Does It Work (Plain Language)

1. Fill  $dp[i][j]$  with the minimum cost to reach each cell.
2. Track the move that led to this cost:
  - If  $dp[i][j]$  came from  $dp[i - 1][j]$ , parent = “up”
  - Else parent = “left”
3. Start from destination  $(n - 1, m - 1)$  and backtrack using parent.
4. Reverse the reconstructed list for the correct order.

## Example

Grid:

|       |   |   |
|-------|---|---|
| 1     | 3 | 1 |
| <hr/> |   |   |
| 1     | 5 | 1 |
| 4     | 2 | 1 |

Minimal path sum: 7 Path:  $(0, 0) \rightarrow (0, 1) \rightarrow (0, 2) \rightarrow (1, 2) \rightarrow (2, 2)$

## Tiny Code

C

```
#include <stdio.h>
#include <limits.h>

#define N 3
#define M 3

int grid[N][M] = {
    {1, 3, 1},
    {1, 5, 1},
    {4, 2, 1}
};

int dp[N][M];
char parent[N][M]; // 'U' = up, 'L' = left
```

```

int min(int a, int b) { return a < b ? a : b; }

int main() {
    dp[0][0] = grid[0][0];

    // First row
    for (int j = 1; j < M; j++) {
        dp[0][j] = dp[0][j-1] + grid[0][j];
        parent[0][j] = 'L';
    }
    // First column
    for (int i = 1; i < N; i++) {
        dp[i][0] = dp[i-1][0] + grid[i][0];
        parent[i][0] = 'U';
    }

    // Fill rest
    for (int i = 1; i < N; i++) {
        for (int j = 1; j < M; j++) {
            if (dp[i-1][j] < dp[i][j-1]) {
                dp[i][j] = dp[i-1][j] + grid[i][j];
                parent[i][j] = 'U';
            } else {
                dp[i][j] = dp[i][j-1] + grid[i][j];
                parent[i][j] = 'L';
            }
        }
    }
}

printf("Min path sum: %d\n", dp[N-1][M-1]);

// Backtrack
int i = N - 1, j = M - 1;
int path[100][2], len = 0;
while (!(i == 0 && j == 0)) {
    path[len][0] = i;
    path[len][1] = j;
    len++;
    if (parent[i][j] == 'U') i--;
    else j--;
}
path[len][0] = 0; path[len][1] = 0;

```

```

len++;

printf("Path: ");
for (int k = len - 1; k >= 0; k--)
    printf("(%d,%d) ", path[k][0], path[k][1]);
printf("\n");
}

```

Python

```

def min_path_sum_path(grid):
    n, m = len(grid), len(grid[0])
    dp = [[0]*m for _ in range(n)]
    parent = [['']*m for _ in range(n)]

    dp[0][0] = grid[0][0]

    for j in range(1, m):
        dp[0][j] = dp[0][j-1] + grid[0][j]
        parent[0][j] = 'L'
    for i in range(1, n):
        dp[i][0] = dp[i-1][0] + grid[i][0]
        parent[i][0] = 'U'

    for i in range(1, n):
        for j in range(1, m):
            if dp[i-1][j] < dp[i][j-1]:
                dp[i][j] = dp[i-1][j] + grid[i][j]
                parent[i][j] = 'U'
            else:
                dp[i][j] = dp[i][j-1] + grid[i][j]
                parent[i][j] = 'L'

    # Backtrack
    path = []
    i, j = n-1, m-1
    while not (i == 0 and j == 0):
        path.append((i, j))
        if parent[i][j] == 'U':
            i -= 1
        else:
            j -= 1

```

```

path.append((0, 0))
path.reverse()

return dp[-1][-1], path

grid = [[1,3,1],[1,5,1],[4,2,1]]
cost, path = min_path_sum_path(grid)
print("Min cost:", cost)
print("Path:", path)

```

## Why It Matters

- Translates numerical DP into navigable routes
- Key in pathfinding, robot navigation, route planning
- Demonstrates parent-pointer technique for 2D grids

## A Gentle Proof (Why It Works)

By construction,  $dp[i][j]$  stores the minimal cost to reach  $(i, j)$ . Since each cell depends only on top and left, storing the better source as `parent[i][j]` ensures each step back leads to a valid prefix of an optimal path. Following parents reconstructs one such optimal path.

## Try It Yourself

1. Try on a  $4 \times 4$  grid with random costs.
2. Modify to allow diagonal moves.
3. Extend for maximum path sum (change min→max).
4. Visualize path arrows ( $\uparrow, \leftarrow$ ).
5. Adapt for graph shortest path with adjacency matrix.

## Test Cases

| Grid                                   | Result | Path                                         |
|----------------------------------------|--------|----------------------------------------------|
| <code>[[1,3,1],[1,5,1],[4,2,1]]</code> | 7      | <code>[(0,0),(0,1),(0,2),(1,2),(2,2)]</code> |
| <code>[[1,2,3],[4,5,6]]</code>         | 12     | <code>[(0,0),(0,1),(0,2),(1,2)]</code>       |

## Complexity

- Time:  $O(nm)$
- Space:  $O(nm)$

Path Reconstruction DP turns shortest paths into visible journeys, showing every choice that built the optimum.

## 488 Sequence Reconstruction

Sequence Reconstruction is the process of recovering an entire sequence from partial or implicit information, typically from DP tables, prefix relations, or pairwise constraints. It is a bridge between solving a problem and interpreting its answer as a sequence.

### What Problem Are We Solving?

You often solve DP problems that count or score possible sequences, but what if you need to recover one valid sequence (or even all)? For example:

1. Given the LIS length, reconstruct one LIS.
2. Given partial orders, reconstruct a sequence that satisfies them.
3. Given prefix sums, rebuild the original array.

Here, we'll explore a general pattern: rebuild the sequence using parent or predecessor states tracked during DP.

### Example: Reconstruct Longest Increasing Subsequence

Given an array `arr`, we first compute  $dp[i] = \text{length of LIS ending at } i$ . We then track predecessors using `parent[i]` to rebuild the actual subsequence.

### Recurrence

$$dp[i] = 1 + \max_{j < i, arr[j] < arr[i]} dp[j]$$

with

$$\text{parent}[i] = \arg \max_{j < i, arr[j] < arr[i]} dp[j]$$

After computing `dp`, we find the index of  $\max(dp)$ , then backtrack using `parent`.

## How Does It Work (Plain Language)

1. Run the LIS DP as usual.
2. Whenever we update  $dp[i]$ , store which previous index gave that improvement.
3. After finishing, find the end index of the best LIS.
4. Walk backward using `parent` until  $-1$ .
5. Reverse the collected indices, that's your LIS.

## Tiny Code

Python

```
def reconstruct_lis(arr):
    n = len(arr)
    dp = [1] * n
    parent = [-1] * n

    for i in range(n):
        for j in range(i):
            if arr[j] < arr[i] and dp[j] + 1 > dp[i]:
                dp[i] = dp[j] + 1
                parent[i] = j

    length = max(dp)
    idx = dp.index(length)

    # Backtrack
    lis = []
    while idx != -1:
        lis.append(arr[idx])
        idx = parent[idx]
    lis.reverse()
    return lis

arr = [10, 9, 2, 5, 3, 7, 101, 18]
print(reconstruct_lis(arr)) # [2, 3, 7, 18]
```

## C Version

```

#include <stdio.h>

int main() {
    int arr[] = {10, 9, 2, 5, 3, 7, 101, 18};
    int n = sizeof(arr)/sizeof(arr[0]);
    int dp[n], parent[n];
    for (int i = 0; i < n; i++) {
        dp[i] = 1;
        parent[i] = -1;
    }

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (arr[j] < arr[i] && dp[j] + 1 > dp[i]) {
                dp[i] = dp[j] + 1;
                parent[i] = j;
            }
        }
    }

    // find max index
    int max_len = 0, idx = 0;
    for (int i = 0; i < n; i++) {
        if (dp[i] > max_len) {
            max_len = dp[i];
            idx = i;
        }
    }

    // reconstruct
    int lis[n], len = 0;
    while (idx != -1) {
        lis[len++] = arr[idx];
        idx = parent[idx];
    }

    printf("LIS: ");
    for (int i = len - 1; i >= 0; i--) printf("%d ", lis[i]);
    printf("\n");
}

```

## Why It Matters

- Shows how DP tables contain full structure, not just values
- Useful in bioinformatics, diff tools, edit tracing, sequence alignment
- Forms the foundation for traceback algorithms

## A Gentle Proof (Why It Works)

By induction on index  $i$ :

- Base case: first element, LIS = `[arr[i]]`
- Inductive step: each `parent[i]` points to the previous LIS endpoint giving max length  
Thus, following parent pointers from the max element recreates a valid LIS, and reversing yields forward order.

## Try It Yourself

1. Change condition to `arr[j] > arr[i]` → Longest Decreasing Subsequence.
2. Modify to track all LIS sequences.
3. Print indices instead of values.
4. Extend to two dimensions (nested envelopes).
5. Combine with binary search LIS to get  $O(n \log n)$  reconstruction.

## Test Cases

| Input                               | LIS                      |
|-------------------------------------|--------------------------|
| <code>[10,9,2,5,3,7,10,1,18]</code> | <code>[2,3,7,18]</code>  |
| <code>[3,10,2,1,20]</code>          | <code>[3,10,20]</code>   |
| <code>[50,3,10,7,40,80]</code>      | <code>[3,7,40,80]</code> |

## Complexity

- Time:  $O(n^2)$
- Space:  $O(n)$

Sequence Reconstruction turns numerical answers into narrative sequences, revealing how each element fits into the optimal story.

## 489 Multi-Choice Reconstruction

Multi-Choice Reconstruction is about retracing selections when a DP problem allows multiple choices per state, such as picking from categories, groups, or configurations. It extends simple parent tracking into multi-dimensional or multi-decision DP, reconstructing a full combination of choices that led to the optimal answer.

### What Problem Are We Solving?

Some DP problems involve choosing one option from several categories, such as:

1. Multi-choice Knapsack, each group has several items; you can pick at most one.
2. Course Scheduling, pick one time slot per subject to maximize free time.
3. Machine Assignment, choose one machine per job for minimal cost.

We need to not only compute the optimal value, but also reconstruct which choices were made across categories.

### Example: Multi-Choice Knapsack

Given  $G$  groups, each containing several items (`weight`, `value`), select one item per group such that the total weight  $W$  and value is maximized.

### State Definition

Let  $dp[g][w] = \max$  value using first  $g$  groups with total weight  $w$ . We will track which item in each group contributed to this value.

### Recurrence

$$dp[g][w] = \max_{(w_i, v_i) \in group[g]} (dp[g - 1][w - w_i] + v_i)$$

To reconstruct, we store:

$$choice[g][w] = i \text{ such that } dp[g][w] \text{ achieved by item } i$$

## How Does It Work (Plain Language)

1. For each group, for each capacity, try every item in the group.
2. Pick the one that gives the highest value.
3. Store which item index gave that best value in `choice`.
4. After filling the table, backtrack from  $(G, W)$  using `choice` to rebuild selected items.

## Tiny Code

Python

```
def multi_choice_knapsack(groups, W):  
    G = len(groups)  
    dp = [[0] * (W + 1) for _ in range(G + 1)]  
    choice = [[-1] * (W + 1) for _ in range(G + 1)]  
  
    for g in range(1, G + 1):  
        for w in range(W + 1):  
            for idx, (wt, val) in enumerate(groups[g - 1]):  
                if wt <= w and dp[g - 1][w - wt] + val > dp[g][w]:  
                    dp[g][w] = dp[g - 1][w - wt] + val  
                    choice[g][w] = idx  
  
    # Backtrack  
    w = W  
    selected = []  
    for g in range(G, 0, -1):  
        idx = choice[g][w]  
        if idx != -1:  
            wt, val = groups[g - 1][idx]  
            selected.append((g - 1, idx, wt, val))  
            w -= wt  
    selected.reverse()  
    return dp[G][W], selected  
  
groups = [  
    [(3, 5), (2, 3)],  
    [(4, 6), (1, 2), (3, 4)],  
    [(2, 4), (1, 1)]  
$$  
print(multi_choice_knapsack(groups, 7))
```

Output:

```
(13, [(0, 0, 3, 5), (1, 1, 1, 2), (2, 0, 2, 4)])
```

## Why It Matters

- Many optimization problems involve multiple nested decisions.
- Useful in resource allocation, scheduling, and multi-constraint planning.
- Reconstruction helps explain why the DP made each choice, crucial for debugging and interpretation.

## A Gentle Proof (Why It Works)

We proceed by induction on  $g$  (group count):

- Base Case:  $g = 1$ , choose the best item under capacity  $w$ .
- Inductive Step: assume all optimal choices up to group  $g - 1$  are correct. For group  $g$ , each  $\text{dp}[g][w]$  is built from  $\text{dp}[g-1][w-w_i] + v_i$ , and storing the index  $i$  ensures reconstructing one valid optimal chain backward from  $(G, W)$ .

Thus, each backtracked choice sequence corresponds to one optimal solution.

## Try It Yourself

1. Add a limit on total number of groups selected.
2. Modify for multiple item selections per group.
3. Print group name instead of index.
4. Extend to 3D DP (group  $\times$  capacity  $\times$  budget).
5. Reconstruct second-best solution by skipping one choice.

## Test Cases

| Groups                                                          | W | Output                            |
|-----------------------------------------------------------------|---|-----------------------------------|
| <code>[[3,5),(2,3)], [(4,6),(1,2),(3,4)], [(2,4),(1,1)]]</code> | 7 | Value=13, picks=(3,5),(1,2),(2,4) |
| <code>[(2,3)], [(2,2),(3,5)]</code>                             | 5 | Value=8                           |
| <code>[(1,1),(2,4)], [(2,2),(3,5)]</code>                       | 4 | Value=6                           |

## Complexity

- Time:  $O(G \cdot W \cdot K)$  where  $K = \max$  group size
- Space:  $O(G \cdot W)$

Multi-Choice Reconstruction turns layered decision DPs into understandable sequences, revealing exactly what was chosen and why.

## 490 Traceback Visualization

Traceback Visualization is about seeing how a DP algorithm reconstructs its answer, turning invisible state transitions into a clear path of decisions. It converts a DP table into a narrative of moves, showing how each optimal solution is formed step by step.

### What Problem Are We Solving?

Most DP problems compute optimal values but hide how those values were reached. Traceback visualization helps us answer:

- Which transitions were taken?
- How do we get from the base case to the solution?
- What pattern does the DP follow through its table?

You're not changing the algorithm, you're revealing its story.

Common examples:

- Longest Common Subsequence (LCS): arrows tracing matches.
- Edit Distance: diagonal for match, up for delete, left for insert.
- Matrix Path Problems: arrows showing minimal path sum.
- Knapsack: table highlights selected cells.

### How Does It Work (Plain Language)

We reconstruct the DP solution visually:

1. Compute dp table as usual.
2. Start from the final state (e.g.  $dp[n][m]$ ).
3. Move backward following transitions that created the optimal value.
4. Record each step (arrow, direction, or explanation).
5. Draw path or print trace.

Each cell's transition reveals why it was chosen, minimal, maximal, or matching condition.

### Example: Edit Distance Visualization

Given strings  $A = \text{"kitten"}$ ,  $B = \text{"sitting"}$ , we compute  $dp[i][j] = \min$  edit distance between prefixes  $A[0..i]$  and  $B[0..j]$ .

We then trace back:

- If  $A[i - 1] = B[j - 1]$ : diagonal (match)
- Else:
  - if  $dp[i][j] = dp[i - 1][j - 1] + 1$ : substitution
  - if  $dp[i][j] = dp[i - 1][j] + 1$ : deletion
  - if  $dp[i][j] = dp[i][j - 1] + 1$ : insertion

Trace path: bottom-right  $\rightarrow$  top-left

### Tiny Code

Python

```
def edit_distance_trace(a, b):  
    n, m = len(a), len(b)  
    dp = [[0]*(m+1) for _ in range(n+1)]  
  
    for i in range(n+1):  
        dp[i][0] = i  
    for j in range(m+1):  
        dp[0][j] = j  
  
    for i in range(1, n+1):  
        for j in range(1, m+1):  
            if a[i-1] == b[j-1]:  
                dp[i][j] = dp[i-1][j-1]  
            else:  
                dp[i][j] = 1 + min(dp[i-1][j-1], dp[i-1][j], dp[i][j-1])  
  
    # Traceback  
    i, j = n, m  
    trace = []  
    while i > 0 or j > 0:  
        if i > 0 and j > 0 and a[i-1] == b[j-1]:  
            trace.append(f"Match {a[i-1]}")  
            i -= 1; j -= 1
```

```

        elif i > 0 and j > 0 and dp[i][j] == dp[i-1][j-1] + 1:
            trace.append(f"Substitute {a[i-1]} -> {b[j-1]}")
            i -= 1; j -= 1
        elif i > 0 and dp[i][j] == dp[i-1][j] + 1:
            trace.append(f"Delete {a[i-1]}")
            i -= 1
        else:
            trace.append(f"Insert {b[j-1]}")
            j -= 1
    trace.reverse()
    return dp[n][m], trace

dist, steps = edit_distance_trace("kitten", "sitting")
print("Distance:", dist)
print("\n".join(steps))

```

Output:

```

Distance: 3
Substitute k -> s
Match i
Match t
Match t
Substitute e -> i
Match n
Insert g

```

## Why It Matters

- Turns abstract numbers into concrete reasoning.
- Great for teaching and debugging DPs.
- Bridges computation and human understanding.
- Helps verify optimal path or detect incorrect transitions.

## A Gentle Proof (Why It Works)

By construction, each  $dp[i][j]$  is the minimal cost achievable. At reconstruction, we select any predecessor  $(i', j')$  satisfying the recurrence equality:

$$dp[i][j] = f(dp[i'][j']) + cost$$

Thus, following these equalities in reverse forms a valid optimal path. Since all  $dp$  states are filled bottom-up, each predecessor was computed earlier, ensuring consistency.

### Try It Yourself

1. Visualize LCS: arrows for match and direction.
2. Visualize min path sum in a grid: print arrows or coordinates.
3. Add ASCII arrows ( $\uparrow \leftarrow$ ) to the traceback.
4. Generate a heatmap with colored path.
5. Show decision explanation (e.g. “Insert g at pos 6”).

### Test Cases

| Input                      | Output Trace             |
|----------------------------|--------------------------|
| (“abc”, “abc”)             | Matches only             |
| (“kitten”, “sitting”)      | Substitute, Insert steps |
| (“intention”, “execution”) | Mixed edits              |
| (“horse”, “ros”)           | Delete + Substitute path |

### Complexity

- Time: same as DP,  $O(nm)$
- Space:  $O(nm)$  for table + path

Traceback visualization transforms the final DP value into a living path, so learners see how the algorithm’s reasoning unfolds across the grid.

# Section 50. Meta-DP and Optimization Templates

## 491 State Compression Template

State Compression is a powerful technique for reducing the dimensionality of dynamic programming problems by encoding multiple substate variables into a single integer mask. This allows us to handle problems that depend on subsets, visited states, or boolean configurations efficiently using bitwise operations.

### What Problem Are We Solving?

Many DP problems depend on a combination of states:

- Which elements have been used?
- Which nodes have been visited?
- What subset is currently active?

Representing these directly can explode in complexity. Instead, we compress the state into an integer bitmask, where each bit indicates the inclusion or status of an element.

For example:

- Traveling Salesman: mask tracks visited cities.
- Subset Sum: mask tracks sums possible with subsets.
- Graph DP: mask encodes chosen vertices.

### How Does It Work (Plain Language)

We represent subsets or multi-variable states with bitmasks:

- 1 bit → included, 0 bit → excluded.
- For  $n$  elements, we have  $2^n$  possible masks.

Each DP state looks like:

$$dp[mask] = \text{best result for subset represented by } mask$$

Transitions iterate over bits set/unset in the mask, updating dependent states.

Key operations:

- `mask | (1 << i)` → include element i
- `mask & (1 << i)` → check if i included
- `mask ^ (1 << i)` → toggle inclusion
- `mask & ~mask` → extract lowest set bit

### Example: Subset DP Template

$$dp[mask] = \min_{i \in mask} (dp[mask \setminus i] + cost[i])$$

Here, each `mask` represents a combination of items, and we build solutions incrementally by adding one element at a time.

### Tiny Code

C

```
#include <stdio.h>
#include <limits.h>

int min(int a, int b) { return a < b ? a : b; }

int main(void) {
    int n = 3;
    int cost[] = {3, 2, 5};
    int dp[1 << 3];

    for (int mask = 0; mask < (1 << n); mask++)
        dp[mask] = INT_MAX / 2;

    dp[0] = 0;

    for (int mask = 1; mask < (1 << n); mask++) {
        for (int i = 0; i < n; i++) {
            if ((mask & (1 << i)) == 0)
                dp[mask] = min(dp[mask], dp[mask ^ (1 << i)] + cost[i]);
        }
    }
}
```

```

        if (mask & (1 << i)) {
            int prev = mask ^ (1 << i);
            dp[mask] = min(dp[mask], dp[prev] + cost[i]);
        }
    }

    printf("Minimum total cost: %d\n", dp[(1 << n) - 1]);
}

```

Python

```

from math import inf

n = 3
cost = [3, 2, 5]
dp = [inf] * (1 << n)
dp[0] = 0

for mask in range(1, 1 << n):
    for i in range(n):
        if mask & (1 << i):
            dp[mask] = min(dp[mask], dp[mask ^ (1 << i)] + cost[i])

print("Minimum total cost:", dp[(1 << n) - 1])

```

## Why It Matters

- Compresses exponential states into manageable integer masks.
- Enables elegant solutions for combinatorial problems.
- Essential for TSP, Assignment, Subset DP, and Bitmask Knapsack.
- Fits perfectly with iterative DP loops.

## A Gentle Proof (Why It Works)

If  $dp[S]$  stores the optimal result for subset  $S$ , and every transition moves from smaller to larger subsets via one addition:

$$dp[S] = \min_{i \in S} (dp[S \setminus i] + cost[i])$$

Then by induction:

- Base case:  $dp[\emptyset]$  is known (often 0).
- Inductive step: each subset  $S$  builds on smaller subsets. All subsets are processed in increasing order of size, ensuring correctness.

### Try It Yourself

1. Implement Subset DP for Sum Over Subsets (SOS DP).
2. Solve Traveling Salesman using state compression.
3. Adapt to Assignment Problem ( $n! \rightarrow 2^n n$  states).
4. Use mask parity (even/odd bits) for combinatorial constraints.
5. Print masks in binary to visualize transitions.

### Test Cases

| Input            | Description           | Output |
|------------------|-----------------------|--------|
| cost = [3, 2, 5] | choose all 3 elements | 10     |
| cost = [1, 2]    | 2 elements            | 3      |
| cost = [5]       | single item           | 5      |

### Complexity

- Time:  $O(n \cdot 2^n)$
- Space:  $O(2^n)$

State Compression DP is your gateway to subset reasoning, compact, powerful, and fundamental for solving exponential combinatorial spaces with structure.

## 492 Transition Optimization Template

Transition Optimization is a core technique for improving the efficiency of DP transitions by precomputing or structuring recurrence updates. Many DP recurrences involve nested loops or repeated evaluations that can be simplified through mathematical properties, monotonicity, or auxiliary data structures.

## What Problem Are We Solving?

In many DPs, each state depends on a range or set of previous states:

$$dp[i] = \min_{j < i} (dp[j] + cost(j, i))$$

Naively, this takes  $O(n^2)$  time. But if  $cost(j, i)$  has special structure (monotonicity, convexity, quadrangle inequality), we can reduce it to  $O(n \log n)$  or even  $O(n)$  using optimized transitions.

Transition optimization finds patterns or data structures to accelerate these computations.

## How Does It Work (Plain Language)

When you notice repeated transitions like:

```
for (int i = 1; i <= n; i++)
    for (int j = 0; j < i; j++)
        dp[i] = min(dp[i], dp[j] + cost(j, i));
```

...you're paying an  $O(n^2)$  cost. But often,  $cost(j, i)$  follows a pattern (e.g. linear, convex, or monotonic), so we can optimize:

- Monotonic Queue Optimization: for sliding window minimums.
- Divide & Conquer DP: when optimal  $j$ 's move monotonically.
- Convex Hull Trick: when  $cost(j, i) = m_j \cdot x_i + b_j$  is linear.
- Knuth Optimization: when quadrangle inequality holds.

Each approach precomputes or narrows transitions.

## Example Transition (Generic)

$$dp[i] = \min_{j < i} (dp[j] + f(j, i))$$

If  $f$  satisfies the Monge property or quadrangle inequality, we can determine that the optimal  $j$  moves in one direction only (monotonic). That means we can use divide & conquer or pointer tricks to find it efficiently.

## Tiny Code

C (Naive Transition)

```
for (int i = 1; i <= n; i++) {
    dp[i] = INF;
    for (int j = 0; j < i; j++) {
        int candidate = dp[j] + cost(j, i);
        if (candidate < dp[i])
            dp[i] = candidate;
    }
}
```

C (Optimized with Monotonic Pointer)

```
int ptr = 0;
for (int i = 1; i <= n; i++) {
    while (ptr + 1 < i && better(ptr + 1, ptr, i))
        ptr++;
    dp[i] = dp[ptr] + cost(ptr, i);
}
```

Here `better(a, b, i)` checks whether `a` gives a smaller cost than `b` for `dp[i]`.

## Python (Sliding Window Optimization)

```
from collections import deque

dp = [0] * (n + 1)
q = deque([0])

for i in range(1, n + 1):
    while len(q) >= 2 and better(q[1], q[0], i):
        q.popleft()
    j = q[0]
    dp[i] = dp[j] + cost(j, i)
    while len(q) >= 2 and cross(q[-2], q[-1], i):
        q.pop()
    q.append(i)
```

This structure appears in Convex Hull Trick and Monotonic Queue Optimization.

## Why It Matters

- Reduces  $O(n^2) \rightarrow O(n \log n)$  or  $O(n)$  transitions.
- Exploits structure (monotonicity, convexity) in DP cost functions.
- Powers major optimizations:
  - Knuth Optimization
  - Divide & Conquer DP
  - Convex Hull Trick
  - Slope Trick
  - Monotone Queue DP

## A Gentle Proof (Why It Works)

If the recurrence satisfies Monotonicity of the Argmin, i.e.:

$$opt[i] \leq opt[i + 1]$$

then the best transition index  $j$  moves non-decreasingly. This means we can find optimal  $j$  for all  $i$  in one sweep, using either:

- Two-pointer traversal (Monotone Queue)
- Divide & Conquer recursion (Knuth or D&C DP)
- Line container (Convex Hull Trick)

By exploiting this structure, we avoid recomputation.

## Try It Yourself

1. Identify a DP where each state depends on a range of previous states.
2. Check if  $\text{cost}(j, i)$  satisfies monotonic or convex properties.
3. Apply divide & conquer optimization to reduce  $O(n^2)$ .
4. Implement Convex Hull Trick for linear cost forms.
5. Use deque-based Monotonic Queue for sliding range DP.

| Case | Recurrence | Optimization |
|------|------------|--------------|
|------|------------|--------------|

## Test Cases

| Case                                      | Recurrence      | Optimization      |
|-------------------------------------------|-----------------|-------------------|
| $dp[i] = \min_{j < i} (dp[j] + c(i - j))$ | $c$ convex      | Convex Hull Trick |
| $dp[i] = \min_{j < i} (dp[j] + w(j, i))$  | Monotone argmin | Divide & Conquer  |
| $dp[i] = \min_{j < i} (dp[j]) + a_i$      | sliding window  | Monotonic Queue   |

## Complexity

- Time:  $O(n)$  to  $O(n \log n)$  (depends on method)
- Space:  $O(n)$

Transition Optimization is the art of seeing structure in cost, once you spot monotonicity or convexity, your DP becomes faster, cleaner, and smarter.

## 493 Space Optimization Template

Space Optimization is the art of trimming away unused dimensions in a DP table by realizing that only a limited subset of previous states is needed at each step. Many classic DPs that start with large  $O(n^2)$  or  $O(nm)$  tables can be reduced to rolling arrays or single-row updates, cutting memory usage drastically.

### What Problem Are We Solving?

Dynamic Programming often uses multi-dimensional arrays:

$$dp[i][j] = \text{answer using first } i \text{ items with capacity } j$$

But not all dimensions are necessary. If each state  $dp[i]$  only depends on previous row  $dp[i - 1]$ , we can reuse memory, keeping just two rows (or even one).

Space Optimization lets us move from  $O(nm)$  to  $O(m)$ , or from 2D  $\rightarrow$  1D, or 3D  $\rightarrow$  2D, without changing logic.

## How Does It Work (Plain Language)

DP updates come from previous states, not all states.

For example, in 0/1 Knapsack:

$$dp[i][w] = \max(dp[i - 1][w], dp[i - 1][w - wt[i]] + val[i])$$

Only  $dp[i-1][*]$  is needed when computing  $dp[i][*]$ . So we can collapse the DP table into a single array  $dp[w]$ , updating it in reverse (to avoid overwriting states we still need).

If transitions depend on current or previous row, choose direction carefully:

- 0/1 Knapsack → reverse loop
- Unbounded Knapsack → forward loop

## Example Transformation

Before (2D DP)

```
int dp[n+1][W+1];
for (int i = 1; i <= n; i++)
    for (int w = 0; w <= W; w++)
        dp[i][w] = max(dp[i-1][w], dp[i-1][w-wt[i]] + val[i]);
```

After (1D DP)

```
int dp[W+1] = {0};
for (int i = 1; i <= n; i++)
    for (int w = W; w >= wt[i]; w--)
        dp[w] = max(dp[w], dp[w-wt[i]] + val[i]);
```

## Tiny Code

C (Rolling Array Example)

```

#include <stdio.h>
#define max(a,b) ((a)>(b)?(a):(b))

int main(void) {
    int n = 3, W = 5;
    int wt[] = {0, 2, 3, 4};
    int val[] = {0, 4, 5, 7};
    int dp[6] = {0};

    for (int i = 1; i <= n; i++)
        for (int w = W; w >= wt[i]; w--)
            dp[w] = max(dp[w], dp[w - wt[i]] + val[i]);

    printf("Max value: %d\n", dp[W]);
}

```

Python (1D Rolling)

```

n, W = 3, 5
wt = [2, 3, 4]
val = [4, 5, 7]

dp = [0] * (W + 1)

for i in range(n):
    for w in range(W, wt[i] - 1, -1):
        dp[w] = max(dp[w], dp[w - wt[i]] + val[i])

print("Max value:", dp[W])

```

## Why It Matters

- Reduces memory from  $O(nm) \rightarrow O(m)$ .
- Makes large DP problems feasible under memory limits.
- Reveals dependency structure in transitions.
- Forms the backbone of iterative bottom-up optimization.

Space optimization is vital for:

- Knapsack, LCS, LIS
- Grid path counting
- Partition problems

- Digit DP (carry compression)

### A Gentle Proof (Why It Works)

Let's define  $dp[i][j]$  depending only on  $dp[i - 1][*]$ . Since each new row is computed solely from the previous one:

$$dp[i][j] = f(dp[i - 1][j], dp[i - 1][j - w_i])$$

So at iteration  $i$ , once  $dp[i][*]$  is complete,  $dp[i-1][*]$  is never used again. By updating in reverse (to preserve dependencies), the 2D table can be rolled into one.

Formally, space can be reduced from  $O(nm)$  to  $O(m)$  if and only if:

1. Each  $dp[i]$  depends on  $dp[i - 1]$ , not  $dp[i]$  itself.
2. Transition direction ensures previous states remain unmodified.

### Try It Yourself

1. Convert your 0/1 Knapsack to 1D DP.
2. Space-optimize the LCS table ( $2D \rightarrow 2$  rows).
3. Apply to “Climbing Stairs” ( $dp[i]$  only needs last 2 values).
4. For Unbounded Knapsack, try forward updates.
5. Compare memory usage before and after.

### Test Cases

| Problem      | Original Space | Optimized Space |
|--------------|----------------|-----------------|
| 0/1 Knapsack | $O(nW)$        | $O(W)$          |
| LCS          | $O(nm)$        | $O(2m)$         |
| Fibonacci    | $O(n)$         | $O(1)$          |

### Complexity

- Time: unchanged
- Space: reduced by 1 dimension
- Tradeoff: direction of iteration matters

Space Optimization is a quiet revolution: by recognizing independence between layers, we free our algorithms from unnecessary memory, one dimension at a time.

## 494 Multi-Dimensional DP Template

Multi-Dimensional DP extends classic one- or two-dimensional formulations into higher-dimensional state spaces, capturing problems where multiple independent variables evolve together. Each dimension corresponds to a decision axis, time, position, capacity, or some discrete property, making it possible to express rich combinatorial or structural relationships.

### What Problem Are We Solving?

Some problems require tracking more than one evolving parameter:

- Knapsack with two capacities →  $dp[i][w_1][w_2]$
- String interleaving →  $dp[i][j][k]$
- Dice sum counting →  $dp[i][sum][count]$
- Grid with keys →  $dp[x][y][mask]$

When multiple independent factors drive state transitions, a single index DP cannot capture them. Multi-Dimensional DP encodes joint state evolution explicitly.

### How Does It Work (Plain Language)

We define a DP table where each axis tracks a property:

$$dp[a][b][c] = \text{best result with parameters } (a, b, c)$$

Transitions update along one or more dimensions:

$$dp[a][b][c] = \min / \max(\text{transitions from neighbors})$$

Think of this as traversing a grid of states, where each move modifies several parameters. The key idea is to fill the table systematically based on topological or nested loops that respect dependency order.

## Example Recurrence

Multi-dimensional structure often looks like:

$$dp[i][j][k] = f(dp[i-1][j'][k'], \text{cost}(i, j, k))$$

Example (2D Knapsack):

$$dp[i][w_1][w_2] = \max(dp[i-1][w_1][w_2], dp[i-1][w_1 - wt_1[i]][w_2 - wt_2[i]] + val[i])$$

## Tiny Code

C (2D Knapsack)

```
#include <stdio.h>
#define max(a,b) ((a)>(b)?(a):(b))

int main(void) {
    int n = 3, W1 = 5, W2 = 5;
    int wt1[] = {0, 2, 3, 4};
    int wt2[] = {0, 1, 2, 3};
    int val[] = {0, 4, 5, 6};
    int dp[4][6][6] = {0};

    for (int i = 1; i <= n; i++) {
        for (int w1 = 0; w1 <= W1; w1++) {
            for (int w2 = 0; w2 <= W2; w2++) {
                dp[i][w1][w2] = dp[i-1][w1][w2];
                if (w1 >= wt1[i] && w2 >= wt2[i]) {
                    dp[i][w1][w2] = max(dp[i][w1][w2],
                                         dp[i-1][w1 - wt1[i]][w2 - wt2[i]] + val[i]);
                }
            }
        }
    }

    printf("Max value: %d\n", dp[n][W1][W2]);
}
```

Python (3D Example: String Interleaving)

```

s1, s2, s3 = "ab", "cd", "acbd"
n1, n2, n3 = len(s1), len(s2), len(s3)

dp = [[[False]*(n3+1) for _ in range(n2+1)] for _ in range(n1+1)]
dp[0][0][0] = True

for i in range(n1+1):
    for j in range(n2+1):
        for k in range(n3+1):
            if k == 0: continue
            if i > 0 and s1[i-1] == s3[k-1] and dp[i-1][j][k-1]:
                dp[i][j][k] = True
            if j > 0 and s2[j-1] == s3[k-1] and dp[i][j-1][k-1]:
                dp[i][j][k] = True

print("Interleaving possible:", dp[n1][n2][n3])

```

## Why It Matters

- Captures multi-factor problems elegantly
- Handles constraints coupling (capacity, index, sum)
- Enables state compression when reduced
- Common in:
  - Multi-resource allocation
  - Interleaving / sequence merging
  - Multi-knapsack / bounded subset
  - Grid navigation with additional properties

Multi-dimensional DPs form the foundation of generalized search spaces, where each variable adds a dimension of reasoning.

## A Gentle Proof (Why It Works)

By induction over the outermost dimension:

If  $dp[i][*][*]$  depends only on  $dp[i-1][*][*]$ , and each transition moves from smaller to larger indices, then the DP fills in topological order, ensuring correctness.

Each additional dimension multiplies the state space but does not alter dependency direction. Thus, correctness holds as long as we respect dimension order and initialize base cases properly.

## Try It Yourself

1. Solve 2D Knapsack with dual capacity.
2. Implement string interleaving check with 3D DP.
3. Model shortest path in 3D grid using  $dp[x][y][z]$ .
4. Add bitmask dimension for subset tracking.
5. Optimize memory using rolling or compression.

## Test Cases

| Problem             | Dimensions        | Example State    | Output     |
|---------------------|-------------------|------------------|------------|
| 2D Knapsack         | 3D (item, w1, w2) | $dp[i][w1][w2]$  | Max value  |
| String Interleaving | 3D                | $dp[i][j][k]$    | True/False |
| Grid with Keys      | 3D                | $dp[x][y][mask]$ | Min steps  |

## Complexity

- Time:  $O(\text{product of dimensions})$
- Space: same order; compressible via rolling
- Tradeoff: richer state space vs feasibility

Multi-Dimensional DP is your tool for multi-constraint reasoning, when life refuses to fit in one dimension, let your DP grow an extra axis.

## 495 Decision Monotonicity

Decision Monotonicity is a structural property in DP recurrences that allows us to optimize transition search. When the optimal decision index for  $dp[i]$  moves in one direction (non-decreasing) as  $i$  increases, we can reduce a naive  $O(n^2)$  DP to  $O(n \log n)$  or even  $O(n)$  using divide-and-conquer or two-pointer techniques.

## What Problem Are We Solving?

In many DPs, each state  $dp[i]$  is computed by choosing a best transition point  $j < i$ :

$$dp[i] = \min_{0 \leq j < i} (dp[j] + cost(j, i))$$

This naive recurrence requires trying all previous states for every  $i$ , leading to  $O(n^2)$  time. But if the index of the optimal  $j$  (called  $opt[i]$ ) satisfies:

$$opt[i] \leq opt[i + 1]$$

then the decision index moves monotonically, and we can search efficiently, either by divide & conquer DP or sliding pointer optimization.

### How Does It Work (Plain Language)

If as  $i$  increases, the best  $j$  never moves backward, we can reuse or narrow the search for each next state.

In other words:

- The “best split point” for  $i = 10$  will be at or after the best split for  $i = 9$ .
- No need to re-check smaller  $j$  again.
- You can sweep  $j$  forward or recursively restrict the range.

This property appears when  $cost(j, i)$  satisfies certain quadrangle inequalities or convexity conditions.

### Example Recurrence

$$dp[i] = \min_{j < i} (dp[j] + (i - j)^2)$$

Here, as  $i$  grows, larger  $j$  become more favorable because  $(i - j)^2$  penalizes small gaps. Thus,  $opt[i]$  increases monotonically.

Another example:

$$dp[i] = \min_{j < i} (dp[j] + c[j] \cdot a[i])$$

where  $a[i]$  is increasing, the convex hull trick applies, and optimal lines appear in increasing order.

### Tiny Code (Two-Pointer Monotonic Search)

C

```

int n = ...;
int dp[MAXN];
int opt[MAXN];
for (int i = 1; i <= n; i++) {
    dp[i] = INF;
    int start = opt[i-1];
    for (int j = start; j <= i; j++) {
        int val = dp[j] + cost(j, i);
        if (val < dp[i]) {
            dp[i] = val;
            opt[i] = j;
        }
    }
}

```

Each  $opt[i]$  begins searching from  $opt[i - 1]$ , cutting redundant checks.

### Python (Divide & Conquer Optimization)

```

def solve(l, r, optL, optR):
    if l > r: return
    mid = (l + r) // 2
    best = (float('inf'), -1)
    for j in range(optL, min(optR, mid) + 1):
        val = dp[j] + cost(j, mid)
        if val < best[0]:
            best = (val, j)
    dp[mid] = best[0]
    opt = best[1]
    solve(l, mid - 1, optL, opt)
    solve(mid + 1, r, opt, optR)

solve(1, n, 0, n-1)

```

### Why It Matters

- Reduces complexity from  $O(n^2) \rightarrow O(n \log n)$  or  $O(n)$
- Enables Divide & Conquer DP, Knuth Optimization, and Convex Hull Trick
- Builds foundation for structured cost functions
- Helps identify monotonic transitions in scheduling, partitioning, or chain DPs

## A Gentle Proof (Why It Works)

If  $opt[i] \leq opt[i+1]$ , then  $dp[i]$ 's optimal transition comes from no earlier than  $opt[i-1]$ . Thus, we can safely restrict search intervals:

$$dp[i] = \min_{j \in [opt[i-1], i-1]} f(j, i)$$

The proof follows from quadrangle inequality:

$$f(a, c) + f(b, d) \leq f(a, d) + f(b, c)$$

which ensures convex-like structure and monotone decisions.

By induction:

- Base:  $opt[1]$  known.
- Step: if  $opt[i] \leq opt[i+1]$ , then the recurrence preserves order.

## Try It Yourself

1. Implement a divide & conquer DP with  $opt$  tracking.
2. Verify monotonicity of  $opt[i]$  experimentally for a sample cost.
3. Apply to partitioning problems like Divide Array into K Segments.
4. Compare  $O(n^2)$  vs optimized  $O(n \log n)$  performance.
5. Check if your cost satisfies quadrangle inequality.

## Test Cases

| Recurrence                                       | Property           | Optimization     |
|--------------------------------------------------|--------------------|------------------|
| $dp[i] = \min_{j < i} (dp[j] + (i - j)^2)$       | convex             | monotone opt     |
| $dp[i] = \min_{j < i} (dp[j] + a[i] \cdot b[j])$ | increasing $a[i]$  | convex hull      |
| Segment DP                                       | $cost(l, r)$ Monge | divide & conquer |

## Complexity

- Time:  $O(n \log n)$  (divide & conquer) or  $O(n)$  (two-pointer)
- Space:  $O(n)$

Decision Monotonicity is the hidden geometry of DP, once you spot that the “best index” moves only forward, your algorithm speeds up dramatically.

## 496 Monge Array Optimization

Monge Array Optimization is a powerful tool for accelerating dynamic programming when the cost matrix satisfies a special inequality known as the Monge property. It guarantees that the argmin of each row moves monotonically across columns, allowing us to use Divide & Conquer DP or SMAWK algorithm for subquadratic optimization.

### What Problem Are We Solving?

Consider a DP of the form:

$$dp[i][j] = \min_{k < j} (dp[i - 1][k] + cost[k][j])$$

If the cost matrix  $cost[k][j]$  satisfies the Monge property, we can compute all  $dp[i][j]$  in  $O(n \log n)$  or  $O(n)$  per layer, instead of the naive  $O(n^2)$ .

This pattern appears in:

- Partition DP (divide sequence into segments)
- Matrix Chain / Knuth DP
- Optimal merge / segmentation

### How Does It Work (Plain Language)

The Monge property states that for all  $a < b$  and  $c < d$ :

$$cost[a][c] + cost[b][d] \leq cost[a][d] + cost[b][c]$$

This means the difference in cost is consistent across diagonals, implying convexity in two dimensions. As a result, the optimal split point moves monotonically:

$$opt[i][j] \leq opt[i][j + 1]$$

We can therefore restrict our search range for  $dp[i][j]$  using Divide & Conquer optimization.

## Example Recurrence

For segment partitioning:

$$dp[i][j] = \min_{k < j} (dp[i-1][k] + cost[k][j])$$

If  $cost[k][j]$  is Monge, then  $opt[i][j] \leq opt[i][j+1]$ . Thus, when computing  $dp[i][j]$ , we only need to search  $k$  in  $[opt[i][j-1], opt[i][j+1]]$ .

## Tiny Code (Divide & Conquer over Monge Matrix)

C (Template)

```
void compute(int i, int l, int r, int optL, int optR) {
    if (l > r) return;
    int mid = (l + r) / 2;
    int best_k = -1;
    long long best_val = LLONG_MAX;
    for (int k = optL; k <= optR && k < mid; k++) {
        long long val = dp_prev[k] + cost[k][mid];
        if (val < best_val) {
            best_val = val;
            best_k = k;
        }
    }
    dp[mid] = best_val;
    compute(i, l, mid - 1, optL, best_k);
    compute(i, mid + 1, r, best_k, optR);
}
```

Each recursive call computes a segment's midpoint and recursively narrows the search range based on monotonicity.

Python (Monge DP Skeleton)

```
def compute(i, l, r, optL, optR):
    if l > r:
        return
    mid = (l + r) // 2
    best = (float('inf'), -1)
    for k in range(optL, min(optR, mid) + 1):
```

```

val = dp_prev[k] + cost[k][mid]
if val < best[0]:
    best = (val, k)
dp[mid] = best[0]
opt[mid] = best[1]
compute(i, l, mid - 1, optL, best[1])
compute(i, mid + 1, r, best[1], optR)

```

## Why It Matters

- Exploits Monge property to skip redundant transitions
- Reduces 2D DP to  $O(n \log n)$  or even  $O(n)$  per layer
- Powers optimizations like:
  - Divide & Conquer DP
  - Knuth Optimization (special Monge case)
  - SMAWK algorithm (row minima in Monge arrays)

Used in:

- Sequence segmentation
- Matrix chain multiplication
- Optimal BST
- Inventory / scheduling models

## A Gentle Proof (Why It Works)

If  $cost$  satisfies Monge inequality:

$$cost[a][c] + cost[b][d] \leq cost[a][d] + cost[b][c]$$

then:

$$opt[j] \leq opt[j + 1]$$

That is, as  $j$  increases, the best  $k$  (split point) cannot move backward. Hence, when computing  $dp[j]$ , we can reuse or narrow the search interval using the previous opt index.

This monotonicity of argmin is the key to divide-and-conquer speedups.

## Try It Yourself

1. Verify Monge property for your cost function.
2. Implement the Divide & Conquer DP template.
3. Test on partition DP with convex segment cost.
4. Compare  $O(n^2)$  vs optimized  $O(n \log n)$  runtime.
5. Explore SMAWK for row minima in Monge matrices.

## Test Cases

| Cost Function                      | Monge? | Optimization |
|------------------------------------|--------|--------------|
| $cost[a][b] = (sum[b] - sum[a])^2$ | Yes    |              |
| $cost[a][b] = (b - a)^2$           | Yes    |              |
| $\$cost[a][b] =$                   | b-a    | \$           |
|                                    |        | No           |

## Complexity

- Time:  $O(n \log n)$  per layer
- Space:  $O(n)$
- Layers: multiply by  $k$  if multi-stage DP

Monge Array Optimization transforms a naive DP table into a structured landscape, once your costs align, transitions fall neatly into place with logarithmic grace.

## 497 Divide & Conquer Template

Divide & Conquer DP is a technique for optimizing DP transitions when the optimal transition index exhibits monotonicity. By recursively dividing the problem and searching only within a limited range for each midpoint, we reduce complexity from  $O(n^2)$  to  $O(n \log n)$  or even  $O(n)$  per layer.

## What Problem Are We Solving?

Many DP formulations involve transitions like:

$$dp[i] = \min_{j < i} (dp[j] + cost(j, i))$$

If  $opt[i] \leq opt[i + 1]$ , meaning the best transition index moves monotonically forward, we can use divide and conquer to find optimal  $j$  efficiently instead of scanning all  $j < i$ .

This structure is common in:

- Partition DP (divide array into  $k$  segments)
- Monge or Convex cost problems
- Segment-based recurrence with monotone argmin

## How Does It Work (Plain Language)

We recursively divide the range  $[L, R]$ , compute  $dp[mid]$  using the best transition from a restricted interval  $[optL, optR]$ , then:

- Left half  $[L, mid - 1]$  searches  $[optL, opt[mid]]$
- Right half  $[mid + 1, R]$  searches  $[opt[mid], optR]$

By maintaining monotone search boundaries, we ensure correctness and avoid redundant checks.

Think of it as a guided binary search over DP indices, powered by structural guarantees.

## Example Recurrence

$$dp[i] = \min_{j < i} (dp[j] + cost(j, i))$$

If  $cost$  satisfies quadrangle inequality or Monge property, then:

$$opt[i] \leq opt[i + 1]$$

Thus, we can recursively compute  $dp$  over subranges.

## Tiny Code (C)

```
#include <stdio.h>
#include <limits.h>

#define INF 1000000000
#define min(a,b) ((a)<(b)?(a):(b))

int n;
```

```

int dp[10005], prev_dp[10005];

// Example cost function (prefix sums)
int prefix[10005];
int cost(int j, int i) {
    int sum = prefix[i] - prefix[j];
    return sum * sum;
}

void compute(int l, int r, int optL, int optR) {
    if (l > r) return;
    int mid = (l + r) / 2;
    int best_k = -1;
    int best_val = INF;

    for (int k = optL; k <= optR && k < mid; k++) {
        int val = prev_dp[k] + cost(k, mid);
        if (val < best_val) {
            best_val = val;
            best_k = k;
        }
    }

    dp[mid] = best_val;

    // Recurse left and right halves
    compute(l, mid - 1, optL, best_k);
    compute(mid + 1, r, best_k, optR);
}

int main(void) {
    n = 5;
    int arr[] = {0, 1, 2, 3, 4, 5};
    for (int i = 1; i <= n; i++) prefix[i] = prefix[i-1] + arr[i];
    for (int i = 0; i <= n; i++) prev_dp[i] = i*i;

    compute(1, n, 0, n-1);

    for (int i = 1; i <= n; i++) printf("dp[%d] = %d\n", i, dp[i]);
}

```

Python

```

def cost(j, i):
    s = prefix[i] - prefix[j]
    return s * s

def compute(l, r, optL, optR):
    if l > r:
        return
    mid = (l + r) // 2
    best = (float('inf'), -1)
    for k in range(optL, min(optR, mid) + 1):
        val = prev_dp[k] + cost(k, mid)
        if val < best[0]:
            best = (val, k)
    dp[mid], opt[mid] = best
    compute(l, mid - 1, optL, best[1])
    compute(mid + 1, r, best[1], optR)

n = 5
arr = [0, 1, 2, 3, 4, 5]
prefix = [0]
for x in arr: prefix.append(prefix[-1] + x)
prev_dp = [i*i for i in range(len(arr))]
dp = [0]*(n+1)
opt = [0]*(n+1)
compute(1, n, 0, n-1)
print(dp[1:])

```

## Why It Matters

- Reduces complexity dramatically:  $O(n \log n)$  per layer
- Works on structured recurrences with monotonic  $opt[i]$
- Forms backbone for:
  - Knuth Optimization
  - Monge Array DP
  - Segment Partition DP

You can think of it as “binary search for DP transitions.”

## A Gentle Proof (Why It Works)

If  $opt[i] \leq opt[i + 1]$ , then each  $dp[mid]$ 's optimal index  $k^*$  lies between  $optL$  and  $optR$ . When dividing the range:

- Left child  $(L, mid - 1)$  searches  $[optL, k^*]$
- Right child  $(mid + 1, R)$  searches  $[k^*, optR]$

By induction, every segment explores only valid transitions. Since each  $k$  is visited  $O(\log n)$  times, total time is  $O(n \log n)$ .

## Try It Yourself

1. Implement partition DP with convex segment cost.
2. Verify monotonicity of  $opt[i]$  numerically.
3. Compare  $O(n^2)$  vs optimized  $O(n \log n)$ .
4. Combine with space optimization (roll arrays).
5. Extend to multi-layer DP (e.g., k-partition).

## Test Cases

| Recurrence                                           | Property | Optimization |
|------------------------------------------------------|----------|--------------|
| $dp[i] = \min_{j < i} (dp[j] + (sum[i] - sum[j])^2)$ | convex   | yes          |
| $dp[i] = \min_{j < i} (dp[j] +  i - j )$             | linear   | yes          |
| $dp[i] = \min_{j < i} (dp[j] + cost[j][i])$          | Monge    | yes          |

## Complexity

- Time:  $O(n \log n)$  per layer
- Space:  $O(n)$
- Layers: multiply by  $k$  for multi-stage DP

Divide & Conquer DP is your scalpel for quadratic DPs, once you find monotonicity, you slice complexity cleanly in half at every level.

## 498 Rerooting Template

Rerooting DP is a powerful tree dynamic programming pattern that lets you compute results for every node as the root, efficiently reusing computations from parent-to-child transitions. It's like rotating the tree root through all nodes without recomputing everything from scratch.

## What Problem Are We Solving?

Given a tree, we often want to compute a property for each node as if it were the root. For example:

- Sum of distances to all nodes
- Size of subtree or value based on children
- Number of valid colorings rooted at each node

Naively, you could rerun DP for every node,  $O(n^2)$ , but rerooting reduces this to  $O(n)$  or  $O(n \log n)$  by cleverly reusing partial results.

## How Does It Work (Plain Language)

1. First pass (postorder): compute DP values bottom-up for a fixed root (usually node 1).
2. Second pass (preorder): propagate results top-down, rerooting along each edge and combining parent contributions.

When moving the root from  $u$  to  $v$ :

- Remove  $v$ 's contribution from  $u$ 's DP.
- Add  $u$ 's contribution (excluding  $v$ ) into  $v$ 's DP.

This way, each node inherits a correct rerooted DP in one traversal.

## Example Problem

Compute sum of distances from every node to all others.

Let:

- $dp[u]$  = sum of distances from  $u$  to all nodes in its subtree
- $sz[u]$  = size of subtree of  $u$

We can reroot using:

$$dp[v] = dp[u] - sz[v] + (n - sz[v])$$

when moving root from  $u$  to child  $v$ .

## Tiny Code (C)

```

#include <stdio.h>
#include <vector>

#define MAXN 100005
using namespace std;

vector<int> adj[MAXN];
int n;
int sz[MAXN];
long long dp[MAXN];
long long ans[MAXN];

void dfs1(int u, int p) {
    sz[u] = 1;
    dp[u] = 0;
    for (int v : adj[u]) if (v != p) {
        dfs1(v, u);
        sz[u] += sz[v];
        dp[u] += dp[v] + sz[v];
    }
}

void dfs2(int u, int p) {
    ans[u] = dp[u];
    for (int v : adj[u]) if (v != p) {
        long long dp_u = dp[u], dp_v = dp[v];
        int sz_u = sz[u], sz_v = sz[v];

        // Move root from u to v
        dp[u] -= dp[v] + sz[v];
        sz[u] -= sz[v];
        dp[v] += dp[u] + sz[u];
        sz[v] += sz[u];

        dfs2(v, u);

        // Restore
        dp[u] = dp_u; dp[v] = dp_v;
        sz[u] = sz_u; sz[v] = sz_v;
    }
}

```

```

int main(void) {
    scanf("%d", &n);
    for (int i = 0; i < n-1; i++) {
        int u, v;
        scanf("%d%d", &u, &v);
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    dfs1(1, -1);
    dfs2(1, -1);
    for (int i = 1; i <= n; i++)
        printf("Sum of distances from %d: %lld\n", i, ans[i]);
}

```

Python

```

from collections import defaultdict

n = 5
adj = defaultdict(list)
edges = [(1,2),(1,3),(3,4),(3,5)]
for u,v in edges:
    adj[u].append(v)
    adj[v].append(u)

sz = [0]*(n+1)
dp = [0]*(n+1)
ans = [0]*(n+1)

def dfs1(u,p):
    sz[u] = 1
    dp[u] = 0
    for v in adj[u]:
        if v == p: continue
        dfs1(v,u)
        sz[u] += sz[v]
        dp[u] += dp[v] + sz[v]

def dfs2(u,p):
    ans[u] = dp[u]
    for v in adj[u]:
        if v == p: continue

```

```

dp_u, dp_v = dp[u], dp[v]
sz_u, sz_v = sz[u], sz[v]

dp[u] -= dp[v] + sz[v]
sz[u] -= sz[v]
dp[v] += dp[u] + sz[u]
sz[v] += sz[u]

dfs2(v,u)

dp[u], dp[v] = dp_u, dp_v
sz[u], sz[v] = sz_u, sz_v

dfs1(1,-1)
dfs2(1,-1)

for i in range(1,n+1):
    print(f"Sum of distances from {i}: {ans[i]}")

```

## Why It Matters

- Enables  $O(n)$  computation of per-node DP values.
- Reuses child and parent information via reversible transitions.
- Crucial for:
  - Distance sums
  - Subtree aggregations
  - Coloring and constraint propagation

You can reroot any tree once you know how to move contributions.

## A Gentle Proof (Why It Works)

The rerooting relation ensures:

- $dp[u]$  stores full-tree values when rooted at  $u$ .
- When rerooting to  $v$ , subtract  $v$ 's contribution from  $u$ , then add  $u$ 's contribution to  $v$ .

Because each edge is traversed twice, total complexity is  $O(n)$ .

This is a direct application of DP reusability under tree decomposition.

## Try It Yourself

1. Compute subtree sums and reroot to get sum of values at distance  $k$ .
2. Apply rerooting to count paths passing through each node.
3. Modify transitions for tree coloring or centroid scoring.
4. Visualize contribution flow parent child.

## Test Cases

| Tree               | Query         | Output     |
|--------------------|---------------|------------|
| Line (1–2–3–4)     | Distance sums | 6, 4, 4, 6 |
| Star (1–2,1–3,1–4) | Distance sums | 3, 5, 5, 5 |
| Balanced tree      | Aggregation   | symmetric  |

## Complexity

- Time:  $O(n)$
- Space:  $O(n)$

Rerooting DP is your “walk the tree” trick, one bottom-up pass, one top-down pass, and you know what every node would see if it stood at the root.

## 499 Iterative DP Pattern

Iterative (bottom-up) dynamic programming is the most systematic and efficient way to compute state-based solutions. Instead of recursion and memoization, we explicitly build tables in increasing order of dependency, turning recurrence relations into simple loops.

### What Problem Are We Solving?

When you have a recurrence like:

$$dp[i] = f(dp[i - 1], dp[i - 2], \dots)$$

you don’t need recursion, you can iterate from base to target. This approach avoids call stack overhead, ensures predictable memory access, and simplifies debugging.

Iterative DP is ideal for:

- Counting problems (e.g. Fibonacci, climbing stairs)
- Path minimization (e.g. shortest path, knapsack)
- Sequence alignment (e.g. LCS, edit distance)

## How Does It Work (Plain Language)

1. Define the state  $dp[i]$ : what does it represent?
2. Identify base cases (e.g.  $dp[0]$ ,  $dp[1]$ ).
3. Establish transition using smaller states.
4. Iterate from smallest to largest index, ensuring dependencies are filled before use.
5. Extract result (e.g.  $dp[n]$  or  $\max_i dp[i]$ ).

The iteration order must match dependency direction.

### Example: Climbing Stairs

You can climb either 1 or 2 steps at a time. Number of ways to reach step  $n$ :

$$dp[i] = dp[i - 1] + dp[i - 2]$$

with base cases  $dp[0] = 1$ ,  $dp[1] = 1$ .

### Tiny Code (C)

```
#include <stdio.h>

int main() {
    int n = 5;
    int dp[6];
    dp[0] = 1;
    dp[1] = 1;
    for (int i = 2; i <= n; i++)
        dp[i] = dp[i-1] + dp[i-2];
    printf("Ways to climb %d stairs: %d\n", n, dp[n]);
}
```

Python

```

n = 5
dp = [0]*(n+1)
dp[0] = dp[1] = 1
for i in range(2, n+1):
    dp[i] = dp[i-1] + dp[i-2]
print(f"Ways to climb {n} stairs: {dp[n]}")

```

## Why It Matters

- Performance: Iteration eliminates recursion overhead.
- Clarity: Each state is computed once, in a known order.
- Memory Optimization: You can reduce space when only recent states are needed (rolling array).
- Foundation: All advanced DPs (knapsack, edit distance, LIS) can be written iteratively.

## A Gentle Proof (Why It Works)

If  $dp[i]$  depends only on smaller indices, then filling  $dp[0 \dots n]$  in order guarantees correctness.

By induction:

- Base cases true by definition.
- Assuming  $dp[0..i - 1]$  correct, then  $dp[i] = f(dp[0..i - 1])$  produces correct result.

No state is used before it's computed.

## Try It Yourself

1. Implement iterative Fibonacci with constant space.
2. Convert recursive knapsack into iterative table form.
3. Write bottom-up LCS for two strings.
4. Try 2D iterative DP for grid paths.

## Test Cases

| Input    | Expected Output |
|----------|-----------------|
| $n = 0$  | 1               |
| $n = 1$  | 1               |
| $n = 5$  | 8               |
| $n = 10$ | 89              |

| Input | Expected Output |
|-------|-----------------|
|       |                 |

## Complexity

- Time:  $O(n)$
- Space:  $O(n)$  (or  $O(1)$  with rolling array)

Iterative DP is the canonical form, the simplest, most direct way to think about recursion unrolled into loops.

## 500 Memoization Template

Memoization is the top-down form of dynamic programming, you solve the problem recursively, but store answers so you never recompute the same state twice. It's the natural bridge between pure recursion and iterative DP.

### What Problem Are We Solving?

Many recursive problems revisit the same subproblems multiple times. For example, Fibonacci recursion:

$$F(n) = F(n - 1) + F(n - 2)$$

recomputes  $F(k)$  many times. Memoization avoids this by caching results after the first computation.

Whenever your recursion tree overlaps, memoization converts exponential time into polynomial time.

### How Does It Work (Plain Language)

1. Define the state: what parameters describe your subproblem?
2. Check if cached: if already solved, return memoized value.
3. Recurse: compute using smaller states.
4. Store result before returning.
5. Return the cached value next time it's needed.

Memoization is ideal for:

- Recursive definitions (Fibonacci, Knapsack, LCS)

- Combinatorial counting with overlapping subproblems
- Tree/graph traversal with repeated subpaths

### Example: Fibonacci with Memoization

$$F(n) = \begin{cases} 1, & n \leq 1, \\ F(n-1) + F(n-2), & \text{otherwise.} \end{cases}$$

We store each  $F(k)$  the first time it's computed.

### Tiny Code (C)

```
#include <stdio.h>

int memo[100];

int fib(int n) {
    if (n <= 1) return 1;
    if (memo[n] != 0) return memo[n];
    return memo[n] = fib(n-1) + fib(n-2);
}

int main() {
    int n = 10;
    printf("Fib(%d) = %d\n", n, fib(n));
}
```

### Python

```
memo = {}
def fib(n):
    if n <= 1:
        return 1
    if n in memo:
        return memo[n]
    memo[n] = fib(n-1) + fib(n-2)
    return memo[n]

print(fib(10))
```

## Why It Matters

- Bridges recursion and iteration: You keep the elegance of recursion with the performance of DP.
- Faster prototypes: Great for quickly building correct solutions.
- Easier to reason: You only define recurrence, not filling order.
- Transition step: Helps derive bottom-up equivalents later.

## A Gentle Proof (Why It Works)

We prove correctness by induction:

- Base case:  $dp[0]$  and  $dp[1]$  defined directly.
- Inductive step: Each call to  $f(n)$  only uses smaller arguments  $f(k)$ , which are correct by the inductive hypothesis.
- Caching: Ensures each  $f(k)$  computed exactly once, guaranteeing  $O(n)$  total calls.

Thus, memoization preserves recursion semantics while achieving optimal time.

## Try It Yourself

1. Write memoized knapsack with signature `solve(i, w)`
2. Memoize subset sum (`solve(i, sum)`)
3. Build LCS recursively with  $(i, j)$  as state
4. Compare memoized and bottom-up versions for runtime

## Test Cases

| Input     | Expected Output |
|-----------|-----------------|
| $fib(0)$  | 1               |
| $fib(1)$  | 1               |
| $fib(5)$  | 8               |
| $fib(10)$ | 89              |

## Complexity

- Time:  $O(n)$  (each state computed once)
- Space:  $O(n)$  recursion + cache

Memoization is the conceptual core of DP, it reveals how subproblems overlap and prepares you for crafting iterative solutions.