Algorithms with TypeScript

A Practical Guide to Data Structures and Algorithms

Zenflow (https://zencoder.ai/zenflow), Claude Opus 4.6

March 13, 2026

Contents

1 Preface **17**
1.1 Who this book is for . . . 17
1.2 Prerequisites . . . 18
1.3 How to use this book . . . 18
1.4 The code . . . 19
1.5 Acknowledgments . . . 19

2 Notation **20**
2.1 Asymptotic notation . . . 20
2.2 Common growth rates . . . 20
2.3 General mathematical notation . . . 21
2.4 Logic and quantifiers . . . 21
2.5 Set notation . . . 22
2.6 Graph notation . . . 22
2.7 Probability notation . . . 23
2.8 Complexity classes . . . 23
2.9 Algorithm and function names . . . 23
2.10 Array and indexing conventions . . . 23
2.11 Formal structures . . . 23
2.12 Code conventions . . . 24

3 Introduction to Algorithms **25**
3.1 What is an algorithm? . . . 25
3.2 Expressing algorithms . . . 26
3.3 Computational procedures that are not algorithms . . . 27
3.4 Why study algorithms? . . . 28
3.5 Introduction to TypeScript . . . 29
3.6 Setting up the development environment . . . 30
3.7 Finding the maximum element . . . 31
3.7.1 The problem . . . 31
3.7.2 A linear scan . . . 31
3.7.3 Correctness . . . 32
3.7.4 Complexity analysis . . . 32
3.8 Finding prime numbers: the Sieve of Eratosthenes . . . 32
3.8.1 The problem . . . 33
3.8.2 A naive approach: trial division . . . 33

3.8.3 The Sieve of Eratosthenes . . . 34
3.8.4 Tracing through an example . . . 34
3.8.5 Why does the sieve work? . . . 35
3.8.6 Complexity analysis . . . 35
3.8.7 Comparing the two approaches . . . 36
3.9 Looking ahead . . . 36
3.10 Exercises . . . 36

4 Analyzing Algorithms **38**
4.1 Why analyze algorithms? . . . 38
4.2 Measuring input size and running time . . . 39
4.3 Asymptotic notation . . . 40
4.3.1 Big-O: upper bounds . . . 40
4.3.2 Big-Omega: lower bounds . . . 41
4.3.3 Big-Theta: tight bounds . . . 41
4.3.4 Summary of notation . . . 42
4.3.5 Common growth rates . . . 42
4.4 Best case, worst case, and average case . . . 43
4.4.1 Insertion sort as a running example . . . 43
4.4.2 Worst-case analysis . . . 43
4.4.3 Best-case analysis . . . 44
4.4.4 Average-case analysis . . . 44
4.4.5 Which case matters? . . . 45
4.5 Amortized analysis . . . 45
4.5.1 The dynamic array example . . . 46
4.5.2 Amortized vs. average case . . . 46
4.6 Recurrence relations . . . 47
4.6.1 Setting up a recurrence . . . 47
4.6.2 Solving recurrences by expansion . . . 47
4.6.3 The recursion tree method . . . 48
4.7 The Master Theorem . . . 49
4.7.1 Limitations of the Master Theorem . . . 50
4.8 Space complexity . . . 50
4.8.1 Time–space trade-offs . . . 51
4.9 Practical considerations . . . 51
4.9.1 Constant factors matter for moderate n . . . 51
4.9.2 Lower-order terms . . . 52
4.9.3 Choosing the right model . . . 52
4.10 Looking ahead . . . 52
4.11 Exercises . . . 53

5 Recursion and Divide-and-Conquer **54**
5.1 Recursion . . . 54
5.1.1 The call stack . . . 55
5.1.2 Common pitfalls . . . 55
5.2 Recursion and mathematical induction . . . 56
5.3 Divide and conquer . . . 56
5.4 Binary search . . . 57

5.4.1 The problem . . . 57
5.4.2 The algorithm . . . 57
5.4.3 Tracing through an example . . . 58
5.4.4 Correctness . . . 58
5.4.5 Complexity analysis . . . 59
5.4.6 Comparison with linear search . . . 59
5.5 Fast exponentiation (exponentiation by squaring) . . . 60
5.5.1 The problem . . . 60
5.5.2 Naive approach . . . 60
5.5.3 Exponentiation by squaring . . . 60
5.5.4 Tracing through an example . . . 61
5.5.5 Correctness . . . 61
5.5.6 Complexity analysis . . . 62
5.6 The Euclidean algorithm for GCD . . . 62
5.6.1 The problem . . . 62
5.6.2 Naive approach . . . 62
5.6.3 The Euclidean algorithm . . . 63
5.6.4 Tracing through an example . . . 63
5.6.5 Correctness . . . 64
5.6.6 Complexity analysis . . . 64
5.7 The closest pair of points . . . 64
5.7.1 The problem . . . 64
5.7.2 Brute-force approach . . . 64
5.7.3 The divide-and-conquer idea . . . 65
5.7.4 The strip optimization . . . 65
5.7.5 Implementation . . . 66
5.7.6 Tracing through an example . . . 68
5.7.7 Correctness . . . 68
5.7.8 Complexity analysis . . . 69
5.7.9 Summary of closest pair . . . 69
5.8 The divide-and-conquer recipe . . . 70
5.9 Looking ahead . . . 70
5.10 Exercises . . . 71

6 Elementary Sorting **72**
6.1 The sorting problem . . . 72
6.2 Stability . . . 73
6.3 In-place sorting . . . 73
6.4 Bubble sort . . . 74
6.4.1 The algorithm . . . 74
6.4.2 Implementation . . . 74
6.4.3 Tracing through an example . . . 75
6.4.4 Correctness . . . 76
6.4.5 Complexity analysis . . . 76
6.4.6 Properties . . . 77
6.5 Selection sort . . . 77
6.5.1 The algorithm . . . 77
6.5.2 Implementation . . . 78

6.5.3 Tracing through an example . . . 78
6.5.4 Correctness . . . 79
6.5.5 Why selection sort is not stable . . . 79
6.5.6 Complexity analysis . . . 79
6.5.7 Properties . . . 80
6.6 Insertion sort . . . 80
6.6.1 The algorithm . . . 80
6.6.2 Implementation . . . 80
6.6.3 Tracing through an example . . . 81
6.6.4 Correctness . . . 82
6.6.5 Complexity analysis . . . 82
6.6.6 Inversions . . . 83
6.6.7 Properties . . . 83
6.7 Comparison of elementary sorts . . . 84
6.8 The comparison-based sorting lower bound . . . 84
6.8.1 The decision tree argument . . . 85
6.8.2 Implications . . . 85
6.9 Looking ahead . . . 86
6.10 Exercises . . . 86

7 Efficient Sorting **88**
7.1 Merge sort . . . 88
7.1.1 The algorithm . . . 88
7.1.2 The merge procedure . . . 89
7.1.3 Tracing through an example . . . 90
7.1.4 Bottom-up implementation . . . 90
7.1.5 Correctness . . . 91
7.1.6 Complexity analysis . . . 91
7.1.7 Properties . . . 92
7.2 Quicksort . . . 92
7.2.1 The partition procedure . . . 93
7.2.2 Tracing through an example . . . 94
7.2.3 Implementation . . . 94
7.2.4 Correctness . . . 95
7.2.5 Complexity analysis . . . 95
7.2.6 Properties . . . 96
7.2.7 Why quicksort is fast in practice . . . 97
7.3 Heapsort . . . 97
7.3.1 The binary heap . . . 97
7.3.2 Heapify . . . 98
7.3.3 Building a heap . . . 98
7.3.4 The heapsort algorithm . . . 99
7.3.5 Tracing through an example . . . 100
7.3.6 Correctness . . . 101
7.3.7 Complexity analysis . . . 101
7.3.8 Properties . . . 101
7.4 Randomized quicksort . . . 102
7.4.1 Motivation . . . 102

7.4.2 The algorithm . . . 102
7.4.3 Expected running time . . . 103
7.4.4 Worst case . . . 104
7.4.5 Properties . . . 104
7.5 Comparison of efficient sorting algorithms . . . 104
7.5.1 Time complexity . . . 105
7.5.2 Space complexity . . . 105
7.5.3 Stability . . . 105
7.5.4 Cache performance . . . 106
7.5.5 Practical recommendations . . . 106
7.6 Chapter summary . . . 106
7.7 Exercises . . . 107

8 Linear-Time Sorting and Selection 108
8.1 Breaking the comparison lower bound . . . 108
8.2 Counting sort . . . 109
8.2.1 The algorithm . . . 109
8.2.2 Implementation . . . 109
8.2.3 Tracing through an example . . . 110
8.2.4 Stability . . . 111
8.2.5 Complexity analysis . . . 111
8.2.6 Properties . . . 111
8.3 Radix sort . . . 112
8.3.1 The algorithm . . . 112
8.3.2 Why least significant digit first? . . . 112
8.3.3 Implementation . . . 112
8.3.4 Tracing through an example . . . 114
8.3.5 Correctness . . . 115
8.3.6 Complexity analysis . . . 115
8.3.7 Properties . . . 116
8.4 Bucket sort . . . 116
8.4.1 The algorithm . . . 116
8.4.2 Implementation . . . 116
8.4.3 Tracing through an example . . . 118
8.4.4 Complexity analysis . . . 118
8.4.5 Properties . . . 119
8.5 Comparison of linear-time sorts . . . 119
8.6 The selection problem . . . 120
8.7 Quickselect . . . 120
8.7.1 The algorithm . . . 120
8.7.2 Implementation . . . 120
8.7.3 Tracing through an example . . . 121
8.7.4 Complexity analysis . . . 122
8.7.5 Properties . . . 122
8.8 Median of medians . . . 122
8.8.1 The algorithm . . . 123
8.8.2 Why groups of 5? . . . 123
8.8.3 Implementation . . . 123

8.8.4 Tracing through an example . . . 125
8.8.5 Complexity analysis . . . 125
8.8.6 Practical considerations . . . 126
8.8.7 Properties . . . 126
8.9 Chapter summary . . . 126
8.10 Exercises . . . 127

9 Arrays, Linked Lists, Stacks, and Queues 129
9.1 Arrays . . . 129
9.1.1 Static arrays in TypeScript . . . 129
9.2 Dynamic arrays . . . 130
9.2.1 The doubling strategy . . . 130
9.2.2 Amortized analysis of append . . . 130
9.2.3 Implementation . . . 131
9.2.4 Complexity summary . . . 133
9.3 Linked lists . . . 133
9.3.1 Singly linked lists . . . 133
9.3.1.1 Implementation . . . 134
9.3.1.2 Tracing through an example . . . 136
9.3.1.3 A limitation of singly linked lists . . . 136
9.3.2 Doubly linked lists . . . 136
9.3.2.1 Implementation . . . 136
9.3.3 Comparing arrays and linked lists . . . 138
9.4 Abstract data types: stacks, queues, and deques . . . 139
9.4.1 Stacks . . . 139
9.4.1.1 Interface . . . 140
9.4.1.2 Implementation . . . 140
9.4.1.3 Applications . . . 141
9.4.1.4 Tracing through an example . . . 141
9.4.2 Queues . . . 141
9.4.2.1 Interface . . . 141
9.4.2.2 Implementation . . . 142
9.4.2.3 Applications . . . 143
9.4.2.4 Tracing through an example . . . 143
9.4.3 Deques . . . 144
9.4.3.1 Implementation . . . 144
9.4.3.2 Using a deque as a stack or queue . . . 145
9.4.3.3 Applications . . . 145
9.5 Complexity comparison . . . 146
9.6 Exercises . . . 146
9.7 Summary . . . 147

10 Hash Tables 148
10.1 The dictionary problem . . . 148
10.1.1 Direct addressing . . . 148
10.2 Hash functions . . . 149
10.2.1 The division method . . . 149
10.2.2 The multiplication method . . . 149

10.2.3 Hashing strings . . . 150
10.2.4 Universal hashing . . . 150
10.3 Collision resolution . . . 151
10.4 Separate chaining . . . 151
10.4.1 How it works . . . 151
10.4.2 Load factor . . . 151
10.4.3 Implementation . . . 151
10.4.4 Dynamic resizing . . . 154
10.4.5 Tracing through an example . . . 154
10.5 Open addressing . . . 155
10.5.1 Linear probing . . . 155
10.5.2 Double hashing . . . 155
10.5.3 Tombstones and lazy deletion . . . 156
10.5.4 Load factor for open addressing . . . 156
10.5.5 Implementation . . . 156
10.5.6 Tracing through linear probing . . . 159
10.6 Chaining vs open addressing . . . 159
10.7 Applications . . . 159
10.7.1 Frequency counting . . . 160
10.7.2 Two-sum problem . . . 160
10.7.3 Anagram detection . . . 160
10.7.4 Deduplication . . . 161
10.8 Complexity summary . . . 161
10.9 Exercises . . . 162
10.10 Summary . . . 162

11 Trees and Binary Search Trees 164
11.1 Tree terminology . . . 164
11.2 Binary trees . . . 165
11.2.1 Representations . . . 165
11.2.2 Properties of binary trees . . . 165
11.3 Tree traversals . . . 166
11.3.1 Inorder traversal (left, root, right) . . . 166
11.3.2 Preorder traversal (root, left, right) . . . 167
11.3.3 Postorder traversal (left, right, root) . . . 167
11.3.4 Level-order traversal (breadth-first) . . . 167
11.3.5 Complexity of traversals . . . 168
11.3.6 Computing height and size . . . 168
11.4 Binary search trees . . . 169
11.4.1 BST node structure . . . 169
11.4.2 Search . . . 170
11.4.3 Insert . . . 170
11.4.4 Tracing through insertions . . . 171
11.4.5 Minimum and maximum . . . 171
11.4.6 Successor and predecessor . . . 172
11.4.7 Tracing successor . . . 173
11.4.8 Delete . . . 173
11.4.9 Tracing deletion . . . 175

11.5 BST performance analysis . . . 176
11.5.1 BST vs hash table . . . 176
11.6 Complexity summary . . . 176
11.7 Exercises . . . 177
11.8 Summary . . . 178

12 Balanced Search Trees 179
12.1 The problem with unbalanced BSTs . . . 179
12.2 Rotations . . . 180
12.3 AVL trees . . . 181
12.3.1 Height bound . . . 181
12.3.2 Node structure . . . 182
12.3.3 Insertion . . . 182
12.3.4 Tracing AVL insertions . . . 184
12.3.5 Deletion . . . 185
12.3.6 AVL complexity . . . 186
12.4 Red-black trees . . . 186
12.4.1 Red-black properties . . . 187
12.4.2 Height bound . . . 187
12.4.3 Node structure and sentinel . . . 187
12.4.4 Insertion . . . 188
12.4.5 Tracing red-black insertion . . . 190
12.4.6 Deletion . . . 191
12.4.7 Verifying red-black properties . . . 191
12.4.8 Red-black complexity . . . 192
12.5 B-trees . . . 193
12.6 Comparison of balanced tree variants . . . 193
12.7 Exercises . . . 194
12.8 Summary . . . 195

13 Heaps and Priority Queues 196
13.1 The priority queue abstraction . . . 196
13.2 Binary heaps . . . 197
13.2.1 Array representation . . . 197
13.2.2 The heap class . . . 198
13.3 Heap operations . . . 199
13.3.1 Sift-up (swim) . . . 199
13.3.2 Sift-down (sink) . . . 200
13.3.3 Insert . . . 201
13.3.4 Extract . . . 201
13.3.5 Decrease-key . . . 202
13.4 Building a heap in $O(n)$ . . . 202
13.4.1 Why is this $O(n)$? . . . 203
13.4.2 Why not sift-up? . . . 203
13.5 The priority queue interface . . . 204
13.6 Min-heap vs. max-heap . . . 205
13.7 Applications . . . 205
13.7.1 Heap sort . . . 205

13.7.2 Running median . . . 206
13.7.3 Event-driven simulation . . . 206
13.7.4 k smallest / largest elements . . . 206
13.8 Complexity summary . . . 206
13.9 Exercises . . . 207
13.10 Summary . . . 207

14 Graphs and Graph Traversal **209**
14.1 What is a graph? . . . 209
14.2 Graph representations . . . 210
14.2.1 Adjacency list . . . 210
14.2.2 Adjacency matrix . . . 212
14.2.3 When to use which? . . . 213
14.3 Breadth-first search (BFS) . . . 214
14.3.1 The algorithm . . . 214
14.3.2 Trace-through . . . 214
14.3.3 Implementation . . . 215
14.3.4 Path reconstruction . . . 216
14.3.5 Complexity . . . 216
14.4 Depth-first search (DFS) . . . 216
14.4.1 The algorithm . . . 217
14.4.2 Trace-through . . . 217
14.4.3 Edge classification . . . 218
14.4.4 Implementation . . . 218
14.4.5 Complexity . . . 220
14.5 Topological sort . . . 220
14.5.1 Applications . . . 220
14.5.2 Kahn's algorithm (BFS-based) . . . 221
14.5.3 DFS-based topological sort . . . 222
14.5.4 Trace-through . . . 223
14.5.5 Complexity . . . 223
14.6 Cycle detection . . . 224
14.6.1 Directed cycle detection . . . 224
14.6.2 Undirected cycle detection . . . 225
14.6.3 Complexity . . . 225
14.7 Connected components . . . 226
14.8 BFS vs. DFS . . . 226
14.9 Exercises . . . 227
14.10 Summary . . . 227

15 Shortest Paths **229**
15.1 The shortest-path problem . . . 229
15.1.1 Negative weights and negative cycles . . . 230
15.1.2 Relaxation . . . 230
15.1.3 Shared result type . . . 230
15.2 Dijkstra's algorithm . . . 231
15.2.1 Intuition . . . 231
15.2.2 Algorithm . . . 231

15.2.3 Implementation . . . 232
15.2.4 Trace-through . . . 233
15.2.5 Complexity . . . 234
15.2.6 Correctness argument . . . 234
15.2.7 When Dijkstra fails . . . 235
15.3 Bellman-Ford algorithm . . . 235
15.3.1 Algorithm . . . 235
15.3.2 Implementation . . . 235
15.3.3 Trace-through . . . 237
15.3.4 Complexity . . . 238
15.3.5 Negative cycle detection . . . 238
15.4 DAG shortest paths . . . 238
15.4.1 Algorithm . . . 238
15.4.2 Implementation . . . 238
15.4.3 Why this works . . . 239
15.4.4 Applications . . . 240
15.4.5 Complexity . . . 240
15.5 Floyd-Warshall algorithm . . . 240
15.5.1 The dynamic programming formulation . . . 240
15.5.2 Space optimization . . . 241
15.5.3 Implementation . . . 241
15.5.4 Negative cycle detection . . . 243
15.5.5 Complexity . . . 243
15.6 Choosing the right algorithm . . . 243
15.7 Exercises . . . 244
15.8 Summary . . . 245

16 Minimum Spanning Trees 246
16.1 The minimum spanning tree problem . . . 246
16.1.1 Where MSTs appear . . . 247
16.2 Theoretical foundation . . . 247
16.2.1 Cuts and light edges . . . 247
16.2.2 The cut property . . . 247
16.2.3 The cycle property . . . 247
16.3 Union-Find: the key data structure for Kruskal's algorithm . . . 248
16.3.1 Union by rank . . . 248
16.3.2 Path compression . . . 249
16.3.3 Combined complexity . . . 249
16.3.4 Implementation . . . 249
16.4 Kruskal's algorithm . . . 251
16.4.1 Algorithm . . . 251
16.4.2 Why it works . . . 251
16.4.3 Trace through an example . . . 251
16.4.4 Implementation . . . 252
16.4.5 Complexity . . . 253
16.5 Prim's algorithm . . . 253
16.5.1 Algorithm . . . 253
16.5.2 Why it works . . . 254

16.5.3 Trace through an example . . . 254
16.5.4 Implementation . . . 254
16.5.5 Complexity . . . 256
16.6 Kruskal's vs. Prim's . . . 256
16.7 Correctness and uniqueness . . . 257
16.7.1 When is the MST unique? . . . 257
16.7.2 Verifying an MST . . . 257
16.8 Exercises . . . 257
16.9 Summary . . . 258

17 Network Flow **259**
17.1 Flow networks . . . 259
17.1.1 Flows . . . 259
17.1.2 Where network flow appears . . . 260
17.2 The Ford-Fulkerson method . . . 260
17.2.1 Residual graphs . . . 261
17.2.2 Augmenting paths . . . 261
17.2.3 The Ford-Fulkerson algorithm . . . 261
17.3 The max-flow min-cut theorem . . . 262
17.3.1 Cuts . . . 262
17.3.2 The theorem . . . 262
17.4 Edmonds-Karp algorithm . . . 263
17.4.1 Why shortest augmenting paths? . . . 263
17.4.2 Pseudocode . . . 263
17.4.3 Trace through an example . . . 264
17.4.4 TypeScript implementation . . . 265
17.4.5 Complexity analysis . . . 267
17.5 Application: maximum bipartite matching . . . 268
17.5.1 The matching problem . . . 268
17.5.2 Reduction to max flow . . . 268
17.5.3 TypeScript implementation . . . 269
17.5.4 Complexity analysis . . . 270
17.5.5 Trace through an example . . . 270
17.6 Beyond Edmonds-Karp . . . 271
17.7 Exercises . . . 272
17.8 Summary . . . 272

18 Dynamic Programming **274**
18.1 When does dynamic programming apply? . . . 274
18.2 Memoization vs tabulation . . . 275
18.2.1 Top-down with memoization . . . 275
18.2.2 Bottom-up with tabulation . . . 275
18.3 A systematic approach to DP . . . 275
18.4 Fibonacci numbers: the introductory example . . . 276
18.4.1 Naive recursion . . . 276
18.4.2 Top-down with memoization . . . 277
18.4.3 Bottom-up with tabulation . . . 277
18.5 Coin change . . . 278

18.5.1 Minimum coins . . . 278
18.5.2 Counting the number of ways . . . 279
18.6 Longest common subsequence . . . 280
18.6.1 The DP formulation . . . 280
18.6.2 Space optimization . . . 282
18.7 Edit distance . . . 282
18.7.1 The DP formulation . . . 282
18.7.2 Recovering the edit script . . . 284
18.8 0/1 Knapsack . . . 284
18.8.1 The DP formulation . . . 284
18.8.2 Space optimization . . . 285
18.9 Matrix chain multiplication . . . 286
18.9.1 The DP formulation . . . 286
18.10 Longest increasing subsequence . . . 288
18.10.1 $O(n^2)$ dynamic programming . . . 288
18.10.2 $O(n \log n)$ patience sorting . . . 289
18.11 Summary of DP problems . . . 290
18.12 Exercises . . . 291
18.13 Chapter summary . . . 292

19 Greedy Algorithms **293**
19.1 The greedy strategy . . . 293
19.2 Proving greedy algorithms correct . . . 294
19.2.1 Greedy stays ahead . . . 294
19.2.2 Exchange argument . . . 294
19.3 Interval scheduling (activity selection) . . . 295
19.3.1 Problem definition . . . 295
19.3.2 Greedy approach . . . 295
19.3.3 Algorithm . . . 295
19.3.4 Correctness proof (greedy stays ahead) . . . 296
19.3.5 Complexity . . . 297
19.3.6 Example . . . 297
19.4 Huffman coding . . . 298
19.4.1 Problem definition . . . 298
19.4.2 Why variable-length codes? . . . 298
19.4.3 Huffman's greedy algorithm . . . 298
19.4.4 Encoding and decoding . . . 300
19.4.5 Correctness proof (exchange argument) . . . 301
19.4.6 Complexity . . . 302
19.4.7 Example . . . 302
19.5 Fractional knapsack . . . 303
19.5.1 Problem definition . . . 303
19.5.2 Why greedy works here (but not for 0/1 knapsack) . . . 304
19.5.3 Algorithm . . . 304
19.5.4 Correctness proof (exchange argument) . . . 306
19.5.5 Complexity . . . 306
19.5.6 Example . . . 306
19.6 When greedy fails . . . 307

19.7 Comparison of algorithms in this chapter . . . 307
19.8 Exercises . . . 308
19.9 Chapter summary . . . 308

20 Disjoint Sets 310
20.1 The disjoint-set problem . . . 310
20.1.1 Where disjoint sets arise . . . 311
20.2 Naive implementations . . . 311
20.2.1 Array-based (quick-find) . . . 311
20.2.2 Linked-list-based (quick-union, unoptimized) . . . 311
20.3 Union by rank . . . 312
20.3.1 Why union by rank helps . . . 312
20.4 Path compression . . . 313
20.4.1 A variant: path halving . . . 313
20.5 Combined complexity: the inverse Ackermann function . . . 314
20.5.1 What is the Ackermann function? . . . 314
20.5.2 The inverse Ackermann function . . . 314
20.5.3 Intuition for the amortized bound . . . 315
20.5.4 Is this optimal? . . . 315
20.6 Implementation . . . 315
20.6.1 Design decisions . . . 317
20.6.2 Complexity summary . . . 317
20.7 Trace through an example . . . 318
20.8 Applications . . . 319
20.8.1 Kruskal's minimum spanning tree . . . 319
20.8.2 Dynamic connectivity . . . 320
20.8.3 Connected components in an image . . . 320
20.8.4 Percolation . . . 320
20.9 Union by rank vs. union by size . . . 321
20.10 Exercises . . . 321
20.11 Summary . . . 322

21 Tries and String Data Structures 323
21.1 The trie (prefix tree) . . . 323
21.1.1 Motivation . . . 323
21.1.2 Structure . . . 324
21.1.3 Operations . . . 324
21.1.4 Complexity analysis . . . 324
21.1.5 Implementation . . . 325
21.1.6 Trace through an example . . . 328
21.2 Compressed tries (radix trees) . . . 329
21.2.1 The problem with standard tries . . . 329
21.2.2 Compressing single-child chains . . . 329
21.2.3 Operations . . . 330
21.2.4 Complexity . . . 331
21.2.5 Implementation . . . 331
21.2.6 Design decisions . . . 334
21.3 Standard trie vs. compressed trie . . . 334

21.4 Applications . . . 335
21.4.1 Autocomplete and search suggestions . . . 335
21.4.2 Spell checking . . . 335
21.4.3 IP routing (longest prefix match) . . . 335
21.4.4 T9 predictive text . . . 336
21.4.5 Bioinformatics . . . 336
21.5 Suffix arrays (conceptual overview) . . . 336
21.6 Exercises . . . 337
21.7 Summary . . . 338

22 String Matching **339**
22.1 The pattern matching problem . . . 339
22.2 Naive string matching . . . 340
22.2.1 Algorithm . . . 340
22.2.2 Trace through an example . . . 340
22.2.3 Implementation . . . 341
22.2.4 Complexity analysis . . . 341
22.3 Rabin-Karp string matching . . . 342
22.3.1 Rolling hash . . . 342
22.3.2 Algorithm . . . 342
22.3.3 Trace through an example . . . 343
22.3.4 Implementation . . . 343
22.3.5 Complexity analysis . . . 345
22.3.6 Why Rabin-Karp matters . . . 345
22.4 Knuth-Morris-Pratt (KMP) . . . 345
22.4.1 The failure function . . . 346
22.4.2 Computing the failure function . . . 346
22.4.3 The KMP search algorithm . . . 347
22.4.4 Step-by-step trace . . . 347
22.4.5 Implementation . . . 349
22.4.6 Complexity analysis . . . 350
22.4.7 Why KMP is important . . . 351
22.5 Comparison and practical considerations . . . 351
22.5.1 Beyond this chapter . . . 352
22.6 Exercises . . . 352
22.7 Summary . . . 353

23 Complexity Classes and NP-Completeness **354**
23.1 Decision problems and languages . . . 354
23.2 The class P . . . 355
23.3 The class NP . . . 355
23.3.1 Examples . . . 356
23.3.2 P $\subseteq$ NP . . . 356
23.4 The class co-NP . . . 356
23.5 The P versus NP question . . . 357
23.6 Polynomial-time reductions . . . 357
23.7 NP-completeness . . . 357
23.7.1 The Cook-Levin theorem . . . 357

23.8 Classic NP-complete problems . . . 358
23.8.1 Boolean satisfiability . . . 358
23.8.2 Graph problems . . . 358
23.8.3 Numeric problems . . . 359
23.8.4 Optimization problems (decision versions) . . . 359
23.9 Proving NP-completeness by reduction: a worked example . . . 359
23.9.1 Step 1: VERTEX COVER is in NP . . . 359
23.9.2 Step 2: 3-SAT $\leq_P$ VERTEX COVER . . . 359
23.10 The reduction landscape . . . 360
23.11 Brute-force illustrations . . . 361
23.11.1 Subset sum (brute force) . . . 361
23.11.2 Traveling salesman (brute force) . . . 363
23.12 Coping with NP-hardness . . . 365
23.12.1 1. Approximation algorithms . . . 365
23.12.2 2. Exact algorithms for special cases . . . 365
23.12.3 3. Pseudo-polynomial algorithms . . . 366
23.12.4 4. Heuristics and metaheuristics . . . 366
23.12.5 5. Randomized algorithms . . . 366
23.13 Summary of complexity classes . . . 366
23.14 Exercises . . . 367
23.15 Chapter summary . . . 368

24 Approximation Algorithms 369
24.1 When exact solutions are infeasible . . . 369
24.2 Approximation ratios . . . 370
24.3 Vertex cover: 2-approximation . . . 371
24.3.1 Problem definition . . . 371
24.3.2 The algorithm . . . 371
24.3.3 Pseudocode . . . 372
24.3.4 Proof of the 2-approximation . . . 372
24.3.5 TypeScript implementation . . . 372
24.3.6 Worked example . . . 373
24.3.7 Tightness of the bound . . . 374
24.4 Greedy set cover: $O(\log n)$-approximation . . . 374
24.4.1 Problem definition . . . 374
24.4.2 The greedy algorithm . . . 374
24.4.3 Proof of the $O(\log n)$-approximation . . . 374
24.4.4 TypeScript implementation . . . 375
24.4.5 Worked example . . . 376
24.4.6 Optimality of the greedy bound . . . 377
24.5 Metric TSP: 2-approximation via MST . . . 377
24.5.1 Problem definition . . . 377
24.5.2 The MST-based algorithm . . . 378
24.5.3 Why this works: the shortcutting argument . . . 378
24.5.4 Proof of the 2-approximation . . . 378
24.5.5 TypeScript implementation . . . 378
24.5.6 Worked example . . . 380
24.5.7 Christofides' algorithm: a better bound . . . 381

24.6 Comparison of approximation algorithms 382
24.7 Beyond the algorithms in this chapter 382
24.8 Exercises 383
24.9 Chapter summary 384

25 Bibliography **385**
25.1 Textbooks 385
25.2 Online resources 386
25.3 Note on authorship and licensing 386

Chapter 1

Preface

Download this book as PDF

Beta: This book is currently in beta and is still under active review. It may contain errors or incomplete sections. Report errors or issues — contributions are welcome via the GitHub repository.

Built with Zenflow by Zencoder using Claude Code and Claude Opus 4.6 by Anthropic

---

This book grew out of a simple observation: most software engineers use algorithms and data structures every day, yet many feel uncertain about the fundamentals. They may use a hash map or call a sorting function without fully understanding the guarantees those abstractions provide, or they may struggle when a problem requires designing a new algorithm from scratch. At the same time, Computer Science students often encounter algorithms in a highly theoretical setting that can feel disconnected from the code they write in practice.

Algorithms with TypeScript bridges that gap. It presents the core algorithms and data structures from a typical undergraduate algorithms curriculum — roughly equivalent to MIT's 6.006 and 6.046 — but uses TypeScript as the language of expression. Every algorithm discussed in the text is implemented, tested, and available in the accompanying repository. The implementations are not pseudocode translated into TypeScript; they are idiomatic, type-safe, and tested with a modern toolchain.

1.1 Who this book is for

This book is written for two audiences:

- **Software engineers** who want to solidify their understanding of algorithms and data structures. Perhaps you learned this material years ago and want a refresher, or perhaps you are self-taught and want to fill in the gaps. Either way, seeing algorithms in a language you likely use at work — TypeScript — makes the material immediately applicable.
- **Computer Science students** who are taking (or preparing to take) an algorithms course. The book follows a standard curricular sequence and includes exercises at the end of every chapter. The TypeScript implementations let you run, modify, and experiment with every algorithm.

1.2 Prerequisites

The book assumes you can read and write basic TypeScript or JavaScript. You should be comfortable with functions, loops, conditionals, arrays, and objects. No prior knowledge of algorithms or data structures is required — we build everything from the ground up, starting with the definition of an algorithm in Chapter 1.

Some chapters use mathematical notation, particularly for complexity analysis. Chapter 2 introduces asymptotic notation (O, Ω, Θ), and the Notation section that follows this preface summarizes all conventions used in the book. A comfort with basic algebra and mathematical reasoning is helpful but not strictly required; we explain each concept as it arises.

1.3 How to use this book

The book is organized into six parts:

- **Part I: Foundations** (Chapters 1–3) introduces the notion of an algorithm, the mathematical tools for analyzing algorithms, and recursion with divide-and-conquer.
- **Part II: Sorting and Selection** (Chapters 4–6) covers the classical sorting algorithms, from elementary $O(n^2)$ methods through $O(n \log n)$ comparison sorts to linear-time non-comparison sorts and selection algorithms.
- **Part III: Data Structures** (Chapters 7–11) presents the fundamental data structures: arrays, linked lists, stacks, queues, hash tables, trees, balanced search trees, heaps, and priority queues.
- **Part IV: Graph Algorithms** (Chapters 12–15) covers graph representations, traversal, shortest paths, minimum spanning trees, and network flow.
- **Part V: Algorithm Design Techniques** (Chapters 16–17) explores dynamic programming and greedy algorithms as general problem-solving strategies.

- **Part VI: Advanced Topics** (Chapters 18–22) covers disjoint sets, tries, string matching, computational complexity, and approximation algorithms.

The parts are designed to be read in order, as later chapters build on concepts and data structures introduced in earlier ones. Within each part, the chapters are largely self-contained — if you are comfortable with the prerequisites, you can often read individual chapters independently.

Each chapter follows a consistent structure: a motivating introduction, formal definitions, detailed algorithm descriptions with step-by-step traces, TypeScript implementations with code snippets, complexity analysis, and exercises. The exercises range from straightforward checks of understanding to more challenging problems that extend the material.

1.4 The code

All implementations live in the `src/` directory of the repository, organized by chapter. Tests are in the `tests/` directory with a parallel structure. To run the full test suite:

```
npm install
npm test
```

The code is written in TypeScript 5 with strict mode enabled, uses ES modules, and is tested with Vitest. See the project README for detailed setup instructions.

We encourage you to read the code alongside the text. The implementations are designed to be clear and readable rather than maximally optimized. Where there is a tension between clarity and performance, we choose clarity and discuss the performance implications in the text.

1.5 Acknowledgments

This book draws inspiration from several excellent texts, most notably Cormen, Leiserson, Rivest, and Stein's *Introduction to Algorithms* (CLRS), Sedgewick and Wayne's *Algorithms*, Niklaus Wirth's *Algorithms + Data Structures = Programs*, and Kleinberg and Tardos's *Algorithm Design*. The MIT OpenCourseWare materials for 6.006 and 6.046 were invaluable in shaping the curriculum. Full references are in the Bibliography.

Chapter 2

Notation

This section summarizes the mathematical and typographical conventions used throughout the book. It is intended as a reference; each symbol is introduced and explained in context when it first appears.

2.1 Asymptotic notation

Symbol	Meaning
$O(g(n))$	Asymptotic upper bound: $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$ (Definition 2.2)
$\Omega(g(n))$	Asymptotic lower bound: $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$ (Definition 2.3)
$\Theta(g(n))$	Tight bound: $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ (Definition 2.4)
$o(g(n))$	Strict upper bound: $f(n)/g(n) \to 0$ as $n \to \infty$
$\omega(g(n))$	Strict lower bound: $g(n)/f(n) \to 0$ as $n \to \infty$

The asymptotic families correspond loosely to the comparison operators: O to $\leq$, Ω to $\geq$, Θ to $=$, o to $<$, and ω to $>$.

2.2 Common growth rates

Growth rate	Name	Example algorithm
$O(1)$	Constant	Hash table lookup (expected)
$O(\log n)$	Logarithmic	Binary search
$O(n)$	Linear	Finding the maximum
$O(n \log n)$	Linearithmic	Merge sort, heap sort
$O(n^2)$	Quadratic	Insertion sort (worst case)
$O(n^3)$	Cubic	Floyd-Warshall
$O(2^n)$	Exponential	Subset sum (brute force)
$O(n!)$	Factorial	TSP (brute force)

2.3 General mathematical notation

Symbol	Meaning
n	Input size (unless otherwise stated)
$T(n)$	Running time as a function of input size
$\lfloor x \rfloor$	Floor: largest integer $\leq x$
$\lceil x \rceil$	Ceiling: smallest integer $\geq x$
$\log n$	Logarithm base 2 (unless base is stated explicitly)
$\log_b n$	Logarithm base b
$\ln n$	Natural logarithm (base e)
H_n	n-th harmonic number: $H_n = \sum_{i=1}^{n} 1/i \approx \ln n$
$n!$	Factorial: $n! = n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1$
$\binom{n}{k}$	Binomial coefficient: $n!/(k!(n-k)!)$
$x \bmod m$	Remainder when x is divided by m
$\sum_{i=a}^{b} f(i)$	Summation of $f(i)$ for i from a to b
$\prod_{i=a}^{b} f(i)$	Product of $f(i)$ for i from a to b
∞	Infinity
$\approx$	Approximately equal

2.4 Logic and quantifiers

Symbol	Meaning
$\Longrightarrow$	Implies (if … then)
$\Longleftrightarrow$	If and only if
$\forall$	For all

Symbol	Meaning
$\exists$	There exists

2.5 Set notation

Symbol	Meaning
$\{a, b, c\}$	Set containing elements a, b, c
$x \in S$	x is a member of set S
$x \notin S$	x is not a member of set S
$A \subseteq B$	A is a subset of B (possibly equal)
$A \subset B$	A is a proper subset of B
$A \cup B$	Union of A and B
$A \cap B$	Intersection of A and B
$A \setminus B$	Set difference: elements in A but not in B
$\|S\|$	Cardinality (number of elements) of set S
$\emptyset$	Empty set
$\mathbb{R}$	Set of real numbers
$\{0, 1\}^*$	Set of all binary strings

2.6 Graph notation

Symbol	Meaning
$G = (V, E)$	Graph G with vertex set V and edge set E
$\|V\|$	Number of vertices
$\|E\|$	Number of edges
(u, v)	Edge from vertex u to vertex v
$w(u, v)$	Weight of edge (u, v)
$w : E \to \mathbb{R}$	Weight function mapping edges to real numbers
$\delta(s, v)$	Shortest-path weight from s to v
$d(u, v)$	Distance between vertices u and v
$c(u, v)$	Capacity of edge (u, v) (in flow networks)
$f(u, v)$	Flow on edge (u, v)
$w(T)$	Total weight of tree T
Adj[v]	Adjacency list of vertex v

Vertices are typically denoted by lowercase letters: u, v, s (source), t (sink). We use $s \rightsquigarrow v$ to denote a path from s to v.

2.7 Probability notation

Symbol	Meaning
$\Pr[A]$	Probability of event A
$\mathbb{E}[X]$	Expected value of random variable X

2.8 Complexity classes

Complexity classes are set in bold: **P**, **NP**, **co-NP**. NP-complete problems are written in small capitals in running text (e.g., SUBSET SUM, SAT, HAMILTONIAN CYCLE).

2.9 Algorithm and function names

In mathematical expressions, algorithm names are typeset in roman (upright) text to distinguish them from variables:

- $\mathrm{parent}(i)$, $\mathrm{left}(i)$, $\mathrm{right}(i)$ for heap index calculations
- $\mathrm{Relax}(u, v, w)$ for shortest-path edge relaxation
- $\mathrm{OPT}(I)$ for the optimal solution value on instance I

Running-time recurrences use $T(n)$. Fibonacci numbers are $F(n)$.

2.10 Array and indexing conventions

All TypeScript implementations use **0-based indexing**: the first element of an array `arr` is `arr[0]`, and an array of n elements has indices $0, 1, \ldots, n-1$.

In mathematical discussion, array ranges are written as $A[\ell..r)$ to denote the subarray from index ℓ (inclusive) to r (exclusive). In heap formulas:

$$\mathrm{parent}(i) = \lfloor (i-1)/2 \rfloor, \quad \mathrm{left}(i) = 2i + 1, \quad \mathrm{right}(i) = 2i + 2$$

2.11 Formal structures

Formal definitions, theorems, and lemmas are set in blockquotes with a label:

Definition X.Y — Title

Statement of the definition.

Proofs end with the symbol □. Examples are labeled **Example X.Y** and numbered within each chapter.

2.12 Code conventions

- All code is TypeScript with strict mode and ES module syntax.
- Generic type parameters (e.g., `T`, `K`, `V`) follow standard TypeScript conventions.
- The shared type `Comparator<T>` is `(a: T, b: T) => number`, returning negative if $a < b$, zero if $a = b$, and positive if $a > b$.
- Code snippets in chapters match the tested implementations in the `src/` directory.

Chapter 3

Introduction to Algorithms

In this chapter we discuss what an algorithm is, how algorithms can be expressed, and why studying them matters. We introduce TypeScript as the language used throughout the book, walk through setting up a development environment, and examine our first two algorithms in detail: finding the maximum of an array and the Sieve of Eratosthenes.

3.1 What is an algorithm?

Let us start with a discussion of what an algorithm is. Intuitively the notion is more or less clear: we are talking about some formal way to describe a computational procedure. According to the Merriam-Webster dictionary, an algorithm is "a set of steps that are followed in order to solve a mathematical problem or to complete a computer process".

Still, this is probably not formal enough. How do we choose the next step from the set of steps? Should the procedure stop eventually? What is the result of executing an algorithm? Many formal definitions of what constitutes an algorithm can be given; however, at this point in the book, without introducing abstract models of computation, we will use the following working definition.

Definition 1.1 — Algorithm

A set of computational steps that specifies a formal computational procedure and has the following properties:

1. After each step is completed, the next step is unambiguously defined, or the algorithm stops its execution if there are no more steps left.

2. It is defined on a set of inputs and for each valid input it stops after a finite number of steps.
3. When it stops it produces a result, which we call its **output**.
4. Its steps and their order of execution can be formally and unambiguously specified using some language or notation.

These four properties capture the essence of what makes a procedure an algorithm. Let us look at each one briefly:

- **Determinism** (property 1): at every point during execution, there is exactly one thing to do next, or the algorithm is done. There is no ambiguity or choice involved.
- **Termination** (property 2): for every valid input, the algorithm eventually finishes. It does not run forever.
- **Output** (property 3): when the algorithm finishes, it produces a well-defined result.
- **Formal specification** (property 4): the algorithm can be written down precisely enough that it could, in principle, be carried out mechanically.

3.2 Expressing algorithms

Algorithms can be expressed in a variety of ways. We can even specify the execution steps using ordinary human language. Let us provide a few simple examples. A trivial first example is multiplying two numbers.

Example 1.1: Integer multiplication.

Steps:

1. *Given two integer numbers, multiply them and return the result.*

All the properties from Definition 1.1 are satisfied. There is only one step; after this step the algorithm stops; the step is formally specified; all pairs of integer numbers are valid inputs; and a valid result will be produced for each of them. If we denote the algorithm for multiplication as mult, then, for example,

$$\text{mult}(2, 5) = 10$$

and we can specify the algorithm more concisely as

$$\text{mult}(x, y) = x \times y$$

So far, while talking about algorithms, we have encountered no TypeScript or any other programming language notation. This is quite intentional: the notion of an algorithm is mathematical and abstract. Of course we can express any algorithm using TypeScript, but that will be just one of the possible formal representations — in this case, one that is also executable by a computer.

A careful reader might be puzzled by our confidence. How can we assert that *any* algorithm can be expressed using TypeScript? Can this claim be proven, given our definition? Is TypeScript powerful enough to express every possible algorithm? It turns out that it is, but we will leave this statement without proof until the end of the book, where we discuss abstract models of computation and give a more rigorous definition of an algorithm (see Chapter 21).

Let us look again at Definition 1.1. It states that we should be able to specify the computational procedure formally. It is now clear why we require this property: given a formal language such as TypeScript, we can specify the algorithm of interest and execute the specification on a machine such as a laptop or smartphone. For the multiplication algorithm we can write:

```
function mult(x: number, y: number): number {
  return x * y;
}
```

The TypeScript specification is more concise and unambiguous than the natural-language version. Throughout the book we will primarily use TypeScript, but keep in mind that the algorithms we discuss can be expressed in other formal notations as well. Many Computer Science textbooks go as far as inventing their own pseudocode to avoid being tied to a particular programming language. We will not go that far and will happily use TypeScript — hence the name of the book, *Algorithms with TypeScript*.

3.3 Computational procedures that are not algorithms

Can we write a computational procedure that is *not* an algorithm? Yes. Consider the following TypeScript function:

```
function getMaximumNumber(): number {
  let x = 0;
  while (true) {
    x++;
  }
```

```
  return x;
}
```

This function never terminates: the `while (true)` loop runs forever, so the `return` statement is never reached. Property 2 of Definition 1.1 is violated — the procedure does not stop after a finite number of steps. This is therefore not an algorithm.

Another example of a non-algorithm is a division function defined on all pairs of numbers:

```
function divide(x: number, y: number): number {
  if (y === 0) {
    throw new Error('Cannot divide by zero');
  }
  return x / y;
}
```

This is not an algorithm according to our definition because the result is not defined for all inputs — when $y = 0$ the procedure throws an error instead of producing an output (property 3 is violated). However, it is easy to fix this:

```
function divide(x: number, y: number): number {
  return y === 0 ? Infinity : x / y;
}
```

In fact, in JavaScript (and TypeScript), dividing by zero returns `Infinity` by default, so we could simply write:

```
function divide(x: number, y: number): number {
  return x / y;
}
```

This *is* an algorithm — but only because of JavaScript's particular treatment of division by zero.

From these examples we see that not every computational procedure that can be formally expressed is an algorithm. The properties in Definition 1.1 are genuine constraints.

3.4 Why study algorithms?

Before we proceed to our first nontrivial examples, let's briefly discuss why studying algorithms is worthwhile.

Correctness. Real-world software often needs to solve well-defined computational problems: sort a list, find the shortest route, compress data, search a database. An algorithm gives us a *proven* solution to such a problem. Understanding the classic algorithms means you can recognize when a problem you face has already been solved — and solved well.

Efficiency. Two algorithms that solve the same problem can differ enormously in how long they take or how much memory they use. Later in this book we will see sorting algorithms that take time proportional to n^2 (where n is the number of elements) and others that take time proportional to $n \log n$. For a million elements, that is the difference between a trillion operations and roughly twenty million — a factor of 50,000. Choosing the right algorithm can be the difference between a program that finishes in seconds and one that takes hours.

Foundation for deeper topics. Algorithms and data structures form the backbone of computer science. Topics like databases, compilers, operating systems, machine learning, and cryptography all build on the ideas we will develop in this book.

Problem-solving skills. Even when you are not directly implementing a classic algorithm, the techniques you learn — divide and conquer, dynamic programming, greedy strategies, graph modeling — give you a powerful toolkit for approaching new problems.

3.5 Introduction to TypeScript

Throughout this book we use TypeScript as our implementation language. TypeScript is a statically typed superset of JavaScript: every valid JavaScript program is also a valid TypeScript program, but TypeScript adds optional type annotations that are checked at compile time.

We chose TypeScript for several reasons:

- **Readability.** TypeScript syntax is familiar to anyone who has worked with JavaScript, Java, C#, or similar C-family languages. Type annotations make function signatures self-documenting.
- **Type safety.** Generic types let us write algorithms that work with any element type while the compiler catches type errors before we run the code.
- **Ubiquity.** TypeScript runs anywhere JavaScript runs: in the browser, on the server (Node.js), and in countless tools. There is no special runtime to install beyond Node.js.
- **Modern features.** Destructuring, iterators, generator functions, and first-class functions make algorithm implementations concise and expressive.

Here is a small example that illustrates some features we will use frequently:

```
// A generic function that returns the first element of a non-empty array
function first<T>(arr: T[]): T {
  if (arr.length === 0) {
    throw new Error('Array must not be empty');
  }
  return arr[0];
}

const name: string = first(['Alice', 'Bob', 'Charlie']); // 'Alice'
const value: number = first([42, 17, 8]);                // 42
```

The <T> syntax introduces a *type parameter*: the function works with arrays of any element type, and the compiler ensures that the return type matches the array's element type. We will use generics extensively when implementing data structures and sorting algorithms.

3.6 Setting up the development environment

To follow along with the code in this book, you will need:

1. **Node.js** (version 18 or later): download from https://nodejs.org or use a version manager such as `nvm`.
2. **A text editor** with TypeScript support. Visual Studio Code works particularly well, but any modern editor will do.

Once Node.js is installed, clone the book's repository and install the dependencies:

```
git clone https://github.com/antivanov/Algorithms-with-JavaScript.git
cd Algorithms-with-JavaScript
npm install
```

The project uses the following tools, all installed automatically by `npm install`:

Tool	Purpose
TypeScript	Static type checking and compilation
Vitest	Fast test runner with native TypeScript support
ESLint	Code quality and consistency checking
Prettier	Automatic code formatting

Useful commands:

```
npm test            # Run all tests
npm run test:watch  # Re-run tests on file changes
npm run typecheck   # Check types without emitting files
npm run lint        # Run the linter
```

Every algorithm in this book has a corresponding test suite. We encourage you to run the tests, read them, and experiment by modifying the implementations.

3.7 Finding the maximum element

Now that we are finished with definitions and setup, let's look at a few more interesting algorithms. The first problem is simple: given an array of numbers, find the largest one.

3.7.1 The problem

Input: An array A of n numbers $A[0], A[1], \ldots, A[n-1]$.

Output: The maximum value in A, or `undefined` if A is empty.

3.7.2 A linear scan

The most natural approach is to scan through the array from left to right, keeping track of the largest value seen so far:

1. Set result to `undefined`.
2. For each element e in A:
   - If result is `undefined` or $e >$ result, set result $= e$.
3. Return result.

Here is the TypeScript implementation:

```ts
export function max(elements: number[]): number | undefined {
  let result: number | undefined;

  for (const element of elements) {
    if (result === undefined || element > result) {
      result = element;
    }
  }
  return result;
}
```

Let us trace through an example. Suppose $A = [2, 1, 4, 2, 3]$:

Step	`element`	`result` before	Comparison	`result` after
1	2	`undefined`	`undefined` → update	2
2	1	2	1 > 2? No	2
3	4	2	4 > 2? Yes	4
4	2	4	2 > 4? No	4
5	3	4	3 > 4? No	4

The function returns 4, which is indeed the maximum.

3.7.3 Correctness

We can argue correctness using a *loop invariant*: at the start of each iteration, `result` holds the maximum of all elements examined so far (or `undefined` if none have been examined).

- **Initialization:** Before the first iteration, no elements have been examined and `result` is `undefined`. The invariant holds trivially.
- **Maintenance:** Suppose the invariant holds at the start of an iteration. If the current `element` is greater than `result` (or `result` is `undefined`), we update `result` to `element`. Otherwise `result` already holds the maximum. In either case, after the iteration `result` is the maximum of all elements seen so far.
- **Termination:** The loop ends when all elements have been examined. By the invariant, `result` holds the maximum of the entire array.

3.7.4 Complexity analysis

The function performs one comparison per element and visits each element exactly once.

- **Time complexity:** $O(n)$, where n is the length of the array.
- **Space complexity:** $O(1)$ — we use only a single variable `result` beyond the input.

Can we do better than $O(n)$? No. Any algorithm that finds the maximum must examine every element at least once: if it skipped an element, that element could have been the maximum. Therefore $O(n)$ is optimal for this problem.

3.8 Finding prime numbers: the Sieve of Eratosthenes

Our second algorithm is more substantial and has a rich history dating back over two thousand years. The goal is to find all prime numbers up to a given number

n.

3.8.1 The problem

Input: A positive integer n.

Output: A list of all prime numbers p with $2 \leq p \leq n$.

Recall that a prime number is an integer greater than 1 whose only positive divisors are 1 and itself. The first few primes are 2, 3, 5, 7, 11, 13, 17, 19, 23, ...

3.8.2 A naive approach: trial division

The most straightforward method is to test each number from 2 to n for primality by checking whether it has any divisors other than 1 and itself:

```
export function primesUpToSlow(number: number): number[] {
  const primes: number[] = [];

  for (let current = 2; current <= number; current++) {
    if (isPrime(current)) {
      primes.push(current);
    }
  }
  return primes;
}

function isPrime(number: number): boolean {
  for (let i = 2; i < number; i++) {
    if (number % i === 0) {
      return false;
    }
  }
  return true;
}
```

For each candidate number k, the `isPrime` function tests all potential divisors from 2 up to $k-1$. If any of them divides k evenly, k is not prime.

This works, but it is slow. For each of the $n-1$ candidates, we may test up to $k-2$ divisors. In the worst case (when k is prime), the `isPrime` check does $O(k)$ work. Summing over all candidates gives roughly $O(n^2)$ time. (We could improve `isPrime` by only testing up to $\sqrt{k}$, which brings the total to approximately $O(n\sqrt{n})$, but there is a fundamentally better approach.)

3.8.3 The Sieve of Eratosthenes

The Sieve of Eratosthenes, attributed to the ancient Greek mathematician Eratosthenes of Cyrene (c. 276–194 BC), takes a different approach. Instead of testing each number individually, it starts by assuming all numbers are prime and then systematically eliminates the ones that are not:

1. Create a boolean array `isPrime[2..n]`, initially all `true`.
2. For each number p starting from 2:
   - If `isPrime[p]` is `true`, then p is prime. Mark all multiples of p (starting from $2p$) as `false`.
3. Collect all indices that remain `true`.

Here is the TypeScript implementation:

```
export function primesUpTo(number: number): number[] {
  const isPrimeNumber: boolean[] = [];
  const primes: number[] = [];
  let current = 2;

  for (let i = 2; i <= number; i++) {
    isPrimeNumber[i] = true;
  }

  while (current <= number) {
    if (isPrimeNumber[current]) {
      primes.push(current);
      for (let j = 2 * current; j <= number; j += current) {
        isPrimeNumber[j] = false;
      }
    }
    current++;
  }
  return primes;
}
```

3.8.4 Tracing through an example

Let us trace the sieve for $n = 20$. We start with all numbers from 2 to 20 marked as potentially prime:

```
 2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
 T  T  T  T  T  T  T  T  T  T  T  T  T  T  T  T  T  T  T
```

$p = 2$: 2 is prime. Cross out multiples of 2: 4, 6, 8, 10, 12, 14, 16, 18, 20.

```
 2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
 T  T  F  T  F  T  F  T  F  T  F  T  F  T  F  T  F  T  F
```

$p = 3$**:** 3 is still marked `true`, so it is prime. Cross out multiples of 3: 6, 9, 12, 15, 18 (some are already crossed out).

```
 2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
 T  T  F  T  F  T  F  F  F  T  F  T  F  F  F  T  F  T  F
```

$p = 4$**:** 4 is marked `false` (not prime). Skip it.

$p = 5$**:** 5 is marked `true`, so it is prime. Cross out multiples of 5: 10, 15, 20 (all already crossed out).

For $p \geq 5$, all multiples up to 20 have already been marked, so no further changes occur. The numbers that remain `true` are:

$$2, 3, 5, 7, 11, 13, 17, 19$$

These are exactly the primes up to 20.

3.8.5 Why does the sieve work?

The key insight is: if a number k is composite (not prime), then $k = a \times b$ for some integers a, b with $2 \leq a \leq b < k$. The smallest such factor a is itself prime (otherwise it could be factored further). When the sieve processes $p = a$, it marks $k = a \times b$ as composite. Therefore, every composite number gets marked `false` by the time the sieve finishes.

Conversely, if a number k is prime, no smaller prime p divides it, so k is never marked `false`. The sieve correctly identifies exactly the prime numbers.

3.8.6 Complexity analysis

How much work does the sieve do? For each prime $p \leq n$, it crosses out at most n/p multiples. The total work is proportional to:

$$\sum_{p \text{ prime}, p \leq n} \frac{n}{p} = n \sum_{p \text{ prime}, p \leq n} \frac{1}{p}$$

A classical result in number theory states that the sum of the reciprocals of the primes up to n grows as $\ln(\ln(n))$. Therefore:

- **Time complexity:** $O(n \log \log n)$.
- **Space complexity:** $O(n)$ for the boolean array.

Compare this with the naive trial-division approach at $O(n\sqrt{n})$. For n = 1,000,000:

Algorithm	Approximate operations
Trial division	$10^6 \times 10^3 = 10^9$
Sieve of Eratosthenes	$10^6 \times \log\log(10^6) \approx 10^6 \times 2.9 \approx 3 \times 10^6$

The sieve is roughly 300 times faster — an enormous difference for large inputs.

3.8.7 Comparing the two approaches

This is our first encounter with a recurring theme in this book: different algorithms for the same problem can have vastly different performance characteristics. The naive approach is simple and easy to understand, but the sieve achieves dramatically better performance by exploiting the structure of the problem.

Throughout the book, we will develop the tools to analyze these differences precisely. In Chapter 2 we formalize the notion of time complexity using asymptotic notation (O, Ω, Θ), which gives us a language for comparing algorithms independently of the specific hardware they run on.

3.9 Looking ahead

In this chapter we defined what an algorithm is, introduced TypeScript as our implementation language, and studied two concrete algorithms. We saw that:

- An algorithm is a well-defined computational procedure that terminates on all valid inputs and produces a result.
- Algorithms can be expressed in many notations; we use TypeScript because it combines readability, type safety, and executability.
- Even for simple problems, the choice of algorithm can dramatically affect performance: the Sieve of Eratosthenes outperforms trial division by orders of magnitude.

In the next chapter, we develop the mathematical framework — asymptotic notation and complexity analysis — that lets us reason precisely about algorithm efficiency. These tools will be essential throughout the rest of the book.

3.10 Exercises

Exercise 1.1. Write a function `min(elements: number[]): number | undefined` that returns the minimum element of an array, analogous to the `max` function.

What is its time complexity?

Exercise 1.2. The `isPrime` function in the trial-division approach tests divisors from 2 all the way up to $k - 1$. Explain why it suffices to test only up to $\lfloor\sqrt{k}\rfloor$. Modify the function accordingly and analyze the improved time complexity for finding all primes up to n.

Exercise 1.3. The Sieve of Eratosthenes as presented starts crossing out multiples of p from $2p$. Show that it is sufficient to start from p^2 instead. Why does this not change the asymptotic time complexity?

Exercise 1.4. A *perfect number* is a positive integer that equals the sum of its proper divisors (e.g., $6 = 1 + 2 + 3$). Write a function `isPerfect(n: number): boolean` and use it to find all perfect numbers up to 10,000. What is the time complexity of your approach?

Exercise 1.5. Consider the following function:

```
function mystery(n: number): number {
  if (n <= 1) return n;
  return mystery(n - 1) + mystery(n - 2);
}
```

Does this function define an algorithm according to Definition 1.1? What does it compute? Try calling it with $n = 5$, $n = 10$, and $n = 40$. What do you observe about the running time? (We will revisit this function in Chapter 16 on dynamic programming.)

Chapter 4

Analyzing Algorithms

In Chapter 1 we saw that two algorithms for the same problem — trial division versus the Sieve of Eratosthenes — can differ enormously in performance. In this chapter we develop the mathematical framework for making such comparisons precise. We introduce asymptotic notation, which lets us describe how an algorithm's resource usage grows with input size, and we study several techniques for analyzing running time: best-, worst-, and average-case analysis, amortized analysis, and recurrence relations.

4.1 Why analyze algorithms?

Suppose you have two sorting algorithms, A and B, and you want to know which one is faster. The most direct approach is to run both on the same input and measure the wall-clock time. This is called *benchmarking*, and it has an important place in software engineering. However, benchmarking has limitations:

- **Hardware dependence.** Algorithm A might be faster on your laptop but slower on a different machine with a different CPU, cache hierarchy, or memory bandwidth.
- **Input dependence.** Algorithm A might be faster on the particular test data you chose, but slower on inputs that arise in practice.
- **Implementation effects.** A clever implementation of a theoretically slower algorithm can outperform a naive implementation of a theoretically faster one.

What we want is a way to compare algorithms *independently* of these factors — a way to reason about the inherent efficiency of an algorithm rather than the efficiency of a particular implementation on a particular machine with a particular input. This is what asymptotic analysis provides.

The idea is to count the number of "basic operations" an algorithm performs as a function of the input size n, and then focus on how that function grows as n becomes large. We ignore constant factors (which depend on the hardware and implementation) and lower-order terms (which become negligible for large n). The result is a concise characterization of an algorithm's scalability.

4.2 Measuring input size and running time

Before we can analyze an algorithm, we need to agree on two things: what counts as the *input size*, and what counts as a *basic operation*.

Input size is usually the most natural measure of how much data the algorithm must process:

- For an array of numbers, the input size is the number of elements n.
- For a graph, the input size is often specified as a pair (V, E) — the number of vertices and edges.
- For a number-theoretic algorithm like the Sieve of Eratosthenes, the input size is the number n itself.

Basic operations are the elementary steps we count. Common choices include comparisons, arithmetic operations, assignments, or array accesses. The specific choice rarely matters for asymptotic analysis, because changing which operation we count changes the total by at most a constant factor.

Definition 2.1 — Running time

The **running time** of an algorithm on a given input is the number of basic operations it performs when executed on that input.

We are usually interested in expressing the running time as a function $T(n)$ of the input size n.

Example 2.1: Running time of `max`.

Recall the `max` function from Chapter 1:

```
export function max(elements: number[]): number | undefined {
  let result: number | undefined;

  for (const element of elements) {
    if (result === undefined || element > result) {
      result = element;
```

```
    }
  }
  return result;
}
```

If we count comparisons as our basic operation, the loop performs exactly one comparison per element (the `element > result` check; the `undefined` check is bookkeeping). For an array of n elements, the running time is $T(n) = n$.

4.3 Asymptotic notation

Rather than stating that an algorithm takes exactly $3n^2 + 7n + 4$ operations, we want to capture the *growth rate* — the fact that the dominant behavior is quadratic. Asymptotic notation gives us a precise way to do this.

4.3.1 Big-O: upper bounds

Definition 2.2 — Big-O notation

Let $f(n)$ and $g(n)$ be functions from the non-negative integers to the non-negative reals. We write

$$f(n) = O(g(n))$$

if there exist constants $c > 0$ and $n_0 \geq 0$ such that

$$f(n) \leq c \cdot g(n) \quad \text{for all } n \geq n_0.$$

In words: f grows no faster than g, up to a constant factor, for sufficiently large n.

Example 2.2. Let $f(n) = 3n^2 + 7n + 4$. We claim $f(n) = O(n^2)$.

Proof. For $n \geq 1$, we have $7n \leq 7n^2$ and $4 \leq 4n^2$, so

$$f(n) = 3n^2 + 7n + 4 \leq 3n^2 + 7n^2 + 4n^2 = 14n^2.$$

Choosing $c = 14$ and $n_0 = 1$ satisfies Definition 2.2. □

Note that $f(n) = O(n^3)$ is also technically true — n^2 is bounded above by n^3 — but it is less informative. By convention, we always state the tightest bound we can prove.

4.3.2 Big-Omega: lower bounds

Definition 2.3 — Big-Omega notation

We write $f(n) = \Omega(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that

$$f(n) \geq c \cdot g(n) \quad \text{for all } n \geq n_0.$$

In words: f grows at least as fast as g, up to a constant factor.

Example 2.3. $3n^2 + 7n + 4 = \Omega(n^2)$.

Proof. For all $n \geq 0$, $3n^2 + 7n + 4 \geq 3n^2 \geq 3 \cdot n^2$. Choose $c = 3$ and $n_0 = 0$. □

Big-Omega is especially useful for expressing *lower bounds* on problems: "any algorithm that solves this problem must take at least $\Omega(g(n))$ time."

4.3.3 Big-Theta: tight bounds

Definition 2.4 — Big-Theta notation

We write $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Equivalently, there exist constants $c_1, c_2 > 0$ and $n_0 \geq 0$ such that

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \quad \text{for all } n \geq n_0.$$

In words: f and g grow at the same rate, up to constant factors.

Example 2.4. From Examples 2.2 and 2.3, we have $3n^2 + 7n + 4 = \Theta(n^2)$.

Big-Theta is the most precise statement: it says the function grows *exactly* like $g(n)$, within constant factors. When we can determine a Big-Theta bound for an algorithm, we have characterized its running time completely (in the asymptotic sense).

4.3.4 Summary of notation

Notation	Meaning	Analogy
$f(n) = O(g(n))$	f grows no faster than g	$f \leq g$
$f(n) = \Omega(g(n))$	f grows at least as fast as g	$f \geq g$
$f(n) = \Theta(g(n))$	f and g grow at the same rate	$f = g$

The analogy to $\leq$, $\geq$, $=$ is informal but helpful for intuition. Formally, all three notations suppress constant factors and describe behavior only for sufficiently large n.

4.3.5 Common growth rates

The following table lists growth rates that appear throughout this book, ordered from slowest to fastest:

Growth rate	Name	Example
$O(1)$	Constant	Array index access
$O(\log n)$	Logarithmic	Binary search
$O(n)$	Linear	Finding the maximum
$O(n \log n)$	Linearithmic	Merge sort, heap sort
$O(n^2)$	Quadratic	Insertion sort (worst case)
$O(n^3)$	Cubic	Floyd-Warshall all-pairs shortest paths
$O(2^n)$	Exponential	Brute-force subset enumeration
$O(n!)$	Factorial	Brute-force permutation enumeration

To appreciate the practical impact, consider an algorithm that performs $T(n)$ operations on a computer executing 10^9 operations per second:

n	n	$n \log_2 n$	n^2	n^3	2^n
10	10 ns	33 ns	100 ns	1 µs	1 µs
100	100 ns	664 ns	10 µs	1 ms	4×10^{13} years
1,000	1 µs	10 µs	1 ms	1 s	—
10^6	1 ms	20 ms	17 min	31.7 years	—
10^9	1 s	30 s	31.7 years	—	—

The table makes a powerful point: the gap between $O(n \log n)$ and $O(n^2)$ is large for a million elements, and the jump to $O(2^n)$ is catastrophic even for modest inputs.

4.4 Best case, worst case, and average case

The running time of an algorithm usually depends on the *specific input*, not just its size. Consider insertion sort.

4.4.1 Insertion sort as a running example

Recall the insertion sort implementation from Chapter 4 (we discuss it fully there, but introduce it here as an analysis example):

```
export function insertionSort<T>(
  elements: T[],
  comparator: Comparator<T> = numberComparator as Comparator<T>,
): T[] {
  const copy = elements.slice(0);

  for (let i = 1; i < copy.length; i++) {
    const toInsert = copy[i]!;
    let insertIndex = i - 1;

    while (insertIndex >= 0 && comparator(toInsert, copy[insertIndex]!) < 0) {
      copy[insertIndex + 1] = copy[insertIndex]!;
      insertIndex--;
    }
    insertIndex++;
    copy[insertIndex] = toInsert;
  }
  return copy;
}
```

The outer loop runs $n - 1$ iterations (for $i = 1, 2, \ldots, n-1$). For each iteration, the inner `while` loop shifts elements to the right until it finds the correct insertion point. The number of shifts depends on the input.

4.4.2 Worst-case analysis

Definition 2.5 — Worst-case running time

The **worst-case running time** $T_{\text{worst}}(n)$ is the maximum running time over all inputs of size n:

$$T_{\text{worst}}(n) = \max_{\text{inputs } I \text{ of size } n} T(I).$$

For insertion sort, the worst case occurs when the array is sorted in reverse order: $[n, n-1, \ldots, 2, 1]$. In this case, every new element must be shifted past all previously sorted elements. The inner loop performs i comparisons in iteration i, so the total number of comparisons is:

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \Theta(n^2).$$

4.4.3 Best-case analysis

Definition 2.6 — Best-case running time

The **best-case running time** $T_{\text{best}}(n)$ is the minimum running time over all inputs of size n:

$$T_{\text{best}}(n) = \min_{\text{inputs } I \text{ of size } n} T(I).$$

For insertion sort, the best case occurs when the array is already sorted. Each new element is already in its correct position, so the inner loop performs zero shifts — just one comparison to discover that no shifting is needed. The total is:

$$\sum_{i=1}^{n-1} 1 = n - 1 = \Theta(n).$$

This is remarkable: insertion sort runs in *linear* time on already-sorted input, matching the theoretical minimum for any comparison-based algorithm that must verify sortedness.

4.4.4 Average-case analysis

Definition 2.7 — Average-case running time

The **average-case running time** is the expected running time over some distribution of inputs. For a uniform distribution over all permutations of n elements:

$$T_{\text{avg}}(n) = \frac{1}{n!} \sum_{\text{permutations } \pi} T(\pi).$$

For insertion sort, consider iteration i: the element being inserted has an equal probability of belonging at any of the $i+1$ positions in the sorted prefix. On average, it must be shifted past half of the sorted elements, so the expected number of comparisons in iteration i is roughly $i/2$. The total expected comparisons are:

$$\sum_{i=1}^{n-1} \frac{i}{2} = \frac{n(n-1)}{4} = \Theta(n^2).$$

The average case is still $\Theta(n^2)$ — the same order of growth as the worst case. The constant factor is half as large, but asymptotically the behavior is the same.

4.4.5 Which case matters?

In practice, **worst-case analysis** is the most commonly used, for several reasons:

1. **Guarantees.** The worst case gives an upper bound that holds for *every* input. This is crucial in real-time systems, web servers, and other contexts where performance must be predictable.
2. **Average case can be misleading.** The "average" depends on the input distribution, which we may not know. If the actual inputs differ from our assumption, the average-case analysis may not apply.
3. **Worst case is often typical.** For many algorithms, the worst case and average case have the same asymptotic growth rate (as we just saw with insertion sort).

We will occasionally discuss best-case and average-case bounds when they provide useful insight, but unless otherwise stated, all complexity bounds in this book refer to the worst case.

4.5 Amortized analysis

Sometimes an operation is expensive *occasionally* but cheap *most of the time*. Amortized analysis gives us a way to average the cost over a sequence of operations, providing a tighter bound than the worst-case cost per operation.

4.5.1 The dynamic array example

Consider a dynamic array (like JavaScript's `Array` or `std::vector` in C++) that supports an `append` operation. The array maintains an internal buffer of some capacity. When the buffer is full and a new element is appended, the array allocates a new buffer of double the capacity and copies all existing elements over. This *resize* operation costs $O(n)$, where n is the current number of elements.

At first glance, this seems concerning: a single `append` can cost $O(n)$. But resizes happen infrequently — only when the size reaches a power of 2. Let us analyze the cost of n consecutive appends starting from an empty array.

The resize operations happen at sizes 1, 2, 4, 8, ..., 2^k, where $2^k \leq n$. The total copying cost across all resizes is:

$$1 + 2 + 4 + 8 + \cdots + 2^k = 2^{k+1} - 1 < 2n.$$

Adding the n non-resize operations (cost 1 each), the total cost of n appends is less than $3n$. Therefore the **amortized cost** per append is:

$$\frac{3n}{n} = O(1).$$

Each individual append may cost $O(n)$ in the worst case, but *averaged over a sequence of operations*, the cost is $O(1)$ per operation.

4.5.2 Amortized vs. average case

It is important to distinguish amortized analysis from average-case analysis:

- **Average case** averages over *random inputs*: we assume a probability distribution on the inputs and compute the expected running time.
- **Amortized analysis** averages over a *sequence of operations* on a *worst-case* input: no probability is involved. The guarantee holds deterministically.

Amortized analysis says: "no matter what sequence of n operations you perform, the total cost is at most $f(n)$, so the amortized cost per operation is $f(n)/n$." This is a worst-case guarantee about the total, not a probabilistic statement.

We will see amortized analysis again in Chapter 7 (dynamic arrays), Chapter 11 (binary heaps), and Chapter 18 (union-find).

4.6 Recurrence relations

When an algorithm calls itself recursively, its running time is naturally expressed as a *recurrence relation*: a formula that expresses $T(n)$ in terms of T applied to smaller values.

4.6.1 Setting up a recurrence

Example 2.5: Binary search. Binary search (discussed in Chapter 3) repeatedly halves the search space:

1. Compare the target with the middle element.
2. If they match, return the index.
3. Otherwise, recurse on the left or right half.

The running time satisfies the recurrence:

$$T(n) = T(n/2) + O(1), \quad T(1) = O(1).$$

The $T(n/2)$ term accounts for the recursive call on half the array, and the $O(1)$ term accounts for the comparison and index computation.

Example 2.6: Merge sort. Merge sort (discussed in Chapter 5) divides the array in half, recursively sorts both halves, and merges the results:

$$T(n) = 2T(n/2) + O(n), \quad T(1) = O(1).$$

The two recursive calls each process half the array ($2T(n/2)$), and the merge step takes $O(n)$ time.

4.6.2 Solving recurrences by expansion

One way to solve a recurrence is to expand it repeatedly until a pattern emerges.

Example 2.7: Solving the binary search recurrence.

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

Expanding:

$$T(n) = T(n/4) + c + c = T(n/4) + 2c$$
$$= T(n/8) + 3c$$

$$= T(n/2^k) + kc$$

The recursion bottoms out when $n/2^k = 1$, i.e., $k = \log_2 n$. Therefore:

$$T(n) = T(1) + c \log_2 n = O(\log n).$$

Example 2.8: Solving the merge sort recurrence.

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

Expanding:

$$T(n) = 2[2T(n/4) + cn/2] + cn = 4T(n/4) + 2cn$$
$$= 4[2T(n/8) + cn/4] + 2cn = 8T(n/8) + 3cn$$

At level k: $T(n) = 2^k T(n/2^k) + kcn$. Setting $k = \log_2 n$:

$$T(n) = nT(1) + cn \log_2 n = O(n \log n).$$

4.6.3 The recursion tree method

A recursion tree is a visual tool for solving recurrences. Each node represents the cost at one level of recursion, and the total cost is the sum over all nodes.

For merge sort with $T(n) = 2T(n/2) + cn$:

```
Level 0:              cn                    → cost cn
                    /    \
Level 1:        cn/2      cn/2              → cost cn
                / \       / \
Level 2:    cn/4  cn/4  cn/4  cn/4          → cost cn
              ...         ...
Level k:   c  c  c  ...  c  c  c           → cost cn
           (n leaves)
```

There are $\log_2 n$ levels, each contributing cn work, so the total is $cn \log_2 n = O(n \log n)$.

4.7 The Master Theorem

The Master Theorem provides a general solution for recurrences of a common form.

Definition 2.8 — The Master Theorem

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined by the recurrence

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

Then $T(n)$ can be bounded asymptotically as follows:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and sufficiently large n, then $T(n) = \Theta(f(n))$.

The three cases correspond to three scenarios:

- **Case 1:** The cost is dominated by the leaves of the recursion tree. The recursive calls do most of the work.
- **Case 2:** The cost is evenly distributed across all levels of the tree. Each level contributes roughly equally.
- **Case 3:** The cost is dominated by the root. The non-recursive work $f(n)$ dominates.

Let us apply the Master Theorem to our earlier examples.

Example 2.9: Binary search. $T(n) = T(n/2) + O(1)$.

Here $a = 1$, $b = 2$, $f(n) = O(1)$. We have $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$. Since $f(n) = \Theta(1) = \Theta(n^{\log_b a})$, Case 2 applies:

$$T(n) = \Theta(\log n).$$

Example 2.10: Merge sort. $T(n) = 2T(n/2) + O(n)$.

Here $a = 2$, $b = 2$, $f(n) = O(n)$. We have $n^{\log_b a} = n^{\log_2 2} = n^1 = n$. Since $f(n) = \Theta(n) = \Theta(n^{\log_b a})$, Case 2 applies:

$$T(n) = \Theta(n \log n).$$

Example 2.11: Strassen's matrix multiplication. $T(n) = 7T(n/2) + O(n^2)$.

Here $a = 7$, $b = 2$, $f(n) = O(n^2)$. We have $n^{\log_b a} = n^{\log_2 7} \approx n^{2.807}$. Since $f(n) = O(n^2) = O(n^{2.807-\epsilon})$ with $\epsilon \approx 0.807$, Case 1 applies:

$$T(n) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.807}).$$

This is better than the naive $O(n^3)$ matrix multiplication.

4.7.1 Limitations of the Master Theorem

The Master Theorem does not cover all recurrences. It requires that the subproblems be of equal size n/b and that $f(n)$ fall into one of the three cases. Recurrences like $T(n) = T(n/3) + T(2n/3) + O(n)$ (which arises in randomized quicksort analysis) do not fit the template directly. For such cases, the recursion-tree method or the Akra–Bazzi theorem can be used.

4.8 Space complexity

So far we have focused on time complexity, but algorithms also consume *memory*. Space complexity measures the amount of additional memory an algorithm uses beyond the input.

Definition 2.9 — Space complexity

The **space complexity** of an algorithm is the maximum amount of memory it uses at any point during execution, measured as a function of the input size.

We distinguish between:

- **Auxiliary space:** the extra memory used beyond the input itself.
- **Total space:** auxiliary space plus the space for the input.

Unless stated otherwise, when we refer to "space complexity" in this book, we mean auxiliary space.

Example 2.12: Space complexity of `max`.

The `max` function from Chapter 1 uses a single variable `result`. Its auxiliary space is $O(1)$.

Example 2.13: Space complexity of merge sort.

Merge sort requires a temporary array of size n for the merge step, plus $O(\log n)$ space for the recursion stack. Its auxiliary space is $O(n)$.

Example 2.14: Space complexity of insertion sort.

Our insertion sort implementation copies the input array (space $O(n)$). An in-place variant that sorts the array directly would use only $O(1)$ auxiliary space.

4.8.1 Time–space trade-offs

Often there is a trade-off between time and space. An algorithm can sometimes be made faster by using more memory, or made more memory-efficient at the cost of additional computation. A classic example:

- **Hash table** lookup: $O(1)$ average time, $O(n)$ space.
- **Linear search** through an unsorted array: $O(n)$ time, $O(1)$ space.

Both solve the problem of finding an element in a collection, but they make different trade-offs. Recognizing and navigating these trade-offs is a recurring theme in algorithm design.

4.9 Practical considerations

Asymptotic analysis is a powerful framework, but it has limitations that a practicing programmer should keep in mind.

4.9.1 Constant factors matter for moderate n

Asymptotic notation hides constant factors. An algorithm with running time $100n$ is $O(n)$, and an algorithm with running time $2n \log n$ is $O(n \log n)$. For $n < 2^{50}$, the "slower" $O(n \log n)$ algorithm is actually faster. In practice, constant factors depend on:

- The number of operations per step.
- Cache behavior — algorithms with good spatial locality are faster in practice.
- Branch prediction — algorithms with predictable control flow benefit from CPU branch predictors.

This is why, for example, insertion sort (which is $O(n^2)$) is often used for small arrays (say, $n < 20$) even inside asymptotically faster algorithms like merge sort. The constant factor is smaller, and for tiny inputs the quadratic term has not yet become dominant.

4.9.2 Lower-order terms

An algorithm that performs $n^2 + 10^6 n$ operations is $\Theta(n^2)$, but for $n < 10^6$, the linear term dominates. Asymptotic analysis describes long-term growth; for small inputs, the actual constants and lower-order terms may be more important.

4.9.3 Choosing the right model

Our analysis assumes a simple model where every basic operation takes the same amount of time. Real computers have caches, pipelines, and memory hierarchies that make some access patterns much faster than others. An algorithm that accesses memory sequentially (like insertion sort) can be significantly faster in practice than one that accesses memory randomly (like binary search on a large array), even if the latter has a better asymptotic bound.

Despite these caveats, asymptotic analysis remains the single most useful tool for comparing algorithms. It correctly predicts which algorithm will win for large enough inputs, and "large enough" usually means "the sizes that actually matter in practice."

4.10 Looking ahead

In this chapter we have developed the fundamental tools for analyzing algorithms:

- **Asymptotic notation** (O, Ω, Θ) captures growth rates while abstracting away constant factors and hardware details.
- **Worst-case analysis** gives reliable upper bounds on running time. Best-case and average-case analyses provide additional insight.
- **Amortized analysis** reveals that operations with occasional expensive steps can still be efficient on average.
- **Recurrence relations** express the running time of recursive algorithms, and the **Master Theorem** provides a quick way to solve common recurrences.
- **Space complexity** measures memory usage and highlights time–space trade-offs.

Armed with these tools, we are ready to analyze every algorithm in this book rigorously. In the next chapter, we explore recursion and the divide-and-conquer

strategy — one of the most powerful algorithm design techniques — and apply our analytical framework to algorithms like binary search and the closest pair of points.

4.11 Exercises

Exercise 2.1. Rank the following functions by asymptotic growth rate, from slowest to fastest. For each consecutive pair, state whether $f = O(g)$, $f = \Omega(g)$, or $f = \Theta(g)$.

$$\log_2 n, \quad n^2, \quad \sqrt{n}, \quad 2^n, \quad n \log n, \quad n!, \quad n, \quad 1$$

Exercise 2.2. Prove or disprove: if $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$. (In other words, is Big-O transitive?)

Exercise 2.3. For each of the following recurrences, use the Master Theorem to determine the asymptotic bound, or explain why the Master Theorem does not apply.

(a) $T(n) = 4T(n/2) + n$

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

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

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

Exercise 2.4. Consider a dynamic array that triples (instead of doubles) its capacity when full. Prove that the amortized cost of an `append` operation is still $O(1)$. How does the constant factor compare to the doubling strategy?

Exercise 2.5. An algorithm processes an array of n elements as follows: for each element, it performs a binary search over the preceding elements. What is the overall time complexity? Express your answer in Big-Theta notation.

Chapter 5

Recursion and Divide-and-Conquer

Recursion is one of the most powerful techniques in algorithm design: a function solving a problem by solving smaller instances of itself. In this chapter we study recursion from the ground up, connect it to mathematical induction, and then develop the divide-and-conquer strategy — splitting a problem into independent subproblems, solving each recursively, and combining the results. We illustrate these ideas with four algorithms: binary search, fast exponentiation, the Euclidean algorithm for greatest common divisors, and the closest pair of points.

5.1 Recursion

A function is *recursive* if it calls itself. This is not mere circularity — each call works on a smaller instance of the problem, and eventually the instances become small enough to solve directly. Every recursive function has two essential ingredients:

1. **Base case.** One or more input sizes for which the answer is immediate, without further recursion.
2. **Recursive case.** For larger inputs, the function reduces the problem to one or more smaller instances and combines the results.

Consider a simple example: computing the factorial $n! = 1 \cdot 2 \cdots n$.

```
function factorial(n: number): number {
  if (n <= 1) return 1;   // base case
  return n * factorial(n - 1); // recursive case
}
```

The base case is $n \leq 1$, where we return 1. The recursive case multiplies n by the factorial of $n-1$. Each recursive call reduces the argument by 1, so the chain of calls eventually reaches the base case:

$$\text{factorial}(4) = 4 \cdot \text{factorial}(3) = 4 \cdot 3 \cdot \text{factorial}(2) = 4 \cdot 3 \cdot 2 \cdot \text{factorial}(1) = 4 \cdot 3 \cdot 2 \cdot 1 = 24.$$

5.1.1 The call stack

When a function calls itself, the runtime maintains a *call stack* — a stack of *frames*, each recording the local variables and return address for one invocation. For `factorial(4)`, the stack grows to depth 4 before the base case is reached:

```
factorial(4)  — waiting for factorial(3)
  factorial(3)  — waiting for factorial(2)
    factorial(2)  — waiting for factorial(1)
      factorial(1)  — returns 1
    factorial(2)  — returns 2 × 1 = 2
  factorial(3)  — returns 3 × 2 = 6
factorial(4)  — returns 4 × 6 = 24
```

Each frame occupies memory, so a recursion of depth d uses $O(d)$ stack space. For `factorial(n)`, the depth is n, so the space complexity is $O(n)$. This overhead can be a concern for very deep recursions, but for many problems the clarity and elegance of the recursive solution outweigh the cost.

5.1.2 Common pitfalls

Two mistakes arise frequently when writing recursive functions:

1. **Missing base case.** Without a base case, the recursion never terminates:

   ```typescript
   function infiniteRecursion(n: number): number {
     return n * infiniteRecursion(n - 1); // no base case!
   }
   ```

   This is not an algorithm in the sense of Definition 1.1 — it does not terminate.

2. **Subproblems that do not shrink.** Even with a base case, the recursion must make progress:

   ```typescript
   function noProgress(n: number): number {
     if (n <= 1) return 1;
     return n * noProgress(n); // n does not decrease!
   }
   ```

This function never reaches the base case for $n > 1$.

5.2 Recursion and mathematical induction

There is a deep connection between recursion and mathematical induction. Induction proves that a property holds for all natural numbers; recursion computes a value for all valid inputs. The structures are parallel:

Induction	Recursion
Base case: prove $P(0)$ (or $P(1)$)	Base case: return a value directly
Inductive step: assuming $P(k)$, prove $P(k+1)$	Recursive case: assuming the recursive call returns the correct result, compute the current result

This parallel is not a coincidence — it is the foundation for proving recursive algorithms correct. To prove that a recursive function computes the right answer, we use *strong induction* (also called *complete induction*): assume the function works correctly for all inputs smaller than n, and show it works correctly for input n.

Definition 3.1 — Correctness of a recursive algorithm

A recursive algorithm is **correct** if:

1. It produces the correct answer on all base cases.
2. If every recursive call on a strictly smaller input returns the correct answer, then the current call also returns the correct answer.

Example 3.1: Correctness of `factorial`.

Base case. When $n \leq 1$, the function returns 1, and indeed $0! = 1! = 1$.

Inductive step. Assume `factorial(k)` returns $k!$ for all $k < n$. Then `factorial(n)` returns $n \cdot \text{factorial}(n-1) = n \cdot (n-1)! = n!$. □

5.3 Divide and conquer

Divide and conquer is a specific recursion pattern that solves a problem by:

1. **Divide:** split the input into two or more smaller subproblems of the same type.

2. **Conquer:** solve each subproblem recursively (or directly if it is small enough).
3. **Combine:** merge the subproblem solutions into a solution for the original problem.

Not every recursive algorithm is divide-and-conquer. The factorial function above reduces the problem by a constant amount (from n to $n-1$), which is sometimes called *decrease and conquer*. True divide-and-conquer algorithms typically split the input by a constant *fraction* (usually in half), leading to logarithmic recursion depth and often dramatically better performance.

The running time of a divide-and-conquer algorithm is typically expressed as a recurrence of the form

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

where a is the number of subproblems, n/b is their size, and $f(n)$ is the cost of dividing and combining. As we saw in Chapter 2, the Master Theorem often gives us the solution directly.

5.4 Binary search

Our first divide-and-conquer algorithm is one of the most important: binary search. It finds the position of a target value in a *sorted* array by repeatedly halving the search space.

5.4.1 The problem

Input: A sorted array $A[0..n-1]$ of numbers and a target value x.

Output: An index i such that $A[i] = x$, or -1 if x is not in A.

5.4.2 The algorithm

The idea is simple: compare x with the middle element of the array.

- If they match, return the index.
- If x is smaller, recurse on the left half.
- If x is larger, recurse on the right half.

Each step eliminates half the remaining elements.

Here is our iterative implementation (an iterative approach avoids the overhead of recursive calls and is standard for binary search):

```
export function binarySearch(arr: number[], element: number): number {
  let low = 0;
  let high = arr.length - 1;

  while (low <= high) {
    const mid = Math.floor((low + high) / 2);
    const midVal = arr[mid]!;

    if (midVal === element) {
      return mid;
    } else if (midVal < element) {
      low = mid + 1;
    } else {
      high = mid - 1;
    }
  }
  return -1;
}
```

Although this implementation is iterative, it mirrors the recursive divide-and-conquer structure exactly: the variables `low` and `high` define the current subproblem, and each iteration halves the range.

5.4.3 Tracing through an example

Let $A = [1, 3, 5, 7, 9, 11, 13]$ and $x = 9$.

Step	`low`	`high`	`mid`	`arr[mid]`	Action
1	0	6	3	7	$9 > 7$: search right half
2	4	6	5	11	$9 < 11$: search left half
3	4	4	4	9	$9 = 9$: found, return 4

After only 3 comparisons, we have found the element in a 7-element array. A linear scan might have taken up to 7 comparisons.

5.4.4 Correctness

We prove correctness using a loop invariant.

Invariant: If x is in A, then x is in A[low..high].

- **Initialization:** Before the loop, low = 0 and high = $n - 1$, so the invariant holds trivially.

- **Maintenance:** If $A[\text{mid}] < x$, then x cannot be in $A[\text{low}..\text{mid}]$ (since A is sorted), so setting low = mid + 1 preserves the invariant. The case $A[\text{mid}] > x$ is symmetric.
- **Termination:** The loop terminates either when x is found or when low > high, meaning the search range is empty. In the latter case, x is not in A, and returning -1 is correct.

5.4.5 Complexity analysis

Each iteration halves the search range. Starting from n elements, after k iterations we have at most $n/2^k$ elements. The loop terminates when $n/2^k < 1$, i.e., after $k = \lceil \log_2 n \rceil$ iterations.

- **Time complexity:** $O(\log n)$.
- **Space complexity:** $O(1)$ (the iterative version uses only a few variables).

Using the Master Theorem on the recursive form: $T(n) = T(n/2) + O(1)$. Here $a = 1$, $b = 2$, $n^{\log_b a} = n^0 = 1$. Since $f(n) = O(1) = \Theta(n^{\log_b a})$, Case 2 gives $T(n) = \Theta(\log n)$.

5.4.6 Comparison with linear search

For comparison, here is the linear search algorithm:

```
export function linearSearch<T>(arr: T[], element: T): number {
  let position = -1;
  let currentIndex = 0;

  while (position < 0 && currentIndex < arr.length) {
    if (arr[currentIndex] === element) {
      position = currentIndex;
    } else {
      currentIndex++;
    }
  }
  return position;
}
```

Linear search works on *any* array (not just sorted ones) but takes $O(n)$ time. Binary search requires a sorted array but is exponentially faster:

Elements	Linear search	Binary search
1,000	1,000 comparisons	10 comparisons
1,000,000	1,000,000 comparisons	20 comparisons

Elements	Linear search	Binary search
10^9	10^9 comparisons	30 comparisons

This dramatic improvement — from linear to logarithmic — is the hallmark of the divide-and-conquer approach. The key insight is that each comparison does not eliminate a single element but *half the remaining elements*.

5.5 Fast exponentiation (exponentiation by squaring)

Our second example addresses the problem of computing b^n efficiently.

5.5.1 The problem

Input: A number b (the base) and a non-negative integer n (the exponent).

Output: The value b^n.

5.5.2 Naive approach

The straightforward approach multiplies b by itself n times:

```
export function powSlow(base: number, power: number): number {
  let result = 1;
  for (let i = 0; i < power; i++) {
    result = result * base;
  }
  return result;
}
```

This performs n multiplications, so it runs in $O(n)$ time.

5.5.3 Exponentiation by squaring

We can do much better by observing a simple mathematical identity:

$$b^n = \begin{cases} 1 & \text{if } n = 0, \\ (b^{n/2})^2 & \text{if } n \text{ is even}, \\ b \cdot b^{n-1} & \text{if } n \text{ is odd}. \end{cases}$$

When n is even, we compute $b^{n/2}$ once and square the result — a single multiplication instead of $n/2$ multiplications. When n is odd, we reduce to an even exponent by extracting one factor of b.

Here is the iterative implementation:

```
export function pow(base: number, power: number): number {
  let result = 1;

  while (power > 0) {
    if (power % 2 === 0) {
      base = base * base;
      power = power / 2;
    } else {
      result = result * base;
      power = power - 1;
    }
  }
  return result;
}
```

5.5.4 Tracing through an example

Let us compute 2^{10}:

Step	base	power	result	Action
1	2	10	1	Even: base $\leftarrow 2^2 = 4$, power $\leftarrow 5$
2	4	5	1	Odd: result $\leftarrow 1 \times 4 = 4$, power $\leftarrow 4$
3	4	4	4	Even: base $\leftarrow 4^2 = 16$, power $\leftarrow 2$
4	16	2	4	Even: base $\leftarrow 16^2 = 256$, power $\leftarrow 1$
5	256	1	4	Odd: result $\leftarrow 4 \times 256 = 1024$, power $\leftarrow 0$

Result: $2^{10} = 1024$. The naive approach would have used 10 multiplications; fast exponentiation used 5.

5.5.5 Correctness

Invariant: At the start of each iteration, result $\times$ base$^{\text{power}}$ equals the original b^n.

- **Initialization.** result = 1, base = b, power = n. The invariant $1 \times b^n = b^n$ holds.
- **Maintenance.**
  - If power is even: we replace base with base^2 and power with power/2. Then result $\times$ (base^2)$^{\text{power}/2}$ = result $\times$ base$^{\text{power}}$. Invariant preserved.

  - If power is odd: we replace result with result × base and power with power − 1. Then $(\text{result} \times \text{base}) \times \text{base}^{\text{power}-1} = \text{result} \times \text{base}^{\text{power}}$. Invariant preserved.
- **Termination.** When power = 0, the invariant gives $\text{result} \times \text{base}^0 = \text{result} = b^n$. □

5.5.6 Complexity analysis

At each "odd" step, the exponent decreases by 1 (making it even). At each "even" step, the exponent halves. After at most two consecutive steps (one odd, one even), the exponent has been at least halved. Therefore the total number of steps is $O(\log n)$.

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

The recurrence for the recursive view is $T(n) = T(n/2) + O(1)$, the same as binary search, giving $\Theta(\log n)$ by the Master Theorem.

5.6 The Euclidean algorithm for GCD

The greatest common divisor (GCD) of two positive integers x and y is the largest integer that divides both. It is one of the oldest algorithms known, recorded by Euclid around 300 BC.

5.6.1 The problem

Input: Two positive integers x and y.

Output: $\gcd(x, y)$, the largest positive integer dividing both x and y.

5.6.2 Naive approach

The brute-force approach tries every candidate from the larger number downward:

```
export function gcdSlow(x: number, y: number): number {
  const max = Math.max(x, y);

  for (let i = max; i >= 2; i--) {
    if (x % i === 0 && y % i === 0) {
      return i;
    }
  }
```

```
  return 1;
}
```

This checks up to $\max(x, y)$ candidates, so its time complexity is $O(\max(x, y))$.

5.6.3 The Euclidean algorithm

The Euclidean algorithm is based on a key observation:

$$\gcd(x, y) = \gcd(y, x \bmod y)$$

This holds because any common divisor of x and y also divides $x \bmod y$ (since $x \bmod y = x - \lfloor x/y \rfloor \cdot y$), and conversely. Since $x \bmod y < y$, the arguments strictly decrease, and the process terminates when the remainder is 0:

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

Here is the implementation:

```
export function gcd(x: number, y: number): number {
  let r = x % y;

  while (r > 0) {
    x = y;
    y = r;
    r = x % y;
  }
  return y;
}
```

5.6.4 Tracing through an example

Let us compute $\gcd(210, 2618)$:

Step	x	y	$r = x \bmod y$
1	210	2618	210
2	2618	210	98
3	210	98	14
4	98	14	0

Result: $\gcd(210, 2618) = 14$.

The naive approach would have tested candidates from 2618 down to 14 — over 2600 iterations. The Euclidean algorithm needed only 4.

5.6.5 Correctness

We prove correctness by induction on the number of iterations.

Base case. If $x \bmod y = 0$, then y divides x, so $\gcd(x, y) = y$. The algorithm returns y. Correct.

Inductive step. Assume the algorithm correctly computes $\gcd(y, r)$ where $r = x \bmod y$. Since $\gcd(x, y) = \gcd(y, x \bmod y)$, the result is correct. □

5.6.6 Complexity analysis

The key insight is that after two consecutive iterations, the value of y is reduced by at least half. Formally: if $r = x \bmod y$, then $r < y$, and one can show that $x \bmod y < x/2$ whenever $y \leq x$. By the Fibonacci-like worst case analysis (due to Gabriel Lamé, 1844):

- **Time complexity:** $O(\log(\min(x, y)))$.
- **Space complexity:** $O(1)$.

This is an exponential improvement over the naive $O(\max(x, y))$ approach.

5.7 The closest pair of points

Our most substantial example brings together all the divide-and-conquer ideas. Given a set of points in the plane, we want to find two points that are closest to each other.

5.7.1 The problem

Input: A set P of $n \geq 2$ points in the plane, where each point is a pair (x, y) of coordinates.

Output: A pair of points $p_1, p_2 \in P$ that minimize the Euclidean distance $d(p_1, p_2) = \sqrt{(p_1.x - p_2.x)^2 + (p_1.y - p_2.y)^2}$.

5.7.2 Brute-force approach

The obvious approach checks all $\binom{n}{2}$ pairs:

```
function bruteForce(pts: readonly Point[]): ClosestPairResult {
  let best: ClosestPairResult = {
```

```
    p1: pts[0]!,
    p2: pts[1]!,
    distance: distance(pts[0]!, pts[1]!),
  };

  for (let i = 0; i < pts.length; i++) {
    for (let j = i + 1; j < pts.length; j++) {
      const d = distance(pts[i]!, pts[j]!);
      if (d < best.distance) {
        best = { p1: pts[i]!, p2: pts[j]!, distance: d };
      }
    }
  }
  return best;
}
```

This runs in $\Theta(n^2)$ time. Can we do better?

5.7.3 The divide-and-conquer idea

The strategy is:

1. **Divide.** Sort the points by x-coordinate and split them into a left half L and a right half R at the median x-value.
2. **Conquer.** Recursively find the closest pair in L and in R. Let δ_L and δ_R be these distances, and let $\delta = \min(\delta_L, \delta_R)$.
3. **Combine.** The overall closest pair is either entirely in L, entirely in R, or *split* — with one point in L and one in R. We have already found the first two cases. For the split case, we need to check if any split pair has distance less than δ.

The crux of the algorithm is the combine step: can we check split pairs efficiently?

5.7.4 The strip optimization

Consider the vertical strip of width 2δ centered on the dividing line (at the median x-coordinate). Any split pair with distance less than δ must have both points in this strip, because otherwise the horizontal distance alone exceeds δ.

Now comes the key geometric insight. Sort the points in the strip by y-coordinate. For any point p in the strip, how many other strip points can be within distance δ of p? Since all such points lie in a $2\delta \times \delta$ rectangle, and

any two points in the same half (left or right) are at least δ apart, a packing argument shows that at most **7** other points in the strip need to be checked.

This means the combine step checks each strip point against at most 7 neighbors — a constant number — so it takes $O(n)$ time (after sorting the strip by y).

5.7.5 Implementation

We define the `Point` and `ClosestPairResult` types:

```
export interface Point {
  x: number;
  y: number;
}

export interface ClosestPairResult {
  p1: Point;
  p2: Point;
  distance: number;
}
```

The distance function:

```
export function distance(a: Point, b: Point): number {
  const dx = a.x - b.x;
  const dy = a.y - b.y;
  return Math.sqrt(dx * dx + dy * dy);
}
```

The main function sorts by x-coordinate and delegates to the recursive helper:

```
export function closestPair(points: readonly Point[]): ClosestPairResult {
  if (points.length < 2) {
    throw new Error('At least 2 points are required');
  }
  const sortedByX = [...points].sort(
    (a, b) => a.x - b.x || a.y - b.y,
  );
  return closestPairRec(sortedByX);
}
```

The recursive function implements the three steps:

```
function closestPairRec(pts: readonly Point[]): ClosestPairResult {
  if (pts.length <= 3) {
    return bruteForce(pts);
```

```
  }

  const mid = Math.floor(pts.length / 2);
  const midPoint = pts[mid]!;

  const left = pts.slice(0, mid);
  const right = pts.slice(mid);

  const leftResult = closestPairRec(left);
  const rightResult = closestPairRec(right);

  let best =
    leftResult.distance <= rightResult.distance
      ? leftResult
      : rightResult;
  const delta = best.distance;

  // Build the strip
  const strip: Point[] = [];
  for (const p of pts) {
    if (Math.abs(p.x - midPoint.x) < delta) {
      strip.push(p);
    }
  }

  // Sort strip by y-coordinate
  strip.sort((a, b) => a.y - b.y);

  // Check each point against at most 7 subsequent points
  for (let i = 0; i < strip.length; i++) {
    for (let j = i + 1; j < strip.length; j++) {
      const dy = strip[j]!.y - strip[i]!.y;
      if (dy >= best.distance) {
        break;
      }
      const d = distance(strip[i]!, strip[j]!);
      if (d < best.distance) {
        best = { p1: strip[i]!, p2: strip[j]!, distance: d };
      }
    }
  }
```

```
  return best;
}
```

5.7.6 Tracing through an example

Consider 6 points:

$$P = \{(2,3),\ (12,30),\ (40,50),\ (5,1),\ (12,10),\ (3,4)\}$$

Step 1: Sort by x. $(2,3),\ (3,4),\ (5,1),\ (12,10),\ (12,30),\ (40,50)$.

Step 2: Divide. Left: $(2,3),\ (3,4),\ (5,1)$. Right: $(12,10),\ (12,30),\ (40,50)$. Dividing line at $x = 12$.

Step 3: Conquer (left). With 3 points, brute force checks all 3 pairs:

- $d((2,3),(3,4)) = \sqrt{2} \approx 1.414$
- $d((2,3),(5,1)) = \sqrt{13} \approx 3.606$
- $d((3,4),(5,1)) = \sqrt{13} \approx 3.606$

Closest in left: $\{(2,3),(3,4)\}$ with $\delta_L = \sqrt{2}$.

Step 3: Conquer (right). Brute force on $(12,10),\ (12,30),\ (40,50)$:

- $d((12,10),(12,30)) = 20$
- $d((12,10),(40,50)) = \sqrt{2384} \approx 48.83$
- $d((12,30),(40,50)) = \sqrt{1184} \approx 34.41$

Closest in right: $\{(12,10),(12,30)\}$ with $\delta_R = 20$.

Step 4: Combine. $\delta = \min(\sqrt{2}, 20) = \sqrt{2} \approx 1.414$. The strip contains all points within $\sqrt{2}$ of $x = 12$ — which includes none of the left points (they are at $x = 2, 3, 5$, all more than $\sqrt{2}$ away from 12) and only $(12,10)$ and $(12,30)$ on the right. The strip pair distance is 20, which does not improve on δ.

Result: The closest pair is $\{(2,3),(3,4)\}$ with distance $\sqrt{2}$.

5.7.7 Correctness

The algorithm correctly finds the closest pair because it considers all three possible cases — closest pair entirely in the left, entirely in the right, or split across the dividing line. The correctness of the strip check follows from the geometric packing argument: any split pair closer than δ must lie in the strip and must appear within 7 positions of each other when sorted by y.

Base case. For 2 or 3 points, brute force checks all pairs. Correct.

Inductive step. Assume the recursive calls return the correct closest pairs in L and R. Then δ is the correct minimum distance within each half. The strip check examines all candidates for a closer split pair. Since the inner loop breaks when the y-distance exceeds δ, and any valid split pair must appear within 7 y-neighbors, no valid candidate is missed. □

5.7.8 Complexity analysis

Let $T(n)$ be the running time. The algorithm:

- Divides the points in half: $O(1)$ (the array is already sorted by x).
- Recursively solves two subproblems: $2T(n/2)$.
- Builds and sorts the strip: $O(n \log n)$ in the worst case (the strip could contain all n points).
- Checks strip pairs: $O(n)$ (each point is compared with at most 7 neighbors).

The combine step is dominated by the strip sort at $O(n \log n)$. The recurrence is:

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

This does not fall neatly into Case 2 of the Master Theorem (where $f(n) = \Theta(n^{\log_b a}) = \Theta(n)$). Solving by the recursion tree method or the Akra-Bazzi theorem gives $T(n) = O(n \log^2 n)$.

However, the initial sort by x-coordinate costs $O(n \log n)$ and is done once. With a more careful implementation (maintaining a pre-sorted-by-y list using a merge step instead of re-sorting the strip), the combine step can be reduced to $O(n)$, giving the optimal recurrence:

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

Our implementation uses the simpler $O(n \log^2 n)$ approach, which is already a substantial improvement over the $O(n^2)$ brute force. In practice, the strip is typically much smaller than n, so the extra logarithmic factor is rarely felt.

- **Time complexity:** $O(n \log^2 n)$ as implemented; $O(n \log n)$ with the merge-based optimization.
- **Space complexity:** $O(n)$ for the sorted arrays and strip.

5.7.9 Summary of closest pair

Approach	Time	Space
Brute force	$\Theta(n^2)$	$O(1)$
Divide-and-conquer (simple)	$O(n \log^2 n)$	$O(n)$
Divide-and-conquer (optimal)	$O(n \log n)$	$O(n)$

The closest pair problem beautifully illustrates the power of divide and conquer. The brute-force approach must check all $\binom{n}{2} = \Theta(n^2)$ pairs. By splitting the problem, solving each half, and cleverly bounding the combine step, we achieve near-linear time.

5.8 The divide-and-conquer recipe

Looking back at our four algorithms, we can identify a common recipe:

1. **Identify a way to shrink the problem.** Binary search halves the array, exponentiation by squaring halves the exponent, the Euclidean algorithm replaces a number with a remainder, and closest pair splits the point set.
2. **Solve the smaller instance(s).** Sometimes there is one subproblem (binary search, exponentiation, GCD); sometimes there are two (closest pair).
3. **Combine.** Binary search and GCD need no combining — the subproblem answer is the final answer. Exponentiation squares the subresult. Closest pair must check the strip.
4. **Analyze with recurrences.** The running time follows from the recurrence $T(n) = aT(n/b) + f(n)$ and the Master Theorem (or recursion tree method when the Master Theorem does not apply directly).

This recipe is a powerful tool for designing new algorithms. When you face a problem, ask: can I split it into smaller instances of the same problem? If so, the divide-and-conquer approach may yield an efficient solution.

5.9 Looking ahead

In this chapter we developed recursion and the divide-and-conquer paradigm:

- **Recursion** solves a problem by reducing it to smaller instances, terminating at base cases. Its correctness is proven by induction.
- **Divide-and-conquer** is a specific recursion pattern: divide into subproblems, conquer recursively, combine the results.
- **Binary search** halves the search space at each step, achieving $O(\log n)$ time.

- **Exponentiation by squaring** computes b^n in $O(\log n)$ multiplications instead of $O(n)$.
- **The Euclidean algorithm** computes GCD in $O(\log(\min(x, y)))$ time, an ancient and elegant application of the divide-and-conquer idea.
- **The closest pair of points** demonstrates a nontrivial combine step, achieving $O(n \log n)$ (or $O(n \log^2 n)$ in the simpler variant) versus $O(n^2)$ brute force.

In the next chapter, we turn to the sorting problem. We begin with three elementary sorting algorithms — bubble sort, selection sort, and insertion sort — all of which run in $O(n^2)$ time. In Chapter 5, we study efficient sorting algorithms — merge sort, quicksort, and heapsort — that use divide-and-conquer to achieve $O(n \log n)$ time.

5.10 Exercises

Exercise 3.1. Write a recursive version of binary search. What is its space complexity? Compare it with the iterative version presented in this chapter.

Exercise 3.2. The *Tower of Hanoi* puzzle has n disks of decreasing size stacked on one of three pegs. The goal is to move all disks to another peg, moving one disk at a time, never placing a larger disk on a smaller one. Write a recursive function `hanoi(n: number, from: string, to: string, via: string): void` that prints the moves. What is the time complexity? Prove that $2^n - 1$ moves are both necessary and sufficient.

Exercise 3.3. Implement a recursive version of the `pow` function (exponentiation by squaring). Analyze its space complexity and compare it with the iterative version.

Exercise 3.4. The *maximum subarray problem* asks for a contiguous subarray of an array of numbers with the largest sum. Design an $O(n \log n)$ divide-and-conquer algorithm for this problem. (Hint: split the array in half; the maximum subarray is entirely in the left half, entirely in the right half, or crossing the midpoint.)

Exercise 3.5. Karatsuba's algorithm multiplies two n-digit numbers using the recurrence $T(n) = 3T(n/2) + O(n)$. Use the Master Theorem to determine its time complexity. How does this compare with the naive $O(n^2)$ multiplication algorithm?

Chapter 6

Elementary Sorting

Sorting is one of the most fundamental problems in computer science. In this chapter we define the sorting problem precisely, introduce the concepts of stability and in-place sorting, and study three elementary sorting algorithms — bubble sort, selection sort, and insertion sort. All three run in $O(n^2)$ time in the worst case, but they differ in important ways: their behavior on nearly sorted input, their stability properties, and their practical performance. We close the chapter by proving that any comparison-based sorting algorithm must make $\Omega(n \log n)$ comparisons in the worst case — a lower bound that the elementary algorithms do not achieve, motivating the efficient algorithms of Chapter 5.

6.1 The sorting problem

Sorting is the problem of rearranging a collection of elements into a specified order. It arises constantly in practice — in database queries, in preparing data for binary search, in eliminating duplicates, in scheduling, and in countless other contexts. Knuth devoted an entire volume of *The Art of Computer Programming* to sorting and searching, calling sorting "perhaps the most deeply studied problem in computer science."

Definition 4.1 — The sorting problem

Input: A sequence of n elements $\langle a_1, a_2, \ldots, a_n \rangle$ and a total ordering $\leq$ on the elements.

Output: A permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

In TypeScript, we express the ordering through a *comparator function*:

```
export type Comparator<T> = (a: T, b: T) => number;
```

The comparator returns a negative number if $a < b$, zero if $a = b$, and a positive number if $a > b$. For numbers in ascending order, the comparator is simply:

```
export const numberComparator: Comparator<number> = (a, b) => a - b;
```

All three sorting algorithms in this chapter accept an optional comparator, defaulting to `numberComparator`. This makes them generic: they can sort arrays of any type, provided an appropriate comparator is supplied.

6.2 Stability

When a sequence contains elements that compare as equal, there is a choice: should the algorithm preserve the original relative order of equal elements, or is any arrangement of equal elements acceptable?

Definition 4.2 — Stable sort

A sorting algorithm is **stable** if, whenever two elements a_i and a_j satisfy $a_i = a_j$ and $i < j$ in the input, then a_i appears before a_j in the output.

Stability matters when elements carry additional data beyond the sort key. For example, suppose we sort a list of students by grade, and two students — Alice and Bob — both have a grade of 90. If Alice appeared before Bob in the original list, a stable sort guarantees she still appears before Bob in the sorted output. An unstable sort might swap them.

Stability also enables *multi-key sorting* by composition: to sort by last name and then by first name, we first sort by first name (using a stable sort), then sort by last name (using a stable sort). The second sort preserves the relative order established by the first sort within each group of equal last names.

Of the three algorithms in this chapter, **bubble sort** and **insertion sort** are stable, while **selection sort** is not.

6.3 In-place sorting

Definition 4.3 — In-place sort

A sorting algorithm is **in-place** if it uses $O(1)$ auxiliary space — that is, a constant amount of memory beyond the input array.

All three algorithms in this chapter are inherently in-place: they sort by swapping and shifting elements within the array, using only a constant number of temporary variables. Our TypeScript implementations copy the input array before sorting (to avoid mutating the caller's data), which adds $O(n)$ auxiliary space for the copy. The sorting logic itself, however, operates in-place on this copy.

6.4 Bubble sort

Bubble sort is perhaps the simplest sorting algorithm. It works by repeatedly scanning the array from left to right, swapping adjacent elements that are out of order. After each complete pass, the largest unsorted element has "bubbled" to its correct position at the end. The process repeats until no swaps are needed, meaning the array is sorted.

6.4.1 The algorithm

1. Repeat the following until no swap occurs during a complete pass:
    - For $i = 1, 2, \ldots, n-1$:
        - If $a[i-1] > a[i]$, swap $a[i-1]$ and $a[i]$.

6.4.2 Implementation

```
export function bubbleSort<T>(
  elements: T[],
  comparator: Comparator<T> = numberComparator as Comparator<T>,
): T[] {
  const copy = elements.slice(0);
  let wasSwapped = true;

  while (wasSwapped) {
    wasSwapped = false;
    for (let i = 1; i < copy.length; i++) {
      if (comparator(copy[i - 1]!, copy[i]!) > 0) {
        const temp = copy[i - 1]!;
        copy[i - 1] = copy[i]!;
        copy[i] = temp;
```

```
        wasSwapped = true;
      }
    }
  }
  return copy;
}
```

The `wasSwapped` flag is an optimization: if a complete pass makes no swaps, the array is already sorted and we can stop early.

6.4.3 Tracing through an example

Let us sort $A = [5, 3, 8, 4, 2]$.

Pass 1:

i	Array before	Comparison	Action	Array after
1	[**5**, **3**, 8, 4, 2]	5 > 3? Yes	Swap	[3, 5, 8, 4, 2]
2	[3, **5**, **8**, 4, 2]	5 > 8? No	—	[3, 5, 8, 4, 2]
3	[3, 5, **8**, **4**, 2]	8 > 4? Yes	Swap	[3, 5, 4, 8, 2]
4	[3, 5, 4, **8**, **2**]	8 > 2? Yes	Swap	[3, 5, 4, 2, 8]

After pass 1, the largest element (8) is in its final position.

Pass 2:

i	Array before	Comparison	Action	Array after
1	[**3**, **5**, 4, 2, 8]	3 > 5? No	—	[3, 5, 4, 2, 8]
2	[3, **5**, **4**, 2, 8]	5 > 4? Yes	Swap	[3, 4, 5, 2, 8]
3	[3, 4, **5**, **2**, 8]	5 > 2? Yes	Swap	[3, 4, 2, 5, 8]
4	[3, 4, 2, **5**, **8**]	5 > 8? No	—	[3, 4, 2, 5, 8]

After pass 2, the second-largest element (5) is in place.

Pass 3:

i	Array before	Comparison	Action	Array after
1	[**3**, **4**, 2, 5, 8]	3 > 4? No	—	[3, 4, 2, 5, 8]
2	[3, **4**, **2**, 5, 8]	4 > 2? Yes	Swap	[3, 2, 4, 5, 8]
3	[3, 2, **4**, **5**, 8]	4 > 5? No	—	[3, 2, 4, 5, 8]
4	[3, 2, 4, **5**, **8**]	5 > 8? No	—	[3, 2, 4, 5, 8]

Pass 4:

i	Array before	Comparison	Action	Array after
1	[**3**, **2**, 4, 5, 8]	3 > 2? Yes	Swap	[2, 3, 4, 5, 8]
2	[2, **3**, **4**, 5, 8]	3 > 4? No	—	[2, 3, 4, 5, 8]
3	[2, 3, **4**, **5**, 8]	4 > 5? No	—	[2, 3, 4, 5, 8]
4	[2, 3, 4, **5**, **8**]	5 > 8? No	—	[2, 3, 4, 5, 8]

Pass 5: No swaps occur → `wasSwapped` remains `false` → algorithm terminates.

Result: $[2, 3, 4, 5, 8]$.

6.4.4 Correctness

We prove correctness using the following loop invariant for the outer loop.

Invariant: After k complete passes, the k largest elements are in their correct final positions at the end of the array, and the algorithm has not changed the relative order of equal elements.

Initialization: Before any passes ($k = 0$), the invariant holds trivially — zero elements are known to be in their final positions.

Maintenance: Consider pass $k + 1$. The inner loop scans from left to right, swapping adjacent out-of-order pairs. The largest element in the unsorted prefix "bubbles" rightward through every comparison, because it is larger than (or equal to) every element it encounters. By the end of the pass, this element has reached position $n-k-1$, which is its correct final position. The swap condition uses strict inequality (>), so equal elements are never swapped — preserving stability.

Termination: The outer loop terminates when a pass makes no swaps, which means the entire array is sorted. In the worst case, $n-1$ passes are needed (when the smallest element starts at the end). By the invariant, after each pass one more element is correctly placed. After $n - 1$ passes, all elements are correctly placed. □

6.4.5 Complexity analysis

Worst case. The worst case occurs when the array is in reverse order. Pass k performs $n-1$ comparisons (our implementation always scans the full remaining array). In the worst case, $n - 1$ passes are needed, giving:

$$T_{\text{worst}}(n) = (n-1) \cdot (n-1) = (n-1)^2 = \Theta(n^2).$$

More precisely, with the optimization of reducing the scan range after each pass (which our implementation does not include), the comparison count would be $\sum_{k=1}^{n-1}(n-k) = n(n-1)/2$, still $\Theta(n^2)$.

Best case. The best case occurs when the array is already sorted. The first pass makes $n-1$ comparisons with no swaps, and the algorithm terminates:

$$T_{\text{best}}(n) = n - 1 = \Theta(n).$$

Average case. On average, bubble sort still performs $\Theta(n^2)$ comparisons and swaps.

Space complexity. $O(1)$ auxiliary space for the in-place sorting logic (plus $O(n)$ for the input copy in our implementation).

6.4.6 Properties

Property	Bubble sort
Worst-case time	$\Theta(n^2)$
Best-case time	$\Theta(n)$
Average-case time	$\Theta(n^2)$
Space	$O(1)$ in-place
Stable	Yes

6.5 Selection sort

Selection sort takes a different approach: instead of bubbling elements rightward, it repeatedly finds the minimum element from the unsorted portion and places it at the beginning.

6.5.1 The algorithm

1. For $i = 0, 1, \ldots, n-2$:
   - Find the index j of the minimum element in $a[i..n-1]$.
   - Swap $a[i]$ and $a[j]$.

After iteration i, the first $i+1$ positions contain the $i+1$ smallest elements in sorted order.

6.5.2 Implementation

```
export function selectionSort<T>(
  elements: T[],
  comparator: Comparator<T> = numberComparator as Comparator<T>,
): T[] {
  const copy = elements.slice(0);

  for (let i = 0; i < copy.length - 1; i++) {
    let remainingMinimum = copy[i]!;
    let indexToSwap = -1;

    for (let j = i + 1; j < copy.length; j++) {
      if (comparator(copy[j]!, remainingMinimum) < 0) {
        remainingMinimum = copy[j]!;
        indexToSwap = j;
      }
    }
    if (indexToSwap >= 0) {
      copy[indexToSwap] = copy[i]!;
      copy[i] = remainingMinimum;
    }
  }
  return copy;
}
```

6.5.3 Tracing through an example

Let us sort $A = [29, 10, 14, 37, 13]$.

i	Unsorted portion	Minimum	Swap	Array after
0	[**29**, 10, 14, 37, 13]	10 (index 1)	Swap $a[0]$ and $a[1]$	[10, 29, 14, 37, 13]
1	[10, **29**, 14, 37, 13]	13 (index 4)	Swap $a[1]$ and $a[4]$	[10, 13, 14, 37, 29]
2	[10, 13, **14**, 37, 29]	14 (index 2)	No swap needed	[10, 13, 14, 37, 29]

i	Unsorted portion	Minimum	Swap	Array after
3	[10, 13, 14, **37**, 29]	29 (index 4)	Swap $a[3]$ and $a[4]$	[10, 13, 14, 29, 37]

Result: $[10, 13, 14, 29, 37]$.

6.5.4 Correctness

Invariant: After iteration i of the outer loop, the subarray $a[0..i]$ contains the $i+1$ smallest elements of the original array, in sorted order, and the remaining elements in $a[i+1..n-1]$ are all greater than or equal to $a[i]$.

Initialization: Before the first iteration ($i = 0$), the sorted prefix is empty. The invariant holds vacuously.

Maintenance: In iteration i, the inner loop scans $a[i..n-1]$ and finds the minimum element. This element is the smallest among all elements not yet in the sorted prefix (since, by the invariant, all smaller elements are already in $a[0..i-1]$). Swapping it into position i extends the sorted prefix by one element, maintaining the invariant.

Termination: After $n-1$ iterations, positions 0 through $n-2$ contain the $n-1$ smallest elements in order. The remaining element at position $n-1$ is necessarily the largest, so the entire array is sorted. □

6.5.5 Why selection sort is not stable

Consider the array $[2_a, 2_b, 1]$, where 2_a and 2_b are equal values distinguished by subscripts to track their original positions. In the first iteration, selection sort finds the minimum (1, at index 2) and swaps it with $a[0]$:

$$[2_a, 2_b, 1] \xrightarrow{\text{swap } a[0] \leftrightarrow a[2]} [1, 2_b, 2_a]$$

Now 2_b appears before 2_a, but in the original array 2_a appeared first. The relative order of equal elements has been reversed. This happens because the swap moves 2_a past 2_b in a single step, without regard for their original order.

6.5.6 Complexity analysis

The inner loop in iteration i performs $n-i-1$ comparisons. The total number of comparisons is:

$$\sum_{i=0}^{n-2}(n-i-1)=\sum_{k=1}^{n-1}k=\frac{n(n-1)}{2}=\Theta(n^2).$$

This count is the same regardless of the input — selection sort always performs exactly $n(n-1)/2$ comparisons, whether the array is sorted, reverse-sorted, or random.

Swaps. Selection sort performs at most $n-1$ swaps (one per outer-loop iteration). This is a notable advantage: if swaps are expensive (for example, when array elements are large objects), selection sort minimizes data movement.

Space complexity. $O(1)$ auxiliary space for the in-place sorting logic.

6.5.7 Properties

Property	Selection sort
Worst-case time	$\Theta(n^2)$
Best-case time	$\Theta(n^2)$
Average-case time	$\Theta(n^2)$
Space	$O(1)$ in-place
Stable	No

6.6 Insertion sort

Insertion sort is the algorithm most people use intuitively when sorting a hand of playing cards. We hold the sorted cards in our left hand and pick up one card at a time from the table with our right hand, inserting it into the correct position among the already-sorted cards.

6.6.1 The algorithm

1. For $i = 1, 2, \ldots, n-1$:
   - Let key $= a[i]$.
   - Insert key into the sorted subarray $a[0..i-1]$ by shifting larger elements one position to the right.

6.6.2 Implementation

```
export function insertionSort<T>(
  elements: T[],
```

```
  comparator: Comparator<T> = numberComparator as Comparator<T>,
): T[] {
  const copy = elements.slice(0);

  for (let i = 1; i < copy.length; i++) {
    const toInsert = copy[i]!;
    let insertIndex = i - 1;

    while (insertIndex >= 0 && comparator(toInsert, copy[insertIndex]!) < 0) {
      copy[insertIndex + 1] = copy[insertIndex]!;
      insertIndex--;
    }
    insertIndex++;
    copy[insertIndex] = toInsert;
  }
  return copy;
}
```

The inner `while` loop shifts elements rightward until it finds the correct position for `toInsert`. The use of strict less-than (`< 0`) in the comparator check means that equal elements are not shifted past each other, which makes the algorithm stable.

6.6.3 Tracing through an example

Let us sort $A = [5, 2, 4, 6, 1, 3]$.

i	`toInsert`	Sorted prefix before	Shifts	Sorted prefix after
1	2	[**5**]	Shift 5 right	[2, 5]
2	4	[2, **5**]	Shift 5 right	[2, 4, 5]
3	6	[2, 4, 5]	None (6 ≥ 5)	[2, 4, 5, 6]
4	1	[**2, 4, 5, 6**]	Shift all four right	[1, 2, 4, 5, 6]
5	3	[1, **2**, **4, 5, 6**]	Shift 4, 5, 6 right	[1, 2, 3, 4, 5, 6]

Result: $[1, 2, 3, 4, 5, 6]$.

Notice how each element is inserted into its correct position within the growing sorted prefix on the left. When the element is already in the right place (like 6 in step 3), no shifting is needed and the inner loop exits immediately.

6.6.4 Correctness

Invariant: At the start of iteration i of the outer loop, the subarray $a[0..i-1]$ is a sorted permutation of the elements originally in those positions.

Initialization: Before the first iteration ($i = 1$), the subarray $a[0..0]$ contains a single element. A single element is trivially sorted.

Maintenance: During iteration i, the element $a[i]$ is removed from its position and inserted into the sorted subarray $a[0..i-1]$. The inner loop finds the correct insertion point by scanning rightward from position $i-1$ and shifting elements that are larger than $a[i]$. After the insertion, $a[0..i]$ is a sorted permutation of the elements originally in $a[0..i]$.

Termination: When $i = n$, the entire array $a[0..n-1]$ is sorted. □

6.6.5 Complexity analysis

The number of comparisons depends on the input.

Worst case. The worst case is a reverse-sorted array. In iteration i, the element must be shifted past all i elements in the sorted prefix, requiring i comparisons. The total is:

$$T_{\text{worst}}(n) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \Theta(n^2).$$

Best case. The best case is an already-sorted array. In each iteration, the inner loop performs one comparison (finding that `toInsert` is already in place) and zero shifts:

$$T_{\text{best}}(n) = \sum_{i=1}^{n-1} 1 = n - 1 = \Theta(n).$$

This is remarkable: insertion sort runs in *linear* time on sorted input, matching the theoretical minimum for any algorithm that must verify sortedness.

Average case. On a random permutation, each element is, on average, shifted past half the elements in the sorted prefix:

$$T_{\text{avg}}(n) = \sum_{i=1}^{n-1} \frac{i}{2} = \frac{n(n-1)}{4} = \Theta(n^2).$$

Nearly sorted input. If each element is at most k positions from its sorted position, the inner loop performs at most k comparisons per element, giving $T(n) = O(nk)$. When k is a small constant, insertion sort runs in linear time. This makes it an excellent choice for "nearly sorted" data and for finishing off the work of a more sophisticated algorithm (for example, some quicksort implementations switch to insertion sort for small subarrays).

Space complexity. $O(1)$ auxiliary space for the in-place sorting logic.

6.6.6 Inversions

The performance of insertion sort is closely tied to the concept of *inversions*.

Definition 4.4 — Inversion

An **inversion** in a sequence $\langle a_1, a_2, \ldots, a_n \rangle$ is a pair (i, j) with $i < j$ and $a_i > a_j$.

Each swap (or shift) in insertion sort eliminates exactly one inversion. Therefore, the number of comparisons insertion sort makes is $\Theta(n + I)$, where I is the number of inversions in the input. A sorted array has $I = 0$ inversions; a reverse-sorted array has $I = n(n-1)/2$, the maximum possible. On average, a random permutation has $I = n(n-1)/4$ inversions.

This connection makes insertion sort the natural choice when we know the input has few inversions — it is *adaptive* to the presortedness of the input.

6.6.7 Properties

Property	Insertion sort
Worst-case time	$\Theta(n^2)$
Best-case time	$\Theta(n)$
Average-case time	$\Theta(n^2)$
Space	$O(1)$ in-place
Stable	Yes
Adaptive	Yes (time depends on inversions)

6.7 Comparison of elementary sorts

Now that we have studied all three algorithms, let us compare them side by side.

Property	Bubble sort	Selection sort	Insertion sort
Worst-case time	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Best-case time	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n)$
Average-case time	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Stable	Yes	No	Yes
Adaptive	Yes	No	Yes
Comparisons (worst)	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Swaps (worst)	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$ shifts

Several observations stand out:

- **Selection sort** always does the same amount of work regardless of the input — it is not adaptive. However, it minimizes the number of swaps ($O(n)$), which matters when moving elements is expensive.
- **Insertion sort** is the best general-purpose choice among the three. It is stable, adaptive, and efficient on small or nearly sorted inputs. In practice, it outperforms both bubble sort and selection sort.
- **Bubble sort** is adaptive (like insertion sort), but in practice it is slower because it performs more data movement per inversion — elements move only one position per swap, while insertion sort shifts an entire block. Bubble sort's main virtue is pedagogical simplicity.

6.8 The comparison-based sorting lower bound

All three elementary sorting algorithms are *comparison-based*: they access the input elements only through pairwise comparisons. Can we do better than $O(n^2)$ with a comparison-based algorithm? The answer is yes — merge sort, heapsort, and quicksort achieve $O(n \log n)$ time, as we will see in Chapter 5. But can we do better than $O(n \log n)$? The answer is no.

Theorem 4.1 — Comparison-based sorting lower bound

Any comparison-based sorting algorithm must make at least $\lceil \log_2(n!) \rceil = \Omega(n \log n)$ comparisons in the worst case to sort n elements.

6.8.1 The decision tree argument

To prove this theorem, we model any comparison-based sorting algorithm as a *decision tree*. Each internal node represents a comparison between two elements (e.g., "is $a_i \le a_j$?"), with two children corresponding to the outcomes "yes" and "no." Each leaf represents a specific output permutation.

For the algorithm to be correct, it must be able to produce every permutation of n elements as output — there must be at least $n!$ leaves. The number of comparisons in the worst case equals the height of the decision tree (the longest root-to-leaf path).

A binary tree of height h has at most 2^h leaves. For the tree to have at least $n!$ leaves:

$$2^h \ge n!$$

Taking logarithms:

$$h \ge \log_2(n!)$$

Using Stirling's approximation, $n! \approx \sqrt{2\pi n}\,(n/e)^n$, we get:

$$\log_2(n!) = \Theta(n \log n).$$

More concretely:

$$\log_2(n!) = \sum_{k=1}^{n} \log_2 k \ge \sum_{k=n/2}^{n} \log_2 k \ge \frac{n}{2} \cdot \log_2 \frac{n}{2} = \frac{n}{2}(\log_2 n - 1) = \Omega(n \log n).$$

Therefore, any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons in the worst case. □

6.8.2 Implications

This lower bound tells us that $O(n \log n)$ algorithms like merge sort and heapsort are *asymptotically optimal* among comparison-based sorts — they cannot be improved in the worst case.

It also tells us that our elementary $O(n^2)$ algorithms are a factor of $n/\log n$ away from optimal. For $n = 1{,}000{,}000$, that factor is roughly 50,000 — the same dramatic gap we noted in the growth-rate table of Chapter 2.

However, the lower bound applies only to comparison-based sorting. Algorithms that exploit additional structure in the input (such as knowing that elements are integers in a bounded range) can sort in $O(n)$ time, as we will see in Chapter 6.

6.9 Looking ahead

In this chapter we studied the sorting problem and three elementary algorithms for solving it:

- **Bubble sort** repeatedly swaps adjacent out-of-order elements. It is simple and stable, with $O(n)$ best-case time, but $O(n^2)$ on average and in the worst case.
- **Selection sort** repeatedly selects the minimum from the unsorted portion. It always takes $\Theta(n^2)$ time but minimizes swaps to $O(n)$. It is not stable.
- **Insertion sort** inserts each element into its correct position in a growing sorted prefix. It is stable, adaptive to the number of inversions, and has $O(n)$ best-case time. It is the practical choice among elementary sorts.
- The **comparison-based lower bound** of $\Omega(n \log n)$ shows that these quadratic algorithms are not optimal.

In Chapter 5, we study three efficient sorting algorithms — merge sort, quicksort, and heapsort — that achieve the $O(n \log n)$ bound. These algorithms use the divide-and-conquer strategy from Chapter 3 to overcome the quadratic barrier.

6.10 Exercises

Exercise 4.1. Trace through bubble sort on the input $[6, 4, 1, 8, 3]$. How many passes are needed? How many total swaps?

Exercise 4.2. Our bubble sort implementation scans the entire array on each pass. Modify the algorithm so that pass k scans only positions 0 through $n-k-1$ (since the last k elements are already in place). Does this change the worst-case asymptotic complexity? Does it improve the constant factor?

Exercise 4.3. Give a concrete example showing that selection sort is not stable. Then describe how selection sort could be modified to become stable (hint: use insertion into a separate output instead of swapping). What is the cost of this modification?

Exercise 4.4. Prove that insertion sort performs exactly $n + I - 1$ comparisons on an input with I inversions (assuming the inner loop always does one comparison to confirm the insertion point even when no shifting is needed). Use this to show that insertion sort is $O(n)$ on inputs with $O(n)$ inversions.

Exercise 4.5. A *sentinel* version of insertion sort places a minimum element at position $a[0]$ before sorting, eliminating the `insertIndex >= 0` bound check in the inner loop. Explain why this is correct and analyze its effect on performance. What are the drawbacks?

Chapter 7

Efficient Sorting

In Chapter 4 we proved that any comparison-based sorting algorithm must make $\Omega(n \log n)$ comparisons in the worst case. The three elementary algorithms we studied — bubble sort, selection sort, and insertion sort — fall short of this bound, requiring $\Theta(n^2)$ time. In this chapter we meet three algorithms that close the gap: merge sort, quicksort, and heapsort. All three achieve $O(n \log n)$ time and are, in different senses, asymptotically optimal. They use the divide-and-conquer strategy from Chapter 3, but apply it in very different ways — merge sort divides trivially and combines carefully, quicksort divides carefully and combines trivially, and heapsort uses a heap data structure to repeatedly extract the maximum. We also study randomized quicksort, which uses random pivot selection to guarantee expected $O(n \log n)$ performance on every input.

7.1 Merge sort

Merge sort is the most straightforward application of divide-and-conquer to sorting. The idea is simple: split the array in half, recursively sort each half, and then merge the two sorted halves into a single sorted array.

7.1.1 The algorithm

1. If the array has zero or one elements, it is already sorted. Return.
2. Divide the array into two halves of roughly equal size.
3. Recursively sort each half.
4. Merge the two sorted halves into a single sorted array.

The key insight is that merging two sorted arrays of total length n takes $O(n)$ time: we scan both arrays from left to right, always taking the smaller of the two current elements.

7.1.2 The merge procedure

The merge step is the heart of the algorithm. Given an array `arr` and indices `start`, `middle`, and `end`, we merge the sorted subarrays `arr[start..middle)` and `arr[middle..end)` into a single sorted subarray `arr[start..end)`.

```
export function merge<T>(
  arr: T[],
  start: number,
  middle: number,
  end: number,
  comparator: Comparator<T> = numberComparator as Comparator<T>,
): void {
  const sorted: T[] = [];
  let i = start;
  let j = middle;

  while (i < middle && j < end) {
    if (comparator(arr[i]!, arr[j]!) <= 0) {
      sorted.push(arr[i]!);
      i++;
    } else {
      sorted.push(arr[j]!);
      j++;
    }
  }
  while (i < middle) {
    sorted.push(arr[i]!);
    i++;
  }
  while (j < end) {
    sorted.push(arr[j]!);
    j++;
  }

  i = start;
  while (i < end) {
    arr[i] = sorted[i - start]!;
    i++;
  }
}
```

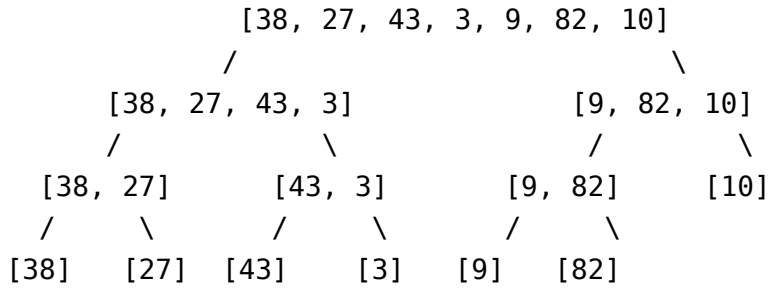
The comparison `<= 0` (rather than `< 0`) ensures *stability*: when two elements are

equal, the one from the left subarray comes first, preserving original order.

7.1.3 Tracing through an example

Let us sort $A = [38, 27, 43, 3, 9, 82, 10]$.

Divide phase (conceptual; our bottom-up implementation avoids this):



Merge phase:

Step	Left	Right	Merged
1	[38]	[27]	[27, 38]
2	[43]	[3]	[3, 43]
3	[27, 38]	[3, 43]	[3, 27, 38, 43]
4	[9]	[82]	[9, 82]
5	[9, 82]	[10]	[9, 10, 82]
6	[3, 27, 38, 43]	[9, 10, 82]	[3, 9, 10, 27, 38, 43, 82]

Result: $[3, 9, 10, 27, 38, 43, 82]$.

7.1.4 Bottom-up implementation

The classic recursive merge sort divides the array top-down and merges bottom-up. An equivalent approach is to skip the divide phase entirely and work bottom-up from the start: first merge pairs of single elements into sorted pairs, then merge pairs of pairs into sorted 4-element runs, and so on, doubling the run length each time.

```
export function mergeSort<T>(
  elements: T[],
  comparator: Comparator<T> = numberComparator as Comparator<T>,
): T[] {
  const copy = elements.slice(0);
  let step = 1;
```

```
  while (step < copy.length) {
    step = step * 2;
    for (let start = 0; start < copy.length; start = start + step) {
      const middle = Math.min(start + step / 2, copy.length);
      const end = Math.min(start + step, copy.length);

      merge(copy, start, middle, end, comparator);
    }
  }
  return copy;
}
```

The bottom-up approach has the same time complexity as the recursive version but avoids the $O(\log n)$ recursion stack overhead.

7.1.5 Correctness

Claim. The merge procedure correctly merges two sorted subarrays.

At each step of the main loop, we choose the smaller of the two current front elements. Since both subarrays are sorted, the current front element of each is the smallest remaining element in that subarray. Therefore, the smaller of the two fronts is the smallest remaining element overall. After the main loop, one subarray is exhausted and we append the remainder of the other (which is already sorted). The result is a sorted permutation of all elements from both subarrays. The `<= 0` comparison ensures that equal elements from the left subarray come first, preserving stability.

Claim. Merge sort correctly sorts the array.

We argue by induction on the run length. In the first iteration (`step = 2`), each merge operates on runs of length 1, which are trivially sorted. Each merge produces a sorted run of length 2. In each subsequent iteration, the runs from the previous iteration are sorted (by the inductive hypothesis), and the merge procedure correctly combines pairs of sorted runs into longer sorted runs. After $\lceil \log_2 n \rceil$ iterations, the entire array is a single sorted run. □

7.1.6 Complexity analysis

Time. At each level of the merge tree, the total work across all merges is $O(n)$ (each element is compared and copied once). The number of levels is $\lceil \log_2 n \rceil$. Therefore:

$$T(n) = O(n \log n).$$

This holds in the best case, worst case, and average case — merge sort is *not* adaptive to the input's presortedness.

The same result follows from the recurrence for the recursive version:

$$T(n) = 2T(n/2) + O(n), \quad T(1) = O(1).$$

By the Master Theorem (case 2, with $a = 2$, $b = 2$, $f(n) = O(n)$), we get $T(n) = O(n \log n)$.

Space. The merge procedure uses an auxiliary array of size up to n to hold merged elements. Combined with the $O(n)$ copy of the input, the total space is $O(n)$. The bottom-up version uses no recursion stack; the recursive version would add $O(\log n)$ stack frames.

7.1.7 Properties

Property	Merge sort
Worst-case time	$\Theta(n \log n)$
Best-case time	$\Theta(n \log n)$
Average-case time	$\Theta(n \log n)$
Space	$O(n)$ auxiliary
Stable	Yes
Adaptive	No

7.2 Quicksort

Quicksort, invented by Tony Hoare in 1959, takes the opposite approach from merge sort. Where merge sort divides trivially (split in half) and combines carefully (merge), quicksort divides carefully (partition) and combines trivially (the subarrays are already in the right place).

The idea: choose a *pivot* element, rearrange the array so that all elements less than the pivot come before it and all elements greater come after it, then recursively sort the two partitions.

7.2.1 The partition procedure

The partition step rearranges `arr[start..end]` around a pivot element and returns the pivot's final index. After partitioning: - All elements to the left of the pivot are ≤ the pivot. - All elements to the right are ≥ the pivot. - The pivot is in its correct final position.

Our implementation chooses the middle element as the pivot, then uses the Lomuto partition scheme: scan from left, moving elements smaller than the pivot to the front.

```
export function partition<T>(
  arr: T[],
  start: number,
  end: number,
  comparator: Comparator<T> = numberComparator as Comparator<T>,
): number | undefined {
  if (start > end || end >= arr.length || start < 0 || end < 0) {
    return undefined;
  }

  const middleIndex = Math.floor((start + end) / 2);
  let storeIndex = start;

  // Move pivot to end
  const pivotTemp = arr[middleIndex]!;
  arr[middleIndex] = arr[end]!;
  arr[end] = pivotTemp;

  for (let i = start; i < end; i++) {
    if (comparator(arr[i]!, arr[end]!) < 0) {
      const temp = arr[storeIndex]!;
      arr[storeIndex] = arr[i]!;
      arr[i] = temp;
      storeIndex++;
    }
  }

  // Move pivot to its final position
  const temp = arr[storeIndex]!;
  arr[storeIndex] = arr[end]!;
  arr[end] = temp;
```

```
  return storeIndex;
}
```

The pivot is first swapped to the end, then `storeIndex` tracks the boundary between elements known to be less than the pivot and elements not yet examined. After the scan, the pivot is swapped into `storeIndex`, its correct position.

7.2.2 Tracing through an example

Let us sort $A = [7, 2, 1, 6, 8, 5, 3, 4]$ with middle-element pivot selection.

First partition (full array, indices 0–7):

The middle index is $\lfloor (0 + 7)/2 \rfloor = 3$, so the pivot is $A[3] = 6$. Swap it to the end:

$[7, 2, 1, 4, 8, 5, 3, \underline{6}]$

Scan with `storeIndex = 0`:

i	$A[i]$	$A[i] < 6$?	Action	`storeIndex`
0	7	No	—	0
1	2	Yes	Swap $A[0]$ and $A[1]$	1
2	1	Yes	Swap $A[1]$ and $A[2]$	2
3	4	Yes	Swap $A[2]$ and $A[3]$	3
4	8	No	—	3
5	5	Yes	Swap $A[3]$ and $A[5]$	4
6	3	Yes	Swap $A[4]$ and $A[6]$	5

Place pivot at `storeIndex = 5`:

$[2, 1, 4, 5, 3, \underline{6}, 7, 8]$

Now 6 is in its final position. Recursively sort $[2, 1, 4, 5, 3]$ (indices 0–4) and $[7, 8]$ (indices 6–7).

The recursion continues, each time placing one element in its final position, until the base cases (subarrays of size 0 or 1) are reached.

7.2.3 Implementation

```
function sort<T>(
  arr: T[],
  start: number,
  end: number,
  comparator: Comparator<T>,
```

```
): void {
  if (start < end) {
    const partitionIndex = partition(arr, start, end, comparator)!;
    sort(arr, start, partitionIndex - 1, comparator);
    sort(arr, partitionIndex + 1, end, comparator);
  }
}

export function quickSort<T>(
  elements: T[],
  comparator: Comparator<T> = numberComparator as Comparator<T>,
): T[] {
  const copy = elements.slice(0);

  sort(copy, 0, copy.length - 1, comparator);
  return copy;
}
```

7.2.4 Correctness

Claim. After `partition(arr, start, end)`, the pivot is in its correct final sorted position.

The partition loop moves all elements less than the pivot to positions before `storeIndex`, and leaves elements greater than or equal to the pivot after `storeIndex`. The pivot is then placed at `storeIndex`. Every element before it is smaller, every element after it is at least as large — this is exactly where the pivot belongs in the sorted output.

Claim. Quicksort correctly sorts the array.

By induction on the subarray size. Subarrays of size 0 or 1 are trivially sorted (base case). For a subarray of size $k > 1$: partition places the pivot correctly, then quicksort recursively sorts the left subarray (elements $<$ pivot) and right subarray (elements $\geq$ pivot). By the inductive hypothesis, both recursive calls produce sorted subarrays. Since every element in the left subarray is $\leq$ pivot $\leq$ every element in the right subarray, the entire array is sorted. $\square$

7.2.5 Complexity analysis

The performance of quicksort depends on the quality of the partition — how evenly the pivot divides the array.

Best case. If the pivot always lands in the middle, each partition splits the array into two roughly equal halves. The recurrence is the same as merge sort:

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

Worst case. If the pivot always lands at one extreme (the smallest or largest element), one partition has $n - 1$ elements and the other has 0. The recurrence becomes:

$$T(n) = T(n-1) + O(n) = O(n^2).$$

This worst case occurs with our middle-element pivot when the input is specially constructed, and with the first-element or last-element pivot strategies on already-sorted or reverse-sorted input.

Average case. On a random permutation with any fixed pivot strategy, the expected running time is $O(n \log n)$. Intuitively, even moderately unbalanced partitions (say, 1:9 splits) only add a constant factor to the recursion depth: the shorter side shrinks by a factor of 10, and $\log_{10} n = O(\log n)$.

More precisely, if we assume each element is equally likely to be the pivot, the expected number of comparisons is:

$$\mathbb{E}[C(n)] = 2(n+1)H_n - 4n \approx 2n \ln n \approx 1.39\, n \log_2 n$$

where $H_n = \sum_{k=1}^{n} 1/k$ is the nth harmonic number. This is only about 39% more comparisons than merge sort's worst case of $n \log_2 n$.

Space. Quicksort sorts in place (aside from our defensive copy of the input). The recursion stack has depth $O(\log n)$ in the best case but $O(n)$ in the worst case. Tail-call optimization or explicit stack management can limit the worst-case stack depth to $O(\log n)$ by always recursing on the smaller partition first.

7.2.6 Properties

Property	Quicksort
Worst-case time	$\Theta(n^2)$
Best-case time	$\Theta(n \log n)$
Average-case time	$\Theta(n \log n)$
Space	$O(\log n)$ stack (in-place)
Stable	No
Adaptive	No

Property	Quicksort

7.2.7 Why quicksort is fast in practice

Despite its $O(n^2)$ worst case, quicksort is often the fastest comparison sort in practice. Several factors contribute:

1. **Cache friendliness.** Quicksort's partition scan accesses elements sequentially, which is excellent for CPU cache performance. Merge sort accesses two separate subarrays during merge, which can cause more cache misses.
2. **Small constant factor.** Quicksort performs fewer data movements than merge sort — partitioning swaps elements in place, while merging copies elements to an auxiliary array and back.
3. **No auxiliary memory.** Quicksort needs only $O(\log n)$ stack space, while merge sort needs $O(n)$ auxiliary space. Less memory allocation means less overhead.
4. **Adaptable.** In practice, quicksort implementations use optimizations like switching to insertion sort for small subarrays, choosing better pivots (median-of-three), and using three-way partitioning for inputs with many duplicates.

7.3 Heapsort

Heapsort uses a *binary heap* to sort an array in place. A binary heap is an array-based data structure that maintains a partial ordering — not fully sorted, but structured enough to find the maximum (or minimum) in $O(1)$ time and restore order in $O(\log n)$ time after a removal.

7.3.1 The binary heap

A *max-heap* is a complete binary tree stored in an array where every node's value is greater than or equal to its children's values. For a node at index i (zero-based):

- Left child: $2i + 1$
- Right child: $2i + 2$
- Parent: $\lfloor (i - 1)/2 \rfloor$

The max-heap property ensures that the root (index 0) holds the largest element.

7.3.2 Heapify

The `heapify` operation takes a node whose children are both valid max-heaps but whose own value may violate the heap property, and "sinks" it down to restore the property:

```
function heapify<T>(
  arr: T[],
  heapSize: number,
  index: number,
  comparator: Comparator<T>,
): void {
  const left = 2 * index + 1;
  const right = 2 * index + 2;
  let indexOfMaximum = index;

  for (const subTreeRootIndex of [left, right]) {
    if (
      subTreeRootIndex < heapSize &&
      comparator(arr[subTreeRootIndex]!, arr[indexOfMaximum]!) > 0
    ) {
      indexOfMaximum = subTreeRootIndex;
    }
  }
  if (indexOfMaximum !== index) {
    const temp = arr[index]!;
    arr[index] = arr[indexOfMaximum]!;
    arr[indexOfMaximum] = temp;
    heapify(arr, heapSize, indexOfMaximum, comparator);
  }
}
```

The element at `index` is compared with its children. If a child is larger, the element is swapped with the largest child, and the process repeats in that child's subtree. Each step moves down one level, so heapify runs in $O(\log n)$ time (the height of the tree).

7.3.3 Building a heap

We can convert an unordered array into a max-heap by calling `heapify` on every non-leaf node, bottom-up:

```
function buildHeap<T>(
  arr: T[],
```

```
  heapSize: number,
  comparator: Comparator<T>,
): void {
  const lastNonLeafIndex = Math.floor((heapSize + 1) / 2) - 1;

  for (let i = lastNonLeafIndex; i >= 0; i--) {
    heapify(arr, heapSize, i, comparator);
  }
}
```

Why bottom-up? The leaves (the bottom half of the array) are trivially valid heaps. By processing nodes from the bottom up, each call to `heapify` encounters a node whose children are already valid heaps — exactly the precondition heapify requires.

Why $O(n)$ and not $O(n \log n)$? A naive analysis says: $n/2$ calls to heapify, each costing $O(\log n)$, giving $O(n \log n)$. But this overestimates. Most nodes are near the bottom and sink only a few levels. The precise cost is:

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot O(h) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n \cdot 2) = O(n).$$

The series $\sum_{h=0}^{\infty} h/2^h = 2$ converges, so building a heap takes linear time.

7.3.4 The heapsort algorithm

1. Build a max-heap from the input array: $O(n)$.
2. Repeat for $i = n-1, n-2, \ldots, 1$:
   - Swap the root (maximum) with element i.
   - Reduce the heap size by 1 (element i is now in its final position).
   - Call heapify on the root to restore the heap property.

```
export function heapSort<T>(
  elements: T[],
  comparator: Comparator<T> = numberComparator as Comparator<T>,
): T[] {
  const arr = elements.slice(0);
  let heapSize = arr.length;

  buildHeap(arr, heapSize, comparator);
  for (let i = arr.length - 1; i > 0; i--) {
    const temp = arr[0]!;
```

```
    arr[0] = arr[i]!;
    arr[i] = temp;
    heapSize--;
    heapify(arr, heapSize, 0, comparator);
  }
  return arr;
}
```

7.3.5 Tracing through an example

Let us sort $A = [4, 10, 3, 5, 1]$.

Build max-heap:

Starting array (as a tree):

```
        4
       / \
     10    3
    / \
   5   1
```

Process non-leaf nodes bottom-up. Node at index 1 (value 10): children are 5, 1. 10 is already larger — no change. Node at index 0 (value 4): children are 10, 3. Swap 4 with 10. Then heapify the subtree: 4 vs children 5, 1 → swap with 5.

```
        10              10
       / \             / \
      4    3   →      5    3
     / \             / \
    5   1           4   1
```

Max-heap: $[10, 5, 3, 4, 1]$.

Extract-max loop:

Step	Swap	Array after swap	Heapify root	Result
1	$A[0] \leftrightarrow A[4]$	[1, 5, 3, 4, **10**]	[5, 4, 3, 1]	[5, 4, 3, 1, 10]
2	$A[0] \leftrightarrow A[3]$	[1, 4, 3, **5**, 10]	[4, 1, 3]	[4, 1, 3, 5, 10]
3	$A[0] \leftrightarrow A[2]$	[3, 1, **4**, 5, 10]	[3, 1]	[3, 1, 4, 5, 10]
4	$A[0] \leftrightarrow A[1]$	[1, **3**, 4, 5, 10]	[1]	[1, 3, 4, 5, 10]

Result: $[1, 3, 4, 5, 10]$.

7.3.6 Correctness

Invariant: At the start of each iteration i of the extract-max loop: - $A[0..i]$ is a max-heap containing the $i + 1$ smallest elements. - $A[i + 1..n - 1]$ contains the $n - i - 1$ largest elements, in sorted order.

Initialization. After `buildHeap`, the entire array is a max-heap and the sorted suffix is empty.

Maintenance. The root $A[0]$ is the largest element in the heap $A[0..i]$. Swapping it with $A[i]$ places it in the correct position (it is the $(i + 1)$th largest overall). Reducing the heap size and calling heapify restores the heap property on $A[0..i - 1]$.

Termination. When $i = 0$, the heap contains a single element (the minimum), which is trivially in its correct position. The array is sorted. □

7.3.7 Complexity analysis

Time. Building the heap takes $O(n)$. The extract-max loop runs $n-1$ times, each iteration performing a swap and a heapify costing $O(\log n)$. Total:

$$T(n) = O(n) + (n - 1) \cdot O(\log n) = O(n \log n).$$

This holds for all inputs — heapsort is not adaptive.

Space. Heapsort sorts in place. The only auxiliary space is $O(1)$ for temporary variables (plus $O(n)$ for our defensive copy).

7.3.8 Properties

Property	Heapsort
Worst-case time	$\Theta(n \log n)$
Best-case time	$\Theta(n \log n)$
Average-case time	$\Theta(n \log n)$
Space	$O(1)$ in-place
Stable	No
Adaptive	No

7.4 Randomized quicksort

Deterministic quicksort's performance depends on the pivot choice. A fixed strategy — first element, last element, middle element — can always be defeated by a carefully constructed input that forces $O(n^2)$ behavior. Randomized quicksort eliminates this vulnerability by choosing the pivot *uniformly at random*.

7.4.1 Motivation

Consider a sorting library used by millions of applications. An adversary who knows the pivot-selection strategy can craft inputs that trigger worst-case behavior, leading to denial-of-service attacks. By choosing the pivot randomly, we ensure that no input is consistently bad — the algorithm's expected performance is $O(n \log n)$ for *every* input, regardless of how it was constructed.

This is a powerful guarantee. It shifts the source of randomness from the input (which an adversary controls) to the algorithm (which the adversary cannot predict).

7.4.2 The algorithm

Randomized quicksort is identical to standard quicksort, except that the partition step selects a random element as the pivot instead of a fixed one:

```
function randomizedPartition<T>(
  arr: T[],
  start: number,
  end: number,
  comparator: Comparator<T>,
): number {
  // Choose a random pivot index in [start, end]
  const randomIndex = start + Math.floor(Math.random() * (end - start + 1));
  let storeIndex = start;

  // Move pivot to end
  const pivotTemp = arr[randomIndex]!;
  arr[randomIndex] = arr[end]!;
  arr[end] = pivotTemp;

  for (let i = start; i < end; i++) {
    if (comparator(arr[i]!, arr[end]!) < 0) {
      const temp = arr[storeIndex]!;
      arr[storeIndex] = arr[i]!;
```

```
      arr[i] = temp;
      storeIndex++;
    }
  }

  // Move pivot to its final position
  const temp = arr[storeIndex]!;
  arr[storeIndex] = arr[end]!;
  arr[end] = temp;

  return storeIndex;
}

export function randomizedQuickSort<T>(
  elements: T[],
  comparator: Comparator<T> = numberComparator as Comparator<T>,
): T[] {
  const copy = elements.slice(0);

  sort(copy, 0, copy.length - 1, comparator);
  return copy;
}
```

The only change from deterministic quicksort is the line that computes the pivot index: `Math.floor(Math.random() * (end - start + 1))` instead of `Math.floor((start + end) / 2)`.

7.4.3 Expected running time

Theorem 5.1. The expected number of comparisons made by randomized quicksort on any input of size n is at most $2n \ln n = O(n \log n)$.

Proof sketch. Let $z_1 < z_2 < \cdots < z_n$ be the elements of the input in sorted order. Define the indicator random variable X_{ij} as 1 if z_i and z_j are ever compared during the execution, and 0 otherwise.

The total number of comparisons is:

$$C = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}.$$

By linearity of expectation:

$$\mathbb{E}[C] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr[z_i \text{ and } z_j \text{ are compared}].$$

Now, z_i and z_j are compared if and only if one of them is chosen as the pivot before any element in $\{z_i, z_{i+1}, \ldots, z_j\}$. Since we choose pivots uniformly at random, the probability that z_i or z_j is chosen first among these $j - i + 1$ elements is $2/(j - i + 1)$.

Therefore:

$$\mathbb{E}[C] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} \leq \sum_{i=1}^{n-1} 2H_n = 2(n-1)H_n \leq 2n \ln n.$$

where $H_n = \sum_{k=1}^{n} 1/k \leq \ln n + 1$ is the nth harmonic number. □

This expected bound holds for *every* input — it is not an average over random inputs. Even on an adversarial input, randomized quicksort makes $O(n \log n)$ expected comparisons.

7.4.4 Worst case

The worst case of $O(n^2)$ still exists in theory: if the random choices happen to always pick the smallest or largest element as pivot. However, the probability of this occurring is astronomically small. For $n = 1000$, the probability of consistently terrible pivots through all recursive calls is effectively zero.

7.4.5 Properties

Property	Randomized quicksort
Worst-case time	$O(n^2)$ (extremely unlikely)
Expected time	$O(n \log n)$ for all inputs
Space	$O(\log n)$ expected stack depth
Stable	No

7.5 Comparison of efficient sorting algorithms

We have now studied four $O(n \log n)$ sorting algorithms. Let us compare them across the dimensions that matter in practice.

7.5.1 Time complexity

Algorithm	Best case	Average case	Worst case
Merge sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Quicksort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$
Randomized quicksort	$\Theta(n \log n)$	$O(n \log n)$ expected	$O(n^2)$
Heapsort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$

Merge sort and heapsort provide *guaranteed* $O(n \log n)$ performance. Quicksort has a theoretical $O(n^2)$ worst case, but randomization makes this practically irrelevant. In terms of constant factors, quicksort (including randomized) typically makes the fewest comparisons on average — about $1.39\, n \log_2 n$ versus merge sort's $n \log_2 n$ comparisons, but with lower overhead per comparison.

7.5.2 Space complexity

Algorithm	Auxiliary space
Merge sort	$O(n)$
Quicksort	$O(\log n)$ stack
Randomized quicksort	$O(\log n)$ expected stack
Heapsort	$O(1)$

Heapsort is the clear winner for space: it sorts truly in place with $O(1)$ extra memory. Quicksort needs $O(\log n)$ stack space (or $O(n)$ in the worst case without tail-call optimization). Merge sort needs $O(n)$ for the auxiliary merge array.

7.5.3 Stability

Algorithm	Stable?
Merge sort	Yes
Quicksort	No
Randomized quicksort	No
Heapsort	No

Merge sort is the only stable $O(n \log n)$ algorithm among the four. This makes it the default choice when stability is required — for example, in database sorting or when composing sorts on multiple keys.

7.5.4 Cache performance

Quicksort has the best cache performance among the four. Its partition scan accesses elements sequentially, making excellent use of CPU cache lines. Merge sort accesses two separate subarrays during merge, which can cause cache misses when the subarrays are far apart in memory. Heapsort has the worst cache performance: heap navigation accesses elements at indices i, $2i + 1$, and $2i + 2$, which jump around the array unpredictably for large arrays.

7.5.5 Practical recommendations

- **General-purpose sorting:** Randomized quicksort (or a tuned variant) is the standard choice. Most standard library sort functions (including V8's `Array.prototype.sort` for large arrays) are based on quicksort variants.
- **Guaranteed worst-case performance:** Use merge sort or heapsort. Merge sort is preferred when stability is needed; heapsort when memory is constrained.
- **Small arrays:** Insertion sort (from Chapter 4) outperforms all of the above for small arrays (typically $n < 10$–20) due to its minimal overhead. Practical quicksort implementations switch to insertion sort for small subarrays.
- **Hybrid algorithms:** The best practical sorts combine multiple algorithms. Timsort (Python, Java) combines merge sort with insertion sort. Introsort (C++ STL) starts with quicksort, switches to heapsort if the recursion depth exceeds $2 \log n$ (to guarantee $O(n \log n)$ worst case), and uses insertion sort for small subarrays.

7.6 Chapter summary

In this chapter we studied four efficient comparison-based sorting algorithms:

- **Merge sort** divides the array in half, sorts each half recursively, and merges the sorted halves. It runs in $\Theta(n \log n)$ time in all cases but requires $O(n)$ auxiliary space. It is stable.
- **Quicksort** partitions the array around a pivot, placing it in its correct position, then recursively sorts the two partitions. It runs in $O(n \log n)$ average time with excellent cache performance, but has $O(n^2)$ worst-case time with a fixed pivot strategy.
- **Heapsort** builds a max-heap and repeatedly extracts the maximum to build the sorted array from right to left. It runs in $\Theta(n \log n)$ time in all cases and uses $O(1)$ auxiliary space, but has poor cache performance.

- **Randomized quicksort** eliminates quicksort's vulnerability to adversarial inputs by choosing pivots uniformly at random. It achieves $O(n \log n)$ expected time on *every* input.

All four algorithms achieve the $\Omega(n \log n)$ lower bound proved in Chapter 4. In the next chapter, we explore a different question: can we sort *faster* than $O(n \log n)$ by using information beyond pairwise comparisons?

7.7 Exercises

Exercise 5.1. Trace through the merge sort algorithm on the input $[12, 11, 13, 5, 6, 7]$. Show the state of the array after each merge operation in the bottom-up approach.

Exercise 5.2. Merge sort's merge procedure uses $O(n)$ auxiliary space. Can we merge two sorted subarrays in place (using $O(1)$ extra space) while maintaining $O(n)$ time? Explain why this is difficult. (Hint: in-place merge algorithms exist, but they either sacrifice time complexity to $O(n \log n)$ or are extremely complex.)

Exercise 5.3. Consider quicksort with the "first element" pivot strategy. Give an input of size n that causes $\Theta(n^2)$ behavior. Then give a different input that causes $\Theta(n \log n)$ behavior. What input causes the worst case for the "middle element" strategy used in our implementation?

Exercise 5.4. Prove that the expected recursion depth of randomized quicksort is $O(\log n)$. (Hint: at each level, with constant probability the pivot falls in the middle half of the array. How many levels until the subproblem size drops to 1?)

Exercise 5.5. Heapsort is not stable. Give a concrete example of an array with duplicate values where heapsort changes the relative order of equal elements. Why does the "swap root with last element" step destroy stability?

Chapter 8

Linear-Time Sorting and Selection

In Chapter 4 we proved a lower bound: every comparison-based sorting algorithm must make $\Omega(n \log n)$ comparisons in the worst case. The efficient algorithms of Chapter 5 — merge sort, quicksort, heapsort — all meet this bound, and none can beat it. But what if we are willing to go beyond pairwise comparisons? If we know something about the structure of the keys — for instance, that they are integers in a bounded range — we can exploit that structure to sort in linear time. In this chapter we study three such algorithms: counting sort, radix sort, and bucket sort. We also turn to a related problem — selection — and present two algorithms that find the kth smallest element in $O(n)$ time without sorting: randomized quickselect and the deterministic median-of-medians algorithm.

8.1 Breaking the comparison lower bound

The $\Omega(n \log n)$ lower bound from Chapter 4 applies to comparison-based sorting: algorithms that learn about the input only by comparing pairs of elements. The decision-tree argument shows that any comparison-based algorithm must traverse a binary tree of height at least $\log_2(n!) = \Theta(n \log n)$, because there are $n!$ possible permutations and each leaf of the decision tree corresponds to one permutation.

This lower bound does *not* apply if we use operations other than comparisons. If the keys are integers, we can look at individual digits. If the keys are bounded, we can use them as array indices. These non-comparison-based operations give us additional information that comparison-based algorithms cannot access, and this is what allows us to sort faster.

The trade-off is generality: comparison-based sorting works for any totally or-

dered type, while the algorithms in this chapter require specific key structure (integers, bounded range, uniform distribution).

8.2 Counting sort

Counting sort is the simplest linear-time sorting algorithm. It works for non-negative integer keys in a known range $[0, k]$ and sorts by *counting* how many times each value appears.

8.2.1 The algorithm

1. Create an array `counts` of size $k + 1$, initialized to zeros.
2. For each element in the input, increment `counts[element]`.
3. Compute prefix sums: replace each `counts[i]` with the sum of all counts for values $\leq i$. After this step, `counts[i]` tells us the position *after* the last occurrence of value i in the sorted output.
4. Walk the input array *in reverse*, placing each element at position `counts[element] - 1` and decrementing the count. Walking in reverse ensures stability.

8.2.2 Implementation

```
export function countingSort(elements: number[]): number[] {
  if (elements.length <= 1) {
    return elements.slice(0);
  }

  const max = Math.max(...elements);
  const counts = new Array<number>(max + 1).fill(0);

  // Count occurrences
  for (const val of elements) {
    counts[val]!++;
  }

  // Compute prefix sums (cumulative counts)
  for (let i = 1; i <= max; i++) {
    counts[i]! += counts[i - 1]!;
  }

  // Build output array in reverse for stability
```

```
  const output = new Array<number>(elements.length);
  for (let i = elements.length - 1; i >= 0; i--) {
    const val = elements[i]!;
    counts[val]!--;
    output[counts[val]!] = val;
  }

  return output;
}
```

8.2.3 Tracing through an example

Let us sort $A = [4, 2, 2, 8, 3, 3, 1]$.

Step 1–2: Count occurrences. The maximum value is 8, so we create `counts` of size 9:

Index	0	1	2	3	4	5	6	7	8
Count	0	1	2	2	1	0	0	0	1

Step 3: Prefix sums. Each entry becomes the cumulative count:

Index	0	1	2	3	4	5	6	7	8
Prefix sum	0	1	3	5	6	6	6	6	7

The prefix sum tells us: 0 elements are ≤ 0, 1 element is ≤ 1, 3 elements are ≤ 2, and so on.

Step 4: Place elements (reverse scan).

i	$A[i]$	`counts[A[i]]` before	Output position	`counts[A[i]]` after
6	1	1	0	0
5	3	5	4	4
4	3	4	3	3
3	8	7	6	6
2	2	3	2	2
1	2	2	1	1
0	4	6	5	5

Result: $[1, 2, 2, 3, 3, 4, 8]$.

Notice that the two 2s and the two 3s appear in the same relative order as in the input — counting sort is *stable*.

8.2.4 Stability

Counting sort's stability is not an accident; it is a consequence of scanning the input in reverse during the placement step. When we encounter the last occurrence of a value (scanning right to left), we place it at the highest available position for that value. The second-to-last occurrence goes one position earlier, and so on. This preserves the original relative order among elements with equal keys.

Stability matters when sorting records by one key while preserving order on another, and it is essential for counting sort's role as a subroutine in radix sort.

8.2.5 Complexity analysis

Time. The algorithm makes four passes: 1. Finding the maximum: $O(n)$. 2. Counting occurrences: $O(n)$. 3. Computing prefix sums: $O(k)$. 4. Placing elements in the output: $O(n)$.

Total: $O(n + k)$, where k is the maximum value.

Space. The `counts` array uses $O(k)$ space, and the output array uses $O(n)$ space. Total: $O(n + k)$.

When is counting sort practical? When $k = O(n)$, counting sort runs in $O(n)$ time and is excellent. When $k \gg n$ (for example, sorting 10 numbers in the range $[0, 10^9]$), the $O(k)$ space and time become prohibitive, and a comparison-based sort would be faster.

8.2.6 Properties

Property	Counting sort
Time	$O(n + k)$
Space	$O(n + k)$
Stable	Yes
In-place	No
Key type	Non-negative integers in $[0, k]$

8.3 Radix sort

Radix sort extends counting sort to handle integers with many digits. Instead of sorting on the entire key at once (which would require a `counts` array as large as the key range), radix sort processes one digit at a time, from least significant to most significant.

8.3.1 The algorithm

1. Find the maximum element to determine the number of digits d.
2. For each digit position from least significant to most significant:
   - Sort the array by that digit using a stable sort (counting sort restricted to digits 0–9).

The key insight is that we must process digits from *least* significant to *most* significant, and each digit sort must be *stable*. After sorting by the units digit, elements with the same units digit are in a consistent order. When we then sort by the tens digit, stability ensures that elements with the same tens digit remain sorted by their units digit — and so on.

8.3.2 Why least significant digit first?

It may seem counterintuitive to start with the least significant digit. Consider sorting $[329, 457, 657, 839, 436, 720, 355]$. If we sorted by the most significant digit first, we would get groups starting with 3, 4, 6, 7. But then sorting by the next digit within each group is exactly the original problem on smaller arrays — we have made no progress toward a linear-time algorithm.

LSD radix sort avoids this by exploiting stability. After sorting by digit i, the relative order of elements that agree on digit i is determined by the previous passes on digits $0, 1, \ldots, i-1$. When we sort by digit $i+1$, stability preserves this order among elements with the same digit at position $i+1$.

8.3.3 Implementation

The digit-level sorting subroutine is a specialized counting sort that operates on a single digit position:

```
export function countingSortByDigit(
  elements: number[],
  exp: number,
): number[] {
  const n = elements.length;
  if (n <= 1) {
```

```
    return elements.slice(0);
  }

  const output = new Array<number>(n);
  const counts = new Array<number>(10).fill(0);

  // Count occurrences of each digit at position exp
  for (const val of elements) {
    const digit = Math.floor(val / exp) % 10;
    counts[digit]!++;
  }

  // Compute prefix sums
  for (let i = 1; i < 10; i++) {
    counts[i]! += counts[i - 1]!;
  }

  // Build output in reverse for stability
  for (let i = n - 1; i >= 0; i--) {
    const val = elements[i]!;
    const digit = Math.floor(val / exp) % 10;
    counts[digit]!--;
    output[counts[digit]!] = val;
  }

  return output;
}
```

The main radix sort function calls this subroutine for each digit position:

```
export function radixSort(elements: number[]): number[] {
  if (elements.length <= 1) {
    return elements.slice(0);
  }

  const max = Math.max(...elements);

  let result = elements.slice(0);

  // Process each digit position from least significant to most significant
  for (let exp = 1; Math.floor(max / exp) > 0; exp *= 10) {
    result = countingSortByDigit(result, exp);
```

```
  }

  return result;
}
```

8.3.4 Tracing through an example

Sort $A = [170, 45, 75, 90, 802, 24, 2, 66]$.

Pass 1: Sort by units digit (exp = 1):

Element	Units digit
170	0
45	5
75	5
90	0
802	2
24	4
2	2
66	6

After stable sort by units digit: $[170, 90, 802, 2, 24, 45, 75, 66]$.

Pass 2: Sort by tens digit (exp = 10):

Element	Tens digit
170	7
90	9
802	0
2	0
24	2
45	4
75	7
66	6

After stable sort by tens digit: $[802, 2, 24, 45, 66, 170, 75, 90]$.

Notice that 802 and 2 both have tens digit 0, and they remain in the order established by Pass 1 (802 before 2) thanks to stability.

Pass 3: Sort by hundreds digit (exp = 100):

Element	Hundreds digit
802	8
2	0
24	0
45	0
66	0
170	1
75	0
90	0

After stable sort by hundreds digit: $[2, 24, 45, 66, 75, 90, 170, 802]$.

Result: $[2, 24, 45, 66, 75, 90, 170, 802]$. Sorted!

8.3.5 Correctness

Claim. After i passes of LSD radix sort, the array is sorted with respect to the last i digits.

Proof by induction. After the first pass, the array is sorted by the units digit (the counting sort is correct). Assume after i passes the array is sorted by the last i digits. Consider two elements a and b after pass $i+1$:

- If a and b differ in digit $i+1$: the sort on digit $i+1$ places them correctly.
- If a and b have the same digit at position $i+1$: since the sort is *stable*, their relative order is preserved from the previous pass, which (by hypothesis) ordered them correctly by their last i digits.

In both cases, the elements are correctly ordered by their last $i+1$ digits. □

8.3.6 Complexity analysis

Time. Radix sort makes d passes, where d is the number of digits in the maximum element. Each pass is a counting sort with $k = 10$ (the radix), which takes $O(n + 10) = O(n)$ time. Total:

$$T(n) = O(d \cdot n) = O(dn).$$

For $d = O(1)$ (bounded number of digits), this is $O(n)$. More generally, if the values are in the range $[0, n^c - 1]$ for some constant c, then $d = O(c \log_{10} n) = O(\log n)$, and radix sort runs in $O(n \log n)$ — no better than comparison sort. Radix sort achieves true linear time only when d is bounded by a constant independent of n.

Space. Each counting sort pass uses $O(n + 10) = O(n)$ auxiliary space.

8.3.7 Properties

Property	Radix sort
Time	$O(dn)$ where d = number of digits
Space	$O(n)$
Stable	Yes
Key type	Non-negative integers

8.4 Bucket sort

Bucket sort works well when the input is drawn from a *uniform distribution* over a known range. It distributes elements into equal-width buckets, sorts each bucket individually (typically with insertion sort), and concatenates the sorted buckets.

8.4.1 The algorithm

1. Determine the range $[\min, \max]$ of the input.
2. Create n empty buckets spanning the range.
3. Place each element in its bucket: element x goes to bucket $\lfloor (x - \min)/(\max - \min) \cdot (n-1) \rfloor$.
4. Sort each bucket using insertion sort.
5. Concatenate all buckets.

8.4.2 Implementation

```ts
export function bucketSort(
  elements: number[],
  bucketCount?: number,
): number[] {
  const n = elements.length;
  if (n <= 1) {
    return elements.slice(0);
  }

  const max = Math.max(...elements);
  const min = Math.min(...elements);

  // If all elements are the same, return a copy
```

```
  if (max === min) {
    return elements.slice(0);
  }

  const numBuckets = bucketCount ?? n;
  const range = max - min;

  // Create empty buckets
  const buckets: number[][] = [];
  for (let i = 0; i < numBuckets; i++) {
    buckets.push([]);
  }

  // Distribute elements into buckets
  for (const val of elements) {
    let index = Math.floor(
      ((val - min) / range) * (numBuckets - 1)
    );
    if (index >= numBuckets) {
      index = numBuckets - 1;
    }
    buckets[index]!.push(val);
  }

  // Sort each bucket using insertion sort and concatenate
  const result: number[] = [];
  for (const bucket of buckets) {
    insertionSortInPlace(bucket);
    for (const val of bucket) {
      result.push(val);
    }
  }

  return result;
}
```

The subroutine `insertionSortInPlace` is efficient for the small bucket sizes expected under uniform distribution:

```
function insertionSortInPlace(arr: number[]): void {
  for (let i = 1; i < arr.length; i++) {
    const key = arr[i]!;
```

```
    let j = i - 1;
    while (j >= 0 && arr[j]! > key) {
      arr[j + 1] = arr[j]!;
      j--;
    }
    arr[j + 1] = key;
  }
}
```

8.4.3 Tracing through an example

Sort $A = [0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68]$ using 10 buckets.

Here the range is $[0.12, 0.94]$, so each bucket covers an interval of width roughly 0.082.

Bucket	Elements
0	[0.17, 0.12]
1	[0.26, 0.21, 0.23]
3	[0.39]
6	[0.72, 0.68]
8	[0.78]
9	[0.94]

After sorting each bucket and concatenating: $[0.12, 0.17, 0.21, 0.23, 0.26, 0.39, 0.68, 0.72, 0.78, 0$

8.4.4 Complexity analysis

Expected time under uniform distribution. If n elements are drawn independently and uniformly from $[0, 1)$, then with n buckets each element lands in a random bucket. The expected number of elements per bucket is 1. By a balls-into-bins argument, the expected total cost of sorting all buckets is:

$$\sum_{i=0}^{n-1} O(\mathbb{E}[n_i^2])$$

where n_i is the number of elements in bucket i. Since each element independently falls into bucket i with probability $1/n$, we have $\mathbb{E}[n_i] = 1$ and $\mathbb{E}[n_i^2] = 2 - 1/n$. Summing over n buckets:

$$\sum_{i=0}^{n-1} O(2 - 1/n) = O(n).$$

Including the $O(n)$ distribution and concatenation steps, the total expected time is $O(n)$.

Worst case. If all elements fall into one bucket, we pay $O(n^2)$ for insertion sort on that bucket. This happens when the distribution is far from uniform.

Space. The buckets collectively hold n elements, plus $O(n)$ for the bucket array structure. Total: $O(n)$.

8.4.5 Properties

Property	Bucket sort
Expected time	$O(n)$ (uniform distribution)
Worst-case time	$O(n^2)$
Space	$O(n)$
Stable	Yes (with stable per-bucket sort)
Key type	Numeric keys in a known range

8.5 Comparison of linear-time sorts

Algorithm	Time	Space	Stable	Assumptions
Counting sort	$O(n + k)$	$O(n + k)$	Yes	Integer keys in $[0, k]$
Radix sort	$O(dn)$	$O(n)$	Yes	Integer keys with d digits
Bucket sort	$O(n)$ expected	$O(n)$	Yes	Uniformly distributed keys

All three algorithms achieve linear time under specific conditions. Counting sort is simplest and best when the key range k is not much larger than n. Radix sort extends counting sort to larger ranges by processing one digit at a time. Bucket sort is ideal for floating-point data with a known, roughly uniform distribution.

None of these algorithms contradicts the $\Omega(n \log n)$ comparison lower bound — they bypass it by using non-comparison operations (indexing into an array by key value, extracting digits).

8.6 The selection problem

We now turn to a different problem. Given an unsorted array of n elements and an integer k (with $0 \leq k < n$), find the kth smallest element — the element that would be at index k if the array were sorted.

Special cases include: - $k = 0$: the minimum (trivially solvable in $O(n)$). - $k = n-1$: the maximum. - $k = \lfloor n/2 \rfloor$: the median.

The naive approach is to sort the array ($O(n \log n)$) and return the element at index k. Can we do better? Yes — we can solve the selection problem in $O(n)$ time.

8.7 Quickselect

Quickselect (also known as Hoare's selection algorithm) is the selection analogue of quicksort. Like quicksort, it partitions the array around a pivot. But unlike quicksort, it only recurses into *one* side — the side that contains the desired element.

8.7.1 The algorithm

1. Choose a random pivot and partition the array.
2. If the pivot lands at position k, we are done.
3. If $k < $ pivot's position, recurse on the left partition.
4. If $k > $ pivot's position, recurse on the right partition.

8.7.2 Implementation

```
export function quickselect(
  elements: number[],
  k: number,
): number {
  if (elements.length === 0) {
    throw new RangeError('Cannot select from an empty array');
  }
  if (k < 0 || k >= elements.length) {
    throw new RangeError(
      `k=${k} is out of bounds for array of length ${elements.length}`,
    );
  }
```

```
  const copy = elements.slice(0);
  return select(copy, 0, copy.length - 1, k);
}

function select(
  arr: number[],
  left: number,
  right: number,
  k: number,
): number {
  if (left === right) {
    return arr[left]!;
  }

  const pivotIndex = randomizedPartition(arr, left, right);

  if (k === pivotIndex) {
    return arr[pivotIndex]!;
  } else if (k < pivotIndex) {
    return select(arr, left, pivotIndex - 1, k);
  } else {
    return select(arr, pivotIndex + 1, right, k);
  }
}
```

The `randomizedPartition` function is identical to the one used in randomized quicksort: choose a random element, swap it to the end, partition using the Lomuto scheme.

8.7.3 Tracing through an example

Find the 3rd smallest element ($k = 2$, zero-indexed) in $A = [7, 3, 9, 1, 5]$.

The sorted array would be $[1, 3, 5, 7, 9]$, so the answer is 5.

Iteration 1: Suppose the random pivot is 7 (index 0). After partitioning: $[3, 1, 5, 7, 9]$, pivot at index 3.

We want $k = 2 < 3$, so recurse on the left partition $[3, 1, 5]$ (indices 0–2).

Iteration 2: Suppose the random pivot is 1 (index 1 of the subarray). After partitioning: $[1, 3, 5, 7, 9]$, pivot at index 0.

We want $k = 2 > 0$, so recurse on the right partition $[3, 5]$ (indices 1–2).

Iteration 3: Suppose the random pivot is 5 (index 2). After partitioning: $[1, 3, 5, 7, 9]$, pivot at index 2.

We want $k = 2 = 2$. Done! Return $A[2] = 5$.

8.7.4 Complexity analysis

Expected time. The analysis is similar to randomized quicksort. With a random pivot, the expected partition splits the array roughly in half. But unlike quicksort, we recurse into only one partition, so the expected work at each level halves:

$$\mathbb{E}[T(n)] = n + \mathbb{E}[T(n/2)] \approx n + n/2 + n/4 + \cdots = 2n = O(n).$$

More precisely, the expected number of comparisons is at most $4n$ (by an analysis similar to the randomized quicksort proof, summing indicator random variables over pairs).

Worst case. If the pivot always lands at one extreme, we have:

$$T(n) = n + T(n-1) = O(n^2).$$

This is the same worst case as quicksort, but it is extremely unlikely with random pivots.

8.7.5 Properties

Property	Quickselect
Expected time	$O(n)$
Worst-case time	$O(n^2)$
Space	$O(n)$ for copy + $O(\log n)$ expected stack
Deterministic	No (randomized)

8.8 Median of medians

Can we achieve $O(n)$ *worst-case* selection? The answer is yes, using a clever pivot-selection strategy called *median of medians* (also known as BFPRT, after its five inventors: Blum, Floyd, Pratt, Rivest, and Tarjan, 1973).

The idea: instead of choosing a random pivot, choose a pivot that is *guaranteed* to be near the median, ensuring that each partition eliminates a constant fraction of the elements.

8.8.1 The algorithm

1. Divide the n elements into groups of 5.
2. Find the median of each group by sorting (5 elements can be sorted in constant time).
3. Recursively compute the median of these $\lceil n/5 \rceil$ medians.
4. Use this "median of medians" as the pivot for partitioning.
5. Recurse into the appropriate partition (just like quickselect).

8.8.2 Why groups of 5?

The choice of 5 is not arbitrary. It is the smallest odd group size that makes the recurrence work out to $O(n)$. The median of medians is guaranteed to be larger than at least $3 \cdot \lceil n/10 \rceil - 2$ elements and smaller than at least $3 \cdot \lceil n/10 \rceil - 2$ elements. This means each recursive call operates on at most roughly $7n/10$ elements.

Here is why: the median of medians is larger than the medians of half the groups (roughly $n/10$ groups), and each of those medians is larger than 2 elements in its group. Therefore, the pivot is larger than at least $3n/10$ elements. By symmetry, it is also smaller than at least $3n/10$ elements. The worst-case partition is therefore at most $7n/10$.

8.8.3 Implementation

```
export function medianOfMedians(
  elements: number[],
  k: number,
): number {
  if (elements.length === 0) {
    throw new RangeError('Cannot select from an empty array');
  }
  if (k < 0 || k >= elements.length) {
    throw new RangeError(
      `k=${k} is out of bounds for array of length ${elements.length}`,
    );
  }

  const copy = elements.slice(0);
  return selectMoM(copy, 0, copy.length - 1, k);
}
```

The core recursive function:

```
function selectMoM(
  arr: number[],
  left: number,
  right: number,
  k: number,
): number {
  // Base case: small enough to sort directly
  if (right - left < 5) {
    insertionSortRange(arr, left, right);
    return arr[k]!;
  }

  // Step 1: Divide into groups of 5, find median of each
  const numGroups = Math.ceil((right - left + 1) / 5);
  for (let i = 0; i < numGroups; i++) {
    const groupLeft = left + i * 5;
    const groupRight = Math.min(groupLeft + 4, right);

    insertionSortRange(arr, groupLeft, groupRight);

    const medianIndex =
      groupLeft + Math.floor((groupRight - groupLeft) / 2);
    swap(arr, medianIndex, left + i);
  }

  // Step 2: Recursively find the median of the medians
  const medianOfMediansIndex =
    left + Math.floor((numGroups - 1) / 2);
  selectMoM(arr, left, left + numGroups - 1, medianOfMediansIndex);

  // Step 3: Partition around the median of medians
  const pivotIndex = partitionAroundPivot(
    arr, left, right, medianOfMediansIndex
  );

  if (k === pivotIndex) {
    return arr[pivotIndex]!;
  } else if (k < pivotIndex) {
    return selectMoM(arr, left, pivotIndex - 1, k);
  } else {
    return selectMoM(arr, pivotIndex + 1, right, k);
```

```
  }
}
```

8.8.4 Tracing through an example

Find the median ($k = 7$, zero-indexed) in:

$A = [12, 3, 5, 7, 19, 26, 4, 1, 8, 15, 20, 11, 9, 2, 6]$.

The sorted array is $[1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 15, 19, 20, 26]$, so the answer at $k = 7$ is 8.

Step 1: Divide into groups of 5 and find medians.

Group	Elements	Sorted	Median
1	[12, 3, 5, 7, 19]	[3, 5, 7, 12, 19]	7
2	[26, 4, 1, 8, 15]	[1, 4, 8, 15, 26]	8
3	[20, 11, 9, 2, 6]	[2, 6, 9, 11, 20]	9

Step 2: Median of medians. The medians are $[7, 8, 9]$. The median of this group is 8.

Step 3: Partition around 8. Using 8 as the pivot, elements < 8 go left, elements > 8 go right:

$$[\underbrace{3, 5, 7, 4, 1, 2, 6}_{<8},\ \underline{8},\ \underbrace{12, 19, 26, 15, 20, 11, 9}_{>8}]$$

The pivot lands at index 7. We want $k = 7$, and the pivot is at index 7. Done! Return 8.

8.8.5 Complexity analysis

Time. Let $T(n)$ be the worst-case time for selecting from n elements.

The algorithm does the following work: - Sorting $\lceil n/5 \rceil$ groups of 5: $O(n)$ total. - Finding the median of medians: $T(\lceil n/5 \rceil)$. - Partitioning: $O(n)$. - Recursing into the larger partition: at most $T(7n/10)$.

This gives the recurrence:

$$T(n) \leq T(n/5) + T(7n/10) + O(n).$$

We claim $T(n) = O(n)$. To verify, assume $T(n) \leq cn$ for some constant c. Then:

$$T(n) \le cn/5 + 7cn/10 + an = cn(1/5 + 7/10) + an = 9cn/10 + an = cn$$

provided $c \ge 10a$. Since $1/5 + 7/10 = 9/10 < 1$, the two recursive calls together operate on a shrinking fraction of the input, and the algorithm runs in $O(n)$ time.

Space. The recursion has depth $O(\log n)$ (each level reduces the problem by a constant factor), so the stack space is $O(\log n)$. Combined with the $O(n)$ copy, total space is $O(n)$.

8.8.6 Practical considerations

While the median-of-medians algorithm is a beautiful theoretical result — it proved that deterministic linear-time selection is possible — it is rarely used in practice. The constant factor hidden in the $O(n)$ is large (roughly 5–10× slower than randomized quickselect for typical inputs). Randomized quickselect is almost always faster in practice because:

1. It avoids the overhead of computing medians of groups.
2. Random pivots are usually good enough.
3. The probability of quadratic behavior is astronomically small.

The practical value of median of medians is primarily as a *fallback*: some implementations (e.g., the `introselect` algorithm in C++ STL) start with quickselect and switch to median of medians if the recursion depth grows too large, guaranteeing worst-case $O(n)$ while maintaining fast average-case performance.

8.8.7 Properties

Property	Median of medians
Worst-case time	$O(n)$
Space	$O(n)$
Deterministic	Yes
Practical	Slower than quickselect due to large constants

8.9 Chapter summary

In this chapter we studied algorithms that break the $\Omega(n \log n)$ comparison-based sorting barrier and solve the selection problem in linear time:

- **Counting sort** sorts non-negative integers in $O(n + k)$ time by counting occurrences and computing prefix sums. It is stable and serves as a building block for radix sort.

- **Radix sort** extends counting sort to handle integers with multiple digits, sorting digit by digit from least significant to most significant. It runs in $O(dn)$ time where d is the number of digits. The key requirement is a stable subroutine sort.
- **Bucket sort** distributes elements into buckets, sorts each bucket, and concatenates. Under a uniform distribution, the expected time is $O(n)$. Its worst case is $O(n^2)$ when all elements land in one bucket.
- **Quickselect** finds the kth smallest element in expected $O(n)$ time by partitioning around a random pivot and recursing into one side. It is the practical algorithm of choice for selection.
- **Median of medians** achieves worst-case $O(n)$ selection through a carefully chosen pivot: the median of group medians. While theoretically optimal, its large constant factor makes it slower than randomized quickselect in practice.

The linear-time sorting algorithms teach an important lesson: algorithmic lower bounds depend on the model of computation. The $\Omega(n \log n)$ bound is real for comparison-based sorting, but by stepping outside the comparison model — using integers as array indices, extracting digits — we can do better. The selection algorithms show that finding a single order statistic is fundamentally easier than fully sorting, requiring only $O(n)$ time regardless of the method.

8.10 Exercises

Exercise 6.1. Trace through counting sort on the input $[3, 0, 1, 3, 1, 0, 2, 3]$. Show the `counts` array after each step (counting, prefix sums, placement). Verify that the sort is stable by tracking the original indices of elements with value 3.

Exercise 6.2. Radix sort processes digits from least significant to most significant, using a *stable* sort at each step. What goes wrong if we process digits from most significant to least significant? Give a concrete example where MSD radix sort (without special handling) produces incorrect output.

Exercise 6.3. Counting sort uses $O(k)$ space for the counts array, where k is the maximum value. If we need to sort n integers in the range $[0, n^2)$, we could use counting sort directly with $k = n^2$, or we could use radix sort with a base-n representation (2 digits). Compare the time and space complexity of both approaches.

Exercise 6.4. Consider a modification of quickselect where, instead of choosing a random pivot, we always choose the first element as the pivot. Describe an input of size n for which this modified quickselect takes $\Theta(n^2)$ time to find the

median. Then describe an input for which it takes $\Theta(n)$ time.

Exercise 6.5. The median-of-medians algorithm divides elements into groups of 5. What happens if we use groups of 3 instead? Set up the recurrence and show that it does *not* solve to $O(n)$. What about groups of 7? (Hint: compute the fraction of elements guaranteed to be eliminated at each step for each group size.)

Chapter 9

Arrays, Linked Lists, Stacks, and Queues

The algorithms of the preceding chapters operate on arrays — contiguous blocks of memory indexed by integers. Arrays are powerful but they are only one of many ways to organize data. In this chapter we study the fundamental data structures that underpin nearly all of computer science: dynamic arrays, linked lists, stacks, queues, and deques. Each offers a different set of trade-offs between time complexity, memory usage, and flexibility. Understanding these structures deeply is essential, because every higher-level data structure — from hash tables to balanced trees to graphs — is built on top of them.

9.1 Arrays

An **array** is the simplest data structure: a contiguous block of memory divided into equal-sized slots, each identified by an integer index. Accessing any element by its index takes $O(1)$ time, because the memory address can be computed directly: if the array starts at address b and each element occupies s bytes, then element i lives at address $b + i \cdot s$.

This direct addressing makes arrays extremely efficient for random access. However, arrays have a fundamental limitation: their size is fixed at creation time. If we need to store more elements than the array can hold, we must allocate a new, larger array and copy all existing elements — an $O(n)$ operation.

9.1.1 Static arrays in TypeScript

TypeScript (and JavaScript) arrays are actually dynamic — they resize automatically behind the scenes. But to understand the foundations, imagine a fixed-size

array:

```
const fixed = new Array<number>(10); // 10 slots, all undefined
fixed[0] = 42;
fixed[9] = 99;
// fixed[10] would be out of bounds in a true static array
```

In languages like C or Java, going beyond the allocated size is either a compile-time error or a runtime crash. JavaScript's built-in arrays hide this complexity, but the cost of resizing is still there — it is just managed for us. Let us see how.

9.2 Dynamic arrays

A **dynamic array** maintains an internal buffer that is larger than the number of elements currently stored. When the buffer fills up, the array allocates a new buffer of double the size and copies all elements over. This doubling strategy gives us amortized $O(1)$ appends while keeping worst-case access at $O(1)$.

9.2.1 The doubling strategy

Suppose our dynamic array has capacity c and currently holds n elements. When we append element $n + 1$:

- If $n < c$: store the element in slot n. Cost: $O(1)$.
- If $n = c$: allocate a new buffer of size $2c$, copy all n elements, then store the new element. Cost: $O(n)$.

The key insight is that expensive copies happen rarely. After a copy doubles the capacity to $2c$, we can perform another c cheap appends before the next copy. This is the essence of **amortized analysis**.

9.2.2 Amortized analysis of append

We use the **aggregate method**. Starting from an empty array with initial capacity 1, suppose we perform n appends. Copies happen when the size reaches 1, 2, 4, 8, ..., up to some power of 2. The total number of element copies across all resizes is:

$$1 + 2 + 4 + 8 + \cdots + 2^{\lfloor \log_2 n \rfloor} \leq 2n$$

So the total cost of n appends is at most n (for the stores) plus $2n$ (for all the copies), giving $3n$ total. The amortized cost per append is therefore $3n/n = O(1)$.

9.2.3 Implementation

Our `DynamicArray<T>` uses a plain JavaScript array as the backing buffer, with explicit capacity management. The initial capacity defaults to 4.

```
export class DynamicArray<T> implements Iterable<T> {
  private data: (T | undefined)[];
  private length: number;

  constructor(initialCapacity = 4) {
    this.data = new Array<T | undefined>(initialCapacity);
    this.length = 0;
  }

  get size(): number {
    return this.length;
  }

  get capacity(): number {
    return this.data.length;
  }

  get(index: number): T {
    this.checkBounds(index);
    return this.data[index] as T;
  }

  set(index: number, value: T): void {
    this.checkBounds(index);
    this.data[index] = value;
  }

  append(value: T): void {
    if (this.length === this.data.length) {
      this.resize(this.data.length * 2);
    }
    this.data[this.length] = value;
    this.length++;
  }

  insert(index: number, value: T): void {
    if (index < 0 || index > this.length) {
```

```
      throw new RangeError(
        `Index ${index} out of bounds for size ${this.length}`
      );
    }
    if (this.length === this.data.length) {
      this.resize(this.data.length * 2);
    }
    for (let i = this.length; i > index; i--) {
      this.data[i] = this.data[i - 1];
    }
    this.data[index] = value;
    this.length++;
  }

  remove(index: number): T {
    this.checkBounds(index);
    const value = this.data[index] as T;
    for (let i = index; i < this.length - 1; i++) {
      this.data[i] = this.data[i + 1];
    }
    this.data[this.length - 1] = undefined;
    this.length--;
    if (
      this.length > 0 &&
      this.length <= this.data.length / 4 &&
      this.data.length > 4
    ) {
      this.resize(Math.max(4, Math.floor(this.data.length / 2)));
    }
    return value;
  }

  private resize(newCapacity: number): void {
    const newData = new Array<T | undefined>(newCapacity);
    for (let i = 0; i < this.length; i++) {
      newData[i] = this.data[i];
    }
    this.data = newData;
  }

  private checkBounds(index: number): void {
```

```
    if (index < 0 || index >= this.length) {
      throw new RangeError(
        `Index ${index} out of bounds for size ${this.length}`
      );
    }
  }
  // ... iterator, toArray, etc.
}
```

Notice that `remove` also implements **shrinking**: when occupancy falls below 25%, the buffer is halved (but never below 4). This prevents a long sequence of removals from wasting memory, and the halving threshold (1/4 rather than 1/2) avoids **thrashing** — a pathological pattern where alternating appends and removes near the boundary trigger repeated resizes.

9.2.4 Complexity summary

Operation	Time	Notes
`get(i)` / `set(i, v)`	$O(1)$	Direct index access
`append(v)`	$O(1)$ amortized	$O(n)$ worst case during resize
`insert(i, v)`	$O(n)$	Must shift elements right
`remove(i)`	$O(n)$	Must shift elements left
`indexOf(v)`	$O(n)$	Linear scan

9.3 Linked lists

A **linked list** stores elements in nodes that are scattered throughout memory, with each node containing a value and a pointer (reference) to the next node. Unlike arrays, linked lists do not require contiguous memory, and inserting or removing an element at a known position takes $O(1)$ time — no shifting required.

The trade-off is that random access is lost: to reach the ith element, we must follow i pointers from the head, taking $O(i)$ time.

9.3.1 Singly linked lists

In a **singly linked list**, each node points to the next node. The list maintains a pointer to the **head** (first node) and, for efficiency, a pointer to the **tail** (last node).

```
head → [10 | •] → [20 | •] → [30 | null]
```

9.3.1.1 Implementation

```
class SinglyNode<T> {
  constructor(
    public value: T,
    public next: SinglyNode<T> | null = null,
  ) {}
}

export class SinglyLinkedList<T> implements Iterable<T> {
  private head: SinglyNode<T> | null = null;
  private tail: SinglyNode<T> | null = null;
  private length: number = 0;

  get size(): number {
    return this.length;
  }

  prepend(value: T): void {
    const node = new SinglyNode(value, this.head);
    this.head = node;
    if (this.tail === null) {
      this.tail = node;
    }
    this.length++;
  }

  append(value: T): void {
    const node = new SinglyNode(value);
    if (this.tail !== null) {
      this.tail.next = node;
    } else {
      this.head = node;
    }
    this.tail = node;
    this.length++;
  }
```

```
  removeFirst(): T | undefined {
    if (this.head === null) return undefined;
    const value = this.head.value;
    this.head = this.head.next;
    if (this.head === null) {
      this.tail = null;
    }
    this.length--;
    return value;
  }

  delete(value: T): boolean {
    if (this.head === null) return false;
    if (this.head.value === value) {
      this.head = this.head.next;
      if (this.head === null) this.tail = null;
      this.length--;
      return true;
    }
    let current = this.head;
    while (current.next !== null) {
      if (current.next.value === value) {
        if (current.next === this.tail) this.tail = current;
        current.next = current.next.next;
        this.length--;
        return true;
      }
      current = current.next;
    }
    return false;
  }

  find(value: T): boolean {
    let current = this.head;
    while (current !== null) {
      if (current.value === value) return true;
      current = current.next;
    }
    return false;
  }
  // ... iterator, toArray, etc.
```

```
}
```

9.3.1.2 Tracing through an example

Starting with an empty singly linked list, let us perform a sequence of operations:

Operation	List state	`size`
`append(10)`	`[10]`	1
`append(20)`	`[10] → [20]`	2
`prepend(5)`	`[5] → [10] → [20]`	3
`removeFirst()` → 5	`[10] → [20]`	2
`delete(20)` → true	`[10]`	1
`append(30)`	`[10] → [30]`	2

Notice that `prepend` and `removeFirst` are both $O(1)$ because they only touch the head pointer. Appending is $O(1)$ because we maintain a tail pointer. However, `delete(value)` requires a linear scan.

9.3.1.3 A limitation of singly linked lists

Removing the *last* element is $O(n)$ in a singly linked list, because we must traverse the entire list to find the node that precedes the tail. The doubly linked list solves this problem.

9.3.2 Doubly linked lists

In a **doubly linked list**, each node has pointers to both the next *and* previous nodes. This enables $O(1)$ removal from both ends.

```
null ← [10 | •] ⇄ [20 | •] ⇄ [30 | •] → null
        ↑                     ↑
       head                  tail
```

9.3.2.1 Implementation

```
class DoublyNode<T> {
  constructor(
    public value: T,
    public prev: DoublyNode<T> | null = null,
    public next: DoublyNode<T> | null = null,
  ) {}
```

```
}

export class DoublyLinkedList<T> implements Iterable<T> {
  private head: DoublyNode<T> | null = null;
  private tail: DoublyNode<T> | null = null;
  private length: number = 0;

  get size(): number {
    return this.length;
  }

  prepend(value: T): void {
    const node = new DoublyNode(value, null, this.head);
    if (this.head !== null) {
      this.head.prev = node;
    } else {
      this.tail = node;
    }
    this.head = node;
    this.length++;
  }

  append(value: T): void {
    const node = new DoublyNode(value, this.tail, null);
    if (this.tail !== null) {
      this.tail.next = node;
    } else {
      this.head = node;
    }
    this.tail = node;
    this.length++;
  }

  removeFirst(): T | undefined {
    if (this.head === null) return undefined;
    const value = this.head.value;
    this.head = this.head.next;
    if (this.head !== null) {
      this.head.prev = null;
    } else {
      this.tail = null;
```

```
    }
    this.length--;
    return value;
  }

  removeLast(): T | undefined {
    if (this.tail === null) return undefined;
    const value = this.tail.value;
    this.tail = this.tail.prev;
    if (this.tail !== null) {
      this.tail.next = null;
    } else {
      this.head = null;
    }
    this.length--;
    return value;
  }

  private removeNode(node: DoublyNode<T>): void {
    if (node.prev !== null) {
      node.prev.next = node.next;
    } else {
      this.head = node.next;
    }
    if (node.next !== null) {
      node.next.prev = node.prev;
    } else {
      this.tail = node.prev;
    }
    this.length--;
  }
  // ... delete, find, iterators, etc.
}
```

The critical advantage is `removeLast`: by following the tail's `prev` pointer, we can unlink the last node in $O(1)$ time without traversing the list. The `removeNode` helper detaches any node from the list in $O(1)$ once we have a reference to it.

The cost of this flexibility is extra memory: each node stores two pointers instead of one. For large collections of small values, this overhead can be significant.

9.3.3 Comparing arrays and linked lists

Operation	Dynamic array	Singly linked list	Doubly linked list
Access by index	$O(1)$	$O(n)$	$O(n)$
Prepend	$O(n)$	$O(1)$	$O(1)$
Append	$O(1)$*	$O(1)$	$O(1)$
Remove first	$O(n)$	$O(1)$	$O(1)$
Remove last	$O(1)$*	$O(n)$	$O(1)$
Insert at known position	$O(n)$	$O(1)$	$O(1)$
Search	$O(n)$	$O(n)$	$O(n)$
Memory per element	Low (contiguous)	+1 pointer	+2 pointers
Cache performance	Excellent	Poor	Poor

* Amortized

When to use which:

- **Dynamic array** when you need fast random access or are iterating sequentially (cache-friendly).
- **Singly linked list** when insertions and deletions at the front dominate.
- **Doubly linked list** when you need efficient removal from both ends or deletion of arbitrary nodes (given a reference).

In practice, arrays and dynamic arrays dominate due to cache locality — modern CPUs are optimized for accessing contiguous memory. Linked lists shine in scenarios where elements are frequently inserted or removed at the endpoints, or when the data is too large to copy during a resize.

9.4 Abstract data types: stacks, queues, and deques

The data structures above — arrays and linked lists — are concrete implementations. Now we turn to **abstract data types** (ADTs): specifications of behavior that can be implemented in multiple ways. A stack, for instance, defines *what* operations are available (push, pop, peek) and *what* they do, without prescribing *how* to store the elements.

9.4.1 Stacks

A **stack** is a Last-In, First-Out (LIFO) collection. The most recently added element is the first one to be removed, like a stack of plates.

9.4.1.1 Interface

```
interface IStack<T> {
  push(value: T): void;     // Add to top
  pop(): T | undefined;      // Remove and return top
  peek(): T | undefined;     // Return top without removing
  readonly size: number;
  readonly isEmpty: boolean;
}
```

9.4.1.2 Implementation

A stack is naturally implemented as a linked list where both `push` and `pop` operate on the head:

```
export class Stack<T> implements IStack<T>, Iterable<T> {
  private head: { value: T; next: unknown } | null = null;
  private length: number = 0;

  get size(): number { return this.length; }
  get isEmpty(): boolean { return this.length === 0; }

  push(value: T): void {
    this.head = { value, next: this.head };
    this.length++;
  }

  pop(): T | undefined {
    if (this.head === null) return undefined;
    const value = this.head.value;
    this.head = this.head.next as typeof this.head;
    this.length--;
    return value;
  }

  peek(): T | undefined {
    return this.head?.value;
  }
}
```

All three operations — `push`, `pop`, `peek` — are $O(1)$.

We could equally implement a stack with a dynamic array (push = append, pop =

remove last). The array-based version has better cache locality, while the linked-list version avoids occasional resize costs. For most purposes in TypeScript, the built-in array with `push`/`pop` is the pragmatic choice; our implementation here serves pedagogical purposes.

9.4.1.3 Applications

Stacks appear throughout computer science:

- **Function call stack.** When a function is called, its local variables and return address are pushed onto the call stack. When it returns, they are popped. This is why recursive algorithms can overflow the stack with too many nested calls.
- **Parenthesis matching.** To check whether brackets are balanced in an expression like `((a + b) * c)`, push each opening bracket and pop when a matching closing bracket is found.
- **Undo/redo.** Text editors push each action onto an undo stack. Undoing pops the most recent action.
- **Depth-first search.** DFS uses a stack (often the call stack via recursion) to track which vertices to visit next.

9.4.1.4 Tracing through an example

Operation	Stack (top → bottom)	Returned
`push(10)`	`10`	—
`push(20)`	`20, 10`	—
`push(30)`	`30, 20, 10`	—
`peek()`	`30, 20, 10`	30
`pop()`	`20, 10`	30
`pop()`	`10`	20
`push(40)`	`40, 10`	—
`pop()`	`10`	40
`pop()`	*(empty)*	10

9.4.2 Queues

A **queue** is a First-In, First-Out (FIFO) collection. Elements are added at the back and removed from the front, like a line of people waiting.

9.4.2.1 Interface

```
interface IQueue<T> {
  enqueue(value: T): void;    // Add to back
  dequeue(): T | undefined;   // Remove and return front
  peek(): T | undefined;      // Return front without removing
  readonly size: number;
  readonly isEmpty: boolean;
}
```

9.4.2.2 Implementation

A queue maps naturally onto a singly linked list with head and tail pointers: enqueue appends at the tail, dequeue removes from the head.

```
interface QueueNode<T> {
  value: T;
  next: QueueNode<T> | null;
}

export class Queue<T> implements IQueue<T>, Iterable<T> {
  private head: QueueNode<T> | null = null;
  private tail: QueueNode<T> | null = null;
  private length: number = 0;

  get size(): number { return this.length; }
  get isEmpty(): boolean { return this.length === 0; }

  enqueue(value: T): void {
    const node: QueueNode<T> = { value, next: null };
    if (this.tail !== null) {
      this.tail.next = node;
    } else {
      this.head = node;
    }
    this.tail = node;
    this.length++;
  }

  dequeue(): T | undefined {
    if (this.head === null) return undefined;
    const value = this.head.value;
    this.head = this.head.next;
```

```
    if (this.head === null) this.tail = null;
    this.length--;
    return value;
  }

  peek(): T | undefined {
    return this.head?.value;
  }
}
```

All operations are $O(1)$.

An array-based queue is trickier: naively dequeuing from the front of an array is $O(n)$ because every element must shift. A **circular buffer** solves this by wrapping indices around modulo the capacity, giving $O(1)$ amortized enqueue and dequeue. Our linked-list implementation avoids this complexity altogether.

9.4.2.3 Applications

- **Breadth-first search.** BFS uses a queue to explore vertices level by level.
- **Task scheduling.** Operating systems use queues to schedule processes for CPU time.
- **Buffering.** Data streams (network packets, keyboard input) are buffered in queues.
- **Level-order tree traversal.** Visiting tree nodes level by level requires a queue.

9.4.2.4 Tracing through an example

Operation	Queue (front → back)	Returned
`enqueue(10)`	`10`	—
`enqueue(20)`	`10, 20`	—
`enqueue(30)`	`10, 20, 30`	—
`peek()`	`10, 20, 30`	10
`dequeue()`	`20, 30`	10
`dequeue()`	`30`	20
`enqueue(40)`	`30, 40`	—
`dequeue()`	`40`	30

9.4.3 Deques

A **deque** (double-ended queue, pronounced "deck") supports insertion and removal at *both* ends in $O(1)$ time. It generalizes both stacks and queues.

9.4.3.1 Implementation

A deque maps directly onto a doubly linked list:

```
interface DequeNode<T> {
  value: T;
  prev: DequeNode<T> | null;
  next: DequeNode<T> | null;
}

export class Deque<T> implements Iterable<T> {
  private head: DequeNode<T> | null = null;
  private tail: DequeNode<T> | null = null;
  private length: number = 0;

  get size(): number { return this.length; }
  get isEmpty(): boolean { return this.length === 0; }

  pushFront(value: T): void {
    const node: DequeNode<T> = { value, prev: null, next: this.head };
    if (this.head !== null) {
      this.head.prev = node;
    } else {
      this.tail = node;
    }
    this.head = node;
    this.length++;
  }

  pushBack(value: T): void {
    const node: DequeNode<T> = { value, prev: this.tail, next: null };
    if (this.tail !== null) {
      this.tail.next = node;
    } else {
      this.head = node;
    }
    this.tail = node;
    this.length++;
```

```
  }

  popFront(): T | undefined {
    if (this.head === null) return undefined;
    const value = this.head.value;
    this.head = this.head.next;
    if (this.head !== null) this.head.prev = null;
    else this.tail = null;
    this.length--;
    return value;
  }

  popBack(): T | undefined {
    if (this.tail === null) return undefined;
    const value = this.tail.value;
    this.tail = this.tail.prev;
    if (this.tail !== null) this.tail.next = null;
    else this.head = null;
    this.length--;
    return value;
  }

  peekFront(): T | undefined { return this.head?.value; }
  peekBack(): T | undefined { return this.tail?.value; }
}
```

All six operations — `pushFront`, `pushBack`, `popFront`, `popBack`, `peekFront`, `peekBack` — are $O(1)$.

9.4.3.2 Using a deque as a stack or queue

A deque subsumes both stacks and queues:

- **As a stack:** use `pushFront` / `popFront` (or `pushBack` / `popBack`).
- **As a queue:** use `pushBack` / `popFront`.

This flexibility makes the deque a useful building block when the access pattern is uncertain, or when both ends are needed.

9.4.3.3 Applications

- **Sliding window maximum.** In the classic interview problem "maximum in every window of size k," a deque holds indices of potential maximums.

Elements are added at the back and removed from the front (when they fall out of the window) or from the back (when a larger element supersedes them).

- **Work-stealing schedulers.** Each thread has a deque of tasks. It pops from its own front, while idle threads steal from other deques' backs.
- **Palindrome checking.** Push characters from both ends; pop from both ends and compare.

9.5 Complexity comparison

	`DynamicArray`	`SinglyLinkedList`	`DoublyLinkedList`	`Stack`	`Queue`	`Deque`
Add front	$O(n)$	$O(1)$	$O(1)$	$O(1)$	—	$O(1)$
Add back	$O(1)$*	$O(1)$	$O(1)$	—	$O(1)$	$O(1)$
Remove front	$O(n)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Remove back	$O(1)$*	$O(n)$	$O(1)$	—	—	$O(1)$
Access by index	$O(1)$	$O(n)$	$O(n)$	—	—	—
Search	$O(n)$	$O(n)$	$O(n)$	—	—	—

* Amortized

9.6 Exercises

Exercise 7.1. Implement a function `isBalanced(expression: string): boolean` that uses a `Stack` to determine whether the parentheses `()`, brackets `[]`, and braces `{}` in an expression are properly balanced. For example, `isBalanced("((a+b)*[c-d])")` should return `false` (mismatched outer parentheses), while `isBalanced("{a*(b+c)}")` should return `true`.

Exercise 7.2. Implement a circular buffer–based queue. Use a fixed-size array and two indices (`front` and `back`) that wrap around using modular arithmetic. Compare its performance characteristics with our linked-list–based `Queue`.

Exercise 7.3. Implement a `MinStack<T>` that supports `push`, `pop`, `peek`, and an additional `min()` operation that returns the minimum element in the stack — all in $O(1)$ time. *Hint:* maintain a second stack that tracks minimums.

Exercise 7.4. Using only two `Stacks`, implement a `Queue`. Analyze the amortized time complexity of `enqueue` and `dequeue`. *Hint:* use one stack for enqueuing and another for dequeuing; transfer elements between them lazily.

Exercise 7.5. Implement a function `slidingWindowMax(arr: number[], k: number): number[]` that returns the maximum value in each window of size k as the window slides from left to right across the array. Use a `Deque` to achieve $O(n)$ time complexity.

9.7 Summary

This chapter introduced the foundational data structures upon which nearly everything else is built:

- **Dynamic arrays** provide $O(1)$ random access and $O(1)$ amortized append via the doubling strategy. Insert and remove at arbitrary positions cost $O(n)$ due to shifting.
- **Singly linked lists** offer $O(1)$ insertion and removal at the head, and $O(1)$ append with a tail pointer, but sacrifice random access and efficient removal from the tail.
- **Doubly linked lists** add back-pointers for $O(1)$ removal at both ends, at the cost of extra memory per node.
- **Stacks** (LIFO) are the workhorse of recursion, expression evaluation, and depth-first search.
- **Queues** (FIFO) power breadth-first search, task scheduling, and buffering.
- **Deques** generalize stacks and queues, supporting $O(1)$ operations at both ends.

The choice between arrays and linked lists comes down to access patterns. If you need random access or sequential iteration (where cache locality matters), use an array. If insertions and deletions at the endpoints dominate, use a linked list. When in doubt, the dynamic array is usually the right default — it is what most languages provide as their standard collection.

In the next chapter, we will use these building blocks to construct **hash tables**, which achieve expected $O(1)$ lookup by combining arrays with a hash function.

Chapter 10

Hash Tables

The data structures of the previous chapter — arrays, linked lists, stacks, and queues — support searching in $O(n)$ time at best. Binary search trees (which we will study in Chapter 9) reduce this to $O(\log n)$, but can we do even better? Hash tables achieve expected $O(1)$ time for insertions, deletions, and lookups by using a hash function to compute the index where each element should be stored. This makes hash tables one of the most important and widely used data structures in software engineering. In this chapter we explore how hash functions work, how to handle collisions when two keys map to the same index, and how to build hash tables that resize dynamically to maintain their performance guarantees.

10.1 The dictionary problem

Many problems reduce to maintaining a collection of key-value pairs that supports three operations:

- **Insert** a new key-value pair (or update the value if the key exists).
- **Lookup** the value associated with a given key.
- **Delete** a key-value pair.

This is the **dictionary** abstract data type (also called a **map** or **associative array**). JavaScript's built-in `Map` and Python's `dict` are both dictionaries backed by hash tables.

10.1.1 Direct addressing

The simplest approach is **direct addressing**: use the key itself as an index into an array. If keys are integers in the range $[0, m-1]$, we allocate an array of size m and store the value for key k at index k. All three operations are $O(1)$.

```typescript
// Direct-address table for integer keys in [0, m-1]
const table = new Array<string | undefined>(1000);
table[42] = 'Alice';         // insert
const name = table[42];      // lookup — O(1)
table[42] = undefined;       // delete
```

Direct addressing has a fatal flaw: the key space must be small and dense. If keys are strings, or integers in the range $[0, 2^{32} - 1]$, allocating an array large enough to hold every possible key is impractical. We need a way to map a large key space into a small array.

10.2 Hash functions

A **hash function** h maps keys from a large universe U to indices in a table of size m:

$$h : U \to \{0, 1, \ldots, m - 1\}$$

Given a key k, the hash function computes $h(k)$, which is the index (or **bucket**) where the key should be stored. A good hash function has two properties:

1. **Determinism.** The same key always produces the same hash.
2. **Uniformity.** Different keys should spread as evenly as possible across the m buckets, minimizing collisions.

10.2.1 The division method

The simplest hash function for integer keys is the **division method**:

$$h(k) = k \bmod m$$

This maps any non-negative integer to $[0, m - 1]$. The choice of m matters: if m is a power of 2, the hash uses only the lowest-order bits of k, which can lead to clustering. Prime values of m tend to distribute keys more uniformly.

10.2.2 The multiplication method

The **multiplication method** avoids the sensitivity to m:

$$h(k) = \lfloor m \cdot (k \cdot A \bmod 1) \rfloor$$

where A is a constant in the range $(0, 1)$. Knuth suggests $A \approx (\sqrt{5} - 1)/2 \approx 0.6180339887$. The expression $k \cdot A \bmod 1$ extracts the fractional part of $k \cdot A$, which is then scaled to $[0, m)$. This method works well regardless of whether m is a power of 2.

10.2.3 Hashing strings

For string keys, we need to convert a sequence of characters into an integer. A standard approach is a **polynomial rolling hash**:

$$h(s) = \left(\sum_{i=0}^{n-1} s[i] \cdot p^{n-1-i} \right) \bmod m$$

where $s[i]$ is the character code at position i, p is a prime base (often 31 or 37), and m is the table size. Variants of this idea include the FNV (Fowler–Noll–Vo) hash, which alternates XOR and multiplication to achieve good distribution with simple operations:

```
function fnvHash(key: string): number {
  let h = 0x811c9dc5; // FNV offset basis
  for (let i = 0; i < key.length; i++) {
    h ^= key.charCodeAt(i);
    h = Math.imul(h, 0x01000193); // FNV prime
  }
  return h >>> 0; // ensure non-negative 32-bit integer
}
```

The `>>> 0` at the end is a JavaScript idiom that converts a possibly negative 32-bit integer to an unsigned 32-bit integer, ensuring we get a non-negative result suitable for use as an array index.

10.2.4 Universal hashing

No single hash function can avoid collisions for every possible input. An adversary who knows the hash function can deliberately choose keys that all hash to the same bucket, degrading performance to $O(n)$.

Universal hashing defeats this by choosing the hash function randomly from a family of functions at construction time. A family $\mathcal{H}$ of hash functions from U to $\{0, \ldots, m-1\}$ is **universal** if, for any two distinct keys $x \neq y$:

$$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq \frac{1}{m}$$

When the hash function is chosen randomly, no input distribution can consistently cause collisions, giving us expected $O(1)$ performance regardless of the input.

10.3 Collision resolution

Since $|U| > m$, multiple keys will inevitably hash to the same bucket — a **collision**. The two primary strategies for handling collisions are **separate chaining** and **open addressing**.

10.4 Separate chaining

In **separate chaining**, each bucket stores a linked list (or chain) of all key-value pairs that hash to that index. Insertions prepend to the chain; lookups and deletions walk the chain until the key is found.

10.4.1 How it works

Consider a hash table with $m = 4$ buckets after inserting keys with hashes as shown:

```
Bucket 0: → (key₁, val₁) → (key₅, val₅) → null
Bucket 1: → (key₂, val₂) → null
Bucket 2: → null
Bucket 3: → (key₃, val₃) → (key₄, val₄) → null
```

Keys 1 and 5 collide at bucket 0; keys 3 and 4 collide at bucket 3. Lookups for key_5 must traverse two nodes in bucket 0.

10.4.2 Load factor

The **load factor** $\alpha = n/m$ is the average number of elements per bucket, where n is the number of stored entries and m is the number of buckets. Under the **simple uniform hashing assumption** (each key is equally likely to hash to any bucket), the expected chain length is α.

- If α is kept constant (say, $\alpha \leq 0.75$), the expected time for any operation is $O(1 + \alpha) = O(1)$.
- If we never resize, α grows with n, and operations degrade to $O(n)$.

10.4.3 Implementation

Our `HashTableChaining<K, V>` maintains an array of bucket heads (each either a chain node or `null`) and doubles the array when $\alpha \geq 0.75$:

```
class ChainNode<K, V> {
  constructor(
    public key: K,
    public value: V,
    public next: ChainNode<K, V> | null = null,
  ) {}
}

export class HashTableChaining<K, V> implements Iterable<[K, V]> {
  private buckets: (ChainNode<K, V> | null)[];
  private count = 0;

  constructor(initialCapacity = 16) {
    const cap = Math.max(1, initialCapacity);
    this.buckets = new Array<ChainNode<K, V> | null>(cap).fill(null);
  }

  get size(): number {
    return this.count;
  }

  get capacity(): number {
    return this.buckets.length;
  }

  get loadFactor(): number {
    return this.count / this.buckets.length;
  }
```

The set method searches the chain at the target bucket. If the key is found, its value is updated; otherwise a new node is prepended:

```
  set(key: K, value: V): V | undefined {
    if (this.count / this.buckets.length >= 0.75) {
      this.resize(this.buckets.length * 2);
    }

    const idx = this.bucketIndex(key);
    let node: ChainNode<K, V> | null = this.buckets[idx]!;

    while (node !== null) {
      if (Object.is(node.key, key)) {
```

```
      const old = node.value;
      node.value = value;
      return old;
    }
    node = node.next;
  }

  // Prepend to the bucket chain
  this.buckets[idx] = new ChainNode(key, value, this.buckets[idx]!);
  this.count++;
  return undefined;
}
```

The `get` and `delete` methods follow the same pattern — compute the bucket index, then walk the chain:

```
get(key: K): V | undefined {
  const idx = this.bucketIndex(key);
  let node: ChainNode<K, V> | null = this.buckets[idx]!;
  while (node !== null) {
    if (Object.is(node.key, key)) {
      return node.value;
    }
    node = node.next;
  }
  return undefined;
}

delete(key: K): boolean {
  const idx = this.bucketIndex(key);
  let node: ChainNode<K, V> | null = this.buckets[idx]!;
  let prev: ChainNode<K, V> | null = null;

  while (node !== null) {
    if (Object.is(node.key, key)) {
      if (prev !== null) {
        prev.next = node.next;
      } else {
        this.buckets[idx] = node.next;
      }
      this.count--;
      return true;
```

```
      }
      prev = node;
      node = node.next;
    }
    return false;
  }
```

We use `Object.is` for key comparison rather than `===` because `Object.is` correctly handles the edge case where `NaN === NaN` is `false` but we want `NaN` keys to match.

10.4.4 Dynamic resizing

When the load factor reaches the threshold, we allocate a new array with double the capacity and rehash every entry:

```
  private resize(newCapacity: number): void {
    const oldBuckets = this.buckets;
    this.buckets = new Array<ChainNode<K, V> | null>(newCapacity).fill(null);
    this.count = 0;

    for (let b = 0; b < oldBuckets.length; b++) {
      let node: ChainNode<K, V> | null = oldBuckets[b]!;
      while (node !== null) {
        this.set(node.key, node.value);
        node = node.next;
      }
    }
  }
```

Resizing costs $O(n)$ in the worst case, but by the same amortized argument as dynamic arrays (Chapter 7), the cost per insertion averages $O(1)$ over a sequence of operations.

10.4.5 Tracing through an example

Let us trace insertions into a hash table with 4 buckets. We use a simple hash $h(k) = k \bmod 4$ for clarity:

Operation	Hash	Bucket state	Size
`set(5, "a")`	$5 \bmod 4 = 1$	B1: `(5,"a")`	1
`set(9, "b")`	$9 \bmod 4 = 1$	B1: `(9,"b")→(5,"a")`	2

Operation	Hash	Bucket state	Size
`set(3, "c")`	3 mod 4 = 3	B1: `(9,"b")→(5,"a")`, B3: `(3,"c")`	3
`set(5, "d")`	5 mod 4 = 1	B1: `(9,"b")→(5,"d")` — value updated	3
`delete(9)`	9 mod 4 = 1	B1: `(5,"d")`	2

Keys 5 and 9 collide at bucket 1. Setting key 5 again updates its value without increasing the size. Deleting key 9 removes it from the chain.

10.5 Open addressing

In **open addressing**, all entries are stored directly in the table array — there are no linked lists. When a collision occurs, we probe a sequence of alternative slots until an empty one is found.

The **probe sequence** for key k is a permutation of the table indices:

$$h(k, 0),\ h(k, 1),\ h(k, 2),\ \ldots,\ h(k, m-1)$$

We try slot $h(k, 0)$ first; if it is occupied, we try $h(k, 1)$, and so on.

10.5.1 Linear probing

The simplest probing strategy is **linear probing**:

$$h(k, i) = (h'(k) + i) \bmod m$$

where $h'(k)$ is the primary hash. This means we simply try the next slot, then the one after that, wrapping around the end of the array.

Linear probing is cache-friendly because it accesses consecutive memory locations. However, it suffers from **primary clustering**: a contiguous block of occupied slots tends to grow, because any key that hashes into the cluster must probe to its end. Long clusters slow down both insertions and lookups.

10.5.2 Double hashing

Double hashing uses a second hash function to compute the probe step:

$$h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$$

where h_1 is the primary hash and h_2 determines the step size. Different keys that collide at h_1 will have different step sizes, breaking up clusters.

For double hashing to work correctly, $h_2(k)$ must be coprime to m so that the probe sequence visits every slot. A common choice is to make m a power of 2 and ensure $h_2(k)$ is always odd.

10.5.3 Tombstones and lazy deletion

Deleting from an open-addressed table is tricky. Simply clearing a slot would break probe sequences: if key A was placed after probing past slot s (which held key B), clearing slot s would make A unreachable.

The solution is **lazy deletion** with **tombstones**. When we delete a key, we mark its slot with a special sentinel value (the tombstone). During lookups, tombstones are treated as occupied (we continue probing past them). During insertions, tombstones can be reused.

```
Slot 0: ── (key₁, val₁)
Slot 1: ── TOMBSTONE       ← deleted entry
Slot 2: ── (key₃, val₃)   ← still reachable past tombstone
Slot 3: ── empty
```

Over time, tombstones accumulate and degrade performance. When we resize the table, tombstones are discarded, restoring clean probe sequences.

10.5.4 Load factor for open addressing

Open addressing is more sensitive to load factor than chaining. As the table fills up, probe sequences get longer. At load factor α, the expected number of probes for an unsuccessful search under uniform hashing is:

$$\frac{1}{1-\alpha}$$

At $\alpha = 0.5$, this is 2 probes. At $\alpha = 0.75$, it is 4. At $\alpha = 0.9$, it is 10. For this reason, open-addressed tables typically resize at $\alpha \leq 0.5$ — more aggressively than chaining tables.

10.5.5 Implementation

Our `HashTableOpenAddressing<K, V>` supports both linear probing and double hashing:

```
const TOMBSTONE = Symbol('TOMBSTONE');
```

```
interface Slot<K, V> {
  key: K;
  value: V;
}

type BucketEntry<K, V> = Slot<K, V> | typeof TOMBSTONE | undefined;

export class HashTableOpenAddressing<K, V> implements Iterable<[K, V]> {
  private slots: BucketEntry<K, V>[];
  private count = 0;
  private tombstoneCount = 0;
  private readonly strategy: 'linear' | 'double-hashing';

  constructor(
    initialCapacity = 16,
    strategy: 'linear' | 'double-hashing' = 'linear',
  ) {
    this.strategy = strategy;
    const cap = nextPowerOf2(Math.max(1, initialCapacity));
    this.slots = new Array<BucketEntry<K, V>>(cap);
  }
```

The set method probes for an empty slot or a matching key:

```
  set(key: K, value: V): V | undefined {
    if ((this.count + this.tombstoneCount) / this.slots.length >= 0.5) {
      this.rebuild(this.slots.length * 2);
    }

    const cap = this.slots.length;
    const h1 = primaryHash(key) % cap;
    const step = this.strategy === 'double-hashing'
      ? secondaryHash(key, cap) : 1;

    let firstTombstone = -1;
    let idx = h1;

    for (let i = 0; i < cap; i++) {
      const slot = this.slots[idx];

      if (slot === undefined) {
        const insertIdx = firstTombstone !== -1 ? firstTombstone : idx;
```

```
        this.slots[insertIdx] = { key, value };
        this.count++;
        if (firstTombstone !== -1) this.tombstoneCount--;
        return undefined;
      }

      if (slot === TOMBSTONE) {
        if (firstTombstone === -1) firstTombstone = idx;
      } else if (Object.is(slot.key, key)) {
        const old = slot.value;
        slot.value = value;
        return old;
      }

      idx = (idx + step) % cap;
    }
  }
```

Notice the `firstTombstone` optimization: if we pass a tombstone during the probe sequence, we remember its position. If the key is not in the table, we insert at the first tombstone rather than probing all the way to an empty slot. This recycles tombstones and prevents them from accumulating.

The resize check counts both live entries and tombstones against the load threshold. When we rebuild, tombstones are discarded:

```
  private rebuild(newCapacity: number): void {
    const cap = nextPowerOf2(Math.max(1, newCapacity));
    const oldSlots = this.slots;
    this.slots = new Array<BucketEntry<K, V>>(cap);
    this.count = 0;
    this.tombstoneCount = 0;

    for (const slot of oldSlots) {
      if (slot !== undefined && slot !== TOMBSTONE) {
        this.set(slot.key, slot.value);
      }
    }
  }
```

10.5.6 Tracing through linear probing

Let us trace insertions into a table of size 8 using linear probing with $h(k) = k \bmod 8$:

Operation	Hash	Probes	Result
`set(3, "a")`	3	3	Slot 3 ← `(3,"a")`
`set(11, "b")`	3	3→4	Collision at 3, slot 4 ← `(11,"b")`
`set(19, "c")`	3	3→4→5	Collision at 3,4, slot 5 ← `(19,"c")`
`delete(11)`	3	3→4	Slot 4 ← TOMBSTONE
`get(19)`	3	3→4→5	Probes past tombstone at 4, finds at 5
`set(27, "d")`	3	3→4	Reuses tombstone at 4 ← `(27,"d")`

The tombstone at slot 4 ensures that `get(19)` does not stop prematurely after passing the deleted slot.

10.6 Chaining vs open addressing

Property	Chaining	Open addressing
Extra memory	Linked list nodes	None (entries in table)
Cache performance	Poor (pointer chasing)	Good (sequential probes)
Load factor tolerance	Works well up to $\alpha \approx 1$	Degrades rapidly above $\alpha \approx 0.7$
Deletion	Simple	Requires tombstones
Worst case (all collisions)	$O(n)$	$O(n)$
Implementation complexity	Simpler	More subtle

In practice, open addressing with linear probing tends to outperform chaining for moderate load factors thanks to cache locality. Chaining is more forgiving when the load factor varies or when deletions are frequent. Modern high-performance hash maps (like Google's SwissTable or Rust's `HashMap`) use sophisticated open-addressing schemes with SIMD-accelerated probing.

10.7 Applications

Hash tables are ubiquitous. Here are a few classic applications:

10.7.1 Frequency counting

Count how many times each word appears in a text:

```
function wordFrequency(words: string[]): Map<string, number> {
  const freq = new Map<string, number>();
  for (const word of words) {
    freq.set(word, (freq.get(word) ?? 0) + 1);
  }
  return freq;
}
```

This runs in $O(n)$ expected time, where n is the number of words. Without a hash table, we would need $O(n \log n)$ (sorting) or $O(n^2)$ (brute force).

10.7.2 Two-sum problem

Given an array of numbers and a target sum, find two elements that add up to the target:

```
function twoSum(nums: number[], target: number): [number, number] | null {
  const seen = new Map<number, number>(); // value → index
  for (let i = 0; i < nums.length; i++) {
    const complement = target - nums[i];
    const j = seen.get(complement);
    if (j !== undefined) return [j, i];
    seen.set(nums[i], i);
  }
  return null;
}
```

Each element is inserted and looked up once, giving $O(n)$ expected time.

10.7.3 Anagram detection

Two strings are anagrams if they contain the same characters with the same frequencies. We can check this by counting character frequencies in both strings and comparing:

```
function areAnagrams(a: string, b: string): boolean {
  if (a.length !== b.length) return false;
  const counts = new Map<string, number>();
  for (const ch of a) counts.set(ch, (counts.get(ch) ?? 0) + 1);
  for (const ch of b) {
    const c = (counts.get(ch) ?? 0) - 1;
```

```
    if (c < 0) return false;
    counts.set(ch, c);
  }
  return true;
}
```

This is $O(n)$ where n is the string length, versus $O(n \log n)$ for sorting both strings and comparing.

10.7.4 Deduplication

Remove duplicate elements from an array while preserving order:

```
function deduplicate<T>(arr: T[]): T[] {
  const seen = new Set<T>();
  const result: T[] = [];
  for (const item of arr) {
    if (!seen.has(item)) {
      seen.add(item);
      result.push(item);
    }
  }
  return result;
}
```

A `Set` is essentially a hash table that stores only keys (no values).

10.8 Complexity summary

Operation	Chaining (expected)	Chaining (worst)	Open addressing (expected)	Open addressing (worst)
Insert	$O(1)$	$O(n)$	$O(1)$	$O(n)$
Lookup	$O(1)$	$O(n)$	$O(1)$	$O(n)$
Delete	$O(1)$	$O(n)$	$O(1)$	$O(n)$
Space	$O(n + m)$	—	$O(m)$	—

The expected $O(1)$ complexities hold under the assumptions that the hash function distributes keys uniformly and the load factor is bounded by a constant.

10.9 Exercises

Exercise 8.1. Implement a function `groupAnagrams(words: string[]): string[][]` that groups an array of words into sub-arrays of anagrams. For example, `groupAnagrams(["eat", "tea", "tan", "ate", "nat", "bat"])` should return `[["eat", "tea", "ate"], ["tan", "nat"], ["bat"]]` (in any order). Use a hash table where the key is the sorted characters of each word.

Exercise 8.2. Our open-addressing implementation uses a load factor threshold of 0.5 and doubles the table when exceeded. Experiment with different thresholds (0.6, 0.7, 0.8) and measure the average number of probes per lookup on random data. At what point does performance degrade noticeably?

Exercise 8.3. Implement a `HashSet<T>` class backed by `HashTableChaining<T, boolean>`. Support `add`, `has`, `delete`, `size`, and iteration. How does this compare to using TypeScript's built-in `Set`?

Exercise 8.4. The **cuckoo hashing** scheme uses two hash functions and two tables. Each key has exactly two possible locations — one in each table. If both are occupied during an insertion, one of the existing keys is "kicked out" and re-inserted using its alternate location. Research cuckoo hashing and explain: (a) why lookup is $O(1)$ worst case, (b) under what conditions insertion might fail, and (c) how to handle insertion failures.

Exercise 8.5. Our hash function uses FNV-1a for strings and a bit-mixing scheme for numbers. Design an experiment to test how uniformly these functions distribute keys. Generate 10,000 random strings (and separately, 10,000 random integers), hash each into a table of 1,000 buckets, and compute the chi-squared statistic. Compare with a theoretically perfect uniform distribution.

10.10 Summary

Hash tables achieve expected $O(1)$ time for insert, lookup, and delete by using a hash function to map keys to array indices. The two main collision resolution strategies are:

- **Separate chaining** stores colliding entries in linked lists at each bucket. It is simple, tolerates high load factors, and handles deletions cleanly. The cost is extra memory for list nodes and poor cache locality.
- **Open addressing** stores all entries directly in the table array, probing for alternative slots on collision. Linear probing is cache-friendly but susceptible to clustering; double hashing eliminates clustering at the cost of additional hash computations. Deletions require tombstones to preserve probe sequences.

The **load factor** $\alpha = n/m$ controls performance. Chaining tables typically resize at $\alpha \geq 0.75$; open-addressed tables at $\alpha \geq 0.5$. Dynamic resizing (doubling the table and rehashing all entries) maintains the load factor within bounds, giving amortized $O(1)$ insertions.

Hash tables are the backbone of frequency counting, deduplication, two-sum–style problems, caching, and countless other applications. Their expected $O(1)$ operations make them the go-to data structure whenever fast key-based access is needed — though their worst-case $O(n)$ behavior means they are not a substitute for balanced search trees when guaranteed performance is required.

In the next chapter, we study **trees and binary search trees**, which provide $O(\log n)$ worst-case operations and support order-based queries that hash tables cannot efficiently answer.

Chapter 11

Trees and Binary Search Trees

Hash tables give us expected $O(1)$ lookups, but they cannot answer order-based queries: what is the smallest key? What is the next key after k? What are all keys in the range $[a, b]$? Trees restore this capability. A binary search tree stores elements in a way that mirrors binary search — at every node, all smaller elements are to the left and all larger elements are to the right. This gives us $O(h)$ search, insert, and delete operations, where h is the height of the tree. In this chapter we develop the fundamental vocabulary of trees, study the four standard traversal orders, and build a complete binary search tree implementation.

11.1 Tree terminology

A **tree** is a connected, acyclic graph. In computer science we almost always work with **rooted trees**, where one node is designated as the **root** and all other nodes are arranged in a parent-child hierarchy descending from it.

Key definitions:

- **Node**: an element of the tree, containing a value and links to its children.
- **Root**: the topmost node; it has no parent.
- **Parent**: the node directly above a given node.
- **Child**: a node directly below a given node.
- **Leaf**: a node with no children (also called an **external node**).
- **Internal node**: a node with at least one child.
- **Sibling**: nodes that share the same parent.
- **Subtree**: the tree rooted at a given node, consisting of that node and all its descendants.
- **Depth** of a node: the number of edges from the root to that node. The root has depth 0.

- **Height** of a node: the number of edges on the longest path from that node down to a leaf. A leaf has height 0.
- **Height of the tree**: the height of the root. An empty tree has height -1 by convention.
- **Level** k: the set of all nodes at depth k.
- **Degree** of a node: the number of children it has.

11.2 Binary trees

A **binary tree** is a tree in which every node has at most two children, called the **left child** and the **right child**. Binary trees are the most fundamental tree structure in computer science, underpinning search trees, heaps, expression parsers, and many other data structures.

11.2.1 Representations

There are two common ways to represent a binary tree:

Linked representation. Each node is an object with a value and two pointers (`left` and `right`). This is the most flexible representation and the one we use throughout this book:

```
class BinaryTreeNode<T> {
  constructor(
    public value: T,
    public left: BinaryTreeNode<T> | null = null,
    public right: BinaryTreeNode<T> | null = null,
  ) {}
}
```

Array representation. For a complete binary tree (where every level except possibly the last is fully filled), we can store nodes in an array by level order. The root is at index 0, and for a node at index i:

- Left child: $2i + 1$
- Right child: $2i + 2$
- Parent: $\lfloor (i - 1)/2 \rfloor$

This representation avoids pointer overhead and is used for binary heaps (Chapter 11).

11.2.2 Properties of binary trees

A binary tree of height h has:

- At most $2^{h+1} - 1$ nodes (when every level is full — a **perfect** binary tree).
- At least $h + 1$ nodes (when every internal node has exactly one child — a **degenerate** or **skewed** tree).
- At most 2^h leaves.

A binary tree with n nodes has height between $\lfloor \log_2 n \rfloor$ and $n - 1$.

11.3 Tree traversals

A **traversal** visits every node in the tree exactly once. The order of visitation defines the traversal type. For a binary tree, there are four standard traversals.

11.3.1 Inorder traversal (left, root, right)

Visit the left subtree, then the root, then the right subtree. For a binary search tree, inorder traversal produces values in sorted order.


```
Inorder: 4, 2, 5, 1, 3, 6
```

```
inorder(): T[] {
  const result: T[] = [];
  this.inorderHelper(this.root, result);
  return result;
}

private inorderHelper(node: BinaryTreeNode<T> | null, result: T[]): void {
  if (node === null) return;
  this.inorderHelper(node.left, result);
  result.push(node.value);
  this.inorderHelper(node.right, result);
}
```

The recursion mirrors the traversal definition directly: recurse left, process the current node, recurse right.

11.3.2 Preorder traversal (root, left, right)

Visit the root first, then the left subtree, then the right subtree. Preorder traversal is useful for serializing a tree (e.g., to reconstruct it later) because the root always comes before its children.

```
Preorder: 1, 2, 4, 5, 3, 6
```

```
private preorderHelper(node: BinaryTreeNode<T> | null, result: T[]): void {
  if (node === null) return;
  result.push(node.value);
  this.preorderHelper(node.left, result);
  this.preorderHelper(node.right, result);
}
```

11.3.3 Postorder traversal (left, right, root)

Visit the left subtree, then the right subtree, then the root. Postorder traversal processes children before their parent, making it useful for deleting a tree (free children before the parent) or evaluating expression trees (evaluate operands before the operator).

```
Postorder: 4, 5, 2, 6, 3, 1
```

```
private postorderHelper(node: BinaryTreeNode<T> | null, result: T[]): void {
  if (node === null) return;
  this.postorderHelper(node.left, result);
  this.postorderHelper(node.right, result);
  result.push(node.value);
}
```

11.3.4 Level-order traversal (breadth-first)

Visit nodes level by level, from left to right. Unlike the three depth-first traversals above, level-order traversal uses a queue rather than recursion:

```
Level-order: 1, 2, 3, 4, 5, 6
```

```
levelOrder(): T[] {
  if (this.root === null) return [];

  const result: T[] = [];
  const queue: BinaryTreeNode<T>[] = [this.root];

  while (queue.length > 0) {
```

```
    const node = queue.shift()!;
    result.push(node.value);
    if (node.left !== null) queue.push(node.left);
    if (node.right !== null) queue.push(node.right);
  }

  return result;
}
```

We enqueue the root, then repeatedly dequeue a node, process it, and enqueue its children. Since every node is enqueued and dequeued exactly once, the traversal is $O(n)$.

11.3.5 Complexity of traversals

All four traversals visit every node exactly once, so they run in $O(n)$ time. The space complexity depends on the traversal:

- **Recursive traversals** (inorder, preorder, postorder): $O(h)$ stack space, where h is the tree height. For a balanced tree this is $O(\log n)$; for a skewed tree it is $O(n)$.
- **Level-order traversal**: $O(w)$ space for the queue, where w is the maximum width (number of nodes at any single level). For a complete binary tree, the last level has up to $n/2$ nodes, so the space is $O(n)$.

11.3.6 Computing height and size

The **height** of a tree is computed recursively: the height of an empty tree is -1, and the height of a non-empty tree is one plus the maximum of the heights of its subtrees:

```
private heightHelper(node: BinaryTreeNode<T> | null): number {
  if (node === null) return -1;
  return 1 + Math.max(
    this.heightHelper(node.left),
    this.heightHelper(node.right),
  );
}
```

The **size** (number of nodes) is similarly recursive:

```
private sizeHelper(node: BinaryTreeNode<T> | null): number {
  if (node === null) return 0;
```

```
  return 1 + this.sizeHelper(node.left) + this.sizeHelper(node.right);
}
```

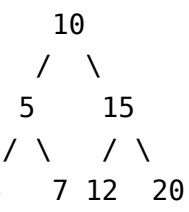
Both run in $O(n)$ time by visiting every node.

11.4 Binary search trees

A **binary search tree** (BST) is a binary tree that satisfies the **BST property**: for every node x,

- all values in x's left subtree are less than x's value, and
- all values in x's right subtree are greater than or equal to x's value.

This property makes the tree a natural implementation of the dictionary abstract data type (Chapter 8), with the added ability to answer order-based queries.



Every node in the left subtree of 10 (namely 3, 5, 7) is less than 10, and every node in the right subtree (12, 15, 20) is greater.

11.4.1 BST node structure

Our BST nodes carry parent pointers, which simplify the successor and predecessor algorithms:

```
class BSTNode<T> {
  constructor(
    public value: T,
    public left: BSTNode<T> | null = null,
    public right: BSTNode<T> | null = null,
    public parent: BSTNode<T> | null = null,
  ) {}
}
```

The `parent` pointer costs one extra reference per node but eliminates the need to maintain an explicit stack when walking up the tree.

11.4.2 Search

To search for a value v, start at the root and compare v with the current node's value. If v is smaller, go left; if larger, go right; if equal, the node is found. If we reach a null pointer, the value is not in the tree.

```
search(value: T): BSTNode<T> | null {
  let current = this.root;
  while (current !== null) {
    const cmp = this.compare(value, current.value);
    if (cmp === 0) return current;
    current = cmp < 0 ? current.left : current.right;
  }
  return null;
}
```

This is exactly binary search applied to a tree structure. At each step we eliminate one subtree, following a single root-to-leaf path. The running time is $O(h)$ where h is the height of the tree.

11.4.3 Insert

To insert a value, we walk the tree as in search until we reach a null position, then place the new node there:

```
insert(value: T): void {
  const newNode = new BSTNode(value);

  if (this.root === null) {
    this.root = newNode;
    return;
  }

  let current = this.root;
  for (;;) {
    if (this.compare(value, current.value) < 0) {
      if (current.left === null) {
        current.left = newNode;
        newNode.parent = current;
        return;
      }
      current = current.left;
    } else {
      if (current.right === null) {
```

```
        current.right = newNode;
        newNode.parent = current;
        return;
      }
      current = current.right;
    }
  }
}
```

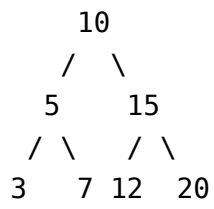
Insertion always adds a new leaf, so the tree's shape depends on the order of insertions. Inserting values in sorted order creates a degenerate (right-skewed) tree of height $n-1$, while inserting in random order produces a tree of expected height $O(\log n)$.

11.4.4 Tracing through insertions

Let us trace the insertion of values 10, 5, 15, 3, 7, 12, 20:

Insert	Tree state
10	`10` — root
5	`10` ← `5` goes left (5 < 10)
15	`10` → `15` goes right (15 ≥ 10)
3	`5` ← `3` goes left (3 < 5)
7	`5` → `7` goes right (7 ≥ 5)
12	`15` ← `12` goes left (12 < 15)
20	`15` → `20` goes right (20 ≥ 15)

The result is a balanced tree of height 2:



If instead we inserted 3, 5, 7, 10, 12, 15, 20 (sorted order), each value would go to the right of the previous one, producing a right-skewed linked list of height 6. This is why balanced BST variants (Chapter 10) are important.

11.4.5 Minimum and maximum

The minimum value in a BST is the leftmost node; the maximum is the rightmost:

```
private minNode(node: BSTNode<T> | null): BSTNode<T> | null {
  if (node === null) return null;
  while (node.left !== null) {
    node = node.left;
  }
  return node;
}

private maxNode(node: BSTNode<T> | null): BSTNode<T> | null {
  if (node === null) return null;
  while (node.right !== null) {
    node = node.right;
  }
  return node;
}
```

Both follow a single path from the given node to a leaf, so they run in $O(h)$ time.

11.4.6 Successor and predecessor

The **in-order successor** of a node x is the node with the smallest value greater than x's value — the next element in sorted order. The **predecessor** is the node with the largest value smaller than x's.

Finding the successor has two cases:

1. **If x has a right subtree**, the successor is the minimum of that subtree (the leftmost node in the right subtree).
2. **If x has no right subtree**, the successor is the lowest ancestor of x whose left child is also an ancestor of x. Intuitively, we walk up the tree until we turn right — the node where we turn is the successor.

```
private successorNode(node: BSTNode<T>): BSTNode<T> | null {
  if (node.right !== null) {
    return this.minNode(node.right);
  }
  let current: BSTNode<T> | null = node;
  let parent = current.parent;
  while (parent !== null && current === parent.right) {
    current = parent;
    parent = parent.parent;
  }
  return parent;
```

```
}
```

The predecessor is symmetric: if x has a left subtree, the predecessor is the maximum of that subtree; otherwise walk up until we turn left.

```
private predecessorNode(node: BSTNode<T>): BSTNode<T> | null {
  if (node.left !== null) {
    return this.maxNode(node.left);
  }
  let current: BSTNode<T> | null = node;
  let parent = current.parent;
  while (parent !== null && current === parent.left) {
    current = parent;
    parent = parent.parent;
  }
  return parent;
}
```

Both operations follow at most one root-to-leaf path, so they are $O(h)$.

11.4.7 Tracing successor

Consider the tree:

- **Successor of 7**: 7 has no right subtree. Walk up: 7 is the right child of 5, so continue. 5 is the left child of 10 — stop. The successor is 10.
- **Successor of 10**: 10 has a right subtree rooted at 15. The minimum of that subtree is 12. The successor is 12.
- **Successor of 20**: 20 has no right subtree. Walk up: 20 is the right child of 15, 15 is the right child of 10, 10 has no parent. No successor exists (20 is the maximum).

11.4.8 Delete

Deletion is the most complex BST operation because removing a node must preserve the BST property. There are three cases:

Case 1: The node is a leaf (no children). Simply remove it by setting the parent's pointer to null.

Case 2: The node has one child. Replace the node with its only child. The child takes the node's position in the tree.

Case 3: The node has two children. Find the node's in-order successor (the minimum of the right subtree). Copy the successor's value into the node, then delete the successor. The successor has at most one child (a right child), so its deletion reduces to Case 1 or 2.

The implementation uses a helper called `transplant` (following CLRS) that replaces one subtree with another:

```
private transplant(u: BSTNode<T>, v: BSTNode<T> | null): void {
  if (u.parent === null) {
    this.root = v;
  } else if (u === u.parent.left) {
    u.parent.left = v;
  } else {
    u.parent.right = v;
  }
  if (v !== null) {
    v.parent = u.parent;
  }
}
```

`transplant(u, v)` replaces the subtree rooted at u with the subtree rooted at v. It updates the parent of u to point to v and sets v's parent pointer.

The full deletion procedure:

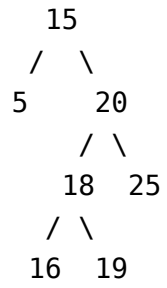
```
private deleteNode(node: BSTNode<T>): void {
  if (node.left === null) {
    // Case 1 or 2a: no left child
    this.transplant(node, node.right);
  } else if (node.right === null) {
    // Case 2b: no right child
    this.transplant(node, node.left);
  } else {
    // Case 3: two children
    const successor = this.minNode(node.right)!;
    if (successor.parent !== node) {
      this.transplant(successor, successor.right);
      successor.right = node.right;
      successor.right.parent = successor;
    }
    this.transplant(node, successor);
```

```
    successor.left = node.left;
    successor.left.parent = successor;
  }
}
```

In Case 3, we find the successor (the minimum of the right subtree). If the successor is not the immediate right child of the node being deleted, we first detach the successor from its current position (transplanting its right child into its place), then connect the node's right subtree to the successor. Finally, we transplant the successor into the deleted node's position and connect the left subtree.

11.4.9 Tracing deletion

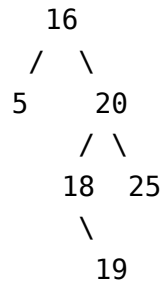
Starting with:



Delete 15 (two children, successor = 16):

1. Successor of 15 is 16 (minimum of right subtree).
2. 16 is not the immediate right child of 15, so first transplant 16 out: 16 has no right child, so its parent (18) gets null as left child.
3. Connect 20's subtree to 16: `16.right = 20`, `20.parent = 16`.
4. Transplant 16 into 15's position: 16 becomes the root.
5. Connect 15's left subtree to 16: `16.left = 5`, `5.parent = 16`.

Result:



The BST property is preserved: $5 < 16$, and all of 18, 19, 20, 25 are greater than 16.

11.5 BST performance analysis

Every operation (search, insert, delete, min, max, successor, predecessor) follows at most one root-to-leaf path, so all run in $O(h)$ time where h is the tree height.

The height depends on the insertion order:

Scenario	Height h	Operation time
Balanced tree (n nodes)	$O(\log n)$	$O(\log n)$
Random insertion order (expected)	$O(\log n)$	$O(\log n)$
Sorted insertion order (worst case)	$O(n)$	$O(n)$

For random insertions, the expected height of a BST with n nodes is approximately $4.311 \ln n$ (a result due to Reed, 2003). This means that on average, a plain BST performs well. However, the worst case is $O(n)$, which is no better than a linked list.

To guarantee $O(\log n)$ operations regardless of insertion order, we need **balanced binary search trees** — trees that automatically restructure themselves to maintain low height. AVL trees and red-black trees (Chapter 10) achieve this guarantee with a constant-factor overhead per operation.

11.5.1 BST vs hash table

Property	BST	Hash table
Search	$O(h)$	$O(1)$ expected
Insert	$O(h)$	$O(1)$ expected
Delete	$O(h)$	$O(1)$ expected
Min / Max	$O(h)$	$O(n)$
Successor / Predecessor	$O(h)$	$O(n)$
Sorted traversal	$O(n)$	$O(n \log n)$ (sort first)
Range query $[a, b]$	$O(h + k)$	$O(n)$

Hash tables are faster for pure lookup workloads, but BSTs support order-based operations that hash tables cannot efficiently provide. When you need sorted iteration, range queries, or finding the nearest key, a BST (especially a balanced one) is the right choice.

11.6 Complexity summary

Operation	Time (average)	Time (worst)	Space
Search	$O(\log n)$	$O(n)$	$O(1)$
Insert	$O(\log n)$	$O(n)$	$O(1)$
Delete	$O(\log n)$	$O(n)$	$O(1)$
Min / Max	$O(\log n)$	$O(n)$	$O(1)$
Successor / Predecessor	$O(\log n)$	$O(n)$	$O(1)$
Inorder traversal	$O(n)$	$O(n)$	$O(h)$
Space (tree itself)	—	—	$O(n)$

The "average" column assumes random insertion order. The "worst" column covers sorted or adversarial insertion order, which produces a degenerate tree.

11.7 Exercises

Exercise 9.1. Given the preorder traversal `[8, 3, 1, 6, 4, 7, 10, 14, 13]` of a BST, reconstruct the tree and write out the inorder and postorder traversals. Verify that the inorder traversal is sorted.

Exercise 9.2. Write an iterative (non-recursive) inorder traversal using an explicit stack. Compare its space usage with the recursive version. Under what circumstances might the iterative version be preferable?

Exercise 9.3. Prove that deleting a node from a BST using the successor-replacement method preserves the BST property. Specifically, argue that after replacing a two-children node with its in-order successor, every node in the left subtree is still less than the replacement, and every node in the right subtree is still greater.

Exercise 9.4. Write a function `isBST(root)` that checks whether a given binary tree satisfies the BST property. Your solution should run in $O(n)$ time. Be careful with the common pitfall of only checking immediate children — for example, the tree with root 10, left child 5, and left child's right child 15 violates the BST property even though each parent-child relationship individually looks correct.

Exercise 9.5. Implement a function `rangeQuery(bst, low, high)` that returns all values in the BST that fall within `[low, high]`, in sorted order. Your solution should run in $O(h + k)$ time where k is the number of values in the range, not $O(n)$. (Hint: adapt the inorder traversal to skip subtrees that cannot contain values in the range.)

11.8 Summary

Trees are hierarchical data structures where each node has a value and links to its children. Binary trees restrict each node to at most two children, and support four standard traversals: inorder (left-root-right), preorder (root-left-right), postorder (left-right-root), and level-order (breadth-first). All traversals run in $O(n)$ time.

A **binary search tree** augments the binary tree with the BST property: left subtree values are less than the node's value, and right subtree values are greater. This enables $O(h)$ search, insert, delete, min, max, successor, and predecessor operations by following a single root-to-leaf path.

The critical limitation of a plain BST is that its height h depends on insertion order. Random insertions yield an expected height of $O(\log n)$, but sorted insertions produce a degenerate tree of height $O(n)$, reducing all operations to linear time. In the next chapter, we study **balanced search trees** — AVL trees and red-black trees — that maintain $O(\log n)$ height through automatic rotations, guaranteeing efficient operations regardless of the input order.

Chapter 12

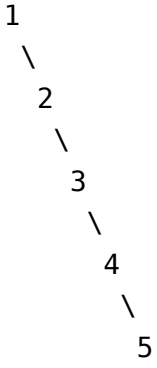
Balanced Search Trees

In Chapter 9 we built a binary search tree that provides $O(h)$ operations — fast when balanced, but potentially $O(n)$ when degenerate. Inserting n keys in sorted order produces a tree that is indistinguishable from a linked list. Balanced search trees solve this problem by restructuring the tree after every insert and delete, guaranteeing that the height remains $O(\log n)$ regardless of the input order. In this chapter we study two classic self-balancing trees: ***AVL trees****, which enforce a strict balance factor constraint, and* ***red-black trees****, which use node coloring to maintain a looser but equally effective bound.*

12.1 The problem with unbalanced BSTs

Recall from Chapter 9 that every BST operation follows a single root-to-leaf path, giving $O(h)$ time. For a balanced tree of n nodes, $h = O(\log n)$, so all operations are logarithmic. But the height depends entirely on the insertion order.

Consider inserting the values 1, 2, 3, 4, 5 in order:



The tree has height 4 (one less than n), and every operation degrades to $O(n)$. Even if the average-case height for random insertions is $O(\log n)$, we cannot

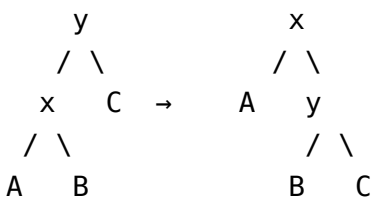
rely on the input being random — an adversary, a sorted file, or even a partially ordered stream can produce the worst case.

We need a tree that **automatically rebalances** after modifications. The key tool is the **rotation** — a local restructuring operation that changes the shape of a subtree without altering the in-order sequence of elements.

12.2 Rotations

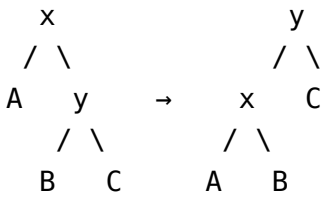
A rotation rearranges a parent-child pair while preserving the BST property. There are two kinds:

Right rotation around node y:



Node x (the left child of y) becomes the new root of the subtree. The subtree B, which was x's right child, becomes y's left child. All BST ordering is preserved: $A < x < B < y < C$.

Left rotation around node x:



This is the mirror image: y (the right child of x) becomes the new root of the subtree.

Both rotations run in $O(1)$ time — they only reassign a constant number of pointers. The critical insight is that rotations change the height of a subtree while keeping the sorted order intact. This is how balanced trees reduce height after an insertion or deletion disturbs the balance.

```
private rotateRight(y: AVLNode<T>): AVLNode<T> {
  const x = y.left!;
  const B = x.right;

  // Perform rotation
  x.right = y;
```

```
  y.left = B;

  // Update parents
  x.parent = y.parent;
  y.parent = x;
  if (B !== null) B.parent = y;

  // Update parent's child pointer
  if (x.parent === null) {
    this.root = x;
  } else if (x.parent.left === y) {
    x.parent.left = x;
  } else {
    x.parent.right = x;
  }

  // Update heights (y first since x is now y's parent)
  this.updateHeight(y);
  this.updateHeight(x);

  return x;
}
```

12.3 AVL trees

The **AVL tree** (named after its inventors Adelson-Velsky and Landis, 1962) is the oldest self-balancing BST. It maintains the following invariant:

> **AVL property:** For every node, the heights of its left and right subtrees differ by at most 1.

The **balance factor** of a node is height(left) − height(right). The AVL property requires that the balance factor of every node is -1, 0, or $+1$.

12.3.1 Height bound

An AVL tree with n nodes has height at most $1.44 \log_2(n+2) - 0.328$. This bound comes from analyzing the **minimum** number of nodes in an AVL tree of height h. Let $N(h)$ be this minimum. Then:

$$N(0) = 1, \quad N(1) = 2, \quad N(h) = 1 + N(h-1) + N(h-2)$$

The minimum AVL tree of height h has a root, a minimum AVL subtree of height $h-1$, and a minimum AVL subtree of height $h-2$ (the heights must differ by at most 1). This recurrence is closely related to the Fibonacci sequence, and its solution gives $N(h) \approx \phi^h/\sqrt{5}$ where $\phi = (1+\sqrt{5})/2 \approx 1.618$ is the golden ratio. Inverting, we get $h < 1.44 \log_2(n+2)$.

This means an AVL tree is at most about 44% taller than a perfectly balanced tree, guaranteeing $O(\log n)$ operations.

12.3.2 Node structure

Each AVL node stores its height explicitly, which makes computing balance factors a constant-time operation:

```
class AVLNode<T> {
  public left: AVLNode<T> | null = null;
  public right: AVLNode<T> | null = null;
  public parent: AVLNode<T> | null = null;
  public height = 0;

  constructor(public value: T) {}
}
```

Helper functions for height and balance factor:

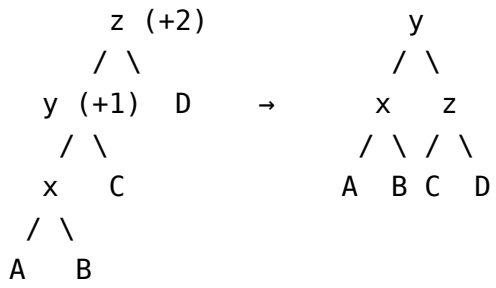
```
private h(node: AVLNode<T> | null): number {
  return node === null ? -1 : node.height;
}

private balanceFactor(node: AVLNode<T>): number {
  return this.h(node.left) - this.h(node.right);
}

private updateHeight(node: AVLNode<T>): void {
  node.height = 1 + Math.max(this.h(node.left), this.h(node.right));
}
```

12.3.3 Insertion

Insertion in an AVL tree starts with a standard BST insert, then walks back up the tree from the new node to the root, checking and fixing the balance factor at each ancestor.

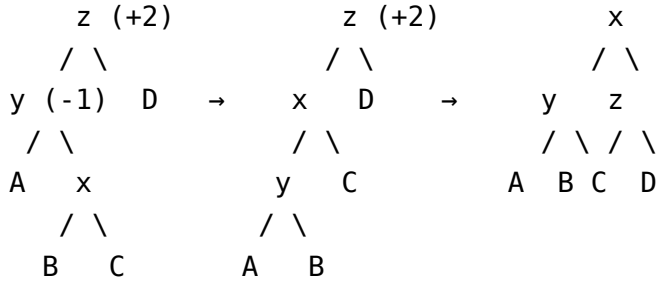
After inserting a new leaf, the balance factor of some ancestors may become +2 or −2. There are four cases, each resolved by one or two rotations:

Case 1: Left-Left (balance factor = +2, left child's balance factor ≥ 0). The left subtree is too tall, and the imbalance is on the left side of the left child. A single right rotation fixes it:



Case 2: Right-Right (balance factor = -2, right child's balance factor ≤ 0). The mirror of Case 1. A single left rotation fixes it.

Case 3: Left-Right (balance factor = +2, left child's balance factor = -1). The left subtree is too tall, but the imbalance is on the *right* side of the left child. A single rotation would not fix it — we need a **double rotation**: first left-rotate the left child, then right-rotate the node:



Case 4: Right-Left (balance factor = -2, right child's balance factor = +1). The mirror of Case 3: right-rotate the right child, then left-rotate the node.

The rebalance procedure:

```
private rebalance(node: AVLNode<T>): AVLNode<T> {
  this.updateHeight(node);
  const bf = this.balanceFactor(node);

  if (bf > 1) {
    // Left-heavy
    if (this.balanceFactor(node.left!) < 0) {
      // Left-Right case: rotate left child left first
      this.rotateLeft(node.left!);
    }
    // Left-Left case (or Left-Right reduced to Left-Left)
    return this.rotateRight(node);
```

```
  }

  if (bf < -1) {
    // Right-heavy
    if (this.balanceFactor(node.right!) > 0) {
      // Right-Left case: rotate right child right first
      this.rotateRight(node.right!);
    }
    // Right-Right case (or Right-Left reduced to Right-Right)
    return this.rotateLeft(node);
  }

  return node;
}
```

After insertion, we walk up from the new node's parent to the root, calling `rebalance` at each ancestor:

```
private rebalanceUp(node: AVLNode<T> | null): void {
  let current = node;
  while (current !== null) {
    const parent = current.parent;
    this.rebalance(current);
    current = parent;
  }
}
```

12.3.4 Tracing AVL insertions

Let us insert 1, 2, 3, 4, 5 — the sequence that degenerates a plain BST into a linked list.

Insert 1: Single node, height 0.

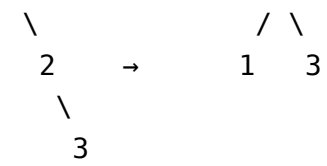
```
  1
```

Insert 2: Standard BST insert to the right. Balance factors are all valid.

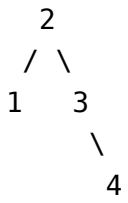
```
  1
   \
    2
```

Insert 3: Insert to the right of 2. Now node 1 has balance factor -2 (Right-Right case). Left-rotate around 1:

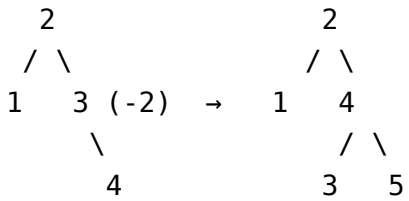
```
  1 (-2)           2
```



Insert 4: Insert to the right of 3. Balance factors are valid (root 2 has balance factor -1).



Insert 5: Insert to the right of 4. Now node 3 has balance factor -2 (Right-Right case). Left-rotate around 3:



After 5 insertions, the tree has height 2 — the minimum possible. A plain BST would have height 4.

12.3.5 Deletion

Deletion in an AVL tree uses the same three-case BST deletion algorithm from Chapter 9, followed by a rebalance walk from the lowest modified ancestor up to the root. The key difference from insertion is that deletion may require rotations at multiple ancestors (insertion requires at most one rotation point, but deletion can cascade):

```
private deleteNode(node: AVLNode<T>): void {
  let rebalanceStart: AVLNode<T> | null;

  if (node.left === null) {
    rebalanceStart = node.parent;
    this.transplant(node, node.right);
  } else if (node.right === null) {
    rebalanceStart = node.parent;
    this.transplant(node, node.left);
  } else {
    const successor = this.minNode(node.right)!;
```

```
    if (successor.parent !== node) {
      rebalanceStart = successor.parent;
      this.transplant(successor, successor.right);
      successor.right = node.right;
      successor.right.parent = successor;
    } else {
      rebalanceStart = successor;
    }
    this.transplant(node, successor);
    successor.left = node.left;
    successor.left.parent = successor;
  }

  this.rebalanceUp(rebalanceStart);
}
```

12.3.6 AVL complexity

Operation	Time	Space
Search	$O(\log n)$	$O(1)$
Insert	$O(\log n)$	$O(1)$
Delete	$O(\log n)$	$O(1)$
Min / Max	$O(\log n)$	$O(1)$
Successor / Predecessor	$O(\log n)$	$O(1)$
Inorder traversal	$O(n)$	$O(\log n)$
Space (tree)	—	$O(n)$

Each node stores one extra field (height), so the per-node overhead is small. Search does zero rotations. Insert does at most 2 rotations (one rotation point), but deletion may rotate at $O(\log n)$ ancestors in the worst case. All rotations are $O(1)$ each.

12.4 Red-black trees

A **red-black tree** is a BST where each node carries a one-bit color attribute — red or black — and five properties constrain how colors can be arranged. Red-black trees allow a slightly less strict balance than AVL trees: the height can be up to $2\log_2(n+1)$ versus AVL's $1.44\log_2(n+2)$. In exchange, they require fewer rotations during insertion and deletion, making them a popular choice

in practice (used in `std::map` in C++, `TreeMap` in Java, and the Linux kernel's scheduling data structure).

12.4.1 Red-black properties

A valid red-black tree satisfies all five of these properties:

1. **Every node is either red or black.**
2. **The root is black.**
3. **Every leaf (NIL) is black.** We use a sentinel NIL node rather than null pointers, which simplifies the algorithms.
4. **If a node is red, both its children are black.** Equivalently, no path from root to leaf has two consecutive red nodes.
5. **For each node, all simple paths from that node to descendant leaves contain the same number of black nodes.** This count is called the **black-height** of the node.

These properties together guarantee that no root-to-leaf path is more than twice as long as any other, which gives the height bound.

12.4.2 Height bound

The black-height of the root is the number of black nodes on any path from root to a leaf (not counting the root itself if we follow the convention, though CLRS counts the root). Because of Property 4 (no two reds in a row), a path of length h has at least $h/2$ black nodes. Because of Property 5 (all paths have the same black-height), the shortest path is all black nodes and the longest alternates red and black. Therefore:

$$h \leq 2 \log_2 (n + 1)$$

This guarantees $O(\log n)$ operations.

12.4.3 Node structure and sentinel

Red-black tree implementations use a sentinel NIL node to represent all external leaves. This avoids null-checks throughout the rotation and fixup code:

```
enum Color {
  Red = 'RED',
  Black = 'BLACK',
}

class RBNode<T> {
```

```
  public left: RBNode<T>;
  public right: RBNode<T>;
  public parent: RBNode<T>;
  public color: Color;

  constructor(public value: T, nil: RBNode<T>, color: Color = Color.Red) {
    this.left = nil;
    this.right = nil;
    this.parent = nil;
    this.color = color;
  }
}
```

The sentinel is a single black node that serves as every leaf and as the parent of the root. When we write `node.left === this.NIL`, we are checking whether the node has no left child.

12.4.4 Insertion

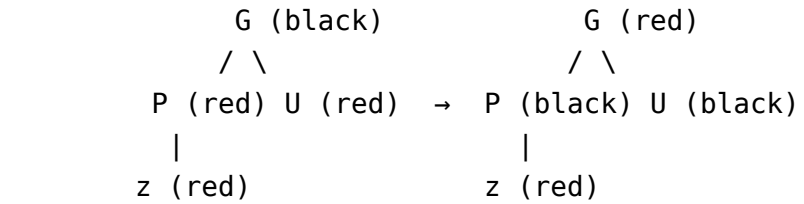
Insertion follows the CLRS `RB-INSERT` algorithm:

1. Insert the new node z as a red leaf using standard BST insertion.
2. Call `insertFixup(z)` to restore the red-black properties.

The new node is colored red because inserting a black node would violate Property 5 (black-height would increase on exactly one path). A red node might violate Property 4 (if its parent is also red) or Property 2 (if it becomes the root), but these are easier to fix.

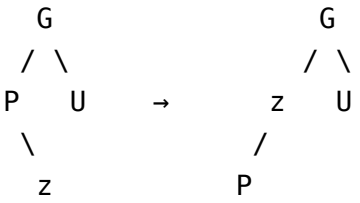
The fixup procedure handles three cases (and their symmetric mirrors when the parent is a right child):

Case 1: Uncle is red. Both the parent and uncle are red. Recolor the parent and uncle black and the grandparent red, then move up to the grandparent and repeat:

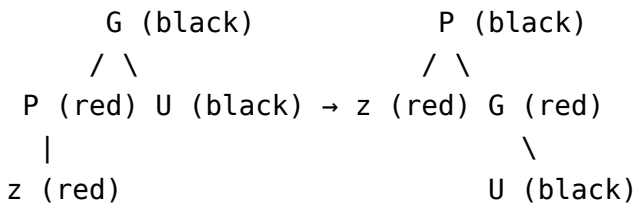


This fixes the local violation but may create a new red-red violation at G and its parent. The fix propagates upward.

Case 2: Uncle is black, z is an opposite-side child. If z is a right child but its parent is a left child (or vice versa), rotate z's parent to convert to Case 3:



Case 3: Uncle is black, z is a same-side child. Rotate the grandparent and recolor:



After Case 3, the subtree root is black with two red children — no further fixing is needed.

The fixup terminates when: - The parent is black (no violation), or - We reach the root (color it black to satisfy Property 2).

```
private insertFixup(z: RBNode<T>): void {
  let node = z;
  while (node.parent.color === Color.Red) {
    if (node.parent === node.parent.parent.left) {
      const uncle = node.parent.parent.right;
      if (uncle.color === Color.Red) {
        // Case 1: uncle is red — recolor
        node.parent.color = Color.Black;
        uncle.color = Color.Black;
        node.parent.parent.color = Color.Red;
        node = node.parent.parent;
      } else {
        if (node === node.parent.right) {
          // Case 2 → rotate to reduce to Case 3
          node = node.parent;
          this.rotateLeft(node);
        }
        // Case 3 — rotate grandparent
        node.parent.color = Color.Black;
        node.parent.parent.color = Color.Red;
        this.rotateRight(node.parent.parent);
```

```
      }
    } else {
      // Symmetric cases (parent is right child)
      // ...
    }
  }
  this.root.color = Color.Black;
}
```

12.4.5 Tracing red-black insertion

Let us insert the same sequence 1, 2, 3, 4, 5 into a red-black tree.

Insert 1: New node is red, but it is the root, so color it black.

```
1(B)
```

Insert 2: Insert as right child of 1. Node 2 is red, parent 1 is black — no violation.

```
1(B)
 \
  2(R)
```

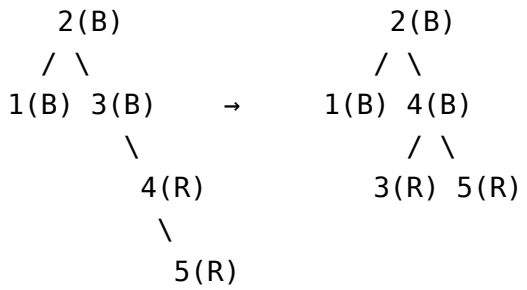
Insert 3: Insert as right child of 2. Now 2 (red) has a red child 3 — violation of Property 4. Uncle of 3 is NIL (black), and 3 is a right child of a right child — Case 3 (Right-Right). Left-rotate grandparent 1 and recolor:

```
1(B)              2(B)
 \               / \
  2(R)   →    1(R) 3(R)
   \
    3(R)
```

Insert 4: Insert as right child of 3. Now 3 (red) has red child 4 — violation. Uncle of 4 is 1 (red) — Case 1. Recolor: 1 and 3 become black, 2 becomes red. But 2 is the root, so immediately color it back to black:

```
  2(B)
 / \
1(B) 3(B)
       \
        4(R)
```

Insert 5: Insert as right child of 4. Now 4 (red) has red child 5 — violation. Uncle of 5 is NIL (black), and 5 is a right child of a right child — Case 3. Left-rotate grandparent 3 and recolor:



After 5 insertions, the tree has height 2 — well-balanced, with valid red-black properties.

12.4.6 Deletion

Red-black deletion is the most complex operation. The algorithm follows CLRS `RB-DELETE`:

1. Perform standard BST deletion to remove the node. Track the color of the node that was actually removed or moved (y's original color) and the node that replaced it (x).
2. If the removed/moved node was black, call `deleteFixup(x)` to restore the properties.

Removing a black node violates Property 5 (black-height consistency). The fixup pushes an "extra black" up the tree until it can be absorbed. There are four cases (and their mirrors):

Case 1: Sibling w is red. Recolor w black and the parent red, then rotate the parent. This converts to one of Cases 2–4 with a black sibling.

Case 2: Sibling w is black, both of w's children are black. Move the extra black up by coloring w red and moving to the parent.

Case 3: Sibling w is black, w's far child is black, near child is red. Rotate w and recolor to convert to Case 4.

Case 4: Sibling w is black, w's far child is red. Rotate the parent, transfer colors, and make the far child black. This absorbs the extra black and terminates the fixup.

The details are intricate, but the key guarantee is that at most 3 rotations are performed per deletion — fewer than AVL deletion's potential $O(\log n)$ rotations.

12.4.7 Verifying red-black properties

For testing and debugging, it is valuable to have a verification method that checks all five properties:

```
verify(): boolean {
  // Property 2: root is black
  if (this.root !== this.NIL && this.root.color !== Color.Black)
    return false;

  return this.verifyNode(this.root) >= 0;
}

private verifyNode(node: RBNode<T>): number {
  if (node === this.NIL) return 0;

  // Property 4: red node must have black children
  if (node.color === Color.Red) {
    if (node.left.color === Color.Red || node.right.color === Color.Red)
      return -1;
  }

  const leftBH = this.verifyNode(node.left);
  const rightBH = this.verifyNode(node.right);

  if (leftBH < 0 || rightBH < 0) return -1;

  // Property 5: equal black-height
  if (leftBH !== rightBH) return -1;

  return leftBH + (node.color === Color.Black ? 1 : 0);
}
```

This recursive procedure returns the black-height of each subtree, verifying Properties 4 and 5 simultaneously in $O(n)$ time.

12.4.8 Red-black complexity

Operation	Time	Rotations (worst case)
Search	$O(\log n)$	0
Insert	$O(\log n)$	2
Delete	$O(\log n)$	3
Min / Max	$O(\log n)$	0
Inorder traversal	$O(n)$	0

The per-node overhead is 1 bit (color), which is often stored in an otherwise

unused alignment bit of a pointer.

12.5 B-trees

B-trees are balanced search trees designed for **external storage** — disks, SSDs, and databases — where the cost of each node access is high. Instead of binary branching, a B-tree of order m allows each node to have up to m children and store up to $m-1$ keys. This high branching factor means fewer levels and fewer disk accesses.

A B-tree of order m satisfies: - Every node has at most m children. - Every non-root internal node has at least $\lceil m/2 \rceil$ children. - The root has at least 2 children (unless it is a leaf). - All leaves are at the same depth. - A node with k children stores $k-1$ keys.

For a B-tree of order 1000 storing one billion keys, the height is at most $\log_{500} 10^9 \approx 3.3$, meaning any key can be found in at most 4 disk reads. This is why B-trees and their variant B+ trees are the backbone of every major database system and filesystem.

We do not implement B-trees in this book because their primary benefit is I/O efficiency, which is difficult to demonstrate in an in-memory setting. The interested reader is referred to CLRS Chapter 18 or Wirth's *Algorithms + Data Structures = Programs* for detailed treatments.

12.6 Comparison of balanced tree variants

Property	AVL tree	Red-black tree	B-tree
Height bound	$\leq 1.44 \log_2(n+2)$	$\leq 2 \log_2(n+1)$	$\leq \log_{\lceil m/2 \rceil} n$
Strictness	Tight (BF $\in \{-1, 0, 1\}$)	Loose (path ratio ≤ 2)	All leaves same depth
Search time	$O(\log n)$	$O(\log n)$	$O(\log_m n)$
Insert rotations	≤ 2	≤ 2	0 (splits instead)
Delete rotations	$O(\log n)$	≤ 3	0 (merges/redistributes)
Per-node overhead	Height (integer)	Color (1 bit)	Variable-size key arrays
Best use case	Lookup-heavy workloads	Insert/delete-heavy	Disk-based storage

When to use which:

- **AVL trees** produce shorter, more tightly balanced trees. If your workload is search-heavy with few modifications, AVL trees will have slightly fewer comparisons per search.
- **Red-black trees** perform fewer rotations per modification. If your workload involves frequent insertions and deletions, red-black trees offer better amortized restructuring cost. Most language standard libraries choose red-black trees.
- **B-trees** are the right choice when data lives on disk and minimizing I/O operations is the priority.

12.7 Exercises

Exercise 10.1. Insert the values 14, 17, 11, 7, 53, 4, 13, 12, 8 into an initially empty AVL tree. After each insertion, draw the tree and show any rotations that occur. Identify which of the four rotation cases (LL, RR, LR, RL) applies in each case.

Exercise 10.2. Prove that an AVL tree with n nodes has height at most $1.44 \log_2(n+2)$. (Hint: define $N(h)$ as the minimum number of nodes in an AVL tree of height h, establish the recurrence $N(h) = 1 + N(h-1) + N(h-2)$, and relate it to the Fibonacci sequence.)

Exercise 10.3. A red-black tree with n internal nodes has height at most $2 \log_2(n+1)$. Prove this. (Hint: show by induction that a subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes, where $bh(x)$ is the black-height of x. Then use Property 4 to relate height to black-height.)

Exercise 10.4. Consider a red-black tree where you insert the keys 1 through 15 in order. Draw the tree after all insertions. What is the resulting height? How does this compare to the height bound $2 \log_2(n+1)$?

Exercise 10.5. AVL trees and red-black trees both guarantee $O(\log n)$ operations, but they make different trade-offs. Design an experiment to compare their performance: insert n random integers, then perform n searches, measuring the total number of comparisons for each tree type. Run the experiment for $n = 10^3, 10^4, 10^5$ and report the average number of comparisons per search. Which tree type performs fewer comparisons per search? Which performs fewer rotations per insertion? Discuss when each tree would be preferred.

12.8 Summary

Balanced search trees solve the fundamental problem of unbalanced BSTs by maintaining height invariants through automatic restructuring. **AVL trees** enforce a strict balance factor constraint (at most 1 difference between subtree heights), achieving a height bound of $1.44 \log_2(n+2)$ through four rotation cases applied during insertion and deletion. **Red-black trees** use a coloring scheme with five properties to maintain a height bound of $2 \log_2(n+1)$, trading slightly taller trees for fewer rotations during modifications — at most 2 per insertion and 3 per deletion.

Both trees guarantee $O(\log n)$ worst-case time for search, insert, delete, min, max, successor, and predecessor. AVL trees are preferred for lookup-heavy workloads due to shorter tree heights, while red-black trees are preferred for modification-heavy workloads due to fewer structural changes. **B-trees**, though not implemented here, extend the balancing concept to high-branching-factor trees optimized for disk access.

The rotations and rebalancing strategies studied in this chapter are fundamental techniques that appear throughout advanced data structures. In the next chapter, we turn to **heaps and priority queues** — another tree-based structure that maintains a different invariant (the heap property) for efficient extraction of minimum or maximum elements.

Chapter 13

Heaps and Priority Queues

*In the previous two chapters we studied binary search trees and their balanced variants — structures that maintain a total ordering of their elements for efficient search, insertion, and deletion. In this chapter we turn to a different kind of tree-based structure: the **binary heap**. A heap does not maintain a full sorted order; instead, it maintains a weaker **heap property** that ensures the minimum (or maximum) element is always at the root. This partial ordering is cheaper to maintain and gives us an efficient implementation of the **priority queue** abstract data type — a collection where we can always extract the highest-priority element in $O(\log n)$ time, insert new elements in $O(\log n)$ time, and peek at the top element in $O(1)$ time.*

13.1 The priority queue abstraction

Many algorithms need a data structure that answers the question: *"What is the most urgent item?"* Consider these examples:

- **Dijkstra's algorithm** (Chapter 13) repeatedly extracts the vertex with the smallest tentative distance.
- **Prim's algorithm** (Chapter 14) repeatedly extracts the lightest edge crossing a cut.
- **Huffman coding** (Chapter 17) repeatedly extracts the two lowest-frequency symbols.
- **Operating system schedulers** select the highest-priority process to run next.
- **Event-driven simulations** process events in chronological order.

In all these cases, the key operation is *extract the element with the highest priority*. A sorted array could answer this in $O(1)$ time, but insertion would cost $O(n)$. An unsorted array allows $O(1)$ insertion but $O(n)$ extraction. We want $O(\log n)$

for both — and that is exactly what a binary heap provides.

A **priority queue** supports the following operations:

Operation	Description
`enqueue(value, priority)`	Insert a value with a given priority
`dequeue()`	Remove and return the highest-priority value
`peek()`	Return the highest-priority value without removing it
`changePriority(value, newPriority)`	Update the priority of an existing value

The binary heap is the most common implementation of a priority queue, and the one we study in this chapter.

13.2 Binary heaps

A **binary heap** is a complete binary tree stored in an array. It satisfies two properties:

1. **Shape property:** The tree is a *complete binary tree* — every level is fully filled except possibly the last, which is filled from left to right. This guarantees the tree has height $\lfloor \log_2 n \rfloor$.
2. **Heap property:** For every node i (other than the root), the value at i's parent is less than or equal to the value at i (for a min-heap) or greater than or equal (for a max-heap).

The shape property means we can represent the tree as a flat array with no pointers. The heap property means the root always holds the minimum (or maximum) element.

13.2.1 Array representation

Because the tree is complete, we can map between tree positions and array indices using simple arithmetic. For a node at index i (using 0-based indexing):

$$\text{parent}(i) = \left\lfloor \frac{i - 1}{2} \right\rfloor$$

$$\text{left}(i) = 2i + 1$$

$$\text{right}(i) = 2i + 2$$

For example, the min-heap containing the values 1, 3, 5, 7, 4, 8, 6 is stored as:

```
Array:   [1, 3, 5, 7, 4, 8, 6]
Index:    0  1  2  3  4  5  6

Tree view:
            1          (index 0)
          /   \
        3       5      (indices 1, 2)
       / \     / \
      7   4   8   6    (indices 3, 4, 5, 6)
```

Node 1 (at index 0) is the root. Its children are at indices 1 and 2. Node 3 (index 1) has children at indices 3 and 4. No pointers are needed — the parent-child relationships are computed from the index.

In TypeScript:

```ts
function parentIndex(i: number): number {
  return Math.floor((i - 1) / 2);
}

function leftIndex(i: number): number {
  return 2 * i + 1;
}

function rightIndex(i: number): number {
  return 2 * i + 2;
}
```

13.2.2 The heap class

Our `BinaryHeap<T>` class stores elements in a flat array and accepts a comparator function to define the ordering. By default it uses ascending numeric comparison, producing a min-heap. Passing `(a, b) => b - a` produces a max-heap.

```ts
export class BinaryHeap<T> {
  private data: T[] = [];
  private readonly compare: Comparator<T>;

  constructor(comparator?: Comparator<T>) {
```

```
    this.compare = (comparator ?? numberComparator) as Comparator<T>;
  }

  get size(): number {
    return this.data.length;
  }

  get isEmpty(): boolean {
    return this.data.length === 0;
  }

  peek(): T | undefined {
    return this.data[0];
  }
  // ...
}
```

The peek operation simply returns the root element at index 0 in $O(1)$ time.

13.3 Heap operations

13.3.1 Sift-up (swim)

When we insert a new element at the end of the array, it may violate the heap property by being smaller than its parent (in a min-heap). **Sift-up** fixes this by repeatedly swapping the element with its parent until the heap property is restored or the element reaches the root.

```
Insert 2 into [1, 3, 5, 7, 4, 8, 6]:

Step 0: [1, 3, 5, 7, 4, 8, 6, 2]    ← 2 appended at index 7
                                        parent(7) = 3, at index 3
                                        2 < 7, so swap

Step 1: [1, 3, 5, 2, 4, 8, 6, 7]    ← 2 now at index 3
                                        parent(3) = 1, at index 1
                                        2 < 3, so swap

Step 2: [1, 2, 5, 3, 4, 8, 6, 7]    ← 2 now at index 1
                                        parent(1) = 0, at index 0
                                        2 > 1, stop
```

The implementation:

```
private siftUp(index: number): void {
  while (index > 0) {
    const parent = parentIndex(index);
    if (this.compare(this.data[index]!, this.data[parent]!) < 0) {
      this.swap(index, parent);
      index = parent;
    } else {
      break;
    }
  }
}
```

Since the tree has height $\lfloor \log_2 n \rfloor$, sift-up performs at most $O(\log n)$ swaps.

13.3.2 Sift-down (sink)

When we remove the root, we replace it with the last element in the array. This element is likely too large for the root position. **Sift-down** fixes this by repeatedly swapping the element with its smaller child (in a min-heap) until the heap property is restored or the element reaches a leaf.

```
Extract min from [1, 2, 5, 3, 4, 8, 6, 7]:

Step 0: Remove root (1), move last element (7) to root:
        [7, 2, 5, 3, 4, 8, 6]

Step 1: Compare 7 with children 2 (left) and 5 (right).
        Smallest child is 2 at index 1. 7 > 2, so swap.
        [2, 7, 5, 3, 4, 8, 6]

Step 2: Compare 7 with children 3 (left) and 4 (right).
        Smallest child is 3 at index 3. 7 > 3, so swap.
        [2, 3, 5, 7, 4, 8, 6]

Step 3: Index 3 has no children within bounds. Stop.
```

The implementation:

```
private siftDown(index: number): void {
  const n = this.data.length;
  while (true) {
    let best = index;
    const left = leftIndex(index);
```

```
    const right = rightIndex(index);

    if (left < n && this.compare(this.data[left]!, this.data[best]!) < 0) {
      best = left;
    }
    if (right < n && this.compare(this.data[right]!, this.data[best]!) < 0) {
      best = right;
    }
    if (best === index) break;
    this.swap(index, best);
    index = best;
  }
}
```

Like sift-up, sift-down performs at most $O(\log n)$ swaps.

13.3.3 Insert

Insertion appends the new element to the end of the array (maintaining the shape property) and then sifts up to restore the heap property:

```
insert(value: T): void {
  this.data.push(value);
  this.siftUp(this.data.length - 1);
}
```

Time: $O(\log n)$. The `push` is $O(1)$ amortized, and sift-up traverses at most $\lfloor \log_2 n \rfloor$ levels.

13.3.4 Extract

Extraction removes the root (the minimum element in a min-heap), replaces it with the last element, and sifts down:

```
extract(): T | undefined {
  if (this.data.length === 0) return undefined;
  if (this.data.length === 1) return this.data.pop()!;

  const root = this.data[0]!;
  this.data[0] = this.data.pop()!;
  this.siftDown(0);
  return root;
}
```

Time: $O(\log n)$. Moving the last element to the root is $O(1)$, and sift-down traverses at most $\lfloor \log_2 n \rfloor$ levels.

13.3.5 Decrease-key

The **decrease-key** operation replaces an element's value with a smaller one (higher priority in a min-heap) and sifts up to restore order. This operation is essential for algorithms like Dijkstra's, where we discover shorter paths and need to update a vertex's tentative distance.

```
decreaseKey(index: number, newValue: T): void {
  if (index < 0 || index >= this.data.length) {
    throw new RangeError(
      `Index ${index} out of bounds [0, ${this.data.length})`
    );
  }
  if (this.compare(newValue, this.data[index]!) > 0) {
    throw new Error('New value has lower priority than the current value');
  }
  this.data[index] = newValue;
  this.siftUp(index);
}
```

Time: $O(\log n)$, since sift-up traverses at most the height of the tree.

Note that decrease-key requires knowing the index of the element to update. In practice, algorithms that use decrease-key maintain a separate map from elements to their heap indices, updating it during every swap.

13.4 Building a heap in $O(n)$

The naive approach to building a heap from n elements is to insert them one at a time: n insertions at $O(\log n)$ each, for $O(n \log n)$ total. But we can do better.

Floyd's build-heap algorithm (1964) starts with the elements in arbitrary order and applies sift-down to every non-leaf node, working from the bottom of the tree to the root:

```
static from<T>(
  elements: T[],
  comparator?: Comparator<T>,
): BinaryHeap<T> {
  const heap = new BinaryHeap<T>(comparator);
  heap.data = elements.slice();
```

```
  heap.buildHeap();
  return heap;
}

private buildHeap(): void {
  for (let i = parentIndex(this.data.length - 1); i >= 0; i--) {
    this.siftDown(i);
  }
}
```

13.4.1 Why is this $O(n)$?

The key insight is that most nodes are near the bottom of the tree, where sift-down is cheap. In a complete binary tree with n nodes:

- $\lceil n/2 \rceil$ nodes are leaves (height 0) — sift-down does 0 swaps
- $\lceil n/4 \rceil$ nodes are at height 1 — sift-down does at most 1 swap
- $\lceil n/8 \rceil$ nodes are at height 2 — sift-down does at most 2 swaps
- In general, $\lceil n/2^{h+1} \rceil$ nodes are at height h, each doing at most h swaps

The total work is:

$$\sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot h \leq \frac{n}{2} \sum_{h=0}^{\infty} \frac{h}{2^h}$$

The series $\sum_{h=0}^{\infty} h/2^h = 2$ (this can be derived by differentiating the geometric series $\sum x^h = 1/(1-x)$ and setting $x = 1/2$). Therefore:

$$\text{Total work} \leq \frac{n}{2} \cdot 2 = n = O(n)$$

This is a remarkable result: building a heap is **linear**, not $O(n \log n)$. The intuition is that the expensive sift-downs (for nodes near the root) apply to very few nodes, while the cheap sift-downs (for nodes near the bottom) apply to many.

13.4.2 Why not sift-up?

If we tried to build a heap by sifting *up* from the first node to the last (simulating n insertions), the analysis would be:

$$\sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot (\lfloor \log_2 n \rfloor - h) = \Theta(n \log n)$$

The problem is that the many leaf nodes would each sift up $O(\log n)$ levels. Floyd's algorithm avoids this by processing nodes top-down (from the perspective of sift-down), so leaves do no work at all.

13.5 The priority queue interface

Our `PriorityQueue<T>` class wraps a `BinaryHeap` to provide a cleaner interface for the common case where each value has an associated numeric priority:

```
export interface PQEntry<T> {
  value: T;
  priority: number;
}

export class PriorityQueue<T> {
  private heap: BinaryHeap<PQEntry<T>>;

  constructor() {
    this.heap = new BinaryHeap<PQEntry<T>>(
      (a, b) => a.priority - b.priority
    );
  }

  enqueue(value: T, priority: number): void {
    this.heap.insert({ value, priority });
  }

  dequeue(): T | undefined {
    const entry = this.heap.extract();
    return entry?.value;
  }

  peek(): T | undefined {
    return this.heap.peek()?.value;
  }

  changePriority(value: T, newPriority: number): boolean {
    const arr = this.heap.toArray();
    const idx = arr.findIndex((e) => Object.is(e.value, value));
    if (idx === -1) return false;

    arr[idx] = { value, priority: newPriority };
```

```
    this.heap = BinaryHeap.from<PQEntry<T>>(
      arr,
      (a, b) => a.priority - b.priority,
    );
    return true;
  }
}
```

Lower numeric priority values are dequeued first. To create a max-priority queue, negate the priorities when enqueuing.

The `changePriority` method finds the entry by value identity (`Object.is`) and rebuilds the heap. This is $O(n)$ due to the linear scan. For Dijkstra's algorithm and similar performance-critical use cases, it is better to use `BinaryHeap` directly with an auxiliary index map for $O(\log n)$ decrease-key — we will see this in Chapter 13.

13.6 Min-heap vs. max-heap

Our implementation uses the comparator pattern to support both min-heaps and max-heaps without separate classes:

```
// Min-heap (default): smallest element at root
const minHeap = new BinaryHeap<number>();

// Max-heap: largest element at root
const maxHeap = new BinaryHeap<number>((a, b) => b - a);
```

The only difference is the comparator. When `compare(a, b) < 0`, element `a` has higher priority and should be closer to the root. For a min-heap, we want the smallest element at the root, so `compare(a, b) = a - b` makes smaller values "win." For a max-heap, `compare(a, b) = b - a` reverses the ordering.

This is the same pattern used by `Array.prototype.sort` in JavaScript and by the `Comparator<T>` type used throughout this book.

13.7 Applications

13.7.1 Heap sort

We saw heap sort in Chapter 5: build a max-heap from the input, then repeatedly extract the maximum and place it at the end of the array. The `BinaryHeap` class in this chapter is the data structure that heap sort uses internally. Heap sort

achieves $O(n \log n)$ worst-case time and $O(1)$ extra space (when done in-place on the array).

13.7.2 Running median

Given a stream of numbers, maintain the median at all times. Use two heaps:

- A **max-heap** for the lower half of the numbers.
- A **min-heap** for the upper half.

When a new number arrives, insert it into the appropriate heap and rebalance so the heaps differ in size by at most 1. The median is the root of the larger heap (or the average of both roots if they are equal in size). Each insertion takes $O(\log n)$.

13.7.3 Event-driven simulation

Model a system as a series of events, each with a timestamp. Store events in a min-heap ordered by time. At each step, extract the earliest event, process it (which may generate new events), and insert any new events. The heap ensures events are always processed in chronological order.

13.7.4 k smallest / largest elements

To find the k smallest elements in an unsorted array of n elements:

1. Build a min-heap in $O(n)$.
2. Extract k times for a total of $O(k \log n)$.

If $k \ll n$, this is much faster than sorting the entire array.

Alternatively, maintain a max-heap of size k. Scan the array; if an element is smaller than the heap's maximum, extract the max and insert the new element. This uses $O(k)$ space and $O(n \log k)$ time.

13.8 Complexity summary

Operation	Time	Space
`peek`	$O(1)$	$O(1)$
`insert`	$O(\log n)$	$O(1)$ amortized
`extract`	$O(\log n)$	$O(1)$
`decreaseKey`	$O(\log n)$	$O(1)$
`buildHeap` (from array)	$O(n)$	$O(n)$ for copy
`size` / `isEmpty`	$O(1)$	$O(1)$

The space for the entire heap is $O(n)$, since it is stored as a contiguous array.

Compared to balanced BSTs, heaps trade away sorted-order iteration and efficient search ($O(n)$ to find an arbitrary element) in exchange for simpler implementation, better constant factors, and cache-friendly array storage. If you only need insert and extract-min, a heap is the right choice.

13.9 Exercises

Exercise 11.1. Starting from an empty min-heap, insert the values 15, 10, 20, 8, 25, 12, 5, 18 one at a time. After each insertion, draw the heap as both a tree and an array. Verify the heap property holds at every step.

Exercise 11.2. Use Floyd's build-heap algorithm to construct a min-heap from the array $[16, 14, 10, 8, 7, 9, 3, 2, 4, 1]$. Show the array after processing each non-leaf node (from right to left). How many total swaps are performed? Compare this with the number of swaps that would result from inserting the elements one by one.

Exercise 11.3. Prove that the number of leaves in a complete binary tree stored in an array of length n is $\lceil n/2 \rceil$. (Hint: the last non-leaf node is at index $\lfloor (n-1)/2 \rfloor$.)

Exercise 11.4. Design a data structure that supports `insert`, `findMin`, and `findMax` in $O(\log n)$ time, and `extractMin` and `extractMax` in $O(\log n)$ time. (Hint: maintain both a min-heap and a max-heap simultaneously, with cross-references between corresponding entries.)

Exercise 11.5. Implement a *running median* data structure that supports `insert(x)` in $O(\log n)$ and `median()` in $O(1)$. Use two heaps: a max-heap for the lower half and a min-heap for the upper half. Write tests that insert a stream of 1000 random numbers and verify the median is correct after each insertion by comparing with a sorted-array baseline.

13.10 Summary

A **binary heap** is a complete binary tree stored in an array that maintains the heap property: every parent has higher priority than its children. This partial ordering — weaker than a sorted order — is cheaper to maintain and provides $O(\log n)$ insertion and extraction of the highest-priority element, with $O(1)$ peek.

The two fundamental operations are **sift-up** (restore order after insertion at the bottom) and **sift-down** (restore order after removal from the root). Floyd's **build-heap** algorithm constructs a heap from an arbitrary array in $O(n)$ time —

a result that follows from the observation that most nodes in a complete tree are near the bottom where sift-down is cheap.

The **priority queue** abstraction — enqueue with a priority, dequeue the highest-priority element — is directly implemented by a binary heap and is central to many graph algorithms (Dijkstra, Prim, Huffman). In the next chapters, we will put priority queues to work: Chapter 12 introduces graphs and graph traversal, and Chapter 13 uses priority queues as the backbone of Dijkstra's shortest-path algorithm.

Chapter 14

Graphs and Graph Traversal

In the previous chapters we studied data structures — arrays, linked lists, trees, heaps, hash tables — that organize data in essentially linear or hierarchical ways. Many real-world problems, however, involve relationships that are neither linear nor hierarchical: road networks, social connections, task dependencies, web links, circuit wiring. The natural abstraction for these problems is the ***graph****. In this chapter we define graphs formally, implement two standard representations, and develop two fundamental traversal algorithms —* ***breadth-first search (BFS)*** *and* ***depth-first search (DFS)*** *— that form the basis for nearly every graph algorithm in the chapters that follow. We also study* ***topological sorting*** *and* ***cycle detection****, two direct applications of graph traversal.*

14.1 What is a graph?

A **graph** $G = (V, E)$ consists of:

- A finite set V of **vertices** (also called nodes).
- A set E of **edges** (also called arcs), where each edge connects two vertices.

If every edge has a direction — going from one vertex to another — the graph is **directed** (a **digraph**). If edges have no direction, the graph is **undirected**. A **weighted** graph assigns a numeric weight to each edge; an **unweighted** graph treats all edges as having equal cost.

Key terminology:

- **Adjacent vertices:** Two vertices u and v are adjacent if there is an edge between them.
- **Incident edge:** An edge is incident to a vertex if the vertex is one of its endpoints.

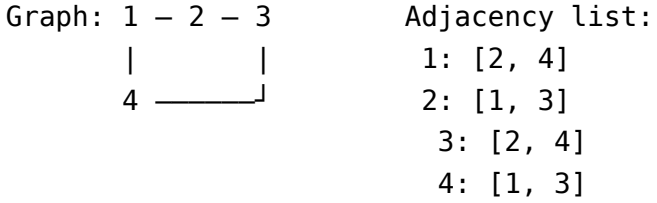
- **Degree:** The number of edges incident to a vertex. In a directed graph, we distinguish **in-degree** (edges entering) and **out-degree** (edges leaving).
- **Path:** A sequence of vertices $v_0, v_1, \ldots, v_k$ where each consecutive pair is connected by an edge. The **length** of the path is the number of edges, k.
- **Simple path:** A path with no repeated vertices.
- **Cycle:** A path where $v_0 = v_k$ and $k \geq 1$. A **simple cycle** has no repeated vertices except $v_0 = v_k$.
- **Connected graph:** An undirected graph where every pair of vertices is connected by some path.
- **Connected component:** A maximal connected subgraph.
- **Strongly connected:** In a directed graph, every vertex is reachable from every other vertex.
- **DAG:** A directed acyclic graph — a directed graph with no cycles.
- **Dense graph:** A graph where $|E| \approx |V|^2$ (many edges relative to vertices).
- **Sparse graph:** A graph where $|E| \ll |V|^2$ (few edges relative to vertices). Most real-world graphs are sparse.

14.2 Graph representations

There are two standard ways to represent a graph in memory: **adjacency lists** and **adjacency matrices**. The choice affects the time and space complexity of graph operations.

14.2.1 Adjacency list

An adjacency list stores, for each vertex, a collection of its neighbors. This is the preferred representation for sparse graphs, which includes most graphs encountered in practice.



Space: $O(V + E)$. For each vertex we store its neighbor list; the total number of entries across all lists is $2|E|$ for undirected graphs (each edge appears twice) or $|E|$ for directed graphs.

Our implementation uses a `Map`-based adjacency list. Each vertex maps to a `Map` of its neighbors and the corresponding edge weights:

```ts
export class Graph<T> {
  private adj: Map<T, Map<T, number>> = new Map();

  constructor(public readonly directed: boolean = false) {}

  addVertex(v: T): void {
    if (!this.adj.has(v)) {
      this.adj.set(v, new Map());
    }
  }

  addEdge(u: T, v: T, weight: number = 1): void {
    this.addVertex(u);
    this.addVertex(v);
    this.adj.get(u)!.set(v, weight);
    if (!this.directed) {
      this.adj.get(v)!.set(u, weight);
    }
  }

  hasEdge(u: T, v: T): boolean {
    return this.adj.get(u)?.has(v) ?? false;
  }

  getNeighbors(v: T): [T, number][] {
    const neighbors = this.adj.get(v);
    if (!neighbors) return [];
    return [...neighbors.entries()];
  }
  // ...
}
```

Using `Map` instead of a plain array gives us $O(1)$ edge lookup and supports arbitrary vertex types — not just integers. The `directed` flag controls whether `addEdge` creates edges in both directions.

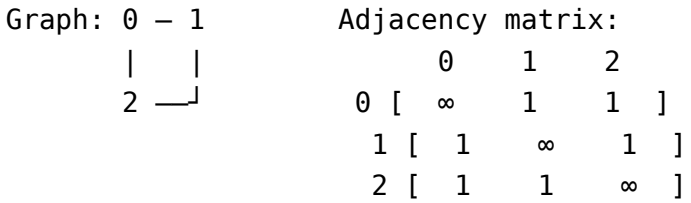
The complexity of common operations with an adjacency list:

Operation	Time
Add vertex	$O(1)$
Add edge	$O(1)$
Remove edge	$O(1)$

Operation	Time
Check edge	$O(1)$
Get neighbors	$O(1)$
Remove vertex	$O(V + E)$
Space	$O(V + E)$

14.2.2 Adjacency matrix

An adjacency matrix stores the graph as a $|V| \times |V|$ matrix M where $M[u][v]$ holds the weight of the edge from u to v (or ∞ if no edge exists). Vertices must be identified by integer indices $0, 1, \ldots, |V| - 1$.



Space: $\Theta(V^2)$, regardless of the number of edges. This makes the adjacency matrix inefficient for sparse graphs but convenient for dense graphs, where the space is similar to an adjacency list.

```
export class GraphMatrix {
  private matrix: number[][];

  constructor(
    size: number,
    public readonly directed: boolean = false,
  ) {
    this.matrix = Array.from({ length: size }, () =>
      Array.from({ length: size }, () => Infinity),
    );
  }

  addEdge(u: number, v: number, weight: number = 1): void {
    this.matrix[u]![v] = weight;
    if (!this.directed) {
      this.matrix[v]![u] = weight;
    }
  }

  hasEdge(u: number, v: number): boolean {
```

```
    return this.matrix[u]![v] !== Infinity;
  }

  getNeighbors(v: number): [number, number][] {
    const result: [number, number][] = [];
    for (let i = 0; i < this.matrix.length; i++) {
      if (this.matrix[v]![i] !== Infinity) {
        result.push([i, this.matrix[v]![i]!]);
      }
    }
    return result;
  }
}
```

The complexity of common operations with an adjacency matrix:

Operation	Time
Add edge	$O(1)$
Remove edge	$O(1)$
Check edge	$O(1)$
Get neighbors	$O(V)$
Space	$O(V^2)$

14.2.3 When to use which?

Criterion	Adjacency list	Adjacency matrix
Space	$O(V + E)$	$O(V^2)$
Edge lookup	$O(1)$ with Map	$O(1)$
Iterate neighbors	$O(\deg(v))$	$O(V)$
Best for	Sparse graphs	Dense graphs
Algorithms	BFS, DFS, Dijkstra, Kruskal	Floyd-Warshall, matrix algorithms

Most graph algorithms iterate over the neighbors of each vertex, making the adjacency list the better choice for sparse graphs. The adjacency matrix is preferred when $|E|$ is close to $|V|^2$ or when constant-time edge lookups with integer indices are important (e.g., Floyd-Warshall in Chapter 13).

Throughout this book, we default to the adjacency list representation.

14.3 Breadth-first search (BFS)

Breadth-first search explores a graph **level by level**: it visits all vertices at distance d from the source before any vertex at distance $d+1$. This guarantees that BFS finds the **shortest path** (fewest edges) from the source to every reachable vertex in an unweighted graph.

14.3.1 The algorithm

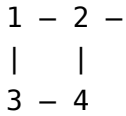
BFS maintains a **queue** of vertices to visit. Starting from a source vertex s:

1. Enqueue s and mark it as discovered with distance 0.
2. While the queue is not empty:
   a. Dequeue a vertex u.
   b. For each neighbor v of u that has not been discovered:
      - Mark v as discovered with distance $d(u) + 1$.
      - Record u as the parent of v.
      - Enqueue v.

The queue ensures that vertices are processed in the order they are discovered, which is the order of increasing distance from s.

14.3.2 Trace-through

Consider the following undirected graph, starting BFS from vertex 1:



Step	Queue (front → back)	Process	Discover	Distance
0	[1]	—	1	d(1)=0
1	[2, 3]	1	2, 3	d(2)=1, d(3)=1
2	[3, 4, 5]	2	4, 5	d(4)=2, d(5)=2
3	[4, 5]	3	—	(4 already discovered)
4	[5]	4	—	—
5	[]	5	—	—

Every vertex is visited exactly once. The distances are correct: 2 and 3 are 1 edge from 1; 4 and 5 are 2 edges from 1.

14.3.3 Implementation

```
export interface BFSResult<T> {
  parent: Map<T, T | undefined>;
  distance: Map<T, number>;
  order: T[];
}

export function bfs<T>(graph: Graph<T>, source: T): BFSResult<T> {
  const parent = new Map<T, T | undefined>();
  const distance = new Map<T, number>();
  const order: T[] = [];

  parent.set(source, undefined);
  distance.set(source, 0);
  order.push(source);

  const queue: T[] = [source];
  let head = 0;

  while (head < queue.length) {
    const u = queue[head++]!;
    const d = distance.get(u)!;

    for (const [v] of graph.getNeighbors(u)) {
      if (!distance.has(v)) {
        distance.set(v, d + 1);
        parent.set(v, u);
        order.push(v);
        queue.push(v);
      }
    }
  }

  return { parent, distance, order };
}
```

We use an array with a `head` pointer as a simple queue (avoiding the overhead of a linked-list queue for this application). The `distance` map also serves as our "visited" set — a vertex has been discovered if and only if it has an entry in `distance`.

14.3.4 Path reconstruction

The parent map produced by BFS encodes a shortest-path tree. To reconstruct the shortest path from source to target, follow parent pointers backward from the target:

```
export function reconstructPath<T>(
  parent: Map<T, T | undefined>,
  source: T,
  target: T,
): T[] | null {
  if (!parent.has(target)) return null;

  const path: T[] = [];
  let current: T | undefined = target;
  while (current !== undefined) {
    path.push(current);
    current = parent.get(current);
  }
  path.reverse();

  if (path[0] !== source) return null;
  return path;
}
```

14.3.5 Complexity

- **Time:** $O(V + E)$. Every vertex is enqueued and dequeued at most once ($O(V)$), and every edge is examined at most once (once for directed, twice for undirected) ($O(E)$).
- **Space:** $O(V)$ for the queue, parent map, and distance map.

BFS is optimal for finding shortest paths in unweighted graphs. For weighted graphs, we need Dijkstra's algorithm (Chapter 13).

14.4 Depth-first search (DFS)

Depth-first search explores a graph by going **as deep as possible** along each branch before backtracking. Where BFS explores level by level (breadth-first), DFS explores path by path (depth-first).

14.4.1 The algorithm

DFS assigns two timestamps to each vertex:

- **Discovery time** $d[v]$: when the vertex is first encountered.
- **Finish time** $f[v]$: when all of v's descendants have been fully explored.

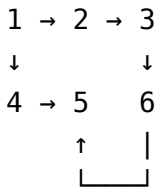
Starting from a source vertex, DFS:

1. Mark the vertex as discovered (record discovery time).
2. For each undiscovered neighbor, recursively visit it.
3. Mark the vertex as finished (record finish time).

If the graph is disconnected, DFS restarts from unvisited vertices, producing a **DFS forest**.

14.4.2 Trace-through

Consider the directed graph:



Starting DFS from vertex 1:

Action	Vertex	Time	Stack (conceptual)
Discover	1	0	[1]
Discover	2	1	[1, 2]
Discover	3	2	[1, 2, 3]
Discover	6	3	[1, 2, 3, 6]
Discover	5	4	[1, 2, 3, 6, 5]
Finish	5	5	[1, 2, 3, 6]
Finish	6	6	[1, 2, 3]
Finish	3	7	[1, 2]
Finish	2	8	[1]
Discover	4	9	[1, 4]
—	(5 already discovered)	—	—
Finish	4	10	[1]
Finish	1	11	[]

The discovery and finish times satisfy the **parenthesis theorem**: for any two vertices u and v, either the intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ are entirely

disjoint (neither is an ancestor of the other) or one is entirely contained within the other (one is an ancestor).

14.4.3 Edge classification

During DFS on a directed graph, every edge (u, v) falls into one of four categories based on the state of v when the edge is explored:

Edge type	Condition	Meaning
Tree edge	v is undiscovered	v is discovered via this edge (part of the DFS tree)
Back edge	v is discovered but not finished	v is an ancestor of u — indicates a **cycle**
Forward edge	v is finished and $d[u] < d[v]$	v is a descendant of u already fully explored via another path
Cross edge	v is finished and $d[u] > d[v]$	v is in a different, already-finished subtree

For **undirected** graphs, only tree edges and back edges are possible. Forward and cross edges cannot occur because every edge is traversed in both directions.

14.4.4 Implementation

```
export type EdgeType = 'tree' | 'back' | 'forward' | 'cross';

export interface ClassifiedEdge<T> {
  from: T;
  to: T;
  type: EdgeType;
}

export interface DFSResult<T> {
  discovery: Map<T, number>;
  finish: Map<T, number>;
  parent: Map<T, T | undefined>;
  order: T[];
  edges: ClassifiedEdge<T>[];
}
```

```ts
export function dfs<T>(
  graph: Graph<T>,
  startOrder?: T[],
): DFSResult<T> {
  const discovery = new Map<T, number>();
  const finish = new Map<T, number>();
  const parent = new Map<T, T | undefined>();
  const order: T[] = [];
  const edges: ClassifiedEdge<T>[] = [];
  let time = 0;

  const vertices = startOrder ?? graph.getVertices();

  function visit(u: T): void {
    discovery.set(u, time++);
    order.push(u);

    for (const [v] of graph.getNeighbors(u)) {
      if (!discovery.has(v)) {
        edges.push({ from: u, to: v, type: 'tree' });
        parent.set(v, u);
        visit(v);
      } else if (!finish.has(v)) {
        if (!graph.directed && parent.get(u) === v) continue;
        edges.push({ from: u, to: v, type: 'back' });
      } else if (graph.directed) {
        if (discovery.get(u)! < discovery.get(v)!) {
          edges.push({ from: u, to: v, type: 'forward' });
        } else {
          edges.push({ from: u, to: v, type: 'cross' });
        }
      }
    }

    finish.set(u, time++);
  }

  for (const v of vertices) {
    if (!discovery.has(v)) {
      parent.set(v, undefined);
      visit(v);
```

```
    }
  }

  return { discovery, finish, parent, order, edges };
}
```

The three-state classification (undiscovered, discovered but not finished, finished) maps directly to the colors used in textbooks: white, gray, black.

For undirected graphs, we skip the edge back to the parent — this is the same undirected edge we just traversed to reach the current vertex, not a true back edge.

14.4.5 Complexity

- **Time:** $O(V + E)$. Each vertex is visited once ($O(V)$), and each edge is examined once for directed graphs or twice for undirected ($O(E)$).
- **Space:** $O(V)$ for the recursion stack, parent map, discovery and finish times. In the worst case (a path graph), the recursion depth is $O(V)$.

14.5 Topological sort

A **topological sort** (or topological ordering) of a DAG is a linear ordering of all its vertices such that for every directed edge (u, v), vertex u appears before v in the ordering. In other words, if there is a path from u to v, then u comes first.

Topological sort is only defined for directed acyclic graphs (DAGs). A directed graph with a cycle has no valid topological ordering — there is no way to place all vertices in a line when some edges point backward.

14.5.1 Applications

- **Build systems** (Make, Bazel): compile source files in dependency order.
- **Task scheduling:** schedule jobs so that each job's prerequisites are completed first.
- **Course prerequisites:** determine a valid order to take courses.
- **Spreadsheet evaluation:** compute cells in an order that respects formula dependencies.
- **Package managers** (npm, apt): install dependencies before dependents.

14.5.2 Kahn's algorithm (BFS-based)

Kahn's algorithm (1962) uses the idea that a vertex with no incoming edges can safely go first in the ordering:

1. Compute the in-degree of every vertex.
2. Add all vertices with in-degree 0 to a queue.
3. While the queue is not empty:
   a. Dequeue a vertex u and add it to the result.
   b. For each neighbor v of u, decrement v's in-degree. If v's in-degree becomes 0, enqueue v.
4. If the result contains all vertices, return it. Otherwise, the graph has a cycle.

```
export function topologicalSortKahn<T>(graph: Graph<T>): T[] | null {
  const vertices = graph.getVertices();
  const inDeg = new Map<T, number>();

  for (const v of vertices) {
    inDeg.set(v, 0);
  }
  for (const v of vertices) {
    for (const [u] of graph.getNeighbors(v)) {
      inDeg.set(u, (inDeg.get(u) ?? 0) + 1);
    }
  }

  const queue: T[] = [];
  for (const [v, deg] of inDeg) {
    if (deg === 0) queue.push(v);
  }

  const order: T[] = [];
  let head = 0;

  while (head < queue.length) {
    const u = queue[head++]!;
    order.push(u);

    for (const [v] of graph.getNeighbors(u)) {
      const newDeg = inDeg.get(v)! - 1;
      inDeg.set(v, newDeg);
      if (newDeg === 0) queue.push(v);
```

```
    }
  }

  return order.length === vertices.length ? order : null;
}
```

Cycle detection: If the graph has a cycle, some vertices will never reach in-degree 0 and will never be enqueued. The algorithm detects this by checking whether all vertices were processed.

14.5.3 DFS-based topological sort

An alternative approach uses DFS. A topological ordering is the **reverse** of the DFS finish-time order: the vertex that finishes last should appear first.

```
export function topologicalSortDFS<T>(graph: Graph<T>): T[] | null {
  const vertices = graph.getVertices();

  const enum Color { White, Gray, Black }

  const color = new Map<T, Color>();
  for (const v of vertices) {
    color.set(v, Color.White);
  }

  const order: T[] = [];
  let hasCycle = false;

  function visit(u: T): void {
    if (hasCycle) return;
    color.set(u, Color.Gray);

    for (const [v] of graph.getNeighbors(u)) {
      const c = color.get(v)!;
      if (c === Color.Gray) {
        hasCycle = true;
        return;
      }
      if (c === Color.White) {
        visit(v);
        if (hasCycle) return;
      }
```

```
    }

    color.set(u, Color.Black);
    order.push(u);
  }

  for (const v of vertices) {
    if (color.get(v) === Color.White) {
      visit(v);
      if (hasCycle) return null;
    }
  }

  order.reverse();
  return order;
}
```

When we encounter a gray vertex (an ancestor on the current DFS path), we have found a back edge, which means the graph has a cycle.

14.5.4 Trace-through

Consider the "dressing order" DAG:

```
undershorts → pants → shoes
                 pants → belt → jacket
shirt → belt
shirt → tie → jacket
socks → shoes
watch (isolated)
```

Kahn's algorithm would start with vertices that have in-degree 0: `undershorts`, `shirt`, `socks`, `watch`. Processing them removes their outgoing edges, reducing in-degrees and producing new zero-in-degree vertices. A valid result:

```
undershorts, shirt, socks, watch, pants, tie, belt, shoes, jacket
```

DFS-based topological sort would produce a different but equally valid ordering based on which vertices are explored first.

14.5.5 Complexity

Both algorithms run in $O(V + E)$ time and $O(V)$ space.

14.6 Cycle detection

Cycle detection determines whether a graph contains a cycle. This is important for:

- Validating that a dependency graph is a DAG (and thus can be topologically sorted).
- Detecting deadlocks in resource allocation graphs.
- Identifying infinite loops in state machines.

14.6.1 Directed cycle detection

A directed graph has a cycle if and only if a DFS discovers a **back edge** — an edge to a vertex that is currently being explored (gray in the three-color scheme).

```
export function hasDirectedCycle<T>(graph: Graph<T>): boolean {
  const enum Color { White, Gray, Black }

  const color = new Map<T, Color>();
  for (const v of graph.getVertices()) {
    color.set(v, Color.White);
  }

  function visit(u: T): boolean {
    color.set(u, Color.Gray);

    for (const [v] of graph.getNeighbors(u)) {
      const c = color.get(v)!;
      if (c === Color.Gray) return true;
      if (c === Color.White && visit(v)) return true;
    }

    color.set(u, Color.Black);
    return false;
  }

  for (const v of graph.getVertices()) {
    if (color.get(v) === Color.White && visit(v)) {
      return true;
    }
  }
  return false;
}
```

The three colors are essential for directed cycle detection. A vertex colored gray is on the current DFS path. If we encounter a gray vertex, we have found a cycle. A black vertex (already finished) is not on the current path — an edge to a black vertex is a cross or forward edge, not evidence of a cycle.

14.6.2 Undirected cycle detection

For undirected graphs, cycle detection is simpler. During DFS, if we encounter a visited vertex that is not the parent of the current vertex, we have found a cycle:

```
export function hasUndirectedCycle<T>(graph: Graph<T>): boolean {
  const visited = new Set<T>();

  function visit(u: T, parent: T | undefined): boolean {
    visited.add(u);

    for (const [v] of graph.getNeighbors(u)) {
      if (!visited.has(v)) {
        if (visit(v, u)) return true;
      } else if (v !== parent) {
        return true;
      }
    }
    return false;
  }

  for (const v of graph.getVertices()) {
    if (!visited.has(v)) {
      if (visit(v, undefined)) return true;
    }
  }
  return false;
}
```

We only need two states (visited / not visited) instead of three, because in an undirected graph every non-tree edge to a visited non-parent vertex indicates a cycle. There are no forward or cross edges to worry about.

14.6.3 Complexity

Both directed and undirected cycle detection run in $O(V + E)$ time and $O(V)$ space, since they are based on DFS.

14.7 Connected components

A **connected component** of an undirected graph is a maximal set of vertices such that every pair is connected by a path. BFS or DFS can find all connected components:

```
components = 0
for each vertex v:
    if v is not visited:
        BFS(v) or DFS(v)   // marks all vertices in v's component
        components += 1
```

Each traversal from an unvisited vertex discovers one component. The total time is $O(V + E)$ since every vertex and edge is examined once across all traversals.

For directed graphs, the analogous concept is **strongly connected components** (SCCs): maximal sets of vertices where every vertex is reachable from every other vertex. Algorithms for finding SCCs (Kosaraju's, Tarjan's) build on DFS and will be discussed in later chapters.

14.8 BFS vs. DFS

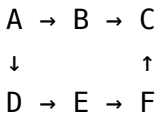
Property	BFS	DFS
Traversal order	Level by level	As deep as possible
Data structure	Queue	Stack (recursion or explicit)
Shortest paths (unweighted)	Yes	No
Edge classification (directed)	Tree, cross	Tree, back, forward, cross
Topological sort	Yes (Kahn's)	Yes (reverse finish order)
Cycle detection	Yes (via Kahn's / BFS topo sort)	Yes (back edge detection)
Memory	$O(V)$ — may store entire level	$O(V)$ — stack depth
Best for	Shortest paths, level-order	Cycle detection, topological sort, backtracking

Both algorithms visit every vertex and edge exactly once (or twice for undirected edges), giving $O(V+E)$ time. The choice between them depends on the problem:

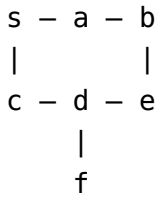
- Use **BFS** when you need shortest paths in an unweighted graph or want to explore vertices in order of distance.
- Use **DFS** when you need to detect cycles, classify edges, compute topological orderings, or explore all paths for backtracking algorithms.

14.9 Exercises

Exercise 12.1. Draw the adjacency list and adjacency matrix for the following directed graph. Which representation uses less space?



Exercise 12.2. Run BFS on the following undirected graph starting from vertex s. Record the discovery order, the distance from s to each vertex, and the BFS tree (parent pointers). Show the state of the queue at each step.



Exercise 12.3. Run DFS on the graph from Exercise 12.2 (treating it as directed with edges going both ways). Record discovery and finish times for each vertex. Verify that the parenthesis theorem holds: for every pair of vertices, the intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ are either disjoint or one contains the other.

Exercise 12.4. A **bipartite graph** is an undirected graph whose vertices can be partitioned into two sets A and B such that every edge connects a vertex in A to a vertex in B. Prove that a graph is bipartite if and only if it contains no odd-length cycle. Then describe an $O(V + E)$ algorithm to determine whether a graph is bipartite, using BFS. (Hint: try to 2-color the graph level by level.)

Exercise 12.5. A tournament is a directed graph where every pair of vertices is connected by exactly one directed edge. Prove that every tournament has a Hamiltonian path (a path that visits every vertex exactly once). Then describe an $O(V \log V)$ algorithm to find one. (Hint: use divide-and-conquer.)

14.10 Summary

A **graph** $G = (V, E)$ models pairwise relationships between objects. The two standard representations — **adjacency list** ($O(V + E)$ space, efficient neighbor

iteration) and **adjacency matrix** ($O(V^2)$ space, $O(1)$ edge lookup) — offer different trade-offs suited to sparse and dense graphs respectively.

Breadth-first search explores vertices level by level using a queue, computing shortest distances in unweighted graphs in $O(V + E)$ time. **Depth-first search** explores as deep as possible using recursion, assigning discovery and finish timestamps that enable edge classification into tree, back, forward, and cross edges.

Two important applications of DFS are **topological sorting** — producing a linear ordering of a DAG's vertices consistent with edge directions — and **cycle detection** — determining whether a graph contains a cycle by looking for back edges. Both run in $O(V + E)$ time.

These traversal algorithms form the foundation for nearly every graph algorithm in the chapters that follow. In Chapter 13, we will combine BFS ideas with the priority queue from Chapter 11 to solve the single-source shortest-path problem on weighted graphs (Dijkstra's algorithm). In Chapter 14, we will use graph traversal to find minimum spanning trees.

Chapter 15

Shortest Paths

*In Chapter 12 we introduced BFS, which finds shortest paths in **unweighted** graphs — that is, paths with the fewest edges. Most real-world graphs, however, carry weights on their edges: travel times on a road map, latencies in a network, costs in a supply chain. In this chapter we study algorithms that find shortest paths in **weighted** graphs, where the length of a path is the sum of its edge weights rather than the number of edges. We present four algorithms, each suited to different settings: Dijkstra's algorithm for graphs with non-negative weights, Bellman-Ford for graphs that may have negative weights, a linear-time algorithm for DAGs, and Floyd-Warshall for computing shortest paths between all pairs of vertices.*

15.1 The shortest-path problem

Given a weighted directed graph $G = (V, E)$ with edge-weight function $w : E \to \mathbb{R}$ and a source vertex s, the **single-source shortest-paths** problem asks: for every vertex $v \in V$, what is the minimum-weight path from s to v?

The **weight** of a path $p = \langle v_0, v_1, \ldots, v_k \rangle$ is

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

The **shortest-path weight** from s to v is

$$\delta(s, v) = \begin{cases} \min\{w(p) : s \overset{p}{\rightsquigarrow} v\} & \text{if a path exists} \\ \infty & \text{otherwise} \end{cases}$$

A **shortest path** from s to v is any path p with $w(p) = \delta(s, v)$.

15.1.1 Negative weights and negative cycles

When all edge weights are non-negative, shortest paths are well-defined. When negative-weight edges exist, a complication arises: a **negative-weight cycle** — a cycle whose total weight is negative — can be traversed repeatedly to make path weights arbitrarily negative. If such a cycle is reachable from the source, shortest-path distances are undefined for any vertex reachable from the cycle.

We will carefully note which algorithms handle negative weights and which detect negative cycles.

15.1.2 Relaxation

All single-source shortest-path algorithms share a common operation: **relaxation**. For each vertex v we maintain an estimate $d[v]$ of the shortest-path weight from the source (initially ∞ for all vertices except the source, which is 0). Relaxing an edge (u, v) checks whether the path through u offers a shorter route to v:

```
Relax(u, v, w):
    if d[u] + w(u, v) < d[v]:
        d[v] = d[u] + w(u, v)
        parent[v] = u
```

The algorithms in this chapter differ in the **order** and **number of times** they relax edges.

15.1.3 Shared result type

Our implementations share a common result type representing shortest-path distances and predecessor pointers:

```
export interface ShortestPathResult<T> {
  dist: Map<T, number>;
  parent: Map<T, T | undefined>;
}
```

The `parent` map allows us to reconstruct the actual shortest path from source to any target:

```
export function reconstructPath<T>(
  parent: Map<T, T | undefined>,
  source: T,
  target: T,
): T[] | null {
  if (!parent.has(target)) return null;
```

```
  const path: T[] = [];
  let current: T | undefined = target;
  while (current !== undefined) {
    path.push(current);
    current = parent.get(current);
  }
  path.reverse();

  if (path[0] !== source) return null;
  return path;
}
```

This is the same backtracking technique we used for BFS path reconstruction in Chapter 12: we follow parent pointers from the target back to the source, then reverse the result.

15.2 Dijkstra's algorithm

Dijkstra's algorithm (1959) solves the single-source shortest-paths problem for graphs with **non-negative** edge weights. It is the workhorse algorithm for shortest paths in practice — used in GPS navigation, network routing (OSPF), and countless other applications.

15.2.1 Intuition

The key insight is greedy: among all vertices whose shortest-path distance is not yet finalized, the one with the smallest current estimate $d[v]$ already has the correct shortest-path distance. Why? Because all edge weights are non-negative, so any other path to v must pass through a vertex with a distance estimate at least as large, making the total at least as long.

This is exactly analogous to BFS, except that instead of a FIFO queue (which processes vertices in order of number of edges), we use a **priority queue** ordered by distance estimates.

15.2.2 Algorithm

1. Initialize $d[s] = 0$ and $d[v] = \infty$ for all other vertices.
2. Insert the source into a min-priority queue with priority 0.
3. While the priority queue is not empty:
    a. Extract the vertex u with the smallest priority.
    b. If u has already been visited, skip it.

c. Mark u as visited.
d. For each neighbor v of u, relax the edge (u, v). If the distance improves, insert v into the priority queue with the new distance.

15.2.3 Implementation

```
import { Graph } from '../12-graphs-and-traversal/graph.js';
import { PriorityQueue } from '../11-heaps-and-priority-queues/priority-queue.js';

export function dijkstra<T>(
  graph: Graph<T>,
  source: T,
): ShortestPathResult<T> {
  const dist = new Map<T, number>();
  const parent = new Map<T, T | undefined>();
  const visited = new Set<T>();

  for (const v of graph.getVertices()) {
    dist.set(v, Infinity);
  }
  dist.set(source, 0);
  parent.set(source, undefined);

  const pq = new PriorityQueue<T>();
  pq.enqueue(source, 0);

  while (!pq.isEmpty) {
    const u = pq.dequeue()!;

    if (visited.has(u)) continue;
    visited.add(u);

    for (const [v, weight] of graph.getNeighbors(u)) {
      const newDist = dist.get(u)! + weight;
      if (newDist < dist.get(v)!) {
        dist.set(v, newDist);
        parent.set(v, u);
        pq.enqueue(v, newDist);
      }
    }
  }
```

```
  return { dist, parent };
}
```

Implementation note: Rather than implementing an explicit decrease-key operation, we insert a new entry into the priority queue whenever we find a shorter path. The `visited` set ensures we process each vertex only once — duplicate entries for already-visited vertices are simply skipped. This is a common practical optimization often called the "lazy Dijkstra" approach, and it does not affect correctness.

15.2.4 Trace-through

Consider the following directed graph with source s:

Edge	Weight
s → t	10
s → y	5
t → y	2
t → x	1
y → t	3
y → x	9
x → z	4
z → x	6
z → s	7

Step-by-step execution from source s:

Step	Extract	$d[s]$	$d[t]$	$d[y]$	$d[x]$	$d[z]$	Action
Init	—	0	∞	∞	∞	∞	Enqueue s with priority 0
1	s	**0**	10	5	∞	∞	Relax $s \to t$ and $s \to y$

Step	Extract	$d[s]$	$d[t]$	$d[y]$	$d[x]$	$d[z]$	Action
2	y	0	8	**5**	14	∞	Relax $y \to t$ (5+3=8 < 10) and $y \to x$ (5+9=14)
3	t	0	**8**	5	9	∞	Relax $t \to x$ (8+1=9 < 14)
4	x	0	8	5	**9**	13	Relax $x \to z$ (9+4=13)
5	z	0	8	5	9	**13**	Done (z → s: 13+7=20 > 0, no update)

Final shortest-path distances: $d[s] = 0$, $d[y] = 5$, $d[t] = 8$, $d[x] = 9$, $d[z] = 13$.

15.2.5 Complexity

- **Time:** $O((V + E) \log V)$ with a binary heap. Each vertex is extracted at most once ($O(V \log V)$ total). Each edge triggers at most one priority queue insertion ($O(E \log V)$ total).
- **Space:** $O(V+E)$ for the graph plus $O(V)$ for the priority queue and distance maps.

With a Fibonacci heap, the time complexity improves to $O(V \log V + E)$, but Fibonacci heaps are complex to implement and have high constant factors. For most practical purposes, the binary-heap version is preferred.

15.2.6 Correctness argument

Dijkstra's algorithm is correct when all edge weights are non-negative. The proof relies on the following loop invariant: when a vertex u is extracted from the priority queue, $d[u] = \delta(s, u)$.

Sketch: Suppose for contradiction that u is the first vertex extracted with $d[u] >$

$\delta(s, u)$. Consider the true shortest path from s to u. Let (x, y) be the first edge on this path where x has already been finalized but y has not. When x was finalized, edge (x, y) was relaxed, so $d[y] \leq \delta(s, y) \leq \delta(s, u) < d[u]$. But then y would have been extracted before u, contradicting our choice of u. (This inequality relies on non-negative weights: each edge on the subpath from y to u contributes a non-negative amount.)

15.2.7 When Dijkstra fails

With negative edge weights, the greedy assumption breaks down. A vertex u may be extracted with a distance estimate that is later revealed to be too high, because a path through a later-discovered vertex with a negative edge reaches u more cheaply. For this reason, Dijkstra's algorithm produces incorrect results on graphs with negative edges.

15.3 Bellman-Ford algorithm

The Bellman-Ford algorithm (1958) solves the single-source shortest-paths problem for graphs with **arbitrary** edge weights — including negative weights. It also detects negative-weight cycles reachable from the source.

15.3.1 Algorithm

1. Initialize $d[s] = 0$ and $d[v] = \infty$ for all other vertices.
2. Repeat $|V| - 1$ times: relax every edge in the graph.
3. Check for negative cycles: scan all edges once more. If any edge can still be relaxed, the graph has a negative-weight cycle reachable from the source.

Why $|V| - 1$ iterations? A shortest path in a graph with no negative cycles has at most $|V| - 1$ edges (it is a simple path). In iteration i, the algorithm correctly computes shortest paths that use at most i edges. After $|V| - 1$ iterations, all shortest paths (with up to $|V| - 1$ edges) are correctly computed.

15.3.2 Implementation

```
import { Graph } from '../12-graphs-and-traversal/graph.js';

export interface BellmanFordResult<T> extends ShortestPathResult<T> {
  hasNegativeCycle: boolean;
}

export function bellmanFord<T>(
```

```
  graph: Graph<T>,
  source: T,
): BellmanFordResult<T> {
  const vertices = graph.getVertices();
  const dist = new Map<T, number>();
  const parent = new Map<T, T | undefined>();

  for (const v of vertices) {
    dist.set(v, Infinity);
  }
  dist.set(source, 0);
  parent.set(source, undefined);

  // Relax all edges V-1 times.
  const V = vertices.length;
  for (let i = 0; i < V - 1; i++) {
    let changed = false;
    for (const u of vertices) {
      const du = dist.get(u)!;
      if (du === Infinity) continue;
      for (const [v, weight] of graph.getNeighbors(u)) {
        const newDist = du + weight;
        if (newDist < dist.get(v)!) {
          dist.set(v, newDist);
          parent.set(v, u);
          changed = true;
        }
      }
    }
    if (!changed) break; // Early termination
  }

  // Check for negative-weight cycles.
  let hasNegativeCycle = false;
  for (const u of vertices) {
    const du = dist.get(u)!;
    if (du === Infinity) continue;
    for (const [v, weight] of graph.getNeighbors(u)) {
      if (du + weight < dist.get(v)!) {
        hasNegativeCycle = true;
        break;
```

```
      }
    }
    if (hasNegativeCycle) break;
  }

  return { dist, parent, hasNegativeCycle };
}
```

Early termination: If no distance estimate changes in an entire pass, all distances are final and we can stop early. This optimization does not improve the worst-case complexity but can significantly speed up the algorithm on graphs where shortest paths have few edges.

15.3.3 Trace-through

Consider the CLRS example graph (directed, with negative edges):

Edge	Weight
s → t	6
s → y	7
t → x	5
t → y	8
t → z	−4
y → x	−3
y → z	9
x → t	−2
z → s	2
z → x	7

Running Bellman-Ford from source s, after all passes converge:

Vertex	$d[v]$	Shortest path from s
s	0	—
t	2	s → y → x → t
x	4	s → y → x
y	7	s → y
z	−2	s → y → x → t → z

The shortest path to z has weight -2, using two negative edges ($y \to x$ and $t \to z$).

15.3.4 Complexity

- **Time:** $O(V \cdot E)$. The outer loop runs at most $V - 1$ times, and each iteration examines all E edges.
- **Space:** $O(V)$ for distances and parent pointers.

15.3.5 Negative cycle detection

The check in the final pass is both necessary and sufficient. If a negative cycle is reachable from the source, then after $V - 1$ relaxation passes, at least one edge on the cycle can still be relaxed — because traversing the cycle one more time would further decrease the distance. Conversely, if no edge can be relaxed, then $d[v] = \delta(s, v)$ for all reachable vertices and no negative cycle exists.

15.4 DAG shortest paths

When the input graph is a **directed acyclic graph** (DAG), we can find shortest paths in $O(V + E)$ time — even with negative edge weights. The idea is simple: process vertices in **topological order**.

15.4.1 Algorithm

1. Compute a topological ordering of the DAG (using Kahn's algorithm or DFS, as described in Chapter 12).
2. Initialize $d[s] = 0$ and $d[v] = \infty$ for all other vertices.
3. For each vertex u in topological order: relax all outgoing edges of u.

Since vertices are processed in topological order, when we relax the edges of u, all vertices that could provide a shorter path to u have already been processed. Every edge is relaxed exactly once.

15.4.2 Implementation

```
import { Graph } from '../12-graphs-and-traversal/graph.js';
import { topologicalSortKahn }
  from '../12-graphs-and-traversal/topological-sort.js';

export function dagShortestPaths<T>(
  graph: Graph<T>,
  source: T,
): ShortestPathResult<T> {
  const order = topologicalSortKahn(graph);
```

```
  if (order === null) {
    throw new Error(
      'Graph contains a cycle; DAG shortest paths requires a DAG',
    );
  }

  const dist = new Map<T, number>();
  const parent = new Map<T, T | undefined>();

  for (const v of graph.getVertices()) {
    dist.set(v, Infinity);
  }
  dist.set(source, 0);
  parent.set(source, undefined);

  for (const u of order) {
    const du = dist.get(u)!;
    if (du === Infinity) continue;

    for (const [v, weight] of graph.getNeighbors(u)) {
      const newDist = du + weight;
      if (newDist < dist.get(v)!) {
        dist.set(v, newDist);
        parent.set(v, u);
      }
    }
  }

  return { dist, parent };
}
```

15.4.3 Why this works

A topological order guarantees that for every edge (u, v), vertex u is processed before v. When we process u and relax its outgoing edges, $d[u]$ is already optimal — all predecessors of u in the graph have already been processed. Therefore, each edge is relaxed exactly once, and after processing all vertices, $d[v] = \delta(s, v)$ for every reachable vertex.

This argument does not require non-negative weights. Even if edge (u, v) has a negative weight, when we process u we have the correct $d[u]$, so the relaxation computes the correct contribution of this edge.

15.4.4 Applications

DAG shortest paths are useful for:

- **Critical path analysis** (PERT/CPM): find the longest path in a project task graph to determine the minimum project duration. (Use negated weights to convert longest-path to shortest-path.)
- **Dynamic programming on DAGs:** many DP problems can be modeled as shortest or longest paths in a DAG.
- **Pipeline scheduling:** determine minimum latency through a pipeline of processing stages.

15.4.5 Complexity

- **Time:** $O(V + E)$ — topological sort takes $O(V + E)$, and relaxing all edges takes $O(V + E)$.
- **Space:** $O(V)$.

This is asymptotically optimal: we must examine every edge at least once, and there are $O(V + E)$ edges and vertices.

15.5 Floyd-Warshall algorithm

The previous three algorithms solve the **single-source** shortest-paths problem: shortest paths from one specific source vertex. The **Floyd-Warshall algorithm** (1962) solves a different problem: **all-pairs shortest paths** — the shortest distance between every pair of vertices simultaneously.

Of course, we could run Dijkstra's algorithm $|V|$ times (once from each vertex) to get all-pairs shortest paths in $O(V(V + E) \log V)$ time. But Floyd-Warshall uses a different approach based on dynamic programming that runs in $O(V^3)$ time, which is simpler to implement and competitive for dense graphs where $E \approx V^2$.

15.5.1 The dynamic programming formulation

Define $d^{(k)}[i][j]$ as the shortest-path weight from vertex i to vertex j using only vertices $\{1, 2, \ldots, k\}$ as intermediate vertices. The recurrence is:

$$d^{(k)}[i][j] = \min(d^{(k-1)}[i][j],\ \ d^{(k-1)}[i][k] + d^{(k-1)}[k][j])$$

In words: the shortest path from i to j through vertices $\{1, \ldots, k\}$ either avoids vertex k entirely (first term) or goes through k (second term).

Base case: $d^{(0)}[i][j] = w(i, j)$ if edge (i, j) exists, ∞ if not, and 0 if $i = j$.

Final answer: $d^{(V)}[i][j] = \delta(i, j)$ for all pairs (i, j).

15.5.2 Space optimization

The three nested loops can update the matrix in place. When computing $d^{(k)}$, the values $d^{(k-1)}[i][k]$ and $d^{(k-1)}[k][j]$ are not modified by including vertex k as an intermediate (setting $i = k$ or $j = k$ doesn't change the result). Therefore, we need only a single 2D matrix rather than V copies.

15.5.3 Implementation

```
import { Graph } from '../12-graphs-and-traversal/graph.js';

export interface FloydWarshallResult<T> {
  dist: number[][];
  next: number[][];
  vertices: T[];
}

export function floydWarshall<T>(
  graph: Graph<T>,
): FloydWarshallResult<T> {
  const vertices = graph.getVertices();
  const V = vertices.length;
  const indexOf = new Map<T, number>();
  for (let i = 0; i < V; i++) {
    indexOf.set(vertices[i]!, i);
  }

  // Initialize distance and next-hop matrices.
  const dist: number[][] = Array.from({ length: V }, () =>
    Array.from({ length: V }, () => Infinity),
  );
  const next: number[][] = Array.from({ length: V }, () =>
    Array.from({ length: V }, () => -1),
  );

  for (let i = 0; i < V; i++) {
    dist[i]![i] = 0;
    next[i]![i] = i;
  }
```

```
  // Seed with direct edges.
  for (const v of vertices) {
    const u = indexOf.get(v)!;
    for (const [neighbor, weight] of graph.getNeighbors(v)) {
      const w = indexOf.get(neighbor)!;
      if (weight < dist[u]![w]!) {
        dist[u]![w] = weight;
        next[u]![w] = w;
      }
    }
  }

  // DP: consider each vertex k as intermediate.
  for (let k = 0; k < V; k++) {
    for (let i = 0; i < V; i++) {
      for (let j = 0; j < V; j++) {
        const through_k = dist[i]![k]! + dist[k]![j]!;
        if (through_k < dist[i]![j]!) {
          dist[i]![j] = through_k;
          next[i]![j] = next[i]![k]!;
        }
      }
    }
  }

  return { dist, next, vertices };
}
```

The `next` matrix tracks the first hop on the shortest path from i to j, enabling path reconstruction:

```
export function reconstructPathFW(
  next: number[][],
  i: number,
  j: number,
): number[] | null {
  if (next[i]![j] === -1) return null;

  const path = [i];
  let current = i;
  while (current !== j) {
    current = next[current]![j]!;
```

```
    if (current === -1) return null;
    path.push(current);
  }
  return path;
}
```

15.5.4 Negative cycle detection

After running Floyd-Warshall, a negative-weight cycle exists if and only if some diagonal entry is negative: $d[i][i] < 0$ for some vertex i. This means there is a path from i back to i with negative total weight.

```
export function hasNegativeCycle(
  result: FloydWarshallResult<unknown>,
): boolean {
  for (let i = 0; i < result.vertices.length; i++) {
    if (result.dist[i]![i]! < 0) return true;
  }
  return false;
}
```

15.5.5 Complexity

- **Time:** $O(V^3)$ — three nested loops, each iterating over V vertices.
- **Space:** $O(V^2)$ for the distance and next-hop matrices.

For dense graphs ($E = \Theta(V^2)$), this matches running Dijkstra V times: $O(V \cdot (V + V^2) \log V) = O(V^3 \log V)$, so Floyd-Warshall is actually faster. For sparse graphs, running Dijkstra from each vertex is preferable.

15.6 Choosing the right algorithm

Algorithm	Weights	Negative cycles	Source	Time	Space
Dijkstra	≥ 0	N/A	Single	$O((V + E) \log V)$	$O(V)$
Bellman-Ford	Any	Detects	Single	$O(V \cdot E)$	$O(V)$
DAG shortest paths	Any	N/A (no cycles)	Single	$O(V + E)$	$O(V)$

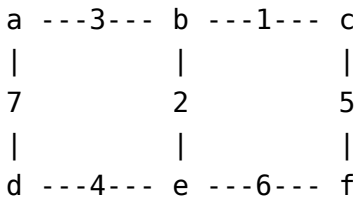
Algorithm	Weights	Negative cycles	Source	Time	Space
Floyd-Warshall	Any	Detects	All pairs	$O(V^3)$	$O(V^2)$

Decision guide:

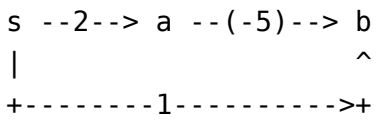
- **Non-negative weights, single source:** Use **Dijkstra**. It is the fastest single-source algorithm for this common case.
- **Negative weights possible, single source:** Use **Bellman-Ford**. It handles negative weights and detects negative cycles.
- **DAG with any weights, single source:** Use **DAG shortest paths**. It is the fastest possible, running in linear time.
- **All-pairs shortest paths, dense graph:** Use **Floyd-Warshall**. Simple to implement and efficient for dense graphs.
- **All-pairs shortest paths, sparse graph:** Run **Dijkstra** from each vertex ($O(V(V+E)\log V)$), or use Johnson's algorithm (which combines Bellman-Ford reweighting with Dijkstra) for $O(V^2 \log V + VE)$.

15.7 Exercises

Exercise 13.1. Run Dijkstra's algorithm on the following undirected graph from source a. Show the state of the priority queue and the distance estimates after each extraction.



Exercise 13.2. Explain why Dijkstra's algorithm produces incorrect results on the following graph with source s:



Show the incorrect distances Dijkstra computes and the correct distances.

Exercise 13.3. Run Bellman-Ford on the graph from Exercise 13.2 and verify that it produces the correct shortest-path distances. How many relaxation passes are needed before the algorithm converges?

Exercise 13.4. Consider a directed graph representing course prerequisites at a university. Each edge (u, v) has a weight representing the "effort" of completing course v after u. Give an $O(V + E)$ algorithm to find the minimum-effort path from a starting course to a target course. What property of this graph makes this possible?

Exercise 13.5. The **transitive closure** of a directed graph $G = (V, E)$ is a graph $G^* = (V, E^*)$ where $(u, v) \in E^*$ if and only if there is a path from u to v in G. Show how to compute the transitive closure using Floyd-Warshall. What is the time complexity? Can you modify the algorithm to use Boolean operations (AND, OR) instead of arithmetic for a constant-factor speedup?

15.8 Summary

The **shortest-path problem** asks for minimum-weight paths in weighted graphs. Four algorithms address different variants of this problem.

Dijkstra's algorithm uses a greedy strategy with a priority queue, extracting vertices in order of increasing distance. It runs in $O((V + E) \log V)$ time but requires non-negative edge weights. It is the standard choice for road networks, routing protocols, and other practical applications.

Bellman-Ford relaxes every edge $V - 1$ times, running in $O(VE)$ time. It handles negative edge weights and detects negative-weight cycles. It is slower than Dijkstra but more general.

DAG shortest paths exploits the absence of cycles by processing vertices in topological order, achieving optimal $O(V + E)$ time. It handles negative weights and is useful for scheduling and critical-path analysis.

Floyd-Warshall computes all-pairs shortest paths using dynamic programming in $O(V^3)$ time and $O(V^2)$ space. It handles negative weights and detects negative cycles. It is simple to implement and efficient for dense graphs.

All four algorithms use **relaxation** as the core operation. They differ in the order of relaxations (greedy by distance, repeated over all edges, topological order, or systematic DP over intermediate vertices) and the resulting time-space trade-offs. In Chapter 14, we will see a related problem — finding **minimum spanning trees** — that also uses edge relaxation but optimizes a different objective.

Chapter 16

Minimum Spanning Trees

*In Chapter 13 we found shortest paths — the lightest routes between specific pairs of vertices. A different but equally important problem arises when we want to connect **all** vertices of a graph as cheaply as possible: laying cable between cities, wiring components on a circuit board, or clustering data points. The answer is a **minimum spanning tree (MST)**. In this chapter we define the MST problem, establish the theoretical foundation — the **cut property** and **cycle property** — that makes greedy algorithms correct, and present two classic algorithms: **Kruskal's algorithm**, which sorts edges and uses a Union-Find data structure, and **Prim's algorithm**, which grows a tree from a single vertex using a priority queue.*

16.1 The minimum spanning tree problem

Let $G = (V, E)$ be a connected, undirected graph with edge-weight function $w : E \to \mathbb{R}$. A **spanning tree** of G is a subgraph $T = (V, E_T)$ that:

1. includes every vertex of G,
2. is connected, and
3. is acyclic (a tree).

Any spanning tree of a graph with V vertices has exactly $V - 1$ edges. A **minimum spanning tree** is a spanning tree whose total edge weight

$$w(T) = \sum_{e \in E_T} w(e)$$

is minimized over all spanning trees of G. An MST is not necessarily unique — a graph can have multiple spanning trees with the same minimum total weight — but the minimum weight itself is unique.

If G is disconnected, no spanning tree exists; instead we can find a **minimum spanning forest**, a collection of MSTs, one for each connected component.

16.1.1 Where MSTs appear

Minimum spanning trees arise naturally in many settings:

- **Network design.** Connecting cities with the least total cable, pipe, or road.
- **Cluster analysis.** Removing the $k-1$ most expensive edges from an MST partitions data into k clusters (single-linkage clustering).
- **Approximation algorithms.** The MST provides a 2-approximation for the metric Travelling Salesman Problem (Chapter 22).
- **Image segmentation.** Treating pixels as vertices and pixel differences as edge weights, the MST captures the structure of an image.

16.2 Theoretical foundation

Both Kruskal's and Prim's algorithms are greedy — they build the MST by making locally optimal edge choices. The **cut property** and **cycle property** guarantee that these local choices lead to a globally optimal solution.

16.2.1 Cuts and light edges

A **cut** $(S, V \setminus S)$ of a graph $G = (V, E)$ is a partition of the vertex set into two non-empty subsets. An edge **crosses** the cut if its endpoints are in different subsets. A cut **respects** a set A of edges if no edge in A crosses the cut. A **light edge** of a cut is a crossing edge with minimum weight among all crossing edges.

16.2.2 The cut property

Theorem (Cut Property). Let A be a subset of some MST of G, and let $(S, V \setminus S)$ be any cut that respects A. Let $e = (u, v)$ be a light edge crossing the cut. Then $A \cup \{e\}$ is a subset of some MST.

Proof sketch. Let T be an MST containing A. If T already contains e, we are done. Otherwise, adding e to T creates a cycle. This cycle must contain another edge e' crossing the cut (since e crosses it and the cycle returns to the same side). Because e is a light edge, $w(e) \leq w(e')$. The tree $T' = T - \{e'\} + \{e\}$ is a spanning tree with $w(T') \leq w(T)$, so T' is also an MST containing $A \cup \{e\}$. □

16.2.3 The cycle property

Theorem (Cycle Property). Let C be any cycle in G, and let e be the unique heaviest edge in C (strictly heavier than all other edges in C). Then e does not

belong to any MST.

Proof sketch. Suppose for contradiction that some MST T contains e. Removing e from T splits T into two components. Since C is a cycle, there exists another edge e' in C connecting these two components. We have $w(e') < w(e)$, so replacing e with e' yields a spanning tree with smaller weight — contradicting the minimality of T. □

The cut property tells us which edges are **safe** to add; the cycle property tells us which edges are **safe to exclude**. Both Kruskal's and Prim's algorithms are instantiations of a generic greedy MST strategy that repeatedly applies the cut property.

16.3 Union-Find: the key data structure for Kruskal's algorithm

Kruskal's algorithm needs to efficiently determine whether adding an edge creates a cycle. This reduces to asking: "Are vertices u and v in the same connected component?" The **Union-Find** (also called **Disjoint Set Union**) data structure answers this question in nearly constant time.

Union-Find maintains a collection of disjoint sets and supports three operations:

- **makeSet(x)** — create a singleton set $\{x\}$.
- **find(x)** — return the representative (root) of the set containing x.
- **union(x, y)** — merge the sets containing x and y.

16.3.1 Union by rank

Each set is stored as a rooted tree. The **rank** of a node is an upper bound on its height. When merging two sets, we attach the shorter tree beneath the taller one, keeping the overall tree shallow:

```
union(x, y):
    rootX = find(x)
    rootY = find(y)
    if rootX == rootY: return           // already in same set
    if rank[rootX] < rank[rootY]:
        parent[rootX] = rootY
    else if rank[rootX] > rank[rootY]:
        parent[rootY] = rootX
    else:
        parent[rootY] = rootX
        rank[rootX] = rank[rootX] + 1
```

Without path compression, union by rank alone guarantees $O(\log n)$ time per `find`.

16.3.2 Path compression

During a `find` operation, we make every node on the path from x to the root point directly to the root. This "flattens" the tree, speeding up subsequent queries:

```
find(x):
    root = x
    while parent[root] != root:
        root = parent[root]
    // Compress: point every node on the path to root
    while x != root:
        next = parent[x]
        parent[x] = root
        x = next
    return root
```

16.3.3 Combined complexity

With both path compression and union by rank, any sequence of m operations on n elements runs in $O(m \cdot \alpha(n))$ time, where α is the **inverse Ackermann function**. This function grows so slowly that $\alpha(n) \leq 4$ for any n up to $2^{2^{2^{65536}}}$ — far beyond the number of atoms in the observable universe. For all practical purposes, each operation is $O(1)$.

16.3.4 Implementation

```
export class UnionFind<T> {
  private parent = new Map<T, T>();
  private rank = new Map<T, number>();
  private _componentCount = 0;

  makeSet(x: T): void {
    if (this.parent.has(x)) return;
    this.parent.set(x, x);
    this.rank.set(x, 0);
    this._componentCount++;
  }

  find(x: T): T {
```

```
    let root = x;
    while (this.parent.get(root) !== root) {
      root = this.parent.get(root)!;
    }
    // Path compression.
    let current = x;
    while (current !== root) {
      const next = this.parent.get(current)!;
      this.parent.set(current, root);
      current = next;
    }
    return root;
  }

  union(x: T, y: T): boolean {
    const rootX = this.find(x);
    const rootY = this.find(y);
    if (rootX === rootY) return false;

    const rankX = this.rank.get(rootX)!;
    const rankY = this.rank.get(rootY)!;
    if (rankX < rankY) {
      this.parent.set(rootX, rootY);
    } else if (rankX > rankY) {
      this.parent.set(rootY, rootX);
    } else {
      this.parent.set(rootY, rootX);
      this.rank.set(rootX, rankX + 1);
    }
    this._componentCount--;
    return true;
  }

  connected(x: T, y: T): boolean {
    return this.find(x) === this.find(y);
  }

  get componentCount(): number {
    return this._componentCount;
  }
}
```

We will revisit Union-Find in greater depth in Chapter 18, including a more thorough discussion of the amortized analysis and additional applications such as dynamic connectivity.

16.4 Kruskal's algorithm

Kruskal's algorithm (1956) builds the MST by processing edges in order of increasing weight. For each edge, it checks whether the edge connects two different components; if so, it adds the edge to the MST and merges the components.

16.4.1 Algorithm

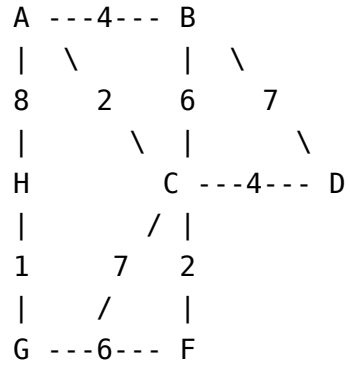
```
Kruskal(G):
    sort edges of G by weight (ascending)
    initialize Union-Find with all vertices
    MST = {}
    for each edge (u, v, w) in sorted order:
        if find(u) != find(v):        // u and v in different components
            MST = MST ∪ {(u, v, w)}
            union(u, v)
    return MST
```

16.4.2 Why it works

Each time Kruskal's adds an edge (u, v), the two components containing u and v define a cut: S is the component containing u, and $V \setminus S$ contains v. Edge (u, v) is the lightest crossing edge (since we process edges in sorted order and all lighter crossing edges have already been processed — either added or rejected because they were within a single component). By the cut property, adding (u, v) is safe.

16.4.3 Trace through an example

Consider this weighted graph:



Sorted edges: $(G, H, 1)$, $(A, C, 2)$, $(C, F, 2)$, $(A, B, 4)$, $(C, D, 4)$, $(B, C, 6)$, $(F, G, 6)$, $(B, D, 7)$, $(C, G, 7)$, $(A, H, 8)$.

Step	Edge	Weight	Action	Components
1	(G, H)	1	Add	$\{G, H\}$, $\{A\}$, $\{B\}$, $\{C\}$, $\{D\}$, $\{F\}$
2	(A, C)	2	Add	$\{G, H\}$, $\{A, C\}$, $\{B\}$, $\{D\}$, $\{F\}$
3	(C, F)	2	Add	$\{G, H\}$, $\{A, C, F\}$, $\{B\}$, $\{D\}$
4	(A, B)	4	Add	$\{G, H\}$, $\{A, B, C, F\}$, $\{D\}$
5	(C, D)	4	Add	$\{G, H\}$, $\{A, B, C, D, F\}$
6	(B, C)	6	Reject	B and C in same component
7	(F, G)	6	Add	$\{A, B, C, D, F, G, H\}$

After adding 6 edges (which is $V - 1$ for our 7-vertex graph), the MST is complete with total weight $1 + 2 + 2 + 4 + 4 + 6 = 19$.

16.4.4 Implementation

```
import type { Edge } from '../types.js';
import { Graph } from '../12-graphs-and-traversal/graph.js';
import { UnionFind } from '../18-disjoint-sets/union-find.js';

export interface MSTResult<T> {
  edges: Edge<T>[];
  weight: number;
}

export function kruskal<T>(graph: Graph<T>): MSTResult<T> {
  const vertices = graph.getVertices();
  const edges = graph.getEdges();

  // Sort edges by weight (ascending).
  edges.sort((a, b) => a.weight - b.weight);

  // Initialize Union-Find with all vertices.
  const uf = new UnionFind<T>();
```

```
  for (const v of vertices) {
    uf.makeSet(v);
  }

  const mstEdges: Edge<T>[] = [];
  let totalWeight = 0;

  for (const edge of edges) {
    if (!uf.connected(edge.from, edge.to)) {
      uf.union(edge.from, edge.to);
      mstEdges.push(edge);
      totalWeight += edge.weight;

      // An MST of V vertices has exactly V - 1 edges.
      if (mstEdges.length === vertices.length - 1) break;
    }
  }

  return { edges: mstEdges, weight: totalWeight };
}
```

16.4.5 Complexity

- **Time:** $O(E \log E)$ for sorting, plus $O(E \cdot \alpha(V))$ for the union-find operations. Since $\log E = O(\log V)$ (because $E \leq V^2$), the total is $O(E \log V)$.
- **Space:** $O(V + E)$ for the edge list and union-find structure.

Kruskal's algorithm is particularly well-suited for **sparse** graphs, where E is much smaller than V^2, and for situations where the edges are already available as a sorted list (e.g., from an external data source).

16.5 Prim's algorithm

Prim's algorithm (1957, independently discovered by Jarnik in 1930) takes a different approach: it grows the MST from a single starting vertex, always adding the lightest edge that connects the tree to a new vertex.

16.5.1 Algorithm

```
Prim(G, start):
    initialize priority queue PQ
    visited = {start}
```

```
    insert all edges from start into PQ
    MST = {}
    while PQ is not empty and |MST| < |V| - 1:
        (u, v, w) = PQ.extractMin()      // lightest frontier edge
        if v in visited: continue         // already in tree
        visited = visited ∪ {v}
        MST = MST ∪ {(u, v, w)}
        for each edge (v, x, w') where x not in visited:
            PQ.insert((v, x, w'))
    return MST
```

16.5.2 Why it works

At each step, the set of visited vertices defines one side of a cut, and the unvisited vertices form the other side. The priority queue ensures that we always select a light edge crossing this cut. By the cut property, this edge is safe to add.

16.5.3 Trace through an example

Using the same graph as before, starting from vertex A:

Step	Extract	Weight	Add to tree	Frontier edges added
0	—	—	start at A	$(A, B, 4)$, $(A, C, 2)$, $(A, H, 8)$
1	(A, C)	2	C	$(C, B, 6)$, $(C, D, 4)$, $(C, F, 2)$, $(C, G, 7)$
2	(C, F)	2	F	$(F, G, 6)$
3	(A, B)	4	B	$(B, D, 7)$
4	(C, D)	4	D	—
5	(F, G)	6	G	$(G, H, 1)$
6	(G, H)	1	H	—

MST weight: $2 + 2 + 4 + 4 + 6 + 1 = 19$ — the same as Kruskal's result.

Notice that the edges may be added in a different order than Kruskal's, but the total weight is identical.

16.5.4 Implementation

```typescript
import type { Edge } from '../types.js';
import { Graph } from '../12-graphs-and-traversal/graph.js';
import { BinaryHeap } from '../11-heaps-and-priority-queues/binary-heap.js';

interface HeapEntry<T> {
```

```
  vertex: T;
  weight: number;
  from: T;
}

export function prim<T>(graph: Graph<T>, start?: T): MSTResult<T> {
  const vertices = graph.getVertices();
  const source = start ?? vertices[0]!;

  const visited = new Set<T>();
  const mstEdges: Edge<T>[] = [];
  let totalWeight = 0;

  // Min-heap ordered by edge weight.
  const heap = new BinaryHeap<HeapEntry<T>>(
    (a, b) => a.weight - b.weight,
  );

  // Seed the heap with all edges from the source.
  visited.add(source);
  for (const [neighbor, weight] of graph.getNeighbors(source)) {
    heap.insert({ vertex: neighbor, weight, from: source });
  }

  while (!heap.isEmpty && visited.size < vertices.length) {
    const entry = heap.extract()!;
    if (visited.has(entry.vertex)) continue;

    // Add this vertex to the tree.
    visited.add(entry.vertex);
    mstEdges.push({
      from: entry.from,
      to: entry.vertex,
      weight: entry.weight,
    });
    totalWeight += entry.weight;

    // Add frontier edges from the newly added vertex.
    for (const [neighbor, weight] of graph.getNeighbors(entry.vertex)) {
      if (!visited.has(neighbor)) {
        heap.insert({ vertex: neighbor, weight, from: entry.vertex });
```

```
      }
    }
  }

  return { edges: mstEdges, weight: totalWeight };
}
```

Our implementation uses a binary heap directly (rather than the `PriorityQueue` wrapper) for efficiency. Each edge may be inserted into the heap, and stale entries (edges to already-visited vertices) are simply discarded on extraction.

16.5.5 Complexity

- **Time:** $O(E \log V)$ with a binary heap. Each of the E edges is inserted into the heap (at most once), and each insertion/extraction costs $O(\log V)$. With a Fibonacci heap, this improves to $O(E + V \log V)$, which is better for dense graphs.
- **Space:** $O(V + E)$ for the visited set and the heap.

Prim's algorithm is well-suited for **dense** graphs, especially with a Fibonacci heap. For sparse graphs, Kruskal's is often simpler and equally efficient.

16.6 Kruskal's vs. Prim's

Feature	Kruskal's	Prim's
Strategy	Global edge sorting	Local vertex growing
Data structure	Union-Find	Priority queue (heap)
Time (binary heap)	$O(E \log V)$	$O(E \log V)$
Time (Fibonacci heap)	—	$O(E + V \log V)$
Best for	Sparse graphs	Dense graphs
Parallelism	Edges can be processed in parallel (with concurrent union-find)	Inherently sequential
Disconnected graphs	Produces spanning forest naturally	Spans only one component per call
Simplicity	Very simple to implement	Slightly more complex

Both algorithms produce MSTs of identical total weight. When the graph is sparse ($E = O(V)$), Kruskal's is often preferred for its simplicity. When the graph

is dense ($E = \Theta(V^2)$) and a Fibonacci heap is available, Prim's has a theoretical edge.

16.7 Correctness and uniqueness

16.7.1 When is the MST unique?

An MST is unique if and only if every cut of the graph has a unique light edge. Equivalently, if all edge weights are distinct, the MST is unique. When edges share weights, there may be multiple MSTs, but they all have the same total weight.

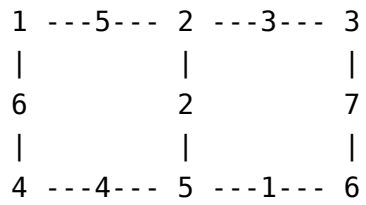
16.7.2 Verifying an MST

Given a claimed MST, we can verify it in $O(E \cdot \alpha(V))$ time by checking:

1. The tree has exactly $V - 1$ edges.
2. The tree spans all vertices (use union-find or BFS/DFS).
3. For every non-tree edge (u, v), the weight of (u, v) is at least as large as the maximum edge weight on the path from u to v in the tree (cycle property).

16.8 Exercises

Exercise 14.1. Run Kruskal's algorithm on the following weighted graph. Show the state of the Union-Find structure after each edge addition and the final MST.



Exercise 14.2. Run Prim's algorithm on the same graph from Exercise 14.1, starting from vertex 1. Show the contents of the priority queue after each step.

Exercise 14.3. Prove that if all edge weights are distinct, the minimum spanning tree is unique. (*Hint: assume two distinct MSTs exist and derive a contradiction using the cycle property.*)

Exercise 14.4. A **bottleneck spanning tree** is a spanning tree that minimizes the weight of its maximum-weight edge. Prove that every MST is a bottleneck spanning tree. Is the converse true?

Exercise 14.5. You are given a connected, weighted, undirected graph G and its MST T. A new edge (u, v, w) is added to G. Describe an efficient algorithm

to update the MST. What is the time complexity? (*Hint: adding the new edge to T creates exactly one cycle.*)

16.9 Summary

A **minimum spanning tree** of a connected, undirected, weighted graph is a spanning tree with minimum total edge weight. The **cut property** guarantees that the lightest edge crossing any cut is safe to include, while the **cycle property** guarantees that the heaviest edge in any cycle is safe to exclude.

Kruskal's algorithm sorts all edges by weight and greedily adds edges that do not create a cycle, using a **Union-Find** data structure for efficient cycle detection. It runs in $O(E \log V)$ time and naturally produces a spanning forest for disconnected graphs.

Prim's algorithm grows the MST from a single vertex, always adding the lightest edge connecting the tree to a new vertex, using a **priority queue** to select the minimum-weight frontier edge. It also runs in $O(E \log V)$ with a binary heap, improving to $O(E + V \log V)$ with a Fibonacci heap.

Both algorithms are greedy, both are correct by the cut property, and both produce MSTs of identical total weight. Kruskal's is typically preferred for sparse graphs and for its simplicity; Prim's is preferred for dense graphs, especially when a Fibonacci heap is available. The Union-Find data structure introduced here — with path compression and union by rank — achieves near-constant amortized time per operation and will reappear in Chapter 18 and in the approximation algorithms of Chapter 22.

Chapter 17

Network Flow

In Chapters 12–14 we studied graphs from the perspective of connectivity and distance — traversals, shortest paths, and spanning trees. In this chapter we shift focus to a fundamentally different question: ***how much "stuff" can we push through a network?*** *Imagine oil flowing through a pipeline, data packets traversing a computer network, or goods moving through a supply chain. Each link has a limited capacity, and we want to maximize the total throughput from a designated* ***source*** *to a designated* ***sink****. This is the* ***maximum flow*** *problem, one of the most versatile tools in combinatorial optimization. We develop the* ***Ford-Fulkerson method****, prove the celebrated* ***max-flow min-cut theorem****, and implement the efficient* ***Edmonds-Karp*** *variant that guarantees polynomial running time. We then show how maximum flow solves the* ***maximum bipartite matching*** *problem — assigning jobs to workers, students to schools, or organs to patients.*

17.1 Flow networks

A **flow network** is a directed graph $G = (V, E)$ in which each edge $(u, v) \in E$ has a non-negative **capacity** $c(u, v) \geq 0$. Two distinguished vertices are the **source** s and the **sink** t, where $s \neq t$. We assume that every vertex lies on some path from s to t (otherwise it is irrelevant to the flow problem).

If $(u, v) \notin E$, we define $c(u, v) = 0$ for convenience.

17.1.1 Flows

A **flow** in G is a function $f : V \times V \to \mathbb{R}$ satisfying two constraints:

1. **Capacity constraint.** For all $u, v \in V$:

$$0 \leq f(u, v) \leq c(u, v)$$

2. **Flow conservation.** For every vertex $u \in V \setminus \{s, t\}$:

$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$$

In words, the flow into any internal vertex equals the flow out of it — flow is neither created nor destroyed except at the source and sink.

The **value** of a flow f is the net flow leaving the source:

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$$

The **maximum flow problem** asks: find a flow f of maximum value $|f|$.

17.1.2 Where network flow appears

Network flow arises in a remarkable variety of applications:

- **Transportation and logistics.** Routing goods through a supply chain with capacity-limited links.
- **Computer networks.** Maximizing data throughput between two hosts.
- **Bipartite matching.** Assigning workers to jobs, students to projects, or doctors to hospitals (we cover this later in the chapter).
- **Image segmentation.** Partitioning an image into foreground and background by finding a minimum cut.
- **Baseball elimination.** Determining whether a team has been mathematically eliminated from contention.
- **Project selection.** Choosing which projects to fund when some projects depend on others.

The power of network flow lies not just in the max-flow problem itself, but in the large number of combinatorial problems that **reduce** to it.

17.2 The Ford-Fulkerson method

The Ford-Fulkerson method (1956) is a general strategy for computing maximum flow. It repeatedly finds **augmenting paths** — paths from source to sink along which more flow can be pushed — and increases the flow until no augmenting path remains.

17.2.1 Residual graphs

Given a flow network G and a flow f, the **residual graph** G_f has the same vertex set as G and contains two types of edges for each original edge (u, v):

1. **Forward edge** (u, v) with residual capacity $c_f(u, v) = c(u, v) - f(u, v)$, representing unused capacity that can still carry more flow.
2. **Reverse edge** (v, u) with residual capacity $c_f(v, u) = f(u, v)$, representing flow that can be "cancelled" — pushed back — to reroute it through a better path.

An edge appears in G_f only if its residual capacity is positive.

17.2.2 Augmenting paths

An **augmenting path** is a simple path from s to t in the residual graph G_f. The **bottleneck capacity** of the path is the minimum residual capacity along its edges:

$$c_f(p) = \min_{(u,v)\in p} c_f(u, v)$$

We can increase the flow by $c_f(p)$ by pushing flow along the augmenting path: for each forward edge, increase the flow; for each reverse edge, decrease the flow on the corresponding original edge.

17.2.3 The Ford-Fulkerson algorithm

```
FordFulkerson(G, s, t):
    Initialize f(u, v) = 0 for all (u, v)
    while there exists an augmenting path p in G_f:
        c_f(p) = min residual capacity along p
        for each edge (u, v) in p:
            if (u, v) is a forward edge:
                f(u, v) = f(u, v) + c_f(p)
            else:  // (u, v) is a reverse edge
                f(v, u) = f(v, u) - c_f(p)
    return f
```

The method is correct but does not specify **how** to find the augmenting path. Different choices lead to different running times. With arbitrary path selection and irrational capacities, Ford-Fulkerson may not even terminate. The **Edmonds-Karp** variant fixes this by using BFS.

17.3 The max-flow min-cut theorem

Before presenting Edmonds-Karp, let us establish the theoretical foundation that justifies the Ford-Fulkerson approach.

17.3.1 Cuts

A **cut** (S, T) of a flow network is a partition of V into two sets S and $T = V \setminus S$ such that $s \in S$ and $t \in T$. The **capacity** of a cut is the sum of capacities of edges crossing from S to T:

$$c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v)$$

Note that we only count edges from S to T, not from T to S.

The **net flow across a cut** is:

$$f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in S} \sum_{v \in T} f(v, u)$$

A key lemma: for any flow f and any cut (S, T), the net flow across the cut equals the value of the flow: $f(S, T) = |f|$. This follows from flow conservation at internal vertices.

Since the flow across any cut cannot exceed the cut's capacity, we get:

$$|f| = f(S, T) \leq c(S, T)$$

This holds for **every** cut — so the maximum flow is at most the minimum cut capacity.

17.3.2 The theorem

Theorem (Max-Flow Min-Cut). In any flow network, the following three conditions are equivalent:

1. f is a maximum flow.
2. The residual graph G_f contains no augmenting path from s to t.
3. $|f| = c(S, T)$ for some cut (S, T).

Proof sketch. (1 $\Rightarrow$ 2): If an augmenting path existed, we could increase the flow, contradicting maximality. (2 $\Rightarrow$ 3): If no augmenting path exists, define S as the set of vertices reachable from s in G_f. Since $t \notin S$, $(S, V \setminus S)$ is a valid cut.

Every edge from S to $V \setminus S$ must be saturated (otherwise the endpoint would be reachable), and every edge from $V \setminus S$ to S must carry zero flow (otherwise the reverse edge would be in G_f). Therefore $f(S,T) = c(S,T)$. (3 $\Rightarrow$ 1): Since $|f| \leq c(S,T)$ for all cuts, equality with some cut implies f is maximum. $\square$

This theorem has a profound consequence: **the maximum flow through a network equals the minimum capacity of any cut separating source from sink**. It also tells us that when the Ford-Fulkerson method terminates (no augmenting path exists), the flow is guaranteed to be maximum. As a bonus, the source-side vertices reachable in the final residual graph give us the minimum cut.

17.4 Edmonds-Karp algorithm

The Edmonds-Karp algorithm (1972) is a refinement of Ford-Fulkerson that uses **breadth-first search** (BFS) to find augmenting paths. By always choosing a **shortest** augmenting path (fewest edges), it guarantees termination in $O(VE)$ augmenting path iterations, giving a total running time of $O(VE^2)$.

17.4.1 Why shortest augmenting paths?

The key insight is that when we always augment along shortest paths, the distances in the residual graph never decrease over successive iterations. More precisely:

Lemma. Let $\delta_f(s,v)$ denote the shortest-path distance (number of edges) from s to v in the residual graph G_f. If Edmonds-Karp augments flow f to obtain flow f', then $\delta_{f'}(s,v) \geq \delta_f(s,v)$ for all v.

This monotonicity property, combined with the observation that each augmenting path saturates at least one edge (which then temporarily disappears from the residual graph), yields:

Theorem. The Edmonds-Karp algorithm performs at most $O(VE)$ augmenting path iterations.

Since each BFS takes $O(V+E)$ time, the total running time is $O(VE^2)$. For dense graphs this is $O(V^5)$; for sparse graphs it is $O(V^2E)$.

17.4.2 Pseudocode

```
EdmondsKarp(G, s, t):
    Initialize f(u, v) = 0 for all (u, v)
    repeat:
        // BFS in residual graph to find shortest augmenting path
```

```
        parent = BFS(G_f, s, t)
        if t is not reachable: break

        // Find bottleneck capacity
        bottleneck = infinity
        v = t
        while v != s:
            u = parent[v]
            bottleneck = min(bottleneck, c_f(u, v))
            v = u

        // Augment flow along the path
        v = t
        while v != s:
            u = parent[v]
            push bottleneck units of flow along (u, v)
            v = u

        maxFlow = maxFlow + bottleneck

    return maxFlow
```

17.4.3 Trace through an example

Consider the following flow network (based on the classic CLRS example):

Edge	Capacity
s → v1	16
s → v2	13
v1 → v2	4
v1 → v3	12
v2 → v1	10
v2 → v4	14
v3 → v2	9
v3 → t	20
v4 → v3	7
v4 → t	4

Iteration 1. BFS finds the shortest path s → v1 → v3 → t (3 edges). Bottleneck = min(16, 12, 20) = 12. Push 12 units. Total flow = 12.

After augmentation, the residual graph has: - s → v1: residual 4 (was 16, used 12) - v1 → v3: residual 0 (saturated) - v3 → v1: residual 12 (reverse edge) - v3 → t: residual 8 (was 20, used 12)

Iteration 2. BFS finds s → v2 → v4 → t (3 edges). Bottleneck = min(13, 14, 4) = 4. Push 4 units. Total flow = 16.

Iteration 3. BFS finds s → v2 → v4 → v3 → t (4 edges). Bottleneck = min(9, 10, 7, 8) = 7. Push 7 units. Total flow = 23.

After iteration 3, no augmenting path exists in the residual graph. The **maximum flow is 23**.

The minimum cut is $S = \{s, v1, v2, v4\}$, $T = \{v3, t\}$. The cut edges and their capacities are:

Cut edge	Capacity
v1 → v3	12
v4 → v3	7
v4 → t	4
Total	**23**

This confirms the max-flow min-cut theorem: the minimum cut capacity equals the maximum flow.

17.4.4 TypeScript implementation

Our implementation uses a self-contained residual graph structure with efficient integer-keyed maps. Vertices of any type are supported — each vertex is assigned a unique integer ID, and edge capacities are stored in a compact map keyed by Cantor-paired vertex IDs.

The result type captures the max flow value, the per-edge flow assignment, and the min-cut:

```
export interface FlowEdge<T> {
  from: T;
  to: T;
  capacity: number;
  flow: number;
}

export interface MaxFlowResult<T> {
  maxFlow: number;
```

```
  flowEdges: FlowEdge<T>[];
  minCut: Set<T>;
}
```

The core algorithm follows the Edmonds-Karp approach — BFS for augmenting paths, bottleneck computation, and flow augmentation:

```
export function edmondsKarp<T>(
  edges: { from: T; to: T; capacity: number }[],
  source: T,
  sink: T,
): MaxFlowResult<T> {
  if (source === sink) {
    throw new Error('Source and sink must be different vertices');
  }

  const residual = new ResidualGraph<T>();
  residual.addVertex(source);
  residual.addVertex(sink);
  for (const { from, to, capacity } of edges) {
    residual.addEdge(from, to, capacity);
  }

  let maxFlow = 0;

  while (true) {
    const parent = residual.bfs(source, sink);
    if (parent === null) break;

    // Find the bottleneck capacity along the path.
    let bottleneck = Infinity;
    let v: T = sink;
    while (v !== source) {
      const u = parent.get(v) as T;
      bottleneck = Math.min(
        bottleneck,
        residual.getResidualCapacity(u, v),
      );
      v = u;
    }

    // Augment flow along the path.
```

```
    v = sink;
    while (v !== source) {
      const u = parent.get(v) as T;
      residual.pushFlow(u, v, bottleneck);
      v = u;
    }

    maxFlow += bottleneck;
  }

  // The min-cut is the set of vertices reachable from the source
  // in the final residual graph (BFS from source with no path to sink).
  const minCut = residual.reachableFrom(source);
  const flowEdges = residual.getFlowEdges();

  return { maxFlow, flowEdges, minCut };
}
```

The residual graph internally maps each vertex to a sequential integer ID and uses Cantor pairing to compute a single numeric key for each edge. This ensures correct behavior even when vertices are objects (where `String()` would not produce unique keys).

After termination, the algorithm computes the **minimum cut** by running BFS from the source in the final residual graph. The set of reachable vertices forms the source side S of the min-cut — exactly as prescribed by the max-flow min-cut theorem.

17.4.5 Complexity analysis

- **Time:** $O(VE^2)$. Each BFS takes $O(V + E)$. The number of augmenting path iterations is bounded by $O(VE)$ because: (a) distances in the residual graph never decrease; and (b) after at most $O(E)$ augmentations at a given distance, some critical edge is permanently saturated, increasing the distance. Since distances are bounded by $O(V)$, we get $O(VE)$ iterations total.
- **Space:** $O(V+E)$ for the residual graph, adjacency lists, and BFS data structures.

17.5 Application: maximum bipartite matching

One of the most elegant applications of network flow is solving the **maximum bipartite matching** problem.

17.5.1 The matching problem

A **bipartite graph** $G = (L \cup R, E)$ has two disjoint vertex sets L (left) and R (right), with edges only between L and R. A **matching** is a subset $M \subseteq E$ such that no vertex appears in more than one edge of M. A **maximum matching** is a matching of largest possible size.

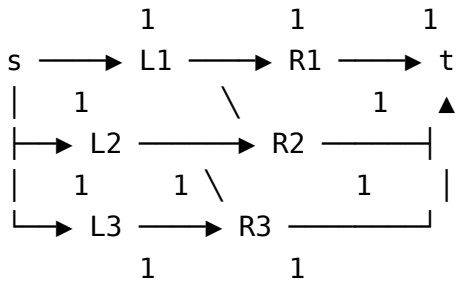
Bipartite matching models many real-world assignment problems:

- **Job assignment.** L = workers, R = jobs, edge (w, j) means worker w is qualified for job j. Maximum matching assigns the most workers to jobs.
- **Course enrollment.** L = students, R = courses. Maximum matching enrolls the most students.
- **Organ donation.** L = donors, R = recipients. Maximum matching saves the most lives.

17.5.2 Reduction to max flow

We reduce bipartite matching to max flow by constructing a flow network:

1. Add a **super-source** s and a **super-sink** t.
2. For each left vertex $l \in L$, add edge (s, l) with capacity 1.
3. For each right vertex $r \in R$, add edge (r, t) with capacity 1.
4. For each bipartite edge $(l, r) \in E$, add edge (l, r) with capacity 1.



Why it works. Since all capacities are 1, any integer flow corresponds to a matching:

- Capacity-1 edges from s to L ensure each left vertex sends at most 1 unit of flow — it is matched to at most one right vertex.
- Capacity-1 edges from R to t ensure each right vertex receives at most 1 unit — it is matched to at most one left vertex.
- An edge (l, r) carries flow 1 if and only if l is matched to r.

The **integrality theorem** for network flow guarantees that when all capacities are integers, there exists a maximum flow that is also integral. Therefore the maximum flow value equals the maximum matching size.

17.5.3 TypeScript implementation

```
export interface BipartiteMatchingResult<L, R> {
  size: number;
  matches: [L, R][];
}

export function bipartiteMatching<L, R>(
  left: L[],
  right: R[],
  edges: [L, R][],
): BipartiteMatchingResult<L, R> {
  const source = { kind: 'source' };
  const sink = { kind: 'sink' };

  const leftVertices = new Map<L, FlowVertex>();
  const rightVertices = new Map<R, FlowVertex>();

  for (const l of left)
    leftVertices.set(l, { kind: 'left', value: l });
  for (const r of right)
    rightVertices.set(r, { kind: 'right', value: r });

  const flowEdges = [];

  for (const lv of leftVertices.values())
    flowEdges.push({ from: source, to: lv, capacity: 1 });
  for (const rv of rightVertices.values())
    flowEdges.push({ from: rv, to: sink, capacity: 1 });
  for (const [l, r] of edges) {
    const lv = leftVertices.get(l);
    const rv = rightVertices.get(r);
    if (lv && rv)
      flowEdges.push({ from: lv, to: rv, capacity: 1 });
  }

  const result = edmondsKarp(flowEdges, source, sink);
```

```
  const matches = [];
  for (const fe of result.flowEdges) {
    if (fe.flow === 1
        && fe.from.kind === 'left'
        && fe.to.kind === 'right') {
      matches.push([fe.from.value, fe.to.value]);
    }
  }

  return { size: result.maxFlow, matches };
}
```

The implementation uses tagged vertex objects (`{ kind: 'left', value: l }`) to prevent name collisions between left vertices, right vertices, the source, and the sink. Since our Edmonds-Karp implementation uses identity-based vertex comparison (via `Map`), these object vertices are compared by reference — exactly what we need.

17.5.4 Complexity analysis

In the constructed flow network, $|V| = |L| + |R| + 2$ and $|E| = |L| + |R| + |E_{\text{bipartite}}|$. With unit capacities, Edmonds-Karp terminates in $O(V)$ augmenting path iterations (since each augmentation increases the flow by 1 and the maximum flow is at most $\min(|L|, |R|)$), giving:

- **Time:** $O(V \cdot E)$ where $V = |L| + |R|$ and E is the number of bipartite edges.
- **Space:** $O(V + E)$ for the flow network.

17.5.5 Trace through an example

Consider assigning workers to jobs:

Worker	Qualified for
Alice	Job 1, Job 2
Bob	Job 1
Carol	Job 2, Job 3

The bipartite graph has $L = \{\text{Alice}, \text{Bob}, \text{Carol}\}$ and $R = \{\text{Job 1}, \text{Job 2}, \text{Job 3}\}$.

Iteration 1. BFS finds s → Alice → Job1 → t. Push 1 unit. Flow = 1.

Iteration 2. BFS finds s → Bob → Job1, but Job1 → t is saturated. Through the reverse edge (Job1 → Alice, residual capacity 1), BFS discovers the path: s → Bob → Job1 → Alice → Job2 → t. Push 1 unit. Flow = 2.

This rerouting is the power of augmenting paths in matching: Bob "steals" Job 1 from Alice, and Alice is reassigned to Job 2.

Iteration 3. BFS finds s → Carol → Job3 → t. Push 1 unit. Flow = 3.

Result: Maximum matching of size 3: {Bob → Job 1, Alice → Job 2, Carol → Job 3}.

Notice how the algorithm found a perfect matching even though a greedy approach (match Alice → Job 1 first) would have left Bob unmatched. The augmenting path through reverse edges enabled the rerouting.

17.6 Beyond Edmonds-Karp

The Edmonds-Karp algorithm is a clean, practical choice for many applications, but faster max-flow algorithms exist:

Algorithm	Time complexity	Notes
Ford-Fulkerson (DFS)	$O(E \cdot f^*)$	f^* = max flow value; not polynomial
Edmonds-Karp (BFS)	$O(VE^2)$	Polynomial; simple to implement
Dinic's algorithm	$O(V^2E)$	Uses blocking flows; faster in practice
Push-relabel	$O(V^2E)$ or $O(V^3)$	No augmenting paths; local operations
Orlin's algorithm	$O(VE)$	Optimal for sparse graphs

For bipartite matching specifically, **Hopcroft-Karp** achieves $O(E\sqrt{V})$ by finding multiple augmenting paths simultaneously.

In practice, Edmonds-Karp and Dinic's are the most commonly implemented. Dinic's algorithm is particularly effective on unit-capacity networks (like bipartite matching), where it achieves $O(E\sqrt{V})$ — matching Hopcroft-Karp.

17.7 Exercises

Exercise 15.1. Consider the following flow network with edges: s → A (capacity 5), s → B (capacity 3), A → t (capacity 4), A → C (capacity 2), B → C (capacity 5), C → t (capacity 6).

(a) Find the maximum flow by tracing Edmonds-Karp (BFS-based augmenting paths).
(b) Identify the minimum cut and verify that its capacity equals the max flow.
(c) What is the flow assignment on each edge?

Exercise 15.2. Prove that in any flow network, the total flow into the sink equals the total flow out of the source. (Hint: sum the flow conservation constraints over all vertices except s and t.)

Exercise 15.3. A company has 4 workers and 4 tasks. The qualification matrix is:

	Task A	Task B	Task C	Task D
Worker 1	Yes	Yes		
Worker 2		Yes	Yes	
Worker 3	Yes		Yes	Yes
Worker 4				Yes

(a) Model this as a bipartite matching problem and find the maximum matching.
(b) Is a perfect matching possible? If so, find one. If not, explain why.

Exercise 15.4. Modify the Edmonds-Karp algorithm to handle **lower bounds** on edge flows: each edge (u, v) has both a capacity $c(u, v)$ and a minimum flow requirement $\ell(u, v)$, so $\ell(u, v) \leq f(u, v) \leq c(u, v)$. Describe how to transform this into a standard max-flow problem. (Hint: introduce excess supply and demand at vertices based on the lower bounds.)

Exercise 15.5. König's theorem states that in a bipartite graph, the size of the maximum matching equals the size of the minimum vertex cover. Using the max-flow min-cut theorem applied to the bipartite matching reduction, prove König's theorem. (Hint: show how the minimum cut in the flow network corresponds to a minimum vertex cover in the bipartite graph.)

17.8 Summary

In this chapter we studied network flow — a rich framework for maximizing throughput in capacity-constrained networks.

- A **flow network** is a directed graph with edge capacities, a source, and a sink. A **flow** assigns values to edges satisfying capacity and conservation constraints.
- The **Ford-Fulkerson method** finds maximum flow by iteratively discovering **augmenting paths** in the **residual graph** and pushing flow along them.
- The **max-flow min-cut theorem** proves that the maximum flow equals the minimum cut capacity — a deep duality result that connects optimization (max flow) with combinatorics (min cut).
- **Edmonds-Karp** uses BFS to find shortest augmenting paths, guaranteeing $O(VE^2)$ time. This polynomial bound makes it practical for moderately sized networks.
- **Maximum bipartite matching** reduces elegantly to max flow: add a super-source and super-sink with unit-capacity edges, and the max flow equals the maximum matching size. The integrality theorem ensures integer solutions.
- The min-cut computed as a by-product of max flow identifies the source-reachable vertices in the final residual graph — useful for applications like image segmentation and network reliability analysis.

Network flow is one of the most versatile tools in algorithm design. Many problems that seem unrelated — assignment, scheduling, connectivity, and partitioning — can be modeled as flow problems and solved efficiently with the algorithms in this chapter.

Chapter 18

Dynamic Programming

In the preceding chapters we met two powerful algorithm design paradigms: divide-and-conquer (Chapter 3) breaks a problem into independent subproblems, and greedy algorithms (Chapter 17) build solutions by making locally optimal choices. Dynamic programming (DP) occupies the territory between them. Like divide-and-conquer, it solves problems by combining solutions to subproblems. But unlike divide-and-conquer, those subproblems ***overlap*** *— the same subproblem is needed by many larger subproblems. Instead of recomputing these answers, DP saves them in a table and reuses them, trading space for an often dramatic reduction in time. In this chapter we develop a systematic approach to dynamic programming and apply it to seven classic problems: Fibonacci numbers, coin change, longest common subsequence, edit distance, 0/1 knapsack, matrix chain multiplication, and the longest increasing subsequence.*

18.1 When does dynamic programming apply?

A problem is amenable to dynamic programming when it exhibits two properties:

1. **Optimal substructure.** An optimal solution to the problem contains optimal solutions to its subproblems. For example, if the shortest path from A to C passes through B, then the sub-path from A to B must itself be a shortest path from A to B.
2. **Overlapping subproblems.** The recursive decomposition of the problem leads to the same subproblems being solved many times. If every subproblem were solved only once, there would be nothing to save — and a straightforward divide-and-conquer approach would suffice.

When both properties hold, we can avoid redundant computation by storing subproblem solutions in a table and looking them up rather than recomputing them.

18.2 Memoization vs tabulation

There are two standard ways to implement dynamic programming:

18.2.1 Top-down with memoization

Start from the original problem and recurse. Before computing a subproblem, check whether its solution is already cached. If so, return the cached value; otherwise, compute it, cache it, and return it. This approach is sometimes called **memoization** (from "memo" — a note to oneself).

Advantages:

- Only solves subproblems that are actually needed.
- The recursive structure mirrors the mathematical recurrence directly.

Disadvantages:

- Recursion overhead (call stack).
- Possible stack overflow on very deep recursions.

18.2.2 Bottom-up with tabulation

Solve subproblems in an order such that when we need a subproblem's solution, it has already been computed. Typically this means solving subproblems from "smallest" to "largest" using iterative loops and storing results in an array or table.

Advantages:

- No recursion overhead.
- Constant per-subproblem overhead.
- Often allows space optimization (keeping only the last row or two of the table).

Disadvantages:

- Must determine a valid computation order in advance.
- May compute subproblems that are not needed for the final answer.

In practice, bottom-up tabulation is more common because it avoids stack overhead and enables space optimizations. We use it for most examples in this chapter.

18.3 A systematic approach to DP

For each problem in this chapter, we follow a five-step recipe:

1. **Define subproblems.** Characterize the space of subproblems in terms of one or more indices (or parameters).
2. **Write the recurrence.** Express the solution to a subproblem in terms of solutions to smaller subproblems.
3. **Identify base cases.** Determine the values of the smallest subproblems directly.
4. **Determine computation order.** Choose an order in which to fill the table so that dependencies are satisfied.
5. **Recover the solution.** Extract the answer from the table, and optionally backtrack to find the actual solution (not just its value).

18.4 Fibonacci numbers: the introductory example

The Fibonacci sequence is defined by:

$$F(0) = 0, \quad F(1) = 1, \quad F(n) = F(n-1) + F(n-2) \quad \text{for } n \geq 2$$

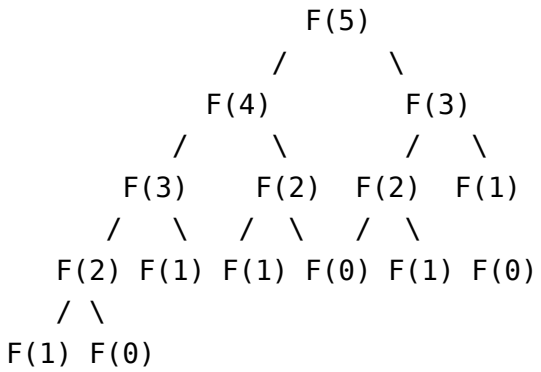
This is the simplest illustration of how DP transforms an exponential algorithm into a linear one.

18.4.1 Naive recursion

Directly translating the recurrence into code:

```
export function fibNaive(n: number): number {
  if (n < 0) throw new RangeError('n must be non-negative');
  if (n <= 1) return n;
  return fibNaive(n - 1) + fibNaive(n - 2);
}
```

The recursion tree for $F(5)$ shows massive redundancy:



$F(3)$ is computed twice, $F(2)$ three times, and so on. The total number of calls

grows exponentially — $O(2^n)$ — because the same subproblems are solved over and over.

18.4.2 Top-down with memoization

Adding a cache eliminates the redundancy:

```
export function fibMemo(n: number): number {
  if (n < 0) throw new RangeError('n must be non-negative');

  const memo = new Map<number, number>();

  function fib(k: number): number {
    if (k <= 1) return k;
    const cached = memo.get(k);
    if (cached !== undefined) return cached;
    const result = fib(k - 1) + fib(k - 2);
    memo.set(k, result);
    return result;
  }

  return fib(n);
}
```

Now each subproblem $F(k)$ is computed at most once and then looked up in $O(1)$ time, giving $O(n)$ total time and $O(n)$ space.

18.4.3 Bottom-up with tabulation

We can go further by eliminating the recursion entirely. Since $F(n)$ only depends on $F(n-1)$ and $F(n-2)$, we need to store only two values at any time:

```
export function fibTabulated(n: number): number {
  if (n < 0) throw new RangeError('n must be non-negative');
  if (n <= 1) return n;

  let prev2 = 0;
  let prev1 = 1;

  for (let i = 2; i <= n; i++) {
    const current = prev1 + prev2;
    prev2 = prev1;
    prev1 = current;
```

```
  }

  return prev1;
}
```

Complexity. Time $O(n)$, space $O(1)$.

The progression from $O(2^n)$ time to $O(n)$ time with $O(1)$ space is the essence of dynamic programming.

18.5 Coin change

The **coin change problem** has two variants:

1. **Minimum coins:** Given denominations $d_1, d_2, \ldots, d_k$ and a target amount A, find the fewest coins that sum to A.
2. **Count ways:** Count the number of distinct combinations of coins that sum to A.

18.5.1 Minimum coins

Sub-problems. Let $dp[i]$ be the minimum number of coins needed to make amount i.

Recurrence.

$$dp[i] = \min_{d_j \le i} \left(dp[i - d_j] + 1 \right)$$

Base case. $dp[0] = 0$ (zero coins to make amount zero).

Computation order. Fill $dp[1], dp[2], \ldots, dp[A]$ in increasing order.

```
export function minCoinChange(
  denominations: number[],
  amount: number,
): MinCoinsResult {
  if (amount < 0) throw new RangeError('amount must be non-negative');
  if (amount === 0) return { minCoins: 0, coins: [] };

  const dp = new Array<number>(amount + 1).fill(Infinity);
  const parent = new Array<number>(amount + 1).fill(-1);

  dp[0] = 0;
```

```
  for (let i = 1; i <= amount; i++) {
    for (const coin of denominations) {
      if (coin <= i && dp[i - coin]! + 1 < dp[i]!) {
        dp[i] = dp[i - coin]! + 1;
        parent[i] = coin;
      }
    }
  }

  if (dp[amount] === Infinity) {
    return { minCoins: -1, coins: [] };
  }

  // Backtrack to recover the coins used.
  const coins: number[] = [];
  let remaining = amount;
  while (remaining > 0) {
    coins.push(parent[remaining]!);
    remaining -= parent[remaining]!;
  }

  return { minCoins: dp[amount]!, coins };
}
```

Complexity. Time $O(A \cdot k)$ where A is the amount and k is the number of denominations. Space $O(A)$.

Example. Denominations $\{1, 5, 6\}$, amount 11. A greedy approach would pick $6 + 5 = 11$ (2 coins), which happens to be optimal. For amount 10, however, greedy picks $6 + 1 + 1 + 1 + 1 = 10$ (5 coins), while the optimal is $5 + 5 = 10$ (2 coins). Dynamic programming always finds the minimum.

18.5.2 Counting the number of ways

To count the number of distinct **combinations** (not permutations) that sum to A, we iterate denominations in the outer loop to avoid counting the same combination multiple times:

```
export function countCoinChange(denominations: number[], amount: number): number {
  if (amount < 0) throw new RangeError('amount must be non-negative');

  const dp = new Array<number>(amount + 1).fill(0);
```

```
  dp[0] = 1; // one way to make 0: use no coins

  for (const coin of denominations) {
    for (let i = coin; i <= amount; i++) {
      dp[i] = dp[i]! + dp[i - coin]!;
    }
  }

  return dp[amount]!;
}
```

Complexity. Time $O(A \cdot k)$, space $O(A)$.

The key subtlety is the loop order. If we iterated amounts in the outer loop and denominations in the inner loop, we would count permutations ($1 + 2$ and $2 + 1$ as separate), not combinations.

18.6 Longest common subsequence

Given two sequences $X = \langle x_1, \ldots, x_m \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$, a **common subsequence** is a sequence that appears (in order, but not necessarily contiguously) in both X and Y. The **longest common subsequence** (LCS) problem asks for a common subsequence of maximum length.

Applications. LCS is fundamental in:

- **`diff` utilities** — computing the minimal set of changes between two files.
- **Bioinformatics** — comparing DNA, RNA, or protein sequences.
- **Version control** — finding differences between file versions.

18.6.1 The DP formulation

Sub-problems. Let $dp[i][j]$ be the length of the LCS of $X[1..i]$ and $Y[1..j]$.

Recurrence.

$$dp[i][j] = \begin{cases} dp[i-1][j-1] + 1 & \text{if } x_i = y_j \\ \max(dp[i-1][j],\ dp[i][j-1]) & \text{otherwise} \end{cases}$$

Base cases. $dp[0][j] = dp[i][0] = 0$ for all i, j.

Computation order. Fill the table row by row, left to right.

The intuition: if the last characters match, they must be part of an optimal alignment, so we include them and recurse on the remaining prefixes. If they do not match, we try dropping the last character from each sequence and take the better result.

```
export function lcs<T>(a: readonly T[], b: readonly T[]): LCSResult<T> {
  const m = a.length;
  const n = b.length;

  const dp: number[][] = Array.from({ length: m + 1 }, () =>
    new Array<number>(n + 1).fill(0),
  );

  for (let i = 1; i <= m; i++) {
    for (let j = 1; j <= n; j++) {
      if (a[i - 1] === b[j - 1]) {
        dp[i]![j] = dp[i - 1]![j - 1]! + 1;
      } else {
        dp[i]![j] = Math.max(dp[i - 1]![j]!, dp[i]![j - 1]!);
      }
    }
  }

  // Backtrack to recover the subsequence.
  const subsequence: T[] = [];
  let i = m;
  let j = n;
  while (i > 0 && j > 0) {
    if (a[i - 1] === b[j - 1]) {
      subsequence.push(a[i - 1]!);
      i--;
      j--;
    } else if (dp[i - 1]![j]! > dp[i]![j - 1]!) {
      i--;
    } else {
      j--;
    }
  }
  subsequence.reverse();

  return { length: dp[m]![n]!, subsequence };
}
```

Complexity. Time $O(mn)$, space $O(mn)$.

Example. For X = ABCBDAB and Y = BDCABA, the LCS has length 4 — one solution is BCBA.

18.6.2 Space optimization

If we only need the LCS **length** (not the actual subsequence), we can reduce space to $O(\min(m, n))$ by keeping only two rows of the table at a time: the previous row and the current row.

18.7 Edit distance

The **edit distance** (or **Levenshtein distance**) between two strings a and b is the minimum number of single-character operations needed to transform a into b. The allowed operations are:

- **Insert** a character into a.
- **Delete** a character from a.
- **Substitute** one character in a with another.

Edit distance is closely related to LCS — in fact, the edit distance between two strings of lengths m and n is $m+n-2\cdot\text{LCS}(a, b)$ when only insertions and deletions are allowed. With substitutions, the relationship is more nuanced.

Applications. Edit distance is used in spell checkers, DNA sequence alignment, natural language processing, and fuzzy string matching.

18.7.1 The DP formulation

Sub-problems. Let $dp[i][j]$ be the edit distance between $a[1..i]$ and $b[1..j]$.

Recurrence.

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

The three terms in the minimum correspond to:

- $dp[i-1][j] + 1$: delete a_i.
- $dp[i][j-1] + 1$: insert b_j.
- $dp[i-1][j-1] + 1$: substitute a_i with b_j.

Base cases. $dp[i][0] = i$ (delete all characters from a) and $dp[0][j] = j$ (insert all characters of b).

```
export function editDistance(a: string, b: string): EditDistanceResult {
  const m = a.length;
  const n = b.length;

  const dp: number[][] = Array.from({ length: m + 1 }, () =>
    new Array<number>(n + 1).fill(0),
  );

  for (let i = 0; i <= m; i++) dp[i]![0] = i;
  for (let j = 0; j <= n; j++) dp[0]![j] = j;

  for (let i = 1; i <= m; i++) {
    for (let 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 +
          Math.min(
            dp[i - 1]![j]!,      // delete
            dp[i]![j - 1]!,      // insert
            dp[i - 1]![j - 1]!,  // substitute
          );
      }
    }
  }

  // ... backtrack to recover operations ...

  return { distance: dp[m]![n]!, operations };
}
```

Complexity. Time $O(mn)$, space $O(mn)$.

Example. `kitten` → `sitting` requires 3 operations:

1. Substitute `k` → `s` (`sitten`)
2. Substitute `e` → `i` (`sittin`)
3. Insert `g` at the end (`sitting`)

18.7.2 Recovering the edit script

By backtracking through the DP table from $dp[m][n]$ to $dp[0][0]$, we can recover the actual sequence of edit operations. At each cell, we determine which operation was used (match, substitute, insert, or delete) by comparing the cell's value with its neighbors. Our implementation returns an array of `EditStep` objects, each recording the operation type and the characters involved.

18.8 0/1 Knapsack

The **0/1 knapsack problem** models a fundamental resource allocation trade-off: given n items, each with a weight w_i and a value v_i, and a knapsack of capacity W, select a subset of items that maximizes total value without exceeding the capacity.

The "0/1" qualifier means each item is either taken or left — no fractions. This distinguishes it from the **fractional knapsack** problem (Chapter 17), which has a greedy solution.

18.8.1 The DP formulation

Sub-problems. Let $dp[i][w]$ be the maximum value achievable using items $1, \ldots, i$ with capacity w.

Recurrence.

$$dp[i][w] = \begin{cases} dp[i-1][w] & \text{if } w_i > w \\ \max(dp[i-1][w],\ dp[i-1][w-w_i] + v_i) & \text{otherwise} \end{cases}$$

For each item, we choose the better of two options: skip it (value stays at $dp[i-1][w]$) or take it (add its value to the best we can do with the remaining capacity).

Base cases. $dp[0][w] = 0$ for all w (no items, no value).

```
export function knapsack(items: KnapsackItem[], capacity: number): KnapsackResult {
  if (capacity < 0) throw new RangeError('capacity must be non-negative');

  const n = items.length;

  const dp: number[][] = Array.from({ length: n + 1 }, () =>
    new Array<number>(capacity + 1).fill(0),
  );

  for (let i = 1; i <= n; i++) {
```

```
    const item = items[i - 1]!;
    for (let w = 0; w <= capacity; w++) {
      dp[i]![w] = dp[i - 1]![w]!;
      if (item.weight <= w) {
        const withItem = dp[i - 1]![w - item.weight]! + item.value;
        if (withItem > dp[i]![w]!) {
          dp[i]![w] = withItem;
        }
      }
    }
  }

  // Backtrack to find which items were selected.
  const selectedItems: number[] = [];
  let w = capacity;
  for (let i = n; i > 0; i--) {
    if (dp[i]![w] !== dp[i - 1]![w]) {
      selectedItems.push(i - 1);
      w -= items[i - 1]!.weight;
    }
  }
  selectedItems.reverse();

  return { maxValue: dp[n]![capacity]!, selectedItems, totalWeight };
}
```

Complexity. Time $O(nW)$, space $O(nW)$.

Important caveat. This is a **pseudo-polynomial** algorithm. The running time depends on the **numeric value** of W, not on the number of bits needed to represent it. If W is exponentially large in the input size, the algorithm becomes exponential. This distinction is crucial when discussing NP-completeness (Chapter 21) — the 0/1 knapsack problem is NP-hard, and the pseudo-polynomial algorithm does not contradict this.

Example. Items: $(w = 10, v = 60)$, $(w = 20, v = 100)$, $(w = 30, v = 120)$. Capacity: 50. The optimal selection is items 2 and 3 (weight 50, value 220).

18.8.2 Space optimization

Since row i depends only on row $i - 1$, we can reduce space to $O(W)$ by using a single 1D array and iterating weights in **decreasing** order (to avoid using an item twice):

```
for each item:
    for w = W down to item.weight:
        dp[w] = max(dp[w], dp[w - item.weight] + item.value)
```

However, this optimization prevents us from backtracking to recover which items were selected, since the full table is no longer available.

18.9 Matrix chain multiplication

Given a chain of n matrices $A_1, A_2, \ldots, A_n$ where matrix A_i has dimensions $p_{i-1} \times p_i$, we want to parenthesize the product $A_1 A_2 \cdots A_n$ to **minimize the total number of scalar multiplications**.

Matrix multiplication is associative, so any parenthesization yields the same result. But the cost varies dramatically. For three matrices with dimensions 10×20, 20×30, 30×40:

- $(A_1 A_2) A_3$: cost $= 10 \cdot 20 \cdot 30 + 10 \cdot 30 \cdot 40 = 6000 + 12000 = 18000$
- $A_1 (A_2 A_3)$: cost $= 20 \cdot 30 \cdot 40 + 10 \cdot 20 \cdot 40 = 24000 + 8000 = 32000$

The first parenthesization is nearly twice as fast.

18.9.1 The DP formulation

Sub-problems. Let $m[i][j]$ be the minimum number of scalar multiplications needed to compute the product $A_i A_{i+1} \cdots A_j$.

Recurrence.

$$m[i][j] = \min_{i \le k < j} \left(m[i][k] + m[k+1][j] + p_{i-1} \cdot p_k \cdot p_j \right)$$

The idea: split the chain at position k, compute the two sub-chains optimally, and add the cost of multiplying the resulting two matrices.

Base cases. $m[i][i] = 0$ (a single matrix requires no multiplication).

Computation order. Solve by increasing chain length $\ell = j - i + 1$.

```
export function matrixChainOrder(dims: number[]): MatrixChainResult {
  if (dims.length < 2) {
    throw new Error('dims must have at least 2 elements (at least one matrix)');
  }

  const n = dims.length - 1;
```

```
  const m: number[][] = Array.from({ length: n + 1 }, () =>
    new Array<number>(n + 1).fill(0),
  );
  const s: number[][] = Array.from({ length: n + 1 }, () =>
    new Array<number>(n + 1).fill(0),
  );

  for (let l = 2; l <= n; l++) {
    for (let i = 1; i <= n - l + 1; i++) {
      const j = i + l - 1;
      m[i]![j] = Infinity;

      for (let k = i; k < j; k++) {
        const cost =
          m[i]![k]! + m[k + 1]![j]! + dims[i - 1]! * dims[k]! * dims[j]!;
        if (cost < m[i]![j]!) {
          m[i]![j] = cost;
          s[i]![j] = k;
        }
      }
    }
  }

  return {
    minCost: m[1]![n]!,
    parenthesization: buildParens(s, 1, n),
    splits: s,
  };
}
```

Complexity. Time $O(n^3)$, space $O(n^2)$.

The split table s records where the optimal split occurs for each sub-chain. We use it to reconstruct the optimal parenthesization recursively:

```
function buildParens(s: number[][], i: number, j: number): string {
  if (i === j) return `A${i}`;
  return `(${buildParens(s, i, s[i]![j]!)}${buildParens(s, s[i]![j]! + 1, j)})`;
}
```

Example. The classic CLRS example with dimensions $[30, 35, 15, 5, 10, 20, 25]$ yields an optimal cost of 15,125 scalar multiplications.

18.10 Longest increasing subsequence

Given a sequence of numbers $a_1, a_2, \ldots, a_n$, the **longest increasing subsequence** (LIS) is the longest subsequence $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ such that $i_1 < i_2 < \cdots < i_k$ and $a_{i_1} < a_{i_2} < \cdots < a_{i_k}$.

Applications. LIS appears in patience sorting, version tracking, and computational geometry (longest chain of points dominated by each other).

18.10.1 O(n²) dynamic programming

Sub-problems. Let $dp[i]$ be the length of the longest increasing subsequence ending at position i.

Recurrence.

$$dp[i] = 1 + \max_{j<i,\ a_j<a_i} dp[j]$$

(If no such j exists, $dp[i] = 1$.)

Base cases. $dp[i] = 1$ for all i (each element is an increasing subsequence of length 1 by itself).

```
export function lisDP(arr: readonly number[]): LISResult {
  const n = arr.length;
  if (n === 0) return { length: 0, subsequence: [] };

  const dp = new Array<number>(n).fill(1);
  const parent = new Array<number>(n).fill(-1);

  for (let i = 1; i < n; i++) {
    for (let j = 0; j < i; j++) {
      if (arr[j]! < arr[i]! && dp[j]! + 1 > dp[i]!) {
        dp[i] = dp[j]! + 1;
        parent[i] = j;
      }
    }
  }

  // Find the index where the LIS ends.
  let bestLen = 0;
  let bestIdx = 0;
  for (let i = 0; i < n; i++) {
```

```
    if (dp[i]! > bestLen) {
      bestLen = dp[i]!;
      bestIdx = i;
    }
  }

  // Backtrack to recover the subsequence.
  const subsequence: number[] = [];
  let idx = bestIdx;
  while (idx !== -1) {
    subsequence.push(arr[idx]!);
    idx = parent[idx]!;
  }
  subsequence.reverse();

  return { length: bestLen, subsequence };
}
```

Complexity. Time $O(n^2)$, space $O(n)$.

18.10.2 O(n log n) patience sorting

We can improve to $O(n \log n)$ using a technique inspired by the card game Patience. Maintain an array `tails` where `tails[i]` is the smallest tail element of all increasing subsequences of length $i + 1$ found so far.

For each element in the input:

- **Binary search** for the leftmost position in `tails` where `tails[pos] >= val`.
- Replace `tails[pos]` with `val` (or extend `tails` if `val` is larger than all current tails).

The key invariant is that `tails` is always sorted, which is what makes binary search possible.

```
export function lisBinarySearch(arr: readonly number[]): LISResult {
  const n = arr.length;
  if (n === 0) return { length: 0, subsequence: [] };

  const tails: number[] = [];
  const tailIndices: number[] = [];
  const parent = new Array<number>(n).fill(-1);

  for (let i = 0; i < n; i++) {
```

```
    const val = arr[i]!;

    let lo = 0;
    let hi = tails.length;
    while (lo < hi) {
      const mid = (lo + hi) >>> 1;
      if (tails[mid]! < val) {
        lo = mid + 1;
      } else {
        hi = mid;
      }
    }

    tails[lo] = val;
    tailIndices[lo] = i;

    if (lo > 0) {
      parent[i] = tailIndices[lo - 1]!;
    }
  }

  // Backtrack to recover the subsequence.
  const length = tails.length;
  const subsequence: number[] = [];
  let idx = tailIndices[length - 1]!;
  for (let k = 0; k < length; k++) {
    subsequence.push(arr[idx]!);
    idx = parent[idx]!;
  }
  subsequence.reverse();

  return { length, subsequence };
}
```

Complexity. Time $O(n \log n)$, space $O(n)$.

Example. For the sequence $[0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15]$, the LIS has length 6. One such subsequence is $[0, 2, 6, 9, 11, 15]$.

18.11 Summary of DP problems

Problem	Sub-problem space	Recurrence	Time	Space
Fibonacci	$F(n)$	$F(n) = F(n-1) + F(n-2)$	$O(n)$	$O(1)$
Min coin change	$dp[i]$: min coins for amount i	$dp[i] = 1+\min dp[i-d_j]$	$O(Ak)$	$O(A)$
LCS	$dp[i][j]$: LCS of prefixes	match or skip	$O(mn)$	$O(mn)$
Edit distance	$dp[i][j]$: edit dist of prefixes	match, sub, ins, del	$O(mn)$	$O(mn)$
0/1 Knapsack	$dp[i][w]$: best value, items $1..i$, cap w	take or skip item i	$O(nW)$	$O(nW)$
Matrix chain	$m[i][j]$: min cost for $A_i \cdots A_j$	split at k	$O(n^3)$	$O(n^2)$
LIS	$dp[i]$: LIS ending at i	extend from $j < i$	$O(n \log n)$	$O(n)$

18.12 Exercises

1. **Rod cutting.** Given a rod of length n and a price table $p[1..n]$ where $p[i]$ is the price of a rod of length i, find the maximum revenue obtainable by cutting the rod into pieces. Write the recurrence, implement both top-down and bottom-up solutions, and analyze their complexity.

2. **Subset sum.** Given a set of positive integers S and a target T, determine whether there exists a subset of S that sums to T. Define the subproblems, write the recurrence, and implement a tabulated solution. What is the relationship between this problem and 0/1 knapsack?

3. **Counting LCS.** Modify the LCS algorithm to count the **number of distinct** longest common subsequences (not just find one). What changes are needed in the recurrence and the table?

4. **Weighted edit distance.** Generalize the edit distance algorithm so that insertions, deletions, and substitutions can have different costs (not all equal to 1). For example, in DNA alignment, a substitution between similar nucleotides might cost less than one between dissimilar nucleotides. Implement this generalization and verify it on a test case.

5. **LIS and LCS connection.** Prove that the LIS problem can be reduced to LCS by computing the LCS of the original sequence and its sorted version. Is this reduction efficient? When would you prefer the $O(n \log n)$ patience-sorting approach over the LCS-based approach?

18.13 Chapter summary

Dynamic programming transforms problems with exponential brute-force solutions into efficient polynomial-time algorithms by exploiting **optimal substructure** and **overlapping subproblems**. The key insight is simple: do not recompute — remember. Whether through top-down memoization or bottom-up tabulation, DP systematically stores solutions to subproblems and builds toward the final answer.

We saw this principle in action across seven problems: from the elementary Fibonacci sequence (which illustrates the core idea) to sophisticated optimization problems like matrix chain multiplication and the knapsack problem. Each problem followed the same five-step recipe: define subproblems, write the recurrence, identify base cases, determine computation order, and recover the solution.

In the next chapter, we turn to **greedy algorithms** — a complementary design paradigm that, when applicable, yields even simpler and more efficient solutions than DP. The key challenge with greedy algorithms is proving that the locally optimal choice at each step leads to a globally optimal solution — a property that holds for some problems but not others. Understanding when to use DP and when to use greedy is one of the most important skills in algorithm design.

Chapter 19

Greedy Algorithms

Dynamic programming (Chapter 16) achieves optimal solutions by methodically exploring all subproblems and combining their answers. Greedy algorithms take a more aggressive approach: at each step they make the locally optimal choice and never look back. When a greedy strategy works, the result is typically a simpler and faster algorithm — often just a single pass over sorted data. The catch is that the locally optimal choice does not always lead to a globally optimal solution, so correctness requires proof. In this chapter we develop two proof techniques — the "greedy stays ahead" argument and the exchange argument — and apply them to three classic problems: interval scheduling, Huffman coding, and fractional knapsack.

19.1 The greedy strategy

A greedy algorithm builds a solution incrementally. At each step it examines the available candidates, selects the one that looks best according to some criterion, and commits to that choice irrevocably. It never reconsiders past decisions or explores alternative combinations.

Contrast this with dynamic programming:

	Dynamic programming	Greedy
Decisions	Deferred — explores all combinations via table	Immediate — commits at each step
Subproblems	Many, overlapping	Typically none (single pass)
Correctness	Optimal substructure + overlapping subproblems	Requires a specific proof (exchange or stays-ahead)

	Dynamic programming	Greedy
Efficiency	Often $O(n^2)$ or $O(n^3)$	Often $O(n \log n)$ or $O(n)$

The greedy strategy works when a problem has:

1. **Optimal substructure.** An optimal solution contains optimal solutions to subproblems.
2. **The greedy-choice property.** A locally optimal choice can always be extended to a globally optimal solution. In other words, we never need to reconsider a greedy choice.

Property 1 is shared with DP. Property 2 is what distinguishes greedy problems: it asserts that committing to the local optimum is safe.

19.2 Proving greedy algorithms correct

Because the greedy-choice property is not obvious, we need rigorous proofs. Two standard techniques are widely used.

19.2.1 Greedy stays ahead

Idea. Show that after each step, the greedy solution is at least as good as any other solution at the same step. If the greedy algorithm stays ahead (or tied) at every step, it must be at least as good as the optimum overall.

Structure of the proof:

1. Define a measure of progress after k steps.
2. Prove by induction that the greedy solution's measure is at least as good as the optimal solution's measure after every step k.
3. Conclude that the final greedy solution is optimal.

We will use this technique for interval scheduling below.

19.2.2 Exchange argument

Idea. Start with an arbitrary optimal solution. Show that it can be transformed — step by step, by "exchanging" its choices for greedy choices — into the greedy solution without worsening the objective. If an optimal solution can always be transformed into the greedy solution, the greedy solution must be optimal.

Structure of the proof:

1. Consider an optimal solution O that differs from the greedy solution G.

2. Identify the first point where O and G differ.
3. Show that modifying O to agree with G at that point does not make O worse.
4. Repeat until $O = G$.

We will use this technique for Huffman coding.

19.3 Interval scheduling (activity selection)

19.3.1 Problem definition

Given n activities, each with a start time s_i and a finish time f_i (where $s_i < f_i$), select the largest subset of mutually compatible activities. Two activities are **compatible** if they do not overlap — that is, one finishes before the other starts.

This problem arises in resource allocation: scheduling the maximum number of non-overlapping jobs on a single machine, booking meeting rooms, or allocating time slots.

19.3.2 Greedy approach

The key insight is to **sort activities by finish time** and greedily select each activity whose start time does not conflict with the previously selected activity.

Why finish time? Consider the alternatives:

- **Sort by start time.** A long early activity could block many shorter ones.
- **Sort by duration.** A short activity in the middle could block two non-overlapping ones.
- **Sort by fewest conflicts.** Counterexamples exist.
- **Sort by finish time.** By always choosing the activity that finishes earliest, we leave as much room as possible for future activities.

19.3.3 Algorithm

1. Sort activities by finish time.
2. Select the first activity.
3. For each subsequent activity: if its start time is ≥ the finish time of the last selected activity, select it.

```
export interface Interval {
  start: number;
  end: number;
}

export interface IntervalSchedulingResult {
```

```
  selected: Interval[];
  count: number;
}

export function intervalScheduling(
  intervals: readonly Interval[],
): IntervalSchedulingResult {
  if (intervals.length === 0) {
    return { selected: [], count: 0 };
  }

  // Sort by finish time (break ties by start time).
  const sorted = intervals.slice().sort((a, b) => {
    if (a.end !== b.end) return a.end - b.end;
    return a.start - b.start;
  });

  const selected: Interval[] = [sorted[0]!];
  let lastEnd = sorted[0]!.end;

  for (let i = 1; i < sorted.length; i++) {
    const interval = sorted[i]!;
    if (interval.start >= lastEnd) {
      selected.push(interval);
      lastEnd = interval.end;
    }
  }

  return { selected, count: selected.length };
}
```

19.3.4 Correctness proof (greedy stays ahead)

Let $G = \{g_1, g_2, \ldots, g_k\}$ be the activities selected by the greedy algorithm (in order of finish time), and let $O = \{o_1, o_2, \ldots, o_m\}$ be an optimal solution (also sorted by finish time). We want to show $k \geq m$.

Lemma (greedy stays ahead). For all $i \leq k$, we have $f(g_i) \leq f(o_i)$ — the i-th greedy activity finishes no later than the i-th optimal activity.

Proof by induction on i:

- **Base case** ($i = 1$). The greedy algorithm picks the activity with the earliest

finish time, so $f(g_1) \leq f(o_1)$.

- **Inductive step.** Assume $f(g_i) \leq f(o_i)$. Since o_{i+1} starts after o_i finishes, we have $s(o_{i+1}) \geq f(o_i) \geq f(g_i)$. Therefore o_{i+1} is compatible with g_i, and the greedy algorithm considers it (or an activity that finishes even earlier). It follows that $f(g_{i+1}) \leq f(o_{i+1})$.

Theorem. $k \geq m$. If $k < m$, then by the lemma, $f(g_k) \leq f(o_k) \leq s(o_{k+1})$, so o_{k+1} is compatible with g_k and the greedy algorithm would have selected it — contradicting the fact that greedy stopped at k activities. Therefore $k = m$, and the greedy solution is optimal. □

19.3.5 Complexity

- **Time:** $O(n \log n)$ for sorting, plus $O(n)$ for the single scan. Total: $O(n \log n)$.
- **Space:** $O(n)$ for the sorted copy and result.

19.3.6 Example

Consider these activities sorted by finish time:

Activity	Start	Finish
A	1	4
B	3	5
C	0	6
D	5	7
E	3	9
F	6	10
G	8	11

The greedy algorithm proceeds:

1. Select **A** [1, 4). Last finish = 4.
2. **B** starts at 3 < 4 — skip.
3. **C** starts at 0 < 4 — skip.
4. **D** starts at 5 ≥ 4 — select. Last finish = 7.
5. **E** starts at 3 < 7 — skip.
6. **F** starts at 6 < 7 — skip.
7. **G** starts at 8 ≥ 7 — select. Last finish = 11.

Result: {A, D, G} — 3 activities. This is optimal.

19.4 Huffman coding

19.4.1 Problem definition

Given an alphabet C of n characters, each with a known frequency $f(c)$, find a **prefix-free binary code** that minimizes the total encoding length:

$$B(T) = \sum_{c \in C} f(c) \cdot d_T(c)$$

where $d_T(c)$ is the depth of character c in the coding tree T (which equals the length of its binary codeword).

A code is **prefix-free** if no codeword is a prefix of another. This guarantees that encoded text can be decoded unambiguously without delimiters.

19.4.2 Why variable-length codes?

Fixed-length codes (like ASCII) use $\lceil \log_2 n \rceil$ bits per character regardless of frequency. If some characters appear much more often than others, variable-length codes can do better: assign shorter codewords to frequent characters and longer ones to rare characters. This is the principle behind data compression formats like ZIP, gzip, and JPEG.

19.4.3 Huffman's greedy algorithm

David Huffman (1952) discovered that the optimal prefix-free code can be built by a simple greedy procedure:

1. Create a leaf node for each character, with its frequency as the key.
2. Insert all leaves into a min-priority queue.
3. While the queue has more than one node:
   a. Extract the two nodes x and y with the lowest frequencies.
   b. Create a new internal node z with $z.freq = x.freq + y.freq$, left child x, and right child y.
   c. Insert z back into the queue.
4. The remaining node is the root of the Huffman tree.
5. Assign code `0` to left edges and `1` to right edges. Each character's codeword is the sequence of bits on the path from root to its leaf.

```
import { BinaryHeap } from '../11-heaps-and-priority-queues/binary-heap.js';

export type HuffmanNode = HuffmanLeaf | HuffmanInternal;
```

```ts
export interface HuffmanLeaf {
  kind: 'leaf';
  char: string;
  freq: number;
}

export interface HuffmanInternal {
  kind: 'internal';
  freq: number;
  left: HuffmanNode;
  right: HuffmanNode;
}

export function buildHuffmanTree(
  frequencies: ReadonlyMap<string, number>,
): HuffmanNode {
  if (frequencies.size === 0) {
    throw new RangeError('frequency map must not be empty');
  }

  // Special case: single character.
  if (frequencies.size === 1) {
    const [char, freq] = [...frequencies][0]!;
    return { kind: 'leaf', char, freq };
  }

  const heap = new BinaryHeap<HuffmanNode>((a, b) => a.freq - b.freq);
  for (const [char, freq] of frequencies) {
    heap.insert({ kind: 'leaf', char, freq });
  }

  while (heap.size > 1) {
    const left = heap.extract()!;
    const right = heap.extract()!;
    const merged: HuffmanInternal = {
      kind: 'internal',
      freq: left.freq + right.freq,
      left,
      right,
    };
    heap.insert(merged);
```

```
  }

  return heap.extract()!;
}
```

The code table is then extracted by a simple tree traversal:

```
export function buildCodeTable(root: HuffmanNode): Map<string, string> {
  const table = new Map<string, string>();

  if (root.kind === 'leaf') {
    table.set(root.char, '0');
    return table;
  }

  function walk(node: HuffmanNode, prefix: string): void {
    if (node.kind === 'leaf') {
      table.set(node.char, prefix);
      return;
    }
    walk(node.left, prefix + '0');
    walk(node.right, prefix + '1');
  }

  walk(root, '');
  return table;
}
```

19.4.4 Encoding and decoding

Encoding replaces each character with its codeword:

```
export function huffmanEncode(text: string): HuffmanEncodingResult {
  if (text.length === 0) {
    throw new RangeError('text must be non-empty');
  }

  const frequencies = new Map<string, number>();
  for (const ch of text) {
    frequencies.set(ch, (frequencies.get(ch) ?? 0) + 1);
  }

  const tree = buildHuffmanTree(frequencies);
```

```
  const codeTable = buildCodeTable(tree);

  let encoded = '';
  for (const ch of text) {
    encoded += codeTable.get(ch)!;
  }

  return { encoded, codeTable, tree };
}
```

Decoding walks the tree from root to leaf for each bit:

```
export function huffmanDecode(
  encoded: string,
  tree: HuffmanNode,
): string {
  if (tree.kind === 'leaf') {
    return tree.char.repeat(encoded.length);
  }

  let result = '';
  let node: HuffmanNode = tree;

  for (const bit of encoded) {
    node = bit === '0'
      ? (node as HuffmanInternal).left
      : (node as HuffmanInternal).right;

    if (node.kind === 'leaf') {
      result += node.char;
      node = tree;
    }
  }

  return result;
}
```

19.4.5 Correctness proof (exchange argument)

We prove that the Huffman algorithm produces an optimal prefix-free code.

Lemma 1. There exists an optimal tree in which the two lowest-frequency characters are siblings at the maximum depth.

Proof. Let T^* be an optimal tree. Let x and y be the two characters with the lowest frequencies. If they are not at the maximum depth or not siblings in T^*, we can swap them with the characters at maximum depth without increasing the cost (because x and y have the lowest frequencies, moving them deeper cannot increase $B(T)$, and moving more frequent characters to shallower positions can only help). □

Lemma 2. Let T' be the tree obtained by replacing the subtree containing siblings x and y with a single leaf z having frequency $f(z) = f(x) + f(y)$. Then $B(T) = B(T') + f(x) + f(y)$.

Proof. In T, x and y are one level deeper than z is in T'. Each contributes $f(c) \cdot 1$ extra to $B(T)$ compared to $B(T')$. □

Theorem. The Huffman algorithm produces an optimal prefix-free code.

Proof by induction on the number of characters n:

- **Base case** ($n = 1$ or $n = 2$). Trivially optimal.
- **Inductive step.** By Lemma 1, there is an optimal tree where the two lowest-frequency characters x, y are siblings at maximum depth. By Lemma 2, replacing them with a merged node z gives a subproblem with $n-1$ characters. By the inductive hypothesis, Huffman solves the subproblem optimally. Since the merge doesn't affect the relative costs of the remaining characters, the full tree is also optimal. □

19.4.6 Complexity

- **Time:** $O(n \log n)$ where n is the number of distinct characters. Each of the $n - 1$ merge steps involves two heap extractions and one insertion, each $O(\log n)$.
- **Space:** $O(n)$ for the tree and heap.
- **Encoding time:** $O(m)$ where m is the length of the input text (after the tree is built).
- **Decoding time:** $O(b)$ where b is the number of bits in the encoded string.

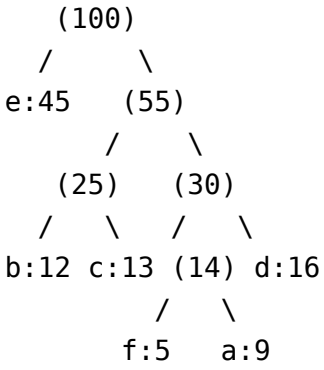
19.4.7 Example

Consider an alphabet with these frequencies:

Character	f	a	b	c	d	e
Frequency	5	9	12	13	16	45

Step-by-step tree construction:

1. Extract `f` (5) and `a` (9) → merge into node (14).
2. Extract `b` (12) and `c` (13) → merge into node (25).
3. Extract (14) and `d` (16) → merge into node (30).
4. Extract (25) and (30) → merge into node (55).
5. Extract `e` (45) and (55) → merge into root (100).



Resulting codes:

Character	Code	Length
e	0	1
b	100	3
c	101	3
f	1100	4
a	1101	4
d	111	3

Total encoding length: $45 \times 1 + 12 \times 3 + 13 \times 3 + 5 \times 4 + 9 \times 4 + 16 \times 3 = 45 + 36 + 39 + 20 + 36 + 48 = 224$ bits.

A fixed-length code would require $\lceil \log_2 6 \rceil = 3$ bits per character, for a total of $100 \times 3 = 300$ bits. Huffman coding saves $300 - 224 = 76$ bits, a 25% reduction.

19.5 Fractional knapsack

19.5.1 Problem definition

Given n items, each with a weight $w_i > 0$ and a value $v_i \geq 0$, and a knapsack with capacity W, maximize the total value of items placed in the knapsack. Unlike the 0/1 knapsack (Chapter 16), here we may take **fractions** of items: for each item i, we choose a fraction $x_i \in [0, 1]$, subject to:

$$\sum_{i=1}^{n} x_i \cdot w_i \leq W$$

Maximize:

$$\sum_{i=1}^{n} x_i \cdot v_i$$

19.5.2 Why greedy works here (but not for 0/1 knapsack)

The fractional knapsack has the greedy-choice property: we should always take as much as possible of the item with the highest value-per-unit-weight ratio v_i/w_i.

For the **0/1 knapsack**, this greedy strategy fails. Consider:

- Item A: weight 10, value 60 (ratio 6)
- Item B: weight 20, value 100 (ratio 5)
- Capacity: 20

Greedy by ratio selects item A (ratio 6), getting value 60. But the optimal solution takes item B for value 100. The constraint that items cannot be split breaks the greedy-choice property, which is why the 0/1 knapsack requires dynamic programming.

In the fractional case, if item A doesn't fill the knapsack, we can take part of item B as well — the "fractional freedom" ensures the greedy choice is always safe.

19.5.3 Algorithm

1. Compute the value-to-weight ratio for each item.
2. Sort items by ratio in descending order.
3. Greedily take as much of each item as possible until the knapsack is full.

```
export interface FractionalKnapsackItem {
  weight: number;
  value: number;
}

export interface PackedItem {
  index: number;
  fraction: number;
  weight: number;
  value: number;
}

export interface FractionalKnapsackResult {
  maxValue: number;
```

```
  totalWeight: number;
  packedItems: PackedItem[];
}

export function fractionalKnapsack(
  items: readonly FractionalKnapsackItem[],
  capacity: number,
): FractionalKnapsackResult {
  if (capacity < 0) {
    throw new RangeError('capacity must be non-negative');
  }

  const indexed = items.map((item, i) => ({
    index: i,
    weight: item.weight,
    value: item.value,
    ratio: item.value / item.weight,
  }));
  indexed.sort((a, b) => b.ratio - a.ratio);

  const packedItems: PackedItem[] = [];
  let remaining = capacity;
  let totalValue = 0;
  let totalWeight = 0;

  for (const item of indexed) {
    if (remaining <= 0) break;

    if (item.weight <= remaining) {
      packedItems.push({
        index: item.index,
        fraction: 1,
        weight: item.weight,
        value: item.value,
      });
      remaining -= item.weight;
      totalValue += item.value;
      totalWeight += item.weight;
    } else {
      const fraction = remaining / item.weight;
      const fractionalValue = item.value * fraction;
```

```
      packedItems.push({
        index: item.index,
        fraction,
        weight: remaining,
        value: fractionalValue,
      });
      totalValue += fractionalValue;
      totalWeight += remaining;
      remaining = 0;
    }
  }

  return { maxValue: totalValue, totalWeight, packedItems };
}
```

19.5.4 Correctness proof (exchange argument)

Theorem. Sorting by v_i/w_i and greedily packing yields an optimal solution.

Proof. Suppose items are sorted so that $v_1/w_1 \geq v_2/w_2 \geq \cdots \geq v_n/w_n$. Let $G = (x_1, \ldots, x_n)$ be the greedy solution and $O = (y_1, \ldots, y_n)$ be an optimal solution. If $G \neq O$, let j be the first index where they differ. By the greedy algorithm, x_j is as large as possible (either 1 or filling the remaining capacity), so $x_j > y_j$.

We can increase y_j and decrease some y_k (where $k > j$, so $v_k/w_k \leq v_j/w_j$) to compensate. Specifically, shift weight δ from item k to item j:

$$\Delta = \delta \cdot (v_j/w_j - v_k/w_k) \geq 0$$

The objective value does not decrease. Repeating this exchange process transforms O into G without ever decreasing the total value, so G is optimal. □

19.5.5 Complexity

- **Time:** $O(n \log n)$ for sorting, plus $O(n)$ for the greedy scan. Total: $O(n \log n)$.
- **Space:** $O(n)$.

19.5.6 Example

Item	Weight	Value	Ratio
A	10	60	6.0
B	20	100	5.0
C	30	120	4.0

Capacity: 50

Greedy packing (sorted by ratio):

1. Take all of A: weight 10, value 60. Remaining capacity: 40.
2. Take all of B: weight 20, value 100. Remaining capacity: 20.
3. Take 20/30 of C: weight 20, value $120 \times \frac{20}{30} = 80$. Remaining capacity: 0.

Total value: $60 + 100 + 80 = 240$.

Compare with the 0/1 knapsack (no fractions allowed), where the optimal is to take A and C for value $60 + 120 = 180$, or A and B for value $60 + 100 = 160$. The ability to take fractions yields a strictly higher value.

19.6 When greedy fails

Not every optimization problem admits a greedy solution. Here are instructive examples where the greedy approach fails:

1. **0/1 Knapsack.** As shown above, the greedy-by-ratio strategy is suboptimal. The integer constraint destroys the greedy-choice property.
2. **Longest path in a graph.** Greedily choosing the longest edge at each step does not yield the longest path. This problem is NP-hard.
3. **Optimal BST.** Greedily placing the most frequent key at the root does not minimize expected search time. This requires DP (similar to matrix chain multiplication).

The lesson: always prove that the greedy-choice property holds before trusting a greedy algorithm. The proofs in this chapter — "greedy stays ahead" and the exchange argument — are the standard tools for doing so.

19.7 Comparison of algorithms in this chapter

Problem	Strategy	Time	Space	Proof technique
Interval scheduling	Sort by finish time	$O(n \log n)$	$O(n)$	Greedy stays ahead

Problem	Strategy	Time	Space	Proof technique
Huffman coding	Merge lowest-frequency pairs	$O(n \log n)$	$O(n)$	Exchange argument
Fractional knapsack	Sort by value/weight ratio	$O(n \log n)$	$O(n)$	Exchange argument

19.8 Exercises

1. **Weighted interval scheduling.** In the weighted variant, each activity has a value v_i, and the goal is to maximize the total value (not the count) of selected non-overlapping activities. Show that the greedy algorithm (sort by finish time) does not solve this problem optimally. Design a dynamic programming algorithm in $O(n \log n)$ time.

2. **Job scheduling with deadlines.** You have n jobs, each taking unit time, with a deadline d_i and a penalty p_i incurred if the job is not completed by its deadline. Design a greedy algorithm that minimizes the total penalty. Prove its correctness.

3. **Optimal merge pattern.** You have n sorted files of sizes $s_1, s_2, \ldots, s_n$. Merging two files of sizes a and b costs $a + b$. Find the merge order that minimizes the total cost. How does this relate to Huffman coding?

4. **Huffman vs fixed-width.** Prove that Huffman coding never uses more bits than a fixed-width encoding. Under what conditions does it use the same number of bits?

5. **Greedy failure.** Consider the coin-change problem with denominations $\{1, 3, 4\}$ and target amount 6. Show that the greedy algorithm (always use the largest denomination that fits) gives a suboptimal solution. What is the optimal solution?

19.9 Chapter summary

Greedy algorithms solve optimization problems by making locally optimal choices at each step. They are simpler and typically faster than dynamic programming — often requiring just a sort followed by a linear scan — but they require careful proof that the greedy-choice property holds.

We studied two proof techniques. The **greedy stays ahead** argument shows that

the greedy solution maintains an advantage over any optimal solution at every step, and we applied it to interval scheduling. The **exchange argument** shows that any optimal solution can be transformed into the greedy solution without loss, and we applied it to Huffman coding and fractional knapsack.

The three problems in this chapter illustrate the range of greedy applications:

- **Interval scheduling** selects the maximum number of non-overlapping activities by always choosing the one that finishes earliest — a $O(n \log n)$ algorithm.
- **Huffman coding** produces optimal prefix-free binary codes by repeatedly merging the two lowest-frequency symbols — also $O(n \log n)$.
- **Fractional knapsack** maximizes value by greedily packing items in order of value-to-weight ratio — $O(n \log n)$.

We also contrasted greedy with DP on the knapsack problem: the fractional variant yields to greedy, while the 0/1 variant requires dynamic programming. Recognizing which problems have the greedy-choice property — and which do not — is a fundamental skill in algorithm design.

Chapter 20

Disjoint Sets

*In Chapter 14 we introduced the **Union-Find** data structure as a tool for Kruskal's minimum spanning tree algorithm. We showed the code and stated that, with path compression and union by rank, each operation runs in amortized near-constant time. In this chapter we give the data structure the thorough treatment it deserves: we motivate the problem, build up from naive solutions, add the two key optimizations one at a time, explain why the combined structure achieves its remarkable $O(\alpha(n))$ amortized bound, and survey the wide range of problems where Union-Find is the right tool.*

20.1 The disjoint-set problem

Many algorithms need to maintain a collection of **disjoint sets** — a partition of elements into non-overlapping groups — and answer questions about which group an element belongs to. The **disjoint-set** (or **union-find**) abstract data type supports three operations:

- **makeSet(x)** — create a new set containing only x.
- **find(x)** — return the **representative** (canonical element) of the set containing x. Two elements are in the same set if and only if `find` returns the same representative.
- **union(x, y)** — merge the set containing x and the set containing y into a single set.

A sequence of n `makeSet` operations followed by m `find` and `union` operations is called an **intermixed sequence** of length $n + m$. Our goal is a data structure that processes the entire sequence as quickly as possible.

20.1.1 Where disjoint sets arise

The disjoint-set problem appears in a surprising number of settings:

- **Kruskal's MST algorithm** (Chapter 14): determine whether adding an edge creates a cycle by checking if two vertices are already in the same component, and merge components when an edge is added.
- **Dynamic connectivity**: given a stream of edge insertions in an undirected graph, answer "Are vertices u and v connected?" after each insertion.
- **Image processing**: in connected-component labeling, pixels are grouped into regions by unioning adjacent pixels that satisfy a similarity criterion.
- **Equivalence classes**: in compilers, type unification during type inference is modeled as a union-find problem.
- **Percolation**: in physics simulations, determining whether a path exists from top to bottom of a grid is equivalent to checking whether top-row and bottom-row elements share a component.
- **Least common ancestors** (offline): Tarjan's offline LCA algorithm uses union-find to batch-process ancestor queries on a tree.
- **Network redundancy**: determining the number of connected components in a network, or detecting when a network becomes fully connected.

20.2 Naive implementations

Before introducing the optimized structure, let us consider two naive approaches. Each is fast for one operation but slow for the other, and understanding their limitations motivates the optimizations.

20.2.1 Array-based (quick-find)

Store an array `id[]` where `id[x]` is the representative of x's set. Two elements are in the same set if and only if they have the same `id` value.

- **find(x)** — return `id[x]`. This is $O(1)$.
- **union(x, y)** — scan the entire array, changing every entry equal to `id[x]` to `id[y]`. This is $O(n)$.

A sequence of $n-1$ union operations (enough to merge n singletons into one set) costs $O(n^2)$ time. For large n, this is too slow.

20.2.2 Linked-list-based (quick-union, unoptimized)

Represent each set as a rooted tree using a `parent[]` array. The representative of a set is the root of its tree: the element r with `parent[r] = r`.

- **find(x)** — follow parent pointers from x to the root. Time is $O(d)$, where d is the depth of x.
- **union(x, y)** — set `parent[find(x)] = find(y)`. This is $O(d_x + d_y)$.

The problem is that trees can become arbitrarily deep. If we perform $n-1$ unions in an unlucky order — always attaching the larger tree beneath the smaller one's root — the tree degenerates into a chain of length n, and `find` costs $O(n)$. A sequence of m `find` operations then costs $O(mn)$.

We need two ideas to fix this: **union by rank** to keep trees shallow, and **path compression** to flatten them over time.

20.3 Union by rank

The first optimization controls tree height by always attaching the **shorter** tree beneath the **taller** one during a union.

Each node x has a **rank** — an upper bound on the height of the subtree rooted at x. Initially, every node has rank 0 (it is a leaf). When we merge two trees:

- If the roots have different ranks, we attach the lower-rank root beneath the higher-rank root. The rank of the new root does not change.
- If the roots have equal rank r, we attach one beneath the other and increment the new root's rank to $r + 1$.

```
union(x, y):
    rootX = find(x)
    rootY = find(y)
    if rootX == rootY: return       // already same set
    if rank[rootX] < rank[rootY]:
        parent[rootX] = rootY
    else if rank[rootX] > rank[rootY]:
        parent[rootY] = rootX
    else:
        parent[rootY] = rootX
        rank[rootX] = rank[rootX] + 1
```

20.3.1 Why union by rank helps

Lemma. With union by rank (and no path compression), a tree with root of rank r contains at least 2^r nodes.

Proof. By induction on the number of union operations. Initially, every node has rank 0 and its tree has $2^0 = 1$ node. The rank of a root increases from r to $r + 1$ only when two trees of rank r are merged. By the inductive hypothesis, each

contains at least 2^r nodes, so the merged tree contains at least $2 \cdot 2^r = 2^{r+1}$ nodes. □

Corollary. The maximum rank of any node is $\lfloor \log_2 n \rfloor$, where n is the total number of elements.

This means that `find(x)` follows at most $\lfloor \log_2 n \rfloor$ parent pointers, so each `find` costs $O(\log n)$. A sequence of m operations costs $O(m \log n)$ — already a major improvement over the naive $O(mn)$.

20.4 Path compression

The second optimization speeds up `find` by making every node on the find path point directly to the root:

```
find(x):
    root = x
    while parent[root] != root:
        root = parent[root]
    // Path compression: point every node on path directly to root
    while x != root:
        next = parent[x]
        parent[x] = root
        x = next
    return root
```

After `find(x)` completes, every node that was between x and the root now has the root as its immediate parent. Future `find` operations on any of these nodes will complete in a single step.

Path compression alone (without union by rank) already achieves $O(\log n)$ amortized time per operation. But the real power comes from combining both optimizations.

20.4.1 A variant: path halving

An alternative to full path compression is **path halving**, where each node on the find path is made to skip its parent and point to its grandparent:

```
find(x):
    while parent[x] != x:
        parent[x] = parent[parent[x]]   // skip to grandparent
        x = parent[x]
    return x
```

Path halving achieves the same asymptotic amortized bound as full path compression and requires only a single pass through the path (no second loop). In practice, both variants perform similarly.

20.5 Combined complexity: the inverse Ackermann function

With both path compression and union by rank, any sequence of m operations on n elements runs in $O(m \cdot \alpha(n))$ time, where α is the **inverse Ackermann function**. This remarkable result was proved by Tarjan in 1975 and later tightened by Tarjan and van Leeuwen.

20.5.1 What is the Ackermann function?

The **Ackermann function** $A(k, j)$ is defined recursively:

$$A(0, j) = j + 1$$

$$A(k, 0) = A(k-1, 1) \quad \text{for } k \geq 1$$

$$A(k, j) = A(k-1, A(k, j-1)) \quad \text{for } k \geq 1, j \geq 1$$

This function grows extraordinarily fast. A few values:

k	$A(k, 1)$
0	2
1	3
2	5
3	13
4	65533
5	$2^{2^{2^{\cdots}}} - 3$ (a tower of 65536 twos)

The value $A(5, 1)$ is so large that it dwarfs the number of atoms in the observable universe ($\approx 10^{80}$).

20.5.2 The inverse Ackermann function

The **inverse Ackermann function** $\alpha(n)$ is defined as:

$$\alpha(n) = \min\{k : A(k, 1) \geq n\}$$

Since A grows so fast, α grows inconceivably slowly:

- $\alpha(n) = 0$ for $n \leq 2$
- $\alpha(n) = 1$ for $n = 3$
- $\alpha(n) = 2$ for $4 \leq n \leq 5$
- $\alpha(n) = 3$ for $6 \leq n \leq 13$
- $\alpha(n) = 4$ for $14 \leq n \leq 65533$
- $\alpha(n) = 5$ for $65534 \leq n \leq A(5, 1)$

For any value of n that could arise in practice — or indeed in any computation on physical hardware — $\alpha(n) \leq 4$. This is why we say union-find operations run in "effectively constant" amortized time.

20.5.3 Intuition for the amortized bound

The formal proof uses a sophisticated potential function argument originally due to Tarjan. Here is the intuition:

1. **Union by rank** ensures that tree heights are at most $O(\log n)$, so the "starting point" for find costs is logarithmic.
2. **Path compression** does not change ranks, so the rank-based height bound still holds as a worst case. However, after a find operation, the compressed nodes have much shorter paths to the root.
3. The key insight is that path compression "pays for itself." A find that traverses a long path is expensive, but it compresses that path, making all subsequent finds along it cheap. The total cost of m finds, amortized, is only $O(m \cdot \alpha(n))$.

To formalize this, Tarjan defines a potential function based on how much "room" each node has for future compression. Each expensive find reduces the potential significantly, ensuring that the amortized cost per operation is bounded by $\alpha(n)$.

20.5.4 Is this optimal?

Yes. Tarjan proved a matching lower bound: in the **pointer machine** model, any data structure for the disjoint-set problem requires $\Omega(m \cdot \alpha(n))$ time for a sequence of m operations on n elements. The union-find structure with path compression and union by rank is **asymptotically optimal**.

20.6 Implementation

Our TypeScript implementation uses a `Map` for the parent and rank arrays, which allows the element type `T` to be any hashable value — not just integers.

```
export class UnionFind<T> {
  private parent = new Map<T, T>();
  private rank = new Map<T, number>();
  private _componentCount = 0;

  makeSet(x: T): void {
    if (this.parent.has(x)) return;
    this.parent.set(x, x);
    this.rank.set(x, 0);
    this._componentCount++;
  }

  find(x: T): T {
    let root = x;
    while (this.parent.get(root) !== root) {
      root = this.parent.get(root)!;
    }
    // Path compression: point every node on path directly to root.
    let current = x;
    while (current !== root) {
      const next = this.parent.get(current)!;
      this.parent.set(current, root);
      current = next;
    }
    return root;
  }

  union(x: T, y: T): boolean {
    const rootX = this.find(x);
    const rootY = this.find(y);
    if (rootX === rootY) return false;

    const rankX = this.rank.get(rootX)!;
    const rankY = this.rank.get(rootY)!;
    if (rankX < rankY) {
      this.parent.set(rootX, rootY);
    } else if (rankX > rankY) {
      this.parent.set(rootY, rootX);
    } else {
      this.parent.set(rootY, rootX);
      this.rank.set(rootX, rankX + 1);
```

```
    }
    this._componentCount--;
    return true;
  }

  connected(x: T, y: T): boolean {
    return this.find(x) === this.find(y);
  }

  get componentCount(): number {
    return this._componentCount;
  }

  get size(): number {
    return this.parent.size;
  }
}
```

20.6.1 Design decisions

Generic type parameter. The `UnionFind<T>` class works with any element type — numbers, strings, or objects — as long as elements can be used as `Map` keys (i.e., identity via `===`). This is more flexible than an array-based implementation that requires elements to be integer indices.

Idempotent `makeSet`. Calling `makeSet(x)` when `x` is already in a set is a no-op. This simplifies client code that may process elements from an unknown source.

Return value of `union`. The method returns `true` if a merge actually happened and `false` if the elements were already in the same set. This is useful for Kruskal's algorithm, which needs to know whether an edge was added to the MST.

Component count. The `componentCount` property tracks the number of disjoint sets, which is useful for dynamic connectivity queries ("How many connected components remain?").

20.6.2 Complexity summary

Operation	Amortized Time
`makeSet`	$O(1)$
`find`	$O(\alpha(n))$

Operation	Amortized Time
`union`	$O(\alpha(n))$
`connected`	$O(\alpha(n))$

Space: $O(n)$ for n elements.

20.7 Trace through an example

Let us trace through a sequence of operations on integers $\{0, 1, 2, 3, 4, 5, 6, 7\}$. We show the parent array and rank array after each operation. An arrow $x \to y$ means `parent[x] = y`; a self-loop means x is a root.

After makeSet(0) through makeSet(7):

```
parent: 0→0  1→1  2→2  3→3  4→4  5→5  6→6  7→7
rank:   0:0  1:0  2:0  3:0  4:0  5:0  6:0  7:0
components: 8
```

Every element is its own root with rank 0.

union(0, 1): Roots 0 and 1 both have rank 0, so attach 1 under 0. Increment rank of 0.

```
parent: 0→0  1→0  2→2  3→3  4→4  5→5  6→6  7→7
rank:   0:1  1:0  2:0  3:0  4:0  5:0  6:0  7:0
components: 7
```

union(2, 3): Attach 3 under 2. Increment rank of 2.

```
parent: 0→0  1→0  2→2  3→2  4→4  5→5  6→6  7→7
rank:   0:1  1:0  2:1  3:0  4:0  5:0  6:0  7:0
components: 6
```

union(4, 5): Attach 5 under 4.

```
parent: 0→0  1→0  2→2  3→2  4→4  5→4  6→6  7→7
rank:   0:1  1:0  2:1  3:0  4:1  5:0  6:0  7:0
components: 5
```

union(6, 7): Attach 7 under 6.

```
parent: 0→0  1→0  2→2  3→2  4→4  5→4  6→6  7→6
rank:   0:1  1:0  2:1  3:0  4:1  5:0  6:1  7:0
components: 4
```

union(0, 2): Roots 0 and 2 both have rank 1. Attach 2 under 0. Increment rank of 0.

```
parent: 0→0  1→0  2→0  3→2  4→4  5→4  6→6  7→6
rank:   0:2  1:0  2:1  3:0  4:1  5:0  6:1  7:0
components: 3
```

union(4, 6): Roots 4 and 6 both have rank 1. Attach 6 under 4.

```
parent: 0→0  1→0  2→0  3→2  4→4  5→4  6→4  7→6
rank:   0:2  1:0  2:1  3:0  4:2  5:0  6:1  7:0
components: 2
```

union(0, 4): Roots 0 and 4 both have rank 2. Attach 4 under 0.

```
parent: 0→0  1→0  2→0  3→2  4→0  5→4  6→4  7→6
rank:   0:3  1:0  2:1  3:0  4:2  5:0  6:1  7:0
components: 1
```

find(7): Follow the path 7 → 6 → 4 → 0. The root is 0. Path compression sets `parent[7] = 0`, `parent[6] = 0`, and `parent[4] = 0` (4 was already pointing to 0).

```
parent: 0→0  1→0  2→0  3→2  4→0  5→4  6→0  7→0
rank:   (unchanged — path compression does not alter ranks)
```

After this find, the next call to `find(7)` completes in a single step.

find(3): Follow 3 → 2 → 0. Path compression sets `parent[3] = 0`.

```
parent: 0→0  1→0  2→0  3→0  4→0  5→4  6→0  7→0
```

Now almost every node points directly to the root. The tree is nearly flat, and future finds will be very fast.

20.8 Applications

20.8.1 Kruskal's minimum spanning tree

The most classic application of Union-Find is in Kruskal's algorithm (Chapter 14). The algorithm sorts edges by weight and processes them in order. For each edge (u, v):

1. Call `find(u)` and `find(v)` to check if u and v are in the same component.
2. If not, call `union(u, v)` and add the edge to the MST.

Without Union-Find, cycle detection would require a full graph traversal for each edge, costing $O(V + E)$ per edge and $O(E(V + E))$ overall. With Union-Find, the total cost of all find and union operations is $O(E \cdot \alpha(V))$, which is effectively $O(E)$.

20.8.2 Dynamic connectivity

In the **dynamic connectivity** problem, we process a stream of edge insertions in an undirected graph and must answer connectivity queries: "Are vertices u and v connected?"

Union-Find handles this directly: when edge (u, v) is inserted, call `union(u, v)`. To answer a connectivity query, call `connected(u, v)`. Each operation runs in amortized $O(\alpha(n))$ time.

Note that standard Union-Find only supports **incremental** connectivity — edges can be added but not removed. Supporting deletions requires more sophisticated data structures (such as link-cut trees or the Euler tour tree), which are beyond the scope of this book.

20.8.3 Connected components in an image

In image processing, **connected-component labeling** groups pixels into regions. Two adjacent pixels are in the same component if they share some property (e.g., similar color).

The algorithm scans the image in raster order (left to right, top to bottom). For each pixel:

1. Call `makeSet` for the pixel.
2. Check the pixel above and to the left. If either neighbor has a similar value, call `union` to merge the current pixel's set with the neighbor's set.
3. After scanning the entire image, each connected component corresponds to one disjoint set.

This is the standard "two-pass" connected-component labeling algorithm. Union-Find makes the second pass (resolving label equivalences) nearly linear.

20.8.4 Percolation

In a percolation simulation, we model a grid of cells where each cell is independently "open" with probability p or "blocked" with probability $1-p$. The question is: does an open path exist from the top row to the bottom row?

We model this with Union-Find:

1. Create a "virtual top" node connected to all open cells in the top row.
2. Create a "virtual bottom" node connected to all open cells in the bottom row.
3. For each open cell, union it with its open neighbors.
4. The system percolates if `connected(virtualTop, virtualBottom)`.

This allows efficient simulation of percolation for many values of p, enabling Monte Carlo estimation of the **percolation threshold** — the critical probability above which percolation almost certainly occurs.

20.9 Union by rank vs. union by size

An alternative to union by rank is **union by size**, which attaches the tree with fewer nodes beneath the tree with more nodes. Both strategies achieve $O(\log n)$ height without path compression and $O(\alpha(n))$ amortized time with path compression. The choice between them is largely a matter of taste:

- **Union by rank** is slightly simpler because rank is a single integer that only increases, never decreases, and is never affected by path compression.
- **Union by size** provides additional information: after the union, the root's size equals the total number of elements in the merged set. This is useful when you need to know component sizes.

Our implementation uses union by rank, following the approach in CLRS.

20.10 Exercises

Exercise 18.1. Starting from eight singleton sets $\{0\}, \{1\}, \ldots, \{7\}$, perform the following operations using union by rank and path compression. Draw the forest after each operation and show how path compression modifies the tree structure.

```
union(0, 1), union(2, 3), union(0, 2),
union(4, 5), union(6, 7), union(4, 6),
union(0, 4), find(7), find(3), find(5)
```

Exercise 18.2. Prove that with union by rank (without path compression), the rank of any root is at most $\lfloor \log_2 n \rfloor$. (*Hint: prove that a tree with root rank r has at least 2^r nodes, by induction on the number of union operations.*)

Exercise 18.3. Consider implementing union-find with path compression but **without** union by rank (i.e., always attaching the second root under the first, regardless of tree heights). What is the amortized time complexity per operation? Is it still $O(\alpha(n))$?

Exercise 18.4. Describe how to use Union-Find to detect whether an undirected graph has a cycle. Process the edges one by one; what condition indicates a cycle? Analyze the time complexity.

Exercise 18.5. A social network has n users. Friendships arrive as a stream of pairs (a, b). You want to determine the **exact moment** when all users become

connected (directly or transitively). Describe an algorithm using Union-Find and analyze its complexity.

(*Hint: maintain a component count and check when it reaches 1.*)

20.11 Summary

The **disjoint-set** (Union-Find) data structure maintains a partition of elements into disjoint sets, supporting `makeSet`, `find`, and `union` operations. Naive implementations achieve at best $O(\log n)$ per operation (with union by rank alone) or $O(n)$ in the worst case (without any optimizations).

Union by rank keeps trees shallow by always attaching the shorter tree beneath the taller one, guaranteeing a maximum height of $O(\log n)$.

Path compression flattens trees during `find` operations by pointing every traversed node directly at the root, making subsequent finds faster.

Together, union by rank and path compression achieve $O(\alpha(n))$ amortized time per operation, where α is the inverse Ackermann function — a function so slow-growing that $\alpha(n) \leq 4$ for any practically conceivable input size. This bound is **optimal**: no pointer-based data structure can do better.

Union-Find is a fundamental building block in algorithm design. Its primary application is **Kruskal's MST algorithm** (Chapter 14), where it provides efficient cycle detection. It also appears in dynamic connectivity, image processing, percolation, type unification in compilers, and many other settings. In Chapter 22, we will see Union-Find used again in approximation algorithms for NP-hard problems.

Chapter 21

Tries and String Data Structures

*The data structures we have studied so far — hash tables, balanced search trees, heaps — work well when keys are atomic values that can be compared or hashed in constant time. But many applications deal with **string** keys: dictionaries, autocomplete systems, IP routing tables, spell checkers, DNA sequence databases. For these, a data structure that exploits the **character-by-character structure** of keys can be far more efficient. The **trie** (from re**trie**val) is such a structure. In this chapter we develop the standard trie, optimize it into a **compressed trie** (radix tree) that eliminates wasted space, survey applications, and briefly introduce suffix arrays for substring search.*

21.1 The trie (prefix tree)

21.1.1 Motivation

Consider storing a dictionary of n words, where the total number of characters across all words is N, and answering these queries:

- **Lookup**: Is a given word in the dictionary?
- **Prefix search**: Are there any words starting with a given prefix?
- **Autocomplete**: List all words starting with a given prefix.

A hash table answers lookup in expected $O(m)$ time, where m is the length of the query word (we must hash the entire word). But it cannot answer prefix queries without scanning every stored word. A balanced BST stores words in sorted order and can answer prefix queries via range searches, but each comparison costs $O(m)$, so lookup costs $O(m \log n)$.

A trie answers all three queries in $O(m)$ time — proportional to the length of

the query, **independent of the number of stored words**. The key insight is that the trie avoids comparing entire keys; instead, it inspects one character at a time.

21.1.2 Structure

A **trie** (also called a **prefix tree**) is a rooted tree where:

- Each edge is labeled with a single character from the alphabet Σ.
- Each node has at most $|\Sigma|$ children (one per character).
- A node may be marked as an **end-of-word** node, indicating that the path from the root to that node spells a complete word.
- The **root** represents the empty prefix.

The crucial property is **prefix sharing**: words that share a common prefix share the same path from the root. For example, “app”, “apple”, and “application” all share the path `a → p → p`.

21.1.3 Operations

Insert(word). Starting from the root, follow (or create) the edge labeled with each character of the word. Mark the final node as an end-of-word.

Search(word). Starting from the root, follow the edge labeled with each character. If at any point the required edge does not exist, the word is not in the trie. If we reach the end of the word, check whether the current node is marked as an end-of-word.

StartsWith(prefix). Like search, but we do not require the final node to be an end-of-word. If we can follow all characters of the prefix, at least one stored word has that prefix.

Delete(word). First verify the word exists. Then unmark the end-of-word flag. If the node has no children and is not an end-of-word for another word, remove it. Propagate this cleanup upward: if a parent becomes childless and is not itself an end-of-word, remove it too. This ensures the trie does not retain unnecessary nodes.

Autocomplete(prefix, limit). Navigate to the node corresponding to the prefix, then collect all words in the subtree (via DFS), stopping after `limit` results.

21.1.4 Complexity analysis

Let m be the length of the key being operated on, and $|\Sigma|$ be the alphabet size.

Operation	Time
`insert`	$O(m)$
`search`	$O(m)$
`startsWith`	$O(m)$
`delete`	$O(m)$
`autocomplete`	$O(m + k)$ where k is the output size

Space. In the worst case a trie stores one node per character of every stored word, for $O(N)$ nodes where N is the total length of all words. Each node stores up to $|\Sigma|$ child pointers, so the total space is $O(N \cdot |\Sigma|)$. In practice, prefix sharing reduces the number of nodes significantly, especially when the stored words share many common prefixes.

When $|\Sigma|$ is small (e.g., DNA alphabet with 4 characters) or when using a hash map for child storage instead of a fixed-size array, the space is close to $O(N)$.

21.1.5 Implementation

Our implementation uses a `Map<string, TrieNode>` for each node's children, which supports arbitrary alphabets and avoids wasting space on unused child slots:

```
export class TrieNode {
  readonly children = new Map<string, TrieNode>();
  isEnd = false;
}

export class Trie {
  private readonly root = new TrieNode();
  private _size = 0;

  get size(): number {
    return this._size;
  }

  insert(word: string): void {
    let node = this.root;
    for (const ch of word) {
      let child = node.children.get(ch);
      if (child === undefined) {
        child = new TrieNode();
        node.children.set(ch, child);
```

```
      }
      node = child;
    }
    if (!node.isEnd) {
      node.isEnd = true;
      this._size++;
    }
  }

  search(word: string): boolean {
    const node = this.findNode(word);
    return node !== null && node.isEnd;
  }

  startsWith(prefix: string): boolean {
    return this.findNode(prefix) !== null;
  }

  private findNode(key: string): TrieNode | null {
    let node: TrieNode = this.root;
    for (const ch of key) {
      const child = node.children.get(ch);
      if (child === undefined) return null;
      node = child;
    }
    return node;
  }
}
```

Insert iterates character by character, creating child nodes as needed. Each character lookup in the `Map` is $O(1)$ expected time, so the total is $O(m)$.

Search and **startsWith** both call `findNode`, which walks the trie following the key's characters. The difference is that `search` additionally checks the `isEnd` flag.

Delete is more involved because we must clean up nodes that are no longer needed:

```
delete(word: string): boolean {
  if (!this.search(word)) return false;
  this.deleteHelper(this.root, word, 0);
  this._size--;
  return true;
```

```
}

private deleteHelper(node: TrieNode, word: string, depth: number): boolean {
  if (depth === word.length) {
    node.isEnd = false;
    return node.children.size === 0;
  }

  const ch = word[depth]!;
  const child = node.children.get(ch);
  if (child === undefined) return false;

  const shouldDeleteChild = this.deleteHelper(child, word, depth + 1);

  if (shouldDeleteChild) {
    node.children.delete(ch);
    return node.children.size === 0 && !node.isEnd;
  }

  return false;
}
```

The `deleteHelper` returns `true` when a node should be removed (it has no children and is not an end-of-word). This propagates up the recursion, cleaning the path.

Autocomplete navigates to the prefix node and then performs a DFS to collect all words in the subtree:

```
autocomplete(prefix: string, limit = Infinity): string[] {
  const node = this.findNode(prefix);
  if (node === null) return [];

  const results: string[] = [];
  this.collectWords(node, prefix, results, limit);
  return results;
}

private collectWords(
  node: TrieNode,
  prefix: string,
  results: string[],
  limit: number,
```

```
): void {
  if (results.length >= limit) return;

  if (node.isEnd) {
    results.push(prefix);
    if (results.length >= limit) return;
  }

  const sortedKeys = [...node.children.keys()].sort();
  for (const ch of sortedKeys) {
    this.collectWords(node.children.get(ch)!, prefix + ch, results, limit);
    if (results.length >= limit) return;
  }
}
```

By iterating children in sorted order, we produce results in lexicographic order.

21.1.6 Trace through an example

Let us insert the words “app”, “apple”, “apply”, and “banana” into an initially empty trie.

After inserting “app”:

```
(root)
  └─ a
     └─ p
        └─ p*
```

An asterisk (*) marks end-of-word nodes.

After inserting “apple”:

```
(root)
  └─ a
     └─ p
        └─ p*
           └─ l
              └─ e*
```

The path `a → p → p` is shared. The new characters `l → e` extend from the existing “app” node.

After inserting “apply”:

```
(root)
```

```
  └─ a
     └─ p
        └─ p*
           ├─ l
           │  ├─ e*
           │  └─ y*
```

The node for `l` now has two children: `e` (for “apple”) and `y` (for “apply”).

After inserting “banana”:

```
(root)
  ├─ a
  │  └─ p
  │     └─ p*
  │        └─ l
  │           ├─ e*
  │           └─ y*
  └─ b
     └─ a
        └─ n
           └─ a
              └─ n
                 └─ a*
```

Now `autocomplete("app")` returns `["app", "apple", "apply"]` — the word “app” itself plus all words in its subtree.

21.2 Compressed tries (radix trees)

21.2.1 The problem with standard tries

In a standard trie, a chain of nodes with a single child wastes space. Consider storing only the word “internationalization” in a trie: it requires 20 nodes, each with exactly one child, plus the root. This is 21 nodes for a single word.

More generally, if the stored words have long unique suffixes, the trie degenerates into long chains. These chains use $O(1)$ space per character but create many nodes, each carrying a child map overhead.

21.2.2 Compressing single-child chains

A **compressed trie** (also called a **radix tree** or **Patricia tree**) eliminates single-child chains by storing an entire substring on each edge rather than a single character. The rule is:

Every internal node (except the root) has at least two children.

If a node has exactly one child and is not an end-of-word, it is merged with that child by concatenating their edge labels.

For example, the standard trie for {“romane”, “romanus”, “romulus”, “rubens”, “ruber”, “rubicon”, “rubicundus”} has many single-child chains. The compressed trie looks like:

```
(root)
  └─ "r"
     ├─ "om"
     │   ├─ "an"
     │   │   ├─ "e"*
     │   │   └─ "us"*
     │   └─ "ulus"*
     └─ "ub"
         ├─ "e"
         │   ├─ "ns"*
         │   └─ "r"*
         └─ "ic"
             ├─ "on"*
             └─ "undus"*
```

Instead of one node per character, each edge carries a substring. The total number of nodes is bounded by $2n+1$ where n is the number of stored words (at most n leaves, at most $n-1$ internal branching nodes, plus the root).

21.2.3 Operations

The operations are conceptually the same as for a standard trie, but each step may match multiple characters at once:

Insert(word). Navigate the trie, matching the word against edge labels. There are three cases:

1. **No matching child.** Create a new leaf node with the remaining suffix as its label.
2. **Edge label is a prefix of the remaining word.** Recurse into the child with the rest of the word.
3. **Edge label and remaining word diverge.** Split the edge: create a new internal node at the divergence point, move the existing child beneath it with a shortened label, and create a new leaf for the remaining suffix.

Search(word). Navigate the trie, matching edge labels character by character. The word is found only if we arrive at a node boundary (not in the middle of an

edge label) and the node is marked as an end-of-word.

StartsWith(prefix). Like search, but the prefix may end in the middle of an edge label — this is acceptable because the label continues with characters that extend the prefix.

Delete(word). Find and unmark the node. If it becomes a leaf, remove it. If its parent now has only one child and is not an end-of-word, merge the parent with its child by concatenating labels. This maintains the compressed trie invariant.

21.2.4 Complexity

Operation	Time
`insert`	$O(m)$
`search`	$O(m)$
`startsWith`	$O(m)$
`delete`	$O(m)$
`autocomplete`	$O(m + k)$

Space. The number of nodes is $O(n)$ where n is the number of stored words — a major improvement over the standard trie's $O(N)$ nodes. However, each node stores a substring label, and the total length of all labels is $O(N)$. So total space is $O(N)$ in terms of characters stored, but with far fewer node objects.

21.2.5 Implementation

The key difference from a standard trie is the **split** operation during insertion. When an edge label and the remaining word diverge at some position, we must create a new branching node:

```
export class CompressedTrieNode {
  readonly children = new Map<string, CompressedTrieNode>();
  label: string;
  isEnd = false;

  constructor(label: string) {
    this.label = label;
  }
}
```

Each child in the map is keyed by the **first character** of its label. This allows $O(1)$ lookup of the correct child for the next character in the key.

The insert helper handles the three cases:

```
private insertHelper(node: CompressedTrieNode, remaining: string): void {
  const firstChar = remaining[0]!;
  const child = node.children.get(firstChar);

  if (child === undefined) {
    // Case 1: no matching child — create a new leaf
    const newNode = new CompressedTrieNode(remaining);
    newNode.isEnd = true;
    node.children.set(firstChar, newNode);
    this._size++;
    return;
  }

  const commonLen = commonPrefixLength(child.label, remaining);

  if (commonLen === child.label.length && commonLen === remaining.length) {
    // Exact match with existing node
    if (!child.isEnd) {
      child.isEnd = true;
      this._size++;
    }
    return;
  }

  if (commonLen === child.label.length) {
    // Case 2: child label is a prefix of remaining — recurse
    this.insertHelper(child, remaining.slice(commonLen));
    return;
  }

  // Case 3: split — labels diverge at position commonLen
  const splitNode = new CompressedTrieNode(
    child.label.slice(0, commonLen),
  );
  node.children.set(firstChar, splitNode);

  // Move existing child beneath the split node
  child.label = child.label.slice(commonLen);
  splitNode.children.set(child.label[0]!, child);

  if (commonLen === remaining.length) {
```

```
    splitNode.isEnd = true;
    this._size++;
  } else {
    const newLeaf = new CompressedTrieNode(remaining.slice(commonLen));
    newLeaf.isEnd = true;
    splitNode.children.set(newLeaf.label[0]!, newLeaf);
    this._size++;
  }
}
```

Search must check that the word ends exactly at a node boundary — not partway through an edge label:

```
private findExactNode(
  node: CompressedTrieNode,
  key: string,
): CompressedTrieNode | null {
  let offset = 0;

  for (;;) {
    if (offset === key.length) return node;

    const child = node.children.get(key[offset]!);
    if (child === undefined) return null;

    const label = child.label;
    const remaining = key.length - offset;

    if (remaining < label.length) {
      // Key ends within this edge's label — not an exact match
      return null;
    }

    if (key.slice(offset, offset + label.length) !== label) {
      return null;
    }

    offset += label.length;
    node = child;
  }
}
```

Delete must maintain the compression invariant by merging nodes when appropriate:

```
private mergeWithChild(
  parent: CompressedTrieNode,
  key: string,
  node: CompressedTrieNode,
): void {
  if (node.children.size !== 1 || node.isEnd) return;

  const entry = [...node.children.entries()][0]!;
  const onlyChild = entry[1];
  onlyChild.label = node.label + onlyChild.label;
  parent.children.set(key, onlyChild);
}
```

When a node loses its end-of-word flag (or a child is deleted) and has exactly one remaining child, we merge the node with that child by concatenating their labels and removing the intermediate node.

21.2.6 Design decisions

Map-based children, keyed by first character. Each child's label starts with a unique character (since we split on divergence), so the first character serves as a unique key. This gives $O(1)$ child lookup.

Separate `findExactNode` and `findNodeForPrefix`. Search requires an exact match at a node boundary, while `startsWith` and `autocomplete` allow partial matches within an edge label. We use two different navigation methods to handle these semantics correctly.

Node count tracking. The `nodeCount()` method allows testing that the trie is properly compressed — for instance, a single word should result in exactly 2 nodes (root + one leaf), not one node per character.

21.3 Standard trie vs. compressed trie

Property	Standard trie	Compressed trie
Nodes	$O(N)$	$O(n)$
Space (total)	$O(N \cdot \lvert\Sigma\rvert)$	$O(N)$
Lookup time	$O(m)$	$O(m)$
Insert time	$O(m)$	$O(m)$

Property	Standard trie	Compressed trie
Implementation	Simpler	More complex (splitting/merging)
Best for	Small alphabets, many short words	Long words, shared prefixes

Where N = total characters across all words, n = number of words, m = query length, $|\Sigma|$ = alphabet size.

For most practical applications the compressed trie is preferred because it uses $O(n)$ nodes regardless of word length, and its operations have the same asymptotic time complexity as the standard trie.

21.4 Applications

21.4.1 Autocomplete and search suggestions

The most visible application of tries is autocomplete. When a user types a prefix in a search box, the system queries a trie to find all stored strings matching that prefix. The trie's structure makes this natural: navigate to the prefix node in $O(m)$ time, then enumerate the subtree.

In practice, autocomplete systems augment the trie with **frequency counts** or **ranking scores** at each end-of-word node, so the most popular completions are returned first.

21.4.2 Spell checking

A trie can serve as the dictionary for a spell checker. Given a misspelled word, we can:

1. **Edit-distance search**: enumerate all words within edit distance 1 or 2 by performing DFS on the trie while tracking allowed edits (insertions, deletions, substitutions). This is far more efficient than computing edit distance against every dictionary word.
2. **Prefix validation**: as the user types, highlight prefixes that cannot lead to any valid word (the trie returns `startsWith(prefix) = false`).

21.4.3 IP routing (longest prefix match)

Internet routers must match an incoming IP address against a routing table to determine the next hop. The routing table contains prefixes of various lengths,

and the router must find the **longest matching prefix**. A trie indexed on the bits of the IP address solves this efficiently: navigate the trie bit by bit, keeping track of the last end-of-word node encountered. This is the standard data structure in router implementations.

Compressed tries (specifically, the **Patricia tree** variant) are particularly well-suited here because IP prefixes tend to be long and share common leading bits.

21.4.4 T9 predictive text

The T9 system for numeric keypads maps each key to several letters (2 → {a, b, c}, 3 → {d, e, f}, etc.). Given a sequence of key presses, T9 must find all dictionary words that match. A trie indexed by the **key mappings** rather than the letters themselves allows efficient lookup.

21.4.5 Bioinformatics

DNA sequences over the alphabet $\{A, C, G, T\}$ are naturally stored in tries with branching factor 4. Suffix tries (discussed below) enable fast substring search in genomic databases.

21.5 Suffix arrays (conceptual overview)

While tries excel at prefix queries, many applications require **substring search**: given a text T of length n, preprocess it so that queries "Does pattern P appear in T?" can be answered quickly.

A **suffix array** is a sorted array of all suffixes of T, represented by their starting positions. For example, for T = "banana":

Index	Suffix
5	"a"
3	"ana"
1	"anana"
0	"banana"
4	"na"
2	"nana"

Since the array is sorted, we can binary-search for any pattern P in $O(m \log n)$ time, where $m = |P|$. With an auxiliary **LCP array** (longest common prefix between consecutive suffixes), this can be improved to $O(m + \log n)$.

Construction. A suffix array can be built in $O(n)$ time using the SA-IS algorithm or in $O(n \log n)$ time using simpler prefix-doubling approaches. The space is $O(n)$ — just an array of integers.

Relation to suffix trees. A **suffix tree** is a compressed trie of all suffixes of T. It supports $O(m)$ substring queries (faster than suffix arrays without LCP) but uses significantly more space — typically 10-20 times the size of the text. Suffix arrays are the preferred choice in practice due to their compact representation and cache-friendly access patterns.

We do not implement suffix arrays in this chapter, as their construction algorithms are more specialized. The key takeaway is that the trie concept extends naturally to substring search when applied to suffixes.

21.6 Exercises

Exercise 19.1. Insert the words “bear”, “bell”, “bid”, “bull”, “buy”, “sell”, “stock”, “stop” into an empty trie. Draw the resulting trie and count the total number of nodes (including the root). Then repeat the exercise with a compressed trie and compare the node counts.

Exercise 19.2. A standard trie over an alphabet of size $|\Sigma|$ with n stored words has at most $N + 1$ nodes (where N is the total number of characters). Prove that a compressed trie has at most $2n + 1$ nodes. (*Hint: every internal node except the root has at least two children, and there are exactly n leaves.*)

Exercise 19.3. Modify the `Trie` class to support **wildcard search**: `search("b.ll")` should match “ball”, “bell”, “bill”, “bull”, etc., where `.` matches any single character. What is the time complexity of your solution?

Exercise 19.4. You are designing an autocomplete system for a search engine. Each query has an associated frequency count. Describe how to modify the trie to return the top-k most frequent completions of a prefix efficiently. What data would you store at each node? What is the time complexity?

(*Hint: consider storing the top-k completions at each node, or augmenting the trie with a priority queue.*)

Exercise 19.5. An IP routing table contains the following prefixes (in binary): “0”, “01”, “011”, “1”, “10”, “100”, “1000”. Build a compressed trie for these prefixes. Given the IP address “10010110” (in binary), trace the longest-prefix-match lookup and identify which prefix matches.

21.7 Summary

A **trie** (prefix tree) is a tree-based data structure that stores strings by their character-by-character structure. Each path from the root to an end-of-word node represents a stored string, and strings that share a common prefix share the same initial path. This yields $O(m)$ lookup, insertion, and deletion, where m is the key length — independent of the number of stored strings.

A **compressed trie** (radix tree) optimizes the standard trie by collapsing chains of single-child nodes into single edges labeled with substrings. This reduces the node count from $O(N)$ to $O(n)$, where N is the total length of all stored strings and n is the number of strings. The time complexity of all operations remains $O(m)$.

Tries are the natural choice for problems involving **prefix queries**: autocomplete, spell checking, IP routing, and predictive text. For **substring queries**, the trie concept extends to suffix trees and suffix arrays, which preprocess a text to enable fast pattern matching.

The trie is one of the most elegant examples of a data structure designed around the structure of the data it stores. Rather than treating keys as opaque objects to be compared or hashed, it decomposes keys into their constituent characters and exploits shared structure. This principle — designing data structures that respect the internal structure of their keys — is a powerful idea that appears throughout computer science.

Chapter 22

String Matching

Given a text T of length n and a pattern P of length m, find all positions in T where P occurs. This deceptively simple problem — searching for a word in a document, a DNA motif in a genome, a keyword in a log file — is one of the most fundamental in computer science. In this chapter we develop three algorithms of increasing sophistication: the naive brute-force approach, the Rabin-Karp algorithm based on rolling hashes, and the Knuth-Morris-Pratt (KMP) algorithm based on the failure function. Each illustrates a different strategy for avoiding redundant comparisons.

22.1 The pattern matching problem

Input. A text string $T[0 \ldots n-1]$ and a pattern string $P[0 \ldots m-1]$, where $m \leq n$.

Output. All indices i such that $T[i \ldots i+m-1] = P$, i.e., all positions where the pattern occurs in the text.

We call each such i a **valid shift**. A shift i is **invalid** if $T[i \ldots i+m-1] \neq P$.

There are $n - m + 1$ possible shifts to check ($i = 0, 1, \ldots, n-m$). The challenge is to avoid checking each one character by character from scratch. The three algorithms in this chapter differ in how they eliminate invalid shifts:

Algorithm	Strategy	Time (worst)	Time (expected)	Space
Naive	Check every shift from scratch	$O(nm)$	$O(nm)$	$O(1)$

Algorithm	Strategy	Time (worst)	Time (expected)	Space
Rabin-Karp	Use hashing to filter shifts	$O(nm)$	$O(n+m)$	$O(1)$
KMP	Use a failure function to skip shifts	$O(n+m)$	$O(n+m)$	$O(m)$

22.2 Naive string matching

The simplest approach: for each possible starting position i in the text, compare the pattern against $T[i \ldots i+m-1]$ character by character. If all m characters match, record i as a valid shift. If any character fails to match, move to position $i+1$ and start over.

22.2.1 Algorithm

```
NAIVE-MATCH(T, P):
    n ← length(T)
    m ← length(P)
    for i ← 0 to n − m:
        j ← 0
        while j < m and T[i + j] = P[j]:
            j ← j + 1
        if j = m:
            report match at position i
```

22.2.2 Trace through an example

Consider $T = \texttt{aabaabaac}$ and $P = \texttt{aabac}$. We have $n = 9$ and $m = 5$.

Shift i	Comparison	Result
0	aabaa vs aabac	Mismatch at $j = 4$ (a $\neq$ c)
1	abaab vs aabac	Mismatch at $j = 1$ (b $\neq$ a)
2	baaba vs aabac	Mismatch at $j = 0$ (b $\neq$ a)
3	aabaa vs aabac	Mismatch at $j = 4$ (a $\neq$ c)
4	abaac vs aabac	Mismatch at $j = 1$ (b $\neq$ a)

No match is found. Notice that at shift 0 we successfully matched four characters

before failing, yet at shift 1 we start the comparison entirely from scratch — discarding all information gained from the previous attempt. The algorithms that follow exploit this wasted information.

22.2.3 Implementation

```
export function naiveMatch(text: string, pattern: string): number[] {
  const n = text.length;
  const m = pattern.length;
  const result: number[] = [];

  if (m === 0) return result;
  if (m > n) return result;

  for (let i = 0; i <= n - m; i++) {
    let j = 0;
    while (j < m && text[i + j] === pattern[j]) {
      j++;
    }
    if (j === m) {
      result.push(i);
    }
  }

  return result;
}
```

22.2.4 Complexity analysis

The outer loop runs $n - m + 1$ times. In the worst case, the inner loop performs m comparisons before discovering a mismatch (e.g., $T = \texttt{aaa...a}$ and $P = \texttt{aaa...ab}$). The total number of character comparisons is therefore $O((n - m + 1) \cdot m) = O(nm)$.

Best case. If the first character of the pattern rarely appears in the text, most shifts are eliminated after a single comparison, giving $O(n)$ in practice.

Average case. For random text over an alphabet of size $|\Sigma|$, the expected number of comparisons per shift is $\frac{1}{1-1/|\Sigma|} \approx 1 + \frac{1}{|\Sigma|}$ (a geometric series), so the expected total is $O(n)$. But for small alphabets (e.g., binary) or structured text (e.g., DNA), the worst case is more likely.

Space. $O(1)$ beyond the output array. No preprocessing is needed.

22.3 Rabin-Karp string matching

The Rabin-Karp algorithm avoids re-examining every character at every shift by using **hashing**. The idea: compute a hash of the pattern and a hash of each text window of length m. If the hashes differ, the window cannot match and we skip it without comparing characters. If the hashes match, we verify character by character to eliminate false positives (hash collisions).

The key insight is that the hash of the next window $T[i+1 \ldots i+m]$ can be computed from the hash of the current window $T[i \ldots i+m-1]$ in $O(1)$ time using a **rolling hash**. This makes the overall hash computation $O(n)$ rather than $O(nm)$.

22.3.1 Rolling hash

We treat each string of length m as a number in base d (where d is the alphabet size) and take the result modulo a prime q:

$$h(T[i \ldots i+m-1]) = \left(\sum_{j=0}^{m-1} T[i+j] \cdot d^{m-1-j} \right) \bmod q$$

When we slide the window one position to the right, the new hash is:

$$h(T[i+1 \ldots i+m]) = \left(d \cdot \left(h(T[i \ldots i+m-1]) - T[i] \cdot d^{m-1} \right) + T[i+m] \right) \bmod q$$

This recurrence removes the contribution of the leftmost character $T[i]$ and adds the new rightmost character $T[i+m]$. The value $d^{m-1} \bmod q$ is a constant that we precompute once.

22.3.2 Algorithm

```
RABIN-KARP(T, P):
    n ← length(T)
    m ← length(P)
    d ← 256                          // alphabet size
    q ← 1000000007                   // large prime
    h ← d^(m−1) mod q                // precomputed weight

    // Initial hashes
    patternHash ← 0
    windowHash ← 0
    for j ← 0 to m − 1:
```

```
        patternHash ← (patternHash · d + P[j]) mod q
        windowHash ← (windowHash · d + T[j]) mod q

    // Slide the window
    for i ← 0 to n − m:
        if windowHash = patternHash:
            if T[i..i+m−1] = P:      // verify to eliminate collisions
                report match at position i
        if i < n − m:
            windowHash ← (d · (windowHash − T[i] · h) + T[i+m]) mod q
            if windowHash < 0:
                windowHash ← windowHash + q
```

22.3.3 Trace through an example

Consider $T = 31415926$ and $P = 1592$. Using $d = 10$ and $q = 13$ for illustration:

- $h = 10^3 \bmod 13 = 1000 \bmod 13 = 12$
- patternHash $= 1592 \bmod 13 = 6$

Shift i	Window	Hash	Match?
0	3141	$3141 \bmod 13 = 7$	No
1	1415	$(10 \cdot (7 - 3 \cdot 12) + 5) \bmod 13 = 10$	No
2	4159	roll… $= 8$	No
3	1592	roll… $= 6$	Hash match! Verify: $1592 = 1592$. Match at $i = 3$.
4	5926	roll… $= 3$	No

22.3.4 Implementation

```
export function rabinKarp(text: string, pattern: string): number[] {
  const n = text.length;
  const m = pattern.length;
  const result: number[] = [];
```

```
  if (m === 0) return result;
  if (m > n) return result;

  const d = 256;           // alphabet size (extended ASCII)
  const q = 1_000_000_007; // prime modulus

  // Precompute d^(m-1) mod q
  let h = 1;
  for (let i = 0; i < m - 1; i++) {
    h = (h * d) % q;
  }

  // Initial hash values
  let patternHash = 0;
  let windowHash = 0;
  for (let i = 0; i < m; i++) {
    patternHash = (patternHash * d + pattern.charCodeAt(i)) % q;
    windowHash = (windowHash * d + text.charCodeAt(i)) % q;
  }

  // Slide the pattern across the text
  for (let i = 0; i <= n - m; i++) {
    if (windowHash === patternHash) {
      let match = true;
      for (let j = 0; j < m; j++) {
        if (text[i + j] !== pattern[j]) {
          match = false;
          break;
        }
      }
      if (match) {
        result.push(i);
      }
    }

    if (i < n - m) {
      windowHash =
        ((windowHash - text.charCodeAt(i) * h) * d +
          text.charCodeAt(i + m)) % q;
      if (windowHash < 0) {
        windowHash += q;
```

```
      }
    }
  }

  return result;
}
```

22.3.5 Complexity analysis

Preprocessing. Computing $d^{m-1} \bmod q$ and the initial hashes takes $O(m)$.

Searching. The rolling hash update at each shift costs $O(1)$. Hash comparisons cost $O(1)$. When hashes match, verification costs $O(m)$.

- **Expected case.** If the hash function distributes uniformly, the probability of a spurious hit (collision) at any shift is $\frac{1}{q}$. The expected total verification cost is $(n - m + 1) \cdot \frac{m}{q}$, which is negligible for large q. Combined with $O(n)$ for rolling hashes, the expected time is $O(n + m)$.
- **Worst case.** If every window produces a collision (e.g., T and P consist entirely of the same character), every hash match requires $O(m)$ verification, giving $O(nm)$ — no better than naive. Choosing a large random prime q makes this scenario astronomically unlikely in practice.

Space. $O(1)$ beyond the output array.

22.3.6 Why Rabin-Karp matters

Rabin-Karp's main advantage over the other algorithms in this chapter is its easy generalization to **multi-pattern search**: given k patterns, compute all their hashes and store them in a set, then check each window's hash against the set. This yields expected $O(n + km)$ time for searching k patterns simultaneously — far better than running KMP k times.

Rabin-Karp is also the foundation of **plagiarism detection** systems: by computing rolling hashes of fixed-length substrings in two documents, matching hashes identify shared passages.

22.4 Knuth-Morris-Pratt (KMP)

The KMP algorithm achieves $O(n + m)$ time in the **worst case**, not just in expectation. The key idea: when a mismatch occurs after matching j characters of the pattern, we have already seen the text characters $T[i \dots i + j - 1]$ and know they equal $P[0 \dots j - 1]$. Instead of restarting from scratch at shift $i + 1$, we can use

this information to determine the longest possible overlap — how far the pattern can be shifted while still maintaining a partial match.

This information is encoded in the **failure function** (also called the **prefix function**).

22.4.1 The failure function

For a pattern $P[0 \ldots m-1]$, define:

$\pi[j]$ = length of the longest proper prefix of $P[0 \ldots j]$ that is also a suffix of $P[0 \ldots j]$

In other words, $\pi[j]$ is the length of the longest string that appears both at the start and the end of $P[0 \ldots j]$, excluding the trivial case of the entire string.

Example. For $P =$ `ababaca`:

j	$P[0 \ldots j]$	Longest proper prefix = suffix	$\pi[j]$
0	`a`	(none)	0
1	`ab`	(none)	0
2	`aba`	`a`	1
3	`abab`	`ab`	2
4	`ababa`	`aba`	3
5	`ababac`	(none)	0
6	`ababaca`	`a`	1

22.4.2 Computing the failure function

The failure function can be computed in $O(m)$ time by recognizing that computing π is itself a pattern-matching problem: we are matching the pattern against itself.

```
COMPUTE-FAILURE(P):
    m ← length(P)
    π[0] ← 0
    k ← 0
    for i ← 1 to m − 1:
        while k > 0 and P[k] ≠ P[i]:
            k ← π[k − 1]            // fall back
        if P[k] = P[i]:
            k ← k + 1
        π[i] ← k
```

```
    return π
```

The variable k tracks the length of the current match between a prefix and a suffix. When a mismatch occurs, we "fall back" to $\pi[k-1]$, which gives the next longest prefix that could still match. This cascade of fallbacks is the heart of KMP.

Why is this $O(m)$? Although the inner `while` loop can execute multiple times for a single i, each fallback decreases k by at least 1. Since k increases by at most 1 per iteration of the outer loop and can never go below 0, the total number of fallback operations across all iterations is at most m. The total work is therefore $O(m)$.

22.4.3 The KMP search algorithm

With the failure function in hand, the search proceeds as follows. We maintain a variable q that tracks how many characters of the pattern are currently matched against the text. On a mismatch, we fall back to $\pi[q-1]$ instead of restarting from 0:

```
KMP-SEARCH(T, P):
    n ← length(T)
    m ← length(P)
    π ← COMPUTE-FAILURE(P)
    q ← 0                                   // characters matched so far
    for i ← 0 to n − 1:
        while q > 0 and P[q] ≠ T[i]:
            q ← π[q − 1]                    // fall back
        if P[q] = T[i]:
            q ← q + 1
        if q = m:
            report match at position i − m + 1
          q ← π[q − 1]                  // continue for overlapping matches
```

22.4.4 Step-by-step trace

Let T = `abababaababaca` and P = `ababaca`. The failure function is $\pi = [0, 0, 1, 2, 3, 0, 1]$.

i	$T[i]$	q before	Action	q after
0	a	0	Match, $q \to 1$	1

i	$T[i]$	q before	Action	q after
1	b	1	Match, $q \to 2$	2
2	a	2	Match, $q \to 3$	3
3	b	3	Match, $q \to 4$	4
4	a	4	Match, $q \to 5$	5
5	b	5	$P[5] =$c $\neq$ b. Fall back: $q \to \pi[4] =$ 3. $P[3] =$b $=$ b. Match, $q \to 4$	4
6	a	4	Match, $q \to 5$	5
7	a	5	$P[5] =$c $\neq$ a. Fall back: $q \to \pi[4] =$ 3. $P[3] =$b $\neq$ a. Fall back: $q \to \pi[2] =$ 1. $P[1] =$b $\neq$ a. Fall back: $q \to \pi[0] =$ 0. $P[0] =$a $=$ a. Match, $q \to 1$	1
8	b	1	Match, $q \to 2$	2
9	a	2	Match, $q \to 3$	3
10	b	3	Match, $q \to 4$	4
11	a	4	Match, $q \to 5$	5
12	c	5	Match, $q \to 6$	6

i	$T[i]$	q before	Action	q after
13	a	6	Match, $q \to 7 = m$. **Match at position** 13−7+1 = 7**.** Fall back: $q \to \pi[6] = 1$	1

The pattern ababaca is found at position 7 in the text.

Notice at $i = 5$: after matching 5 characters, we discovered a mismatch. Instead of going back to shift 1 and starting over, the failure function told us that the last 3 matched characters (aba) form a prefix of the pattern, so we could continue from $q = 3$. This is the savings that gives KMP its efficiency.

22.4.5 Implementation

```
export function computeFailure(pattern: string): number[] {
  const m = pattern.length;
  const failure = new Array<number>(m).fill(0);

  let k = 0;

  for (let i = 1; i < m; i++) {
    while (k > 0 && pattern[k] !== pattern[i]) {
      k = failure[k - 1]!;
    }

    if (pattern[k] === pattern[i]) {
      k++;
    }

    failure[i] = k;
  }

  return failure;
}

export function kmpSearch(text: string, pattern: string): number[] {
  const n = text.length;
```

```
  const m = pattern.length;
  const result: number[] = [];

  if (m === 0) return result;
  if (m > n) return result;

  const failure = computeFailure(pattern);

  let q = 0;

  for (let i = 0; i < n; i++) {
    while (q > 0 && pattern[q] !== text[i]) {
      q = failure[q - 1]!;
    }

    if (pattern[q] === text[i]) {
      q++;
    }

    if (q === m) {
      result.push(i - m + 1);
      q = failure[q - 1]!;
    }
  }

  return result;
}
```

22.4.6 Complexity analysis

Failure function computation. $O(m)$ as argued above.

Search phase. By the same amortized argument: q increases by at most 1 per iteration of the outer loop, and each fallback in the `while` loop decreases q by at least 1. Since $q \geq 0$ always, the total number of fallback operations is at most n. Combined with the n iterations of the outer loop, the search phase takes $O(n)$.

Total. $O(n+m)$ in the worst case. This is optimal — we must read every character of both the text and the pattern at least once.

Space. $O(m)$ for the failure function array.

22.4.7 Why KMP is important

KMP is significant not just for its efficiency, but for the ideas it introduces:

1. **The failure function** captures the self-similarity structure of the pattern. This concept appears in many other string algorithms.
2. **Amortized analysis with a potential function.** The argument that the total number of fallbacks is bounded is a clean example of amortized analysis — the variable q serves as the potential.
3. **Online processing.** KMP processes the text left to right, one character at a time, never looking back. This makes it suitable for streaming data.

22.5 Comparison and practical considerations

Criterion	Naive	Rabin-Karp	KMP
Worst-case time	$O(nm)$	$O(nm)$	$O(n+m)$
Expected time	$O(n)$*	$O(n+m)$	$O(n+m)$
Extra space	$O(1)$	$O(1)$	$O(m)$
Preprocessing	None	$O(m)$	$O(m)$
Multi-pattern	Run k times	Natural extension	Run k times**
Implementation complexity	Trivial	Moderate	Moderate

* Over random text with a large alphabet.

** The Aho-Corasick algorithm extends KMP to multi-pattern matching in $O(n + m_1 + ... + m_k)$ time.

In practice:

- For short patterns or one-off searches, the naive algorithm is often the fastest due to its simplicity and cache-friendliness. Most standard library `indexOf` implementations use optimized variants of the naive approach (with heuristics like Boyer-Moore's bad-character rule).
- Rabin-Karp shines when searching for **multiple patterns simultaneously** or when the alphabet is small and patterns are long (making hashing effective).
- KMP is the right choice when **worst-case guarantees** matter (e.g., processing untrusted input where an adversary might craft pathological text/pattern combinations).

22.5.1 Beyond this chapter

The string matching algorithms presented here search for exact occurrences of a fixed pattern. Important extensions include:

- **Boyer-Moore** and its variants (bad-character and good-suffix heuristics): often the fastest in practice for single-pattern search on natural language text, achieving sublinear average time.
- **Aho-Corasick**: extends KMP to match multiple patterns simultaneously by building a trie of patterns augmented with failure links.
- **Suffix arrays and suffix trees** (introduced in Chapter 19): preprocess the text rather than the pattern, enabling $O(m)$ or $O(m + \log n)$ queries after $O(n)$ or $O(n \log n)$ construction.
- **Approximate matching**: finding occurrences that are within a given edit distance of the pattern, which connects to the dynamic programming techniques of Chapter 16.

22.6 Exercises

Exercise 20.1. Trace the naive string matching algorithm on $T =$ `aabaabaaab` and $P =$ `aab`. Count the total number of character comparisons. Then trace KMP on the same input and count comparisons. By what factor does KMP reduce the work?

Exercise 20.2. Compute the failure function for the pattern `aabaabaaa`. Show the π table and trace through the computation step by step. Verify your answer by checking that each $\pi[j]$ correctly identifies the longest proper prefix of $P[0 \ldots j]$ that is also a suffix.

Exercise 20.3. The Rabin-Karp algorithm uses a prime modulus q to reduce hash collisions. What happens if q is too small? Construct a concrete example where T and P consist of different characters but produce the same hash for every window when $q = 3$ and $d = 2$. How does the algorithm handle this situation?

Exercise 20.4. Modify the KMP algorithm to find only the **first** occurrence of the pattern and return immediately. Then modify it to find the **last** occurrence. What are the time complexities of your modified versions?

Exercise 20.5. A **circular string** is one where the end wraps around to the beginning: the circular string `abcd` contains the substring `dab`. Describe how to use any of the string matching algorithms in this chapter to search for a pattern in a circular string of length n. What is the time complexity?

(*Hint: consider searching in $T \| T$ — the text concatenated with itself — but be*

careful about reporting duplicate matches.)

22.7 Summary

The string matching problem — finding all occurrences of a pattern P of length m in a text T of length n — admits several algorithmic approaches.

The **naive algorithm** checks each of the $n - m + 1$ possible shifts by comparing characters one by one, taking $O(nm)$ time in the worst case. It requires no preprocessing and no extra space, making it suitable for short patterns or large alphabets where mismatches occur quickly.

The **Rabin-Karp algorithm** improves on the naive approach by using a rolling hash to filter out non-matching shifts in $O(1)$ time each. Only when hashes match does it verify character by character. With a good hash function, the expected running time is $O(n + m)$, though the worst case remains $O(nm)$. Its main strength is easy extension to multi-pattern search.

The **Knuth-Morris-Pratt algorithm** achieves $O(n+m)$ time in the worst case by preprocessing the pattern into a failure function that encodes its self-similarity structure. When a mismatch occurs, the failure function determines exactly how far to shift the pattern without missing any potential matches and without re-examining any text characters. The failure function computation and the search each use an elegant amortized argument: a counter that increases by at most 1 per step and decreases on fallbacks, bounding the total work.

These three algorithms illustrate a progression of ideas — from brute force to hashing to finite automaton-like preprocessing — that recur throughout algorithm design. The choice among them in practice depends on the use case: naive for simplicity, Rabin-Karp for multi-pattern search, and KMP when worst-case guarantees matter.

Chapter 23

Complexity Classes and NP-Completeness

Throughout this book we have analyzed algorithms by their running time as a function of input size: $O(n \log n)$ for merge sort, $O(V + E)$ for BFS, $O(nW)$ for knapsack. An implicit assumption has been that every problem we studied has an efficient — polynomial-time — solution. But not all problems do. Some of the most natural and practically important computational problems appear to resist all attempts at efficient solution. In this chapter we develop the theoretical framework of complexity classes — P, NP, and co-NP — that categorizes problems by the computational resources they require. We then introduce the concept of NP-completeness, which identifies a class of problems that are, in a precise sense, the "hardest" problems in NP. Understanding this theory is essential for every computer scientist: it tells us when to stop searching for an efficient algorithm and instead reach for approximation, heuristics, or special-case solutions.

23.1 Decision problems and languages

Complexity theory is formalized in terms of **decision problems** — problems with a yes/no answer. While this may seem restrictive, optimization problems can always be rephrased as decision problems. For example:

- **Optimization:** Find the shortest Hamiltonian cycle (TSP).
- **Decision:** Is there a Hamiltonian cycle of length $\leq k$?

If we can solve the decision version efficiently, we can typically solve the optimization version by binary searching on k.

Formally, a decision problem corresponds to a **language** $L \subseteq \{0, 1\}^*$: the set of all binary strings (encodings of inputs) for which the answer is "yes." An algorithm

decides L if, given any input x, it correctly outputs "yes" if $x \in L$ and "no" if $x \notin L$.

23.2 The class P

Definition. P is the class of decision problems solvable by a deterministic Turing machine in time polynomial in the input size n:

$$\mathbf{P} = \bigcup_{k \geq 0} \text{DTIME}(n^k)$$

In practical terms, a problem is in P if there exists an algorithm that solves every instance of size n in $O(n^k)$ time for some constant k.

Almost every algorithm in this book solves a problem in P:

Problem	Algorithm	Time
Sorting	Merge sort	$O(n \log n)$
Shortest path	Dijkstra	$O((V + E) \log V)$
MST	Kruskal	$O(E \log E)$
Maximum flow	Edmonds-Karp	$O(V E^2)$
String matching	KMP	$O(n + m)$

P captures the intuitive notion of "efficiently solvable." While $O(n^{100})$ is technically polynomial, in practice all known polynomial algorithms for natural problems have small exponents.

23.3 The class NP

Definition. NP (Nondeterministic Polynomial time) is the class of decision problems for which a "yes" answer can be **verified** in polynomial time given an appropriate **certificate** (also called a witness).

More precisely, a language L is in NP if there exists a polynomial-time verifier V and a polynomial p such that:

$$x \in L \iff \exists\, c \text{ with } |c| \leq p(|x|) \text{ such that } V(x, c) = \text{yes}$$

The certificate c is a "proof" that x is a yes-instance, and V checks this proof in polynomial time.

Key point: NP does **not** stand for "not polynomial." It stands for **nondeterministic** polynomial time. A nondeterministic machine can "guess" the certificate and verify it in polynomial time.

23.3.1 Examples

Problem	Certificate	Verification
HAMILTONIAN CYCLE	A permutation of vertices	Check it forms a valid cycle: $O(V)$
SUBSET SUM	A subset of numbers	Check the sum equals the target: $O(n)$
SAT	A truth assignment	Evaluate the formula: $O(n)$
GRAPH COLORING	A color assignment	Check no adjacent vertices share a color: $O(V + E)$
CLIQUE	A set of k vertices	Check all pairs are adjacent: $O(k^2)$

23.3.2 P $\subseteq$ NP

Every problem in P is also in NP. If we can solve a problem in polynomial time, we can certainly verify a "yes" answer in polynomial time — we simply ignore the certificate and solve the problem from scratch. The deep open question is whether the converse holds.

23.4 The class co-NP

Definition. co-NP is the class of decision problems whose **complement** is in NP. Equivalently, a problem is in co-NP if "no" answers can be verified in polynomial time.

For example, "Is this formula unsatisfiable?" is in co-NP: if the formula is satisfiable, a satisfying assignment serves as a short certificate for a "no" answer to the unsatisfiability question. But proving unsatisfiability — providing a certificate that no satisfying assignment exists — appears to require exponential-length proofs in general.

It is known that **P** $\subseteq$ **NP** $\cap$ **co-NP**. Whether **NP** = **co-NP** is another major open question in complexity theory.

23.5 The P versus NP question

The most famous open problem in theoretical computer science — and one of the seven Clay Millennium Prize Problems — asks:

Is P = NP?

If P = NP, then every problem whose solution can be efficiently verified can also be efficiently solved. This would have profound consequences: public-key cryptography would be broken, many optimization problems in logistics, biology, and AI would become tractable, and mathematical proof search would be automatable.

Most researchers believe **P ≠ NP**, based on decades of failed attempts to find polynomial algorithms for NP-complete problems. But a proof remains elusive.

23.6 Polynomial-time reductions

To compare the difficulty of problems, we use **polynomial-time reductions**.

Definition. A polynomial-time reduction from problem A to problem B (written $A \leq_P B$) is a polynomial-time computable function f such that for all inputs x:

$$x \in A \iff f(x) \in B$$

If $A \leq_P B$, then B is "at least as hard as" A:

- If B is in P, then A is in P (we can solve A by reducing to B and solving B).
- If A is not in P, then B is not in P either.

Reductions are **transitive**: if $A \leq_P B$ and $B \leq_P C$, then $A \leq_P C$.

23.7 NP-completeness

Definition. A problem B is **NP-hard** if every problem A in NP satisfies $A \leq_P B$.

Definition. A problem B is **NP-complete** if: 1. $B \in$ **NP**, and 2. B is NP-hard.

NP-complete problems are the "hardest" problems in NP: if any one of them can be solved in polynomial time, then **every** problem in NP can be solved in polynomial time, and P = NP.

23.7.1 The Cook-Levin theorem

The foundational result in NP-completeness theory is:

Theorem (Cook 1971, Levin 1973). The Boolean satisfiability problem (SAT) is NP-complete.

SAT: Given a Boolean formula ϕ in conjunctive normal form (CNF), is there a truth assignment to its variables that makes ϕ true?

The proof (which we state without proving) shows that any computation of a nondeterministic Turing machine can be encoded as a Boolean formula in polynomial time. This means SAT is universal — every NP problem reduces to it.

Once SAT was shown to be NP-complete, the floodgates opened. Proving that a new problem B is NP-complete requires just two steps:

1. Show $B \in \mathbf{NP}$ (exhibit a polynomial-time verifier).
2. Show that some known NP-complete problem A reduces to B: $A \leq_P B$.

By transitivity, this means every NP problem reduces to B.

23.8 Classic NP-complete problems

Thousands of problems have been shown to be NP-complete. Here are some of the most important, organized by domain.

23.8.1 Boolean satisfiability

SAT. Given a CNF formula (conjunction of clauses, each a disjunction of literals), is it satisfiable?

3-SAT. A restriction of SAT where each clause has exactly 3 literals. Despite the restriction, 3-SAT remains NP-complete (SAT reduces to 3-SAT by clause splitting). 3-SAT is the starting point for most NP-completeness reductions because its structure is simple yet expressive.

Note that **2-SAT** is in P — it can be solved in linear time using strongly connected components. The jump from 2 to 3 literals per clause is where tractability breaks down.

23.8.2 Graph problems

VERTEX COVER. Given a graph $G = (V, E)$ and an integer k, is there a set $S \subseteq V$ with $|S| \leq k$ such that every edge has at least one endpoint in S?

INDEPENDENT SET. Given G and k, is there a set $S \subseteq V$ with $|S| \geq k$ such that no two vertices in S are adjacent? (Complement of vertex cover: S is independent $\iff$ $V \setminus S$ is a vertex cover.)

CLIQUE. Given G and k, does G contain a complete subgraph on k vertices?

HAMILTONIAN CYCLE. Given G, does it contain a cycle that visits every vertex exactly once?

GRAPH COLORING. Given G and k, can the vertices be colored with k colors so that no two adjacent vertices share a color? NP-complete for $k \geq 3$.

23.8.3 Numeric problems

SUBSET SUM. Given a set S of integers and a target t, is there a subset of S that sums to exactly t?

PARTITION. Given a multiset of integers, can it be partitioned into two subsets with equal sum? (A special case of subset sum with $t = \text{total}/2$.)

BIN PACKING. Given items of various sizes and bins of capacity C, can all items be packed into k bins?

23.8.4 Optimization problems (decision versions)

TRAVELING SALESMAN (TSP). Given a complete weighted graph and a bound B, is there a Hamiltonian cycle of total weight $\leq B$?

SET COVER. Given a universe U, a collection of subsets $S_1, \ldots, S_m$, and an integer k, is there a sub-collection of $\leq k$ sets whose union is U?

23.9 Proving NP-completeness by reduction: a worked example

We prove that **VERTEX COVER** is NP-complete by reducing from 3-SAT.

23.9.1 Step 1: VERTEX COVER is in NP

Certificate: A set S of at most k vertices. **Verification:** Check $|S| \leq k$ and that every edge $(u, v) \in E$ has $u \in S$ or $v \in S$. This takes $O(V + E)$ time. ✓

23.9.2 Step 2: 3-SAT $\leq_P$ VERTEX COVER

Given a 3-SAT formula ϕ with n variables $x_1, \ldots, x_n$ and m clauses $C_1, \ldots, C_m$, we construct a graph G and a number k such that:

$$\phi \text{ is satisfiable} \iff G \text{ has a vertex cover of size } \leq k$$

Construction:

1. **Variable gadgets.** For each variable x_i, create two vertices x_i and $\overline{x}_i$ connected by an edge. Any vertex cover must include at least one of $\{x_i, \overline{x}_i\}$ — this models the truth assignment.
2. **Clause gadgets.** For each clause $C_j = (\ell_a \vee \ell_b \vee \ell_c)$, create a triangle on three new vertices a_j, b_j, c_j. Any vertex cover must include at least 2 of these 3 vertices.
3. **Connection edges.** Connect a_j to the vertex representing literal ℓ_a (that is, x_i if $\ell_a = x_i$, or $\overline{x}_i$ if $\ell_a = \overline{x}_i$). Similarly for b_j and c_j.
4. **Set $k = n + 2m$.**

Correctness (sketch):

- ($\Rightarrow$) If ϕ is satisfiable, pick the true literal from each variable gadget (n vertices), and for each clause triangle, pick the 2 vertices whose connection edges lead to **false** literals (which are not in the cover from step 1). This gives a vertex cover of size $n + 2m$.
- ($\Leftarrow$) If G has a vertex cover of size $\leq k = n + 2m$, then exactly 1 vertex per variable gadget and exactly 2 per clause triangle are chosen (since we need at least $n + 2m$). The vertex not covered in each clause triangle must have its connection edge covered by the variable-gadget vertex — meaning the corresponding literal is true. So ϕ is satisfiable.

The construction takes polynomial time (the graph has $2n + 3m$ vertices and $n + 6m$ edges), so this is a valid polynomial-time reduction. Since 3-SAT is NP-complete and reduces to VERTEX COVER, and VERTEX COVER is in NP, VERTEX COVER is NP-complete. □

23.10 The reduction landscape

Many NP-completeness proofs follow chains of reductions from SAT or 3-SAT:

```
SAT
 └─→ 3-SAT
      ├─→ INDEPENDENT SET ──→ CLIQUE
      ├─→ VERTEX COVER
      ├─→ HAMILTONIAN CYCLE ──→ TSP
      ├─→ SUBSET SUM ──→ PARTITION ──→ BIN PACKING
      ├─→ GRAPH COLORING
      └─→ SET COVER
```

Each arrow represents a polynomial-time reduction. The diversity of these problems — spanning logic, graphs, numbers, and optimization — is what makes

NP-completeness so remarkable: all these seemingly unrelated problems are computationally equivalent.

23.11 Brute-force illustrations

To make the exponential nature of NP-complete problems concrete, we implement brute-force solvers for two classic problems. These are educational implementations — they work correctly but have exponential running times that make them impractical for large inputs.

23.11.1 Subset sum (brute force)

The brute-force approach enumerates all 2^n subsets of the input set and checks whether any of them sums to the target.

Algorithm:

```
SUBSET-SUM-BRUTE(S, t):
    n ← |S|
    for mask ← 1 to 2^n − 1:
        sum ← 0
        subset ← ∅
        for i ← 0 to n − 1:
            if bit i of mask is set:
                sum ← sum + S[i]
                add S[i] to subset
        if sum = t:
            return (true, subset)
    return (false, ∅)
```

Implementation:

```typescript
export interface SubsetSumResult {
  found: boolean;
  subset: number[];
}

export function subsetSum(
  nums: readonly number[],
  target: number,
): SubsetSumResult {
  const n = nums.length;

  if (n > 30) {
```

```
    throw new RangeError(
      `input size ${n} is too large for brute-force enumeration (max 30)`,
    );
  }

  if (target === 0) {
    return { found: true, subset: [] };
  }

  const total = 1 << n;

  for (let mask = 1; mask < total; mask++) {
    let sum = 0;
    const subset: number[] = [];

    for (let i = 0; i < n; i++) {
      if (mask & (1 << i)) {
        sum += nums[i]!;
        subset.push(nums[i]!);
      }
    }

    if (sum === target) {
      return { found: true, subset };
    }
  }

  return { found: false, subset: [] };
}
```

Complexity:

- **Time:** $O(2^n \cdot n)$. There are 2^n subsets, and summing each takes $O(n)$.
- **Space:** $O(n)$ for the current subset.

Note that the dynamic programming approach from Chapter 16 can solve subset sum in $O(n \cdot t)$ time when t is bounded. However, $O(n \cdot t)$ is **pseudo-polynomial** — polynomial in the numeric value of t, not in the number of bits needed to encode t. The subset sum problem remains NP-complete because the target t can be exponentially large relative to the input length.

23.11.2 Traveling salesman (brute force)

The brute-force TSP solver generates all $(n-1)!$ permutations of cities (fixing the starting city) and evaluates each tour.

Algorithm:

```
TSP-BRUTE(dist[0..n-1][0..n-1]):
    bestDist ← ∞
    bestTour ← nil
    for each permutation π of {1, 2, ..., n-1}:
        cost ← dist[0][π[0]]
        for i ← 0 to n − 3:
            cost ← cost + dist[π[i]][π[i+1]]
        cost ← cost + dist[π[n−2]][0]
        if cost < bestDist:
            bestDist ← cost
            bestTour ← (0, π[0], ..., π[n−2])
    return (bestTour, bestDist)
```

Implementation:

```typescript
export type DistanceMatrix = readonly (readonly number[])[];

export interface TSPResult {
  tour: number[];
  distance: number;
}

export function tspBruteForce(dist: DistanceMatrix): TSPResult {
  const n = dist.length;

  if (n === 0) {
    throw new RangeError('distance matrix must not be empty');
  }
  if (n > 12) {
    throw new RangeError(
      `input size ${n} is too large for brute-force TSP (max 12)`,
    );
  }

  if (n === 1) return { tour: [0], distance: 0 };
  if (n === 2) {
    return { tour: [0, 1], distance: dist[0]![1]! + dist[1]![0]! };
```

```
  }

  const remaining = Array.from({ length: n - 1 }, (_, i) => i + 1);
  let bestDistance = Infinity;
  let bestTour: number[] = [];

  function tourCost(perm: number[]): number {
    let cost = dist[0]![perm[0]!]!;
    for (let i = 0; i < perm.length - 1; i++) {
      cost += dist[perm[i]!]![perm[i + 1]!]!;
    }
    cost += dist[perm[perm.length - 1]!]![0]!;
    return cost;
  }

  function heapPermute(arr: number[], size: number): void {
    if (size === 1) {
      const cost = tourCost(arr);
      if (cost < bestDistance) {
        bestDistance = cost;
        bestTour = [0, ...arr];
      }
      return;
    }
    for (let i = 0; i < size; i++) {
      heapPermute(arr, size - 1);
      const swapIdx = size % 2 === 0 ? i : 0;
      const temp = arr[swapIdx]!;
      arr[swapIdx] = arr[size - 1]!;
      arr[size - 1] = temp;
    }
  }

  heapPermute(remaining, remaining.length);
  return { tour: bestTour, distance: bestDistance };
}
```

Complexity:

- **Time:** $O(n!)$. We fix city 0 and generate all $(n-1)!$ permutations of the remaining cities. Each permutation requires $O(n)$ to evaluate, giving $O(n \cdot (n-1)!) = O(n!)$ total.

- **Space:** $O(n)$ for the recursion stack and current permutation.

The factorial growth makes this approach completely impractical beyond about 12–15 cities:

n	$(n-1)!$ permutations
5	24
8	5,040
10	362,880
12	39,916,800
15	87,178,291,200
20	$\approx 1.2 \times 10^{17}$

For practical TSP instances (hundreds or thousands of cities), we need approximation algorithms (Chapter 22), branch-and-bound, or metaheuristics like simulated annealing and genetic algorithms.

23.12 Coping with NP-hardness

When faced with an NP-hard problem, giving up is not the answer. Several strategies can yield useful solutions:

23.12.1 1. Approximation algorithms

Accept a solution that is provably close to optimal. For example:

- **Vertex cover:** A simple greedy algorithm achieves a 2-approximation — it always finds a cover at most twice the size of the optimum (Chapter 22).
- **Metric TSP:** An MST-based algorithm achieves a 2-approximation when the triangle inequality holds (Chapter 22).
- **Set cover:** A greedy algorithm achieves an $O(\log n)$-approximation (Chapter 22).

The key advantage is a **guaranteed approximation ratio** — we know how far from optimal the solution can be.

23.12.2 2. Exact algorithms for special cases

Many NP-hard problems become tractable for restricted inputs:

- **TSP on planar graphs** can be solved in $O(2^{O(\sqrt{n})} \cdot n)$ time.
- **Vertex cover** parameterized by k can be solved in $O(2^k \cdot n)$ time (fixed-parameter tractable).

- **2-SAT** is solvable in linear time, even though 3-SAT is NP-complete.
- **Tree-width bounded graphs** admit polynomial-time algorithms for many NP-hard problems.

23.12.3 3. Pseudo-polynomial algorithms

Problems like subset sum and knapsack have algorithms running in $O(n \cdot W)$ time, where W is a numeric parameter. When W is small relative to n, these algorithms are practical despite the problem's NP-completeness. See the dynamic programming chapter (Chapter 16) for implementations.

23.12.4 4. Heuristics and metaheuristics

When provable guarantees are not needed, heuristic methods often find good solutions quickly:

- **Local search:** Start with a random solution and iteratively improve it by making small changes (e.g., 2-opt for TSP, which swaps pairs of edges).
- **Simulated annealing:** Like local search, but occasionally accepts worse solutions to escape local optima, with the probability of acceptance decreasing over time.
- **Genetic algorithms:** Maintain a population of solutions, combine them via crossover, and apply mutation to explore the search space.
- **Branch and bound:** Systematically explore the solution space, pruning branches that provably cannot improve on the best solution found so far.

23.12.5 5. Randomized algorithms

Randomization can sometimes break through worst-case barriers:

- **Random sampling** can quickly find satisfying assignments for SAT instances that are not too constrained.
- **Randomized rounding** of linear programming relaxations yields good approximations for many NP-hard problems.

23.13 Summary of complexity classes

Class	Informal definition	Examples
P	Efficiently solvable (polynomial time)	Sorting, shortest path, MST, max flow

Class	Informal definition	Examples
NP	Efficiently verifiable (polynomial-time certificate for “yes”)	SAT, TSP, subset sum, clique, coloring
co-NP	Efficiently verifiable “no” answers	Tautology, primality (also in P)
NP-complete	Hardest problems in NP (every NP problem reduces to them)	3-SAT, vertex cover, TSP, subset sum
NP-hard	At least as hard as NP-complete (but may not be in NP)	Halting problem, optimal chess play

Relationships: $\mathbf{P} \subseteq \mathbf{NP} \cap \mathbf{co\text{-}NP} \subseteq \mathbf{NP} \cup \mathbf{co\text{-}NP}$.

Whether any of these inclusions are strict is unknown (except that NP-hard $\not\subseteq$ NP, since NP-hard includes undecidable problems).

23.14 Exercises

1. **NP membership.** Show that the CLIQUE problem is in NP by describing a certificate and a polynomial-time verifier. What is the running time of your verifier?

2. **Reduction practice.** Prove that INDEPENDENT SET is NP-complete by reducing from VERTEX COVER. (Hint: S is an independent set in G if and only if $V \setminus S$ is a vertex cover.)

3. **Subset sum variants.** The PARTITION problem asks whether a multiset of integers can be divided into two subsets of equal sum. Show that PARTITION is NP-complete by reducing from SUBSET SUM. (Hint: given a SUBSET SUM instance (S, t), construct a PARTITION instance by adding appropriate elements.)

4. **Pseudo-polynomial vs polynomial.** Explain why the $O(nW)$ dynamic programming algorithm for 0/1 knapsack does not prove P = NP, even though knapsack is NP-complete. What is the relationship between W and the input size?

5. **Brute-force analysis.** Suppose you have a computer that can evaluate 10^9 TSP tours per second. How long would it take to solve a 20-city instance by brute force? A 25-city instance? Express your answers in meaningful time units (seconds, years, etc.).

23.15 Chapter summary

This chapter introduced the theoretical framework for classifying computational problems by their inherent difficulty.

P contains problems solvable in polynomial time — the "efficiently solvable" problems that have been our focus throughout this book. **NP** contains problems whose solutions can be verified in polynomial time, even if finding a solution may be hard. The question of whether P = NP — whether efficient verification implies efficient solution — is the most important open problem in computer science.

NP-complete problems, identified through polynomial-time reductions, are the hardest problems in NP: solving any one of them efficiently would solve all of them. The Cook-Levin theorem established SAT as the first NP-complete problem, and thousands more have been identified through chains of reductions — from satisfiability to graph problems (vertex cover, clique, Hamiltonian cycle), to numeric problems (subset sum, partition), to optimization problems (TSP, set cover).

We implemented brute-force solvers for two NP-complete problems to illustrate their exponential nature:

- **Subset sum** by exhaustive enumeration of all 2^n subsets: $O(2^n \cdot n)$ time.
- **TSP** by exhaustive enumeration of all $(n-1)!$ permutations: $O(n!)$ time.

When facing NP-hard problems in practice, we have several coping strategies: **approximation algorithms** with provable guarantees (Chapter 22), **exact algorithms for special cases** (e.g., fixed-parameter tractability, bounded treewidth), **pseudo-polynomial algorithms** (e.g., DP for knapsack when the target is small), and **heuristics** (local search, simulated annealing, genetic algorithms). The theory of NP-completeness tells us not that these problems are unsolvable, but that we should not expect a polynomial-time algorithm that works optimally on all instances — and guides us toward the right tool for each situation.

Chapter 24

Approximation Algorithms

Throughout this book we have designed algorithms that solve problems exactly and efficiently. But in the previous chapter we saw that many important optimization problems — minimum vertex cover, set cover, traveling salesman — are NP-hard: no polynomial-time algorithm is known, and most researchers believe none exists. Approximation algorithms offer a powerful middle ground: polynomial-time algorithms that produce solutions provably close to optimal. Instead of finding the best solution, we settle for one that is guaranteed to be within a known factor of the best. In this chapter we formalize approximation ratios, then study three classical algorithms: a 2-approximation for vertex cover, a greedy $O(\log n)$-approximation for set cover, and a 2-approximation for metric TSP via minimum spanning trees.

24.1 When exact solutions are infeasible

Chapter 21 demonstrated that brute-force approaches to NP-hard problems are impractical for all but the smallest inputs. A brute-force TSP solver exhausts $(n-1)!$ permutations, which is infeasible beyond about 12–15 cities. A brute-force subset sum examines 2^n subsets, limiting us to roughly 30 elements.

For real-world instances — routing delivery trucks through hundreds of stops, selecting facilities to cover a service region, or allocating resources across a network — we need algorithms that:

1. **Run in polynomial time** (ideally $O(n^2)$ or $O(n^3)$, not $O(2^n)$).
2. **Provide a quality guarantee** — we can bound how far the solution is from optimal.

Approximation algorithms deliver both.

24.2 Approximation ratios

Let $\mathcal{A}$ be a polynomial-time algorithm for an optimization problem, and let $\text{OPT}(I)$ denote the cost of an optimal solution for instance I.

Definition. Algorithm $\mathcal{A}$ has **approximation ratio** $\rho(n)$ if, for every instance I of size n:

$$\max\left(\frac{\mathcal{A}(I)}{\text{OPT}(I)}, \frac{\text{OPT}(I)}{\mathcal{A}(I)}\right) \leq \rho(n)$$

The ratio is always ≥ 1. For minimization problems, $\mathcal{A}(I) \leq \rho(n) \cdot \text{OPT}(I)$. For maximization problems, $\mathcal{A}(I) \geq \text{OPT}(I)/\rho(n)$.

An algorithm with approximation ratio ρ is called a **ρ-approximation algorithm**.

Some important distinctions:

- A **constant-factor approximation** has $\rho(n) = c$ for some constant c (e.g., the 2-approximation for vertex cover).
- A **logarithmic approximation** has $\rho(n) = O(\log n)$ (e.g., greedy set cover).
- A **polynomial-time approximation scheme (PTAS)** achieves ratio $1 + \epsilon$ for any constant $\epsilon > 0$, though the running time may depend on $1/\epsilon$.
- A **fully polynomial-time approximation scheme (FPTAS)** is a PTAS whose running time is polynomial in both n and $1/\epsilon$.

Not all NP-hard problems can be approximated equally well. Under standard complexity assumptions:

Problem	Best known ratio	Hardness of approximation
Vertex cover	2	Cannot do better than ≈ 1.36 unless P = NP
Set cover	$\ln n + 1$	Cannot do better than $(1 - \epsilon) \ln n$ unless P = NP

Problem	Best known ratio	Hardness of approximation
Metric TSP	1.5 (Christofides)	Cannot do better than $\frac{123}{122}$ unless P = NP
General TSP	—	No constant-factor approximation unless P = NP
MAX-3SAT	7/8	Cannot do better than $7/8 + \epsilon$ unless P = NP
Knapsack	FPTAS	Has a $(1+\epsilon)$-approximation for any $\epsilon > 0$

24.3 Vertex cover: 2-approximation

24.3.1 Problem definition

Given an undirected graph $G = (V, E)$, a **vertex cover** is a subset $C \subseteq V$ such that every edge in E has at least one endpoint in C. The **minimum vertex cover** problem asks for a cover of smallest size.

Vertex cover is one of Karp's 21 NP-complete problems (1972) and has a natural relationship to the independent set problem: S is an independent set if and only if $V \setminus S$ is a vertex cover.

24.3.2 The algorithm

The 2-approximation is elegantly simple:

1. Start with an empty cover C and the full edge set E'.
2. Pick an arbitrary uncovered edge (u, v) from E'.
3. Add **both** endpoints u and v to C.
4. Remove all edges incident to u or v from E'.
5. Repeat until E' is empty.

The key insight is that the edges we pick in step 2 form a **matching** M — a set of edges that share no endpoints. Every vertex cover must include at least one endpoint of each matching edge, so OPT $\geq |M|$. Our algorithm adds exactly 2 vertices per matching edge, giving $|C| = 2|M| \leq 2 \cdot \text{OPT}$.

24.3.3 Pseudocode

```
APPROX-VERTEX-COVER(G):
    C ← ∅
    E' ← E
    while E' ≠ ∅:
        pick any edge (u, v) ∈ E'
        C ← C ∪ {u, v}
        remove all edges incident to u or v from E'
    return C
```

24.3.4 Proof of the 2-approximation

Claim: $|C| \leq 2 \cdot \text{OPT}$.

Proof. Let M be the set of edges selected by the algorithm. By construction:

1. No two edges in M share an endpoint (each time we select an edge, we remove all incident edges). So M is a matching.
2. The algorithm adds both endpoints of each matching edge: $|C| = 2|M|$.
3. Any vertex cover must include at least one endpoint of every edge, including every edge in M. Since matching edges are disjoint, the optimal cover needs at least $|M|$ vertices: OPT $\geq |M|$.
4. Therefore $|C| = 2|M| \leq 2 \cdot \text{OPT}$. □

24.3.5 TypeScript implementation

```ts
import { Graph } from '../12-graphs-and-traversal/graph.js';

export interface VertexCoverResult<T> {
  cover: Set<T>;
  size: number;
}

export function vertexCover<T>(graph: Graph<T>): VertexCoverResult<T> {
  if (graph.directed) {
    throw new Error('Vertex cover requires an undirected graph');
  }
```

```
  const cover = new Set<T>();
  const edges = graph.getEdges();

  for (const edge of edges) {
    // If neither endpoint is already covered, add both.
    if (!cover.has(edge.from) && !cover.has(edge.to)) {
      cover.add(edge.from);
      cover.add(edge.to);
    }
  }

  return { cover, size: cover.size };
}
```

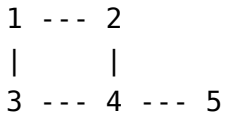
Note that the implementation iterates over edges and skips any edge that already has a covered endpoint — this is equivalent to "removing incident edges" in the pseudocode, since we only select an edge when both endpoints are uncovered.

Complexity:

- **Time:** $O(V + E)$ — we iterate over all edges once.
- **Space:** $O(V + E)$ — for the edge list and the cover set.

24.3.6 Worked example

Consider this graph:



Edges: (1,2), (1,3), (2,4), (3,4), (4,5).

Suppose the algorithm processes edges in order:

1. Pick (1,2): add 1 and 2 to C. Remove (1,2), (1,3), (2,4).
2. Remaining edges: (3,4), (4,5). Pick (3,4): add 3 and 4 to C. Remove (3,4), (4,5).
3. No edges remain. $C = \{1, 2, 3, 4\}$, $|C| = 4$.

The matching was $M = \{(1, 2), (3, 4)\}$, so $|M| = 2$.

The optimal cover is $\{2, 4\}$ or $\{1, 4\}$ with OPT = 2. Our algorithm returned $|C| = 4 = 2 \cdot \text{OPT}$, which is exactly the worst case of the 2-approximation guarantee.

24.3.7 Tightness of the bound

The factor of 2 is tight for this algorithm. Consider the complete bipartite graph $K_{n,n}$ with n vertices on each side. The optimal vertex cover selects one side: OPT $= n$. A maximal matching has n edges (one from each left vertex to a right vertex), and the algorithm adds both endpoints: $|C| = 2n = 2 \cdot \text{OPT}$.

Whether vertex cover can be approximated with a ratio better than 2 in polynomial time is a major open problem. The best known lower bound (assuming the Unique Games Conjecture) is $2 - \epsilon$ for any $\epsilon > 0$.

24.4 Greedy set cover: $O(\log n)$-approximation

24.4.1 Problem definition

Given a universe $U = \{e_1, e_2, \ldots, e_n\}$ and a collection $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ of subsets of U whose union is U, the **set cover** problem asks for the smallest subcollection of $\mathcal{S}$ that covers every element of U.

Set cover is a fundamental NP-hard problem that generalizes vertex cover (each vertex corresponds to a "set" of its incident edges, and the universe is the edge set).

24.4.2 The greedy algorithm

The greedy strategy is intuitive: at each step, select the subset that covers the most currently-uncovered elements.

```
GREEDY-SET-COVER(U, S):
    C ← ∅                    // selected subsets
    uncovered ← U
    while uncovered ≠ ∅:
        select S_i ∈ S maximizing |S_i ∩ uncovered|
        C ← C ∪ {S_i}
        uncovered ← uncovered \ S_i
    return C
```

24.4.3 Proof of the $O(\log n)$-approximation

Theorem. The greedy algorithm produces a cover of size at most $H(\max_i |S_i|) \cdot \text{OPT}$, where $H(k) = 1 + \frac{1}{2} + \cdots + \frac{1}{k} \leq \ln k + 1$ is the k-th harmonic number.

Proof sketch. We use a charging argument. When the greedy algorithm selects a set S_i that covers k new elements, we "charge" each newly covered element a cost of $\frac{1}{k}$.

Consider any element e that was covered when t elements remained uncovered. The greedy choice covers at least t/OPT elements (because the optimal solution uses OPT sets to cover everything, so by pigeonhole, some set covers at least t/OPT of the remaining elements). So element e's charge is at most OPT/t.

Summing over all elements in the order they were covered:

$$\text{Greedy cost} \le \sum_{t=1}^{n} \frac{\mathrm{OPT}}{t} = \mathrm{OPT} \cdot H(n) \le \mathrm{OPT} \cdot (\ln n + 1) \quad \square$$

24.4.4 TypeScript implementation

```ts
export interface SetCoverResult<T> {
  selectedIndices: number[];
  selectedSets: ReadonlySet<T>[];
  count: number;
}

export function setCover<T>(
  universe: ReadonlySet<T>,
  subsets: readonly ReadonlySet<T>[],
): SetCoverResult<T> {
  if (universe.size === 0) {
    return { selectedIndices: [], selectedSets: [], count: 0 };
  }

  const uncovered = new Set<T>(universe);
  const selectedIndices: number[] = [];
  const selectedSets: ReadonlySet<T>[] = [];
  const used = new Set<number>();

  while (uncovered.size > 0) {
    let bestIndex = -1;
    let bestCount = 0;

    for (let i = 0; i < subsets.length; i++) {
      if (used.has(i)) continue;
      let count = 0;
      for (const elem of subsets[i]!) {
        if (uncovered.has(elem)) count++;
      }
      if (count > bestCount) {
```

```
        bestCount = count;
        bestIndex = i;
      }
    }

    if (bestIndex === -1 || bestCount === 0) {
      throw new Error(
        'Subsets do not cover the entire universe; ' +
          `${uncovered.size} element(s) remain uncovered`,
      );
    }

    used.add(bestIndex);
    selectedIndices.push(bestIndex);
    selectedSets.push(subsets[bestIndex]!);

    for (const elem of subsets[bestIndex]!) {
      uncovered.delete(elem);
    }
  }

  return { selectedIndices, selectedSets, count: selectedIndices.length };
}
```

Complexity:

- **Time:** $O(|U| \cdot |\mathcal{S}| \cdot \max_i |S_i|)$ in the worst case. Each of the at most $|\mathcal{S}|$ iterations scans all subsets, and each scan examines up to $\max_i |S_i|$ elements.
- **Space:** $O(|U| + \sum_i |S_i|)$.

24.4.5 Worked example

Universe: $U = \{1, 2, 3, 4, 5, 6\}$

Subsets:

Set	Elements
S_1	$\{1, 2, 3\}$
S_2	$\{2, 4\}$
S_3	$\{3, 4, 5\}$
S_4	$\{5, 6\}$

Iteration 1: Uncovered = $\{1, 2, 3, 4, 5, 6\}$.

- S_1 covers 3 elements, S_2 covers 2, S_3 covers 3, S_4 covers 2.
- Tie between S_1 and S_3; pick S_1.
- Uncovered = $\{4, 5, 6\}$.

Iteration 2: S_2 covers 1 ($\{4\}$), S_3 covers 2 ($\{4, 5\}$), S_4 covers 2 ($\{5, 6\}$). Pick S_3.

- Uncovered = $\{6\}$.

Iteration 3: S_2 covers 0, S_4 covers 1 ($\{6\}$). Pick S_4.

- Uncovered = $\emptyset$.

Result: $\{S_1, S_3, S_4\}$, 3 subsets. The optimal solution is also 3 (e.g., $\{S_1, S_2, S_4\}$), so the greedy algorithm found an optimal solution in this case.

24.4.6 Optimality of the greedy bound

The $O(\log n)$ approximation ratio is essentially the best possible for set cover. Under standard complexity assumptions, no polynomial-time algorithm can achieve a ratio better than $(1 - \epsilon) \ln n$ for any $\epsilon > 0$.

24.5 Metric TSP: 2-approximation via MST

24.5.1 Problem definition

The **Traveling Salesman Problem (TSP)** asks for the shortest Hamiltonian cycle (a tour visiting every vertex exactly once and returning to the start) in a complete weighted graph.

General TSP is not only NP-hard but also inapproximable: no polynomial-time algorithm can achieve any constant approximation ratio unless P = NP. (The proof: if we could approximate within any factor ρ, we could solve the NP-complete Hamiltonian cycle problem by assigning weight 1 to existing edges and weight $\rho n + 1$ to missing edges.)

However, many practical TSP instances satisfy the **triangle inequality**: for all vertices u, v, w:

$$d(u, w) \leq d(u, v) + d(v, w)$$

This holds for Euclidean distances, shortest-path distances in networks, and most other natural distance metrics. The resulting **metric TSP** admits constant-factor approximations.

24.5.2 The MST-based algorithm

The algorithm exploits a fundamental relationship between MSTs and optimal tours:

1. **Compute an MST** of the complete graph.
2. **Perform a DFS preorder traversal** of the MST.
3. **The preorder sequence**, with a return edge to the start, forms the tour.

```
APPROX-METRIC-TSP(G, d):
    T ← MST(G)                              // Prim's or Kruskal's
    tour ← DFS-PREORDER(T, starting from vertex 0)
    return tour
```

24.5.3 Why this works: the shortcutting argument

Consider the **full walk** of the MST: start at the root, and traverse every edge twice (once going down, once returning). This walk visits every vertex but may visit some vertices multiple times. Its total cost is exactly $2 \cdot w(T)$, where $w(T)$ is the MST weight.

The preorder traversal is a **shortcut** of this full walk: whenever the walk would revisit an already-visited vertex, we skip directly to the next unvisited vertex. By the triangle inequality, skipping vertices can only decrease the total distance:

$$d(u, w) \leq d(u, v) + d(v, w)$$

So the shortcutted tour costs at most $2 \cdot w(T)$.

24.5.4 Proof of the 2-approximation

Claim: The MST-based tour has cost at most $2 \cdot \text{OPT}$.

Proof.

1. **MST ≤ OPT:** Removing any edge from the optimal tour yields a spanning tree. Since the MST is the minimum-weight spanning tree: $w(T) \leq \text{OPT}$.
2. **Tour ≤ 2 · MST:** The full walk costs $2 \cdot w(T)$, and the shortcutted preorder tour costs at most this (by the triangle inequality).
3. Combining: Tour $\leq 2 \cdot w(T) \leq 2 \cdot \text{OPT}$. □

24.5.5 TypeScript implementation

```
import type { DistanceMatrix } from '../21-complexity/tsp-brute-force.js';
import { Graph } from '../12-graphs-and-traversal/graph.js';
```

```
import { prim } from '../14-minimum-spanning-trees/prim.js';

export interface MetricTSPResult {
  tour: number[];
  distance: number;
}

export function metricTSP(dist: DistanceMatrix): MetricTSPResult {
  const n = dist.length;

  if (n === 0) throw new RangeError('distance matrix must not be empty');
  for (let i = 0; i < n; i++) {
    if (dist[i]!.length !== n) {
      throw new Error(
        `distance matrix must be square (row ${i} has ` +
          `${dist[i]!.length} columns, expected ${n})`,
      );
    }
  }
  if (n === 1) return { tour: [0], distance: 0 };
  if (n === 2) {
    return { tour: [0, 1], distance: dist[0]![1]! + dist[1]![0]! };
  }

  // Build a complete undirected graph.
  const graph = new Graph<number>(false);
  for (let i = 0; i < n; i++) graph.addVertex(i);
  for (let i = 0; i < n; i++) {
    for (let j = i + 1; j < n; j++) {
      graph.addEdge(i, j, dist[i]![j]!);
    }
  }

  // Step 1: Compute MST.
  const mst = prim(graph, 0);

  // Build MST adjacency list.
  const mstAdj = new Map<number, number[]>();
  for (let i = 0; i < n; i++) mstAdj.set(i, []);
  for (const edge of mst.edges) {
    mstAdj.get(edge.from)!.push(edge.to);
```

```
    mstAdj.get(edge.to)!.push(edge.from);
  }

  // Step 2: DFS preorder traversal.
  const tour: number[] = [];
  const visited = new Set<number>();

  function dfsPreorder(v: number): void {
    visited.add(v);
    tour.push(v);
    for (const neighbor of mstAdj.get(v)!) {
      if (!visited.has(neighbor)) dfsPreorder(neighbor);
    }
  }

  dfsPreorder(0);

  // Step 3: Compute tour distance.
  let distance = 0;
  for (let i = 0; i < tour.length - 1; i++) {
    distance += dist[tour[i]!]![tour[i + 1]!]!;
  }
  distance += dist[tour[tour.length - 1]!]![tour[0]!]!;

  return { tour, distance };
}
```

Complexity:

- **Time:** $O(V^2 \log V)$ — constructing the complete graph is $O(V^2)$, and Prim's algorithm on a complete graph with a binary heap is $O(E \log V) = O(V^2 \log V)$.
- **Space:** $O(V^2)$ for the adjacency list of the complete graph.

24.5.6 Worked example

Consider 4 cities at the corners of a unit square:

```
  1 -------- 2
  |          |
  |          |
  0 -------- 3
```

Distance matrix (Euclidean):

	0	1	2	3
0	0	1	$\sqrt{2}$	1
1	1	0	1	$\sqrt{2}$
2	$\sqrt{2}$	1	0	1
3	1	$\sqrt{2}$	1	0

Step 1: MST (using Prim's from vertex 0):

- Add edge 0–1 (weight 1)
- Add edge 1–2 (weight 1)
- Add edge 0–3 (weight 1)

MST weight = 3. MST edges: 0–1, 1–2, 0–3.

Step 2: DFS preorder from 0:

Visit 0 → visit 1 → visit 2 → backtrack to 1 → backtrack to 0 → visit 3 → backtrack to 0.

Preorder: [0, 1, 2, 3].

Step 3: Tour cost:

$d(0,1) + d(1,2) + d(2,3) + d(3,0) = 1 + 1 + 1 + 1 = 4$.

The optimal tour is also 4 (the perimeter of the square), so the approximation is exact in this case.

OPT = 4, MST weight = 3, and $2 \times 3 = 6 \geq 4$ — the guarantee holds.

24.5.7 Christofides' algorithm: a better bound

While we implemented the 2-approximation for its simplicity, a better algorithm exists. **Christofides' algorithm** (1976) achieves a $\frac{3}{2}$-approximation:

1. Compute an MST T.
2. Find the set O of vertices with odd degree in T.
3. Compute a minimum-weight perfect matching M on the vertices in O.
4. Combine T and M to get an Eulerian multigraph.
5. Find an Eulerian circuit.
6. Shortcut to a Hamiltonian cycle.

The key insight is that combining the MST with a minimum perfect matching on odd-degree vertices produces an Eulerian graph (all degrees even), whose Euler tour can be shortcutted. Since the minimum matching costs at most $\frac{1}{2}$OPT (by a

pairing argument on the optimal tour), the total cost is at most $w(T) + \frac{1}{2}\text{OPT} \leq \frac{3}{2}\text{OPT}$.

Christofides' algorithm remained the best known approximation for metric TSP for nearly 50 years, until a very slight improvement was achieved by Karlin, Klein, and Oveis Gharan in 2021.

24.6 Comparison of approximation algorithms

Problem	Algorithm	Ratio	Time	Approach
Vertex cover	Matching-based	2	$O(V + E)$	Pick both endpoints of a maximal matching
Set cover	Greedy	$\ln n + 1$	$O(\lvert U\rvert \cdot \lvert \mathcal{S}\rvert \cdot k)$	Pick set covering most uncovered elements
Metric TSP	MST-based	2	$O(V^2 \log V)$	MST + DFS preorder + shortcutting
Metric TSP	Christofides	1.5	$O(V^3)$	MST + minimum matching + Euler tour

24.7 Beyond the algorithms in this chapter

Approximation algorithms form a rich and active area of research. Some important topics we have not covered include:

- **LP relaxation and rounding:** Many approximation algorithms work by solving a linear programming relaxation of an integer program and then rounding the fractional solution to an integer one. This technique yields tight results for problems like weighted vertex cover and MAX-SAT.
- **Semidefinite programming:** For problems like MAX-CUT, the Goemans-Williamson algorithm uses semidefinite programming to achieve an approximation ratio of approximately 0.878, which is optimal assuming the Unique Games Conjecture.

- **Primal-dual methods:** These construct both a feasible solution and a lower bound simultaneously, useful for network design problems.
- **The PCP theorem:** The celebrated PCP (Probabilistically Checkable Proofs) theorem provides the theoretical foundation for hardness of approximation results, showing that for many problems, achieving certain approximation ratios is as hard as solving the problem exactly.

24.8 Exercises

1. **Vertex cover on trees.** Show that the minimum vertex cover of a tree can be computed exactly in polynomial time using dynamic programming. (Hint: root the tree and compute, for each vertex, the minimum cover of its subtree with and without including that vertex.) Does this contradict the NP-hardness of vertex cover?

2. **Weighted set cover.** Generalize the greedy set cover algorithm to the weighted case, where each subset S_i has a cost c_i and we want to minimize the total cost of selected subsets. Show that the greedy algorithm (pick the set with the smallest cost per newly covered element) achieves the same $O(\log n)$ approximation ratio.

3. **TSP triangle inequality failure.** Construct a graph with 4 vertices where the triangle inequality is violated, and show that the MST-based algorithm produces a tour whose cost exceeds $2 \cdot \text{OPT}$. Explain why the shortcutting argument fails.

4. **MAX-SAT approximation.** Consider the following simple algorithm for MAX-SAT: independently set each variable to true with probability $\frac{1}{2}$. Show that this randomized algorithm satisfies at least $\frac{m}{2}$ clauses in expectation when each clause has at least one literal, and at least $\frac{7m}{8}$ clauses when each clause has exactly 3 literals. (Here m is the number of clauses.) Can you derandomize this algorithm?

5. **Tight examples.** For each of the three algorithms in this chapter, describe a family of instances where the approximation ratio approaches the proven bound. That is: find graphs where the vertex cover algorithm returns a cover of size approaching $2 \cdot \text{OPT}$, set cover instances where the greedy algorithm uses $\Omega(\log n) \cdot \text{OPT}$ sets, and metric TSP instances where the MST tour approaches $2 \cdot \text{OPT}$.

24.9 Chapter summary

Approximation algorithms provide a principled approach to NP-hard optimization problems: polynomial-time algorithms with **provable guarantees** on solution quality.

We studied three classical examples:

- **Vertex cover 2-approximation:** Pick an arbitrary uncovered edge, add both endpoints. The selected edges form a matching, and any cover needs at least one vertex per matching edge, giving a factor-2 guarantee. Runs in $O(V + E)$ time.
- **Greedy set cover $O(\log n)$-approximation:** Repeatedly select the subset covering the most uncovered elements. A charging argument shows the greedy cost is at most $H(n)$ times optimal, where $H(n) \leq \ln n + 1$ is the harmonic number. This ratio is essentially tight: no polynomial-time algorithm can do significantly better unless P = NP.
- **Metric TSP 2-approximation via MST:** Compute a minimum spanning tree, perform a DFS preorder traversal, and return the resulting tour. The MST provides a lower bound on OPT, and the triangle inequality ensures the shortcutted tour costs at most twice the MST weight. Christofides' algorithm improves this to a $\frac{3}{2}$-approximation.

The study of approximation algorithms reveals a rich structure within NP-hard problems. Some problems (like knapsack) admit $(1 + \epsilon)$-approximations for any $\epsilon > 0$. Others (like vertex cover) admit constant-factor approximations but resist improvements below specific thresholds. Still others (like general TSP) cannot be approximated at all. Understanding where a problem falls in this landscape guides us toward the most effective algorithmic approach.

Chapter 25

Bibliography

25.1 Textbooks

- Cormen, T.H., Leiserson, C.E., Rivest, R.L., and Stein, C. *Introduction to Algorithms*, 4th edition. MIT Press, 2022. The comprehensive reference for algorithm design and analysis, commonly known as CLRS. Our curriculum and many proofs follow its presentation.
- Kleinberg, J. and Tardos, E. *Algorithm Design*. Addison-Wesley, 2005. An excellent treatment of algorithm design techniques, particularly dynamic programming, greedy algorithms, and network flow.
- Sedgewick, R. and Wayne, K. *Algorithms*, 4th edition. Addison-Wesley, 2011. A practically oriented textbook with Java implementations. Its approach to presenting algorithms alongside working code influenced the style of this book.
- Skiena, S. *The Algorithm Design Manual*, 3rd edition. Springer, 2020. A unique combination of algorithm design techniques and a catalogue of algorithmic problems, useful as both a textbook and a reference.
- Wirth, N. *Algorithms + Data Structures = Programs*. Prentice Hall, 1976. Also available at https://people.inf.ethz.ch/wirth/AD.pdf. A classic that pioneered the idea of teaching algorithms through a real programming language (Pascal). The title captures a philosophy this book shares.
- Knuth, D.E. *The Art of Computer Programming*, Volumes 1–4A. Addison-Wesley, 1997–2011. The definitive, encyclopedic treatment of algorithms and their analysis. An invaluable reference for the mathematically inclined reader.
- Aho, A.V., Hopcroft, J.E., and Ullman, J.D. *The Design and Analysis of Com-*

puter Algorithms. Addison-Wesley, 1974. A foundational textbook that established many of the standard approaches to algorithm analysis.

- Dasgupta, S., Papadimitriou, C.H., and Vazirani, U.V. *Algorithms*. McGraw-Hill, 2006. A concise and elegant textbook that is freely available from the authors. Particularly strong on number theory and NP-completeness.
- Sipser, M. *Introduction to the Theory of Computation*, 3rd edition. Cengage Learning, 2012. The standard reference for computational complexity theory, NP-completeness, and the theory of computation.

25.2 Online resources

- MIT OpenCourseWare. *6.006 Introduction to Algorithms*. https://ocw.mit.edu/courses/6-006-introduction-to-algorithms-spring-2020/. Lecture videos, notes, and problem sets covering the material in Parts I–IV of this book.
- MIT OpenCourseWare. *6.046J Design and Analysis of Algorithms*. https://ocw.mit.edu/courses/6-046j-design-and-analysis-of-algorithms-spring-2015/. The follow-on course covering advanced algorithm design techniques, network flow, and computational complexity.

25.3 Note on authorship and licensing

A substantial part of this book was created with the assistance of Zenflow, using Claude Code and Claude Opus 4.6.

Bugs and errors should be reported at https://github.com/amoilanen/Algorithms-with-TypeScript.