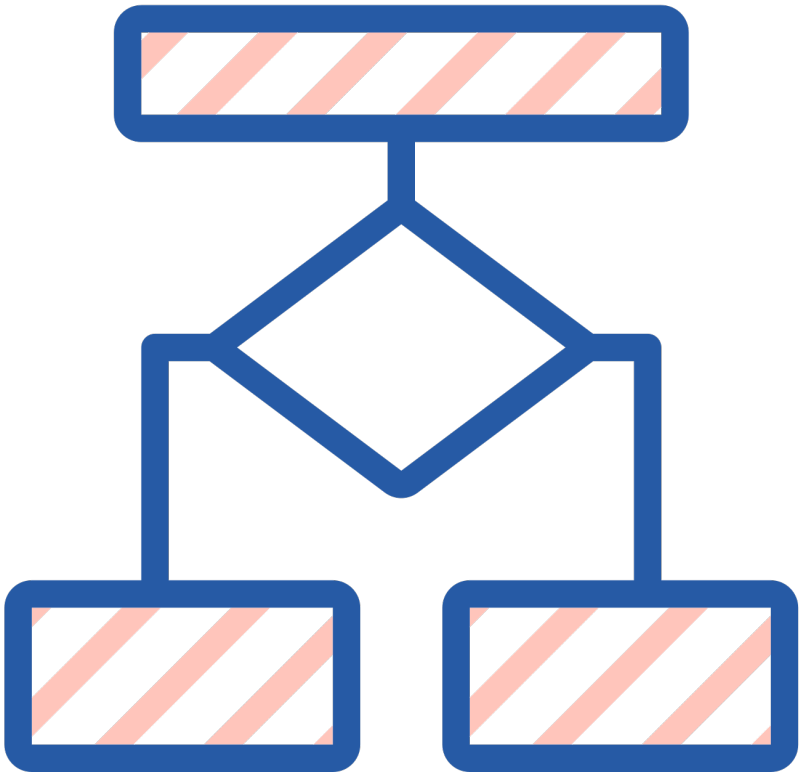
JavaScript Algorithms



readbytes

JavaScript Algorithms

Guide to Problem Solving and Data Structures

readbytes.github.io

2025-07-16

This page is intentionally left blank.

Contents

1 Introduction **17**
- 1.1 Why Study Algorithms in JavaScript? . . . 17
- 1.2 Using JS to Understand Efficiency . . . 17
- 1.3 A Quick Tour of Big-O Notation . . . 19
  - 1.3.1 What Is Big-O? . . . 19
  - 1.3.2 Common Big-O Classes (With Analogies) . . . 20
  - 1.3.3 Visualizing Growth . . . 21
  - 1.3.4 Why It Matters . . . 21
  - 1.3.5 Final Thoughts . . . 21

2 Analyzing Algorithms **23**
- 2.1 Time and Space Complexity . . . 23
  - 2.1.1 Time Complexity: Measuring Speed . . . 23
  - 2.1.2 Space Complexity: Measuring Memory Usage . . . 24
  - 2.1.3 Time vs. Space: The Trade-Off . . . 24
  - 2.1.4 Conclusion . . . 25
- 2.2 Asymptotic Notation (O, Ω, Θ) . . . 25
  - 2.2.1 Big-O Notation (O): The Worst Case . . . 25
  - 2.2.2 Big-Omega Notation (Ω): The Best Case . . . 26
  - 2.2.3 Big-Theta Notation (Θ): The Tight Bound . . . 26
  - 2.2.4 Summary Table . . . 27
  - 2.2.5 Why It Matters . . . 27
- 2.3 Real-World Performance in JS Engines . . . 27
  - 2.3.1 JIT Compilation: Faster with Time . . . 28
  - 2.3.2 Garbage Collection and Memory Pressure . . . 28
  - 2.3.3 Caching and CPU-Level Optimizations . . . 28
  - 2.3.4 Benchmarking in JavaScript . . . 29
  - 2.3.5 Interpreting Benchmark Results . . . 29
  - 2.3.6 Conclusion . . . 29

3 Mathematical Tools (Lightweight) **31**
- 3.1 Logarithms, Exponents, and Series . . . 31
  - 3.1.1 Exponents: Rapid Growth . . . 31
  - 3.1.2 Logarithms: The Opposite of Exponents . . . 31
  - 3.1.3 Comparing Growth Rates Visually . . . 32
  - 3.1.4 Series: Summing Patterns . . . 32
  - 3.1.5 Final Thoughts . . . 33
- 3.2 Recurrence Relations (Brief) . . . 33
  - 3.2.1 What Is a Recurrence Relation? . . . 33
  - 3.2.2 Example: Recursive Fibonacci . . . 34
  - 3.2.3 Improving with Memoization . . . 34
  - 3.2.4 General Strategy for Recurrence Analysis . . . 34

3.2.5 Final Thoughts . . . 35
3.3 Growth Rates of Functions . . . 35
3.3.1 Why Growth Rates Matter . . . 35
3.3.2 Common Growth Rates . . . 35
3.3.3 JavaScript Examples: Comparing Growth . . . 36
3.3.4 Visualizing Growth . . . 37
3.3.5 Conclusion . . . 37

4 Sorting and Order: Elementary Sorting Algorithms 39
4.1 Bubble Sort . . . 39
4.1.1 How Bubble Sort Works . . . 39
4.1.2 JavaScript Implementation . . . 42
4.1.3 Visualization of the Sorting Process . . . 42
4.1.4 Time Complexity . . . 43
4.1.5 When Should You Use Bubble Sort? . . . 43
4.1.6 Summary . . . 43
4.2 Insertion Sort . . . 43
4.2.1 How Insertion Sort Works . . . 43
4.2.2 JavaScript Implementation . . . 46
4.2.3 Visualizing Insertion Sort . . . 47
4.2.4 Performance Comparison with Bubble Sort . . . 47
4.2.5 Use Cases and Limitations . . . 48
4.2.6 Conclusion . . . 49
4.3 Selection Sort . . . 49
4.3.1 How Selection Sort Works . . . 49
4.3.2 JavaScript Implementation . . . 52
4.3.3 Performance Characteristics . . . 52
4.3.4 Visualizing the Sorting Steps . . . 53
4.3.5 Comparison to Other Elementary Sorts . . . 53
4.3.6 Conclusion . . . 54

5 Sorting and Order: Divide and Conquer 56
5.1 Merge Sort . . . 56
5.1.1 The Divide-and-Conquer Paradigm . . . 56
5.1.2 Merge Sort in Action . . . 56
5.1.3 JavaScript Implementation . . . 59
5.1.4 Recursion Flow and Merging (Visualized) . . . 60
5.1.5 Why Merge Step Guarantees Sorted Output . . . 60
5.1.6 Time and Space Complexity . . . 60
5.1.7 When to Use Merge Sort . . . 61
5.1.8 Conclusion . . . 61
5.2 Quick Sort . . . 61
5.2.1 The Quick Sort Strategy . . . 62
5.2.2 Visual Example . . . 62
5.2.3 JavaScript Implementation . . . 66

5.2.4 Recursion Flow Diagram . . . 67
5.2.5 Time Complexity . . . 67
5.2.6 Pivot Choice Matters . . . 67
5.2.7 Practical Tips for JS Developers . . . 68
5.2.8 Conclusion . . . 68
5.3 Time Complexity and Recursion Trees . . . 68
5.3.1 What Is a Recursion Tree? . . . 69
5.3.2 Merge Sort as a Recursion Tree . . . 69
5.3.3 JavaScript Code for Merge Sort Breakdown . . . 69
5.3.4 Quick Sort and Recursion Tree Variability . . . 70
5.3.5 JavaScript Code for Recursive Depth . . . 70
5.3.6 Intuition Behind the `log n` Factor . . . 71
5.3.7 Summary . . . 71

6 Sorting and Order: Linear-Time Sorting 77
6.1 Counting Sort . . . 77
6.1.1 When Is Counting Sort Efficient? . . . 77
6.1.2 How Counting Sort Works . . . 77
6.1.3 JavaScript Implementation . . . 78
6.1.4 Time and Space Complexity . . . 78
6.1.5 Limitations . . . 79
6.1.6 When Counting Sort Outperforms . . . 79
6.1.7 Conclusion . . . 79
6.2 Radix Sort . . . 79
6.2.1 How Radix Sort Works . . . 80
6.2.2 LSD Radix Sort Step-by-Step . . . 80
6.2.3 JavaScript Implementation (LSD Radix Sort) . . . 80
6.2.4 Complexity and Practical Considerations . . . 81
6.2.5 Handling Strings or Fixed-Length Keys . . . 81
6.2.6 When to Use Radix Sort in JavaScript . . . 82
6.2.7 Summary . . . 82
6.3 Bucket Sort . . . 82
6.3.1 How Bucket Sort Works . . . 82
6.3.2 JavaScript Example . . . 83
6.3.3 Visualizing Bucket Sort . . . 83
6.3.4 When to Use Bucket Sort . . . 84
6.3.5 Performance . . . 84
6.4 When These Work in JS Contexts . . . 84
6.4.1 Constraints in JavaScript Environments . . . 85
6.4.2 Comparing to `.sort()` . . . 85
6.4.3 Example: Benchmark Comparison . . . 86
6.4.4 Best Practices . . . 86

7 Arrays and Linked Lists 88
7.1 Native Arrays and Their Limits . . . 88

7.1.1 Key Features of Native Arrays . . . 96
7.1.2 Performance Implications . . . 97
7.1.3 When Native Arrays Become Inefficient . . . 97
7.1.4 Example: Performance Difference . . . 97
7.1.5 Conclusion . . . 98
7.2 Implementing Singly/Doubly Linked Lists . . . 98
7.2.1 Concept: Linked Lists vs Arrays . . . 98
7.2.2 Singly Linked List . . . 99
7.2.3 Doubly Linked List . . . 105
7.2.4 When to Use Linked Lists . . . 109
7.2.5 Summary . . . 110

8 Stacks and Queues 112
8.1 Implementing from Scratch . . . 112
8.1.1 Stack: LIFO (Last In, First Out) . . . 112
8.1.2 Queue: FIFO (First In, First Out) . . . 115
8.1.3 Queue Using Linked List . . . 118
8.1.4 Example: BFS (Breadth-First Search) . . . 119
8.1.5 Summary . . . 120
8.2 Use Cases in Frontend/Backend . . . 120
8.2.1 Undo/Redo Functionality (Frontend Stack) . . . 120
8.2.2 Call Stack (JavaScript Runtime) . . . 121
8.2.3 Event Queue and Task Scheduling (Frontend Queue) . . . 121
8.2.4 Task Processing and Job Queues (Backend Queue) . . . 122
8.2.5 Why Use These Structures? . . . 123
8.2.6 Summary . . . 123

9 Hash Tables 125
9.1 JavaScript Objects and `Map` . . . 125
9.1.1 Key Differences: Object vs `Map` . . . 125
9.1.2 Using JavaScript Objects as Hash Tables . . . 125
9.1.3 Using `Map` for Better Key Control . . . 126
9.1.4 Iterating with `Map` . . . 126
9.1.5 When to Use Each . . . 127
9.1.6 Summary . . . 127
9.2 Hash Functions and Collisions . . . 127
9.2.1 What Is a Hash Function? . . . 128
9.2.2 What Are Collisions? . . . 128
9.2.3 Collision Resolution Strategies . . . 128
9.2.4 Trade-Offs and Challenges . . . 129
9.2.5 Summary . . . 129
9.3 Custom Hash Table Implementation . . . 130
9.3.1 Building Blocks . . . 130
9.3.2 Step 1: Basic Hash Function . . . 130
9.3.3 Step 2: Hash Table Class with Chaining . . . 131

9.3.4 Example Usage . . . 132
9.3.5 Limitations of This Implementation . . . 134
9.3.6 Potential Improvements . . . 134
9.3.7 Summary . . . 135

10 Trees and Binary Search Trees **137**
10.1 Recursive Tree Traversals . . . 137
10.1.1 Tree Structure Example . . . 137
10.1.2 Preorder Traversal: `Node Left Right` . . . 141
10.1.3 Inorder Traversal: `Left Node Right` . . . 142
10.1.4 Postorder Traversal: `Left Right Node` . . . 142
10.1.5 Example: Tree Node Class . . . 143
10.1.6 Visual Comparison Summary . . . 143
10.1.7 Why Recursion Works . . . 143
10.1.8 Practical Use Cases . . . 144
10.1.9 Summary . . . 144
10.2 Balanced vs Unbalanced Trees . . . 144
10.2.1 What Is a Balanced Tree? . . . 144
10.2.2 What Is an Unbalanced Tree? . . . 145
10.2.3 Height and Its Impact on Performance . . . 145
10.2.4 Consequences of Unbalance . . . 146
10.2.5 Balancing Strategies (Conceptual) . . . 146
10.2.6 JavaScript Example: Searching Balanced vs Skewed Trees . . . 146
10.2.7 Summary . . . 147
10.3 Binary Search Tree from Scratch . . . 147
10.3.1 BST Node Structure . . . 152
10.3.2 Insertion . . . 152
10.3.3 Searching . . . 153
10.3.4 Deletion . . . 153
10.3.5 Inorder Traversal (Sorting) . . . 154
10.3.6 Complete Example and Test Cases . . . 155
10.3.7 Key Takeaways . . . 155
10.3.8 Summary . . . 155

11 Heaps and Priority Queues **158**
11.1 Min-Heap, Max-Heap . . . 158
11.1.1 Complete Binary Tree Structure . . . 161
11.1.2 Min-Heap vs Max-Heap . . . 161
11.1.3 Heap Property (Heap Invariant) . . . 162
11.1.4 Core Heap Operations . . . 162
11.1.5 JavaScript Min-Heap Example . . . 162
11.1.6 Visualizing Min-Heap Operations . . . 164
11.1.7 Why Use Heaps? . . . 164
11.1.8 Summary . . . 164
11.2 Heap Sort . . . 164

11.2.1 Phase 1: Build the Heap . . . 165
11.2.2 Phase 2: Extract Elements and Heapify . . . 165
11.2.3 Why It Works . . . 168
11.2.4 JavaScript Implementation . . . 168
11.2.5 Time and Space Complexity . . . 169
11.2.6 Pros and Cons . . . 170
11.2.7 Visual Aid: Max-Heap Tree for `[4, 10, 3, 5, 1]` . . . 170
11.2.8 Tip: Reversing for Descending Order . . . 170
11.2.9 Summary . . . 170
11.3 Implementing a Priority Queue . . . 171
11.3.1 Common Use Cases . . . 171
11.3.2 How Heaps Make This Efficient . . . 171
11.3.3 JavaScript Implementation . . . 171
11.3.4 Time Complexity Summary . . . 173
11.3.5 Summary . . . 174

12 Balanced Search Trees **176**
12.1 Red-Black Trees (Conceptual Overview) . . . 176
12.1.1 What Makes a Tree "Red-Black"? . . . 176
12.1.2 Rotations and Recoloring . . . 176
12.1.3 Visual Representation . . . 180
12.1.4 Why Red-Black Trees? . . . 181
12.1.5 Real-World Usage . . . 181
12.1.6 Summary . . . 181
12.2 AVL Trees (Optional in JavaScript) . . . 181
12.2.1 Balance Factor and Rebalancing . . . 182
12.2.2 Rotations in AVL Trees . . . 185
12.2.3 AVL vs. Red-Black Trees . . . 185
12.2.4 JavaScript-Style Pseudocode (Simplified) . . . 185
12.2.5 Why This Is Optional . . . 186
12.2.6 Summary . . . 186
12.3 B-Trees and Their Real-World Relevance . . . 186
12.3.1 What Makes a B-Tree Different? . . . 187
12.3.2 Why B-Trees Matter for Storage . . . 187
12.3.3 How It Works (Simplified) . . . 188
12.3.4 B-Trees in JavaScript? . . . 188
12.3.5 Summary . . . 189

13 Graph Representations **191**
13.1 Adjacency Lists and Matrices in JS . . . 191
13.1.1 Adjacency List . . . 192
13.1.2 Adjacency Matrix . . . 193
13.1.3 Comparison: List vs Matrix . . . 194
13.1.4 Performance Implications for Algorithms . . . 194
13.1.5 Summary . . . 195

13.2 Directed vs Undirected Graphs . . . 195
13.2.1 Directed Graphs (Digraphs) . . . 197
13.2.2 Undirected Graphs . . . 198
13.2.3 Visual Example . . . 198
13.2.4 Key Differences . . . 199
13.2.5 Summary . . . 199
13.3 Weighted vs Unweighted Graphs . . . 199
13.3.1 What Is a Weighted Graph? . . . 200
13.3.2 Representation in JavaScript . . . 200
13.3.3 Unweighted Graphs . . . 201
13.3.4 Key Differences . . . 201
13.3.5 Real-World Applications . . . 201
13.3.6 Algorithms That Use Weights . . . 202
13.3.7 Summary . . . 202

14 Graphs: Traversals **204**
14.1 Breadth-First Search (BFS) . . . 204
14.1.1 BFS Fundamentals . . . 204
14.1.2 JavaScript Implementation . . . 206
14.1.3 Traversal Order (Step-by-Step) . . . 207
14.1.4 Time and Space Complexity . . . 208
14.1.5 Real-World Use Cases . . . 208
14.1.6 BFS vs DFS (Quick Note) . . . 208
14.1.7 Summary . . . 208
14.2 Depth-First Search (DFS) . . . 209
14.2.1 How DFS Works . . . 209
14.2.2 Recursive DFS Implementation (JavaScript) . . . 211
14.2.3 Iterative DFS Using a Stack . . . 212
14.2.4 DFS Traversal Example . . . 212
14.2.5 Applications of DFS . . . 213
14.2.6 Time and Space Complexity . . . 213
14.2.7 Summary . . . 213
14.3 Applications (Pathfinding, Web Crawling) . . . 214
14.3.1 Pathfinding in Maps and Mazes . . . 214
14.3.2 Web Crawling and Site Traversal . . . 215
14.3.3 Choosing BFS vs DFS . . . 215
14.3.4 Performance & Trade-offs . . . 215
14.3.5 Summary . . . 216

15 Graph Shortest Paths **218**
15.1 Dijkstra's Algorithm . . . 218
15.1.1 How Dijkstras Algorithm Works . . . 218
15.1.2 Graph and Priority Queue Setup . . . 222
15.1.3 JavaScript Implementation . . . 222
15.1.4 Time Complexity . . . 225

15.1.5 Real-World Applications . . . 225
15.1.6 Summary . . . 225
15.2 Bellman-Ford Algorithm . . . 225
15.2.1 When to Use Bellman-Ford . . . 225
15.2.2 Core Idea: Relaxation . . . 226
15.2.3 JavaScript Implementation . . . 230
15.2.4 Negative Cycle Detection . . . 231
15.2.5 Time Complexity . . . 231
15.2.6 Real-World Applications . . . 231
15.2.7 Dijkstra vs Bellman-Ford . . . 232
15.2.8 Summary . . . 232
15.3 A* Search (Practical Frontend Use)** . . . 232
15.3.1 What Is A* Search? . . . 232
15.3.2 JavaScript Implementation . . . 236
15.3.3 Heuristic Function Design . . . 238
15.3.4 Performance Considerations . . . 238
15.3.5 Real-World Applications . . . 238
15.3.6 Summary . . . 239

16 Minimum Spanning Trees 241
16.1 Prim's Algorithm . . . 241
16.1.1 How Prims Algorithm Works . . . 241
16.1.2 Graph Representation . . . 244
16.1.3 JavaScript Implementation . . . 245
16.1.4 Visual Step Example . . . 247
16.1.5 Time Complexity . . . 247
16.1.6 Real-World Applications . . . 247
16.1.7 Summary . . . 247
16.2 Kruskal's Algorithm . . . 248
16.2.1 How Kruskals Algorithm Works . . . 248
16.2.2 Union-Find Data Structure in JavaScript . . . 251
16.2.3 Kruskals Algorithm in JavaScript . . . 252
16.2.4 Example Usage . . . 252
16.2.5 Cycle Detection . . . 253
16.2.6 Time Complexity . . . 253
16.2.7 Kruskal vs. Prim . . . 253
16.2.8 Real-World Use Cases . . . 254
16.2.9 Summary . . . 254
16.3 Real-World Applications . . . 254
16.3.1 Network Cabling and Infrastructure Design . . . 254
16.3.2 Clustering in Machine Learning . . . 255
16.3.3 Image Segmentation . . . 255
16.3.4 Traffic and Utility Networks . . . 256
16.3.5 Algorithm Choice in Practice . . . 256
16.3.6 Summary . . . 256

17 Principles of Dynamic Programming **258**
17.1 Overlapping Subproblems and Memoization 258
17.1.1 Overlapping Subproblems 258
17.1.2 Memoization: The Top-Down Optimization 258
17.1.3 Comparing Performance 259
17.1.4 When to Use Memoization 259
17.1.5 Summary 259
17.2 Bottom-Up vs Top-Down Approaches 260
17.2.1 Top-Down Approach: Memoization 260
17.2.2 Bottom-Up Approach: Tabulation 261
17.2.3 Side-by-Side Comparison 261
17.2.4 Example: Longest Common Subsequence (LCS) 262
17.2.5 Practical Tips for JavaScript Developers 262
17.2.6 Summary 263

18 Principles of Dynamic Programming Classic Problems **265**
18.1 Fibonacci Variants 265
18.1.1 Classic Fibonacci Recap 265
18.1.2 Memoized Fibonacci 265
18.1.3 Iterative Fibonacci (Bottom-Up) 266
18.1.4 Variant: Counting Ways to Climb Stairs 266
18.1.5 Underlying DP Principles 267
18.1.6 Summary 267
18.2 Longest Common Subsequence (LCS) 268
18.2.1 Why LCS Matters 268
18.2.2 Recursive Formulation 268
18.2.3 Naive Recursive Implementation 268
18.2.4 Dynamic Programming: Bottom-Up Approach 269
18.2.5 JavaScript Implementation with Comments 269
18.2.6 Reconstructing the LCS (Optional) 270
18.2.7 Practical Applications 270
18.2.8 Summary 271
18.3 Knapsack Problem 271
18.3.1 Problem Statement 271
18.3.2 Recursive Relation 271
18.3.3 Bottom-Up Tabulation (JavaScript) 272
18.3.4 Space Optimization 273
18.3.5 0/1 vs Fractional Knapsack 273
18.3.6 Optimization Tips 274
18.3.7 Real-World Applications 274
18.3.8 Summary 274
18.4 Grid-Based Pathfinding 274
18.4.1 Problem 1: Count Unique Paths 275
18.4.2 Problem 2: Minimum Cost Path 275
18.4.3 Problem 3: Paths with Obstacles 276

18.4.4 Real-World Use Cases . . . 277
18.4.5 Optimization Tips . . . 277
18.4.6 Summary . . . 277

19 Greedy Algorithms 279
19.1 When Greedy Works . . . 279
19.1.1 The Greedy Paradigm . . . 279
19.1.2 Conditions for Greedy Optimality . . . 279
19.1.3 Greedy vs. Dynamic Programming . . . 280
19.1.4 Examples of Greedy-Appropriate Problems . . . 280
19.1.5 How to Identify Greedy Problems . . . 280
19.1.6 Conclusion . . . 281
19.2 Activity Selection . . . 281
19.2.1 Problem Statement . . . 281
19.2.2 Why Greedy Works . . . 281
19.2.3 Greedy Strategy . . . 282
19.2.4 JavaScript Implementation . . . 282
19.2.5 Real-World Applications . . . 283
19.2.6 Key Takeaways . . . 283
19.3 Huffman Encoding . . . 284
19.3.1 The Problem . . . 284
19.3.2 The Greedy Approach . . . 290
19.3.3 JavaScript Implementation . . . 290
19.3.4 Why Its Greedy . . . 292
19.3.5 Applications in Data Compression . . . 292
19.3.6 Summary . . . 292
19.4 Interval Scheduling . . . 292
19.4.1 Problem Definition . . . 293
19.4.2 Greedy Strategy: Earliest Finish Time . . . 296
19.4.3 JavaScript Implementation . . . 297
19.4.4 Variations and Constraints . . . 298
19.4.5 Real-World Applications . . . 298
19.4.6 Summary . . . 298

20 Backtracking and Recursion 300
20.1 Combinatorial Problems . . . 300
20.1.1 What Is Backtracking? . . . 300
20.1.2 Example 1: Generating All Subsets . . . 300
20.1.3 Example 2: Generating All Combinations (k-length) . . . 301
20.1.4 Backtracking Efficiency and Complexity . . . 302
20.1.5 Real-World Use Cases . . . 302
20.1.6 Summary . . . 303
20.2 Subsets, Permutations, and N-Queens . . . 303
20.2.1 Generating All Subsets . . . 303
20.2.2 Generating All Permutations . . . 304

20.2.3 The N-Queens Problem . . . 305
20.2.4 Backtracking Pattern and Debugging Tips . . . 306
20.2.5 Relevance to Constraint Satisfaction . . . 307
20.2.6 Summary . . . 307
20.3 Solving Puzzles Recursively . . . 307
20.3.1 Recursive Exploration and Base Cases . . . 307
20.3.2 Example 1: Maze Solving . . . 308
20.3.3 Example 2: Sudoku Solver (Simplified 44) . . . 309
20.3.4 Managing Recursion Depth and Performance . . . 310
20.3.5 Real-World Relevance . . . 310
20.3.6 Summary . . . 310

21 Real-World Applications 313
21.1 Autocomplete with Tries . . . 313
21.1.1 What Is a Trie? . . . 313
21.1.2 Why Use a Trie for Autocomplete? . . . 314
21.1.3 JavaScript Trie Implementation . . . 314
21.1.4 Example Usage . . . 315
21.1.5 Performance Benefits . . . 315
21.1.6 Real-World Applications . . . 316
21.1.7 Summary . . . 316
21.2 Debouncing/Throttling as Algorithmic Patterns . . . 316
21.2.1 Why Do We Need These Patterns? . . . 316
21.2.2 Debouncing: Wait Until the User Stops . . . 317
21.2.3 Throttling: Limit Call Frequency . . . 317
21.2.4 Comparing Debounce vs Throttle . . . 318
21.2.5 Common Pitfalls . . . 318
21.2.6 Best Practices . . . 319
21.2.7 Summary . . . 319
21.3 Scheduling Algorithms in UIs . . . 319
21.3.1 Why Scheduling Matters in the UI . . . 319
21.3.2 `requestAnimationFrame`: UI-Friendly Animations . . . 320
21.3.3 `setTimeout` and Task Prioritization . . . 320
21.3.4 Cooperative Multitasking with `yield` and Microtasks . . . 321
21.3.5 Task Queues and Idle Callbacks . . . 321
21.3.6 Real-World Constraints and Trade-Offs . . . 322
21.3.7 Summary . . . 322
21.4 Caching Strategies (LRU, LFU) in JS . . . 323
21.4.1 Understanding Cache Hits and Misses . . . 323
21.4.2 Least Recently Used (LRU) Cache . . . 323
21.4.3 Least Frequently Used (LFU) Cache . . . 324
21.4.4 Choosing Between LRU and LFU . . . 326
21.4.5 Memory Considerations . . . 326
21.4.6 Summary . . . 327

22 Algorithmic Thinking in Practice **329**
22.1 How to Approach a Problem . . . 329
22.1.1 Step 1: Understand the Problem . . . 329
22.1.2 Step 2: Identify Inputs and Outputs . . . 329
22.1.3 Step 3: Brainstorm Ideas . . . 329
22.1.4 Step 4: Choose an Algorithmic Strategy . . . 330
22.1.5 Step 5: Write Pseudocode . . . 330
22.1.6 Step 6: Implement the Solution . . . 330
22.1.7 Step 7: Validate and Test . . . 330
22.1.8 Additional Tips for Breaking Down Problems . . . 331
22.1.9 Emphasizing Reasoning Over Coding . . . 331
22.1.10 Summary . . . 331
22.2 Brute Force → Greedy → DP . . . 331
22.2.1 Stage 1: Brute Force The Exhaustive Search . . . 331
22.2.2 Stage 2: Greedy Quick Heuristic . . . 332
22.2.3 Stage 3: Dynamic Programming Optimal and Efficient . . . 333
22.2.4 Recognizing Which Approach to Use . . . 333
22.2.5 Summary . . . 334
22.3 Thinking in Constraints . . . 334
22.3.1 Why Constraints Matter . . . 334
22.3.2 Constraints Guide Algorithm Choices . . . 335
22.3.3 Example: Handling Large Input Efficiently . . . 335
22.3.4 Memory vs. Speed Trade-offs . . . 335
22.3.5 When Constraints Force Algorithmic Changes . . . 336
22.3.6 Edge Cases and Constraints . . . 336
22.3.7 Summary: Critical Thinking in Constraints . . . 337
22.3.8 Final Thought . . . 337

Chapter 1.

Introduction

1. Why Study Algorithms in JavaScript?
2. Using JS to Understand Efficiency
3. A Quick Tour of Big-O Notation

1 Introduction

1.1 Why Study Algorithms in JavaScript?

At the heart of every software application lies a set of algorithms—step-by-step instructions that define how to solve problems and perform tasks efficiently. Whether you're building a search engine, sorting a list of emails, finding the shortest path on a map, or recommending content on a social platform, algorithms are the engine that makes it all possible. Understanding how algorithms work—and how to design your own—is one of the most important skills a programmer can develop.

So why study algorithms in JavaScript? First, let's talk about accessibility. JavaScript is everywhere. It runs in virtually every modern web browser, on mobile devices, and even on servers via Node.js. If you have a browser, you can write, run, and test algorithms instantly—no special setup required. This makes JavaScript an ideal language for learning and experimenting with algorithmic concepts.

Second, JavaScript's flexibility and dynamic nature allow you to focus on the core logic of an algorithm without getting bogged down by complex syntax or rigid typing rules. For beginners, this means less time wrestling with language quirks and more time mastering the ideas that matter: How do you sort data efficiently? How do you search through a dataset? How do you make your code scale when the input size grows?

Studying algorithms in JavaScript doesn't just make you better at solving coding interview problems (though that's a nice bonus). It helps you think more clearly and solve problems methodically. For instance, consider a common real-world problem: organizing a list of names alphabetically. You could do it manually, but with a well-designed sorting algorithm—such as merge sort or quicksort—you can solve the problem efficiently, even with thousands of names.

Another example: imagine you're building a simple to-do list app. As users add more tasks, you might want to implement a feature that automatically prioritizes urgent items. Behind that feature is an algorithm—perhaps one that scores tasks based on deadlines and importance, then sorts them accordingly.

Even in daily life, we encounter algorithmic thinking. Finding the shortest route on a GPS? That's Dijkstra's algorithm at work. Searching for a contact in your phone? A search algorithm is doing the heavy lifting. Algorithms power the systems we interact with every day, often behind the scenes.

1.2 Using JS to Understand Efficiency

One of the most valuable aspects of studying algorithms is learning how to evaluate and improve the efficiency of your code. Efficiency matters because it determines how your

application performs when handling large inputs, multiple users, or real-time operations. Fortunately, JavaScript offers simple yet powerful tools to help you experiment with algorithms and understand how different approaches can affect performance.

At the core of this exploration is the ability to **measure how long your code takes to run**. JavaScript provides the `performance.now()` function—part of the Web Performance API—which returns a high-resolution timestamp. You can use this to time how long a block of code takes to execute, down to the millisecond. This makes it easy to test and compare different algorithmic approaches to the same problem.

Let's look at an example. Suppose we want to write a function that checks if a number exists in an array. We'll compare two approaches: one using a **loop** and another using the **`includes()`** method.

Full runnable code:

```
function linearSearch(arr, target) {
  for (let i = 0; i < arr.length; i++) {
    if (arr[i] === target) return true;
  }
  return false;
}

const array = Array.from({ length: 1_000_000 }, (_, i) => i);
const target = 999_999;

const t1 = performance.now();
linearSearch(array, target);
const t2 = performance.now();
console.log(`Linear search took ${(t2 - t1).toFixed(2)} ms`);

const t3 = performance.now();
array.includes(target);
const t4 = performance.now();
console.log(`Array.includes took ${(t4 - t3).toFixed(2)} ms`);
```

By wrapping each method call with `performance.now()`, we can observe how long each approach takes. In this case, both methods perform similarly for small arrays, but timing tests help reveal the differences as input sizes grow. This kind of **hands-on benchmarking** is crucial when learning how algorithms behave under stress.

Let's look at another example: comparing two ways to compute the sum of the first `n` numbers—one iterative and one mathematical.

Full runnable code:

```
function sumIterative(n) {
  let total = 0;
  for (let i = 1; i <= n; i++) {
    total += i;
  }
  return total;
}
```

```
function sumFormula(n) {
  return (n * (n + 1)) / 2;
}

const N = 1_000_000_000;

let start = performance.now();
sumIterative(N);
let end = performance.now();
console.log(`Iterative sum took ${(end - start).toFixed(2)} ms`);

start = performance.now();
sumFormula(N);
end = performance.now();
console.log(`Formula sum took ${(end - start).toFixed(2)} ms`);
```

Here, the iterative version performs a billion additions, while the formula uses a single arithmetic expression. The result? The formula runs almost instantly, while the loop takes significantly longer. This illustrates the concept of **algorithmic complexity**—some solutions scale much better than others as input size increases.

Using JavaScript in this way allows you to **experiment, observe, and refine** your algorithms in real time. You don't need specialized tools or environments—just a browser or a Node.js terminal. This immediate feedback loop helps you develop an intuitive sense for performance, which is key to writing efficient and effective code.

By practicing performance testing in JavaScript, you'll build a deeper understanding of not just what your code does—but how well it does it.

1.3 A Quick Tour of Big-O Notation

When we talk about the efficiency of an algorithm, we often mean more than just how fast it runs on your computer. We want to know how its runtime grows as the size of the input increases. That's where **Big-O notation** comes in. Big-O gives us a formal language to describe **how algorithms scale**, allowing us to compare different approaches not just by speed, but by how they behave in the worst-case scenarios.

1.3.1 What Is Big-O?

Big-O notation describes the **upper bound** of an algorithm's growth rate. It tells us how the runtime (or space usage) increases relative to the size of the input—commonly denoted as `n`. We ignore constants and lower-order terms, focusing only on the dominant behavior as `n` grows very large.

Think of it like this: if you're trying to estimate how long a road trip will take, you don't worry about a single red light or a stop for coffee—you care about the speed limit and the total distance. Big-O works the same way: it strips away minor details to focus on long-term performance.

1.3.2 Common Big-O Classes (With Analogies)

Let's look at a few key complexity classes with relatable examples and JavaScript code.

O(1) – Constant Time

> *No matter how large the input is, the algorithm takes the same amount of time.*

Example: Looking up a value in an object by key.

```
const user = { id: 123, name: "Alex" };
console.log(user["id"]); // Always takes the same amount of time
```

Analogy: Grabbing a book from a specific shelf—you know exactly where it is.

O(n) – Linear Time

> *The time increases directly with the size of the input.*

Example: Looping through an array to find a value.

```
function findValue(arr, target) {
  for (let item of arr) {
    if (item === target) return true;
  }
  return false;
}
```

Analogy: Reading every page of a book to find a specific word. The more pages, the longer it takes.

O(n²) – Quadratic Time

> *The time increases with the square of the input size. Usually occurs in nested loops.*

Example: Comparing every pair of items in an array.

```
function hasDuplicates(arr) {
  for (let i = 0; i < arr.length; i++) {
    for (let j = i + 1; j < arr.length; j++) {
      if (arr[i] === arr[j]) return true;
    }
  }
  return false;
}
```

Analogy: If everyone in a room shakes hands with everyone else, the number of handshakes grows rapidly as more people join.

1.3.3 Visualizing Growth

Let's imagine how runtime changes with input size:

Input Size (n)	O(1)	O(n)	$O(n^2)$
10	1	10	100
100	1	100	10,000
1,000	1	1,000	1,000,000

As you can see, quadratic algorithms become impractical very quickly. This is why performance tuning often focuses on reducing complexity.

1.3.4 Why It Matters

Suppose you're designing a search feature. You could use a simple loop (O(n)) or a more advanced data structure like a binary search tree (O(log n), which we'll discuss in later chapters). Even if both seem fast with small inputs, the more your user base grows, the more these differences matter.

In JavaScript, you can use performance testing to observe this in action. Try timing an O(n) vs. $O(n^2)$ function with increasing array sizes. You'll quickly see how the slower-growing algorithm outpaces the inefficient one as input scales.

1.3.5 Final Thoughts

Big-O notation gives you a language to reason about performance, regardless of programming language or hardware. It's not about exact timings but about **growth trends**. Understanding Big-O helps you choose better solutions, avoid performance pitfalls, and write code that scales gracefully as your data grows. As we dive deeper into algorithms and data structures, Big-O will be your guide to evaluating whether your solution is just good enough—or truly efficient.

Chapter 2.

Analyzing Algorithms

1. Time and Space Complexity
2. Asymptotic Notation (O, Ω, Θ)
3. Real-World Performance in JS Engines

2 Analyzing Algorithms

2.1 Time and Space Complexity

When evaluating the efficiency of an algorithm, it's essential to consider two key dimensions: **time complexity** and **space complexity**. These concepts help us understand not just how *fast* an algorithm runs, but also how much *memory* it uses. Mastering both allows developers to build applications that perform well and scale reliably.

2.1.1 Time Complexity: Measuring Speed

Time complexity describes how the number of operations grows relative to the size of the input, typically denoted as `n`. It helps us estimate how long an algorithm will take to complete as the input increases.

Let's look at a simple example:

```
function sumArray(arr) {
  let total = 0;
  for (let i = 0; i < arr.length; i++) {
    total += arr[i]; // One operation per element
  }
  return total;
}
```

Here, the loop runs once for each element in the array. If the array has 1,000 elements, the function performs roughly 1,000 addition operations. So, the time complexity is **O(n)**—it scales linearly with input size.

If we add nested loops, we increase the number of operations dramatically:

```
function compareAllPairs(arr) {
  for (let i = 0; i < arr.length; i++) {
    for (let j = 0; j < arr.length; j++) {
      console.log(arr[i], arr[j]);
    }
  }
}
```

This function performs `n × n` operations for an input of size `n`, making its time complexity **O(n²)**—quadratic time. Even small increases in input size can lead to large performance drops with such algorithms.

2.1.2 Space Complexity: Measuring Memory Usage

Space complexity refers to how much **additional memory** an algorithm uses as the input grows. This doesn't count the input itself—only the memory your algorithm needs to do its work.

Here's an example with **constant space** usage:

```
function findMax(arr) {
  let max = arr[0];
  for (let i = 1; i < arr.length; i++) {
    if (arr[i] > max) max = arr[i];
  }
  return max;
}
```

No matter how big the array is, this function only uses a few variables (`max`, `i`), so the space complexity is **O(1)**.

Now compare that with a function that creates a new array:

```
function doubleValues(arr) {
  const result = [];
  for (let i = 0; i < arr.length; i++) {
    result.push(arr[i] * 2);
  }
  return result;
}
```

Here, we create a new array of the same size as the input, so the space complexity is **O(n)**.

2.1.3 Time vs. Space: The Trade-Off

In many cases, you can't optimize for both time and space simultaneously—you have to make a **trade-off**. For example, caching results can save time but use more memory:

```
const cache = {};
function fibonacci(n) {
  if (n <= 1) return n;
  if (cache[n]) return cache[n];
  cache[n] = fibonacci(n - 1) + fibonacci(n - 2);
  return cache[n];
}
```

This **memoized** version of Fibonacci improves time complexity dramatically (from exponential to linear), but it uses **additional space** to store results.

Choosing between time and space efficiency depends on your application. On memory-constrained devices (like mobile phones), space may be the priority. On systems with large datasets, optimizing for time might matter more to keep users happy.

2.1.4 Conclusion

Understanding time and space complexity helps you write smarter code. It allows you to estimate performance, spot inefficiencies, and make informed trade-offs. As you progress through this book, we'll explore more complex algorithms—but these foundational ideas will always guide our analysis.

2.2 Asymptotic Notation (O, Ω, Θ)

Asymptotic notation is a mathematical tool used to describe the efficiency of algorithms as input size grows very large. It helps us classify algorithms based on their **growth rate**, or how their runtime or memory usage scales with input. There are three main notations used for this purpose:

- **Big-O (O)** – the **upper bound**: the worst-case scenario.
- **Big-Omega (Ω)** – the **lower bound**: the best-case scenario.
- **Big-Theta (Θ)** – the **tight bound**: the exact growth rate when both upper and lower bounds are the same.

Understanding these notations allows developers to make **precise comparisons** between algorithms, and to make performance guarantees about how a program will behave in the best, worst, or average case.

2.2.1 Big-O Notation (O): The Worst Case

Big-O notation describes the **maximum number of steps** an algorithm will take, regardless of the input. It gives us a **guarantee that the algorithm won't take longer than this upper bound**.

Formal definition: An algorithm is **O(f(n))** if its runtime does not grow faster than some constant multiple of `f(n)` as `n` becomes large.

Example: Linear Search

```
function linearSearch(arr, target) {
  for (let i = 0; i < arr.length; i++) {
    if (arr[i] === target) return i;
  }
  return -1;
}
```

- Best case: `target` is the first item → 1 comparison.
- Worst case: `target` is not in the array → `n` comparisons.
- **Time complexity:** `O(n)`

Big-O captures this **worst-case** behavior: we might have to look at every element in the array.

2.2.2 Big-Omega Notation (Ω): The Best Case

Big-Omega describes the **minimum time** an algorithm takes for any input. This can be useful when considering **optimistic scenarios**, although it doesn't guarantee consistent performance.

Formal definition: An algorithm is **Ω(f(n))** if there is a constant multiple of `f(n)` that the algorithm will always take at least, for sufficiently large `n`.

Example: Again, Linear Search

```
function linearSearch(arr, target) {
  for (let i = 0; i < arr.length; i++) {
    if (arr[i] === target) return i;
  }
  return -1;
}
```

- Best case: `target` is at index 0 → 1 comparison.
- **Best-case complexity:** `Ω(1)`

This shows that while the worst-case is `O(n)`, the best-case could be much faster.

2.2.3 Big-Theta Notation (Θ): The Tight Bound

Big-Theta represents the **exact growth rate** of an algorithm when the best and worst cases are similar. If an algorithm is both `O(f(n))` and `Ω(f(n))`, we say it's `Θ(f(n))`.

Formal definition: An algorithm is **Θ(f(n))** if it grows no faster and no slower than `f(n)` for large `n`. It provides both upper and lower bounds.

Example: Summing Array Elements

```
function sumArray(arr) {
  let total = 0;
  for (let i = 0; i < arr.length; i++) {
    total += arr[i];
  }
  return total;
}
```

- Always performs `n` additions regardless of input values.
- **Time complexity:** `Θ(n)`

Because this function always does the same number of steps, no matter what values are in

the array, both best and worst cases are linear.

2.2.4 Summary Table

Notation	Meaning	Describes	Used For
O(f(n))	Upper bound	Worst-case	Performance limits
Ω(f(n))	Lower bound	Best-case	Optimistic limit
Θ(f(n))	Tight bound	Exact growth	Balanced analysis

2.2.5 Why It Matters

Understanding all three notations gives a **more complete picture** of algorithm behavior. For example, an algorithm with best-case $\Omega(1)$ and worst-case $O(n^2)$ might seem efficient in specific scenarios but could perform terribly at scale.

JavaScript developers especially benefit from this analysis when building UI interactions, handling large data, or optimizing backend performance. Asymptotic notation equips you to choose or design algorithms that not only work, but also scale predictably under pressure.

In the chapters to come, we'll analyze many algorithms using these notations so you can apply this knowledge in real-world code.

2.3 Real-World Performance in JS Engines

While theoretical analysis with Big-O notation gives us a powerful framework to compare algorithms, real-world performance often tells a more nuanced story. JavaScript runs inside highly optimized engines—like V8 (Chrome, Node.js), SpiderMonkey (Firefox), and JavaScriptCore (Safari)—which use advanced techniques such as **just-in-time (JIT) compilation**, **caching**, and **garbage collection**. These optimizations can make some algorithms faster or slower than expected in practice.

Understanding these factors helps you benchmark algorithms meaningfully and make smart decisions when optimizing code.

2.3.1 JIT Compilation: Faster with Time

Modern JavaScript engines use **JIT compilation** to convert JavaScript code into machine code while it's running. Unlike ahead-of-time (AOT) compiled languages, JS starts out interpreted and then "learns" how your code behaves over time.

This means:

- **Hot code paths** (code that's run frequently) are aggressively optimized.
- Less-used code might never be compiled and stays slower.
- Sometimes, code can be *de-optimized* if it behaves unpredictably.

As a result, running a function several times may yield better performance than running it just once—something to keep in mind when benchmarking.

2.3.2 Garbage Collection and Memory Pressure

JavaScript is a **garbage-collected language**, which means memory is automatically cleaned up when no longer needed. While this simplifies development, it introduces **runtime pauses** that can affect performance measurements.

For example, algorithms that create many short-lived objects may perform worse than expected due to the overhead of memory allocation and cleanup:

```
function createTempArrays(n) {
  for (let i = 0; i < n; i++) {
    const temp = new Array(1000).fill(i);
  }
}
```

Here, each iteration creates a new array and fills it—this creates memory pressure and can trigger garbage collection, affecting overall timing.

2.3.3 Caching and CPU-Level Optimizations

Some performance gains or slowdowns may come from **browser-level or CPU-level caching**. For example, accessing data in a **contiguous array** is generally faster than working with scattered object properties, due to better **cache locality**.

2.3.4 Benchmarking in JavaScript

To measure real-world performance, you can use the **`performance.now()`** API, which provides sub-millisecond precision:

Full runnable code:

```
function testLoop(n) {
  let sum = 0;
  for (let i = 0; i < n; i++) {
    sum += i;
  }
  return sum;
}

const iterations = 5;
for (let i = 0; i < iterations; i++) {
  const start = performance.now();
  testLoop(1_000_000);
  const end = performance.now();
  console.log(`Run ${i + 1}: ${(end - start).toFixed(2)} ms`);
}
```

2.3.5 Interpreting Benchmark Results

When benchmarking:

- **Run multiple iterations**: One run isn't reliable. Multiple runs show variation and trends.
- **Ignore outliers**: First runs may be slower due to cold caches or compilation delays.
- **Use similar inputs**: Changing input sizes can skew results in subtle ways.
- **Test in realistic environments**: Browser vs. Node.js performance may vary significantly.

There are also libraries like Benchmark.js that provide more robust benchmarking capabilities, including statistical analysis.

2.3.6 Conclusion

In the real world, algorithm performance is affected by far more than Big-O. JIT compilation, garbage collection, memory access patterns, and engine-specific quirks all play a role. As a JavaScript developer, you don't need to understand the internals of every engine—but being aware of these factors helps you write code that's not just theoretically sound, but also **practically fast**. Combine theory with testing to make informed performance decisions.

Chapter 3.

Mathematical Tools (Lightweight)

1. Logarithms, Exponents, and Series
2. Recurrence Relations (Brief)
3. Growth Rates of Functions

3 Mathematical Tools (Lightweight)

3.1 Logarithms, Exponents, and Series

When analyzing algorithms, we often encounter mathematical patterns—particularly **logarithms**, **exponents**, and **series**. These concepts may sound intimidating, but they are surprisingly intuitive and essential for understanding how algorithms grow and behave at scale. This section introduces these ideas with real-world analogies and simple JavaScript examples.

3.1.1 Exponents: Rapid Growth

An **exponent** represents repeated multiplication. If you see `2 `, it means multiplying 2 by itself `n` times. Exponential functions grow **very fast** as `n` increases.

Full runnable code:

```
function powerOfTwo(n) {
  return 2 ** n;
}

console.log(powerOfTwo(4)); // 16
console.log(powerOfTwo(10)); // 1024
```

In algorithm analysis, exponential time (`O(2 )`) often means the algorithm becomes **infeasible for large inputs**. For example, a brute-force solution to the Traveling Salesman Problem checks all possible routes and runs in exponential time.

Real-world analogy: Think of folding a piece of paper in half repeatedly. After just 10 folds, it's over 1,000 times thicker!

3.1.2 Logarithms: The Opposite of Exponents

A **logarithm** answers the question: "To what power must we raise a number (the base) to get another number?"

For example: `log (8) = 3` because $2^3 = 8$.

In computer science, **binary logarithms** (`log `) are common, since many algorithms involve splitting data in half (e.g. binary search).

Full runnable code:

```
function binaryLog(n) {
  return Math.log2(n);
}

console.log(binaryLog(8)); // 3
console.log(binaryLog(1024)); // 10
```

Logarithmic time (`O(log n)`) is **very efficient**. An algorithm with this complexity, like binary search, becomes only slightly slower as the input grows larger.

Real-world analogy: Imagine guessing a number between 1 and 1,000. If you always guess the midpoint and eliminate half the possibilities each time, you'll find the answer in at most `log (1000)  10` steps.

3.1.3 Comparing Growth Rates Visually

Let's compare how different functions grow:

Full runnable code:

```
for (let n = 1; n <= 64; n *= 2) {
  const linear = n;
  const quadratic = n * n;
  const logarithmic = Math.log2(n);
  console.log(`n=${n}, log (n)=${logarithmic.toFixed(2)}, n=${linear}, n²=${quadratic}`);
}
```

Output snippet:

```
n=1, log (n)=0.00, n=1, n²=1
n=2, log (n)=1.00, n=2, n²=4
n=4, log (n)=2.00, n=4, n²=16
n=8, log (n)=3.00, n=8, n²=64
n=16, log (n)=4.00, n=16, n²=256
...
```

This clearly shows how **logarithmic growth is much slower** than linear or quadratic growth. That's why `O(log n)` algorithms are highly scalable.

3.1.4 Series: Summing Patterns

A **series** is the sum of a sequence of numbers. Algorithms that involve loops or recursive calls often have runtimes that follow arithmetic or geometric series.

Example: Arithmetic Series (1 + 2 + 3 + ... + n):

Full runnable code:

```
function arithmeticSum(n) {
  return (n * (n + 1)) / 2;
}

console.log(arithmeticSum(100)); // 5050
```

This formula, discovered by Gauss, helps us analyze nested loops and recursive functions that grow in a predictable way.

3.1.5 Final Thoughts

These basic mathematical tools—exponents for explosive growth, logarithms for efficient reduction, and series for cumulative behavior—help explain why some algorithms are fast and others are not. Even a light understanding of these concepts empowers you to make better design decisions, interpret Big-O notation more clearly, and write more efficient JavaScript code. As we continue through this book, these ideas will come up often—especially in sorting, searching, and recursive algorithms.

3.2 Recurrence Relations (Brief)

When analyzing recursive algorithms, it's often helpful to express their runtime using **recurrence relations**. A recurrence relation is an equation that defines the cost of a problem in terms of the cost of smaller subproblems. In essence, it's a mathematical way to describe the **self-referential nature** of recursion.

This concept may sound abstract, but it's extremely practical—especially when analyzing classic recursive algorithms like **Fibonacci numbers**, **merge sort**, or **binary search**.

3.2.1 What Is a Recurrence Relation?

A **recurrence relation** expresses the time complexity `T(n)` of an algorithm based on its subproblem size(s). Here's a simple format:

`T(n) = T(n - 1) + c`

This might describe a recursive function that reduces the input by 1 each time and does a constant amount of work (`c`) per call. Solving the recurrence tells us how the total cost grows as `n` increases.

3.2.2 Example: Recursive Fibonacci

Let's look at the Fibonacci sequence, where each number is the sum of the two preceding ones:

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

We can express the time cost as:

`T(n) = T(n - 1) + T(n - 2) + c`

This recurrence means that for each `n`, we do two recursive calls—one with `n - 1` and one with `n - 2`—plus a small amount of extra work (`c`). Solving this recurrence yields **exponential time complexity**, specifically **O(2)**.

You can verify the slowness by timing larger calls:

```
const t1 = performance.now();
console.log(fib(35)); // Very slow!
const t2 = performance.now();
console.log(`fib(35) took ${(t2 - t1).toFixed(2)} ms`);
```

3.2.3 Improving with Memoization

By storing intermediate results, we can avoid redundant calls:

```
function fibMemo(n, memo = {}) {
  if (n in memo) return memo[n];
  if (n <= 1) return n;
  memo[n] = fibMemo(n - 1, memo) + fibMemo(n - 2, memo);
  return memo[n];
}
```

This version turns the recurrence into:

`T(n) = T(n - 1) + c`

Because each subproblem is only solved once, the time complexity becomes **O(n)**—a huge improvement.

3.2.4 General Strategy for Recurrence Analysis

To analyze a recursive algorithm:

1. **Write the recurrence** based on how the function calls itself.
2. **Identify the base case**, which gives a fixed cost (e.g., `T(1) = c`).

3. **Solve or estimate the recurrence** using pattern recognition or mathematical tools.

For example, merge sort has the recurrence:

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

This resolves to **O(n log n)**, showing that dividing and merging is highly efficient.

3.2.5 Final Thoughts

Recurrence relations give us a structured way to understand recursive algorithm performance. While we won't dive into complex solutions here, even simple recurrences help us estimate and compare algorithms. As you learn more recursive techniques in JavaScript, keep this tool in mind—it will sharpen your ability to reason about code performance at a deeper level.

3.3 Growth Rates of Functions

One of the most important skills in algorithm analysis is understanding how different functions grow as input size increases. Growth rate determines how well an algorithm will scale, especially when handling large amounts of data. By comparing growth rates—**constant**, **logarithmic**, **linear**, **quadratic**, and **exponential**—you can predict performance and choose the right algorithm for the job.

3.3.1 Why Growth Rates Matter

Imagine two algorithms that solve the same problem:

- One takes 10 milliseconds for 100 inputs and scales linearly (`O(n)`).
- The other takes 1 millisecond for 100 inputs but grows quadratically ($O(n^2)$).

Which is better? At small inputs, the second one seems faster. But with 10,000 inputs, the linear algorithm might finish in under a second, while the quadratic one could take **hours**.

Growth rate dominates actual speed when input size becomes large.

3.3.2 Common Growth Rates

Here's a table comparing how different complexities grow as `n` increases:

n	O(1)	O(log n)	O(n)	$O(n^2)$	O(2)
1	1	0	1	1	2
10	1	3.3	10	100	1024
100	1	6.6	100	10,000	1.27e30
1,000	1	9.9	1,000	1,000,000	∞

You can see that **logarithmic growth** barely increases, while **exponential growth** explodes even with modest input sizes.

3.3.3 JavaScript Examples: Comparing Growth

Let's simulate how long various functions take as input grows:

Full runnable code:

```
function constantFunction(n) {
  return n + 1;
}

function linearFunction(n) {
  let sum = 0;
  for (let i = 0; i < n; i++) {
    sum++;
  }
  return sum;
}

function quadraticFunction(n) {
  let count = 0;
  for (let i = 0; i < n; i++) {
    for (let j = 0; j < n; j++) {
      count++;
    }
  }
  return count;
}

function exponentialFunction(n) {
  if (n <= 1) return 1;
  return exponentialFunction(n - 1) + exponentialFunction(n - 1);
}

// Now, let's compare their execution times:

function benchmark(fn, n) {
  const t1 = performance.now();
  fn(n);
  const t2 = performance.now();
  console.log(`${fn.name}(${n}) took ${(t2 - t1).toFixed(2)} ms`);
}
```

```
const input = 500;
benchmark(constantFunction, input);
benchmark(linearFunction, input);
benchmark(quadraticFunction, input);

benchmark(exponentialFunction, 20); // Smaller input to avoid long delay
```

Sample Output:

```
constantFunction(500) took 0.01 ms
linearFunction(500) took 0.03 ms
quadraticFunction(500) took 18.12 ms
exponentialFunction(20) took 91.75 ms
```

This shows how real execution time increases sharply for higher-complexity algorithms, even at moderate input sizes.

3.3.4 Visualizing Growth

Even without a graphing library, you can get a sense of growth:

Full runnable code:

```
for (let n = 1; n <= 128; n *= 2) {
  console.log(`n=${n}, log(n)=${Math.log2(n)}, n²=${n * n}, 2ⁿ=${2 ** n}`);
}
```

Output (trimmed):

```
n=1, log(n)=0, n²=1, 2ⁿ=2
n=2, log(n)=1, n²=4, 2ⁿ=4
n=4, log(n)=2, n²=16, 2ⁿ=16
n=8, log(n)=3, n²=64, 2ⁿ=256
n=16, log(n)=4, n²=256, 2ⁿ=65536
```

This highlights how exponential functions quickly outpace all others.

3.3.5 Conclusion

Understanding growth rates is fundamental to analyzing and comparing algorithms. Big-O notation tells us about these rates, but visualizing and testing them in JavaScript makes the concept real. As you design algorithms, always consider how performance changes with input size—because small inefficiencies grow into big problems faster than you think.

Chapter 4.

Sorting and Order: Elementary Sorting Algorithms

1. Bubble Sort
2. Insertion Sort
3. Selection Sort

4 Sorting and Order: Elementary Sorting Algorithms

4.1 Bubble Sort

Bubble sort is one of the simplest sorting algorithms—both to understand and to implement. It works by **repeatedly stepping through a list**, comparing adjacent elements, and **swapping them if they are in the wrong order**. This "bubbling" process pushes the largest unsorted element to the end of the array with each full pass, much like how bubbles rise to the surface of water.

Despite its simplicity, bubble sort is **inefficient for large datasets** and is rarely used in production code. However, it's a great educational tool for introducing algorithmic thinking, comparisons, and swaps.

4.1.1 How Bubble Sort Works

Let's break it down step-by-step with an example. Consider this array:

`[5, 3, 8, 4, 2]`

Pass 1:

- Compare 5 and 3 → Swap → `[3, 5, 8, 4, 2]`
- Compare 5 and 8 → No Swap
- Compare 8 and 4 → Swap → `[3, 5, 4, 8, 2]`
- Compare 8 and 2 → Swap → `[3, 5, 4, 2, 8]`

Largest number (8) is now at the end.

Pass 2:

- Compare 3 and 5 → No Swap
- Compare 5 and 4 → Swap → `[3, 4, 5, 2, 8]`
- Compare 5 and 2 → Swap → `[3, 4, 2, 5, 8]`

And so on, until the array is fully sorted.

Run the following code in browser to see the demo:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Bubble Sort Step Visualizer</title>
  <style>
    canvas {
      border: 1px solid #ccc;
      display: block;
      margin: 20px auto;
```

```
    }
    button {
      display: block;
      margin: 0 auto;
      padding: 10px 20px;
      font-size: 16px;
    }
  </style>
</head>
<body>

<canvas id="canvas" width="500" height="300"></canvas>
<button id="next">Show Next Step in bubble sort</button>

<script>
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');
const nextBtn = document.getElementById('next');

// Settings
const array = [100, 200, 80, 160, 120];
const barWidth = 60;
const spacing = 20;
const baseY = 280;

// Bubble Sort indices
let i = 0;
let j = 0;
let phase = 'compare'; // 'compare' or 'swap'
let animProgress = 0;
let animationId = null;
let isSwapping = false;

function drawArray(highlightA = -1, highlightB = -1, swapProgress = 0) {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  array.forEach((height, idx) => {
    const x = spacing + idx * (barWidth + spacing);
    const y = baseY - height;

    ctx.fillStyle = (idx === highlightA || idx === highlightB) ? 'red' : 'steelblue';

    let offsetX = 0;
    if (idx === highlightA && swapProgress) offsetX = (barWidth + spacing) * swapProgress;
    if (idx === highlightB && swapProgress) offsetX = -(barWidth + spacing) * swapProgress;

    ctx.fillRect(x + offsetX, y, barWidth, height);
    ctx.fillStyle = 'black';
    ctx.fillText(array[idx], x + offsetX + 20, baseY + 15);
  });
}

function animateSwap(a, b, callback) {
  isSwapping = true;
  animProgress = 0;

  function step() {
    animProgress += 0.05;
    if (animProgress >= 1) {
```

```
      // Swap values in the array
      [array[a], array[b]] = [array[b], array[a]];
      isSwapping = false;
      drawArray();
      cancelAnimationFrame(animationId);
      callback();
      return;
    }
    drawArray(a, b, animProgress);
    animationId = requestAnimationFrame(step);
  }

  step();
}

function stepSort() {
  if (isSwapping) return;

  if (i < array.length - 1) {
    if (j < array.length - i - 1) {
      const a = j;
      const b = j + 1;

      if (phase === 'compare') {
        drawArray(a, b); // Highlight comparison
        phase = 'swap';
      } else if (phase === 'swap') {
        if (array[a] > array[b]) {
          animateSwap(a, b, () => {
            j++;
            phase = 'compare';
          });
        } else {
          drawArray(); // No swap needed
          j++;
          phase = 'compare';
        }
      }
    } else {
      j = 0;
      i++;
    }
  } else {
    drawArray(); // Final state
    nextBtn.disabled = true;
  }
}

// Initial draw
drawArray();

// Button handler
nextBtn.addEventListener('click', stepSort);
</script>

</body>
</html>
```

4.1.2 JavaScript Implementation

Here's a clean implementation of bubble sort in JavaScript, with comments to explain each part:

Full runnable code:

```javascript
function bubbleSort(arr) {
  let swapped;
  const n = arr.length;

  do {
    swapped = false;

    for (let i = 0; i < n - 1; i++) {
      // Compare adjacent elements
      if (arr[i] > arr[i + 1]) {
        // Swap if elements are in the wrong order
        [arr[i], arr[i + 1]] = [arr[i + 1], arr[i]];
        swapped = true;
      }
    }
    // Repeat the loop if any swaps occurred
  } while (swapped);

  return arr;
}

// Example
console.log(bubbleSort([5, 3, 8, 4, 2])); // Output: [2, 3, 4, 5, 8]
```

This version uses a `do...while` loop to continue looping until a full pass completes with no swaps—indicating the array is sorted.

4.1.3 Visualization of the Sorting Process

Here's a text-based step-by-step visualization of the array `[5, 3, 8, 4, 2]`:

```
Start:      [5, 3, 8, 4, 2]
Pass 1:     [3, 5, 4, 2, 8]
Pass 2:     [3, 4, 2, 5, 8]
Pass 3:     [3, 2, 4, 5, 8]
Pass 4:     [2, 3, 4, 5, 8]
Sorted:     [2, 3, 4, 5, 8]
```

Each pass pushes the next-largest number to its final position.

4.1.4 Time Complexity

- **Worst-case**: $O(n^2)$ — when the array is in reverse order.
- **Best-case**: $O(n)$ — if the array is already sorted (with an optimization that checks if any swaps occurred).
- **Space complexity**: $O(1)$ — in-place sorting.

4.1.5 When Should You Use Bubble Sort?

In practice: **almost never**.

Bubble sort is vastly outperformed by more advanced algorithms like quicksort or merge sort, especially as input sizes grow. Its **only real use case** is educational—teaching beginners how sorting algorithms work by walking through a simple, intuitive process.

4.1.6 Summary

Bubble sort is a stepping stone for understanding how comparisons and swaps can lead to a sorted array. It is easy to code, easy to visualize, and sets the stage for deeper algorithmic thinking. While inefficient in real-world applications, its simplicity makes it a classic example for learning sorting fundamentals.

4.2 Insertion Sort

Insertion sort is a simple and intuitive sorting algorithm that builds the final sorted array one element at a time. It works similarly to how people often sort playing cards in their hands—by inserting each new card into its correct position among the previously sorted cards.

While not efficient on large unsorted datasets, insertion sort shines when the array is **nearly sorted** or **very small**, making it a practical choice in specific scenarios like hybrid sorting algorithms or early exit conditions.

4.2.1 How Insertion Sort Works

The algorithm starts with the assumption that the **first element is already sorted**, and then it:

1. Picks the next element.
2. Compares it with elements before it.
3. Shifts larger elements one position to the right.
4. Inserts the picked element into its correct spot.

Let's walk through an example with this array:

`[4, 3, 1, 5, 2]`

- First element `4` is already "sorted".
- Compare `3` with `4` → Insert `3` before `4`: `[3, 4, 1, 5, 2]`
- Compare `1` with `4`, `3` → Insert at the start: `[1, 3, 4, 5, 2]`
- Compare `5` → Already in correct place.
- Compare `2` with `5`, `4`, `3` → Insert before `3`: `[1, 2, 3, 4, 5]`

Run the following code in browser to see the demo:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Insertion Sort with Detailed Step Explanation</title>
  <style>
    #list {
      font-family: monospace;
      text-align: center;
      margin-top: 40px;
      font-size: 24px;
      letter-spacing: 12px;
    }
    #explanation {
      text-align: center;
      margin-top: 20px;
      font-size: 18px;
      font-family: sans-serif;
    }
    .sorted {
      color: black;
    }
    .unsorted {
      color: gray;
    }
    .key {
      color: red;
    }
    .empty {
      visibility: hidden;
    }
    button {
      display: block;
      margin: 30px auto;
      padding: 10px 20px;
      font-size: 18px;
    }
  </style>
</head>
```

```
<body>

<div id="list"></div>
<div id="explanation"></div>
<button id="next">Next Step</button>

<script>
const values = [3, 2, 1, 6, 5, 7, 9, 0, 4, 8];
let i = 1;
let j;
let key = null;
let originalIndex = null;
let phase = 'start'; // 'start', 'shift', 'insert'
let explanationDiv = document.getElementById('explanation');

function renderArray({ keyIndex = -1, compareIndex = -1, sortedEnd = 0, hideIndex = -1 } = {}) {
  const listDiv = document.getElementById('list');
  listDiv.innerHTML = '';

  values.forEach((val, idx) => {
    const span = document.createElement('span');

    if (idx === hideIndex) {
      span.textContent = val;
      span.className = 'empty';
    } else if (idx === keyIndex) {
      span.textContent = val;
      span.className = 'key';
    } else if (idx < sortedEnd) {
      span.textContent = val;
      span.className = 'sorted';
    } else {
      span.textContent = val;
      span.className = 'unsorted';
    }

    listDiv.appendChild(span);
  });
}

function explain(msg) {
  explanationDiv.textContent = msg;
}

function step() {
  if (i >= values.length) {
    renderArray({ sortedEnd: values.length });
    explain("Sorting complete.");
    document.getElementById('next').disabled = true;
    return;
  }

  if (phase === 'start') {
    key = values[i];
    originalIndex = i;
    j = i - 1;
    renderArray({ keyIndex: i, compareIndex: j, sortedEnd: i });
    explain(`Compare ${values[j]} and ${key} (key at index ${i})`);
```

```
    phase = 'shift';
  } else if (phase === 'shift') {
    if (j >= 0 && values[j] > key) {
      values[j + 1] = values[j]; // shift right
      renderArray({ keyIndex: originalIndex, hideIndex: j + 1, sortedEnd: i });
      explain(`Shift ${values[j]} from index ${j} → index ${j + 1}`);
      j--;
    } else {
      phase = 'insert';
    }
  } else if (phase === 'insert') {
    values[j + 1] = key;
    renderArray({ keyIndex: j + 1, sortedEnd: i + 1 });
    explain(`Insert ${key} (from index ${originalIndex}) at index ${j + 1}`);
    i++;
    phase = 'start';
  }
}

// Initial render
renderArray({ sortedEnd: 1 });
explain("Start with first element sorted");

// Button
document.getElementById('next').addEventListener('click', step);
</script>

</body>
</html>
```

4.2.2 JavaScript Implementation

Here's a clean implementation of insertion sort with explanatory comments:

Full runnable code:

```
function insertionSort(arr) {
  for (let i = 1; i < arr.length; i++) {
    let current = arr[i];       // Element to insert
    let j = i - 1;

    // Shift elements to the right to make space for insertion
    while (j >= 0 && arr[j] > current) {
      arr[j + 1] = arr[j];
      j--;
    }

    // Insert the element at its correct position
    arr[j + 1] = current;
  }

  return arr;
}
```

```
// Example
console.log(insertionSort([4, 3, 1, 5, 2])); // Output: [1, 2, 3, 4, 5]
```

4.2.3 Visualizing Insertion Sort

Let's visualize the sorting process on `[4, 3, 1, 5, 2]`:

```
Initial:      [4, 3, 1, 5, 2]
Step 1:       [3, 4, 1, 5, 2]   // Insert 3 before 4
Step 2:       [1, 3, 4, 5, 2]   // Insert 1 before 3
Step 3:       [1, 3, 4, 5, 2]   // 5 already in place
Step 4:       [1, 2, 3, 4, 5]   // Insert 2 before 3
```

The key insight is that each step maintains a growing "sorted" section of the array.

4.2.4 Performance Comparison with Bubble Sort

Let's benchmark insertion sort and bubble sort on a small dataset using `performance.now()`:

Full runnable code:

```
function bubbleSort(arr) {
  let swapped;
  do {
    swapped = false;
    for (let i = 0; i < arr.length - 1; i++) {
      if (arr[i] > arr[i + 1]) {
        [arr[i], arr[i + 1]] = [arr[i + 1], arr[i]];
        swapped = true;
      }
    }
  } while (swapped);
  return arr;
}
function insertionSort(arr) {
  for (let i = 1; i < arr.length; i++) {
    let current = arr[i];       // Element to insert
    let j = i - 1;

    // Shift elements to the right to make space for insertion
    while (j >= 0 && arr[j] > current) {
      arr[j + 1] = arr[j];
      j--;
    }

    // Insert the element at its correct position
    arr[j + 1] = current;
```

```
  }

  return arr;
}
function benchmarkSort(name, sortFn, data) {
  const arr = [...data]; // Clone input
  const start = performance.now();
  sortFn(arr);
  const end = performance.now();
  console.log(`${name} took ${(end - start).toFixed(3)} ms`);
}

const smallSorted = Array.from({ length: 1000 }, (_, i) => i);
const smallReversed = [...smallSorted].reverse();

console.log("Nearly sorted data:");
benchmarkSort("Insertion Sort", insertionSort, smallSorted);
benchmarkSort("Bubble Sort", bubbleSort, smallSorted);

console.log("Reversed data:");
benchmarkSort("Insertion Sort", insertionSort, smallReversed);
benchmarkSort("Bubble Sort", bubbleSort, smallReversed);
```

Expected Results (Varies by System):

```
Nearly sorted data:
Insertion Sort took 0.12 ms
Bubble Sort took 5.87 ms

Reversed data:
Insertion Sort took 6.43 ms
Bubble Sort took 10.92 ms
```

Insertion sort clearly **outperforms bubble sort** on nearly sorted data and even holds its own on reversed input.

4.2.5 Use Cases and Limitations

When Insertion Sort Is Useful:

- Small arrays (e.g., < 20 elements)
- Nearly sorted data
- As a helper in hybrid sorts like Timsort

When to Avoid It:

- Large, randomly ordered datasets
- Time-critical code requiring `O(n log n)` efficiency

4.2.6 Conclusion

Insertion sort is more practical than bubble sort and provides better performance in real-world scenarios involving small or mostly sorted datasets. Its simple mechanics make it ideal for understanding how sorting works and serve as a foundation for learning more advanced algorithms. While it's not suitable for large-scale data processing, insertion sort is a valuable tool in the algorithmic toolkit.

4.3 Selection Sort

3. Selection Sort

Selection sort is a straightforward comparison-based sorting algorithm that works by **repeatedly selecting the smallest (or largest) element** from the unsorted portion of the array and placing it in its correct position. Like bubble sort and insertion sort, it is not efficient on large datasets but is helpful for understanding basic sorting logic.

4.3.1 How Selection Sort Works

The algorithm maintains two parts in the array:

- A **sorted portion** (grows from left to right),
- An **unsorted portion** (shrinks as elements are placed correctly).

In each pass:

1. Find the minimum element in the unsorted portion.
2. Swap it with the first element of the unsorted portion.

Let's walk through an example using `[5, 3, 8, 4, 2]`.

- Pass 1: Minimum is `2` → swap with `5` → `[2, 3, 8, 4, 5]`
- Pass 2: Minimum is `3` → already in place
- Pass 3: Minimum is `4` → swap with `8` → `[2, 3, 4, 8, 5]`
- Pass 4: Minimum is `5` → swap with `8` → `[2, 3, 4, 5, 8]`

Each pass places one element in its final position.

Run the following code in browser to see the demo:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Selection Sort Visualized</title>
```

```
  <style>
    #list {
      font-family: monospace;
      text-align: center;
      margin-top: 40px;
      font-size: 24px;
      letter-spacing: 12px;
    }
    #explanation {
      text-align: center;
      margin-top: 20px;
      font-size: 18px;
      font-family: sans-serif;
    }
    .sorted {
      color: black;
    }
    .unsorted {
      color: gray;
    }
    .selected {
      color: red;
    }
    .min {
      color: goldenrod;
    }
    button {
      display: block;
      margin: 30px auto;
      padding: 10px 20px;
      font-size: 18px;
    }
  </style>
</head>
<body>

<div id="list"></div>
<div id="explanation"></div>
<button id="next">Next Step</button>

<script>
const values = [100, 200, 80, 160, 120];
let i = 0;                  // current index to place min value
let j = 1;                  // current scan index
let minIndex = 0;           // index of the current minimum
let phase = 'scan';         // 'scan' | 'swap' | 'next'
const explanation = document.getElementById('explanation');

function render({ selected = -1, min = -1, sortedEnd = 0 } = {}) {
  const list = document.getElementById('list');
  list.innerHTML = '';

  values.forEach((val, idx) => {
    const span = document.createElement('span');
    span.textContent = val + ' ';
    if (idx < sortedEnd) {
      span.className = 'sorted';
    } else if (idx === selected) {
```

```
      span.className = 'selected';
    } else if (idx === min) {
      span.className = 'min';
    } else {
      span.className = 'unsorted';
    }
    list.appendChild(span);
  });
}

function explain(msg) {
  explanation.textContent = msg;
}

function step() {
  if (i >= values.length) {
    render({ sortedEnd: values.length });
    explain("Sorting complete.");
    document.getElementById('next').disabled = true;
    return;
  }

  if (phase === 'scan') {
    render({ selected: i, min: minIndex, sortedEnd: i });
    if (j < values.length) {
      explain(`Compare ${values[j]} and current min ${values[minIndex]}`);
      if (values[j] < values[minIndex]) {
        minIndex = j;
      }
      j++;
    } else {
      phase = 'swap';
    }
  } else if (phase === 'swap') {
    if (minIndex !== i) {
      explain(`Swap ${values[i]} (index ${i}) with ${values[minIndex]} (index ${minIndex})`);
      [values[i], values[minIndex]] = [values[minIndex], values[i]];
    } else {
      explain(`No swap needed, ${values[i]} is already minimum`);
    }
    render({ sortedEnd: i + 1 });
    phase = 'next';
  } else if (phase === 'next') {
    i++;
    j = i + 1;
    minIndex = i;
    if (i < values.length) {
      phase = 'scan';
      render({ selected: i, min: minIndex, sortedEnd: i });
      explain(`Start scanning for minimum from index ${i}`);
    } else {
      step(); // end
    }
  }
}

// Initial draw
render({ sortedEnd: 0 });
```

```
explain("Start Selection Sort");

// Button
document.getElementById('next').addEventListener('click', step);
</script>

</body>
</html>
```

4.3.2 JavaScript Implementation

Here's a clean and annotated version of selection sort in JavaScript:

Full runnable code:

```
function selectionSort(arr) {
  const n = arr.length;

  for (let i = 0; i < n - 1; i++) {
    let minIndex = i;

    // Find the index of the minimum element
    for (let j = i + 1; j < n; j++) {
      if (arr[j] < arr[minIndex]) {
        minIndex = j;
      }
    }

    // Swap the found minimum element with the first unsorted element
    if (minIndex !== i) {
      [arr[i], arr[minIndex]] = [arr[minIndex], arr[i]];
    }
  }

  return arr;
}

// Example usage
console.log(selectionSort([5, 3, 8, 4, 2])); // Output: [2, 3, 4, 5, 8]
```

4.3.3 Performance Characteristics

Metric	Value
Time (Worst)	$O(n^2)$
Time (Best)	$O(n^2)$
Time (Average)	$O(n^2)$

Metric	Value
Space Complexity	O(1) (in-place)
Stable Sort	NO (not stable)

Unlike bubble or insertion sort, **selection sort always performs $O(n^2)$ comparisons**, regardless of input order. This is because it scans the remaining unsorted elements to find the minimum, even if the array is already sorted.

However, it performs **fewer swaps** than bubble sort—only one per pass—making it slightly better in environments where writing to memory is expensive (e.g., flash memory).

4.3.4 Visualizing the Sorting Steps

Initial: `[5, 3, 8, 4, 2]`

Pass 1 (min = 2): `[2, 3, 8, 4, 5]`

Pass 2 (min = 3): `[2, 3, 8, 4, 5]`

Pass 3 (min = 4): `[2, 3, 4, 8, 5]`

Pass 4 (min = 5): `[2, 3, 4, 5, 8]`

4.3.5 Comparison to Other Elementary Sorts

Feature	Bubble Sort	Insertion Sort	Selection Sort
Best-Case Time	O(n)	O(n)	$O(n^2)$
Worst-Case Time	$O(n^2)$	$O(n^2)$	$O(n^2)$
Stable			NO
Swaps per Pass	Many	Few	One
Best for	Teaching	Nearly sorted	Small datasets

Insertion sort tends to outperform both bubble and selection sort on real data. However, selection sort's minimal swapping can be an advantage in specialized systems where write operations are costly.

4.3.6 Conclusion

Selection sort is simple to understand and implement, and it introduces the concept of **finding an extremum (minimum/maximum)** repeatedly. While not efficient on large or complex datasets, it's useful for small arrays or when minimizing memory writes is important. Understanding how it works builds a strong foundation for grasping more advanced comparison-based sorting techniques later in the book.

Chapter 5.

Sorting and Order: Divide and Conquer

1. Merge Sort
2. Quick Sort
3. Time Complexity and Recursion Trees

5 Sorting and Order: Divide and Conquer

5.1 Merge Sort

Merge sort is a classic example of a **divide-and-conquer** algorithm. It works by **dividing** the input array into smaller subarrays, **conquering** each by recursively sorting them, and finally **combining** (merging) the sorted subarrays back into a fully sorted array. It is one of the most efficient general-purpose sorting algorithms and is particularly useful for sorting large datasets.

5.1.1 The Divide-and-Conquer Paradigm

Divide-and-conquer is a powerful algorithm design technique that follows three main steps:

1. **Divide**: Split the problem into smaller subproblems.
2. **Conquer**: Solve each subproblem recursively.
3. **Combine**: Merge the solutions of the subproblems into a final result.

In merge sort:

- **Divide**: The array is split into two halves.
- **Conquer**: Each half is sorted recursively using merge sort.
- **Combine**: Two sorted halves are merged into one sorted array.

5.1.2 Merge Sort in Action

Let's say we want to sort the array `[4, 1, 7, 3]`.

1. **Divide**:
    - `[4, 1]` and `[7, 3]`
    - Then further: `[4]`, `[1]`, `[7]`, `[3]`
2. **Conquer** (Sort small arrays):
    - `[4]` and `[1]` → merge to `[1, 4]`
    - `[7]` and `[3]` → merge to `[3, 7]`
3. **Combine**:
    - `[1, 4]` and `[3, 7]` → merge to `[1, 3, 4, 7]`

The result is a fully sorted array.

Run the following code in browser to see the demo:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Merge Sort - Persistent Canvas Visualization</title>
  <style>
    canvas {
      display: block;
      margin: 20px auto;
      border: 1px solid #ccc;
      background-color: #fff;
    }
    button {
      display: block;
      margin: 10px auto;
      padding: 10px 20px;
      font-size: 18px;
    }
  </style>
</head>
<body>

<canvas id="canvas" width="600" height="700"></canvas>
<button id="next">Next Step</button>

<script>
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');

const array = [8, 7, 3, 4, 9, 2, 1, 6, 5, 0];
const boxWidth = 50;
const boxHeight = 30;
const spacing = 10;

let steps = [];
let currentStep = 0;

// Prepare all steps: each will be drawn at y = depth * levelHeight
const levelHeight = 70;
function mergeSortTrace(arr, depth = 0, offset = 0) {
  const n = arr.length;
  const start = offset;
  const end = offset + n - 1;

  const stepSplit = {
    type: 'split',
    array: [...arr],
    depth,
    offset,
    start,
    end,
  };
  steps.push(stepSplit);

  if (n <= 1) return arr;

  const mid = Math.floor(n / 2);
  const left = arr.slice(0, mid);
```

```
  const right = arr.slice(mid);

  const sortedLeft = mergeSortTrace(left, depth + 1, offset);
  const sortedRight = mergeSortTrace(right, depth + 1, offset + mid);

  // Merge step
  const merged = [];
  let i = 0, j = 0;
  while (i < sortedLeft.length && j < sortedRight.length) {
    if (sortedLeft[i] < sortedRight[j]) {
      merged.push(sortedLeft[i++]);
    } else {
      merged.push(sortedRight[j++]);
    }
  }
  while (i < sortedLeft.length) merged.push(sortedLeft[i++]);
  while (j < sortedRight.length) merged.push(sortedRight[j++]);

  const stepMerge = {
    type: 'merge',
    array: [...merged],
    depth,
    offset,
    start,
    end,
  };
  steps.push(stepMerge);

  return merged;
}

// Draw a row of boxes representing the array at a given y offset
function drawArray(arr, startIndex, y, color = 'black') {
  ctx.font = '16px monospace';
  ctx.textAlign = 'center';
  ctx.textBaseline = 'middle';

  arr.forEach((val, i) => {
    const x = 50 + (startIndex + i) * (boxWidth + spacing);
    ctx.strokeStyle = '#000';
    ctx.fillStyle = '#fff';
    ctx.fillRect(x, y, boxWidth, boxHeight);
    ctx.strokeRect(x, y, boxWidth, boxHeight);

    ctx.fillStyle = color;
    ctx.fillText(val, x + boxWidth / 2, y + boxHeight / 2);
  });
}

// Show the original array once at the top
function drawOriginalArray() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  ctx.fillStyle = '#000';
  ctx.font = '18px sans-serif';
  ctx.fillText("Original Array:", 50, 20);
  drawArray(array, 0, 30, 'black');
}
```

```
// Draw all steps up to and including currentStep
function renderSteps() {
  drawOriginalArray();
  for (let s = 0; s <= currentStep; s++) {
    const step = steps[s];
    const y = 100 + step.depth * levelHeight;
    const color = step.type === 'split' ? '#666' : 'green';
    drawArray(step.array, step.offset, y, color);
  }
}

// Step forward
function nextStep() {
  if (currentStep < steps.length - 1) {
    currentStep++;
    renderSteps();
  } else {
    document.getElementById('next').disabled = true;
  }
}

// Initialize
mergeSortTrace(array);
renderSteps();
document.getElementById('next').addEventListener('click', nextStep);
</script>

</body>
</html>
```

5.1.3 JavaScript Implementation

Here's a step-by-step merge sort implementation with comments:

Full runnable code:

```
function mergeSort(arr) {
  // Base case: arrays with 0 or 1 element are already sorted
  if (arr.length <= 1) return arr;

  // Split array into two halves
  const mid = Math.floor(arr.length / 2);
  const left = arr.slice(0, mid);
  const right = arr.slice(mid);

  // Recursively sort both halves and merge them
  return merge(mergeSort(left), mergeSort(right));
}

function merge(left, right) {
  const result = [];
  let i = 0, j = 0;
```

```
  // Merge elements in sorted order
  while (i < left.length && j < right.length) {
    if (left[i] <= right[j]) {
      result.push(left[i++]);
    } else {
      result.push(right[j++]);
    }
  }

  // Add remaining elements (only one of these will run)
  return result.concat(left.slice(i)).concat(right.slice(j));
}

// Example
console.log(mergeSort([4, 1, 7, 3])); // Output: [1, 3, 4, 7]
```

5.1.4 Recursion Flow and Merging (Visualized)

To understand how the algorithm flows, here's a diagram for sorting `[4, 1, 7, 3]`:

```
mergeSort([4, 1, 7, 3])
+- mergeSort([4, 1])
   +- mergeSort([4]) → [4]
   +- mergeSort([1]) → [1]
   +- merge([4], [1]) → [1, 4]
+- mergeSort([7, 3])
   +- mergeSort([7]) → [7]
   +- mergeSort([3]) → [3]
   +- merge([7], [3]) → [3, 7]
+- merge([1, 4], [3, 7]) → [1, 3, 4, 7]
```

Each recursive call further splits the array until it reaches size 1. Then the merging phase begins, combining smaller sorted arrays into larger ones.

5.1.5 Why Merge Step Guarantees Sorted Output

The merge step always compares the **smallest elements** of the two subarrays and appends the smaller one to the result. Since each subarray is already sorted from previous recursive calls, the final merged array is guaranteed to be sorted.

5.1.6 Time and Space Complexity

Aspect	Value
Time (All Cases)	O(n log n)
Space	O(n) (due to extra arrays)
Stable Sort	(preserves relative order)

- **O(n log n)**: The array is split log (n) times, and merging takes O(n) time at each level.
- **Stable**: If two elements are equal, merge sort preserves their original order.

5.1.7 When to Use Merge Sort

Merge sort is ideal when:

- You need a **guaranteed O(n log n)** sort time.
- **Stability** is important (e.g., sorting objects by a secondary key).
- Working with **linked lists** (where merge can be done in-place).

It's often used in hybrid sorting algorithms like **Timsort**, which powers the built-in `Array.prototype.sort()` in many JavaScript engines.

5.1.8 Conclusion

Merge sort is a powerful, efficient, and elegant algorithm that exemplifies the divide-and-conquer paradigm. Its guaranteed performance, stability, and predictable recursion structure make it an essential tool in your algorithmic toolkit. By understanding how merge sort works—both conceptually and in code—you'll be better equipped to tackle complex sorting problems and recursion-based algorithms.

5.2 Quick Sort

Quick sort is one of the fastest and most commonly used sorting algorithms in practice. Like merge sort, it uses the **divide-and-conquer** strategy, but instead of merging sorted arrays, it focuses on **partitioning** the input array around a **pivot** element. Quick sort typically performs better than merge sort in real-world scenarios due to its in-place sorting and lower overhead.

5.2.1 The Quick Sort Strategy

Quick sort follows this approach:

1. **Choose a pivot** element from the array.
2. **Partition** the array into two subarrays:
    - Elements less than the pivot.
    - Elements greater than or equal to the pivot.
3. **Recursively apply quick sort** to the left and right subarrays.
4. Combine the results (though the combining is implicit, since sorting is done in-place).

Unlike merge sort, quick sort doesn't merge arrays—it simply ensures that the pivot ends up in its final sorted position, with smaller elements to the left and larger to the right.

5.2.2 Visual Example

Let's sort `[5, 3, 8, 4, 2, 7, 1]` with pivot `4`.

1. Pivot: `4`
2. Partition:
    - Left: `[3, 2, 1]` (less than 4)
    - Right: `[5, 8, 7]` (greater than 4)
3. Recurse on `[3, 2, 1]` and `[5, 8, 7]`
4. Final sorted result: `[1, 2, 3, 4, 5, 7, 8]`

Run the following code in browser to see the demo:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Quick Sort Visualizer with Canvas</title>
  <style>
    body {
      font-family: sans-serif;
      text-align: center;
      margin-top: 20px;
    }
    canvas {
      border: 1px solid #ccc;
      background: #fff;
      display: block;
      margin: 20px auto;
    }
    button {
```

```
      padding: 10px 20px;
      font-size: 16px;
      margin: 10px;
    }
    #explanation {
      font-size: 18px;
      margin-top: 10px;
    }
  </style>
</head>
<body>

<canvas id="canvas" width="600" height="400"></canvas>
<button id="next">Next Step</button>
<div id="explanation"></div>

<script>
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');
const explanation = document.getElementById('explanation');

const barWidth = 50;
const spacing = 10;
const bottom = 300;

const originalArray = [33, 10, 55, 71, 29, 3, 42];
let array = [...originalArray];
let steps = [];
let currentStep = 0;

// Prepare Quick Sort trace steps
function traceQuickSort(arr, start = 0, end = arr.length - 1) {
  if (start >= end) return;

  const pivotIndex = end;
  const pivotValue = arr[pivotIndex];
  let i = start;

  steps.push({ type: 'pivot', array: [...arr], pivot: pivotIndex });

  for (let j = start; j < end; j++) {
    steps.push({
      type: 'compare',
      array: [...arr],
      pivot: pivotIndex,
      j,
      i
    });

    if (arr[j] < pivotValue) {
      [arr[i], arr[j]] = [arr[j], arr[i]];
      steps.push({
        type: 'swap',
        array: [...arr],
        i,
        j
      });
      i++;
```

```
    }
  }

  [arr[i], arr[pivotIndex]] = [arr[pivotIndex], arr[i]];
  steps.push({
    type: 'pivot-swap',
    array: [...arr],
    pivotFinal: i
  });

  traceQuickSort(arr, start, i - 1);
  traceQuickSort(arr, i + 1, end);
}

function drawArrow(x1, y1, x2, y2) {
  ctx.beginPath();
  ctx.moveTo(x1, y1);
  ctx.bezierCurveTo(x1, y1 - 40, x2, y2 - 40, x2, y2);
  ctx.strokeStyle = 'blue';
  ctx.lineWidth = 2;
  ctx.stroke();

  const headlen = 8;
  const angle = Math.atan2(y2 - y1, x2 - x1);
  ctx.beginPath();
  ctx.moveTo(x2, y2);
  ctx.lineTo(x2 - headlen * Math.cos(angle - Math.PI / 6), y2 - headlen * Math.sin(angle - Math.PI / 6)
  ctx.lineTo(x2 - headlen * Math.cos(angle + Math.PI / 6), y2 - headlen * Math.sin(angle + Math.PI / 6)
  ctx.lineTo(x2, y2);
  ctx.fillStyle = 'blue';
  ctx.fill();
}

function drawBars(arr, highlight = {}) {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  const maxVal = Math.max(...arr);

  arr.forEach((val, index) => {
    const height = (val / maxVal) * 200;
    const x = index * (barWidth + spacing) + 50;
    const y = bottom - height;

    // Bar color
    ctx.fillStyle = 'gray';
    if (highlight.pivot === index) ctx.fillStyle = 'red';
    else if (highlight.j === index) ctx.fillStyle = 'orange';
    else if (highlight.i === index) ctx.fillStyle = 'gold';

    ctx.fillRect(x, y, barWidth, height);
    ctx.strokeRect(x, y, barWidth, height);

    ctx.fillStyle = 'black';
    ctx.font = '14px monospace';
    ctx.fillText(val, x + barWidth / 2, bottom + 15);
  });

  // Draw arrow from j to pivot
  if ('j' in highlight && 'pivot' in highlight) {
```

```
    const xj = highlight.j * (barWidth + spacing) + 50 + barWidth / 2;
    const xp = highlight.pivot * (barWidth + spacing) + 50 + barWidth / 2;
    drawArrow(xj, bottom + 5, xp, bottom + 5);
  }
}

// Animate a swap between bars
function animateSwap(arr, i, j, callback) {
  const frames = 10;
  let frame = 0;
  const fromX = i * (barWidth + spacing) + 50;
  const toX = j * (barWidth + spacing) + 50;
  const diff = (toX - fromX) / frames;

  function animate() {
    ctx.clearRect(0, 0, canvas.width, canvas.height);

    const maxVal = Math.max(...arr);
    arr.forEach((val, index) => {
      let x = index * (barWidth + spacing) + 50;
      if (index === i) x += frame * diff;
      else if (index === j) x -= frame * diff;

      const height = (val / maxVal) * 200;
      const y = bottom - height;

      ctx.fillStyle = (index === i || index === j) ? 'purple' : 'gray';
      ctx.fillRect(x, y, barWidth, height);
      ctx.strokeRect(x, y, barWidth, height);

      ctx.fillStyle = 'black';
      ctx.fillText(val, x + barWidth / 2, bottom + 15);
    });

    frame++;
    if (frame <= frames) requestAnimationFrame(animate);
    else callback();
  }

  animate();
}

function renderStep() {
  if (currentStep >= steps.length) return;

  const step = steps[currentStep];
  explanation.textContent = '';

  if (step.type === 'pivot') {
    drawBars(step.array, { pivot: step.pivot });
    explanation.textContent = `Select pivot: ${step.array[step.pivot]}`;
  } else if (step.type === 'compare') {
    drawBars(step.array, { j: step.j, pivot: step.pivot });
    explanation.textContent = `Compare arr[${step.j}] = ${step.array[step.j]} with pivot = ${step.array
  } else if (step.type === 'swap') {
    animateSwap(step.array, step.i, step.j, () => {
      drawBars(step.array);
      explanation.textContent = `Swap arr[${step.i}] = ${step.array[step.i]} with arr[${step.j}] = ${st
```

```
    });
  } else if (step.type === 'pivot-swap') {
    drawBars(step.array, { pivot: step.pivotFinal });
    explanation.textContent = `Place pivot at index ${step.pivotFinal}`;
  }
}

// Prepare steps and render first
traceQuickSort([...array]);
renderStep();

document.getElementById('next').addEventListener('click', () => {
  if (currentStep < steps.length - 1) {
    currentStep++;
    renderStep();
  } else {
    explanation.textContent = "Sorting complete.";
    document.getElementById('next').disabled = true;
  }
});
</script>

</body>
</html>
```

5.2.3 JavaScript Implementation

Here's a full quick sort implementation using the **Lomuto partition scheme** (simple and intuitive for beginners):

Full runnable code:

```
function quickSort(arr, left = 0, right = arr.length - 1) {
  if (left < right) {
    const pivotIndex = partition(arr, left, right);
    // Recursively sort elements before and after partition
    quickSort(arr, left, pivotIndex - 1);
    quickSort(arr, pivotIndex + 1, right);
  }
  return arr;
}

function partition(arr, left, right) {
  const pivot = arr[right]; // Pivot is the last element
  let i = left - 1;

  for (let j = left; j < right; j++) {
    if (arr[j] < pivot) {
      i++;
      [arr[i], arr[j]] = [arr[j], arr[i]]; // Swap
    }
  }
```

```
  // Place pivot in its correct position
  [arr[i + 1], arr[right]] = [arr[right], arr[i + 1]];
  return i + 1; // Return pivot index
}

// Example usage
console.log(quickSort([5, 3, 8, 4, 2, 7, 1]));
// Output: [1, 2, 3, 4, 5, 7, 8]
```

5.2.4 Recursion Flow Diagram

Here's a simplified call tree of how quick sort works on `[5, 3, 8, 4, 2]`:

```
quickSort([5, 3, 8, 4, 2])
+- pivot = 2 → [1st partition]
   +- quickSort([...left of pivot]) → [3, 1]
   +- quickSort([...right of pivot]) → [8, 5, 4]
     +- Further recursive calls...
```

At each level, the pivot reduces the unsorted region, shrinking the problem size.

5.2.5 Time Complexity

Case	Time Complexity	Description
Best	O(n log n)	Balanced partitions
Average	O(n log n)	Random data, good pivot
Worst	$O(n^2)$	Poor pivot (e.g., always smallest/largest)

- Worst-case arises when the pivot divides the array into very uneven parts (e.g., sorting a sorted array with last element as pivot).
- **Average case is very fast**, and modern implementations use strategies to avoid worst-case behavior.

5.2.6 Pivot Choice Matters

The efficiency of quick sort heavily depends on the choice of pivot. Here are some strategies:

- **Last element** (used above): simple but risky on sorted arrays.
- **Middle element**: a reasonable default.
- **Random element**: helps avoid worst-case.

- **Median-of-three**: choose the median of first, middle, and last—reduces chance of imbalance.

You can easily implement random pivoting in JavaScript by swapping a random index with the last one before partitioning:

```
function randomPartition(arr, left, right) {
  const randIndex = left + Math.floor(Math.random() * (right - left + 1));
  [arr[randIndex], arr[right]] = [arr[right], arr[randIndex]];
  return partition(arr, left, right);
}
```

5.2.7 Practical Tips for JS Developers

- **Use in-place sorting** when memory is limited.
- Avoid using quick sort on small arrays where insertion sort may be faster due to lower overhead.
- For large datasets or guaranteed performance, use merge sort or built-in methods like `Array.prototype.sort()` (which often use hybrid approaches).
- Be cautious of performance when sorting nearly sorted or reverse-sorted data with poor pivot selection.

5.2.8 Conclusion

Quick sort is a powerful and elegant algorithm that often outperforms merge sort in real-world use due to its in-place nature and average-case efficiency of O(n log n). Understanding how partitioning and pivot selection work gives you deeper control over its performance and applicability. As you continue learning sorting techniques, quick sort is a crucial algorithm to master, both in concept and implementation.

5.3 Time Complexity and Recursion Trees

Understanding the time complexity of recursive algorithms can be tricky at first, but recursion trees offer an intuitive and visual way to analyze how work is divided and accumulated in divide-and-conquer algorithms like **merge sort** and **quick sort**.

In this section, we'll learn how to:

- Use recursion trees to estimate time complexity,
- Apply this method to merge sort and quick sort,
- Connect the math to real code using JavaScript.

5.3.1 What Is a Recursion Tree?

A recursion tree is a diagram that maps out how a recursive algorithm breaks down a problem. Each node in the tree represents a recursive call, and the amount of **work** done at that level. By adding up the work across all levels, we can estimate the algorithm's total time complexity.

5.3.2 Merge Sort as a Recursion Tree

Merge sort splits the array into two halves at each step and does linear work (O(n)) during the merge step.

Let's visualize the recursion tree for sorting an array of size `n = 8`:

```
Level 0:           mergeSort(8)                  → O(8)
Level 1:     mergeSort(4)   mergeSort(4)         → O(4) + O(4)
Level 2:   m(2) m(2)       m(2) m(2)             → 4 × O(2)
Level 3:  m(1)m(1)m(1)m(1) m(1)m(1)m(1)m(1)      → 8 × O(1)
```

Each level does **O(n)** total work:

- Level 0: 1 call × O(8)
- Level 1: 2 calls × O(4)
- Level 2: 4 calls × O(2)
- Level 3: 8 calls × O(1)

Total number of levels: `log n` (splitting in half each time)

Total time:

`O(n) work per level × log n levels → O(n log n)`

5.3.3 JavaScript Code for Merge Sort Breakdown

```
function mergeSort(arr) {
  if (arr.length <= 1) return arr;

  const mid = Math.floor(arr.length / 2);
  const left = mergeSort(arr.slice(0, mid));
  const right = mergeSort(arr.slice(mid));

  return merge(left, right); // O(n) merge step
}
```

Each call to `mergeSort` divides the array in half and merges results. The recursion tree helps us track the depth and cost at each level.

5.3.4 Quick Sort and Recursion Tree Variability

Quick sort's recursion tree **depends on pivot choice**.

Best/Average Case

Each partition splits the array roughly in half, like merge sort.

```
Level 0:        quickSort(8)          → O(8)
Level 1:    qS(4)        qS(4)        → 2 × O(4)
Level 2: qS(2) qS(2) qS(2) qS(2)     → 4 × O(2)
...
```

Same pattern: `log n` levels, O(n) per level → **O(n log n)** time.

Worst Case

If the pivot is the smallest or largest element every time:

```
quickSort(8)
 +- quickSort(7)
     +- quickSort(6)
         ...
```

- Tree depth: n
- Work per level: O(n), O(n–1), …, O(1)

Total time:

`O(n²) (like a linked list of recursive calls)`

5.3.5 JavaScript Code for Recursive Depth

Here's how you can trace recursive depth in quick sort:

Full runnable code:

```
function quickSort(arr, depth = 0) {
  if (arr.length <= 1) return arr;

  const pivot = arr[arr.length - 1];
  const left = [], right = [];

  for (let i = 0; i < arr.length - 1; i++) {
    if (arr[i] < pivot) left.push(arr[i]);
    else right.push(arr[i]);
  }

  console.log(" ".repeat(depth * 2) + `quickSort: [${arr.join(", ")}]`);

  return [
```

```
    ...quickSort(left, depth + 1),
    pivot,
    ...quickSort(right, depth + 1)
  ];
}

quickSort([5, 3, 8, 1, 4]);
```

This logs recursive depth with indentation—mirroring the tree shape.

5.3.6 Intuition Behind the `log n` Factor

When an algorithm **cuts the input size in half** each time (like merge sort or best-case quick sort), it forms a **binary tree** of calls. A binary tree of depth `log n` means the recursion continues only about `log n` levels deep.

Combine this with `O(n)` work at each level, and you get the hallmark **O(n log n)** complexity of divide-and-conquer.

5.3.7 Summary

Recursion trees help demystify the time complexity of recursive algorithms:

- **Merge sort** always has O(n log n) time due to balanced recursion and linear merging.
- **Quick sort** is O(n log n) in the best case, but can degrade to $O(n^2)$ with poor pivot choices.
- The `log n` factor comes from the number of times you can divide the input in half.

By connecting recursion trees to your JavaScript implementations, you'll build stronger intuition about how recursive algorithms scale—an essential skill in algorithm design.

The following animation shows the steps in quick sort.

Run the following code in browser to see the demo:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Quick Sort Visualizer</title>
  <style>
    body {
      font-family: sans-serif;
      display: flex;
      flex-direction: row;
      margin: 0;
      padding: 0;
```

```
    }
    canvas {
      background: #fff;
      border-right: 1px solid #ccc;
    }
    #controls {
      padding: 20px;
      width: 250px;
    }
    #controls button {
      padding: 10px 20px;
      font-size: 16px;
      margin-bottom: 10px;
    }
    #stack {
      font-family: monospace;
      white-space: pre;
      background: #f9f9f9;
      padding: 10px;
      border: 1px solid #ccc;
      max-height: 400px;
      overflow-y: auto;
    }
  </style>
</head>
<body>

<div id="controls">
  <button id="next">Next Step</button>
  <div>Comparisons: <span id="cmp">0</span></div>
  <div>Swaps: <span id="swp">0</span></div>
  <h3>Call Stack</h3>
  <div id="stack"></div>
</div>
<canvas id="canvas" width="700" height="700"></canvas>


<script>
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");
const stackDisplay = document.getElementById("stack");
const cmpDisplay = document.getElementById("cmp");
const swpDisplay = document.getElementById("swp");

const boxSize = 40;
const spacing = 10;
const baseY = 40;
const levelHeight = 80;
const array = [33, 10, 55, 71, 29, 3, 42];

let comparisons = 0;
let swaps = 0;

let frames = [];
let currentFrame = 0;
let positions = []; // To track box positions

function generateQuickSortTrace(arr) {
```

```
  let steps = [];
  function recurse(start, end, depth, parentPos = null) {
    if (start > end) return;

    const pivot = end;
    steps.push({
      type: "partition",
      start, end, pivot, depth, parentPos,
      snapshot: arr.slice()
    });

    let i = start;
    for (let j = start; j < end; j++) {
      steps.push({
        type: "compare",
        i, j, pivot, array: arr.slice(), depth
      });
      comparisons++;
      if (arr[j] < arr[pivot]) {
        [arr[i], arr[j]] = [arr[j], arr[i]];
        steps.push({
          type: "swap",
          i, j, array: arr.slice(), depth
        });
        swaps++;
        i++;
      }
    }

    [arr[i], arr[pivot]] = [arr[pivot], arr[i]];
    steps.push({
      type: "pivotSwap",
      i, pivot, array: arr.slice(), depth
    });
    swaps++;

    // recurse left and right
    steps.push({ type: "recurse", depth, start, end, mid: i });
    recurse(start, i - 1, depth + 1, { start, end, depth });
    recurse(i + 1, end, depth + 1, { start, end, depth });
  }
  recurse(0, arr.length - 1, 0);
  return steps;
}

const trace = generateQuickSortTrace([...array]);

function drawBox(x, y, value, color = "lightgray") {
  ctx.fillStyle = color;
  ctx.fillRect(x, y, boxSize, boxSize);
  ctx.strokeStyle = "black";
  ctx.strokeRect(x, y, boxSize, boxSize);
  ctx.fillStyle = "black";
  ctx.textAlign = "center";
  ctx.textBaseline = "middle";
  ctx.fillText(value, x + boxSize / 2, y + boxSize / 2);
}
```

```
function drawArrow(x1, y1, x2, y2) {
  ctx.beginPath();
  ctx.moveTo(x1, y1);
  const cpX = (x1 + x2) / 2;
  ctx.bezierCurveTo(cpX, y1 - 40, cpX, y2 + 40, x2, y2);
  ctx.strokeStyle = "blue";
  ctx.lineWidth = 2;
  ctx.stroke();
}

function drawFrame(step) {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  let depthMap = {};

  for (let i = 0; i <= currentFrame; i++) {
    const s = trace[i];
    if (s.snapshot) {
      const arr = s.snapshot;
      const y = baseY + s.depth * levelHeight;
      const offsetX = (canvas.width - arr.length * (boxSize + spacing)) / 2;
      depthMap[`${s.start}-${s.end}-${s.depth}`] = [];

      for (let j = s.start; j <= s.end; j++) {
        const x = j * (boxSize + spacing) + offsetX;
        drawBox(x, y, arr[j], j === s.pivot ? "salmon" : "lightblue");
        depthMap[`${s.start}-${s.end}-${s.depth}`].push({ x, y });

        // track for arrow connection
        if (!positions[s.depth]) positions[s.depth] = {};
        positions[s.depth][j] = { x, y };
      }

      if (s.parentPos) {
        const key = `${s.parentPos.start}-${s.parentPos.end}-${s.parentPos.depth}`;
        const parentBoxes = depthMap[key];
        if (parentBoxes) {
          const childBoxes = depthMap[`${s.start}-${s.end}-${s.depth}`];
          if (childBoxes) {
            drawArrow(
              parentBoxes[Math.floor(parentBoxes.length / 2)].x + boxSize / 2,
              parentBoxes[0].y + boxSize,
              childBoxes[Math.floor(childBoxes.length / 2)].x + boxSize / 2,
              childBoxes[0].y
            );
          }
        }
      }
    }

    if (s.type === "compare") {
      const y = baseY + s.depth * levelHeight;
      const offsetX = (canvas.width - s.array.length * (boxSize + spacing)) / 2;
      for (let j = 0; j < s.array.length; j++) {
        const x = j * (boxSize + spacing) + offsetX;
        let color = "lightgray";
        if (j === s.pivot) color = "salmon";
        else if (j === s.j) color = "orange";
        drawBox(x, y, s.array[j], color);
```

```
      }
    }

    if (s.type === "swap" || s.type === "pivotSwap") {
      const y = baseY + s.depth * levelHeight;
      const offsetX = (canvas.width - s.array.length * (boxSize + spacing)) / 2;
      for (let j = 0; j < s.array.length; j++) {
        const x = j * (boxSize + spacing) + offsetX;
        let color = "lightgray";
        if (j === s.i || j === s.j || j === s.pivot) color = "gold";
        drawBox(x, y, s.array[j], color);
      }
    }
  }

  cmpDisplay.textContent = comparisons;
  swpDisplay.textContent = swaps;

  // Show call stack
  const callStack = [];
  for (let i = 0; i <= currentFrame; i++) {
    const s = trace[i];
    if (s.type === "recurse") {
      callStack.push(`quicksort(${s.start}, ${s.end})`);
    }
  }
  stackDisplay.textContent = callStack.reverse().join("\n");
}

// Init first frame
drawFrame(trace[0]);

document.getElementById("next").addEventListener("click", () => {
  if (currentFrame < trace.length - 1) {
    currentFrame++;
    drawFrame(trace[currentFrame]);
  }
});
</script>

</body>
</html>
```

Chapter 6.

Sorting and Order: Linear-Time Sorting

1. Counting Sort
2. Radix Sort
3. Bucket Sort
4. When These Work in JS Contexts

6 Sorting and Order: Linear-Time Sorting

6.1 Counting Sort

Counting sort is a **non-comparison-based** sorting algorithm designed specifically for sorting integers within a **limited range**. Unlike comparison-based algorithms (like quick sort or merge sort), counting sort does not sort by comparing elements. Instead, it counts the occurrences of each value and uses that count to build the final sorted array.

Because it avoids element-to-element comparisons, counting sort can achieve **linear time complexity**, O(n), under the right conditions. However, it's not universally applicable—it only works well when sorting integers or values that can be mapped to a small set of non-negative integers.

6.1.1 When Is Counting Sort Efficient?

Counting sort shines when:

- The input consists of **non-negative integers**.
- The **range of values (k)** is not significantly larger than the number of elements (n).

For example, sorting 1,000 student test scores ranging from 0–100 is ideal for counting sort.

If the range of numbers is very large (e.g., 0–1,000,000), but you only have 1,000 elements, the space overhead becomes excessive, making counting sort impractical.

6.1.2 How Counting Sort Works

The algorithm follows these steps:

1. **Count the occurrences** of each value in the input array.
2. **Accumulate** counts to determine the positions of elements.
3. **Place** each element in its sorted position using the count data.

Let's sort `[4, 2, 2, 8, 3, 3, 1]`:

1. Count each value (from 0 to 8): `counts = [0, 1, 2, 2, 1, 0, 0, 0, 1]`
2. Accumulate positions (optional for stable version).
3. Use the count array to build the sorted result: `[1, 2, 2, 3, 3, 4, 8]`

6.1.3 JavaScript Implementation

Here's a simple and clear implementation of counting sort in JavaScript:

```javascript
function countingSort(arr, maxValue) {
  // Step 1: Initialize count array with zeros
  const count = new Array(maxValue + 1).fill(0);

  // Step 2: Count occurrences of each value
  for (let i = 0; i < arr.length; i++) {
    count[arr[i]]++;
  }

  // Step 3: Build the sorted array
  const result = [];
  for (let i = 0; i < count.length; i++) {
    while (count[i] > 0) {
      result.push(i);
      count[i]--;
    }
  }

  return result;
}

// Example usage
const data = [4, 2, 2, 8, 3, 3, 1];
console.log(countingSort(data, 8)); // Output: [1, 2, 2, 3, 3, 4, 8]
```

> Note: You must supply `maxValue` or compute it dynamically for this function to work.

6.1.4 Time and Space Complexity

Metric	Complexity
Time	O(n + k)
Space	O(k)
Stable	NO (unless modified)
Comparison-based	NO

- **n** = number of elements
- **k** = range of input values (max - min + 1)

In the best scenarios, when `k` is small and close to `n`, the algorithm runs in linear time—**faster than any comparison-based sort**.

6.1.5 Limitations

- Only works for **integers or discrete values**.
- Not efficient when the value range (`k`) is **much larger than `n`**.
- Standard implementation is **not stable**, but can be modified to preserve order.

Counting sort is **not suitable** for:

- Floating-point numbers,
- Arbitrary objects,
- Large sparse value ranges.

6.1.6 When Counting Sort Outperforms

Let's compare performance:

Dataset	Best Algorithm
`[4, 2, 2, 8, 3, 3, 1]`	Counting Sort (small range)
1,000 integers from 0–50	Counting Sort
1,000,000 integers from 0–1,000,000	Quick Sort (range too large)

When the range of values is small relative to the size of the input, counting sort can outperform even the fastest comparison-based sorts like quick sort and merge sort.

6.1.7 Conclusion

Counting sort is a powerful, specialized algorithm that leverages **frequency counting instead of comparison** to sort data. It achieves **linear time performance** when used under the right conditions—particularly when sorting a large number of small-range integers. While it's not universally applicable, it's an important tool to have when performance matters and the data fits its constraints.

6.2 Radix Sort

Radix sort is a powerful **non-comparison-based** sorting algorithm that sorts numbers by processing individual digits. Instead of comparing whole numbers directly, radix sort breaks them down into digits and sorts the numbers **digit-by-digit**, starting either from the **least significant digit (LSD)** or the **most significant digit (MSD)**.

6.2.1 How Radix Sort Works

The core idea of radix sort is to sort numbers based on their digits, grouping and ordering elements repeatedly by each digit's value until the entire list is sorted. Radix sort relies on a **stable sorting algorithm** (like counting sort) at each digit level to maintain relative order.

There are two main variants:

- **LSD Radix Sort (Least Significant Digit first):** Sorting begins with the rightmost digit and moves leftward. This variant is commonly used for sorting integers and works well with fixed-length numbers or zero-padded values.
- **MSD Radix Sort (Most Significant Digit first):** Sorting begins with the leftmost digit and proceeds rightward. It's more suited for variable-length keys, such as strings or numbers with different digit counts.

6.2.2 LSD Radix Sort Step-by-Step

Suppose we want to sort `[170, 45, 75, 90, 802, 24, 2, 66]`:

1. Sort by the least significant digit (units place): `[170, 90, 802, 2, 24, 45, 75, 66]`
2. Sort by the next digit (tens place): `[802, 2, 24, 45, 66, 170, 75, 90]`
3. Sort by the most significant digit (hundreds place): `[2, 24, 45, 66, 75, 90, 170, 802]`

After processing all digits, the array is fully sorted.

6.2.3 JavaScript Implementation (LSD Radix Sort)

Full runnable code:

```
function getDigit(num, place) {
  return Math.floor(Math.abs(num) / Math.pow(10, place)) % 10;
}

function digitCount(num) {
  if (num === 0) return 1;
  return Math.floor(Math.log10(Math.abs(num))) + 1;
}

function mostDigits(nums) {
  let maxDigits = 0;
  for (let num of nums) {
    maxDigits = Math.max(maxDigits, digitCount(num));
  }
```

```
  return maxDigits;
}

function radixSort(arr) {
  const maxDigitCount = mostDigits(arr);

  for (let k = 0; k < maxDigitCount; k++) {
    // Create buckets for each digit (0 to 9)
    let buckets = Array.from({ length: 10 }, () => []);

    for (let num of arr) {
      const digit = getDigit(num, k);
      buckets[digit].push(num);
    }

    // Flatten buckets back into array
    arr = [].concat(...buckets);
  }
  return arr;
}

// Example usage
const nums = [170, 45, 75, 90, 802, 24, 2, 66];
console.log(radixSort(nums)); // Output: [2, 24, 45, 66, 75, 90, 170, 802]
```

6.2.4 Complexity and Practical Considerations

- **Time complexity:** O(d × (n + k)) Where `d` = number of digits, `n` = number of elements, and `k` = digit range (usually 10 for base-10).
- Radix sort can outperform comparison-based sorts like quick sort when:
  - Numbers have a **fixed or small number of digits**.
  - The digit range (`k`) is small.
- **Space complexity:** Requires extra space for buckets, generally O(n + k).
- Radix sort is **stable** because it uses a stable sort (like counting sort) at each digit level.

6.2.5 Handling Strings or Fixed-Length Keys

Radix sort can easily extend to sort **strings** or fixed-length keys by processing characters from either left to right (MSD) or right to left (LSD). This is especially useful in applications like sorting words, IP addresses, or dates.

6.2.6 When to Use Radix Sort in JavaScript

- When sorting large arrays of **fixed-length integers** efficiently.
- Sorting strings of uniform length, like fixed-length identifiers.
- When guaranteed linear-time sorting outperforms comparison-based sorts.

6.2.7 Summary

Radix sort is a versatile and efficient sorting algorithm that processes elements digit-by-digit. Its stable, linear-time behavior makes it a valuable tool in specialized scenarios, especially with integer keys of limited digit length or fixed-length strings. Understanding how digit extraction and stable bucket sorting work will help you implement radix sort effectively in JavaScript and leverage its speed advantages where applicable.

6.3 Bucket Sort

Bucket Sort is a *distribution-based* sorting algorithm particularly effective for sorting numbers that are **uniformly distributed** over a range. Unlike comparison-based sorts like quicksort or mergesort, bucket sort leverages the distribution of the input values to achieve **linear-time performance** in the average case.

6.3.1 How Bucket Sort Works

The core idea of bucket sort is to divide the input array into a number of "buckets," sort the contents of each bucket individually (often using another sorting algorithm like insertion sort), and then concatenate the sorted buckets to form the final sorted array.

Here's the step-by-step breakdown:

1. **Create Buckets**: Based on the range of input values, create an array of empty buckets.
2. **Distribute Input Into Buckets**: Place each element in the array into one of the buckets. This distribution typically uses a simple formula to determine the right bucket for a value.
3. **Sort Each Bucket**: Sort the contents of each bucket. If the number of elements in each bucket is small, a simple algorithm like insertion sort is very efficient.
4. **Concatenate Buckets**: Combine the buckets in order to get the final sorted output.

6.3.2 JavaScript Example

Below is a basic implementation of bucket sort in JavaScript, suitable for sorting an array of floating-point numbers in the range [0, 1).

```javascript
function bucketSort(arr, bucketCount = 10) {
  if (arr.length === 0) return [];

  // Step 1: Create buckets
  const buckets = Array.from({ length: bucketCount }, () => []);

  // Step 2: Distribute values into buckets
  for (const value of arr) {
    const index = Math.floor(value * bucketCount);
    buckets[index].push(value);
  }

  // Step 3: Sort individual buckets
  for (let i = 0; i < bucketCount; i++) {
    buckets[i].sort((a, b) => a - b); // Using built-in sort
  }

  // Step 4: Concatenate all buckets
  return buckets.flat();
}

// Example usage:
const input = [0.78, 0.17, 0.39, 0.26, 0.94, 0.21, 0.12, 0.23, 0.68];
console.log(bucketSort(input));
```

6.3.3 Visualizing Bucket Sort

Suppose you have the input array:

```
[0.78, 0.17, 0.39, 0.26, 0.94, 0.21]
```

You could divide this into 5 buckets:

```
Bucket 0: [0.17]
Bucket 1: [0.21, 0.26]
Bucket 2: [0.39]
Bucket 3: [0.78]
Bucket 4: [0.94]
```

Each bucket is sorted:

```
Bucket 0: [0.17]
Bucket 1: [0.21, 0.26]
...
```

Concatenated result:

```
[0.17, 0.21, 0.26, 0.39, 0.78, 0.94]
```

6.3.4 When to Use Bucket Sort

Bucket sort performs exceptionally well when:

- The data is **uniformly distributed** across a known range.
- You're dealing with **floating-point values** between 0 and 1, or normalized datasets.
- You want to achieve **linear average-case performance**, especially when other sorts may degrade.

However, if the input is *skewed* or unevenly distributed, some buckets may end up with many elements while others are nearly empty, resulting in a performance closer to O(n log n) or worse.

6.3.5 Performance

- **Best/Average Case**: O(n + k), where n is the number of elements and k is the number of buckets.
- **Worst Case**: $O(n^2)$, when all elements fall into a single bucket (and that bucket is sorted using a poor algorithm).
- **Space Complexity**: O(n + k)

In JavaScript, bucket sort is most beneficial when working with numeric datasets in analytics, simulations, or normalized measurements. Its simplicity, especially when combined with built-in sorts, makes it practical and efficient for these scenarios.

6.4 When These Work in JS Contexts

Linear-time sorting algorithms like **Counting Sort**, **Radix Sort**, and **Bucket Sort** are powerful tools, but their effectiveness in JavaScript depends heavily on the context in which they're used. Unlike general-purpose comparison-based sorts like quicksort or mergesort (used under the hood by JavaScript's native `.sort()`), linear-time sorts are specialized—they can outperform `.sort()` but only under certain conditions.

6.4.1 Constraints in JavaScript Environments

JavaScript is a high-level, dynamic language with some built-in flexibility—but also some limitations that matter when implementing linear-time sorting:

Input Data Types

- Linear-time sorts assume numeric input, and often **non-negative integers** or **normalized floats**.
- JavaScript arrays can contain mixed types (e.g., strings, numbers, `undefined`), but linear sorts **do not handle mixed types well**.
- If your data is not already numeric, you may need a pre-processing step to transform it into a usable format.

Value Ranges

- Counting sort requires that you know the **maximum value** in advance and that the range is not too large.
- Radix sort is ideal for **fixed-length integers** or strings with known character sets.
- Bucket sort works best when data is **uniformly distributed** across a known range.

When these assumptions break down—such as sorting large sparse integers or skewed float distributions—performance and memory efficiency can degrade significantly.

Memory Usage

- These algorithms typically allocate additional arrays or objects (e.g., buckets, count arrays).
- In large-scale applications (e.g., sorting millions of items), this can result in high memory usage or even slowdowns due to JavaScript's garbage collection.
- Counting sort is especially memory-sensitive when used on data with a large range.

6.4.2 Comparing to `.sort()`

JavaScript's native `.sort()` method is well-optimized and works well in general-purpose cases:

```
[5, 3, 8, 1].sort((a, b) => a - b);
```

Internally, modern engines like V8 (used by Chrome and Node.js) implement hybrid algorithms—usually **Timsort**, a variant of merge and insertion sort that handles many real-world cases efficiently.

So why use a linear-time sort?

- **Speed**: If the input is large but falls into a constrained numeric range, linear-time sorts can be **much faster**.
- **Control**: In environments like competitive programming or performance-critical back-

end systems, you may need tighter control over the sorting mechanism.

- **Stability**: Counting sort and radix sort are naturally stable (preserve input order of equal elements), which `.sort()` doesn't guarantee unless implemented explicitly.

6.4.3 Example: Benchmark Comparison

```
const data = Array.from({ length: 1e6 }, () => Math.floor(Math.random() * 1000)); // small range

console.time('Native sort');
data.slice().sort((a, b) => a - b);
console.timeEnd('Native sort');

console.time('Counting sort');
countingSort(data); // assume implementation from earlier section
console.timeEnd('Counting sort');
```

With a small range (0–999), counting sort often outperforms native `.sort()` by a large margin.

6.4.4 Best Practices

- Use **Counting Sort** when data is dense in a small integer range.
- Use **Radix Sort** for large arrays of integers or fixed-length strings.
- Use **Bucket Sort** for normalized floats or uniformly distributed datasets.
- Use **`.sort()`** for general-purpose, mixed-type, or unknown-range data.

In summary, linear-time sorting can provide *significant speedups* in the right context. But in JavaScript, it's essential to weigh the **performance gains** against **code complexity**, **memory overhead**, and **data preparation costs**.

Chapter 7.

Arrays and Linked Lists

1. Native Arrays and Their Limits
2. Implementing Singly/Doubly Linked Lists

7 Arrays and Linked Lists

7.1 Native Arrays and Their Limits

JavaScript's native `Array` is one of the most flexible and widely used data structures in the language. Unlike fixed-size arrays in lower-level languages like C or Java, JavaScript arrays are *dynamic*, allowing elements to be added, removed, or modified at any index. While this makes them extremely convenient for general-purpose programming, it also hides performance trade-offs that are important to understand when solving algorithmic problems or working with large datasets.

Run the following code in browser to see the demo:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>Interactive Array Visualization</title>
<style>
  body {
    font-family: Arial, sans-serif;
    text-align: center;
  }
  canvas {
    border: 1px solid black;
    background: #fafafa;
    display: block;
    margin: 20px auto;
  }
  #controls {
    margin: 10px auto;
    max-width: 700px;
  }
  input, button {
    font-size: 16px;
    padding: 6px 8px;
    margin: 4px;
  }
  #message {
    height: 24px;
    margin-top: 8px;
    font-weight: bold;
  }
</style>
</head>
<body>

<h2>Interactive Array Visualization</h2>

<div id="controls">
  <input type="number" id="indexInput" placeholder="Index" min="0" />
  <input type="number" id="valueInput" placeholder="Value" />
  <button id="accessBtn">Access</button>
  <button id="updateBtn">Update</button>
  <button id="insertBtn">Insert</button>
```

```
  <button id="deleteBtn">Delete</button>
</div>

<canvas id="canvas" width="700" height="150"></canvas>
<div id="message"></div>

<script>
  const canvas = document.getElementById('canvas');
  const ctx = canvas.getContext('2d');
  const indexInput = document.getElementById('indexInput');
  const valueInput = document.getElementById('valueInput');
  const accessBtn = document.getElementById('accessBtn');
  const updateBtn = document.getElementById('updateBtn');
  const insertBtn = document.getElementById('insertBtn');
  const deleteBtn = document.getElementById('deleteBtn');
  const message = document.getElementById('message');

  let array = [10, 22, 35, 47, 59, 63, 78];
  const cellWidth = 80;
  const cellHeight = 60;
  const startY = 50;
  let scrollX = 0;
  const scrollStep = 40;

  // For animations
  let highlightIndex = -1;
  let animationFrames = [];

  // Calculate total width needed for array
  function totalWidth() {
    return array.length * cellWidth;
  }

  // Draw the array with scrolling offset and highlight
  function drawArray() {
    ctx.clearRect(0, 0, canvas.width, canvas.height);
    ctx.font = "18px Arial";
    ctx.textAlign = "center";
    ctx.textBaseline = "middle";

    const visibleCount = Math.floor(canvas.width / cellWidth);

    for (let i = 0; i < array.length; i++) {
      const x = i * cellWidth - scrollX;
      if (x + cellWidth < 0 || x > canvas.width) continue; // skip if out of visible area

      // Highlight box if needed
      if (i === highlightIndex) {
        ctx.fillStyle = 'orange';
        ctx.fillRect(x, startY, cellWidth, cellHeight);
      }

      // Draw box border
      ctx.strokeStyle = "black";
      ctx.lineWidth = 2;
      ctx.strokeRect(x, startY, cellWidth, cellHeight);

      // Draw value
```

```
    ctx.fillStyle = i === highlightIndex ? "black" : "black";
    ctx.fillText(array[i], x + cellWidth / 2, startY + cellHeight / 2);

    // Draw index below
    ctx.fillStyle = "gray";
    ctx.font = "14px Arial";
    ctx.fillText(i, x + cellWidth / 2, startY + cellHeight + 20);

    ctx.font = "18px Arial";
  }
}

// Animate highlight for access/update
function animateHighlight(index, duration = 1000) {
  highlightIndex = index;
  drawArray();
  setTimeout(() => {
    highlightIndex = -1;
    drawArray();
  }, duration);
}

// Animate insertion with sliding boxes to right
function animateInsert(index, value) {
  // Insert a placeholder at the end
  array.push(value);
  let frame = 0;
  const totalFrames = 20;
  const startScroll = scrollX;

  function animate() {
    frame++;
    const progress = frame / totalFrames;

    // Scroll if needed to show inserted item
    const targetScroll = Math.max(0, (array.length - Math.floor(canvas.width / cellWidth)) * cellWidt
    scrollX = startScroll + (targetScroll - startScroll) * progress;

    // Shift elements right starting from the inserted index
    // We'll redraw array offsetting the elements >= index by progress * cellWidth
    ctx.clearRect(0, 0, canvas.width, canvas.height);
    ctx.font = "18px Arial";
    ctx.textAlign = "center";
    ctx.textBaseline = "middle";

    for (let i = 0; i < array.length; i++) {
      let x = i * cellWidth - scrollX;
      if (x + cellWidth < 0 || x > canvas.width) continue;

      if (i === index) {
        // The new inserted box slides in from left (or opacity fade)
        const insertX = x - cellWidth * (1 - progress);
        ctx.fillStyle = 'lightgreen';
        ctx.fillRect(insertX, startY, cellWidth, cellHeight);
        ctx.strokeStyle = "black";
        ctx.lineWidth = 2;
        ctx.strokeRect(insertX, startY, cellWidth, cellHeight);
        ctx.fillStyle = "black";
```

```
        ctx.fillText(array[i], insertX + cellWidth / 2, startY + cellHeight / 2);
        ctx.fillStyle = "gray";
        ctx.font = "14px Arial";
        ctx.fillText(i, insertX + cellWidth / 2, startY + cellHeight + 20);
        ctx.font = "18px Arial";
      } else if (i > index) {
        // Shift right by progress * cellWidth
        const shiftedX = x + cellWidth * progress;
        ctx.fillStyle = 'white';
        ctx.fillRect(shiftedX, startY, cellWidth, cellHeight);
        ctx.strokeStyle = "black";
        ctx.lineWidth = 2;
        ctx.strokeRect(shiftedX, startY, cellWidth, cellHeight);
        ctx.fillStyle = "black";
        ctx.fillText(array[i], shiftedX + cellWidth / 2, startY + cellHeight / 2);
        ctx.fillStyle = "gray";
        ctx.font = "14px Arial";
        ctx.fillText(i, shiftedX + cellWidth / 2, startY + cellHeight + 20);
        ctx.font = "18px Arial";
      } else {
        // Normal boxes
        ctx.strokeStyle = "black";
        ctx.lineWidth = 2;
        ctx.strokeRect(x, startY, cellWidth, cellHeight);
        ctx.fillStyle = "black";
        ctx.fillText(array[i], x + cellWidth / 2, startY + cellHeight / 2);
        ctx.fillStyle = "gray";
        ctx.font = "14px Arial";
        ctx.fillText(i, x + cellWidth / 2, startY + cellHeight + 20);
        ctx.font = "18px Arial";
      }
    }

    if (frame < totalFrames) {
      requestAnimationFrame(animate);
    } else {
      // Insert is done, no highlight
      drawArray();
    }
  }

  animate();
}

// Animate delete by sliding boxes left
function animateDelete(index) {
  let frame = 0;
  const totalFrames = 20;
  const startScroll = scrollX;

  function animate() {
    frame++;
    const progress = frame / totalFrames;

    // Scroll if needed to keep array visible
    const maxScroll = Math.max(0, (array.length - 1 - Math.floor(canvas.width / cellWidth)) * cellWid
    scrollX = startScroll + (maxScroll - startScroll) * progress;
```

```
    ctx.clearRect(0, 0, canvas.width, canvas.height);
    ctx.font = "18px Arial";
    ctx.textAlign = "center";
    ctx.textBaseline = "middle";

    for (let i = 0; i < array.length; i++) {
      if (i === index) continue; // this one is deleting, so skip drawing

      let x = i * cellWidth - scrollX;
      if (x + cellWidth < 0 || x > canvas.width) continue;

      if (i > index) {
        // Shift left by progress * cellWidth
        const shiftedX = x - cellWidth * progress;
        ctx.strokeStyle = "black";
        ctx.lineWidth = 2;
        ctx.strokeRect(shiftedX, startY, cellWidth, cellHeight);
        ctx.fillStyle = "black";
        ctx.fillText(array[i], shiftedX + cellWidth / 2, startY + cellHeight / 2);
        ctx.fillStyle = "gray";
        ctx.font = "14px Arial";
        ctx.fillText(i - 1, shiftedX + cellWidth / 2, startY + cellHeight + 20);
        ctx.font = "18px Arial";
      } else {
        // Normal boxes
        ctx.strokeStyle = "black";
        ctx.lineWidth = 2;
        ctx.strokeRect(x, startY, cellWidth, cellHeight);
        ctx.fillStyle = "black";
        ctx.fillText(array[i], x + cellWidth / 2, startY + cellHeight / 2);
        ctx.fillStyle = "gray";
        ctx.font = "14px Arial";
        ctx.fillText(i, x + cellWidth / 2, startY + cellHeight + 20);
        ctx.font = "18px Arial";
      }
    }

    if (frame < totalFrames) {
      requestAnimationFrame(animate);
    } else {
      // Actually remove the element now
      array.splice(index, 1);
      drawArray();
    }
  }

  animate();
}

// Animate update by flashing the box
function animateUpdate(index, newValue) {
  let frame = 0;
  const totalFrames = 30;

  function animate() {
    frame++;
    ctx.clearRect(0, 0, canvas.width, canvas.height);
    ctx.font = "18px Arial";
```

```
    ctx.textAlign = "center";
    ctx.textBaseline = "middle";

    for (let i = 0; i < array.length; i++) {
      const x = i * cellWidth - scrollX;
      if (x + cellWidth < 0 || x > canvas.width) continue;

      if (i === index) {
        // Flashing effect by changing fillStyle
        const intensity = Math.abs(Math.sin(frame / 5));
        ctx.fillStyle = `rgba(255, 165, 0, ${intensity})`; // orange glow
        ctx.fillRect(x, startY, cellWidth, cellHeight);
      }
      ctx.strokeStyle = "black";
      ctx.lineWidth = 2;
      ctx.strokeRect(x, startY, cellWidth, cellHeight);

      ctx.fillStyle = "black";
      ctx.fillText(i === index ? newValue : array[i], x + cellWidth / 2, startY + cellHeight / 2);
      ctx.fillStyle = "gray";
      ctx.font = "14px Arial";
      ctx.fillText(i, x + cellWidth / 2, startY + cellHeight + 20);
      ctx.font = "18px Arial";
    }

    if (frame < totalFrames) {
      requestAnimationFrame(animate);
    } else {
      // Finalize update
      array[index] = newValue;
      drawArray();
    }
  }

  animate();
}

// Scroll canvas horizontally (left/right buttons could be added)
function scrollLeft() {
  scrollX = Math.max(0, scrollX - scrollStep);
  drawArray();
}
function scrollRight() {
  const maxScroll = Math.max(0, totalWidth() - canvas.width);
  scrollX = Math.min(maxScroll, scrollX + scrollStep);
  drawArray();
}

// Button event handlers
accessBtn.onclick = () => {
  const idx = parseInt(indexInput.value);
  if (isNaN(idx) || idx < 0 || idx >= array.length) {
    message.textContent = "Invalid index for access";
    return;
  }
  message.textContent = `Accessed element at index ${idx}: ${array[idx]}`;
  animateHighlight(idx);
};
```

```
updateBtn.onclick = () => {
  const idx = parseInt(indexInput.value);
  const val = parseInt(valueInput.value);
  if (isNaN(idx) || idx < 0 || idx >= array.length) {
    message.textContent = "Invalid index for update";
    return;
  }
  if (isNaN(val)) {
    message.textContent = "Invalid value for update";
    return;
  }
  message.textContent = `Updating element at index ${idx} to ${val}`;
  animateUpdate(idx, val);
};

insertBtn.onclick = () => {
  const idx = parseInt(indexInput.value);
  const val = parseInt(valueInput.value);
  if (isNaN(idx) || idx < 0 || idx > array.length) {
    message.textContent = "Invalid index for insert";
    return;
  }
  if (isNaN(val)) {
    message.textContent = "Invalid value for insert";
    return;
  }
  message.textContent = `Inserting ${val} at index ${idx}`;
  // Insert visually with animation
  // Actually splice after animation completes
  // We insert a temporary placeholder now and slide later
  array.splice(idx, 0, val); // Insert to array but animate sliding boxes

  // Animate sliding boxes to the right, then finalize
  let frame = 0;
  const totalFrames = 20;
  const startScroll = scrollX;

  function animate() {
    frame++;
    const progress = frame / totalFrames;

    // Scroll if needed to show inserted item
    const targetScroll = Math.max(0, (array.length - Math.floor(canvas.width / cellWidth)) * cellWidt
    scrollX = startScroll + (targetScroll - startScroll) * progress;

    ctx.clearRect(0, 0, canvas.width, canvas.height);
    ctx.font = "18px Arial";
    ctx.textAlign = "center";
    ctx.textBaseline = "middle";

    for (let i = 0; i < array.length; i++) {
      let x = i * cellWidth - scrollX;
      if (x + cellWidth < 0 || x > canvas.width) continue;

      if (i === idx) {
        const insertX = x - cellWidth * (1 - progress);
        ctx.fillStyle = 'lightgreen';
        ctx.fillRect(insertX, startY, cellWidth, cellHeight);
```

```
        ctx.strokeStyle = "black";
        ctx.lineWidth = 2;
        ctx.strokeRect(insertX, startY, cellWidth, cellHeight);
        ctx.fillStyle = "black";
        ctx.fillText(array[i], insertX + cellWidth / 2, startY + cellHeight / 2);
        ctx.fillStyle = "gray";
        ctx.font = "14px Arial";
        ctx.fillText(i, insertX + cellWidth / 2, startY + cellHeight + 20);
        ctx.font = "18px Arial";
      } else if (i > idx) {
        const shiftedX = x + cellWidth * progress;
        ctx.strokeStyle = "black";
        ctx.lineWidth = 2;
        ctx.strokeRect(shiftedX, startY, cellWidth, cellHeight);
        ctx.fillStyle = "black";
        ctx.fillText(array[i], shiftedX + cellWidth / 2, startY + cellHeight / 2);
        ctx.fillStyle = "gray";
        ctx.font = "14px Arial";
        ctx.fillText(i, shiftedX + cellWidth / 2, startY + cellHeight + 20);
        ctx.font = "18px Arial";
      } else {
        ctx.strokeStyle = "black";
        ctx.lineWidth = 2;
        ctx.strokeRect(x, startY, cellWidth, cellHeight);
        ctx.fillStyle = "black";
        ctx.fillText(array[i], x + cellWidth / 2, startY + cellHeight / 2);
        ctx.fillStyle = "gray";
        ctx.font = "14px Arial";
        ctx.fillText(i, x + cellWidth / 2, startY + cellHeight + 20);
        ctx.font = "18px Arial";
      }
    }

    if (frame < totalFrames) {
      requestAnimationFrame(animate);
    } else {
      drawArray();
    }
  }

  animate();
};

deleteBtn.onclick = () => {
  const idx = parseInt(indexInput.value);
  if (isNaN(idx) || idx < 0 || idx >= array.length) {
    message.textContent = "Invalid index for delete";
    return;
  }
  message.textContent = `Deleting element at index ${idx}`;
  animateDelete(idx);
};

// Scroll on mouse wheel horizontally
canvas.addEventListener('wheel', (e) => {
  e.preventDefault();
  if (e.deltaY < 0) {
    scrollLeft();
```

```
    } else {
      scrollRight();
    }
  });

  drawArray();
</script>
</body>
</html>
```

7.1.1 Key Features of Native Arrays

Dynamic Sizing

JavaScript arrays grow and shrink as needed. You can push, pop, or even assign a value to a distant index, and the array adjusts automatically.

```
const arr = [];
arr[100] = 'hello';
console.log(arr.length); // 101
```

Assigning a value to index 100 creates a sparse array with undefined values in the gaps (indices 0 through 99).

Sparse Arrays

JavaScript arrays can have missing indices. These are not the same as `undefined` values—they are actual *holes* in the array.

```
const sparse = [1];
sparse[5] = 10;
console.log(sparse);        // [1, <4 empty items>, 10]
console.log(sparse.length); // 6
```

Sparse arrays can negatively impact performance because they prevent the JavaScript engine from optimizing the array as a dense, contiguous block in memory.

Flexible Indexing and Types

Unlike typed arrays in other languages, JavaScript arrays can contain mixed types:

```
const mix = [42, "text", { key: "value" }, [1, 2], null];
```

While this is powerful, it adds overhead and weakens guarantees around performance and structure.

7.1.2 Performance Implications

Access Time

Accessing elements by index (e.g. `arr[i]`) is generally **O(1)**—constant time—even for sparse arrays.

Insertion and Deletion

- **Push/Pop**: Adding or removing from the end (`push`, `pop`) is typically **O(1)**.
- **Unshift/Shift**: Adding or removing from the beginning (`unshift`, `shift`) is **O(n)** because all elements must be reindexed.
- **Splice/Delete**: Deleting an element with `splice()` causes shifting, resulting in **O(n)** operations.

```
const a = [1, 2, 3, 4, 5];
a.splice(2, 1); // Removes value at index 2
console.log(a); // [1, 2, 4, 5]
```

This shifts all elements after index 2 left by one, which is costly for large arrays.

Hidden Costs

Under the hood, JavaScript engines like V8 optimize arrays by categorizing them as *packed* (dense) or *holey* (sparse). Dense arrays are faster to process, but sparse or mixed-type arrays lose these optimizations, causing slower reads, writes, and iterations.

7.1.3 When Native Arrays Become Inefficient

Native arrays can become inefficient in scenarios such as:

- **Frequent Insertions/Deletions at the Start or Middle**: Constant reindexing makes this costly.
- **Sparse Data Representation**: Sparse arrays increase memory usage and slow down iteration.
- **Large Data Sets Requiring Consistent Performance**: With millions of elements, small inefficiencies compound.

7.1.4 Example: Performance Difference

```
// Inefficient for large arrays
const big = Array(1e6).fill(0);
console.time('shift');
big.shift(); // O(n) operation
console.timeEnd('shift');
```

Even though it's just removing the first element, the time complexity is linear due to reindexing.

7.1.5 Conclusion

JavaScript's native arrays are powerful, flexible, and suitable for many everyday tasks. However, understanding their underlying mechanics and limitations is crucial for writing performant code. When you find yourself needing frequent insertions or deletions in the middle of a dataset—or when working with very large or sparse data—consider switching to a more appropriate structure like a linked list or a custom data structure tailored to your use case.

7.2 Implementing Singly/Doubly Linked Lists

Linked lists are fundamental data structures that store elements in a *sequence of nodes*, where each node contains data and one or more references (or "pointers") to other nodes. Unlike arrays, linked lists do **not require contiguous memory** and allow for efficient insertions and deletions—especially in the middle or beginning of a list.

7.2.1 Concept: Linked Lists vs Arrays

Feature	Array	Linked List
Memory layout	Contiguous	Non-contiguous (nodes linked)
Access by index	O(1)	O(n)
Insertion (start)	O(n)	O(1) (linked list)
Deletion (middle)	O(n)	O(1) with reference

There are two common types:

- **Singly Linked List**: Each node points only to the *next* node.
- **Doubly Linked List**: Each node points to *both* the next and previous nodes.

7.2.2 Singly Linked List

Structure

```
[HEAD] → [data|next] → [data|next] → null
```

Node Class

```
class ListNode {
  constructor(value) {
    this.value = value;
    this.next = null;
  }
}
```

SinglyLinkedList Class

```
class SinglyLinkedList {
  constructor() {
    this.head = null;
  }

  insertAtHead(value) {
    const newNode = new ListNode(value);
    newNode.next = this.head;
    this.head = newNode;
  }

  insertAtEnd(value) {
    const newNode = new ListNode(value);
    if (!this.head) {
      this.head = newNode;
      return;
    }
    let current = this.head;
    while (current.next) {
      current = current.next;
    }
    current.next = newNode;
  }

  delete(value) {
    if (!this.head) return;

    if (this.head.value === value) {
      this.head = this.head.next;
      return;
    }

    let current = this.head;
    while (current.next && current.next.value !== value) {
      current = current.next;
    }

    if (current.next) {
      current.next = current.next.next;
    }
```

```
  }

  search(value) {
    let current = this.head;
    while (current) {
      if (current.value === value) return true;
      current = current.next;
    }
    return false;
  }

  traverse() {
    let current = this.head;
    const values = [];
    while (current) {
      values.push(current.value);
      current = current.next;
    }
    return values;
  }
}
```

Run the following code in browser to see the demo:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>Singly Linked List Visualization</title>
<style>
  body {
    font-family: Arial, sans-serif;
    text-align: center;
  }
  canvas {
    border: 1px solid black;
    background: #fafafa;
    display: block;
    margin: 20px auto;
  }
  #controls {
    margin: 10px auto;
    max-width: 700px;
  }
  input, button {
    font-size: 16px;
    padding: 6px 8px;
    margin: 4px;
  }
  #message {
    height: 24px;
    margin-top: 8px;
    font-weight: bold;
  }
</style>
</head>
<body>
```

```
<h2>Singly Linked List Visualization</h2>

<div id="controls">
  <input type="number" id="valueInput" placeholder="Value" />
  <input type="number" id="posInput" placeholder="Position (0-based)" />
  <button id="insertBtn">Insert</button>
  <button id="deleteBtn">Delete</button>
  <button id="highlightBtn">Highlight Node</button>
</div>

<canvas id="canvas" width="900" height="200"></canvas>
<div id="message"></div>

<script>
  const canvas = document.getElementById('canvas');
  const ctx = canvas.getContext('2d');

  const valueInput = document.getElementById('valueInput');
  const posInput = document.getElementById('posInput');
  const insertBtn = document.getElementById('insertBtn');
  const deleteBtn = document.getElementById('deleteBtn');
  const highlightBtn = document.getElementById('highlightBtn');
  const message = document.getElementById('message');

  const nodeWidth = 70;
  const nodeHeight = 40;
  const nodeSpacing = 50;
  const startX = 50;
  const startY = 100;

  // Linked list nodes stored as objects {value, next}
  let head = null;

  // Highlight node index (-1 = none)
  let highlightIndex = -1;

  // Build linked list from array for easy initialization
  function buildListFromArray(arr) {
    head = null;
    let prev = null;
    for (let val of arr) {
      const node = { value: val, next: null };
      if (!head) head = node;
      else prev.next = node;
      prev = node;
    }
  }

  // Convert linked list to array for easy iteration
  function listToArray() {
    const arr = [];
    let current = head;
    while (current) {
      arr.push(current);
      current = current.next;
    }
    return arr;
  }
```

```
// Draw a single node box at (x,y)
function drawNode(x, y, node, isHighlighted = false) {
  ctx.fillStyle = isHighlighted ? '#f9d342' : 'lightblue';
  ctx.strokeStyle = 'black';
  ctx.lineWidth = 2;
  ctx.fillRect(x, y, nodeWidth, nodeHeight);
  ctx.strokeRect(x, y, nodeWidth, nodeHeight);

  ctx.fillStyle = 'black';
  ctx.font = '18px Arial';
  ctx.textAlign = 'center';
  ctx.textBaseline = 'middle';
  ctx.fillText(node.value, x + nodeWidth / 2, y + nodeHeight / 2);

  // Draw "next" box on right side of node
  const nextX = x + nodeWidth;
  const nextY = y;
  ctx.fillStyle = 'white';
  ctx.fillRect(nextX, nextY, 20, nodeHeight);
  ctx.strokeRect(nextX, nextY, 20, nodeHeight);

  // Draw arrow from next box to next node (if any)
  if (node.next) {
    const arrowStartX = nextX + 20;
    const arrowStartY = nextY + nodeHeight / 2;
    const arrowEndX = nextX + nodeSpacing;
    const arrowEndY = arrowStartY;

    // Line
    ctx.beginPath();
    ctx.moveTo(arrowStartX, arrowStartY);
    ctx.lineTo(arrowEndX, arrowEndY);
    ctx.stroke();

    // Arrowhead
    ctx.beginPath();
    ctx.moveTo(arrowEndX - 10, arrowEndY - 7);
    ctx.lineTo(arrowEndX, arrowEndY);
    ctx.lineTo(arrowEndX - 10, arrowEndY + 7);
    ctx.fillStyle = 'black';
    ctx.fill();
  }
}

// Draw head pointer label
function drawHeadPointer(x, y) {
  ctx.fillStyle = 'black';
  ctx.font = '16px Arial';
  ctx.textAlign = 'center';
  ctx.fillText('head', x + nodeWidth / 2, y - 20);

  // Arrow down
  ctx.beginPath();
  ctx.moveTo(x + nodeWidth / 2, y - 10);
  ctx.lineTo(x + nodeWidth / 2, y);
  ctx.stroke();
}
```

```
// Draw the whole linked list
function drawList() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  const nodes = listToArray();
  if (nodes.length === 0) {
    ctx.fillStyle = 'black';
    ctx.font = '20px Arial';
    ctx.fillText('List is empty', canvas.width / 2, canvas.height / 2);
    return;
  }

  // Draw nodes horizontally spaced
  nodes.forEach((node, idx) => {
    const x = startX + idx * (nodeWidth + nodeSpacing);
    drawNode(x, startY, node, idx === highlightIndex);
    if (idx === 0) drawHeadPointer(x, startY);
  });
}

// Insert node at position with animation
function insertNode(value, position) {
  if (position < 0) {
    message.textContent = "Invalid position";
    return;
  }
  const newNode = { value, next: null };

  if (!head || position === 0) {
    // Insert at head
    newNode.next = head;
    head = newNode;
    highlightIndex = 0;
    drawList();
    message.textContent = `Inserted ${value} at position 0`;
    return;
  }

  let prev = head;
  let idx = 0;
  while (prev && idx < position - 1) {
    prev = prev.next;
    idx++;
  }
  if (!prev) {
    message.textContent = "Position out of bounds";
    return;
  }
  newNode.next = prev.next;
  prev.next = newNode;

  highlightIndex = position;
  drawList();
  message.textContent = `Inserted ${value} at position ${position}`;
}

// Delete node at position with animation
function deleteNode(position) {
```

```
  if (!head || position < 0) {
    message.textContent = "Invalid position";
    return;
  }
  if (position === 0) {
    const deletedVal = head.value;
    head = head.next;
    highlightIndex = -1;
    drawList();
    message.textContent = `Deleted node at position 0 (value ${deletedVal})`;
    return;
  }

  let prev = head;
  let idx = 0;
  while (prev.next && idx < position - 1) {
    prev = prev.next;
    idx++;
  }
  if (!prev.next) {
    message.textContent = "Position out of bounds";
    return;
  }
  const deletedVal = prev.next.value;
  prev.next = prev.next.next;
  highlightIndex = -1;
  drawList();
  message.textContent = `Deleted node at position ${position} (value ${deletedVal})`;
}

// Highlight node at position
function highlightNode(position) {
  const nodes = listToArray();
  if (position < 0 || position >= nodes.length) {
    message.textContent = "Invalid position to highlight";
    return;
  }
  highlightIndex = position;
  drawList();
  message.textContent = `Highlighted node at position ${position} (value ${nodes[position].value})`;
}

insertBtn.onclick = () => {
  const val = parseInt(valueInput.value);
  const pos = parseInt(posInput.value);
  if (isNaN(val)) {
    message.textContent = "Enter a valid value to insert";
    return;
  }
  if (isNaN(pos) || pos < 0) {
    message.textContent = "Enter a valid position to insert";
    return;
  }
  insertNode(val, pos);
};

deleteBtn.onclick = () => {
  const pos = parseInt(posInput.value);
```

```
    if (isNaN(pos) || pos < 0) {
      message.textContent = "Enter a valid position to delete";
      return;
    }
    deleteNode(pos);
  };

  highlightBtn.onclick = () => {
    const pos = parseInt(posInput.value);
    if (isNaN(pos) || pos < 0) {
      message.textContent = "Enter a valid position to highlight";
      return;
    }
    highlightNode(pos);
  };

  // Initialize list with some values
  buildListFromArray([10, 20, 30, 40, 50]);
  drawList();
</script>

</body>
</html>
```

7.2.3 Doubly Linked List

Structure

`null <- [prev|data|next]  [prev|data|next]  [prev|data|next] → null`

Run the following code in browser to see the demo:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Doubly Linked List - Canvas</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      text-align: center;
      padding: 20px;
    }
    canvas {
      border: 1px solid #aaa;
      margin-top: 20px;
      background: #fafafa;
    }
    .controls {
      margin-bottom: 10px;
    }
    input, button {
      font-size: 16px;
```

```
      padding: 5px;
      margin: 4px;
    }
  </style>
</head>
<body>

<h2>Doubly Linked List (Canvas)</h2>

<div class="controls">
  <input type="number" id="valueInput" placeholder="Value" />
  <input type="number" id="posInput" placeholder="Position" />
  <button onclick="insertNode()">Insert</button>
  <button onclick="deleteNode()">Delete</button>
  <button onclick="highlightNode()">Highlight</button>
</div>

<canvas id="canvas" width="1000" height="200"></canvas>

<script>
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');

const NODE_WIDTH = 90;
const NODE_HEIGHT = 60;
const FIELD_WIDTH = NODE_WIDTH / 3;
const SPACING = 50;
const START_X = 50;
const START_Y = 80;

let list = [10, 20, 30, 40];
let highlightIndex = -1;

function drawNode(x, y, value, highlight = false) {
  // Box
  ctx.fillStyle = highlight ? '#f5a623' : '#4a90e2';
  ctx.strokeStyle = 'black';
  ctx.lineWidth = 2;
  ctx.fillRect(x, y, NODE_WIDTH, NODE_HEIGHT);
  ctx.strokeRect(x, y, NODE_WIDTH, NODE_HEIGHT);

  // Partition fields
  ctx.beginPath();
  ctx.moveTo(x + FIELD_WIDTH, y);
  ctx.lineTo(x + FIELD_WIDTH, y + NODE_HEIGHT);
  ctx.moveTo(x + 2 * FIELD_WIDTH, y);
  ctx.lineTo(x + 2 * FIELD_WIDTH, y + NODE_HEIGHT);
  ctx.stroke();

  // Labels
  ctx.fillStyle = 'white';
  ctx.font = '12px Arial';
  ctx.textAlign = 'center';
  ctx.fillText('.prev', x + FIELD_WIDTH / 2, y + 15);
  ctx.fillText(value, x + NODE_WIDTH / 2, y + NODE_HEIGHT / 2 + 5);
  ctx.fillText('.next', x + FIELD_WIDTH * 2.5, y + 15);
}
```

```
function drawArrow(fromX, fromY, toX, toY, color = 'black') {
  const headlen = 8;
  const angle = Math.atan2(toY - fromY, toX - fromX);

  ctx.strokeStyle = color;
  ctx.fillStyle = color;
  ctx.lineWidth = 2;
  ctx.beginPath();
  ctx.moveTo(fromX, fromY);
  ctx.lineTo(toX, toY);
  ctx.stroke();

  // Arrowhead
  ctx.beginPath();
  ctx.moveTo(toX, toY);
  ctx.lineTo(toX - headlen * Math.cos(angle - Math.PI / 6), toY - headlen * Math.sin(angle - Math.PI / 
  ctx.lineTo(toX - headlen * Math.cos(angle + Math.PI / 6), toY - headlen * Math.sin(angle + Math.PI / 
  ctx.lineTo(toX, toY);
  ctx.fill();
}

function drawList() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  let x = START_X;
  const y = START_Y;

  for (let i = 0; i < list.length; i++) {
    drawNode(x, y, list[i], i === highlightIndex);

    // Connect .next to next node
    if (i < list.length - 1) {
      const fromX = x + FIELD_WIDTH * 2.5;
      const fromY = y + NODE_HEIGHT / 2;
      const toX = x + NODE_WIDTH + SPACING - FIELD_WIDTH / 2;
      const toY = fromY;
      drawArrow(fromX, fromY, toX, toY, '#000');
    }

    // Connect .prev from next node
    if (i > 0) {
      const fromX = x + FIELD_WIDTH / 2;
      const fromY = y + NODE_HEIGHT / 2;
      const toX = x - SPACING + NODE_WIDTH - FIELD_WIDTH * 2.5;
      const toY = fromY;
      drawArrow(fromX, fromY, toX, toY, 'gray');
    }

    x += NODE_WIDTH + SPACING;
  }

  // Head label
  if (list.length > 0) {
    ctx.fillStyle = 'black';
    ctx.font = '14px Arial';
    ctx.textAlign = 'center';
    ctx.fillText('head', START_X + NODE_WIDTH / 2, START_Y - 20);
    drawArrow(START_X + NODE_WIDTH / 2, START_Y - 10, START_X + NODE_WIDTH / 2, START_Y, 'black');
```

```
  }
}

function insertNode() {
  const val = parseInt(document.getElementById("valueInput").value);
  const pos = parseInt(document.getElementById("posInput").value);
  if (isNaN(val) || isNaN(pos) || pos < 0 || pos > list.length) {
    alert("Enter valid value and position.");
    return;
  }
  list.splice(pos, 0, val);
  highlightIndex = pos;
  drawList();
}

function deleteNode() {
  const pos = parseInt(document.getElementById("posInput").value);
  if (isNaN(pos) || pos < 0 || pos >= list.length) {
    alert("Invalid position.");
    return;
  }
  list.splice(pos, 1);
  highlightIndex = -1;
  drawList();
}

function highlightNode() {
  const pos = parseInt(document.getElementById("posInput").value);
  if (isNaN(pos) || pos < 0 || pos >= list.length) {
    alert("Invalid position to highlight.");
    return;
  }
  highlightIndex = pos;
  drawList();
}

// Initial render
drawList();
</script>

</body>
</html>
```

Node Class

```
class DoublyListNode {
  constructor(value) {
    this.value = value;
    this.next = null;
    this.prev = null;
  }
}
```

DoublyLinkedList Class

```
class DoublyLinkedList {
  constructor() {
```

```
    this.head = null;
    this.tail = null;
  }

  insertAtEnd(value) {
    const newNode = new DoublyListNode(value);
    if (!this.tail) {
      this.head = this.tail = newNode;
    } else {
      this.tail.next = newNode;
      newNode.prev = this.tail;
      this.tail = newNode;
    }
  }

  delete(value) {
    let current = this.head;
    while (current && current.value !== value) {
      current = current.next;
    }

    if (!current) return;

    if (current.prev) current.prev.next = current.next;
    else this.head = current.next;

    if (current.next) current.next.prev = current.prev;
    else this.tail = current.prev;
  }

  traverseForward() {
    let current = this.head;
    const values = [];
    while (current) {
      values.push(current.value);
      current = current.next;
    }
    return values;
  }

  traverseBackward() {
    let current = this.tail;
    const values = [];
    while (current) {
      values.push(current.value);
      current = current.prev;
    }
    return values;
  }
}
```

7.2.4 When to Use Linked Lists

Use linked lists when:

- You need **frequent insertions/deletions** (especially not at the end).
- You don't need **random access** (which arrays handle better).
- You want to avoid shifting elements when modifying the list.

Example:

- Implementing a browser history (back/forward): doubly linked list.
- Building a queue or stack from scratch: singly linked list.

7.2.5 Summary

Operation	Array (Dynamic)	Singly LL	Doubly LL
Access by Index	O(1)	O(n)	O(n)
Insert at Head	O(n)	O(1)	O(1)
Insert at Tail	O(1)	O(n)	O(1)
Delete Node	O(n)	O(n)	O(n)
Traverse	O(n)	O(n)	O(n)

While arrays are often the default structure in JavaScript, linked lists offer critical advantages in certain situations. By understanding how they work and when to use them, you'll gain more control over performance and algorithmic efficiency.

Chapter 8.

Stacks and Queues

1. Implementing from Scratch
2. Use Cases in Frontend/Backend

8 Stacks and Queues

8.1 Implementing from Scratch

Stacks and queues are two of the most fundamental data structures in computer science. Though simple in concept, they are incredibly powerful tools for managing data in a controlled, sequential manner. Whether you're parsing expressions, handling undo/redo logic, or implementing search algorithms, stacks and queues provide predictable, efficient behavior.

8.1.1 Stack: LIFO (Last In, First Out)

A **stack** allows access only to the *top* element—like a pile of plates. You can **push** (add) items onto the top and **pop** (remove) the topmost item.

Stack Operations

- `push(value)`: Add an element to the top.
- `pop()`: Remove and return the top element.
- `peek()`: Return the top element without removing it.
- `isEmpty()`: Check if the stack is empty.

Run the following code in browser to see the demo:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Stack Visualization (Canvas)</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      text-align: center;
      padding: 20px;
    }
    canvas {
      border: 1px solid #aaa;
      background: #f9f9f9;
      margin-top: 20px;
    }
    input, button {
      font-size: 16px;
      padding: 6px 10px;
      margin: 5px;
    }
  </style>
</head>
<body>

<h2>Stack Visualization (Canvas)</h2>
```

```
<div>
  <input type="number" id="valueInput" placeholder="Value to push" />
  <button onclick="push()">Push</button>
  <button onclick="pop()">Pop</button>
  <button onclick="highlightTop()">Highlight Top</button>
</div>

<canvas id="canvas" width="300" height="400"></canvas>

<script>
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");

const STACK_X = 100;
const STACK_BOTTOM = 350;
const BOX_WIDTH = 100;
const BOX_HEIGHT = 40;
const BOX_GAP = 10;

let stack = [];
let highlightTopIndex = -1;

function drawStack() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  for (let i = 0; i < stack.length; i++) {
    const y = STACK_BOTTOM - i * (BOX_HEIGHT + BOX_GAP);

    // Draw box
    ctx.fillStyle = (i === highlightTopIndex) ? "#f5a623" : "#4a90e2";
    ctx.strokeStyle = "#333";
    ctx.lineWidth = 2;
    ctx.fillRect(STACK_X, y, BOX_WIDTH, BOX_HEIGHT);
    ctx.strokeRect(STACK_X, y, BOX_WIDTH, BOX_HEIGHT);

    // Draw value
    ctx.fillStyle = "white";
    ctx.font = "18px Arial";
    ctx.textAlign = "center";
    ctx.textBaseline = "middle";
    ctx.fillText(stack[i], STACK_X + BOX_WIDTH / 2, y + BOX_HEIGHT / 2);
  }

  // Draw "Top →" label
  if (stack.length > 0) {
    const topY = STACK_BOTTOM - (stack.length - 1) * (BOX_HEIGHT + BOX_GAP);
    ctx.fillStyle = "black";
    ctx.font = "16px Arial";
    ctx.fillText("Top →", STACK_X - 30, topY + BOX_HEIGHT / 2);
  }
}

function push() {
  const val = parseInt(document.getElementById("valueInput").value);
  if (isNaN(val)) {
    alert("Enter a valid number to push.");
    return;
  }
```

```
  stack.push(val);
  highlightTopIndex = stack.length - 1;
  drawStack();
}

function pop() {
  if (stack.length === 0) {
    alert("Stack is empty.");
    return;
  }
  stack.pop();
  highlightTopIndex = -1;
  drawStack();
}

function highlightTop() {
  if (stack.length === 0) {
    alert("Stack is empty.");
    return;
  }
  highlightTopIndex = stack.length - 1;
  drawStack();
}

// Initial draw
drawStack();
</script>

</body>
</html>
```

Stack Using Array

```
class Stack {
  constructor() {
    this.items = [];
  }

  push(value) {
    this.items.push(value);
  }

  pop() {
    return this.items.pop();
  }

  peek() {
    return this.items[this.items.length - 1];
  }

  isEmpty() {
    return this.items.length === 0;
  }
}
```

Example: Expression Evaluation (Postfix)

```
function evaluatePostfix(expr) {
  const stack = new Stack();
  const tokens = expr.split(' ');

  for (const token of tokens) {
    if (!isNaN(token)) {
      stack.push(Number(token));
    } else {
      const b = stack.pop();
      const a = stack.pop();
      switch (token) {
        case '+': stack.push(a + b); break;
        case '-': stack.push(a - b); break;
        case '*': stack.push(a * b); break;
        case '/': stack.push(a / b); break;
      }
    }
  }

  return stack.pop();
}

console.log(evaluatePostfix("3 4 + 2 *")); // 14
```

Stacks simplify parsing and evaluation by tracking operands and applying operations in order.

8.1.2 Queue: FIFO (First In, First Out)

A **queue** processes items in the order they arrive—like a line at a coffee shop. You **enqueue** elements at the back and **dequeue** them from the front.

Queue Operations

- `enqueue(value)`: Add an element to the back.
- `dequeue()`: Remove and return the front element.
- `peek()`: View the front element.
- `isEmpty()`: Check if the queue is empty.

Run the following code in browser to see the demo:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Queue Visualization (Canvas)</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      text-align: center;
      padding: 20px;
```

```
    }
    canvas {
      border: 1px solid #aaa;
      background: #f9f9f9;
      margin-top: 20px;
    }
    input, button {
      font-size: 16px;
      padding: 6px 10px;
      margin: 5px;
    }
  </style>
</head>
<body>

<h2>Queue Visualization (Canvas)</h2>

<div>
  <input type="number" id="valueInput" placeholder="Value to enqueue" />
  <button onclick="enqueue()">Enqueue</button>
  <button onclick="dequeue()">Dequeue</button>
  <button onclick="highlightHead()">Highlight Head</button>
</div>

<canvas id="canvas" width="900" height="200"></canvas>

<script>
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");

const START_X = 50;
const START_Y = 80;
const BOX_WIDTH = 80;
const BOX_HEIGHT = 50;
const BOX_GAP = 20;

let queue = [];
let highlightHeadIndex = -1;

function drawQueue() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  for (let i = 0; i < queue.length; i++) {
    const x = START_X + i * (BOX_WIDTH + BOX_GAP);

    // Draw box
    ctx.fillStyle = (i === 0 && highlightHeadIndex === 0) ? "#f5a623" : "#4a90e2";
    ctx.strokeStyle = "#333";
    ctx.lineWidth = 2;
    ctx.fillRect(x, START_Y, BOX_WIDTH, BOX_HEIGHT);
    ctx.strokeRect(x, START_Y, BOX_WIDTH, BOX_HEIGHT);

    // Draw value
    ctx.fillStyle = "white";
    ctx.font = "18px Arial";
    ctx.textAlign = "center";
    ctx.textBaseline = "middle";
    ctx.fillText(queue[i], x + BOX_WIDTH / 2, START_Y + BOX_HEIGHT / 2);
```

```
  }

  // Labels for head and tail
  if (queue.length > 0) {
    const headX = START_X;
    const tailX = START_X + (queue.length - 1) * (BOX_WIDTH + BOX_GAP);

    // Head label
    ctx.fillStyle = "black";
    ctx.font = "14px Arial";
    ctx.fillText("Head →", headX + BOX_WIDTH / 2, START_Y - 10);

    // Tail label
    ctx.fillText("Tail →", tailX + BOX_WIDTH / 2, START_Y + BOX_HEIGHT + 20);
  }
}

function enqueue() {
  const val = parseInt(document.getElementById("valueInput").value);
  if (isNaN(val)) {
    alert("Enter a valid number to enqueue.");
    return;
  }
  queue.push(val);
  highlightHeadIndex = -1;
  drawQueue();
}

function dequeue() {
  if (queue.length === 0) {
    alert("Queue is empty.");
    return;
  }
  queue.shift();
  highlightHeadIndex = -1;
  drawQueue();
}

function highlightHead() {
  if (queue.length === 0) {
    alert("Queue is empty.");
    return;
  }
  highlightHeadIndex = 0;
  drawQueue();
}

// Initial draw
drawQueue();
</script>

</body>
</html>
```

Queue Using Array (Simple)

```
class Queue {
  constructor() {
    this.items = [];
  }

  enqueue(value) {
    this.items.push(value);
  }

  dequeue() {
    return this.items.shift(); // O(n) in JS
  }

  peek() {
    return this.items[0];
  }

  isEmpty() {
    return this.items.length === 0;
  }
}
```

Note: `shift()` is O(n) in JavaScript arrays. For high-performance queues, a **linked list** or **two-stack queue** is preferred.

8.1.3 Queue Using Linked List

```
class QueueNode {
  constructor(value) {
    this.value = value;
    this.next = null;
  }
}

class LinkedQueue {
  constructor() {
    this.head = this.tail = null;
  }

  enqueue(value) {
    const node = new QueueNode(value);
    if (!this.tail) {
      this.head = this.tail = node;
    } else {
      this.tail.next = node;
      this.tail = node;
    }
  }

  dequeue() {
    if (!this.head) return null;
    const value = this.head.value;
    this.head = this.head.next;
```

```
    if (!this.head) this.tail = null;
    return value;
  }

  peek() {
    return this.head?.value ?? null;
  }

  isEmpty() {
    return !this.head;
  }
}
```

8.1.4 Example: BFS (Breadth-First Search)

A queue is essential for breadth-first traversal in graphs or trees.

```
function bfs(graph, start) {
  const visited = new Set();
  const queue = new LinkedQueue();

  queue.enqueue(start);
  visited.add(start);

  while (!queue.isEmpty()) {
    const node = queue.dequeue();
    console.log(node);

    for (const neighbor of graph[node]) {
      if (!visited.has(neighbor)) {
        visited.add(neighbor);
        queue.enqueue(neighbor);
      }
    }
  }
}

const graph = {
  A: ['B', 'C'],
  B: ['D'],
  C: ['E'],
  D: [],
  E: ['F'],
  F: []
};

bfs(graph, 'A');
// Output: A B C D E F
```

8.1.5 Summary

Feature	Stack	Queue
Order	LIFO	FIFO
Use Cases	Undo, parsing	BFS, task queues
JS Backing	Array, LinkedList	Array, LinkedList
Performance	Push/Pop: O(1)	Enqueue: O(1), Dequeue: O(1) (Linked)

Stacks and queues simplify a wide range of problems by enforcing strict order on how elements are processed. While JavaScript doesn't offer built-in stack or queue types, implementing them from scratch gives you deeper insight and more control—essential when solving algorithmic challenges or building data flow systems.

8.2 Use Cases in Frontend/Backend

Stacks and queues are not just theoretical data structures—they are deeply embedded in how modern JavaScript applications function. From browser behavior to server-side task processing, stacks and queues help manage **control flow**, **user interactions**, and **asynchronous execution**. In this section, we'll explore key use cases in frontend and backend development, with examples that show how these structures simplify real-world problems.

8.2.1 Undo/Redo Functionality (Frontend Stack)

A common UI feature, especially in text editors or drawing apps, is the ability to *undo* and *redo* user actions. This is a classic stack use case.

```
class HistoryManager {
  constructor() {
    this.undoStack = [];
    this.redoStack = [];
  }

  do(action) {
    this.undoStack.push(action);
    this.redoStack = []; // Clear redo on new action
  }

  undo() {
    if (this.undoStack.length === 0) return;
    const action = this.undoStack.pop();
    this.redoStack.push(action);
    // logic to reverse `action`
```

```
  }

  redo() {
    if (this.redoStack.length === 0) return;
    const action = this.redoStack.pop();
    this.undoStack.push(action);
    // logic to reapply `action`
  }
}
```

Each operation is pushed onto a stack. Undo pops from the undo stack and pushes to the redo stack, and vice versa—LIFO behavior in action.

8.2.2 Call Stack (JavaScript Runtime)

Behind the scenes, the **JavaScript engine** uses a call stack to manage function execution. Every time a function is called, it's *pushed* onto the stack. When the function completes, it's *popped* off.

```
function a() {
  b();
}
function b() {
  c();
}
function c() {
  console.log("End");
}

a();
```

This results in:

```
Call Stack:
- c()
- b()
- a()
```

Understanding this model helps developers trace **stack overflows**, **recursion limits**, and **debugging call traces** in the browser.

8.2.3 Event Queue and Task Scheduling (Frontend Queue)

The **event loop** in JavaScript uses a queue to handle asynchronous tasks. When you use `setTimeout`, `fetch`, or DOM events, the callbacks are queued for execution.

```
console.log("Start");
```

```
setTimeout(() => {
  console.log("Timeout");
}, 0);

console.log("End");
```

Output:

```
Start
End
Timeout
```

The callback to `setTimeout` is added to the **task queue** and executed after the current call stack is clear. This asynchronous queueing system allows JavaScript to be non-blocking.

8.2.4 Task Processing and Job Queues (Backend Queue)

On the server side (e.g., with Node.js), queues are vital for processing background jobs, rate-limiting requests, or managing task retries.

Example: Basic Task Queue for Batch Processing

```
class TaskQueue {
  constructor() {
    this.queue = [];
    this.processing = false;
  }

  enqueue(task) {
    this.queue.push(task);
    this.process();
  }

  async process() {
    if (this.processing) return;
    this.processing = true;

    while (this.queue.length > 0) {
      const task = this.queue.shift(); // FIFO
      await task();
    }

    this.processing = false;
  }
}

// Usage
const queue = new TaskQueue();

queue.enqueue(() => fetchDataFromAPI(1));
queue.enqueue(() => fetchDataFromAPI(2));
```

This simple queue ensures tasks are processed in order and **not concurrently**, which is useful for rate-limited APIs or sequential operations.

8.2.5 Why Use These Structures?

- **Predictability**: Stacks and queues enforce clear rules (LIFO, FIFO), making program flow easier to reason about.
- **Performance**: These structures offer constant-time insertions/removals, which is essential for real-time apps.
- **Modularity**: They help encapsulate logic like command history, job scheduling, or deferred execution cleanly.

8.2.6 Summary

Structure	Use Case	Environment
Stack	Undo/Redo, Call Stack	Frontend
Queue	Event Loop, Async Jobs, Task Queue	Frontend & Backend

Understanding stacks and queues helps you write better-performing, more predictable code—whether you're building a real-time UI or managing background jobs in a serverless function.

Chapter 9.

Hash Tables

1. JavaScript Objects and `Map`
2. Hash Functions and Collisions
3. Custom Hash Table Implementation

9 Hash Tables

9.1 JavaScript Objects and `Map`

Hash tables are a foundational data structure that provide **fast key-based access** to values, typically in constant time (O(1)) for insertion, lookup, and deletion. In JavaScript, you don't have to implement hash tables from scratch to use them—**Objects** and the ES6 `Map` class both serve as built-in hash table implementations.

However, they differ in **key types**, **performance characteristics**, and **usage patterns**. Understanding these differences is crucial for choosing the right one for your needs.

9.1.1 Key Differences: Object vs `Map`

Feature	`Object`	`Map`
Key types	Strings and Symbols only	Any value (objects, numbers, etc.)
Key order	Unordered (in practice, insertion order for string keys)	Guaranteed insertion order
Iteration support	Requires `for...in`, `Object.entries()`	Native with `.forEach()`, spread
Performance (large sets)	Slightly slower in many cases	Optimized for frequent inserts/lookups
Intended use	Structured data, prototypes	General-purpose key-value storage

9.1.2 Using JavaScript Objects as Hash Tables

JavaScript objects (`{}`) allow key-value mapping, where keys are automatically converted to strings (unless they are Symbols). This makes them effective for many simple hash table use cases.

Basic Operations

```
const userScores = {};
userScores["Alice"] = 85;
userScores["Bob"] = 92;

console.log(userScores["Bob"]); // 92
delete userScores["Alice"];
```

Limitation: Non-string Keys

```
const obj = {};
const key1 = {};
const key2 = {};

obj[key1] = "value1";
obj[key2] = "value2";

console.log(obj[key1]); // "value2" - key1 and key2 are both converted to "[object Object]"
```

Objects **cannot distinguish between different object keys**, which can lead to unexpected overwrites.

9.1.3 Using `Map` for Better Key Control

`Map` is an ES6 class designed for hash-table-like behavior, with **any value** as a key (including objects, arrays, and functions). It also maintains the **insertion order** of entries.

Basic Operations

```
const userMap = new Map();

userMap.set("Alice", 85);
userMap.set({ name: "Bob" }, 92);

console.log(userMap.get("Alice")); // 85
console.log(userMap.has("Alice")); // true

userMap.delete("Alice");
```

Unlike objects, `Map` correctly treats different object references as unique keys:

```
const obj1 = { id: 1 };
const obj2 = { id: 1 };

userMap.set(obj1, "A");
console.log(userMap.get(obj2)); // undefined
```

9.1.4 Iterating with `Map`

`Map` provides built-in iteration that works with `forEach`, spread syntax, and `for...of`.

```
for (const [key, value] of userMap) {
  console.log(key, value);
}

// Or convert to array:
console.log([...userMap]); // [[key1, value1], [key2, value2], ...]
```

Objects require conversion:

```
const obj = { a: 1, b: 2 };
Object.entries(obj).forEach(([key, value]) => {
  console.log(key, value);
});
```

9.1.5 When to Use Each

Use Case	Recommended Structure
Static configuration or known property names	`Object`
Dynamic key-value pairs	`Map`
Non-string keys (objects, numbers, etc.)	`Map`
Need for insertion order	`Map`
JSON serialization	`Object`

For most general-purpose applications where you want full control over keys and better iteration behavior, **`Map` is the better choice**. Use `Object` when you're dealing with a fixed set of string keys, like structured JSON or configuration data.

9.1.6 Summary

JavaScript gives you two tools to implement hash tables: **Objects** and **`Map`**. While objects have long served this purpose, `Map` provides more consistent behavior, especially when working with dynamic or non-string keys. Choosing the right one helps improve **clarity**, **performance**, and **reliability** in your code.

9.2 Hash Functions and Collisions

At the heart of every hash table lies a **hash function**—a mathematical process that transforms a key (like a string or object) into an index in an array. This index determines where the associated value should be stored. The goal of a good hash function is to distribute keys **evenly and quickly** across the storage array to minimize collisions and maximize performance.

9.2.1 What Is a Hash Function?

A **hash function** takes an input key and returns a numeric hash value:

```
function simpleHash(str, size) {
  let hash = 0;
  for (let i = 0; i < str.length; i++) {
    hash += str.charCodeAt(i);
  }
  return hash % size;
}
```

For example, both `"abc"` and `"cab"` could result in the same hash if their character sums match—a **collision**.

9.2.2 What Are Collisions?

A **collision** occurs when two different keys are hashed to the same index in the array. Since hash tables store data at indices determined by hash functions, a collision means two values are competing for the same spot.

Analogy:

Think of a hash table like a row of lockers. A hash function assigns students to lockers. If two students are assigned locker #4, that's a collision. The locker must now accommodate both, or one must be reassigned.

9.2.3 Collision Resolution Strategies

There are two major approaches to handling collisions:

Chaining

In **chaining**, each array index points to a list (or "bucket") of entries that share the same hash.

```
// Pseudo structure:
table[hash] = [ { key: "a", value: 1 }, { key: "b", value: 2 } ];
```

When inserting:

- Compute hash.
- If the slot is empty, create a new bucket.
- If not, add the entry to the bucket.

When looking up:

- Compute hash.
- Search the bucket for the matching key.

Pros:

- Simple to implement.
- Handles unlimited collisions per index.

Cons:

- Requires extra memory for lists.
- Performance can degrade if many keys land in the same bucket.

Open Addressing

In **open addressing**, if a collision occurs, the algorithm finds another open slot in the array using a predefined strategy like **linear probing**, **quadratic probing**, or **double hashing**.

Example of linear probing:

```
// If hash("apple") = 3 but slot 3 is full, check 4, then 5, etc.
```

Pros:

- Avoids using additional memory for buckets.
- Cache-friendly due to array locality.

Cons:

- Performance drops as the table fills up.
- Needs resizing logic to maintain efficiency.

9.2.4 Trade-Offs and Challenges

Creating an effective hash table isn't just about writing a hash function—it's about balancing:

- **Speed**: A hash function should be fast to compute.
- **Uniformity**: Keys should be evenly distributed to avoid clusters.
- **Scalability**: As more data is added, the table should resize or rehash efficiently.
- **Load Factor**: A measure of how full the table is. A high load factor increases collision risk.

9.2.5 Summary

Hash functions are the foundation of hash tables, enabling fast access by converting keys to array indices. But collisions are inevitable due to limited storage space. To resolve them, we use techniques like **chaining** (linked lists at each index) or **open addressing**

(finding alternative slots). The choice of strategy affects performance, memory usage, and complexity—understanding these trade-offs is key to building efficient hashing systems in JavaScript or any language.

9.3 Custom Hash Table Implementation

To truly understand how hash tables work, it's valuable to build one from scratch. In this section, we'll implement a basic hash table in JavaScript, complete with a simple hash function and basic collision handling using **separate chaining** (linked lists or arrays at each index).

This exercise will highlight both the mechanics and challenges of building hash tables, from hashing keys to resolving collisions and managing load.

9.3.1 Building Blocks

Our custom hash table will include:

- A basic **hash function** to map string keys to numeric indices.
- A **storage array** to hold key-value pairs.
- **Separate chaining** for collision resolution using arrays as buckets.

9.3.2 Step 1: Basic Hash Function

We'll start with a simple (but not cryptographically secure) hash function that sums character codes.

```
function hash(key, size) {
  let hashValue = 0;
  for (let i = 0; i < key.length; i++) {
    hashValue += key.charCodeAt(i);
  }
  return hashValue % size;
}
```

> This function distributes keys in a repeatable way but can lead to collisions, especially with similar strings.

9.3.3 Step 2: Hash Table Class with Chaining

We'll create a class called `HashTable` that uses an array of buckets. Each bucket will be an array to hold key-value pairs.

```
class HashTable {
  constructor(size = 16) {
    this.size = size;
    this.buckets = Array(size).fill(null).map(() => []);
  }

  _hash(key) {
    return hash(key, this.size);
  }

  set(key, value) {
    const index = this._hash(key);
    const bucket = this.buckets[index];

    // Check if key already exists and update
    for (let pair of bucket) {
      if (pair[0] === key) {
        pair[1] = value;
        return;
      }
    }

    // Otherwise, insert new key-value pair
    bucket.push([key, value]);
  }

  get(key) {
    const index = this._hash(key);
    const bucket = this.buckets[index];

    for (let pair of bucket) {
      if (pair[0] === key) {
        return pair[1];
      }
    }

    return undefined;
  }

  delete(key) {
    const index = this._hash(key);
    const bucket = this.buckets[index];

    for (let i = 0; i < bucket.length; i++) {
      if (bucket[i][0] === key) {
        bucket.splice(i, 1);
        return true;
      }
    }

    return false;
  }
```

```
  has(key) {
    return this.get(key) !== undefined;
  }

  keys() {
    const allKeys = [];
    for (let bucket of this.buckets) {
      for (let pair of bucket) {
        allKeys.push(pair[0]);
      }
    }
    return allKeys;
  }
}
```

9.3.4 Example Usage

```
const table = new HashTable();

table.set("name", "Alice");
table.set("age", 30);
table.set("occupation", "Engineer");

console.log(table.get("name"));       // "Alice"
console.log(table.has("age"));        // true
console.log(table.get("location"));   // undefined

table.delete("age");
console.log(table.has("age"));        // false

console.log(table.keys());            // ['name', 'occupation']
```

Full runnable code:

```
// === Step 1: Basic Hash Function ===
function hash(key, size) {
  let hashValue = 0;
  for (let i = 0; i < key.length; i++) {
    hashValue += key.charCodeAt(i);
  }
  return hashValue % size;
}

// === Step 2: Hash Table Class with Chaining ===
class HashTable {
  constructor(size = 16) {
    this.size = size;
    this.buckets = Array(size).fill(null).map(() => []);
  }

  _hash(key) {
    return hash(key, this.size);
  }
```

```
  set(key, value) {
    const index = this._hash(key);
    const bucket = this.buckets[index];

    for (let pair of bucket) {
      if (pair[0] === key) {
        pair[1] = value;
        return;
      }
    }

    bucket.push([key, value]);
  }

  get(key) {
    const index = this._hash(key);
    const bucket = this.buckets[index];

    for (let pair of bucket) {
      if (pair[0] === key) {
        return pair[1];
      }
    }

    return undefined;
  }

  delete(key) {
    const index = this._hash(key);
    const bucket = this.buckets[index];

    for (let i = 0; i < bucket.length; i++) {
      if (bucket[i][0] === key) {
        bucket.splice(i, 1);
        return true;
      }
    }

    return false;
  }

  has(key) {
    return this.get(key) !== undefined;
  }

  keys() {
    const allKeys = [];
    for (let bucket of this.buckets) {
      for (let pair of bucket) {
        allKeys.push(pair[0]);
      }
    }
    return allKeys;
  }
}

// === Example Usage ===
function testHashTable() {
```

```
  const table = new HashTable();

  table.set("name", "Alice");
  table.set("age", 30);
  table.set("occupation", "Engineer");

  console.log("Get name:", table.get("name"));          // "Alice"
  console.log("Has age:", table.has("age"));            // true
  console.log("Get location:", table.get("location")); // undefined

  table.delete("age");
  console.log("Has age after delete:", table.has("age")); // false

  console.log("Keys:", table.keys()); // ['name', 'occupation']
}

// Run test
testHashTable();
```

9.3.5 Limitations of This Implementation

While this implementation works for basic cases, it has several limitations:

Poor Hash Distribution

The simple hash function is prone to collisions, especially with short or similar strings. In real-world applications, better hash functions (like DJB2 or FNV-1a) should be used.

Fixed Size

The hash table has a fixed number of buckets (`size = 16`). As more elements are added, performance degrades due to growing bucket arrays. A production hash table should **resize** (rehash) when the **load factor** (items / size) passes a threshold.

Key Type Support

This implementation assumes string keys. Real-world hash tables (like `Map`) can support any value type, including objects. Supporting this requires object identity and more sophisticated hashing.

9.3.6 Potential Improvements

To make this hash table more robust:

- Use a **better hash function** to improve key distribution.
- Track the number of elements and **dynamically resize** the internal array when needed.
- Support **arbitrary key types** (objects, numbers) by adding serialization or key

mapping logic.
- Convert buckets from arrays to **linked lists** for consistent insert/delete time.

9.3.7 Summary

Implementing a hash table from scratch gives deep insight into how JavaScript structures like `Map` and `Object` work under the hood. Through hashing, collision resolution, and bucket storage, we've built a simplified but functional version of a critical data structure. Try modifying the hash function or resizing logic—this experimentation will reinforce how essential efficiency, distribution, and memory handling are in algorithm design.

Chapter 10.

Trees and Binary Search Trees

1. Recursive Tree Traversals
2. Balanced vs Unbalanced Trees
3. Binary Search Tree from Scratch

10 Trees and Binary Search Trees

10.1 Recursive Tree Traversals

In tree data structures, **traversal** refers to visiting each node in a specific order. Unlike arrays or linked lists, trees are hierarchical and non-linear, so multiple traversal strategies exist. The three most common **depth-first traversal** techniques are:

- **Preorder Traversal**
- **Inorder Traversal**
- **Postorder Traversal**

Each of these follows a different order in processing the **current node**, its **left subtree**, and **right subtree**.

All three can be naturally implemented using **recursion**, which fits the hierarchical nature of trees: each node's children are themselves roots of smaller subtrees.

10.1.1 Tree Structure Example

To illustrate, consider this binary tree:

This structure will help visualize the different traversal orders.

Run the following code in browser to see the demo:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Tree Traversal Visualization (Canvas)</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      text-align: center;
      margin: 20px;
    }
    canvas {
      border: 1px solid #ccc;
      background: #f9f9f9;
    }
    button {
      margin: 5px;
```

```
      padding: 8px 16px;
      font-size: 16px;
    }
    #visitedOutput {
      margin-top: 20px;
      font-size: 18px;
      min-height: 30px;
    }
    .node-box {
      display: inline-block;
      background: #4a90e2;
      color: white;
      padding: 5px 10px;
      border-radius: 5px;
      margin: 2px;
    }
    .message {
      margin-top: 10px;
      color: green;
      font-weight: bold;
    }
  </style>
</head>
<body>

<h2>Tree Traversal Visualization (Canvas)</h2>

<div>
  <button onclick="startTraversal('preorder')">Preorder</button>
  <button onclick="startTraversal('inorder')">Inorder</button>
  <button onclick="startTraversal('postorder')">Postorder</button>
</div>

<canvas id="canvas" width="800" height="400"></canvas>

<div id="visitedOutput"></div>
<div id="message" class="message"></div>

<script>
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");
const visitedOutput = document.getElementById("visitedOutput");
const messageBox = document.getElementById("message");

const NODE_RADIUS = 20;
const LEVEL_HEIGHT = 80;

class TreeNode {
  constructor(value, x = 0, y = 0) {
    this.value = value;
    this.left = null;
    this.right = null;
    this.x = x;
    this.y = y;
  }
}

// Sample Binary Tree
```

```
//        A
//      /   \
//     B     C
//    / \   / \
//   D   E F   G

const A = new TreeNode("A");
const B = new TreeNode("B");
const C = new TreeNode("C");
const D = new TreeNode("D");
const E = new TreeNode("E");
const F = new TreeNode("F");
const G = new TreeNode("G");

A.left = B; A.right = C;
B.left = D; B.right = E;
C.left = F; C.right = G;

const treeRoot = A;

// Assign coordinates for display
function assignCoordinates(node, depth, index, spacing) {
  if (!node) return;
  node.x = index * spacing + spacing / 2;
  node.y = depth * LEVEL_HEIGHT + NODE_RADIUS * 2;
  assignCoordinates(node.left, depth + 1, index * 2, spacing / 2);
  assignCoordinates(node.right, depth + 1, index * 2 + 1, spacing / 2);
}

function drawTree(node) {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  drawConnections(node);
  drawNodes(node);
}

function drawConnections(node) {
  if (!node) return;
  ctx.strokeStyle = "#aaa";
  ctx.lineWidth = 2;

  if (node.left) {
    ctx.beginPath();
    ctx.moveTo(node.x, node.y);
    ctx.lineTo(node.left.x, node.left.y);
    ctx.stroke();
    drawConnections(node.left);
  }

  if (node.right) {
    ctx.beginPath();
    ctx.moveTo(node.x, node.y);
    ctx.lineTo(node.right.x, node.right.y);
    ctx.stroke();
    drawConnections(node.right);
  }
}

function drawNodes(node, highlightNode = null) {
```

```
  if (!node) return;

  // Draw current node
  ctx.beginPath();
  ctx.fillStyle = (node === highlightNode) ? "#f5a623" : "#4a90e2";
  ctx.strokeStyle = "#333";
  ctx.lineWidth = 2;
  ctx.arc(node.x, node.y, NODE_RADIUS, 0, 2 * Math.PI);
  ctx.fill();
  ctx.stroke();

  // Draw text
  ctx.fillStyle = "#fff";
  ctx.font = "16px Arial";
  ctx.textAlign = "center";
  ctx.textBaseline = "middle";
  ctx.fillText(node.value, node.x, node.y);

  drawNodes(node.left, highlightNode);
  drawNodes(node.right, highlightNode);
}

// Traversal functions
function preorder(node, result = []) {
  if (!node) return result;
  result.push(node);
  preorder(node.left, result);
  preorder(node.right, result);
  return result;
}

function inorder(node, result = []) {
  if (!node) return result;
  inorder(node.left, result);
  result.push(node);
  inorder(node.right, result);
  return result;
}

function postorder(node, result = []) {
  if (!node) return result;
  postorder(node.left, result);
  postorder(node.right, result);
  result.push(node);
  return result;
}

function startTraversal(type) {
  visitedOutput.innerHTML = "";
  messageBox.innerText = "";

  let order;
  if (type === "preorder") {
    order = preorder(treeRoot);
  } else if (type === "inorder") {
    order = inorder(treeRoot);
  } else if (type === "postorder") {
    order = postorder(treeRoot);
```

```
  }

  let i = 0;
  function step() {
    if (i >= order.length) {
      messageBox.innerText = `YES ${type[0].toUpperCase() + type.slice(1)} traversal complete!`;
      return;
    }
    drawTree(treeRoot);
    drawNodes(treeRoot, order[i]);

    // Show visited node under canvas
    const nodeBox = document.createElement("span");
    nodeBox.className = "node-box";
    nodeBox.textContent = order[i].value;
    visitedOutput.appendChild(nodeBox);

    i++;
    setTimeout(step, 800);
  }

  step();
}

// Initialize
assignCoordinates(treeRoot, 0, 1, canvas.width / 2);
drawTree(treeRoot);
</script>

</body>
</html>
```

10.1.2 Preorder Traversal: `Node Left Right`

In **preorder**, you visit the current node before its children.

JavaScript Implementation

```
function preorder(node) {
  if (!node) return;

  console.log(node.value);      // Visit node
  preorder(node.left);          // Traverse left subtree
  preorder(node.right);         // Traverse right subtree
}
```

Order of Visit (for the sample tree):

`A → B → D → E → C → F`

Applications

- Copying trees

- Generating prefix expressions from expression trees
- Serializing a tree structure

10.1.3 Inorder Traversal: `Left Node Right`

In **inorder**, you process the left subtree first, then the current node, then the right subtree.

JavaScript Implementation

```
function inorder(node) {
  if (!node) return;

  inorder(node.left);           // Traverse left subtree
  console.log(node.value);      // Visit node
  inorder(node.right);          // Traverse right subtree
}
```

Order of Visit (for the sample tree):

`D → B → E → A → C → F`

Applications

- **Binary Search Trees**: Inorder traversal returns values in **sorted order**.
- Expression trees: Producing **infix notation**.

10.1.4 Postorder Traversal: `Left Right Node`

In **postorder**, you visit children before the parent.

JavaScript Implementation

```
function postorder(node) {
  if (!node) return;

  postorder(node.left);         // Traverse left subtree
  postorder(node.right);        // Traverse right subtree
  console.log(node.value);      // Visit node
}
```

Order of Visit (for the sample tree):

`D → E → B → F → C → A`

Applications

- Deleting or freeing trees (children removed before parents)

- Generating **postfix expressions** from expression trees
- Evaluating expression trees

10.1.5 Example: Tree Node Class

Here's a simple class to define binary tree nodes for our examples:

```
class TreeNode {
  constructor(value) {
    this.value = value;
    this.left = null;
    this.right = null;
  }
}
```

```
// Build sample tree:
//          A
//         / \
//        B   C
//       / \   \
//      D   E   F

const root = new TreeNode("A");
root.left = new TreeNode("B");
root.right = new TreeNode("C");
root.left.left = new TreeNode("D");
root.left.right = new TreeNode("E");
root.right.right = new TreeNode("F");
```

Now you can call any traversal:

```
inorder(root);   // Output: D B E A C F
```

10.1.6 Visual Comparison Summary

Traversal	Order	Example Output
Preorder	Node → Left → Right	A B D E C F
Inorder	Left → Node → Right	D B E A C F
Postorder	Left → Right → Node	D E B F C A

10.1.7 Why Recursion Works

Recursive traversal mirrors the recursive nature of trees: each node is the root of its own subtree. By breaking the tree into smaller problems (left and right subtrees), recursion

processes nodes elegantly in the desired order.

10.1.8 Practical Use Cases

- **Expression Trees**: Preorder generates prefix (Polish) notation, inorder gives infix (human-readable), and postorder gives postfix (Reverse Polish Notation).
- **Searching**: In binary search trees, an inorder traversal visits nodes in sorted order.
- **File Systems**: Directory trees can be traversed recursively using preorder or postorder depending on the task (e.g. scanning or deleting files).

10.1.9 Summary

Recursive tree traversals—**preorder**, **inorder**, and **postorder**—are fundamental tools for navigating and processing trees. Each follows a distinct order and serves specific use cases in algorithms and real-world applications. With a few lines of recursive JavaScript code, you can explore and manipulate complex tree structures with clarity and control.

10.2 Balanced vs Unbalanced Trees

A fundamental concept in working with trees, especially **Binary Search Trees (BSTs)**, is whether the tree is **balanced** or **unbalanced**. The shape of a tree dramatically impacts how efficiently you can perform operations like search, insertion, and deletion.

10.2.1 What Is a Balanced Tree?

A tree is considered **balanced** when the heights of its left and right subtrees for every node are roughly equal. More formally, for any node in the tree, the difference in height between its left and right child subtrees is at most 1 (this is the condition for **AVL trees**, a common balanced BST).

Balanced trees keep their height **as small as possible**, close to `O(log n)` for `n` nodes, meaning operations that depend on height remain efficient.

10.2.2 What Is an Unbalanced Tree?

An **unbalanced** tree is one where the subtrees differ significantly in height. In the worst case, a BST can degenerate into a structure like a linked list, with all nodes skewed to one side (left or right).

10.2.3 Height and Its Impact on Performance

The **height** of a tree is the length of the longest path from the root to a leaf node.

Tree Type	Height	Search Time Complexity
Balanced BST	~log (n)	O(log n)
Unbalanced (skewed)	~n (linear)	O(n)

Diagram: Balanced vs Skewed BST (6 nodes)

Balanced BST:

```
Tree:

      4
    /   \
   2     6
  / \   /
 1   3 5
```

Skewed BST:

```
1
 \
  2
   \
    3
     \
      4
       \
        5
         \
          6
```

Notice how the balanced tree has height 3 (log 6 2.58), but the skewed tree's height is 6, the number of nodes.

10.2.4 Consequences of Unbalance

In an unbalanced tree:

- **Search operations degrade** to linear time, as you may need to traverse many nodes.
- **Insertion and deletion** can also become costly.
- This negates the main advantage of BSTs, which is fast searching.

10.2.5 Balancing Strategies (Conceptual)

To maintain efficient operations, self-balancing trees adjust their structure automatically during insertions and deletions:

- **AVL Trees**: Strictly maintain balance by checking heights and performing rotations after insert/delete.
- **Red-Black Trees**: Maintain a looser balance using color properties and rotations for balancing.
- **B-Trees**: Used in databases and filesystems, optimized for storage on disk blocks.

These trees ensure height remains close to `O(log n)` and preserve fast search times.

10.2.6 JavaScript Example: Searching Balanced vs Skewed Trees

Full runnable code:

```
class TreeNode {
  constructor(value) {
    this.value = value;
    this.left = null;
    this.right = null;
  }
}

// Insert nodes to create a skewed BST (like a linked list)
function createSkewedTree() {
  let root = new TreeNode(1);
  let current = root;
  for (let i = 2; i <= 6; i++) {
    current.right = new TreeNode(i);
    current = current.right;
  }
  return root;
}

// Insert nodes to create a balanced BST manually
function createBalancedTree() {
  const root = new TreeNode(4);
```

```
  root.left = new TreeNode(2);
  root.right = new TreeNode(6);
  root.left.left = new TreeNode(1);
  root.left.right = new TreeNode(3);
  root.right.left = new TreeNode(5);
  return root;
}

function searchBST(root, target) {
  if (!root) return false;
  if (root.value === target) return true;
  if (target < root.value) return searchBST(root.left, target);
  return searchBST(root.right, target);
}

const balanced = createBalancedTree();
const skewed = createSkewedTree();

console.time("Balanced Search");
console.log(searchBST(balanced, 6)); // true
console.timeEnd("Balanced Search");

console.time("Skewed Search");
console.log(searchBST(skewed, 6)); // true
console.timeEnd("Skewed Search");
```

You'll notice the skewed tree search takes longer because it must traverse more nodes sequentially, while the balanced tree cuts down the search path quickly.

10.2.7 Summary

The **balance** of a tree is critical for maintaining the **logarithmic efficiency** of search and update operations. Balanced trees keep their height small and operations fast, while unbalanced trees risk degenerating into inefficient structures. Self-balancing trees like AVL and Red-Black trees automate maintaining this balance, ensuring predictable performance—making them essential for practical use of BSTs in real applications.

10.3 Binary Search Tree from Scratch

A **Binary Search Tree (BST)** is a powerful data structure that maintains sorted data and allows fast insertion, search, and deletion operations. Each node in a BST has at most two children: a **left** child containing values less than the node's value, and a **right** child containing values greater than the node's value. This property ensures that in-order traversal of a BST returns values in **sorted order**.

In this section, we'll build a BST from scratch in JavaScript, covering insertion, search, and

deletion with clear explanations.

Run the following code in browser to see the demo:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Animated AVL Tree Visualization</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      text-align: center;
      margin: 20px;
    }
    canvas {
      border: 1px solid #ccc;
      background: #f4f4f4;
      display: block;
      margin: 10px auto;
    }
    input, button {
      padding: 6px 10px;
      font-size: 16px;
      margin: 5px;
    }
    #output {
      margin-top: 15px;
      font-size: 18px;
    }
  </style>
</head>
<body>

<h2>Animated AVL Tree with Search Path Highlight</h2>

<input type="number" id="valueInput" placeholder="Enter number">
<button onclick="insert()">Insert</button>
<button onclick="remove()">Delete</button>
<button onclick="showInorder()">Inorder Traversal</button>

<canvas id="canvas" width="1000" height="500"></canvas>
<div id="output"></div>

<script>
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");
const output = document.getElementById("output");

class AVLNode {
  constructor(value) {
    this.value = value;
    this.left = null;
    this.right = null;
    this.height = 1;
    this.x = 0;
    this.y = 0;
  }
```

```
}

let root = null;
let highlightNodes = [];

function height(n) {
  return n ? n.height : 0;
}
function updateHeight(n) {
  n.height = 1 + Math.max(height(n.left), height(n.right));
}
function balanceFactor(n) {
  return height(n.left) - height(n.right);
}
function rotateRight(y) {
  const x = y.left;
  const T2 = x.right;
  x.right = y;
  y.left = T2;
  updateHeight(y);
  updateHeight(x);
  return x;
}
function rotateLeft(x) {
  const y = x.right;
  const T2 = y.left;
  y.left = x;
  x.right = T2;
  updateHeight(x);
  updateHeight(y);
  return y;
}
function balance(node) {
  updateHeight(node);
  const bf = balanceFactor(node);
  if (bf > 1) {
    if (balanceFactor(node.left) < 0) node.left = rotateLeft(node.left);
    return rotateRight(node);
  }
  if (bf < -1) {
    if (balanceFactor(node.right) > 0) node.right = rotateRight(node.right);
    return rotateLeft(node);
  }
  return node;
}

function insertNode(node, value, path = []) {
  if (!node) {
    highlightNodes = path;
    return new AVLNode(value);
  }
  path.push(node);
  if (value < node.value) node.left = insertNode(node.left, value, path);
  else if (value > node.value) node.right = insertNode(node.right, value, path);
  else {
    highlightNodes = path;
    return node;
  }
```

```
  return balance(node);
}

function deleteNode(node, value, path = []) {
  if (!node) {
    highlightNodes = path;
    return null;
  }
  path.push(node);
  if (value < node.value) node.left = deleteNode(node.left, value, path);
  else if (value > node.value) node.right = deleteNode(node.right, value, path);
  else {
    if (!node.left || !node.right) return node.left || node.right;
    const minLarger = minValueNode(node.right);
    node.value = minLarger.value;
    node.right = deleteNode(node.right, minLarger.value);
  }
  return balance(node);
}

function minValueNode(n) {
  while (n.left) n = n.left;
  return n;
}

function inorder(node, res = []) {
  if (!node) return res;
  inorder(node.left, res);
  res.push(node.value);
  inorder(node.right, res);
  return res;
}

// === Tree Layout and Drawing ===

function layoutTree(node, depth = 0, range = [50, canvas.width - 50]) {
  if (!node) return;
  const [start, end] = range;
  const mid = (start + end) / 2;
  node.x = mid;
  node.y = 60 + depth * 70;
  layoutTree(node.left, depth + 1, [start, mid - 30]);
  layoutTree(node.right, depth + 1, [mid + 30, end]);
}

function drawTree(node) {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  layoutTree(node);
  drawEdges(node);
  drawNodes(node);
}

function drawEdges(n) {
  if (!n) return;
  ctx.strokeStyle = "#999";
  if (n.left) {
    ctx.beginPath();
    ctx.moveTo(n.x, n.y);
```

```
    ctx.lineTo(n.left.x, n.left.y);
    ctx.stroke();
    drawEdges(n.left);
  }
  if (n.right) {
    ctx.beginPath();
    ctx.moveTo(n.x, n.y);
    ctx.lineTo(n.right.x, n.right.y);
    ctx.stroke();
    drawEdges(n.right);
  }
}

function drawNodes(n) {
  if (!n) return;
  ctx.beginPath();
  ctx.fillStyle = highlightNodes.includes(n) ? "#f39c12" : "#3498db";
  ctx.arc(n.x, n.y, 20, 0, Math.PI * 2);
  ctx.fill();
  ctx.strokeStyle = "#000";
  ctx.stroke();

  ctx.fillStyle = "#fff";
  ctx.font = "16px Arial";
  ctx.textAlign = "center";
  ctx.textBaseline = "middle";
  ctx.fillText(n.value, n.x, n.y);

  drawNodes(n.left);
  drawNodes(n.right);
}

// === Controls ===

function insert() {
  const val = parseInt(document.getElementById("valueInput").value);
  if (isNaN(val)) return;
  highlightNodes = [];
  root = insertNode(root, val);
  drawTree(root);
  setTimeout(() => {
    highlightNodes = [];
    drawTree(root);
  }, 1000);
  document.getElementById("valueInput").value = "";
  output.textContent = "";
}

function remove() {
  const val = parseInt(document.getElementById("valueInput").value);
  if (isNaN(val)) return;
  highlightNodes = [];
  root = deleteNode(root, val);
  drawTree(root);
  setTimeout(() => {
    highlightNodes = [];
    drawTree(root);
  }, 1000);
```

```
  document.getElementById("valueInput").value = "";
  output.textContent = "";
}

function showInorder() {
  const result = inorder(root);
  output.textContent = "Inorder Traversal (Sorted): " + result.join(", ");
}

// Initial draw
drawTree(root);
</script>

</body>
</html>
```

10.3.1 BST Node Structure

First, define a simple `TreeNode` class that holds a value and pointers to left and right children.

```
class TreeNode {
  constructor(value) {
    this.value = value;
    this.left = null;
    this.right = null;
  }
}
```

10.3.2 Insertion

When inserting a value into a BST:

- If the tree is empty, the new node becomes the root.
- Otherwise, compare the new value with the current node:
  - If less, recurse left.
  - If greater, recurse right.
- Insert the new node where the subtree is empty (`null`).

```
class BinarySearchTree {
  constructor() {
    this.root = null;
  }

  insert(value) {
    this.root = this._insertRec(this.root, value);
  }
```

```
  _insertRec(node, value) {
    if (node === null) {
      return new TreeNode(value);
    }

    if (value < node.value) {
      node.left = this._insertRec(node.left, value);
    } else if (value > node.value) {
      node.right = this._insertRec(node.right, value);
    }
    // If value equals node.value, duplicates are ignored here

    return node;
  }
}
```

10.3.3 Searching

Searching a BST takes advantage of the order property:

- If the node is `null`, the value isn't found.
- If the current node's value equals the target, return true.
- If target $<$ node's value, search left.
- Otherwise, search right.

```
  search(value) {
    return this._searchRec(this.root, value);
  }

  _searchRec(node, value) {
    if (node === null) return false;
    if (node.value === value) return true;

    if (value < node.value) {
      return this._searchRec(node.left, value);
    } else {
      return this._searchRec(node.right, value);
    }
  }
```

10.3.4 Deletion

Deletion is the trickiest BST operation because it requires maintaining the BST property after removal.

There are three cases when deleting a node:

1. **Node with no children (leaf):** Simply remove it by returning `null`.

2. **Node with one child:** Replace the node with its child.
3. **Node with two children:** Replace the node's value with the **minimum value in its right subtree** (in-order successor), then delete that successor.

```
delete(value) {
  this.root = this._deleteRec(this.root, value);
}

_deleteRec(node, value) {
  if (node === null) return null;

  if (value < node.value) {
    node.left = this._deleteRec(node.left, value);
  } else if (value > node.value) {
    node.right = this._deleteRec(node.right, value);
  } else {
    // Node to delete found

    // Case 1: No children
    if (!node.left && !node.right) {
      return null;
    }

    // Case 2: One child
    if (!node.left) return node.right;
    if (!node.right) return node.left;

    // Case 3: Two children
    const minRight = this._minValueNode(node.right);
    node.value = minRight.value;
    node.right = this._deleteRec(node.right, minRight.value);
  }
  return node;
}

_minValueNode(node) {
  while (node.left !== null) {
    node = node.left;
  }
  return node;
}
```

10.3.5 Inorder Traversal (Sorting)

An in-order traversal of a BST visits nodes in sorted ascending order:

```
inorder() {
  const result = [];
  this._inorderRec(this.root, result);
  return result;
}

_inorderRec(node, result) {
  if (!node) return;
```

```
    this._inorderRec(node.left, result);
    result.push(node.value);
    this._inorderRec(node.right, result);
  }
```

10.3.6 Complete Example and Test Cases

```
const bst = new BinarySearchTree();

bst.insert(50);
bst.insert(30);
bst.insert(70);
bst.insert(20);
bst.insert(40);
bst.insert(60);
bst.insert(80);

console.log(bst.search(40));   // true
console.log(bst.search(25));   // false

console.log("Inorder:", bst.inorder());
// Output: [20, 30, 40, 50, 60, 70, 80]

bst.delete(20);  // Delete leaf
bst.delete(30);  // Delete node with one child
bst.delete(50);  // Delete node with two children (root)

console.log("Inorder after deletes:", bst.inorder());
// Output: [40, 60, 70, 80]
```

10.3.7 Key Takeaways

- BSTs maintain **order** by placing smaller values in the left subtree and larger in the right.
- Recursive insertion and search naturally fit the tree's hierarchical structure.
- Deletion requires careful handling to maintain tree properties.
- Inorder traversal returns sorted values, making BSTs useful for dynamic sorting and searching.

10.3.8 Summary

Building a BST from scratch demystifies its internal workings. By mastering insertion, search, and deletion, you gain insight into how balanced trees and more advanced data

structures function. BSTs form the backbone of many algorithms and libraries, making them a foundational skill in JavaScript algorithmic programming.

Chapter 11.

Heaps and Priority Queues

1. Min-Heap, Max-Heap
2. Heap Sort
3. Implementing a Priority Queue

11 Heaps and Priority Queues

11.1 Min-Heap, Max-Heap

A **heap** is a specialized tree-based data structure that satisfies two important properties:

- It is a **complete binary tree**, meaning all levels are fully filled except possibly the last, which is filled from left to right.
- It maintains the **heap order property**, which differs depending on whether it is a min-heap or max-heap.

Heaps are widely used in algorithms requiring **priority access** to elements, such as priority queues and heap sort.

Run the following code in browser to see the demo:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Heap Visualizer</title>
  <style>
    body {
      font-family: Arial;
      text-align: center;
      padding: 20px;
    }
    canvas {
      border: 1px solid #ccc;
      background: #f8f8f8;
      margin-top: 10px;
    }
    input, button, select {
      margin: 5px;
      padding: 6px 10px;
      font-size: 16px;
    }
    #output {
      margin-top: 15px;
      font-size: 18px;
    }
  </style>
</head>
<body>

<h2>Heap Visualizer (Min/Max Toggle + Step-by-Step)</h2>

<input type="number" id="valueInput" placeholder="Enter number">
<button onclick="prepareInsert()">Insert</button>
<button onclick="prepareDelete()">Delete Root</button>
<select id="heapType" onchange="setHeapType()">
  <option value="max">Max-Heap</option>
  <option value="min">Min-Heap</option>
</select>
<button onclick="stepForward()">Next</button>
```

```
<button onclick="stepBackward()">Back</button>

<canvas id="canvas" width="1000" height="500"></canvas>
<div id="output"></div>

<script>
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");

let heap = [];
let heapType = "max";
let animations = [];
let currentStep = 0;

function compare(a, b) {
  return heapType === "min" ? a < b : a > b;
}

function setHeapType() {
  heapType = document.getElementById("heapType").value;
}

function prepareInsert() {
  const value = parseInt(document.getElementById("valueInput").value);
  if (isNaN(value)) return;
  const steps = [];
  heap.push(value);
  let idx = heap.length - 1;
  steps.push({ type: 'insert', index: idx, value });
  while (idx > 0) {
    const parent = Math.floor((idx - 1) / 2);
    steps.push({ type: 'compare', i: idx, j: parent });
    if (compare(heap[idx], heap[parent])) {
      steps.push({ type: 'swap', i: idx, j: parent });
      [heap[idx], heap[parent]] = [heap[parent], heap[idx]];
      idx = parent;
    } else break;
  }
  steps.push({ type: 'done' });
  animations = steps;
  currentStep = 0;
  drawHeap();
}

function prepareDelete() {
  if (heap.length === 0) return;
  const steps = [];
  const last = heap.pop();
  if (heap.length === 0) {
    drawHeap();
    return;
  }
  heap[0] = last;
  let idx = 0;
  steps.push({ type: 'replaceRoot', value: last });
  while (true) {
    const left = 2 * idx + 1;
    const right = 2 * idx + 2;
```

```
    let target = idx;
    if (left < heap.length && compare(heap[left], heap[target])) target = left;
    if (right < heap.length && compare(heap[right], heap[target])) target = right;
    if (target !== idx) {
      steps.push({ type: 'swap', i: idx, j: target });
      [heap[idx], heap[target]] = [heap[target], heap[idx]];
      idx = target;
    } else break;
  }
  steps.push({ type: 'done' });
  animations = steps;
  currentStep = 0;
  drawHeap();
}

function stepForward() {
  if (currentStep >= animations.length) return;
  const step = animations[currentStep];
  if (step.type === 'swap') {
    [heap[step.i], heap[step.j]] = [heap[step.j], heap[step.i]];
  }
  currentStep++;
  drawHeap(step);
}

function stepBackward() {
  if (currentStep <= 0) return;
  currentStep--;
  const step = animations[currentStep];
  if (step.type === 'swap') {
    [heap[step.i], heap[step.j]] = [heap[step.j], heap[step.i]];
  }
  drawHeap(step);
}

function layoutHeap() {
  const positions = [];
  const levelHeight = 70;
  for (let i = 0; i < heap.length; i++) {
    const level = Math.floor(Math.log2(i + 1));
    const indexInLevel = i - (2 ** level - 1);
    const nodes = 2 ** level;
    const spacing = canvas.width / (nodes + 1);
    const x = spacing * (indexInLevel + 1);
    const y = 50 + level * levelHeight;
    positions.push({ x, y });
  }
  return positions;
}

function drawHeap(highlight = null) {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  const pos = layoutHeap();

  // Lines
  for (let i = 1; i < heap.length; i++) {
    const parent = Math.floor((i - 1) / 2);
    ctx.beginPath();
```

```
    ctx.moveTo(pos[i].x, pos[i].y);
    ctx.lineTo(pos[parent].x, pos[parent].y);
    ctx.strokeStyle = "#aaa";
    ctx.stroke();
  }

  // Nodes
  for (let i = 0; i < heap.length; i++) {
    ctx.beginPath();
    ctx.arc(pos[i].x, pos[i].y, 20, 0, 2 * Math.PI);
    ctx.fillStyle = (highlight && (highlight.i === i || highlight.j === i)) ? "#f39c12" : "#3498db";
    ctx.fill();
    ctx.strokeStyle = "#000";
    ctx.stroke();
    ctx.fillStyle = "#fff";
    ctx.font = "16px Arial";
    ctx.textAlign = "center";
    ctx.textBaseline = "middle";
    ctx.fillText(heap[i], pos[i].x, pos[i].y);
  }
  document.getElementById("output").textContent = `Heap Array: [${heap.join(', ')}]`;
}

</script>

</body>
</html>
```

11.1.1 Complete Binary Tree Structure

A **complete binary tree** ensures the tree is balanced and dense. This property allows heaps to be efficiently implemented using arrays, where:

- The root is at index 0.
- For a node at index `i`:
  - Left child is at `2i + 1`
  - Right child is at `2i + 2`
  - Parent is at `Math.floor((i - 1) / 2)`

11.1.2 Min-Heap vs Max-Heap

- **Min-Heap**: The value of each node is **less than or equal to** its children. The smallest element is always at the root.

  Use case: Efficiently retrieving the minimum element, like scheduling tasks with the earliest deadline.

- **Max-Heap**: The value of each node is **greater than or equal to** its children. The largest element is always at the root.

  Use case: Implementing priority queues where the highest priority element is needed quickly.

11.1.3 Heap Property (Heap Invariant)

The heap property is essential for priority access:

- In a **min-heap**, every parent node is smaller than or equal to its children.
- In a **max-heap**, every parent node is greater than or equal to its children.

This property guarantees that accessing the root gives the **minimum** or **maximum** element in constant time, `O(1)`.

11.1.4 Core Heap Operations

The two fundamental operations for heaps are:

- **Insertion**: Add a new element to the bottom of the tree (end of the array), then "bubble it up" to restore heap order.
- **Extraction (removeMin or removeMax)**: Remove the root element and replace it with the last element, then "bubble it down" (heapify) to restore heap order.

11.1.5 JavaScript Min-Heap Example

```
class MinHeap {
  constructor() {
    this.heap = [];
  }

  getParentIndex(i) {
    return Math.floor((i - 1) / 2);
  }

  getLeftChildIndex(i) {
    return 2 * i + 1;
  }

  getRightChildIndex(i) {
    return 2 * i + 2;
  }
```

```
  swap(i, j) {
    [this.heap[i], this.heap[j]] = [this.heap[j], this.heap[i]];
  }

  insert(value) {
    this.heap.push(value);
    this.bubbleUp();
  }

  bubbleUp() {
    let index = this.heap.length - 1;
    while (
      index > 0 &&
      this.heap[index] < this.heap[this.getParentIndex(index)]
    ) {
      this.swap(index, this.getParentIndex(index));
      index = this.getParentIndex(index);
    }
  }

  extractMin() {
    if (this.heap.length === 0) return null;
    if (this.heap.length === 1) return this.heap.pop();

    const min = this.heap[0];
    this.heap[0] = this.heap.pop();
    this.bubbleDown();
    return min;
  }

  bubbleDown() {
    let index = 0;
    const length = this.heap.length;

    while (true) {
      const left = this.getLeftChildIndex(index);
      const right = this.getRightChildIndex(index);
      let smallest = index;

      if (left < length && this.heap[left] < this.heap[smallest]) {
        smallest = left;
      }
      if (right < length && this.heap[right] < this.heap[smallest]) {
        smallest = right;
      }
      if (smallest === index) break;

      this.swap(index, smallest);
      index = smallest;
    }
  }
}
```

11.1.6 Visualizing Min-Heap Operations

Suppose we insert the values `[10, 15, 20, 17, 25]` step-by-step:

1. Insert 10 → Heap: `[10]`
2. Insert 15 → Heap: `[10, 15]` (no swaps)
3. Insert 20 → Heap: `[10, 15, 20]` (no swaps)
4. Insert 17 → Heap: `[10, 15, 20, 17]` (no swaps)
5. Insert 25 → Heap: `[10, 15, 20, 17, 25]` (no swaps)

Now extract the minimum:

- Remove root (10), replace with last (25) → `[25, 15, 20, 17]`
- Bubble down swaps 25 with 15 → `[15, 25, 20, 17]`
- Bubble down swaps 25 with 17 → `[15, 17, 20, 25]`

11.1.7 Why Use Heaps?

- **Fast access to min or max**: Root is always the min (min-heap) or max (max-heap).
- **Efficient insertions and deletions**: Both operations take `O(log n)` due to tree height.
- **Applications**: Priority queues, scheduling, graph algorithms (Dijkstra's shortest path), and heap sort.

11.1.8 Summary

Heaps are complete binary trees that maintain a heap order—either min-heap or max-heap—enabling efficient priority access. Implementing heaps with arrays leverages their completeness for easy parent-child index calculations. The core operations of insertion and extraction rely on bubbling elements up or down to maintain the heap invariant. Understanding heaps lays the groundwork for efficient priority queue implementations and sorting algorithms like heap sort.

11.2 Heap Sort

Heap sort is an efficient, comparison-based sorting algorithm that uses a **binary heap** to organize and sort elements. It's particularly useful when **in-place sorting** is required and **consistent time complexity** is desired across best, average, and worst cases.

Heap sort works in **two phases**:

1. **Build a max-heap** (or min-heap) from the input array.
2. **Repeatedly extract** the root (maximum or minimum), swap it to the end, and re-heapify the remaining portion.

11.2.1 Phase 1: Build the Heap

A **max-heap** ensures the largest value is at the root. This is ideal when sorting in ascending order—by repeatedly removing the max, we can construct the sorted array from end to start.

We start heapifying from the last non-leaf node and move upwards, ensuring each subtree satisfies the heap property.

11.2.2 Phase 2: Extract Elements and Heapify

Once the heap is built:

- Swap the root with the last item in the heap.
- Reduce the heap size by 1.
- Re-heapify the root to restore heap order.

Repeat until the heap is empty.

Run the following code in browser to see the demo:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Heap Sort Visualization</title>
  <style>
    body {
      font-family: Arial;
      text-align: center;
      padding: 20px;
    }
    canvas {
      border: 1px solid #ccc;
      background: #f8f8f8;
      margin-top: 10px;
      width:100%;
      height:400px;
    }
    input, button {
      font-size: 16px;
      margin: 5px;
      padding: 5px 10px;
    }
    #arrays {
```

```
      display: flex;
      justify-content: space-around;
      font-size: 18px;
      margin-top: 10px;
    }
    #arrays div {
      width: 30%;
    }
  </style>
</head>
<body>
<h2>Heap Sort Visualization</h2>
<input type="text" id="arrayInput" placeholder="e.g. 5,3,8,1,6">
<button onclick="startHeapSort()">Start Sort</button>
<canvas id="canvas" height="400"></canvas>
<div id="arrays">
  <div><strong>Original Array:</strong><br><span id="original"></span></div>
  <div><strong>Heap:</strong><br><span id="heap"></span></div>
  <div><strong>Sorted Result:</strong><br><span id="sorted"></span></div>
</div>
<script>
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");
let original = [];
let heap = [];
let sorted = [];
let steps = [];
let stepIndex = 0;

function startHeapSort() {
  const input = document.getElementById("arrayInput").value;
  original = input.split(',').map(x => parseInt(x.trim())).filter(x => !isNaN(x));
  heap = [...original];
  sorted = [];
  buildHeap(heap);
  steps = createSortSteps([...heap]);
  stepIndex = 0;
  draw();
  runSteps();
}

function buildHeap(arr) {
  for (let i = Math.floor(arr.length / 2) - 1; i >= 0; i--) {
    heapify(arr, i, arr.length);
  }
}

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

```
function createSortSteps(arr) {
  const steps = [];
  for (let i = arr.length - 1; i >= 0; i--) {
    steps.push({ type: 'swap', i: 0, j: i });
    [arr[0], arr[i]] = [arr[i], arr[0]];
    steps.push({ type: 'heapify', stop: i });
    heapify(arr, 0, i);
    steps.push({ type: 'recordSorted', val: arr[i] });
  }
  return steps;
}

function runSteps() {
  if (stepIndex >= steps.length) return;
  const step = steps[stepIndex];
  if (step.type === 'swap') {
    [heap[step.i], heap[step.j]] = [heap[step.j], heap[step.i]];
  } else if (step.type === 'heapify') {
    // Redraw after heapify
  } else if (step.type === 'recordSorted') {
    sorted.unshift(step.val);
  }
  draw();
  stepIndex++;
  setTimeout(runSteps, 700);
}

function layoutHeap(arr) {
  const positions = [];
  const levelHeight = 70;
  for (let i = 0; i < arr.length; i++) {
    const level = Math.floor(Math.log2(i + 1));
    const indexInLevel = i - (2 ** level - 1);
    const nodes = 2 ** level;
    const spacing = canvas.width / (nodes + 1);
    const x = spacing * (indexInLevel + 1);
    const y = 50 + level * levelHeight;
    positions.push({ x, y });
  }
  return positions;
}

function draw() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  const pos = layoutHeap(heap);
  for (let i = 1; i < heap.length; i++) {
    const parent = Math.floor((i - 1) / 2);
    if (parent >= 0 && pos[i] && pos[parent]) {
      ctx.beginPath();
      ctx.moveTo(pos[i].x, pos[i].y);
      ctx.lineTo(pos[parent].x, pos[parent].y);
      ctx.strokeStyle = "#aaa";
      ctx.stroke();
    }
  }
  for (let i = 0; i < heap.length; i++) {
    ctx.beginPath();
    ctx.arc(pos[i].x, pos[i].y, 20, 0, Math.PI * 2);
```

```
    ctx.fillStyle = "#3498db";
    ctx.fill();
    ctx.stroke();
    ctx.fillStyle = "#fff";
    ctx.font = "16px Arial";
    ctx.textAlign = "center";
    ctx.textBaseline = "middle";
    ctx.fillText(heap[i], pos[i].x, pos[i].y);
  }
  document.getElementById("original").textContent = original.join(", ");
  document.getElementById("heap").textContent = heap.join(", ");
  document.getElementById("sorted").textContent = sorted.join(", ");
}
</script>
</body>
</html>
```

11.2.3 Why It Works

By always moving the **largest remaining value** to the end of the array, heap sort gradually builds a sorted section at the array's tail, **in-place**, without using additional memory.

11.2.4 JavaScript Implementation

Full runnable code:

```
function heapSort(arr) {
  const n = arr.length;

  // Build max heap
  for (let i = Math.floor(n / 2) - 1; i >= 0; i--) {
    heapify(arr, n, i);
  }

  // Extract elements from heap one by one
  for (let i = n - 1; i > 0; i--) {
    // Move current root to end
    [arr[0], arr[i]] = [arr[i], arr[0]];

    // Heapify the reduced heap
    heapify(arr, i, 0);
  }

  return arr;
}

// Heapify a subtree rooted at index i
function heapify(arr, heapSize, i) {
  let largest = i;
```

```
  const left = 2 * i + 1;
  const right = 2 * i + 2;

  // If left child is larger
  if (left < heapSize && arr[left] > arr[largest]) {
    largest = left;
  }

  // If right child is larger
  if (right < heapSize && arr[right] > arr[largest]) {
    largest = right;
  }

  // If root is not largest, swap and continue
  if (largest !== i) {
    [arr[i], arr[largest]] = [arr[largest], arr[i]];
    heapify(arr, heapSize, largest);
  }
}
let a= [4, 10, 3, 5, 1];
heapSort(a);
console.log(a);
// Output: [1, 3, 4, 5, 10]
```

Step-by-step visualization:

1. **Build max-heap** from `[4, 10, 3, 5, 1]` → `[10, 5, 3, 4, 1]`
2. **Extract max (10)**, move to end → `[1, 5, 3, 4, 10]` Heapify → `[5, 4, 3, 1, 10]`
3. **Extract max (5)**, move to end → `[1, 4, 3, 5, 10]` Heapify → `[4, 1, 3, 5, 10]`
4. **Continue** until fully sorted → `[1, 3, 4, 5, 10]`

11.2.5 Time and Space Complexity

Operation	Complexity
Build heap	O(n)
Heapify (per op)	O(log n)
Total sort	**O(n log n)**

- **Space**: O(1) (in-place)
- **Stable**: NO No (relative order of equal elements is not preserved)

11.2.6 Pros and Cons

Advantages:

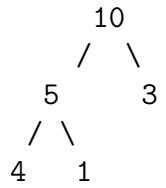
- Consistent O(n log n) performance
- No additional memory required (in-place)
- Good choice for large arrays when stability isn't required

Disadvantages:

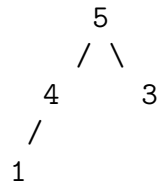
- Not stable
- More data movement than merge sort
- Generally slower than quicksort for small arrays due to larger constant factors

11.2.7 Visual Aid: Max-Heap Tree for `[4, 10, 3, 5, 1]`

Initial array:



After extracting root `10` and heapifying:



Sorted portion builds from right to left in the array.

11.2.8 Tip: Reversing for Descending Order

To sort in **descending order**, build a **min-heap** instead and extract the smallest repeatedly.

11.2.9 Summary

Heap sort is a powerful and predictable sorting algorithm based on the heap data structure. By building a max-heap and extracting elements in-place, it provides consistent `O(n log`

n) performance. Though not stable, its memory efficiency and structured nature make it valuable in scenarios where space and predictability matter more than speed or stability.

11.3 Implementing a Priority Queue

A **priority queue** is an abstract data structure similar to a regular queue, but with one major difference: **each element has a priority**, and elements are dequeued based on **priority order**, not arrival order.

Unlike a standard queue (FIFO), where the first item in is the first out, a **priority queue dequeues the item with the highest priority** (or lowest, depending on the implementation).

11.3.1 Common Use Cases

Priority queues are widely used in:

- **Task scheduling** (e.g., OS process management)
- **Event-driven systems** (e.g., simulations)
- **Shortest path algorithms**, like Dijkstra's
- **Job queues** in backend systems
- **Autocomplete systems**, based on frequency scores

11.3.2 How Heaps Make This Efficient

A **binary heap** is the ideal structure for implementing a priority queue because:

- Insertion: O(log n)
- Removal of the highest-priority element (usually the root): O(log n)
- Accessing the highest priority element (peek): O(1)

We use a **min-heap** to extract the element with the **lowest numerical priority value** (e.g., priority 1 is higher than 3). You could flip the comparison to make it a max-priority queue instead.

11.3.3 JavaScript Implementation

Let's implement a **min-priority queue**, where lower numbers mean higher priority.

Full runnable code:

```
class PriorityQueue {
  constructor() {
    this.heap = [];
  }

  // Helper functions to navigate the heap
  getParent(i) {
    return Math.floor((i - 1) / 2);
  }

  getLeft(i) {
    return 2 * i + 1;
  }

  getRight(i) {
    return 2 * i + 2;
  }

  swap(i, j) {
    [this.heap[i], this.heap[j]] = [this.heap[j], this.heap[i]];
  }

  // Insert an element with a priority
  enqueue(value, priority) {
    const node = { value, priority };
    this.heap.push(node);
    this.bubbleUp();
  }

  // Move new node up to maintain heap order
  bubbleUp() {
    let index = this.heap.length - 1;
    while (
      index > 0 &&
      this.heap[index].priority < this.heap[this.getParent(index)].priority
    ) {
      this.swap(index, this.getParent(index));
      index = this.getParent(index);
    }
  }

  // Remove and return the highest priority item
  dequeue() {
    if (this.heap.length === 0) return null;
    if (this.heap.length === 1) return this.heap.pop();

    const root = this.heap[0];
    this.heap[0] = this.heap.pop();
    this.bubbleDown();
    return root;
  }

  // Re-heapify the root
  bubbleDown() {
    let index = 0;
    const length = this.heap.length;
```

```
    while (true) {
      const left = this.getLeft(index);
      const right = this.getRight(index);
      let smallest = index;

      if (
        left < length &&
        this.heap[left].priority < this.heap[smallest].priority
      ) {
        smallest = left;
      }
      if (
        right < length &&
        this.heap[right].priority < this.heap[smallest].priority
      ) {
        smallest = right;
      }
      if (smallest === index) break;

      this.swap(index, smallest);
      index = smallest;
    }
  }

  // Peek at the highest-priority item without removing it
  peek() {
    return this.heap.length > 0 ? this.heap[0] : null;
  }

  isEmpty() {
    return this.heap.length === 0;
  }
}

const pq = new PriorityQueue();

pq.enqueue("Low priority task", 5);
pq.enqueue("Urgent bug fix", 1);
pq.enqueue("Feature development", 3);

console.log(JSON.stringify(pq.peek(),null,2));
// { value: 'Urgent bug fix', priority: 1 }

console.log(JSON.stringify(pq.dequeue(),null,2));
// { value: 'Urgent bug fix', priority: 1 }

console.log(JSON.stringify(pq.dequeue(),null,2));
// { value: 'Feature development', priority: 3 }
```

Each task is executed in order of **ascending priority value**, not insertion order.

11.3.4 Time Complexity Summary

Operation	Time Complexity
`enqueue`	O(log n)
`dequeue`	O(log n)
`peek`	O(1)

These performance guarantees come from the underlying heap structure.

11.3.5 Summary

Priority queues are crucial in many algorithmic and real-time systems where urgency dictates order. A **binary heap** makes them efficient by enabling quick access to the highest-priority item and maintaining order with minimal overhead. By implementing a priority queue from scratch in JavaScript, you not only gain insight into data structures but also prepare for scenarios involving efficient task handling, scheduling, and graph algorithms.

Chapter 12.

Balanced Search Trees

1. Red-Black Trees (Conceptual Overview)
2. AVL Trees (Optional in JavaScript)
3. B-Trees and Their Real-World Relevance

12 Balanced Search Trees

12.1 Red-Black Trees (Conceptual Overview)

A **Red-Black Tree** is a type of **self-balancing binary search tree (BST)** that ensures operations like insertion, deletion, and search happen in $\mathbf{O(\log n)}$ time, even in the worst case. This performance guarantee is achieved by maintaining a set of **coloring and structural rules** that keep the tree's height roughly logarithmic in the number of nodes.

12.1.1 What Makes a Tree "Red-Black"?

Each node in a red-black tree has an additional attribute: a **color**, either **red** or **black**. The tree follows these **five rules**, known as the **red-black properties**, to stay balanced:

1. **Each node is either red or black.**
2. **The root is always black.**
3. **All leaves (null nodes) are considered black.**
4. **Red nodes cannot have red children** (no two red nodes can appear consecutively).
5. **Every path from a node to its descendant null leaves must contain the same number of black nodes** (called the **black height**).

These rules restrict how unbalanced the tree can become, without enforcing perfect balance. The beauty of red-black trees is that they allow *some* imbalance to avoid constant restructuring while still guaranteeing efficient operations.

12.1.2 Rotations and Recoloring

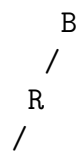
To maintain the red-black properties during **insertions** or **deletions**, the tree may perform:

- **Rotations** (left or right): These rearrange nodes locally while preserving BST order.
- **Recoloring**: The colors of nodes are changed to fix any violations of the rules above.

Let's walk through a simplified case.

Insertion Example (High-Level View)

Suppose we insert a red node that causes **two consecutive red nodes** (a violation of property #4):



R <- problem: two reds in a row

To fix this:

- If the **uncle** node is red, we can **recolor** (flip colors).
- If the uncle is black, we may need to **rotate** and recolor.

After a left or right rotation and recoloring, the tree maintains its structure and returns to a balanced state.

Run the following code in browser to see the demo:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Red-Black Tree Visualization</title>
  <style>
    body {
      font-family: sans-serif;
      text-align: center;
      padding: 20px;
    }
    canvas {
      border: 1px solid #ccc;
      background: #f4f4f4;
    }
    input, button {
      padding: 6px 10px;
      margin: 5px;
      font-size: 16px;
    }
  </style>
</head>
<body>
  <h2>Red-Black Tree Visualizer</h2>
  <input type="number" id="valueInput" placeholder="Enter number">
  <button onclick="insertValue()">Insert</button>
  <canvas id="canvas" width="600" height="500"></canvas>

  <script>
    const canvas = document.getElementById('canvas');
    const ctx = canvas.getContext('2d');

    const RED = 'RED';
    const BLACK = 'BLACK';

    class Node {
      constructor(value, color = RED, parent = null) {
        this.value = value;
        this.color = color;
        this.parent = parent;
        this.left = null;
        this.right = null;
        this.x = 0;
        this.y = 0;
      }
```

```
}

let root = null;

function insertValue() {
  const val = parseInt(document.getElementById('valueInput').value);
  if (isNaN(val)) return;
  root = insert(root, val);
  fixTree(root, val);
  layoutTree(root);
  drawTree();
  document.getElementById('valueInput').value = '';
}

function insert(node, value, parent = null) {
  if (!node) return new Node(value, RED, parent);
  if (value < node.value) {
    node.left = insert(node.left, value, node);
  } else if (value > node.value) {
    node.right = insert(node.right, value, node);
  }
  return node;
}

function fixTree(node, value) {
  let inserted = findNode(node, value);
  while (inserted !== root && inserted.parent.color === RED) {
    let parent = inserted.parent;
    let grandparent = parent.parent;
    if (!grandparent) break;

    if (parent === grandparent.left) {
      let uncle = grandparent.right;
      if (uncle && uncle.color === RED) {
        parent.color = BLACK;
        uncle.color = BLACK;
        grandparent.color = RED;
        inserted = grandparent;
      } else {
        if (inserted === parent.right) {
          inserted = parent;
          rotateLeft(inserted);
        }
        parent.color = BLACK;
        grandparent.color = RED;
        rotateRight(grandparent);
      }
    } else {
      let uncle = grandparent.left;
      if (uncle && uncle.color === RED) {
        parent.color = BLACK;
        uncle.color = BLACK;
        grandparent.color = RED;
        inserted = grandparent;
      } else {
        if (inserted === parent.left) {
          inserted = parent;
          rotateRight(inserted);
```

```
        }
        parent.color = BLACK;
        grandparent.color = RED;
        rotateLeft(grandparent);
      }
    }
  }
  root.color = BLACK;
}

function rotateLeft(x) {
  const y = x.right;
  x.right = y.left;
  if (y.left) y.left.parent = x;
  y.parent = x.parent;
  if (!x.parent) root = y;
  else if (x === x.parent.left) x.parent.left = y;
  else x.parent.right = y;
  y.left = x;
  x.parent = y;
}

function rotateRight(x) {
  const y = x.left;
  x.left = y.right;
  if (y.right) y.right.parent = x;
  y.parent = x.parent;
  if (!x.parent) root = y;
  else if (x === x.parent.right) x.parent.right = y;
  else x.parent.left = y;
  y.right = x;
  x.parent = y;
}

function findNode(node, value) {
  if (!node || node.value === value) return node;
  return value < node.value ? findNode(node.left, value) : findNode(node.right, value);
}

function layoutTree(node, depth = 0, range = [50, canvas.width - 50]) {
  if (!node) return;
  const [start, end] = range;
  const mid = (start + end) / 2;
  node.x = mid;
  node.y = 50 + depth * 70;
  layoutTree(node.left, depth + 1, [start, mid - 20]);
  layoutTree(node.right, depth + 1, [mid + 20, end]);
}

function drawTree() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  drawEdges(root);
  drawNodes(root);
}

function drawEdges(node) {
  if (!node) return;
  if (node.left) {
```

```
        ctx.beginPath();
        ctx.moveTo(node.x, node.y);
        ctx.lineTo(node.left.x, node.left.y);
        ctx.stroke();
        drawEdges(node.left);
      }
      if (node.right) {
        ctx.beginPath();
        ctx.moveTo(node.x, node.y);
        ctx.lineTo(node.right.x, node.right.y);
        ctx.stroke();
        drawEdges(node.right);
      }
    }

    function drawNodes(node) {
      if (!node) return;
      ctx.beginPath();
      ctx.arc(node.x, node.y, 20, 0, 2 * Math.PI);
      ctx.fillStyle = node.color === RED ? '#e74c3c' : '#2c3e50';
      ctx.fill();
      ctx.stroke();

      ctx.fillStyle = '#fff';
      ctx.textAlign = 'center';
      ctx.textBaseline = 'middle';
      ctx.font = '16px Arial';
      ctx.fillText(node.value, node.x, node.y);

      drawNodes(node.left);
      drawNodes(node.right);
    }
  </script>
</body>
</html>
```

12.1.3 Visual Representation

Here's a visual sketch of how a left rotation works:

```
Tree:
    A                    B
     \      →           / \
      B                A   C
       \
        C
```

A rotation moves B up, A down to the left, and keeps C on the right, preserving BST order while helping rebalance.

12.1.4 Why Red-Black Trees?

Red-black trees don't aim for perfect balance but instead enforce rules that **prevent worst-case degeneracy**, such as descending into a linked list (as can happen with naive BSTs).

They guarantee:

- **Height** $\leq 2 \times \log(n)$ → keeps operations fast
- **Insert/Delete in O(log n)** time consistently

12.1.5 Real-World Usage

Due to their efficient and predictable performance, red-black trees are commonly used in:

- **JavaScript's V8 engine**: for `Map` and `Set` under the hood
- **Java's `TreeMap` and `TreeSet`**
- **C++ STL's `std::map` and `std::set`**
- Database indexing systems (where B-trees may not be ideal)

They're favored for their **performance guarantees** without excessive overhead.

12.1.6 Summary

Red-black trees offer a smart compromise between performance and complexity. By enforcing simple coloring and structural rules, they ensure that binary search trees remain efficient under all conditions. While their internal mechanics can be intricate, their **conceptual model**—balanced through **rotations and recoloring**—makes them a practical foundation for many built-in and custom data structures in JavaScript and beyond.

12.2 AVL Trees (Optional in JavaScript)

AVL trees are one of the earliest types of **self-balancing binary search trees**, named after their inventors Adelson-Velsky and Landis. Like red-black trees, AVL trees maintain balance to ensure **O(log n)** time complexity for search, insertion, and deletion. However, AVL trees maintain a **stricter balance**, which leads to more frequent rebalancing operations.

12.2.1 Balance Factor and Rebalancing

The core concept of an AVL tree is the **balance factor** of a node, defined as:

```
balanceFactor = height(left subtree) - height(right subtree)
```

Valid balance factors for all nodes are `-1`, `0`, or `1`. If a node's balance factor becomes less than `-1` or greater than `1`, the tree becomes unbalanced and requires **rotation** to restore balance.

There are **four cases** of imbalance:

1. **Left-Left (LL)** → Right rotation
2. **Right-Right (RR)** → Left rotation
3. **Left-Right (LR)** → Left-Right double rotation
4. **Right-Left (RL)** → Right-Left double rotation

Run the following code in browser to see the demo:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>AVL Tree Visualization</title>
  <style>
    body { font-family: sans-serif; text-align: center; }
    canvas { border: 1px solid #ccc; background: #f9f9f9; }
    input, button { font-size: 16px; margin: 5px; padding: 5px 10px; }
  </style>
</head>
<body>
  <h2>AVL Tree Visualizer</h2>
  <input type="number" id="valueInput" placeholder="Enter number">
  <button onclick="insertValue()">Insert</button>
  <canvas id="canvas" width="600" height="500"></canvas>

  <script>
    const canvas = document.getElementById('canvas');
    const ctx = canvas.getContext('2d');

    class AVLNode {
      constructor(value) {
        this.value = value;
        this.left = null;
        this.right = null;
        this.height = 1;
        this.x = 0;
        this.y = 0;
      }
    }

    let root = null;

    function height(node) {
      return node ? node.height : 0;
    }
```

```
function updateHeight(node) {
  node.height = 1 + Math.max(height(node.left), height(node.right));
}

function getBalance(node) {
  return node ? height(node.left) - height(node.right) : 0;
}

function rotateRight(y) {
  const x = y.left;
  const T2 = x.right;
  x.right = y;
  y.left = T2;
  updateHeight(y);
  updateHeight(x);
  return x;
}

function rotateLeft(x) {
  const y = x.right;
  const T2 = y.left;
  y.left = x;
  x.right = T2;
  updateHeight(x);
  updateHeight(y);
  return y;
}

function insert(node, value) {
  if (!node) return new AVLNode(value);
  if (value < node.value) node.left = insert(node.left, value);
  else if (value > node.value) node.right = insert(node.right, value);
  else return node;

  updateHeight(node);
  const balance = getBalance(node);

  if (balance > 1 && value < node.left.value) return rotateRight(node);
  if (balance < -1 && value > node.right.value) return rotateLeft(node);
  if (balance > 1 && value > node.left.value) {
    node.left = rotateLeft(node.left);
    return rotateRight(node);
  }
  if (balance < -1 && value < node.right.value) {
    node.right = rotateRight(node.right);
    return rotateLeft(node);
  }

  return node;
}

function insertValue() {
  const val = parseInt(document.getElementById('valueInput').value);
  if (isNaN(val)) return;
  root = insert(root, val);
  layoutTree(root);
  drawTree();
  document.getElementById('valueInput').value = '';
```

```
    }

    function layoutTree(node, depth = 0, range = [50, canvas.width - 50]) {
      if (!node) return;
      const [start, end] = range;
      const mid = (start + end) / 2;
      node.x = mid;
      node.y = 50 + depth * 70;
      layoutTree(node.left, depth + 1, [start, mid - 20]);
      layoutTree(node.right, depth + 1, [mid + 20, end]);
    }

    function drawTree() {
      ctx.clearRect(0, 0, canvas.width, canvas.height);
      drawEdges(root);
      drawNodes(root);
    }

    function drawEdges(node) {
      if (!node) return;
      if (node.left) {
        ctx.beginPath();
        ctx.moveTo(node.x, node.y);
        ctx.lineTo(node.left.x, node.left.y);
        ctx.stroke();
        drawEdges(node.left);
      }
      if (node.right) {
        ctx.beginPath();
        ctx.moveTo(node.x, node.y);
        ctx.lineTo(node.right.x, node.right.y);
        ctx.stroke();
        drawEdges(node.right);
      }
    }

    function drawNodes(node) {
      if (!node) return;
      ctx.beginPath();
      ctx.arc(node.x, node.y, 20, 0, 2 * Math.PI);
      ctx.fillStyle = '#2ecc71';
      ctx.fill();
      ctx.stroke();

      ctx.fillStyle = '#fff';
      ctx.textAlign = 'center';
      ctx.textBaseline = 'middle';
      ctx.font = '16px Arial';
      ctx.fillText(node.value, node.x, node.y);

      drawNodes(node.left);
      drawNodes(node.right);
    }
  </script>
</body>
</html>
```

12.2.2 Rotations in AVL Trees

Rotations are local tree restructuring operations that preserve the binary search property. Here's a simplified example of a **right rotation** for the LL case:

```
tree:

    C                   B
   /      →           / \
  B                 A   C
 /
A
```

After rotation, the subtree is rebalanced, and the height differences are corrected.

12.2.3 AVL vs. Red-Black Trees

Feature	AVL Tree	Red-Black Tree
Balance Strictness	More strictly balanced	Less strict (allows more imbalance)
Rebalancing Cost	More rotations	Fewer rotations
Lookup Performance	Faster (shallower trees)	Slightly slower
Insertion/Deletion	More costly (more rebalancing)	Cheaper (fewer adjustments)

AVL trees are ideal when you have more **read-heavy** workloads (frequent lookups), while **red-black trees** perform better in **write-heavy** environments (frequent inserts/deletes).

12.2.4 JavaScript-Style Pseudocode (Simplified)

A simplified AVL insertion pseudocode might look like this:

```
function insert(node, value) {
  if (!node) return new AVLNode(value);

  if (value < node.value) {
    node.left = insert(node.left, value);
  } else {
    node.right = insert(node.right, value);
  }

  updateHeight(node);
  const balance = getBalance(node);

  // Rotate if unbalanced
  if (balance > 1 && value < node.left.value) {
    return rotateRight(node); // Left-Left
```

```
  }
  if (balance < -1 && value > node.right.value) {
    return rotateLeft(node); // Right-Right
  }
  if (balance > 1 && value > node.left.value) {
    node.left = rotateLeft(node.left);
    return rotateRight(node); // Left-Right
  }
  if (balance < -1 && value < node.right.value) {
    node.right = rotateRight(node.right);
    return rotateLeft(node); // Right-Left
  }

  return node;
}
```

This captures the essence of AVL insertion: insert recursively, update height, check balance, and rotate if needed.

12.2.5 Why This Is Optional

In real-world JavaScript development, AVL trees are **rarely implemented manually**. Most JavaScript applications rely on native structures (`Map`, `Set`) or libraries that use red-black trees internally. AVL trees are more commonly studied for academic purposes or implemented in low-level systems (e.g., embedded, database internals).

That said, understanding AVL trees is a great way to **deepen your grasp of tree balancing** and rotations—useful if you're working on performance-critical or memory-sensitive systems.

12.2.6 Summary

AVL trees provide a stricter balancing approach than red-black trees by maintaining a tight control over subtree height differences. While more rotation-heavy, they offer slightly faster lookups in exchange. Due to their complexity and specialized use, AVL trees are considered an **advanced and optional topic** for most JavaScript developers, but they are foundational in understanding the broader world of balanced search trees.

12.3 B-Trees and Their Real-World Relevance

B-trees are a class of self-balancing, multi-way search trees designed specifically for **efficient storage and retrieval of large datasets**, especially on **disk-based systems**. Unlike binary search trees (BSTs), where each node has at most two children, B-trees can have

many children per node, dramatically reducing the tree's height and minimizing disk read/write operations. This makes them ideal for **databases**, **file systems**, and **persistent key-value stores**.

12.3.1 What Makes a B-Tree Different?

In a traditional BST, each node holds a single key and has at most two children. In contrast, a B-tree node:

- Can hold **multiple keys** (not just one).
- Has **multiple children**, typically ranging from *m/2 to m*, where m is the maximum number of children.
- Keeps keys **sorted**, so binary search can be used within a node.
- Always stays **balanced** — all leaf nodes are at the same depth.

Example: B-tree of Order 4

- Each node can have **up to 3 keys** and **4 children**.
- A node with keys `[10, 20, 30]` divides its children into ranges:

  ```
  [ <10 ] - [10-20] - [20-30] - [ >30 ]
  ```

This organization allows the tree to store **much more data per node**, reducing the number of levels and therefore the number of disk accesses.

12.3.2 Why B-Trees Matter for Storage

Disk and SSD operations are expensive compared to in-memory operations. B-trees optimize performance by:

- **Minimizing tree height**: Fewer levels mean fewer disk accesses.
- **Maximizing data per node**: Entire blocks (pages) of data are loaded into memory at once.
- **Reducing traversal time**: Because internal nodes can store many keys, a single node read can guide access to many elements.

Databases like **MySQL (InnoDB engine)**, **PostgreSQL**, and even **modern file systems** (e.g., NTFS, HFS+) use **B-trees or B+ trees** internally.

12.3.3 How It Works (Simplified)

Inserting into a B-tree involves:

1. Finding the correct leaf node.
2. Inserting the key into the node in sorted order.
3. If the node becomes overfull (too many keys), it **splits**, and the middle key is promoted to the parent.

This controlled promotion keeps the tree balanced without requiring complex rotations like in AVL or Red-Black trees.

Visual Example

```
Before insertion:
[20 | 40]

Children:
<20   20-40   >40

Inserting 25 → goes into 20-40 node

After split:
Parent becomes: [20 | 30 | 40]
```

Each split pushes keys up, maintaining a balanced and shallow structure.

12.3.4 B-Trees in JavaScript?

B-trees are **not commonly used in in-memory JavaScript applications** because:

- JavaScript apps typically handle smaller datasets that fit comfortably in memory.
- Native structures like `Map` or `Set` (often backed by Red-Black trees) are optimized for general-purpose use.

However, B-trees become relevant in:

- **Browser databases** like IndexedDB or LevelDB.
- **JavaScript-based databases** (e.g., PouchDB, NeDB).
- Applications involving **large local files**, **offline data syncing**, or **custom database engines**.

12.3.5 Summary

B-trees are a foundational structure for building efficient, scalable storage systems. Their ability to store multiple keys per node and minimize tree height makes them perfect for **disk-based or paged-memory environments**, such as **databases and file systems**. While not often implemented manually in JavaScript, understanding B-trees reveals why certain database operations are fast and helps prepare you for backend or systems-level development where performance and scale matter most.

Chapter 13.

Graph Representations

1. Adjacency Lists and Matrices in JS
2. Directed vs Undirected Graphs
3. Weighted vs Unweighted Graphs

13 Graph Representations

13.1 Adjacency Lists and Matrices in JS

Graphs are powerful data structures used to model relationships between entities—think social networks, maps, or dependency graphs. To work with graphs in JavaScript, we need a way to represent the nodes (vertices) and their connections (edges). The two most common representations are:

- **Adjacency List**
- **Adjacency Matrix**

Each has its own trade-offs in terms of memory usage and performance for different operations. Let's explore both.

Run the following code in browser to see the demo:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Graph Visualizer</title>
  <style>
    body {
      font-family: sans-serif;
      text-align: center;
      padding: 20px;
    }
    canvas {
      border: 1px solid #ccc;
      background: #f9f9f9;
      cursor: pointer;
    }
    input, button {
      font-size: 16px;
      padding: 6px;
      margin: 5px;
    }
  </style>
</head>
<body>
<h2>Graph Visualizer</h2>
<p>Click on canvas to add nodes. Click two nodes to add an edge between them.</p>
<canvas id="canvas" width="600" height="600"></canvas>
<script>
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');
let nodes = [];
let edges = [];
let selectedNode = null;

canvas.addEventListener('click', handleCanvasClick);

function handleCanvasClick(event) {
  const rect = canvas.getBoundingClientRect();
```

```
  const x = event.clientX - rect.left;
  const y = event.clientY - rect.top;
  const clickedNode = nodes.find(n => Math.hypot(n.x - x, n.y - y) < 20);
  if (clickedNode) {
    if (selectedNode && selectedNode !== clickedNode) {
      edges.push({ from: selectedNode, to: clickedNode });
      selectedNode = null;
    } else {
      selectedNode = clickedNode;
    }
  } else {
    nodes.push({ x, y, label: String.fromCharCode(65 + nodes.length) });
  }
  drawGraph();
}

function drawGraph() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  // Draw edges
  ctx.strokeStyle = '#333';
  ctx.lineWidth = 2;
  for (const edge of edges) {
    ctx.beginPath();
    ctx.moveTo(edge.from.x, edge.from.y);
    ctx.lineTo(edge.to.x, edge.to.y);
    ctx.stroke();
  }

  // Draw nodes
  for (const node of nodes) {
    ctx.beginPath();
    ctx.arc(node.x, node.y, 20, 0, 2 * Math.PI);
    ctx.fillStyle = node === selectedNode ? '#f39c12' : '#3498db';
    ctx.fill();
    ctx.strokeStyle = '#2c3e50';
    ctx.stroke();

    ctx.fillStyle = '#fff';
    ctx.font = '16px Arial';
    ctx.textAlign = 'center';
    ctx.textBaseline = 'middle';
    ctx.fillText(node.label, node.x, node.y);
  }
}
</script>
</body>
</html>
```

13.1.1 Adjacency List

An **adjacency list** stores each node along with a list (or set) of its neighbors. This is a memory-efficient way to represent **sparse graphs**, where most node pairs are not directly

connected.

JavaScript Example Directed Graph

```
class GraphList {
  constructor() {
    this.adjList = new Map();
  }

  addVertex(vertex) {
    if (!this.adjList.has(vertex)) {
      this.adjList.set(vertex, []);
    }
  }

  addEdge(src, dest) {
    this.addVertex(src);
    this.addVertex(dest);
    this.adjList.get(src).push(dest);
  }

  getNeighbors(vertex) {
    return this.adjList.get(vertex) || [];
  }
}
```

Undirected Graph

To make it undirected, add both directions:

```
addEdge(src, dest) {
  this.addVertex(src);
  this.addVertex(dest);
  this.adjList.get(src).push(dest);
  this.adjList.get(dest).push(src);
}
```

13.1.2 Adjacency Matrix

An **adjacency matrix** uses a 2D array to represent connections. The cell at `matrix[i][j]` is `1` (or `true`) if there's an edge from vertex `i` to vertex `j`, and `0` otherwise. This is great for **dense graphs** or when you need **constant-time edge lookups**.

JavaScript Example Directed Graph

```
class GraphMatrix {
  constructor(size) {
    this.size = size;
    this.matrix = Array.from({ length: size }, () => Array(size).fill(0));
  }

  addEdge(src, dest) {
    this.matrix[src][dest] = 1;
```

```
  }

  getNeighbors(vertex) {
    return this.matrix[vertex]
      .map((val, index) => (val ? index : -1))
      .filter(index => index !== -1);
  }
}
```

Undirected Graph

Make it symmetric:

```
addEdge(src, dest) {
  this.matrix[src][dest] = 1;
  this.matrix[dest][src] = 1;
}
```

13.1.3 Comparison: List vs Matrix

Feature	Adjacency List	Adjacency Matrix
Space Complexity	O(V + E)	O(V^2)
Edge Lookup	O(degree)	O(1)
Add Edge	O(1)	O(1)
Best For	Sparse graphs	Dense graphs
Iterating Neighbors	Fast	Requires filtering

- `V` = number of vertices
- `E` = number of edges

An adjacency list scales better when graphs are large but sparsely connected. Conversely, adjacency matrices are useful when the number of edges is close to the number of vertex pairs (e.g., in fully connected graphs).

13.1.4 Performance Implications for Algorithms

- **Depth-First Search (DFS)** and **Breadth-First Search (BFS)** typically use adjacency lists for efficiency.
- **Dijkstra's algorithm** benefits from adjacency lists with a priority queue, especially on sparse graphs.
- **Floyd-Warshall**, a shortest-path algorithm for all-pairs, is easier to implement with an adjacency matrix.

13.1.5 Summary

Choosing between an **adjacency list** and an **adjacency matrix** depends on your use case. For most real-world applications (like social graphs or road maps), adjacency lists offer better memory efficiency and flexibility. However, for dense graphs or algorithms needing quick edge access, adjacency matrices shine. Understanding both allows you to tailor your implementation to the specific needs of your application or algorithm.

13.2 Directed vs Undirected Graphs

In graph theory, the direction of edges plays a crucial role in how relationships are represented and how algorithms behave. Graphs are broadly classified as **directed** or **undirected**, depending on whether their edges have a direction.

Run the following code in browser to see the demo:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Directed & Undirected Graph Visualizer</title>
  <style>
    body { font-family: sans-serif; text-align: center; padding: 20px; }
    canvas { border: 1px solid #ccc; background: #f9f9f9; cursor: pointer; }
    input, button, select {
      font-size: 16px;
      padding: 6px;
      margin: 5px;
    }
  </style>
</head>
<body>
<h2>Directed vs Undirected Graph Visualizer</h2>
<p>Click to add nodes. Click two nodes to add an edge. Toggle graph type below.</p>
<select id="graphType">
  <option value="undirected">Undirected</option>
  <option value="directed">Directed</option>
</select>
<canvas id="canvas" width="600" height="600"></canvas>
<script>
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');
const graphTypeSelector = document.getElementById('graphType');
let graphType = graphTypeSelector.value;

let nodes = [];
let edges = [];
let selectedNode = null;

graphTypeSelector.addEventListener('change', () => {
  graphType = graphTypeSelector.value;
```

```
  drawGraph();
});

canvas.addEventListener('click', handleCanvasClick);

function handleCanvasClick(event) {
  const rect = canvas.getBoundingClientRect();
  const x = event.clientX - rect.left;
  const y = event.clientY - rect.top;
  const clickedNode = nodes.find(n => Math.hypot(n.x - x, n.y - y) < 20);
  if (clickedNode) {
    if (selectedNode && selectedNode !== clickedNode) {
      edges.push({ from: selectedNode, to: clickedNode });
      selectedNode = null;
    } else {
      selectedNode = clickedNode;
    }
  } else {
    nodes.push({ x, y, label: String.fromCharCode(65 + nodes.length) });
  }
  drawGraph();
}

function drawArrow(fromX, fromY, toX, toY, directed) {
  ctx.beginPath();
  ctx.moveTo(fromX, fromY);
  ctx.lineTo(toX, toY);
  ctx.stroke();

  if (directed) {
    const headlen = 10;
    const angle = Math.atan2(toY - fromY, toX - fromX);
    ctx.beginPath();
    ctx.moveTo(toX, toY);
    ctx.lineTo(toX - headlen * Math.cos(angle - Math.PI / 6), toY - headlen * Math.sin(angle - Math.PI /
    ctx.lineTo(toX - headlen * Math.cos(angle + Math.PI / 6), toY - headlen * Math.sin(angle + Math.PI /
    ctx.lineTo(toX, toY);
    ctx.fill();
  }
}

function drawGraph() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  ctx.strokeStyle = '#333';
  ctx.lineWidth = 2;
  ctx.fillStyle = '#333';

  for (const edge of edges) {
    const dx = edge.to.x - edge.from.x;
    const dy = edge.to.y - edge.from.y;
    const dist = Math.hypot(dx, dy);
    const unitX = dx / dist;
    const unitY = dy / dist;
    const fromX = edge.from.x + unitX * 20;
    const fromY = edge.from.y + unitY * 20;
    const toX = edge.to.x - unitX * 20;
    const toY = edge.to.y - unitY * 20;
```

```
    drawArrow(fromX, fromY, toX, toY, graphType === 'directed');
  }

  for (const node of nodes) {
    ctx.beginPath();
    ctx.arc(node.x, node.y, 20, 0, 2 * Math.PI);
    ctx.fillStyle = node === selectedNode ? '#f39c12' : '#3498db';
    ctx.fill();
    ctx.strokeStyle = '#2c3e50';
    ctx.stroke();

    ctx.fillStyle = '#fff';
    ctx.font = '16px Arial';
    ctx.textAlign = 'center';
    ctx.textBaseline = 'middle';
    ctx.fillText(node.label, node.x, node.y);
  }
}
</script>
</body>
</html>
```

13.2.1 Directed Graphs (Digraphs)

In a **directed graph**, each edge has a direction, pointing from one vertex to another. If there's an edge from node `A` to `B`, it means $\mathbf{A} \rightarrow \mathbf{B}$, but not necessarily $\mathbf{B} \rightarrow \mathbf{A}$.

JavaScript Representation

```
class DirectedGraph {
  constructor() {
    this.adjList = new Map();
  }

  addVertex(vertex) {
    if (!this.adjList.has(vertex)) {
      this.adjList.set(vertex, []);
    }
  }

  addEdge(from, to) {
    this.addVertex(from);
    this.addVertex(to);
    this.adjList.get(from).push(to);
  }
}
```

Use Cases

- **Social media (e.g., Twitter)**: A user can follow another without reciprocation.
- **Dependency graphs**: Task A must be completed before Task B.

- **Web crawling**: Links go one-way from one page to another.

Implications for Algorithms

Traversal (DFS, BFS) must respect edge direction. For example, if `A → B` but not `B → A`, starting from `B` won't reach `A`.

13.2.2 Undirected Graphs

In an **undirected graph**, edges have **no direction**. A connection between nodes `A` and `B` implies **A B**—the relationship is mutual.

JavaScript Representation

```
class UndirectedGraph {
  constructor() {
    this.adjList = new Map();
  }

  addVertex(vertex) {
    if (!this.adjList.has(vertex)) {
      this.adjList.set(vertex, []);
    }
  }

  addEdge(v1, v2) {
    this.addVertex(v1);
    this.addVertex(v2);
    this.adjList.get(v1).push(v2);
    this.adjList.get(v2).push(v1); // Bidirectional
  }
}
```

Use Cases

- **Road maps**: You can typically drive both directions between cities.
- **Friend networks (e.g., Facebook)**: A friendship goes both ways.
- **Network topologies**: Wired connections are often bidirectional.

Implications for Algorithms

Since edges are bidirectional, DFS and BFS can traverse in either direction, making connectivity checks (e.g., is everything connected?) straightforward.

13.2.3 Visual Example

- **Directed Graph**

```
A → B
B → C
```

Traversal from `A` can reach `C`, but `C` cannot reach `A`.

- **Undirected Graph**

```
A - B - C
```

You can go from any node to any other in both directions.

13.2.4 Key Differences

Feature	Directed Graph	Undirected Graph
Edge Direction	One-way (A → B)	Two-way (A B)
Representation	Single edge per direction	Two-way edge stored twice
Use Case Examples	Twitter, workflows	Road maps, mutual networks
Traversal Complexity	Direction-sensitive	Easier full traversal

13.2.5 Summary

Understanding the distinction between **directed** and **undirected** graphs is essential when designing algorithms or modeling data. Directed graphs are great for modeling **asymmetric relationships**, while undirected graphs fit **mutual or symmetric** interactions. Choosing the correct type of graph affects how you implement traversal, search, and even pathfinding algorithms in JavaScript.

13.3 Weighted vs Unweighted Graphs

Graphs can be **weighted** or **unweighted**, depending on whether their edges carry additional data, typically representing cost, distance, time, or capacity. Understanding the difference is essential for choosing the right algorithms and data structures when solving real-world problems.

13.3.1 What Is a Weighted Graph?

In a **weighted graph**, each edge has a numerical value (called a **weight**) that represents some cost associated with moving between two vertices.

Examples of Edge Weights:

- Distance between cities (in kilometers)
- Time to transmit a message (in milliseconds)
- Cost to perform an operation (in dollars)

In contrast, an **unweighted graph** treats all connections equally—each edge has the same cost (often implied to be 1).

13.3.2 Representation in JavaScript

Adjacency List (Weighted)

Instead of storing just a neighbor, we store both the neighbor and the weight:

```
class WeightedGraph {
  constructor() {
    this.adjList = new Map();
  }

  addVertex(vertex) {
    if (!this.adjList.has(vertex)) {
      this.adjList.set(vertex, []);
    }
  }

  addEdge(from, to, weight) {
    this.addVertex(from);
    this.addVertex(to);
    this.adjList.get(from).push({ node: to, weight });
  }
}
```

Example:

```
const graph = new WeightedGraph();
graph.addEdge("A", "B", 5);
graph.addEdge("A", "C", 2);
// A → B (5), A → C (2)
```

Adjacency Matrix (Weighted)

A 2D matrix can store weights directly:

```
const size = 3;
const matrix = Array.from({ length: size }, () => Array(size).fill(Infinity));
```

```
matrix[0][1] = 4; // weight from node 0 to node 1
matrix[1][2] = 7; // weight from node 1 to node 2
```

Using `Infinity` or `null` to indicate no connection helps differentiate between zero-weight edges and absent edges.

13.3.3 Unweighted Graphs

In unweighted graphs, edge values are all uniform (typically assumed to be 1), so you don't store weights:

```
adjList = {
  A: ["B", "C"],
  B: ["A", "D"]
};
```

Traversals like **Breadth-First Search (BFS)** are commonly used in unweighted graphs to find the shortest path by hop count.

13.3.4 Key Differences

Feature	Weighted Graph	Unweighted Graph
Edge Data	Includes cost/weight	No extra data (uniform cost)
Shortest Path	Requires Dijkstra's or A*	Can use BFS
Representation	Needs weight field or matrix values	Simple arrays or booleans
Common Use Cases	Routing, navigation, network cost	Social networks, reachability

13.3.5 Real-World Applications

- **Routing & Navigation**: Maps use edge weights to represent driving distance or time.
- **Network Optimization**: Weights can represent bandwidth, latency, or transfer cost.
- **Game AI**: Weighted graphs help find efficient paths through terrain.
- **Logistics**: Route planning, delivery optimization, and scheduling.

13.3.6 Algorithms That Use Weights

Weighted graphs change how you approach algorithms:

- **Dijkstra's Algorithm**: Finds shortest path considering edge weights.
- **Bellman-Ford**: Handles graphs with negative weights.
- **Prim's and Kruskal's Algorithms**: Find minimum spanning trees.

In contrast, **BFS** is sufficient for unweighted graphs when edge count is more important than total weight.

13.3.7 Summary

Understanding the difference between **weighted** and **unweighted** graphs helps you model problems accurately and choose efficient algorithms. In JavaScript, you can adapt both adjacency lists and matrices to store weights easily. Weighted graphs unlock powerful real-world applications like routing, cost analysis, and decision-making across various domains.

Chapter 14.

Graphs: Traversals

1. Breadth-First Search (BFS)
2. Depth-First Search (DFS)
3. Applications (Pathfinding, Web Crawling)

14 Graphs: Traversals

14.1 Breadth-First Search (BFS)

Breadth-First Search (BFS) is a fundamental graph traversal algorithm that explores nodes in a **level-order** manner. Starting from a given source node, it visits all immediate neighbors first, then their neighbors, and so on. This strategy makes BFS ideal for **shortest path discovery** in unweighted graphs and for exploring all nodes reachable from a source.

14.1.1 BFS Fundamentals

BFS works by systematically visiting nodes in the order they are discovered, maintaining a **queue** to track which nodes to visit next. This ensures that nodes are visited in the order of their **distance** (in edges) from the source node.

Algorithm Overview:

1. Start from a source node.
2. Enqueue it and mark it as visited.
3. While the queue is not empty:
    - Dequeue the front node.
    - Visit all unvisited neighbors and enqueue them.

Run the following code in browser to see the demo:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>BFS Graph Visualization</title>
  <style>
    body { font-family: sans-serif; text-align: center; padding: 20px; }
    canvas { border: 1px solid #ccc; background: #f9f9f9; }
    button { font-size: 16px; padding: 8px 12px; margin: 10px; }
  </style>
</head>
<body>
  <h2>Breadth-First Search (BFS) Visualization</h2>
  <button onclick="runBFS()">Run BFS</button>
  <canvas id="canvas" width="600" height="600"></canvas>

  <script>
    const canvas = document.getElementById("canvas");
    const ctx = canvas.getContext("2d");

    // Predefined nodes with fixed positions
    const nodes = [
```

```
    { label: 'A', x: 150, y: 100 },
    { label: 'B', x: 300, y: 100 },
    { label: 'C', x: 450, y: 100 },
    { label: 'D', x: 150, y: 250 },
    { label: 'E', x: 300, y: 250 },
    { label: 'F', x: 450, y: 250 },
    { label: 'G', x: 300, y: 400 }
  ];

  // Edges between nodes (undirected)
  const edges = [
    ['A', 'B'], ['A', 'D'], ['B', 'C'], ['B', 'E'],
    ['C', 'F'], ['D', 'E'], ['E', 'F'], ['E', 'G']
  ];

  // Build adjacency list
  const adjList = {};
  for (const node of nodes) {
    adjList[node.label] = [];
  }
  for (const [from, to] of edges) {
    adjList[from].push(to);
    adjList[to].push(from);
  }

  let bfsOrder = [];

  function runBFS() {
    bfsOrder = [];
    const visited = new Set();
    const queue = ['A'];
    visited.add('A');

    while (queue.length > 0) {
      const current = queue.shift();
      bfsOrder.push(current);
      for (const neighbor of adjList[current]) {
        if (!visited.has(neighbor)) {
          visited.add(neighbor);
          queue.push(neighbor);
        }
      }
    }
    animateBFS(0);
  }

  // Animate BFS traversal by highlighting visited nodes
  function animateBFS(index) {
    if (index >= bfsOrder.length) return;
    drawGraph(bfsOrder.slice(0, index + 1));
    setTimeout(() => animateBFS(index + 1), 800);
  }

  // Draw the graph with optional highlighted nodes
  function drawGraph(highlighted = []) {
    ctx.clearRect(0, 0, canvas.width, canvas.height);

    // Draw edges
```

```
      ctx.strokeStyle = "#999";
      for (const [from, to] of edges) {
        const fromNode = nodes.find(n => n.label === from);
        const toNode = nodes.find(n => n.label === to);
        ctx.beginPath();
        ctx.moveTo(fromNode.x, fromNode.y);
        ctx.lineTo(toNode.x, toNode.y);
        ctx.stroke();
      }

      // Draw nodes
      for (const node of nodes) {
        ctx.beginPath();
        ctx.arc(node.x, node.y, 25, 0, 2 * Math.PI);
        ctx.fillStyle = highlighted.includes(node.label) ? "#27ae60" : "#3498db"; // green when visited
        ctx.fill();
        ctx.stroke();

        ctx.fillStyle = "#fff";
        ctx.font = "16px Arial";
        ctx.textAlign = "center";
        ctx.textBaseline = "middle";
        ctx.fillText(node.label, node.x, node.y);
      }
    }

    drawGraph();
  </script>
</body>
</html>
```

14.1.2 JavaScript Implementation

Here's how you can implement BFS using an adjacency list representation of a graph:

Full runnable code:

```
class Graph {
  constructor() {
    this.adjList = new Map();
  }

  addVertex(vertex) {
    if (!this.adjList.has(vertex)) {
      this.adjList.set(vertex, []);
    }
  }

  addEdge(v1, v2) {
    this.addVertex(v1);
    this.addVertex(v2);
    this.adjList.get(v1).push(v2); // For undirected: also add v2 → v1
  }
```

```
  bfs(start) {
    const visited = new Set();
    const queue = [start];
    const result = [];

    visited.add(start);

    while (queue.length > 0) {
      const current = queue.shift();
      result.push(current);

      for (const neighbor of this.adjList.get(current)) {
        if (!visited.has(neighbor)) {
          visited.add(neighbor);
          queue.push(neighbor);
        }
      }
    }

    return result;
  }
}

const g = new Graph();
g.addEdge("A", "B");
g.addEdge("A", "C");
g.addEdge("B", "D");
g.addEdge("C", "E");

console.log(g.bfs("A")); // Output: [ 'A', 'B', 'C', 'D', 'E' ]
```

14.1.3 Traversal Order (Step-by-Step)

Given the graph:

```
   A
  / \
 B   C
 |    \
 D     E
```

Starting BFS from `"A"` would visit nodes in the following order:

1. Start with `A`, enqueue it → `[A]`
2. Visit `A`, enqueue `B`, `C` → `[B, C]`
3. Visit `B`, enqueue `D` → `[C, D]`
4. Visit `C`, enqueue `E` → `[D, E]`
5. Visit `D` → `[E]`
6. Visit `E` → `[]`

Result: `["A", "B", "C", "D", "E"]`

14.1.4 Time and Space Complexity

- **Time Complexity**: O(V + E) Each node and edge is visited at most once.
- **Space Complexity**: O(V) To store the queue and visited set.

14.1.5 Real-World Use Cases

- **Shortest Path in Unweighted Graphs**: BFS guarantees the shortest path (by edge count) from a source to all reachable nodes.
- **Social Network Analysis**: Discover degrees of separation between people (e.g., "friend of a friend").
- **Peer-to-Peer Networking (e.g., BitTorrent)**: Find closest peers efficiently.
- **Web Crawling**: Explore all reachable links from a seed URL level-by-level.
- **GPS/Mapping (when edge weights are uniform)**: Find the fastest path when all roads have equal weight.

14.1.6 BFS vs DFS (Quick Note)

While **DFS** dives deep before backtracking, **BFS** expands outward layer by layer. This makes BFS better for **finding the shortest path**, while DFS can be more efficient for searching deeply nested structures or for **cycle detection**.

14.1.7 Summary

Breadth-First Search is a cornerstone algorithm in graph theory and practical computing. Its use of a queue enables level-order traversal, which is especially powerful for solving **shortest path problems in unweighted graphs**. Whether you're building a web crawler, analyzing networks, or implementing a multiplayer game, mastering BFS in JavaScript gives you a solid foundation for solving real-world problems efficiently.

14.2 Depth-First Search (DFS)

Depth-First Search (DFS) is a graph traversal technique that explores as deeply as possible along a branch before backtracking. Unlike **Breadth-First Search (BFS)**, which explores neighbors level by level, DFS dives into one path and only reverses course when it hits a dead end. This nature of DFS makes it ideal for tasks that require full path exploration, such as **cycle detection**, **topological sorting**, and **solving puzzles or mazes**.

14.2.1 How DFS Works

DFS can be implemented in two ways:

- **Recursively**, using the call stack to manage traversal depth.
- **Iteratively**, using an explicit **stack** to simulate recursion.

DFS uses a "visited" set or map to avoid revisiting nodes and getting stuck in cycles.

Run the following code in browser to see the demo:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>DFS Graph Visualization</title>
  <style>
    body { font-family: sans-serif; text-align: center; padding: 20px; }
    canvas { border: 1px solid #ccc; background: #f9f9f9; }
    button { font-size: 16px; padding: 8px 12px; margin: 10px; }
  </style>
</head>
<body>
  <h2>Depth-First Search (DFS) Visualization</h2>
  <button onclick="runDFS()">Run DFS</button>
  <canvas id="canvas" width="600" height="600"></canvas>

  <script>
    const canvas = document.getElementById("canvas");
    const ctx = canvas.getContext("2d");

    const nodes = [
      { label: 'A', x: 150, y: 100 },
      { label: 'B', x: 300, y: 100 },
      { label: 'C', x: 450, y: 100 },
      { label: 'D', x: 150, y: 250 },
      { label: 'E', x: 300, y: 250 },
      { label: 'F', x: 450, y: 250 },
      { label: 'G', x: 300, y: 400 }
    ];

    const edges = [
      ['A', 'B'], ['A', 'D'], ['B', 'C'], ['B', 'E'],
      ['C', 'F'], ['D', 'E'], ['E', 'F'], ['E', 'G']
```

```
];

const adjList = {};
for (const node of nodes) {
  adjList[node.label] = [];
}
for (const [from, to] of edges) {
  adjList[from].push(to);
  adjList[to].push(from); // undirected graph
}

const visited = new Set();
let dfsOrder = [];

function runDFS() {
  visited.clear();
  dfsOrder = [];
  dfs('A');
  animateDFS(0);
}

function dfs(nodeLabel) {
  visited.add(nodeLabel);
  dfsOrder.push(nodeLabel);
  for (const neighbor of adjList[nodeLabel]) {
    if (!visited.has(neighbor)) {
      dfs(neighbor);
    }
  }
}

function animateDFS(index) {
  if (index >= dfsOrder.length) return;
  drawGraph(dfsOrder.slice(0, index + 1));
  setTimeout(() => animateDFS(index + 1), 800);
}

function drawGraph(highlighted = []) {
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  // Draw edges
  ctx.strokeStyle = "#999";
  for (const [from, to] of edges) {
    const fromNode = nodes.find(n => n.label === from);
    const toNode = nodes.find(n => n.label === to);
    ctx.beginPath();
    ctx.moveTo(fromNode.x, fromNode.y);
    ctx.lineTo(toNode.x, toNode.y);
    ctx.stroke();
  }

  // Draw nodes
  for (const node of nodes) {
    ctx.beginPath();
    ctx.arc(node.x, node.y, 25, 0, 2 * Math.PI);
    ctx.fillStyle = highlighted.includes(node.label) ? "#e74c3c" : "#3498db";
    ctx.fill();
    ctx.stroke();
```

```
        ctx.fillStyle = "#fff";
        ctx.font = "16px Arial";
        ctx.textAlign = "center";
        ctx.textBaseline = "middle";
        ctx.fillText(node.label, node.x, node.y);
      }
    }

    drawGraph();
  </script>
</body>
</html>
```

14.2.2 Recursive DFS Implementation (JavaScript)

Full runnable code:

```
class Graph {
  constructor() {
    this.adjList = new Map();
  }

  addVertex(vertex) {
    if (!this.adjList.has(vertex)) {
      this.adjList.set(vertex, []);
    }
  }

  addEdge(v1, v2) {
    this.addVertex(v1);
    this.addVertex(v2);
    this.adjList.get(v1).push(v2);
    // For undirected: also this.adjList.get(v2).push(v1);
  }

  dfsRecursive(start) {
    const visited = new Set();
    const result = [];

    const dfs = (vertex) => {
      if (!vertex || visited.has(vertex)) return;
      visited.add(vertex);
      result.push(vertex);

      for (const neighbor of this.adjList.get(vertex)) {
        dfs(neighbor);
      }
    };

    dfs(start);
    return result;
  }
}
```

```
const g = new Graph();
g.addEdge("A", "B");
g.addEdge("A", "C");
g.addEdge("B", "D");
g.addEdge("C", "E");

console.log(g.dfsRecursive("A")); // Output: [ 'A', 'B', 'D', 'C', 'E' ]
```

14.2.3 Iterative DFS Using a Stack

```
dfsIterative(start) {
  const visited = new Set();
  const result = [];
  const stack = [start];

  while (stack.length > 0) {
    const current = stack.pop();

    if (!visited.has(current)) {
      visited.add(current);
      result.push(current);

      // Add neighbors in reverse to maintain order
      for (const neighbor of [...this.adjList.get(current)].reverse()) {
        stack.push(neighbor);
      }
    }
  }

  return result;
}
```

This version manually manages the stack. It's functionally similar to the recursive approach but avoids recursion depth limits (especially in large graphs).

14.2.4 DFS Traversal Example

Graph:

```
   A
  / \
 B   C
 |    \
 D     E
```

DFS from `A` might visit: `A → B → D → C → E`

Note: The exact traversal order may vary based on how neighbors are added.

14.2.5 Applications of DFS

Cycle Detection

In directed or undirected graphs, DFS can detect back edges, which indicate cycles. If a node is encountered again before the recursion unwinds, a cycle is present.

Maze Solving / Puzzle Traversal

DFS can explore all possible paths through a structure like a maze or puzzle board. It naturally follows a path to the end before trying alternatives.

Topological Sorting

In Directed Acyclic Graphs (DAGs), DFS is used to order tasks where some must come before others. For example, course scheduling or build systems.

Connected Components

By running DFS from unvisited nodes, we can find all separate components in an undirected graph, useful in clustering or segmentation problems.

14.2.6 Time and Space Complexity

- **Time Complexity**: O(V + E) Each node and edge is visited once.
- **Space Complexity**:
  - Recursive: O(V) call stack (worst case)
  - Iterative: O(V) stack + visited set

14.2.7 Summary

Depth-First Search is a flexible and powerful traversal strategy that excels when you need to explore paths fully, backtrack, or analyze graph structures. Both **recursive and iterative** implementations are easy to express in JavaScript and have wide-ranging applications, from solving puzzles to analyzing dependencies in systems. Understanding DFS equips you to handle complex graph problems with depth-oriented logic and elegant solutions.

14.3 Applications (Pathfinding, Web Crawling)

Both **Breadth-First Search (BFS)** and **Depth-First Search (DFS)** are foundational algorithms for traversing graphs. While their mechanisms differ—BFS explores **level by level**, and DFS explores **deep before wide**—they each shine in different real-world tasks. This section explores their applications in two major areas: **pathfinding** and **web crawling**.

14.3.1 Pathfinding in Maps and Mazes

Use Case: Finding the shortest route in a grid or maze.

BFS is ideal for this because it explores all nodes at a given distance before moving deeper. In an **unweighted grid**, the first time BFS reaches a target cell guarantees the shortest path (by number of steps).

Example: Shortest Path in a Grid (BFS)

```
function bfsShortestPath(grid, start, end) {
  const rows = grid.length;
  const cols = grid[0].length;
  const directions = [[0,1],[1,0],[0,-1],[-1,0]];
  const visited = Array.from({ length: rows }, () => Array(cols).fill(false));
  const queue = [[...start, 0]]; // [row, col, steps]

  while (queue.length > 0) {
    const [r, c, steps] = queue.shift();
    if (r === end[0] && c === end[1]) return steps;
    if (visited[r][c]) continue;
    visited[r][c] = true;

    for (const [dr, dc] of directions) {
      const nr = r + dr, nc = c + dc;
      if (nr >= 0 && nc >= 0 && nr < rows && nc < cols && grid[nr][nc] === 0) {
        queue.push([nr, nc, steps + 1]);
      }
    }
  }

  return -1; // No path
}
```

Here, the grid is a 2D array where `0` is a walkable cell and `1` is a wall. BFS ensures the shortest route is found.

14.3.2 Web Crawling and Site Traversal

Use Case: Visiting all linked pages from a starting URL.

Web pages form a graph—nodes are pages, and edges are hyperlinks. **DFS** is often used in basic crawlers for **deep exploration**, while **BFS** can be used when limiting the crawl to a particular depth or breadth.

Example: Simulated Web Crawler (DFS)

```
function crawl(url, graph, visited = new Set()) {
  if (visited.has(url)) return;
  console.log("Crawling:", url);
  visited.add(url);

  for (const link of graph[url] || []) {
    crawl(link, graph, visited);
  }
}

// Simulated web
const web = {
  "home": ["about", "contact"],
  "about": ["team", "home"],
  "team": [],
  "contact": ["home"]
};

crawl("home", web);
```

This recursively visits all linked pages starting from `"home"`, preventing cycles using a `Set`.

14.3.3 Choosing BFS vs DFS

Task	Best Fit	Reason
Shortest path (unweighted)	**BFS**	Guarantees the shortest number of steps
Maze solving (any path)	**DFS**	Finds any path, potentially faster with pruning
Web crawling	**DFS/BFS**	DFS for depth, BFS for breadth or max-depth limits
Connected components	**DFS**	Efficient for grouping related nodes

14.3.4 Performance & Trade-offs

- **BFS** uses more memory since it stores all neighbors at the current depth.

- **DFS** is lighter in memory but may go deep unnecessarily if not guided.
- For huge or infinite graphs (like the web), adding depth limits or priority rules is essential.

14.3.5 Summary

BFS and DFS both power a wide range of real-world applications. Use **BFS** when you care about **minimal steps** or **shortest paths** (like in maps and routing), and use **DFS** when you need **complete exploration** or are working with **structured, tree-like data** (like site maps or puzzle paths). With flexible JavaScript implementations and clear trade-offs, these algorithms are powerful tools in your problem-solving toolkit.

Chapter 15.

Graph Shortest Paths

1. Dijkstra's Algorithm
2. Bellman-Ford Algorithm
3. A* Search (Practical Frontend Use)**

15 Graph Shortest Paths

15.1 Dijkstra's Algorithm

Dijkstra's Algorithm is a classic algorithm used to compute the **shortest path from a source node** to all other nodes in a **weighted graph**—as long as edge weights are **non-negative**. It's widely used in applications like GPS navigation, network routing, and real-time systems where optimal paths are crucial.

15.1.1 How Dijkstras Algorithm Works

Dijkstra's algorithm keeps track of the **shortest known distance** to each node and updates it as shorter paths are found. It uses a **priority queue** (typically a min-heap) to efficiently select the next node with the smallest tentative distance.

Core Steps:

1. Initialize distances from the source to all nodes as `Infinity`, except the source itself (`0`).
2. Use a priority queue to always process the node with the **smallest tentative distance**.
3. For each unvisited neighbor of the current node:
    - Calculate the new distance through the current node.
    - If it's smaller than the known distance, update it and reinsert the neighbor in the queue.
4. Repeat until all nodes have been processed.

Run the following code in browser to see the demo:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>Dijkstra's Algorithm Visualization</title>
<style>
  body { font-family: sans-serif; text-align: center; padding: 20px; }
  canvas { border: 1px solid #ccc; background: #f9f9f9; }
  button { font-size: 16px; padding: 8px 12px; margin: 10px; }
  #info { margin-top: 10px; font-family: monospace; }
</style>
</head>
<body>

<h2>Dijkstra's Algorithm Visualization</h2>
<button onclick="startDijkstra()">Start Dijkstra</button>
<canvas id="canvas" width="600" height="600"></canvas>
```

```
<div id="info"></div>

<script>
  const canvas = document.getElementById("canvas");
  const ctx = canvas.getContext("2d");
  const infoDiv = document.getElementById("info");

  // Nodes with fixed positions
  const nodes = [
    { label: 'A', x: 150, y: 100 },
    { label: 'B', x: 350, y: 100 },
    { label: 'C', x: 550, y: 100 },
    { label: 'D', x: 150, y: 300 },
    { label: 'E', x: 350, y: 300 },
    { label: 'F', x: 550, y: 300 },
    { label: 'G', x: 350, y: 450 }
  ];

  // Edges with weights (undirected)
  // Each edge: [fromLabel, toLabel, weight]
  const edges = [
    ['A', 'B', 4], ['A', 'D', 2],
    ['B', 'C', 3], ['B', 'E', 5],
    ['C', 'F', 1],
    ['D', 'E', 8], ['D', 'G', 10],
    ['E', 'F', 2], ['E', 'G', 6],
    ['F', 'G', 3]
  ];

  // Build adjacency list: label -> [{to, weight}, ...]
  const adjList = {};
  for (const node of nodes) adjList[node.label] = [];
  for (const [from, to, w] of edges) {
    adjList[from].push({ to, weight: w });
    adjList[to].push({ to: from, weight: w }); // undirected
  }

  // Dijkstra state
  let distances = {};
  let previous = {};
  let visited = new Set();
  let pq = []; // priority queue of {label, dist}
  let currentStep = 0;
  let steps = [];

  // Priority queue helper functions
  function enqueue(label, dist) {
    pq.push({ label, dist });
    pq.sort((a,b) => a.dist - b.dist);
  }
  function dequeue() {
    return pq.shift();
  }

  function startDijkstra() {
    distances = {};
    previous = {};
    visited = new Set();
```

```
  pq = [];
  currentStep = 0;
  steps = [];

  for (const node of nodes) {
    distances[node.label] = Infinity;
    previous[node.label] = null;
  }
  distances['A'] = 0;
  enqueue('A', 0);

  // Run algorithm stepwise, recording steps for animation
  while (pq.length > 0) {
    const { label: u, dist: uDist } = dequeue();
    if (visited.has(u)) continue;
    visited.add(u);

    // Save step snapshot
    steps.push({
      current: u,
      distances: {...distances},
      previous: {...previous},
      visited: new Set(visited)
    });

    for (const { to: v, weight } of adjList[u]) {
      if (!visited.has(v)) {
        const alt = distances[u] + weight;
        if (alt < distances[v]) {
          distances[v] = alt;
          previous[v] = u;
          enqueue(v, alt);
        }
      }
    }
  }
  currentStep = 0;
  animateStep();
}

function animateStep() {
  if (currentStep >= steps.length) {
    infoDiv.textContent = "Dijkstra completed! Final distances: " + JSON.stringify(distances, null, 2)
    drawGraph(steps[steps.length-1]);
    return;
  }
  drawGraph(steps[currentStep]);
  infoDiv.textContent = `Processing node: ${steps[currentStep].current}\n` +
    `Distances: ${JSON.stringify(steps[currentStep].distances, null, 2)}\n` +
    `Visited: ${[...steps[currentStep].visited].join(', ')}`;
  currentStep++;
  setTimeout(animateStep, 1200);
}

// Draw graph with current step info
function drawGraph(step) {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
```

```
    // Draw edges with weights
    ctx.strokeStyle = "#999";
    ctx.lineWidth = 2;
    ctx.fillStyle = "#000";
    ctx.font = "14px Arial";

    for (const [from, to, w] of edges) {
      const fromNode = nodes.find(n => n.label === from);
      const toNode = nodes.find(n => n.label === to);
      ctx.beginPath();
      ctx.moveTo(fromNode.x, fromNode.y);
      ctx.lineTo(toNode.x, toNode.y);
      ctx.stroke();

      // Draw weight label at midpoint with offset
      const midX = (fromNode.x + toNode.x) / 2;
      const midY = (fromNode.y + toNode.y) / 2;
      ctx.fillStyle = "#000";
      ctx.fillText(w, midX + 5, midY - 5);
    }

    // Draw nodes
    for (const node of nodes) {
      const dist = step.distances[node.label];
      const isVisited = step.visited.has(node.label);
      const isCurrent = step.current === node.label;

      ctx.beginPath();
      ctx.arc(node.x, node.y, 25, 0, 2 * Math.PI);
      ctx.fillStyle = isCurrent ? '#e67e22' : (isVisited ? '#2ecc71' : '#3498db');
      ctx.fill();
      ctx.stroke();

      ctx.fillStyle = '#fff';
      ctx.font = '16px Arial';
      ctx.textAlign = 'center';
      ctx.textBaseline = 'middle';
      ctx.fillText(node.label, node.x, node.y);

      // Draw distance inside node
      ctx.font = '12px Arial';
      ctx.fillText(dist === Infinity ? '∞' : dist, node.x, node.y + 30);
    }

    // Draw shortest path tree edges (bold)
    ctx.strokeStyle = '#27ae60';
    ctx.lineWidth = 4;
    for (const nodeLabel in step.previous) {
      const prev = step.previous[nodeLabel];
      if (prev) {
        const fromNode = nodes.find(n => n.label === prev);
        const toNode = nodes.find(n => n.label === nodeLabel);
        ctx.beginPath();
        ctx.moveTo(fromNode.x, fromNode.y);
        ctx.lineTo(toNode.x, toNode.y);
        ctx.stroke();
      }
    }
```

```
  }

  // Initial draw
  drawGraph({
    current: null,
    distances: nodes.reduce((acc, n) => { acc[n.label] = Infinity; return acc; }, {}),
    previous: {},
    visited: new Set()
  });
</script>

</body>
</html>
```

15.1.2 Graph and Priority Queue Setup

To implement Dijkstra's in JavaScript, we need:

1. **Adjacency list** to store the graph.
2. A simple **priority queue** (we'll use a min-heap).
3. A **distance map** to track the shortest known distance to each node.

15.1.3 JavaScript Implementation

Priority Queue (Min-Heap)

```
class PriorityQueue {
  constructor() {
    this.values = [];
  }

  enqueue(val, priority) {
    this.values.push({ val, priority });
    this.bubbleUp();
  }

  bubbleUp() {
    let idx = this.values.length - 1;
    const element = this.values[idx];

    while (idx > 0) {
      let parentIdx = Math.floor((idx - 1) / 2);
      let parent = this.values[parentIdx];

      if (element.priority >= parent.priority) break;
      this.values[parentIdx] = element;
      this.values[idx] = parent;
      idx = parentIdx;
    }
```

```
  }

  dequeue() {
    const min = this.values[0];
    const end = this.values.pop();
    if (this.values.length > 0) {
      this.values[0] = end;
      this.sinkDown();
    }
    return min;
  }

  sinkDown() {
    let idx = 0;
    const length = this.values.length;
    const element = this.values[0];

    while (true) {
      let leftIdx = 2 * idx + 1;
      let rightIdx = 2 * idx + 2;
      let swap = null;

      if (leftIdx < length) {
        if (this.values[leftIdx].priority < element.priority) {
          swap = leftIdx;
        }
      }

      if (rightIdx < length) {
        if (
          this.values[rightIdx].priority < (swap === null ? element.priority : this.values[leftIdx].pri
        ) {
          swap = rightIdx;
        }
      }

      if (swap === null) break;

      this.values[idx] = this.values[swap];
      this.values[swap] = element;
      idx = swap;
    }
  }

  isEmpty() {
    return this.values.length === 0;
  }
}
```

Dijkstras Algorithm

```
class Graph {
  constructor() {
    this.adjList = new Map();
  }

  addVertex(vertex) {
```

```
    if (!this.adjList.has(vertex)) this.adjList.set(vertex, []);
  }

  addEdge(v1, v2, weight) {
    this.addVertex(v1);
    this.addVertex(v2);
    this.adjList.get(v1).push({ node: v2, weight });
    // For undirected graph, also add: this.adjList.get(v2).push({ node: v1, weight });
  }

  dijkstra(start) {
    const distances = new Map();
    const pq = new PriorityQueue();
    const previous = new Map();

    for (const vertex of this.adjList.keys()) {
      distances.set(vertex, Infinity);
      previous.set(vertex, null);
    }

    distances.set(start, 0);
    pq.enqueue(start, 0);

    while (!pq.isEmpty()) {
      const { val: current } = pq.dequeue();

      for (const neighbor of this.adjList.get(current)) {
        const distance = distances.get(current) + neighbor.weight;
        if (distance < distances.get(neighbor.node)) {
          distances.set(neighbor.node, distance);
          previous.set(neighbor.node, current);
          pq.enqueue(neighbor.node, distance);
        }
      }
    }

    return { distances, previous };
  }
}
```

Usage Example

```
const g = new Graph();
g.addEdge("A", "B", 4);
g.addEdge("A", "C", 2);
g.addEdge("B", "E", 3);
g.addEdge("C", "D", 2);
g.addEdge("D", "E", 3);
g.addEdge("C", "F", 4);
g.addEdge("E", "F", 1);

const result = g.dijkstra("A");
console.log(result.distances);
```

15.1.4 Time Complexity

- **With Binary Heap**: O((V + E) * log V)
- **Without Heap** (linear scan): $O(V^2)$, which is inefficient for large graphs

15.1.5 Real-World Applications

- **GPS and Navigation Systems**: Find the fastest route based on distance or time.
- **Network Routing**: Optimize data packet travel in routers and switches.
- **Game Development**: Pathfinding for AI movement in maps or terrains.
- **Logistics and Delivery**: Optimize routes for deliveries and pickups.

15.1.6 Summary

Dijkstra's algorithm is a reliable and efficient way to find the **shortest path in graphs with non-negative weights**. By using a **priority queue**, it ensures we always expand the most promising node next, making it both optimal and greedy. Mastering this algorithm is essential for solving real-world routing and optimization problems efficiently in JavaScript.

15.2 Bellman-Ford Algorithm

The **Bellman-Ford algorithm** is a classic solution for finding the **shortest paths from a single source** in a **weighted graph**, even when **negative edge weights** are present. Unlike Dijkstra's algorithm, Bellman-Ford doesn't assume all weights are positive, making it crucial in scenarios where penalties, costs, or debt-like relationships exist.

15.2.1 When to Use Bellman-Ford

- Graphs may contain **negative edge weights**.
- Detecting **negative weight cycles** is essential.
- You need a simpler, more robust (but slower) algorithm than Dijkstra.

15.2.2 Core Idea: Relaxation

The Bellman-Ford algorithm works by performing a series of **relaxation steps**:

1. **Relaxation** means trying to improve the shortest distance to each vertex by checking all edges.
2. This process is repeated **(V - 1)** times, where `V` is the number of vertices.
3. After `V - 1` iterations, if any edge can still be relaxed, a **negative cycle exists**.

Run the following code in browser to see the demo:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>Bellman-Ford Algorithm Visualization</title>
<style>
  body { font-family: sans-serif; text-align: center; padding: 20px; }
  canvas { border: 1px solid #ccc; background: #f9f9f9; }
  button { font-size: 16px; padding: 8px 12px; margin: 10px; }
  #info { margin-top: 10px; font-family: monospace; white-space: pre-wrap; }
</style>
</head>
<body>

<h2>Bellman-Ford Algorithm Visualization</h2>
<button onclick="startBellmanFord()">Start Bellman-Ford</button>
<canvas id="canvas" width="600" height="600"></canvas>
<div id="info"></div>

<script>
  const canvas = document.getElementById("canvas");
  const ctx = canvas.getContext("2d");
  const infoDiv = document.getElementById("info");

  // Nodes with positions
  const nodes = [
    { label: 'A', x: 150, y: 100 },
    { label: 'B', x: 350, y: 100 },
    { label: 'C', x: 550, y: 100 },
    { label: 'D', x: 150, y: 300 },
    { label: 'E', x: 350, y: 300 },
    { label: 'F', x: 550, y: 300 },
  ];

  // Directed edges with weights: [from, to, weight]
  const edges = [
    ['A', 'B', 6],
    ['A', 'D', 7],
    ['B', 'C', 5],
    ['B', 'E', -4],
    ['B', 'D', 8],
    ['C', 'B', -2],
    ['D', 'E', 9],
    ['E', 'C', 7],
    ['E', 'F', 2],
    ['F', 'A', 2],
```

```
  ['F', 'C', -5]
];

// Build adjacency list for edges
const adjList = {};
for (const node of nodes) adjList[node.label] = [];
for (const [from, to, w] of edges) {
  adjList[from].push({ to, weight: w });
}

let distances = {};
let previous = {};
let steps = [];
let currentStep = 0;
let negativeCycleDetected = false;

function startBellmanFord() {
  distances = {};
  previous = {};
  steps = [];
  currentStep = 0;
  negativeCycleDetected = false;

  for (const node of nodes) {
    distances[node.label] = Infinity;
    previous[node.label] = null;
  }
  distances['A'] = 0;

  // Bellman-Ford relaxation steps
  const V = nodes.length;

  // Relax edges V-1 times
  for (let i = 1; i < V; i++) {
    for (const [u, v, w] of edges) {
      if (distances[u] !== Infinity && distances[u] + w < distances[v]) {
        distances[v] = distances[u] + w;
        previous[v] = u;
        steps.push({
          step: `Relax edge ${u} -> ${v} (${w})`,
          distances: { ...distances },
          previous: { ...previous },
          currentEdge: [u, v, w],
          iteration: i,
          negativeCycle: false
        });
      }
    }
  }

  // Check for negative cycles
  for (const [u, v, w] of edges) {
    if (distances[u] !== Infinity && distances[u] + w < distances[v]) {
      negativeCycleDetected = true;
      steps.push({
        step: `Negative cycle detected on edge ${u} -> ${v} (${w})`,
        distances: { ...distances },
        previous: { ...previous },
```

```
        currentEdge: [u, v, w],
        iteration: V,
        negativeCycle: true
      });
      break;
    }
  }

  if (steps.length === 0) {
    infoDiv.textContent = "No updates needed, shortest paths are initial distances.";
    drawGraph({
      distances: distances,
      previous: previous,
      currentEdge: null,
      negativeCycle: false
    });
    return;
  }

  animateStep();
}

function animateStep() {
  if (currentStep >= steps.length) {
    infoDiv.textContent += negativeCycleDetected
      ? "\nNegative cycle detected, shortest paths not valid!"
      : "\nAlgorithm completed successfully.";
    drawGraph(steps[steps.length - 1]);
    return;
  }
  const step = steps[currentStep];
  infoDiv.textContent = `Iteration: ${step.iteration}\n${step.step}\n\nDistances:\n` +
    Object.entries(step.distances).map(([k,v]) => `${k}: ${v === Infinity ? 'ω' : v}`).join('\n') +
    (step.negativeCycle ? '\n\nNegative cycle detected!' : '');
  drawGraph(step);
  currentStep++;
  setTimeout(animateStep, 1300);
}

function drawGraph(step) {
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  // Draw edges with weights and highlight current edge
  ctx.lineWidth = 2;
  for (const [from, to, w] of edges) {
    const fromNode = nodes.find(n => n.label === from);
    const toNode = nodes.find(n => n.label === to);

    ctx.beginPath();
    ctx.strokeStyle = (step.currentEdge && step.currentEdge[0] === from && step.currentEdge[1] === to
    ctx.moveTo(fromNode.x, fromNode.y);
    ctx.lineTo(toNode.x, toNode.y);
    ctx.stroke();

    // Draw arrow for directed edge
    drawArrow(fromNode.x, fromNode.y, toNode.x, toNode.y, ctx.strokeStyle === '#e74c3c');

    // Draw weight label
```

```
    const midX = (fromNode.x + toNode.x) / 2;
    const midY = (fromNode.y + toNode.y) / 2;
    ctx.fillStyle = '#000';
    ctx.font = '14px Arial';
    ctx.fillText(w, midX + 5, midY - 5);
  }

  // Draw nodes
  for (const node of nodes) {
    const dist = step.distances[node.label];
    const isUpdatedNode = step.currentEdge && (step.currentEdge[1] === node.label);
    ctx.beginPath();
    ctx.arc(node.x, node.y, 25, 0, 2 * Math.PI);
    ctx.fillStyle = isUpdatedNode ? '#f39c12' : '#3498db';
    ctx.fill();
    ctx.stroke();

    ctx.fillStyle = '#fff';
    ctx.font = '16px Arial';
    ctx.textAlign = 'center';
    ctx.textBaseline = 'middle';
    ctx.fillText(node.label, node.x, node.y);

    // Distance below node
    ctx.font = '12px Arial';
    ctx.fillText(dist === Infinity ? '∞' : dist, node.x, node.y + 30);
  }

  // Draw shortest path tree edges (bold)
  ctx.strokeStyle = '#27ae60';
  ctx.lineWidth = 4;
  for (const nodeLabel in step.previous) {
    const prev = step.previous[nodeLabel];
    if (prev) {
      const fromNode = nodes.find(n => n.label === prev);
      const toNode = nodes.find(n => n.label === nodeLabel);
      ctx.beginPath();
      ctx.moveTo(fromNode.x, fromNode.y);
      ctx.lineTo(toNode.x, toNode.y);
      ctx.stroke();
    }
  }
}

function drawArrow(fromX, fromY, toX, toY, highlight) {
  const headlen = 10;
  const angle = Math.atan2(toY - fromY, toX - fromX);
  ctx.fillStyle = highlight ? '#e74c3c' : '#999';

  ctx.beginPath();
  ctx.moveTo(toX, toY);
  ctx.lineTo(toX - headlen * Math.cos(angle - Math.PI / 6), toY - headlen * Math.sin(angle - Math.PI /
  ctx.lineTo(toX - headlen * Math.cos(angle + Math.PI / 6), toY - headlen * Math.sin(angle + Math.PI /
  ctx.closePath();
  ctx.fill();
}

// Initial draw with no distances
```

```
  drawGraph({
    distances: nodes.reduce((acc, n) => { acc[n.label] = Infinity; return acc; }, {}),
    previous: {},
    currentEdge: null,
    negativeCycle: false
  });
</script>

</body>
</html>
```

15.2.3 JavaScript Implementation

Let's implement Bellman-Ford with a simple edge list representation.

Full runnable code:

```
class Graph {
  constructor() {
    this.vertices = new Set();
    this.edges = []; // Each edge: { from, to, weight }
  }

  addEdge(from, to, weight) {
    this.vertices.add(from);
    this.vertices.add(to);
    this.edges.push({ from, to, weight });
  }

  bellmanFord(start) {
    const distances = {};
    const predecessors = {};

    // Step 1: Initialize distances
    for (let vertex of this.vertices) {
      distances[vertex] = Infinity;
      predecessors[vertex] = null;
    }
    distances[start] = 0;

    const V = this.vertices.size;

    // Step 2: Relax edges repeatedly
    for (let i = 0; i < V - 1; i++) {
      for (let { from, to, weight } of this.edges) {
        if (distances[from] + weight < distances[to]) {
          distances[to] = distances[from] + weight;
          predecessors[to] = from;
        }
      }
    }

    // Step 3: Check for negative-weight cycles
```

```
    for (let { from, to, weight } of this.edges) {
      if (distances[from] + weight < distances[to]) {
        throw new Error("Graph contains a negative weight cycle");
      }
    }

    return { distances, predecessors };
  }
}

const g = new Graph();
g.addEdge("A", "B", 4);
g.addEdge("A", "C", 2);
g.addEdge("C", "B", -2);
g.addEdge("B", "D", 2);
g.addEdge("C", "D", 3);

const result = g.bellmanFord("A");
console.log(JSON.stringify(result.distances,null,2)); // { A: 0, B: 0, C: 2, D: 2 }
```

This shows that even with a negative edge (C → B with weight -2), Bellman-Ford still computes correct shortest paths.

15.2.4 Negative Cycle Detection

If the graph contains a cycle where the total weight is negative (e.g., a loop with net cost -3), Bellman-Ford will detect it and throw an error or return a signal that no shortest path exists due to infinite reduction.

15.2.5 Time Complexity

- **Time**: O(V × E) Slower than Dijkstra (O((V + E) log V) with a min-heap), especially for dense graphs.
- **Space**: O(V) for storing distances and predecessors.

15.2.6 Real-World Applications

- **Financial Systems**: Detecting arbitrage opportunities (negative cycles).
- **Network Routing**: Calculating paths where costs may include penalties.
- **AI & Game Development**: Evaluating graphs with penalties or risk factors.
- **Compiler Design**: Finding optimal register allocations with weighted graphs.

15.2.7 Dijkstra vs Bellman-Ford

Feature	Dijkstra	Bellman-Ford
Handles negative weights	NO No	YES Yes
Detects negative cycles	NO No	YES Yes
Time complexity	Faster (with heap)	Slower (O(V × E))
Implementation complexity	Higher (needs heap)	Simpler

15.2.8 Summary

The **Bellman-Ford algorithm** is a powerful and flexible choice for shortest path problems where edge weights may be negative or where cycle detection is necessary. Though not as fast as Dijkstra's algorithm, its broader applicability makes it a valuable tool in any JavaScript algorithmist's toolkit.

15.3 A* Search (Practical Frontend Use)**

The **A*** (A-star) search algorithm is a powerful and efficient pathfinding technique, particularly popular in **games**, **UI navigation**, and **interactive frontend tools**. It builds upon Dijkstra's algorithm by incorporating a **heuristic function**, which estimates the cost from a node to the goal. This makes A* both **optimal** and **goal-directed**, reducing unnecessary exploration.

15.3.1 What Is A* Search?

A* combines the best of:

- **Dijkstra's algorithm**, which finds the shortest path using actual costs (`g(n)`), and
- **Greedy Best-First Search**, which uses a heuristic guess (`h(n)`) to move toward the goal.

It uses a priority function:

```
f(n) = g(n) + h(n)
```

Where:

- `g(n)` = cost from start to node `n`
- `h(n)` = estimated cost from `n` to goal (heuristic)

- `f(n)` = total estimated cost of the path through `n`

Run the following code in browser to see the demo:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>A* Pathfinding Visualization</title>
<style>
  body { font-family: sans-serif; text-align: center; padding: 20px; }
  canvas { border: 1px solid #ccc; background: #f9f9f9; display: block; margin: 0 auto; }
  button { font-size: 16px; padding: 8px 12px; margin: 10px; }
</style>
</head>
<body>

<h2>A* Algorithm Visualization on Grid</h2>
<button onclick="startAStar()">Start A*</button>
<canvas id="canvas" width="600" height="600"></canvas>

<script>
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');

const ROWS = 20;
const COLS = 20;
const CELL_SIZE = 30;

// Grid representation: 0 = walkable, 1 = obstacle
const grid = [];
for(let r=0; r<ROWS; r++){
  grid[r] = [];
  for(let c=0; c<COLS; c++){
    grid[r][c] = 0;
  }
}

// Add some obstacles
const obstacleCoords = [
  [5,5],[5,6],[5,7],[5,8],[5,9],[5,10],
  [10,3],[11,3],[12,3],[13,3],[14,3],
  [15,10],[15,11],[15,12],[15,13],[15,14],
  [8,15],[9,15],[10,15],[11,15],[12,15]
];
for(const [r,c] of obstacleCoords){
  grid[r][c] = 1;
}

const start = {r:0, c:0};
const goal = {r:ROWS-1, c:COLS-1};

function drawGrid(openSet = [], closedSet = [], path = []) {
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  for(let r=0; r<ROWS; r++) {
    for(let c=0; c<COLS; c++) {
      let color = '#3498db'; // default walkable: blue
```

```
      if(grid[r][c] === 1) color = '#555'; // obstacle: dark gray

      // Highlight sets
      if (closedSet.some(n => n.r === r && n.c === c)) color = '#e74c3c'; // closed: red
      if (openSet.some(n => n.r === r && n.c === c)) color = '#2ecc71'; // open: green
      if (path.some(n => n.r === r && n.c === c)) color = '#f1c40f'; // path: yellow
      if(r === start.r && c === start.c) color = '#27ae60'; // start: dark green
      if(r === goal.r && c === goal.c) color = '#9b59b6'; // goal: purple

      ctx.fillStyle = color;
      ctx.fillRect(c*CELL_SIZE, r*CELL_SIZE, CELL_SIZE, CELL_SIZE);
      ctx.strokeStyle = '#222';
      ctx.strokeRect(c*CELL_SIZE, r*CELL_SIZE, CELL_SIZE, CELL_SIZE);
    }
  }
}

// Helper for heuristic (Manhattan distance)
function heuristic(a, b) {
  return Math.abs(a.r - b.r) + Math.abs(a.c - b.c);
}

// Get neighbors of a cell (4 directions)
function neighbors(node) {
  const results = [];
  const {r,c} = node;
  if(r>0 && grid[r-1][c] === 0) results.push({r:r-1,c});
  if(r<ROWS-1 && grid[r+1][c] === 0) results.push({r:r+1,c});
  if(c>0 && grid[r][c-1] === 0) results.push({r,c:c-1});
  if(c<COLS-1 && grid[r][c+1] === 0) results.push({r,c:c+1});
  return results;
}

// A* algorithm variables
let openSet = [];
let closedSet = [];
let cameFrom = {};
let gScore = {};
let fScore = {};
let finished = false;

function nodeKey(node) {
  return `${node.r},${node.c}`;
}

function startAStar() {
  openSet = [start];
  closedSet = [];
  cameFrom = {};
  gScore = {};
  fScore = {};
  finished = false;

  for(let r=0; r<ROWS; r++){
    for(let c=0; c<COLS; c++){
      gScore[nodeKey({r,c})] = Infinity;
      fScore[nodeKey({r,c})] = Infinity;
    }
```

```
  }
  gScore[nodeKey(start)] = 0;
  fScore[nodeKey(start)] = heuristic(start, goal);

  animateStep();
}

function reconstructPath(current) {
  const path = [];
  while (nodeKey(current) in cameFrom) {
    path.push(current);
    current = cameFrom[nodeKey(current)];
  }
  path.push(start);
  return path.reverse();
}

function animateStep() {
  if (openSet.length === 0) {
    // No path found
    info("No path found!");
    drawGrid([], closedSet, []);
    return;
  }
  // Pick node in openSet with lowest fScore
  let currentIndex = 0;
  for(let i=1; i<openSet.length; i++){
    if(fScore[nodeKey(openSet[i])] < fScore[nodeKey(openSet[currentIndex])]){
      currentIndex = i;
    }
  }
  const current = openSet[currentIndex];

  if(current.r === goal.r && current.c === goal.c){
    finished = true;
    const path = reconstructPath(current);
    drawGrid(openSet, closedSet, path);
    info("Path found! Length: " + (path.length -1));
    return;
  }

  openSet.splice(currentIndex, 1);
  closedSet.push(current);

  for(const neighbor of neighbors(current)){
    if(closedSet.some(n => n.r === neighbor.r && n.c === neighbor.c)) continue;

    const tentativeG = gScore[nodeKey(current)] + 1;
    if(!openSet.some(n => n.r === neighbor.r && n.c === neighbor.c)){
      openSet.push(neighbor);
    } else if(tentativeG >= gScore[nodeKey(neighbor)]){
      continue;
    }

    cameFrom[nodeKey(neighbor)] = current;
    gScore[nodeKey(neighbor)] = tentativeG;
    fScore[nodeKey(neighbor)] = tentativeG + heuristic(neighbor, goal);
  }
```

```
  drawGrid(openSet, closedSet, []);
  setTimeout(() => animateStep(), 100);
}

function info(text) {
  // Could add an info panel if desired
  console.log(text);
}

drawGrid();
</script>

</body>
</html>
```

15.3.2 JavaScript Implementation

Let's build a simplified A* search to navigate a **2D grid**, like in games or interactive maps.

Grid and Heuristic Setup

```
function heuristic(a, b) {
  // Manhattan distance (suitable for 4-directional grids)
  return Math.abs(a[0] - b[0]) + Math.abs(a[1] - b[1]);
}
```

A* Algorithm

Full runnable code:

```
function aStar(grid, start, goal) {
  const rows = grid.length;
  const cols = grid[0].length;
  const openSet = new Set([start.toString()]);
  const cameFrom = new Map();

  const gScore = {};
  const fScore = {};

  for (let r = 0; r < rows; r++) {
    for (let c = 0; c < cols; c++) {
      gScore[[r, c]] = Infinity;
      fScore[[r, c]] = Infinity;
    }
  }

  gScore[start] = 0;
  fScore[start] = heuristic(start, goal);

  const queue = [{ node: start, f: fScore[start] }];

  while (queue.length > 0) {
```

```
    queue.sort((a, b) => a.f - b.f); // Priority queue by f(n)
    const { node: current } = queue.shift();
    const key = current.toString();

    if (key === goal.toString()) {
      // Reconstruct path
      const path = [];
      let curr = key;
      while (curr) {
        path.unshift(curr.split(',').map(Number));
        curr = cameFrom.get(curr);
      }
      return path;
    }

    openSet.delete(key);

    for (let [dr, dc] of [[0,1],[1,0],[0,-1],[-1,0]]) {
      const nr = current[0] + dr;
      const nc = current[1] + dc;
      if (nr < 0 || nc < 0 || nr >= rows || nc >= cols || grid[nr][nc] === 1) continue;

      const neighbor = [nr, nc];
      const neighborKey = neighbor.toString();
      const tentativeG = gScore[key] + 1;

      if (tentativeG < gScore[neighborKey]) {
        cameFrom.set(neighborKey, key);
        gScore[neighborKey] = tentativeG;
        fScore[neighborKey] = tentativeG + heuristic(neighbor, goal);
        if (!openSet.has(neighborKey)) {
          openSet.add(neighborKey);
          queue.push({ node: neighbor, f: fScore[neighborKey] });
        }
      }
    }
  }

  return []; // No path found
}
function heuristic(a, b) {
  // Manhattan distance (suitable for 4-directional grids)
  return Math.abs(a[0] - b[0]) + Math.abs(a[1] - b[1]);
}
const grid = [
  [0, 0, 0, 0],
  [1, 1, 0, 1],
  [0, 0, 0, 0]
];

const start = [0, 0];
const goal = [2, 3];

const path = aStar(grid, start, goal);
console.log(path);
// Output: a list of coordinates forming the shortest path
```

Grid Visualization

```
S . . .
# # . #
. . . G
```

Where `S` = start, `G` = goal, `#` = wall. A* will explore only optimal directions toward the goal, guided by the heuristic.

15.3.3 Heuristic Function Design

A* relies heavily on a good **heuristic**:

- **Manhattan distance**: `|x1 - x2| + |y1 - y2|` (for 4-directional grids)
- **Euclidean distance**: `sqrt((x1 - x2)^2 + (y1 - y2)^2)` (for diagonal movement)
- **Zero heuristic**: A* becomes Dijkstra's algorithm

Trade-offs:

- **Admissibility**: The heuristic must **never overestimate** the actual cost.
- A stronger (but admissible) heuristic makes A* faster.
- Over-aggressive heuristics can result in suboptimal paths.

15.3.4 Performance Considerations

- **Time complexity**: O(E), but depends on heuristic quality and branching factor.
- **Space complexity**: O(V), for open set and score maps.
- For large grids or dense graphs, optimizations like **binary heaps** or **early goal detection** improve performance.

15.3.5 Real-World Applications

- **Game AI**: Characters finding paths on a game map.
- **Drag-and-drop UI design**: Calculating valid movement areas.
- **Interactive maps**: Real-time navigation or logistics planning.
- **Robot movement**: Path planning in simulation or real environments.

15.3.6 Summary

A* Search is a flexible and efficient algorithm that finds **optimal paths** while reducing unnecessary exploration. It's a go-to tool for pathfinding in **frontend JavaScript**, especially in interactive and visual contexts like games or layout systems. With a well-chosen heuristic and efficient implementation, A* delivers performance and precision in equal measure.

Chapter 16.

Minimum Spanning Trees

1. Prim's Algorithm
2. Kruskal's Algorithm
3. Real-World Applications

16 Minimum Spanning Trees

16.1 Prim's Algorithm

In graph theory, a **Minimum Spanning Tree (MST)** is a subset of edges in a **connected, weighted, undirected graph** that connects all the vertices together without cycles and with the minimum possible total edge weight. **Prim's algorithm** is a greedy technique to construct an MST by **expanding from a starting node**, always choosing the smallest edge that adds a new vertex to the tree.

16.1.1 How Prims Algorithm Works

Prim's algorithm grows a tree by continuously selecting the **lowest-weight edge** that connects a visited node to an unvisited one.

Step-by-step Process:

1. Start from any arbitrary node.
2. Mark it as visited.
3. Push all its edges into a **priority queue** sorted by weight.
4. At each step:
    - Extract the smallest edge from the queue.
    - If it connects to an unvisited node, add that node and the edge to the MST.
    - Push all adjacent edges of the new node into the queue.
5. Repeat until all vertices are included.

Run the following code in browser to see the demo:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>Minimum Spanning Tree (Kruskal's Algorithm)</title>
<style>
  body { font-family: sans-serif; text-align: center; padding: 20px; }
  canvas { border: 1px solid #ccc; background: #f9f9f9; display: block; margin: 0 auto; }
  button { font-size: 16px; padding: 8px 12px; margin: 10px; }
  #info { font-family: monospace; margin-top: 10px; }
</style>
</head>
<body>

<h2>Minimum Spanning Tree (Kruskal's Algorithm) Visualization</h2>
<button onclick="startKruskal()">Start Kruskal's MST</button>
<canvas id="canvas" width="600" height="600"></canvas>
```

```
<div id="info"></div>

<script>
  const canvas = document.getElementById("canvas");
  const ctx = canvas.getContext("2d");
  const infoDiv = document.getElementById("info");

  const nodes = [
    { label: 'A', x: 100, y: 100 },
    { label: 'B', x: 300, y: 80 },
    { label: 'C', x: 500, y: 120 },
    { label: 'D', x: 150, y: 300 },
    { label: 'E', x: 350, y: 320 },
    { label: 'F', x: 550, y: 320 },
    { label: 'G', x: 400, y: 450 },
  ];

  // Undirected edges: [from, to, weight]
  const edges = [
    ['A', 'B', 7],
    ['A', 'D', 5],
    ['B', 'C', 8],
    ['B', 'D', 9],
    ['B', 'E', 7],
    ['C', 'E', 5],
    ['D', 'E', 15],
    ['D', 'G', 6],
    ['E', 'F', 8],
    ['E', 'G', 9],
    ['F', 'G', 11],
  ];

  // Disjoint Set (Union-Find) data structure
  class DisjointSet {
    constructor(elements) {
      this.parent = {};
      this.rank = {};
      for (const el of elements) {
        this.parent[el] = el;
        this.rank[el] = 0;
      }
    }

    find(u) {
      if (this.parent[u] !== u) {
        this.parent[u] = this.find(this.parent[u]);
      }
      return this.parent[u];
    }

    union(u, v) {
      const rootU = this.find(u);
      const rootV = this.find(v);
      if (rootU === rootV) return false;
      if (this.rank[rootU] < this.rank[rootV]) {
        this.parent[rootU] = rootV;
      } else if (this.rank[rootV] < this.rank[rootU]) {
        this.parent[rootV] = rootU;
```

```
    } else {
      this.parent[rootV] = rootU;
      this.rank[rootU]++;
    }
    return true;
  }
}

let mstEdges = [];
let stepIndex = 0;
let sortedEdges = [];
let ds;

function startKruskal() {
  mstEdges = [];
  stepIndex = 0;
  sortedEdges = edges.slice().sort((a, b) => a[2] - b[2]);
  ds = new DisjointSet(nodes.map(n => n.label));
  infoDiv.textContent = 'Starting Kruskal\'s Algorithm...';
  animateStep();
}

function animateStep() {
  if (stepIndex >= sortedEdges.length) {
    infoDiv.textContent = 'Kruskal\'s Algorithm completed. MST edges selected: ' + mstEdges.map(e =>
    drawGraph();
    return;
  }
  const [u, v, w] = sortedEdges[stepIndex];
  if (ds.union(u, v)) {
    mstEdges.push([u, v, w]);
    infoDiv.textContent = `Adding edge ${u} - ${v} (weight: ${w}) to MST`;
  } else {
    infoDiv.textContent = `Skipping edge ${u} - ${v} (weight: ${w}) - would form a cycle`;
  }
  drawGraph();
  stepIndex++;
  setTimeout(animateStep, 1200);
}

function drawGraph() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  // Draw all edges (light gray)
  ctx.strokeStyle = '#ccc';
  ctx.lineWidth = 2;
  for (const [from, to, w] of edges) {
    const fromNode = nodes.find(n => n.label === from);
    const toNode = nodes.find(n => n.label === to);
    ctx.beginPath();
    ctx.moveTo(fromNode.x, fromNode.y);
    ctx.lineTo(toNode.x, toNode.y);
    ctx.stroke();

    // Draw weight
    const midX = (fromNode.x + toNode.x) / 2;
    const midY = (fromNode.y + toNode.y) / 2;
    ctx.fillStyle = '#555';
```

```
      ctx.font = '14px Arial';
      ctx.fillText(w, midX + 5, midY - 5);
    }

    // Draw MST edges (green)
    ctx.strokeStyle = '#27ae60';
    ctx.lineWidth = 5;
    for (const [from, to] of mstEdges) {
      const fromNode = nodes.find(n => n.label === from);
      const toNode = nodes.find(n => n.label === to);
      ctx.beginPath();
      ctx.moveTo(fromNode.x, fromNode.y);
      ctx.lineTo(toNode.x, toNode.y);
      ctx.stroke();
    }

    // Draw nodes
    for (const node of nodes) {
      ctx.beginPath();
      ctx.arc(node.x, node.y, 25, 0, 2 * Math.PI);
      ctx.fillStyle = '#3498db';
      ctx.fill();
      ctx.strokeStyle = '#2980b9';
      ctx.lineWidth = 2;
      ctx.stroke();

      ctx.fillStyle = '#fff';
      ctx.font = '18px Arial';
      ctx.textAlign = 'center';
      ctx.textBaseline = 'middle';
      ctx.fillText(node.label, node.x, node.y);
    }
  }

  drawGraph();
</script>

</body>
</html>
```

16.1.2 Graph Representation

We'll use an **adjacency list** to represent the graph, and a **min-heap priority queue** to always fetch the smallest available edge.

16.1.3 JavaScript Implementation

Priority Queue (Min-Heap)

```
class MinHeap {
  constructor() {
    this.heap = [];
  }

  insert(item) {
    this.heap.push(item);
    this._bubbleUp();
  }

  extractMin() {
    if (this.heap.length <= 1) return this.heap.pop();
    const min = this.heap[0];
    this.heap[0] = this.heap.pop();
    this._sinkDown();
    return min;
  }

  _bubbleUp() {
    let idx = this.heap.length - 1;
    const element = this.heap[idx];
    while (idx > 0) {
      const parentIdx = Math.floor((idx - 1) / 2);
      if (this.heap[parentIdx].weight <= element.weight) break;
      [this.heap[idx], this.heap[parentIdx]] = [this.heap[parentIdx], this.heap[idx]];
      idx = parentIdx;
    }
  }

  _sinkDown() {
    let idx = 0;
    const length = this.heap.length;
    const element = this.heap[0];

    while (true) {
      let left = 2 * idx + 1;
      let right = 2 * idx + 2;
      let smallest = idx;

      if (left < length && this.heap[left].weight < this.heap[smallest].weight) smallest = left;
      if (right < length && this.heap[right].weight < this.heap[smallest].weight) smallest = right;

      if (smallest === idx) break;

      [this.heap[idx], this.heap[smallest]] = [this.heap[smallest], this.heap[idx]];
      idx = smallest;
    }
  }

  isEmpty() {
    return this.heap.length === 0;
  }
}
```

Prims Algorithm on a Graph

```
class Graph {
  constructor() {
    this.adjList = new Map();
  }

  addEdge(u, v, weight) {
    if (!this.adjList.has(u)) this.adjList.set(u, []);
    if (!this.adjList.has(v)) this.adjList.set(v, []);
    this.adjList.get(u).push({ node: v, weight });
    this.adjList.get(v).push({ node: u, weight }); // Undirected graph
  }

  primMST(start) {
    const visited = new Set();
    const mst = [];
    const heap = new MinHeap();

    visited.add(start);
    for (let edge of this.adjList.get(start)) {
      heap.insert({ from: start, to: edge.node, weight: edge.weight });
    }

    while (!heap.isEmpty()) {
      const { from, to, weight } = heap.extractMin();
      if (visited.has(to)) continue;

      visited.add(to);
      mst.push({ from, to, weight });

      for (let neighbor of this.adjList.get(to)) {
        if (!visited.has(neighbor.node)) {
          heap.insert({ from: to, to: neighbor.node, weight: neighbor.weight });
        }
      }
    }

    return mst;
  }
}
```

Example Usage

```
const g = new Graph();
g.addEdge('A', 'B', 3);
g.addEdge('A', 'D', 1);
g.addEdge('B', 'D', 3);
g.addEdge('B', 'C', 1);
g.addEdge('C', 'D', 1);
g.addEdge('C', 'E', 5);
g.addEdge('D', 'E', 6);

const mst = g.primMST('A');
console.log(mst);
```

16.1.4 Visual Step Example

```
Vertices: A, B, C, D, E
Start at A → choose edge with weight 1 (A-D)
From D → choose edge D-C (1)
From C → choose edge C-B (1)
From B → skip D (already visited), then done
```

Final MST edges: A-D, D-C, C-B — total weight = 3

16.1.5 Time Complexity

- **With min-heap + adjacency list**: O(E log V)
- **Without heap**: $O(V^2)$

Where:

- `V` = number of vertices
- `E` = number of edges

16.1.6 Real-World Applications

- **Network Design**: Build efficient communication, electrical, or road networks with minimal cost.
- **Image Processing**: Region grouping and segmentation.
- **Clustering Algorithms**: Like single-linkage clustering in data science.
- **Approximation Algorithms**: Used as part of heuristics in NP-hard problems.

16.1.7 Summary

Prim's algorithm efficiently builds a **Minimum Spanning Tree** by incrementally expanding the frontier with the smallest possible edge. Its greedy nature ensures optimality in undirected graphs. With a solid JavaScript implementation using priority queues, it's practical for frontend visualizations, backend route planning, or any application needing cost-efficient connectivity.

16.2 Kruskal's Algorithm

Kruskal's algorithm is a classic greedy algorithm used to find a **Minimum Spanning Tree (MST)** of a connected, undirected, weighted graph. It takes a different approach from Prim's algorithm. Instead of growing a tree from a starting vertex, Kruskal's algorithm **sorts all edges by weight** and adds them one by one to the MST, **only if they don't form a cycle**.

To efficiently detect cycles, Kruskal's algorithm relies on a **Union-Find** (also known as Disjoint Set Union) data structure.

16.2.1 How Kruskals Algorithm Works

1. Sort all edges in **ascending order** based on their weights.
2. Initialize a Union-Find data structure where each node is in its own set.
3. For each edge:
    - If the two vertices of the edge belong to **different sets**, include the edge in the MST and **union** their sets.
    - If the two vertices are in the **same set**, skip the edge (it would form a cycle).
4. Repeat until the MST has `V - 1` edges, where `V` is the number of vertices.

Run the following code in browser to see the demo:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>Kruskal's Algorithm Visualization</title>
<style>
  body { font-family: sans-serif; text-align: center; padding: 20px; }
  canvas { border: 1px solid #ccc; background: #f9f9f9; display: block; margin: 0 auto; }
  button { font-size: 16px; padding: 8px 12px; margin: 10px; }
  #info { font-family: monospace; margin-top: 10px; }
</style>
</head>
<body>

<h2>Kruskal's Algorithm Visualization</h2>
<button onclick="startKruskal()">Start Kruskal</button>
<canvas id="canvas" width="600" height="600"></canvas>
<div id="info"></div>

<script>
  const canvas = document.getElementById("canvas");
  const ctx = canvas.getContext("2d");
  const infoDiv = document.getElementById("info");
```

```
// Graph nodes
const nodes = [
  { label: 'A', x: 100, y: 100 },
  { label: 'B', x: 300, y: 80 },
  { label: 'C', x: 500, y: 120 },
  { label: 'D', x: 150, y: 300 },
  { label: 'E', x: 350, y: 320 },
  { label: 'F', x: 550, y: 320 },
  { label: 'G', x: 400, y: 450 },
];

// Undirected weighted edges: [from, to, weight]
const edges = [
  ['A', 'B', 7],
  ['A', 'D', 5],
  ['B', 'C', 8],
  ['B', 'D', 9],
  ['B', 'E', 7],
  ['C', 'E', 5],
  ['D', 'E', 15],
  ['D', 'G', 6],
  ['E', 'F', 8],
  ['E', 'G', 9],
  ['F', 'G', 11],
];

// Union-Find (Disjoint Set) for cycle detection
class DisjointSet {
  constructor(elements) {
    this.parent = {};
    this.rank = {};
    for (const el of elements) {
      this.parent[el] = el;
      this.rank[el] = 0;
    }
  }
  find(u) {
    if (this.parent[u] !== u) {
      this.parent[u] = this.find(this.parent[u]);
    }
    return this.parent[u];
  }
  union(u, v) {
    const rootU = this.find(u);
    const rootV = this.find(v);
    if (rootU === rootV) return false;
    if (this.rank[rootU] < this.rank[rootV]) {
      this.parent[rootU] = rootV;
    } else if (this.rank[rootV] < this.rank[rootU]) {
      this.parent[rootV] = rootU;
    } else {
      this.parent[rootV] = rootU;
      this.rank[rootU]++;
    }
    return true;
  }
}
```

```
let mstEdges = [];
let stepIndex = 0;
let sortedEdges = [];
let ds;

function startKruskal() {
  mstEdges = [];
  stepIndex = 0;
  sortedEdges = edges.slice().sort((a, b) => a[2] - b[2]);
  ds = new DisjointSet(nodes.map(n => n.label));
  infoDiv.textContent = 'Starting Kruskal's Algorithm...';
  animateStep();
}

function animateStep() {
  if (stepIndex >= sortedEdges.length) {
    infoDiv.textContent = 'Kruskal's Algorithm completed. MST edges: ' + mstEdges.map(e => `${e[0]}-${
    drawGraph();
    return;
  }
  const [u, v, w] = sortedEdges[stepIndex];
  if (ds.union(u, v)) {
    mstEdges.push([u, v, w]);
    infoDiv.textContent = `Adding edge ${u} - ${v} (weight: ${w}) to MST`;
  } else {
    infoDiv.textContent = `Skipping edge ${u} - ${v} (weight: ${w}) - would form cycle`;
  }
  drawGraph();
  stepIndex++;
  setTimeout(animateStep, 1200);
}

function drawGraph() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  // Draw all edges (light gray)
  ctx.strokeStyle = '#ccc';
  ctx.lineWidth = 2;
  for (const [from, to, w] of edges) {
    const fromNode = nodes.find(n => n.label === from);
    const toNode = nodes.find(n => n.label === to);
    ctx.beginPath();
    ctx.moveTo(fromNode.x, fromNode.y);
    ctx.lineTo(toNode.x, toNode.y);
    ctx.stroke();

    // Edge weight
    const midX = (fromNode.x + toNode.x) / 2;
    const midY = (fromNode.y + toNode.y) / 2;
    ctx.fillStyle = '#555';
    ctx.font = '14px Arial';
    ctx.fillText(w, midX + 5, midY - 5);
  }

  // Draw MST edges (green & thick)
  ctx.strokeStyle = '#27ae60';
  ctx.lineWidth = 5;
  for (const [from, to] of mstEdges) {
```

```
      const fromNode = nodes.find(n => n.label === from);
      const toNode = nodes.find(n => n.label === to);
      ctx.beginPath();
      ctx.moveTo(fromNode.x, fromNode.y);
      ctx.lineTo(toNode.x, toNode.y);
      ctx.stroke();
    }

    // Draw nodes
    for (const node of nodes) {
      ctx.beginPath();
      ctx.arc(node.x, node.y, 25, 0, 2 * Math.PI);
      ctx.fillStyle = '#3498db';
      ctx.fill();
      ctx.strokeStyle = '#2980b9';
      ctx.lineWidth = 2;
      ctx.stroke();

      ctx.fillStyle = '#fff';
      ctx.font = '18px Arial';
      ctx.textAlign = 'center';
      ctx.textBaseline = 'middle';
      ctx.fillText(node.label, node.x, node.y);
    }
  }

  drawGraph();
</script>

</body>
</html>
```

16.2.2 Union-Find Data Structure in JavaScript

```
class UnionFind {
  constructor(elements) {
    this.parent = {};
    this.rank = {};

    for (let el of elements) {
      this.parent[el] = el;
      this.rank[el] = 0;
    }
  }

  find(x) {
    if (this.parent[x] !== x) {
      this.parent[x] = this.find(this.parent[x]); // Path compression
    }
    return this.parent[x];
  }

  union(x, y) {
```

```
    const rootX = this.find(x);
    const rootY = this.find(y);

    if (rootX === rootY) return false; // Cycle detected

    // Union by rank
    if (this.rank[rootX] < this.rank[rootY]) {
      this.parent[rootX] = rootY;
    } else if (this.rank[rootX] > this.rank[rootY]) {
      this.parent[rootY] = rootX;
    } else {
      this.parent[rootY] = rootX;
      this.rank[rootX]++;
    }

    return true;
  }
}
```

16.2.3 Kruskals Algorithm in JavaScript

```
function kruskalMST(vertices, edges) {
  const uf = new UnionFind(vertices);
  const mst = [];

  // Sort edges by weight
  edges.sort((a, b) => a.weight - b.weight);

  for (let { from, to, weight } of edges) {
    if (uf.union(from, to)) {
      mst.push({ from, to, weight });
    }
    if (mst.length === vertices.length - 1) break;
  }

  return mst;
}
```

16.2.4 Example Usage

```
const vertices = ['A', 'B', 'C', 'D', 'E'];
const edges = [
  { from: 'A', to: 'B', weight: 3 },
  { from: 'A', to: 'D', weight: 1 },
  { from: 'B', to: 'D', weight: 3 },
  { from: 'B', to: 'C', weight: 1 },
  { from: 'C', to: 'D', weight: 1 },
  { from: 'C', to: 'E', weight: 5 },
  { from: 'D', to: 'E', weight: 6 }
```

```
];

const mst = kruskalMST(vertices, edges);
console.log(mst);
```

This will output the edges of the minimum spanning tree, such as:

```
[
  { from: 'A', to: 'D', weight: 1 },
  { from: 'B', to: 'C', weight: 1 },
  { from: 'C', to: 'D', weight: 1 },
  { from: 'C', to: 'E', weight: 5 }
]
```

16.2.5 Cycle Detection

Kruskal's algorithm avoids cycles by using Union-Find:

- Before adding an edge, it checks if both nodes belong to the same set.
- If they do, adding the edge would create a cycle, so it is skipped.
- If they don't, their sets are merged, and the edge is added.

16.2.6 Time Complexity

- **Sorting edges**: O(E log E)
- **Union-Find operations**: near O(1) per operation (amortized with path compression)
- **Total time**: **O(E log E)**

Where:

- `E` = number of edges
- `V` = number of vertices

16.2.7 Kruskal vs. Prim

Feature	Kruskal's Algorithm	Prim's Algorithm
Graph type	Works well with **sparse** graphs	Best with **dense** graphs
Data structure	Union-Find (Disjoint Set)	Priority Queue (Min-Heap)
Edge selection	Global edge sort	Local (adjacent) edge search
Cycle detection	Union-Find	Avoids via visited set

Feature	Kruskal's Algorithm	Prim's Algorithm

16.2.8 Real-World Use Cases

- **Network design**: Building low-cost road, electrical, or communication networks.
- **Clustering algorithms**: In unsupervised learning, MST can help group data.
- **Approximate solutions**: In routing problems like the traveling salesman, MSTs help build heuristics.
- **Image segmentation**: Graph-based methods in computer vision use MSTs to group pixels.

16.2.9 Summary

Kruskal's algorithm offers a simple, greedy solution for finding MSTs, particularly effective on **sparse graphs**. Its reliance on **edge sorting** and **Union-Find** ensures both correctness and performance. With a clean JavaScript implementation, it's a must-know tool for solving graph optimization problems in frontend tools, simulations, and network design.

16.3 Real-World Applications

Minimum Spanning Tree (MST) algorithms like **Prim's** and **Kruskal's** are more than academic exercises — they power real-world systems that need efficient connectivity, cost minimization, and clustering. From designing physical infrastructure to solving abstract data problems, MSTs help make intelligent decisions in systems where connections have weights or costs.

16.3.1 Network Cabling and Infrastructure Design

One of the most direct applications of MSTs is in the design of **communication networks** — whether it's laying out fiber-optic cables, electric power lines, or roadways.

Example:

A telecom company wants to connect five cities with fiber. Each potential connection (edge) has a cost based on distance and terrain. Using **Kruskal's algorithm**, the company can

build a network that connects all cities **with the least total cost** and **no redundant loops**, which would waste resources.

```
// Imagine edges: [ {from: 'A', to: 'B', weight: 10}, ... ]
// Use kruskalMST(vertices, edges) to get the cheapest cable layout
```

In JavaScript simulations or back-end planning tools, such MST algorithms can evaluate multiple network layout scenarios and pick the optimal one.

16.3.2 Clustering in Machine Learning

In **unsupervised learning**, MSTs are used for clustering data points. The idea is to construct an MST from the dataset (viewed as a complete graph with edge weights based on similarity or distance), then **remove the longest edges** to break it into clusters.

This method avoids assumptions about cluster shapes and can work well with non-convex data.

Example:

In a visual clustering demo built with JavaScript and D3.js, you can:

1. Compute distances between data points.
2. Build an MST using Prim's algorithm.
3. Delete `k-1` heaviest edges to form `k` clusters. This is known as **single-linkage clustering**.

16.3.3 Image Segmentation

In **computer vision**, MSTs are used to segment images by grouping similar pixels.

Each pixel is treated as a node, and the edge weight between two pixels represents color or texture difference. An MST groups nearby similar pixels and helps identify object boundaries.

Example:

In a JavaScript-based image processing app (e.g., with Canvas API), one could:

- Convert image into a graph of pixels.
- Use MST to partition regions with low color variance. This technique underlies algorithms like **Felzenszwalb-Huttenlocher segmentation**.

16.3.4 Traffic and Utility Networks

City planners and utility companies use MSTs to design:

- **Traffic sensor networks**
- **Water supply pipelines**
- **Power grids**

These networks must connect all points efficiently while minimizing construction costs. MSTs help eliminate redundant or circular paths that do not add value.

A JavaScript simulation might visualize nodes (homes, sensors) and edges (pipes, wires), then apply Prim's algorithm to simulate real-world infrastructure decisions.

16.3.5 Algorithm Choice in Practice

- **Kruskal's Algorithm** is preferred when the graph has **many sparse edges** or comes as a pre-sorted edge list (common in file-based input).
- **Prim's Algorithm** works well with **dense graphs** and when you already have an adjacency list (as you might in JavaScript apps using object maps).

In web environments, libraries like D3.js can help **visualize MST construction step-by-step**, enhancing educational or planning tools.

16.3.6 Summary

Minimum Spanning Trees are essential in a wide array of domains: from minimizing infrastructure costs to discovering meaningful data groupings and improving digital image analysis. By applying Kruskal's or Prim's algorithm — both of which can be implemented effectively in JavaScript — developers and analysts can solve real-world problems efficiently. MSTs demonstrate how core algorithmic principles directly power modern systems in networking, AI, and visualization.

Chapter 17.

Principles of Dynamic Programming

1. Overlapping Subproblems and Memoization
2. Bottom-Up vs Top-Down Approaches

17 Principles of Dynamic Programming

17.1 Overlapping Subproblems and Memoization

Dynamic programming (DP) is a powerful technique for solving problems with **overlapping subproblems** and **optimal substructure**. Understanding these concepts is key to leveraging DP effectively.

17.1.1 Overlapping Subproblems

A problem exhibits **overlapping subproblems** when it can be broken down into smaller subproblems that are **repeatedly solved multiple times** during a naive recursive process.

Consider the classic example: calculating Fibonacci numbers.

The Fibonacci sequence is defined as:

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

The naive recursive function directly follows this definition:

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

While elegant, this approach has a major problem: **redundant calculations**. For example, `fib(5)` calculates `fib(3)` twice, `fib(2)` three times, and so on, causing exponential time complexity.

17.1.2 Memoization: The Top-Down Optimization

To avoid repeated work, **memoization** stores (or "memoizes") the results of subproblems so that future calls can reuse them without recomputation.

Here's a memoized version of Fibonacci in JavaScript:

```
function fibMemo(n, memo = {}) {
  if (n <= 1) return n;
  if (memo[n]) return memo[n];

  memo[n] = fibMemo(n - 1, memo) + fibMemo(n - 2, memo);
  return memo[n];
}
```

How it works:

- The `memo` object caches computed Fibonacci values keyed by `n`.
- Before recursive calls, the function checks if the result already exists.
- If yes, it immediately returns the cached result, skipping recursion.
- Otherwise, it computes and caches the result.

This simple change transforms exponential time complexity to **linear time O(n)**, because each subproblem is solved only once.

17.1.3 Comparing Performance

```
console.time('Naive');
console.log(fib(40));  // Slow, exponential time
console.timeEnd('Naive');

console.time('Memoized');
console.log(fibMemo(40)); // Fast, linear time
console.timeEnd('Memoized');
```

On most machines, the naive approach for `fib(40)` might take seconds, while the memoized approach completes almost instantly.

17.1.4 When to Use Memoization

Memoization is ideal for problems with:

- **Recursive structure:** Solutions defined in terms of smaller subproblems.
- **Overlapping subproblems:** Subproblems that appear multiple times.
- **Deterministic results:** Same inputs always yield same outputs.

Typical examples include:

- Fibonacci numbers
- Factorials (though trivial with iteration)
- Calculating binomial coefficients
- Edit distance between strings
- Optimal game strategies

17.1.5 Summary

Overlapping subproblems cause exponential work in naive recursive algorithms due to repeated calculations. **Memoization** is a top-down dynamic programming technique that caches these results to avoid recomputation, yielding significant performance improvements.

In JavaScript, memoization is straightforward to implement using objects or maps, making it an accessible and practical tool in many algorithmic scenarios. Understanding and applying memoization lays the foundation for more advanced dynamic programming techniques explored later in this book.

17.2 Bottom-Up vs Top-Down Approaches

Dynamic Programming (DP) can be implemented using two primary strategies: **Top-Down (with Memoization)** and **Bottom-Up (with Tabulation)**. Both approaches solve problems by breaking them into subproblems and reusing previously computed results, but they differ in how and when these computations are performed.

17.2.1 Top-Down Approach: Memoization

The **top-down** approach starts solving the main problem and recursively breaks it into smaller subproblems. Each subproblem's result is cached (memoized) to avoid redundant computations.

- **Implementation Style:** Recursive function calls with caching.
- **Execution:** On-demand subproblem solving, only compute what's needed.
- **Advantages:**
  - Intuitive and closely follows problem definition.
  - Easier to write initially, especially if the problem is naturally recursive.
  - Can be more efficient if only a subset of subproblems is required.
- **Challenges:**
  - Stack overflow risk with deep recursion.
  - Slight overhead due to recursive calls.
  - Debugging recursive code can be tricky.

Example: Fibonacci (Top-Down)

```
function fibMemo(n, memo = {}) {
  if (n <= 1) return n;
  if (memo[n]) return memo[n];

  memo[n] = fibMemo(n - 1, memo) + fibMemo(n - 2, memo);
  return memo[n];
}
```

17.2.2 Bottom-Up Approach: Tabulation

The **bottom-up** method solves all smaller subproblems first, then combines them to solve bigger subproblems iteratively. It builds a table (usually an array) to store intermediate results.

- **Implementation Style:** Iterative loops filling a table from smallest to largest subproblem.
- **Execution:** All subproblems solved systematically regardless of necessity.
- **Advantages:**
  - Avoids recursion and its associated stack overhead.
  - Often faster in practice due to simple iteration.
  - Easier to debug since code flow is straightforward.
- **Challenges:**
  - May compute unnecessary subproblems.
  - Requires careful planning of the order of computation.

Example: Fibonacci (Bottom-Up)

```
function fibTab(n) {
  if (n <= 1) return n;

  const dp = [0, 1];
  for (let i = 2; i <= n; i++) {
    dp[i] = dp[i - 1] + dp[i - 2];
  }
  return dp[n];
}
```

17.2.3 Side-by-Side Comparison

Aspect	Top-Down (Memoization)	Bottom-Up (Tabulation)
Code style	Recursive with cache	Iterative with table
Execution order	Demand-driven, solves subproblems as needed	Systematic, solves all subproblems
Space complexity	Depends on recursion depth + cache size	Table size (usually same as cache)
Risk	Stack overflow on deep recursion	Generally safer (no recursion)
Debugging	Recursive flow can be harder to trace	Easier due to linear iteration
Performance	Sometimes slower due to recursion overhead	Often faster due to iteration and cache locality

17.2.4 Example: Longest Common Subsequence (LCS)

LCS finds the longest subsequence common to two strings.

Top-Down Memoization:

```
function lcsMemo(s1, s2, i = 0, j = 0, memo = {}) {
  const key = `${i},${j}`;
  if (key in memo) return memo[key];
  if (i === s1.length || j === s2.length) return 0;

  if (s1[i] === s2[j]) {
    memo[key] = 1 + lcsMemo(s1, s2, i + 1, j + 1, memo);
  } else {
    memo[key] = Math.max(
      lcsMemo(s1, s2, i + 1, j, memo),
      lcsMemo(s1, s2, i, j + 1, memo)
    );
  }
  return memo[key];
}
```

Bottom-Up Tabulation:

```
function lcsTab(s1, s2) {
  const m = s1.length, n = s2.length;
  const dp = Array.from({ length: m + 1 }, () => Array(n + 1).fill(0));

  for (let i = 1; i <= m; i++) {
    for (let j = 1; j <= n; j++) {
      if (s1[i - 1] === s2[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]);
      }
    }
  }
  return dp[m][n];
}
```

17.2.5 Practical Tips for JavaScript Developers

- **Debugging:** Bottom-up code is easier to step through with a debugger, since recursion can complicate call stacks.
- **Performance:** Bottom-up may be faster for very large input sizes due to iteration overhead reduction.
- **Memory:** Both approaches typically use similar amounts of memory, but bottom-up can sometimes optimize space by reusing rows or columns.
- **Stack Limitations:** JavaScript engines have limited stack size; prefer bottom-up if deep recursion is expected.
- **Use Memoization for Clarity:** For complex recursive problems, memoization keeps code cleaner and easier to write initially.

17.2.6 Summary

Both top-down and bottom-up dynamic programming methods aim to optimize recursive problems by avoiding repeated computations. The **top-down approach** uses recursion with memoization to cache results, while the **bottom-up approach** iteratively builds solutions from the smallest subproblems up. Choosing between them depends on problem structure, input size, and debugging needs. JavaScript developers benefit from understanding both styles to write efficient, maintainable DP solutions.

Chapter 18.

Principles of Dynamic Programming Classic Problems

1. Fibonacci Variants
2. Longest Common Subsequence (LCS)
3. Knapsack Problem
4. Grid-Based Pathfinding

18 Principles of Dynamic Programming Classic Problems

18.1 Fibonacci Variants

The Fibonacci sequence is a classic example often used to introduce dynamic programming (DP). Its basic form counts numbers defined by the sum of the two preceding ones. But beyond this simple definition, there are many interesting **variants** that share the same underlying principles of overlapping subproblems and optimal substructure, making them perfect DP candidates.

18.1.1 Classic Fibonacci Recap

The classic Fibonacci sequence is defined as:

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

A **naive recursive** implementation in JavaScript is straightforward but inefficient due to repeated calculations:

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

This approach has exponential time complexity $O(2^n)$.

18.1.2 Memoized Fibonacci

Applying **memoization** caches intermediate results to avoid redundant work:

```
function fibMemo(n, memo = {}) {
  if (n <= 1) return n;
  if (memo[n]) return memo[n];

  memo[n] = fibMemo(n - 1, memo) + fibMemo(n - 2, memo);
  return memo[n];
}
```

This reduces time complexity to linear $O(n)$, since each subproblem is solved once.

18.1.3 Iterative Fibonacci (Bottom-Up)

The **bottom-up** approach iteratively builds the sequence:

```
function fibIter(n) {
  if (n <= 1) return n;

  let prev = 0, curr = 1;
  for (let i = 2; i <= n; i++) {
    [prev, curr] = [curr, prev + curr];
  }
  return curr;
}
```

This approach also runs in $O(n)$ but uses constant space $O(1)$, making it efficient for large inputs.

18.1.4 Variant: Counting Ways to Climb Stairs

A popular Fibonacci variant is the problem of counting the number of ways to climb a staircase where you can take either **1 or 2 steps at a time**.

- If the staircase has n steps, let $W(n)$ be the number of ways to reach the top.
- Base cases:

$$W(0) = 1 \quad \text{(one way: do nothing)}$$

$$W(1) = 1 \quad \text{(one step)}$$

- Recurrence:

$$W(n) = W(n-1) + W(n-2)$$

This mirrors Fibonacci logic but starts with different base cases.

Recursive solution:

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

Memoized:

```
function climbStairsMemo(n, memo = {}) {
  if (n <= 1) return 1;
```

```
  if (memo[n]) return memo[n];

  memo[n] = climbStairsMemo(n - 1, memo) + climbStairsMemo(n - 2, memo);
  return memo[n];
}
```

Iterative:

```
function climbStairsIter(n) {
  if (n <= 1) return 1;

  let oneStepBefore = 1, twoStepsBefore = 1, allWays = 0;
  for (let i = 2; i <= n; i++) {
    allWays = oneStepBefore + twoStepsBefore;
    twoStepsBefore = oneStepBefore;
    oneStepBefore = allWays;
  }
  return allWays;
}
```

18.1.5 Underlying DP Principles

Each variant demonstrates key DP ideas:

- **Overlapping subproblems:** Many calls recompute the same values without caching.
- **Optimal substructure:** The solution to a problem builds from solutions to smaller subproblems.
- **Memoization vs Tabulation:** Memoization caches results top-down; tabulation builds from the bottom-up.
- **Performance improvements:** Memoization and iteration change exponential recursive calls to linear-time solutions.

18.1.6 Summary

Fibonacci variants like the staircase problem highlight the power of dynamic programming in optimizing recursive problems with overlapping subproblems. Whether using naive recursion, memoization, or bottom-up iteration, understanding these implementations prepares you for tackling more complex DP challenges with confidence and efficiency in JavaScript.

18.2 Longest Common Subsequence (LCS)

The **Longest Common Subsequence (LCS)** problem is a fundamental challenge in string comparison. Given two strings, the goal is to find the length of the longest subsequence common to both strings. A **subsequence** differs from a substring because its characters need not be contiguous, but they must appear in the same relative order.

18.2.1 Why LCS Matters

LCS is widely used in:

- **Diff tools:** To highlight differences between text files by identifying common parts.
- **DNA sequence analysis:** To compare genetic sequences and detect similarities.
- **Version control:** To merge changes and detect conflicts.
- **Natural language processing:** For text similarity and alignment.

18.2.2 Recursive Formulation

Given two strings, `X` and `Y`, define:

- $LCS(i, j)$ = length of the LCS of prefixes $X[0..i-1]$ and $Y[0..j-1]$.

The recursive relation is:

$$LCS(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ 1 + LCS(i-1, j-1) & \text{if } X[i-1] = Y[j-1] \\ \max(LCS(i-1, j), LCS(i, j-1)) & \text{if } X[i-1] \neq Y[j-1] \end{cases}$$

This means:

- If either string is empty, LCS length is 0.
- If the current characters match, increment the LCS length by 1 and move diagonally.
- If they don't match, take the maximum LCS by either moving left or up in the table.

18.2.3 Naive Recursive Implementation

A straightforward recursive solution closely follows the formula but suffers from overlapping subproblems, leading to exponential time complexity:

```
function lcsRecursive(X, Y, i = X.length, j = Y.length) {
  if (i === 0 || j === 0) return 0;
```

```
  if (X[i - 1] === Y[j - 1]) {
    return 1 + lcsRecursive(X, Y, i - 1, j - 1);
  } else {
    return Math.max(
      lcsRecursive(X, Y, i - 1, j),
      lcsRecursive(X, Y, i, j - 1)
    );
  }
}
```

18.2.4 Dynamic Programming: Bottom-Up Approach

To optimize, use a **2D table** to store solutions for all prefix pairs, avoiding redundant calculations.

- Create a table `dp` with dimensions `(m+1) x (n+1)` where `m = X.length` and `n = Y.length`.
- Initialize the first row and column with zeros (base cases).
- Fill the table row by row:
  - If characters match, `dp[i][j] = 1 + dp[i-1][j-1]`.
  - Otherwise, `dp[i][j] = max(dp[i-1][j], dp[i][j-1])`.

At the end, `dp[m][n]` contains the length of the LCS.

18.2.5 JavaScript Implementation with Comments

```
function lcs(X, Y) {
  const m = X.length;
  const n = Y.length;

  // Initialize DP table with zeros
  const dp = Array.from({ length: m + 1 }, () => Array(n + 1).fill(0));

  // Build the table from bottom up
  for (let i = 1; i <= m; i++) {
    for (let j = 1; j <= n; j++) {
      if (X[i - 1] === Y[j - 1]) {
        dp[i][j] = 1 + dp[i - 1][j - 1];
      } else {
        dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
      }
    }
  }

  // dp[m][n] is the length of LCS for X and Y
```

```
  return dp[m][n];
}
```

18.2.6 Reconstructing the LCS (Optional)

To find the actual subsequence, backtrack from `dp[m][n]`:

```
function getLCS(X, Y) {
  const m = X.length;
  const n = Y.length;
  const dp = Array.from({ length: m + 1 }, () => Array(n + 1).fill(0));

  // Fill dp table
  for (let i = 1; i <= m; i++) {
    for (let j = 1; j <= n; j++) {
      if (X[i - 1] === Y[j - 1]) {
        dp[i][j] = 1 + dp[i - 1][j - 1];
      } else {
        dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
      }
    }
  }

  // Backtrack to find the LCS string
  let i = m, j = n;
  let lcsStr = '';

  while (i > 0 && j > 0) {
    if (X[i - 1] === Y[j - 1]) {
      lcsStr = X[i - 1] + lcsStr; // prepend matching char
      i--;
      j--;
    } else if (dp[i - 1][j] > dp[i][j - 1]) {
      i--;
    } else {
      j--;
    }
  }

  return lcsStr;
}
```

18.2.7 Practical Applications

- **Diff Tools:** Tools like Git's `diff` use LCS algorithms to compare file versions, showing insertions, deletions, and modifications.
- **Bioinformatics:** LCS is used to compare DNA, RNA, or protein sequences, aiding in gene identification and evolutionary analysis.
- **Text Processing:** LCS helps detect plagiarism and find similar documents or sentences.

- **Version Control Systems:** Understanding how code changes between commits relies on LCS-based algorithms.

18.2.8 Summary

The Longest Common Subsequence problem exemplifies dynamic programming's power to optimize recursive solutions by using tabulation and memoization. Its applications span computer science, biology, and software engineering, making it a foundational algorithm to master. Implementing LCS in JavaScript through a 2D DP table not only improves efficiency but also provides a practical way to compare sequences effectively.

18.3 Knapsack Problem

The **0/1 Knapsack Problem** is one of the most famous and fundamental problems in dynamic programming. It involves decision-making under constraints and showcases how DP can systematically explore subproblems to arrive at an optimal solution.

18.3.1 Problem Statement

Given:

- `n` items, each with:
  - a **weight** `w[i]`
  - a **value** `v[i]`
- A **knapsack capacity** `W`

Goal:

> Choose a subset of items such that the **total value is maximized** without exceeding the weight limit `W`.

Each item can either be **included once or not at all** — hence the name **0/1 Knapsack**.

18.3.2 Recursive Relation

We define `dp[i][w]` as the maximum value that can be obtained using the first `i` items and capacity `w`.

The recurrence relation is:

$$dp[i][w] = \begin{cases} dp[i-1][w] & \text{if } w_i > w \quad \text{(skip item)} \\ \max(dp[i-1][w],\ v_i + dp[i-1][w-w_i]) & \text{otherwise} \end{cases}$$

This means:

- If the current item is too heavy, we can't include it.
- Otherwise, we choose the better of:
  - skipping the item
  - including the item and adding its value

18.3.3 Bottom-Up Tabulation (JavaScript)

Let's write a bottom-up DP solution with a 2D table.

```
function knapsack(weights, values, capacity) {
  const n = weights.length;
  const dp = Array.from({ length: n + 1 }, () =>
    Array(capacity + 1).fill(0)
  );

  for (let i = 1; i <= n; i++) {
    for (let w = 0; w <= capacity; w++) {
      if (weights[i - 1] > w) {
        // Can't include item i-1
        dp[i][w] = dp[i - 1][w];
      } else {
        // Max of including or excluding the item
        dp[i][w] = Math.max(
          dp[i - 1][w],
          values[i - 1] + dp[i - 1][w - weights[i - 1]]
        );
      }
    }
  }

  return dp[n][capacity]; // Max value achievable
}
```

Example:

```
const weights = [2, 1, 3];
const values = [4, 2, 5];
const capacity = 4;

console.log(knapsack(weights, values, capacity)); // Output: 7
```

We can choose items 1 (weight 1, value 2) and 3 (weight 3, value 5) for a total value of 7.

18.3.4 Space Optimization

The DP table can be reduced to a 1D array, since each row only depends on the previous one. Iterate **backwards** over capacity to avoid overwriting results:

```
function knapsackOptimized(weights, values, capacity) {
  const dp = Array(capacity + 1).fill(0);
  const n = weights.length;

  for (let i = 0; i < n; i++) {
    for (let w = capacity; w >= weights[i]; w--) {
      dp[w] = Math.max(dp[w], values[i] + dp[w - weights[i]]);
    }
  }

  return dp[capacity];
}
```

This reduces space complexity from $O(nW)$ to $O(W)$, which can be a major win in constrained environments.

18.3.5 0/1 vs Fractional Knapsack

The **Fractional Knapsack Problem** allows you to take partial items (e.g., 0.5 of an item). This variant can be solved greedily by selecting items based on their **value-to-weight ratio** and filling the knapsack accordingly.

```
function fractionalKnapsack(weights, values, capacity) {
  const items = values.map((v, i) => ({
    value: v,
    weight: weights[i],
    ratio: v / weights[i],
  }));

  // Sort by descending value/weight ratio
  items.sort((a, b) => b.ratio - a.ratio);

  let totalValue = 0;
  for (const item of items) {
    if (capacity >= item.weight) {
      totalValue += item.value;
      capacity -= item.weight;
    } else {
      totalValue += item.ratio * capacity;
      break;
    }
  }

  return totalValue;
}
```

Time complexity: $O(n \log n)$ due to sorting.

Note: Fractional knapsack doesn't apply to 0/1 problems because taking parts of items isn't allowed.

18.3.6 Optimization Tips

- Use **space-optimized 1D DP** when only the final result is needed.
- If item values or weights are small integers, use **value-based DP**.
- Use **bitmask DP** or **memoization with caching** for very constrained variations (e.g., item limits or constraints).
- For large `n` or `W`, heuristic or approximate methods (like greedy) may be preferable.

18.3.7 Real-World Applications

- **Budget allocation:** Choose a subset of projects under a budget for maximum ROI.
- **Resource management:** Decide which files to back up given storage limits.
- **Packing problems:** E-commerce or shipping scenarios where space and value matter.
- **Cryptography & security:** Optimization under size and cost constraints.

18.3.8 Summary

The 0/1 Knapsack problem exemplifies dynamic programming's strengths in solving optimization problems under constraints. Its recursive structure, combined with tabulation and space optimization, enables efficient solutions in JavaScript. Variants like fractional knapsack and real-world applications further emphasize its importance across domains such as logistics, resource planning, and finance. Understanding and implementing the knapsack problem equips developers with a vital tool for tackling complex optimization tasks.

18.4 Grid-Based Pathfinding

Grid-based pathfinding problems are a classic domain for applying dynamic programming (DP), particularly in 2D matrix scenarios. These problems simulate movement across a grid while optimizing for criteria like **path count**, **minimum cost**, or **obstacle avoidance**. They're foundational for applications in **robotics**, **game development**, and **logistics systems**.

18.4.1 Problem 1: Count Unique Paths

Statement: Given an `m x n` grid, how many unique paths exist from the top-left corner to the bottom-right corner, moving only **right** or **down**?

Approach: Bottom-Up DP

We create a 2D DP table where `dp[i][j]` represents the number of ways to reach cell `(i, j)`.

- Base case: `dp[0][j] = 1` and `dp[i][0] = 1` (only one way to reach cells in the first row or column).
- Transition: `dp[i][j] = dp[i - 1][j] + dp[i][j - 1]`.

```
function countUniquePaths(m, n) {
  const dp = Array.from({ length: m }, () => Array(n).fill(1));

  for (let i = 1; i < m; i++) {
    for (let j = 1; j < n; j++) {
      dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
    }
  }

  return dp[m - 1][n - 1];
}
```

Example:

```
countUniquePaths(3, 3); // Output: 6
```

There are 6 distinct ways to reach the bottom-right corner in a 3×3 grid.

18.4.2 Problem 2: Minimum Cost Path

Statement: Given a grid of costs, find the **minimum cost** to reach the bottom-right corner from the top-left, moving only right or down.

Approach: DP Table

Let `grid[i][j]` be the cost of cell `(i, j)`. We build a `dp[i][j]` table for minimum cumulative cost.

```
function minCostPath(grid) {
  const m = grid.length;
  const n = grid[0].length;
  const dp = Array.from({ length: m }, () => Array(n).fill(0));

  dp[0][0] = grid[0][0];

  // Initialize first row
  for (let j = 1; j < n; j++) {
    dp[0][j] = dp[0][j - 1] + grid[0][j];
```

```
  }

  // Initialize first column
  for (let i = 1; i < m; i++) {
    dp[i][0] = dp[i - 1][0] + grid[i][0];
  }

  // Fill rest of the table
  for (let i = 1; i < m; i++) {
    for (let j = 1; j < n; j++) {
      dp[i][j] = grid[i][j] + Math.min(dp[i - 1][j], dp[i][j - 1]);
    }
  }

  return dp[m - 1][n - 1];
}
```

Example:

```
minCostPath([
  [1, 3, 1],
  [1, 5, 1],
  [4, 2, 1]
]); // Output: 7
```

Path: $1 \to 3 \to 1 \to 1 \to 1 =$ total cost 7

18.4.3 Problem 3: Paths with Obstacles

Statement: Modify the unique paths problem: some grid cells contain obstacles (represented by `1`), and cannot be stepped on.

Approach:

- If a cell has an obstacle, `dp[i][j] = 0`.
- Otherwise, use the same logic as before.

```
function uniquePathsWithObstacles(grid) {
  const m = grid.length;
  const n = grid[0].length;
  const dp = Array.from({ length: m }, () => Array(n).fill(0));

  // Starting point
  if (grid[0][0] === 0) dp[0][0] = 1;

  // First column
  for (let i = 1; i < m; i++) {
    if (grid[i][0] === 0) dp[i][0] = dp[i - 1][0];
  }

  // First row
  for (let j = 1; j < n; j++) {
```

```
    if (grid[0][j] === 0) dp[0][j] = dp[0][j - 1];
  }

  // Fill the rest
  for (let i = 1; i < m; i++) {
    for (let j = 1; j < n; j++) {
      if (grid[i][j] === 0) {
        dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
      }
    }
  }

  return dp[m - 1][n - 1];
}
```

18.4.4 Real-World Use Cases

1. **Robot Navigation** A robot navigating a warehouse grid needs to avoid obstacles and find the shortest or safest route.
2. **Game Development** Grid-based maps in strategy or puzzle games often require players or AI to find optimal paths.
3. **Route Planning in Logistics** Delivery routes across a city grid with blocked roads or detours can be modeled as grid-based DP problems.

18.4.5 Optimization Tips

- Use **1D arrays** to reduce space in problems like unique paths.
- For sparse grids or many obstacles, **BFS or A*** may be better than DP.
- When reconstructing paths, maintain a **parent pointer grid**.

18.4.6 Summary

Dynamic programming offers elegant and efficient solutions to a variety of grid-based pathfinding problems — from counting all possible routes to navigating around obstacles and minimizing costs. These techniques are not just theoretical; they power practical systems like automated robots, path planning tools, and interactive games. Mastering these grid-based DP patterns enables you to tackle many spatial and decision-based challenges effectively in JavaScript.

Chapter 19.

Greedy Algorithms

1. When Greedy Works
2. Activity Selection
3. Huffman Encoding
4. Interval Scheduling

19 Greedy Algorithms

19.1 When Greedy Works

Greedy algorithms are a powerful class of algorithms that solve optimization problems by making a sequence of locally optimal choices—choices that seem best at each individual step—without revisiting or revising past decisions. The central idea is simple: at every step, pick the best available option and hope that this sequence of local decisions leads to a globally optimal solution. While this approach is intuitive and often fast, it does not always work. Understanding *when* greedy strategies yield correct results is essential for applying them effectively.

19.1.1 The Greedy Paradigm

A greedy algorithm operates in a step-by-step manner:

1. **Greedy choice**: At each stage, select the best local option available.
2. **Feasibility**: Ensure that the current choice does not violate the problem constraints.
3. **Irrevocability**: Once a choice is made, it is never changed or undone.

Because of their simplicity and efficiency, greedy algorithms typically run in linear or linearithmic time, and are easy to implement. However, their correctness depends on whether the locally optimal decisions also lead to a *globally optimal* outcome.

19.1.2 Conditions for Greedy Optimality

There are two key conditions that must be met for a greedy algorithm to work correctly:

1. **Greedy-choice property**: A globally optimal solution can be arrived at by making a series of locally optimal choices. This means that there exists an optimal solution that starts with the greedy choice.
2. **Optimal substructure**: The problem's solution can be constructed optimally from optimal solutions to its subproblems. This is also a requirement for dynamic programming, but dynamic programming allows for multiple overlapping subproblems, while greedy algorithms typically process each subproblem just once.

When both of these properties hold, a greedy algorithm can be used to solve the problem efficiently.

19.1.3 Greedy vs. Dynamic Programming

At first glance, greedy algorithms and dynamic programming appear similar because both exploit the optimal substructure property. The key difference lies in the treatment of subproblems:

- **Dynamic Programming** solves all subproblems and uses their results to build the solution. It is more cautious and revisits decisions to guarantee optimality.
- **Greedy Algorithms** make one pass and commit to choices immediately. They avoid recomputation and typically require less memory.

For example, consider the **0/1 Knapsack Problem** versus the **Fractional Knapsack Problem**. The 0/1 version requires dynamic programming, since the best choice for one item may depend on later options. In contrast, the fractional version can be solved greedily: sort items by value-to-weight ratio, and pick as much of the highest ratio items as possible. Here, both greedy-choice and optimal substructure properties hold, so the greedy approach yields the optimal solution.

19.1.4 Examples of Greedy-Appropriate Problems

Greedy algorithms are especially effective in problems involving:

- **Scheduling**: Selecting a subset of non-overlapping intervals (e.g., activity selection).
- **Compression**: Constructing optimal prefix-free codes (e.g., Huffman coding).
- **Graph traversal**: Finding minimum spanning trees (e.g., Prim's or Kruskal's algorithms).
- **Pathfinding**: Dijkstra's algorithm for shortest paths in graphs with non-negative weights.

In each of these, the problem structure allows greedy choices to accumulate into an optimal solution.

19.1.5 How to Identify Greedy Problems

To decide whether a greedy approach is appropriate:

- **Check for the greedy-choice property**: Can you always choose the best-looking option and still build a globally optimal solution?
- **Check for optimal substructure**: Can the solution be constructed from solutions to smaller instances of the same problem?
- **Test small examples**: Try greedy strategies on simple inputs. If they consistently yield the best answer, the problem may be amenable to a greedy solution.
- **Compare with dynamic programming**: If a known dynamic programming solution

exists, examine whether some subproblems are being solved redundantly. This might indicate an opportunity for a greedy simplification.

19.1.6 Conclusion

Greedy algorithms offer elegant and efficient solutions to a subset of optimization problems, provided certain structural conditions are met. They are most effective when each decision naturally leads closer to the best overall outcome. While not universally applicable, recognizing the greedy-choice property and optimal substructure can help identify when a greedy approach is both correct and advantageous.

19.2 Activity Selection

The **Activity Selection Problem** is a classic example of where a greedy algorithm produces an optimal solution. The problem involves choosing the largest possible number of activities (or intervals) that do not overlap in time. Each activity has a **start time** and an **end time**, and the goal is to select the **maximum number of non-conflicting activities** from a given list.

19.2.1 Problem Statement

Formally, you're given `n` activities, each with a start time `start[i]` and finish time `end[i]`. An activity `i` is compatible with activity `j` if their time intervals do not overlap: that is, either `end[i] <= start[j]` or `end[j] <= start[i]`.

Your task is to select the largest subset of non-overlapping activities.

19.2.2 Why Greedy Works

This problem has both:

- **Optimal substructure**: An optimal solution to the whole problem contains within it optimal solutions to subproblems.
- **Greedy-choice property**: If we always pick the activity that finishes earliest (among those that start after the last chosen activity), we can still build a globally optimal solution.

Intuitively, finishing activities early leaves more room for subsequent ones. So, by always selecting the activity with the **earliest finish time** that doesn't conflict with the ones already selected, we can maximize the number of activities.

19.2.3 Greedy Strategy

1. **Sort the activities** in increasing order of finish times.
2. **Select the first activity** (it finishes earliest).
3. **Iterate through the remaining activities**:
    - If an activity's start time is greater than or equal to the finish time of the last selected activity, select it.

This process ensures that each choice leaves as much space as possible for future selections.

19.2.4 JavaScript Implementation

Here is a simple implementation of the greedy solution in JavaScript:

```javascript
function activitySelection(activities) {
  // Sort activities by end time
  activities.sort((a, b) => a.end - b.end);

  const selected = [];
  let lastEndTime = 0;

  for (const activity of activities) {
    if (activity.start >= lastEndTime) {
      selected.push(activity);
      lastEndTime = activity.end;
    }
  }

  return selected;
}

// Example usage:
const activities = [
  { start: 1, end: 4 },
  { start: 3, end: 5 },
  { start: 0, end: 6 },
  { start: 5, end: 7 },
  { start: 8, end: 9 },
  { start: 5, end: 9 }
];

const result = activitySelection(activities);
console.log("Selected activities:", result);
```

Output:

```
Selected activities: [ { start: 1, end: 4 }, { start: 5, end: 7 }, { start: 8, end: 9 }
```

This solution has a time complexity of **O(n log n)** due to the sorting step, and **O(n)** for the selection phase.

19.2.5 Real-World Applications

The activity selection problem models many **scheduling** and **resource allocation** scenarios, such as:

- **Classroom scheduling**: Assigning lectures to a classroom so that no two lectures overlap.
- **Meeting rooms**: Scheduling the maximum number of meetings in a single room.
- **CPU task scheduling**: Running as many non-overlapping processes on a single core as possible.
- **Job interviews**: A recruiter wants to meet as many candidates as possible without overlapping interviews.

In all these cases, the greedy approach is ideal when:

- Only one resource (e.g., one room or machine) is available.
- The objective is to maximize **quantity** (e.g., number of tasks), not the **value** of each task.

19.2.6 Key Takeaways

- The activity selection problem is a fundamental example of a greedy algorithm that is provably optimal.
- The greedy strategy relies on **sorting by finish time**, then choosing non-overlapping activities.
- This approach is fast, efficient, and practical for many real-life scheduling scenarios.

By understanding the structure of this problem and why greedy decisions work here, you build intuition for applying similar strategies in more complex scheduling and optimization tasks.

19.3 Huffman Encoding

Huffman Encoding is a classic greedy algorithm used in data compression to assign variable-length codes to input characters, with shorter codes assigned to more frequent characters. The key idea is to reduce the total number of bits required to encode a message, making compression efficient without losing any information. Huffman coding is widely used in file formats like ZIP and JPEG, and in data transmission protocols.

19.3.1 The Problem

Given a set of characters and their frequencies (i.e., how often each character appears in a message), construct a **binary prefix code** that minimizes the total encoded length of the message.

- **Prefix code**: No code is a prefix of another (e.g., if `A = 0`, `B` cannot be `01`).
- **Goal**: Minimize the total cost, calculated as `frequency × code length` summed over all characters.

For example, suppose we have the characters:

Character	Frequency
A	5
B	9
C	12
D	13
E	16
F	45

We want to generate binary codes such that frequently used characters have shorter codes and less frequent ones have longer codes.

Run the following code in browser to see the demo:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>Huffman Encoding with Explanation Overlays</title>
<style>
  body {
    font-family: sans-serif;
    text-align: center;
    padding: 20px;
    background: #f4f4f4;
  }
  canvas {
```

```
    border: 1px solid #ccc;
    background: #fff;
    display: block;
    margin: 0 auto;
  }
  button {
    font-size: 16px;
    padding: 8px 12px;
    margin: 10px;
  }
  #freqTable, #codes, #explanation {
    max-width: 700px;
    margin: 10px auto;
    text-align: left;
    font-family: monospace;
    background: #fff;
    padding: 12px;
    border-radius: 6px;
    box-shadow: 0 0 8px #ccc;
    white-space: pre-wrap;
  }
  #explanation {
    font-size: 15px;
    line-height: 1.4;
    color: #222;
  }
</style>
</head>
<body>

<h2>Huffman Encoding Visualization with Detailed Explanation</h2>
<button onclick="startHuffman()">Start Huffman Encoding</button>
<canvas id="canvas" width="600" height="400"></canvas>

<div id="freqTable"></div>
<div id="explanation"></div>
<div id="codes"></div>

<script>
  const canvas = document.getElementById('canvas');
  const ctx = canvas.getContext('2d');
  const freqTableDiv = document.getElementById('freqTable');
  const codesDiv = document.getElementById('codes');
  const explanationDiv = document.getElementById('explanation');

  // Example input string
  const inputString = "this is an example of huffman encoding";

  // Calculate frequencies
  function calcFrequencies(str) {
    const freq = {};
    for (const ch of str) {
      freq[ch] = (freq[ch] || 0) + 1;
    }
    return freq;
  }

  class Node {
```

```
  constructor(char, freq, left = null, right = null) {
    this.char = char;
    this.freq = freq;
    this.left = left;
    this.right = right;
    this.x = 0;
    this.y = 0;
  }
}

function buildPriorityQueue(freqs) {
  const nodes = [];
  for (const [char, freq] of Object.entries(freqs)) {
    nodes.push(new Node(char, freq));
  }
  return nodes.sort((a,b) => a.freq - b.freq);
}

let nodesQueue;
let mergeSteps = [];
let root = null;

function startHuffman() {
  freqTableDiv.textContent = "Frequency Table:\n" + Object.entries(freqs).map(([c,f]) => `'${c}': ${f
  codesDiv.textContent = "";
  explanationDiv.textContent = "Starting Huffman Encoding...\n\n" +
    "We begin by building a priority queue of nodes from characters sorted by frequency.";
  nodesQueue = buildPriorityQueue(freqs);
  mergeSteps = [];
  root = null;
  drawInitialQueue();
  setTimeout(buildTreeStepwise, 1500);
}

function buildTreeStepwise() {
  if (nodesQueue.length === 1) {
    root = nodesQueue[0];
    drawTree(root);
    explanationDiv.textContent = "All nodes merged! Huffman tree construction completed.\n\n" +
      "Now, we generate Huffman codes by traversing the tree from root to leaves:\n" +
      "- Left edge adds '0' to the code\n- Right edge adds '1' to the code";
    displayCodes(root);
    return;
  }

  // Take two smallest nodes
  const left = nodesQueue.shift();
  const right = nodesQueue.shift();

  explanationDiv.textContent =
    `Step ${mergeSteps.length + 1}: Merge two nodes with smallest frequencies:\n` +
    ` - '${left.char === null ? 'Internal' : left.char}' (freq: ${left.freq})\n` +
    ` - '${right.char === null ? 'Internal' : right.char}' (freq: ${right.freq})\n\n` +
    "Create new internal node with combined frequency and add back to the priority queue.";

  const merged = new Node(null, left.freq + right.freq, left, right);
  mergeSteps.push({left, right, merged});
  nodesQueue.push(merged);
```

```
  nodesQueue.sort((a,b) => a.freq - b.freq);

  drawMergeStep(mergeSteps.length - 1);
  setTimeout(buildTreeStepwise, 2500);
}

function assignPositions(node, depth=0, xMin=0, xMax=canvas.width) {
  if (!node) return;
  node.y = 50 + depth * 70;
  if (!node.left && !node.right) {
    node.x = (xMin + xMax) / 2;
    return;
  }
  assignPositions(node.left, depth+1, xMin, (xMin+xMax)/2);
  assignPositions(node.right, depth+1, (xMin+xMax)/2, xMax);
  node.x = (node.left.x + node.right.x) / 2;
}

function drawTree(node) {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  assignPositions(node);
  ctx.strokeStyle = "#333";
  ctx.fillStyle = "#0074D9";
  ctx.lineWidth = 2;
  ctx.font = "14px monospace";
  ctx.textAlign = "center";
  ctx.textBaseline = "middle";

  function drawNode(n) {
    if (!n) return;
    if(n.left) {
      ctx.beginPath();
      ctx.moveTo(n.x, n.y);
      ctx.lineTo(n.left.x, n.left.y);
      ctx.stroke();
      drawNode(n.left);
    }
    if(n.right) {
      ctx.beginPath();
      ctx.moveTo(n.x, n.y);
      ctx.lineTo(n.right.x, n.right.y);
      ctx.stroke();
      drawNode(n.right);
    }

    ctx.beginPath();
    ctx.arc(n.x, n.y, 20, 0, 2 * Math.PI);
    ctx.fill();
    ctx.stroke();

    ctx.fillStyle = "#fff";
    if(n.char === null){
      ctx.fillText(n.freq, n.x, n.y);
    } else {
      ctx.fillText(n.char, n.x, n.y-6);
      ctx.fillText(n.freq, n.x, n.y+10);
    }
    ctx.fillStyle = "#0074D9";
```

```
  }
  drawNode(node);
}

function drawMergeStep(stepIdx) {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  const step = mergeSteps[stepIdx];
  assignPositions(step.merged);
  ctx.strokeStyle = "#ccc";
  ctx.fillStyle = "#3498db";
  ctx.lineWidth = 2;
  ctx.font = "14px monospace";
  ctx.textAlign = "center";
  ctx.textBaseline = "middle";

  // Draw all previous merges (lighter)
  for (let i=0; i<stepIdx; i++) {
    drawNodeSubtree(mergeSteps[i].merged, "#ccc", "#3498db");
  }

  // Draw current merged subtree highlighted
  drawNodeSubtree(step.merged, "#e67e22", "#d35400");

  // Draw remaining nodesQueue (leaves or unmerged)
  for (const node of nodesQueue) {
    if (!mergeSteps.some(ms => ms.merged === node)) {
      ctx.beginPath();
      ctx.arc(node.x || 0, node.y || 0, 20, 0, 2 * Math.PI);
      ctx.fillStyle = "#3498db";
      ctx.fill();
      ctx.strokeStyle = "#2980b9";
      ctx.stroke();
      ctx.fillStyle = "#fff";
      ctx.fillText(node.char, node.x || 0, (node.y || 0)-6);
      ctx.fillText(node.freq, node.x || 0, (node.y || 0)+10);
    }
  }

  function drawNodeSubtree(node, edgeColor, fillColor) {
    if (!node) return;
    ctx.strokeStyle = edgeColor;
    ctx.fillStyle = fillColor;
    ctx.lineWidth = 2;

    if(node.left){
      ctx.beginPath();
      ctx.moveTo(node.x, node.y);
      ctx.lineTo(node.left.x, node.left.y);
      ctx.stroke();
      drawNodeSubtree(node.left, edgeColor, fillColor);
    }
    if(node.right){
      ctx.beginPath();
      ctx.moveTo(node.x, node.y);
      ctx.lineTo(node.right.x, node.right.y);
      ctx.stroke();
      drawNodeSubtree(node.right, edgeColor, fillColor);
    }
```

```
    ctx.beginPath();
    ctx.arc(node.x, node.y, 20, 0, 2 * Math.PI);
    ctx.fill();
    ctx.stroke();

    ctx.fillStyle = "#fff";
    if(node.char === null){
      ctx.fillText(node.freq, node.x, node.y);
    } else {
      ctx.fillText(node.char, node.x, node.y-6);
      ctx.fillText(node.freq, node.x, node.y+10);
    }
  }
}

function generateCodes(node, prefix = '', codes = {}) {
  if (!node) return codes;
  if (node.char !== null) {
    codes[node.char] = prefix;
  }
  generateCodes(node.left, prefix + '0', codes);
  generateCodes(node.right, prefix + '1', codes);
  return codes;
}

function displayCodes(rootNode) {
  const codes = generateCodes(rootNode);
  let text = "Huffman Codes:\n";
  for (const [ch, code] of Object.entries(codes)) {
    const displayChar = ch === ' ' ? "' '" : ch;
    text += `${displayChar} : ${code}\n`;
  }
  codesDiv.textContent = text + "\n" +
    "Each code is derived by traversing the tree:\n" +
    "- '0' when going left\n" +
    "- '1' when going right\n" +
    "This yields prefix-free binary codes for lossless compression.";
}

function drawInitialQueue() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  const stepNodes = nodesQueue.slice();
  // Lay out nodes evenly horizontally
  const gap = canvas.width / (stepNodes.length + 1);
  stepNodes.forEach((node, i) => {
    node.x = gap * (i+1);
    node.y = canvas.height / 2;
    // Draw node
    ctx.beginPath();
    ctx.arc(node.x, node.y, 20, 0, 2 * Math.PI);
    ctx.fillStyle = "#3498db";
    ctx.fill();
    ctx.strokeStyle = "#2980b9";
    ctx.lineWidth = 2;
    ctx.stroke();

    ctx.fillStyle = "#fff";
    ctx.font = "14px monospace";
```

```
      ctx.textAlign = "center";
      ctx.textBaseline = "middle";
      ctx.fillText(node.char, node.x, node.y - 8);
      ctx.fillText(node.freq, node.x, node.y + 10);
    });
  }

  // Frequencies of input string
  const freqs = calcFrequencies(inputString);

  // Initial display
  freqTableDiv.textContent = "Frequency Table:\n" + Object.entries(freqs).map(([c,f]) => `'${c}': ${f}`
  codesDiv.textContent = "";
  explanationDiv.textContent = "Press the button to start Huffman Encoding.";
  drawTree(null);
</script>

</body>
</html>
```

19.3.2 The Greedy Approach

Huffman's algorithm uses a greedy strategy and a **priority queue** (min-heap) to build a binary tree with minimum total weighted path length:

1. **Start with leaf nodes** for each character, each with its frequency.
2. **Insert all nodes** into a min-heap ordered by frequency.
3. While there is more than one node in the heap:
   - Remove the two nodes with the lowest frequencies.
   - Create a new node with these two as children, and frequency equal to their sum.
   - Insert the new node back into the heap.
4. The remaining node becomes the **root of the Huffman tree**.

The tree is then traversed to assign binary codes: left edges represent `0`, and right edges represent `1`.

19.3.3 JavaScript Implementation

Here is a simplified JavaScript implementation of Huffman encoding using a priority queue:

```
class Node {
  constructor(char, freq, left = null, right = null) {
    this.char = char;
    this.freq = freq;
    this.left = left;
```

```
    this.right = right;
  }
}

// Priority queue using array (simplified)
function buildHuffmanTree(charFreqs) {
  let queue = [];

  for (const [char, freq] of Object.entries(charFreqs)) {
    queue.push(new Node(char, freq));
  }

  queue.sort((a, b) => a.freq - b.freq);

  while (queue.length > 1) {
    const left = queue.shift();
    const right = queue.shift();

    const newNode = new Node(null, left.freq + right.freq, left, right);
    queue.push(newNode);
    queue.sort((a, b) => a.freq - b.freq);
  }

  return queue[0]; // Root of Huffman tree
}

function generateCodes(node, prefix = "", codes = {}) {
  if (!node.left && !node.right) {
    codes[node.char] = prefix;
    return codes;
  }
  if (node.left) generateCodes(node.left, prefix + "0", codes);
  if (node.right) generateCodes(node.right, prefix + "1", codes);
  return codes;
}

// Example
const frequencies = {
  A: 5, B: 9, C: 12, D: 13, E: 16, F: 45
};

const huffmanTree = buildHuffmanTree(frequencies);
const codes = generateCodes(huffmanTree);

console.log("Huffman Codes:", codes);
```

Sample Output:

```
Huffman Codes: { F: '0', C: '100', D: '101', A: '1100', B: '1101', E: '111' }
```

In this output, the most frequent character `F` has the shortest code (`0`), while less frequent characters like `A` have longer codes (`1100`). This minimizes the overall size of encoded messages.

19.3.4 Why Its Greedy

Huffman's algorithm makes greedy decisions by always combining the two least frequent symbols (or subtrees) at each step. This locally optimal choice ensures that more frequent symbols end up closer to the root (shorter codes), which in turn minimizes the weighted sum of the tree depths.

The correctness of this greedy strategy is supported by the **Greedy-Choice Property** and **Optimal Substructure**, which guarantee that combining the least frequent items at each step produces an optimal prefix code.

19.3.5 Applications in Data Compression

Huffman coding is widely used in both theoretical and real-world compression tools:

- **File formats**: ZIP, GZIP, JPEG, PNG use Huffman encoding as part of their compression pipeline.
- **Multimedia codecs**: MP3 and MPEG video formats include Huffman-like coding for entropy compression.
- **Network protocols**: Efficient encoding for transmitting data over bandwidth-constrained connections.

Its strength lies in its ability to compress data **without loss**, preserving full fidelity of the original content while reducing space.

19.3.6 Summary

Huffman encoding is a powerful example of greedy algorithms in action. By always combining the least frequent symbols first, we build an optimal prefix-free encoding tree. The approach is efficient, easy to implement, and highly effective in practice. Huffman's algorithm is a foundational concept in computer science and a great demonstration of how greedy techniques can lead to provably optimal solutions.

19.4 Interval Scheduling

The **interval scheduling problem** is a fundamental problem in resource allocation and scheduling, where the goal is to choose a subset of non-overlapping intervals from a given list. This is closely related to the **activity selection problem**, but can vary in complexity depending on the constraints of the specific scenario.

These problems commonly arise in scheduling systems, such as meeting room assignments, booking conference slots, CPU job scheduling, or managing shared resources over time.

19.4.1 Problem Definition

In the most basic form, interval scheduling is defined as:

> **Given**: A list of intervals, each with a `start` and `end` time. **Goal**: Select the maximum number of mutually non-overlapping intervals.

For example, imagine a shared meeting room and several teams wanting to book it. Each booking request is an interval, and your goal is to accept as many non-overlapping meetings as possible using a single room.

Run the following code in browser to see the demo:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>Interval Scheduling Problem Visualization</title>
<style>
  body {
    font-family: sans-serif;
    padding: 20px;
    background: #fafafa;
    text-align: center;
  }
  canvas {
    border: 1px solid #ccc;
    background: #fff;
    display: block;
    margin: 10px auto;
  }
  button {
    font-size: 16px;
    padding: 8px 14px;
    margin: 10px;
  }
  #explanation {
    max-width: 700px;
    margin: 20px auto;
    font-family: monospace;
    background: #fff;
    padding: 12px;
    border-radius: 6px;
    box-shadow: 0 0 8px #ccc;
    white-space: pre-wrap;
    text-align: left;
  }
</style>
</head>
<body>
```

```
<h2>Interval Scheduling Problem Visualization</h2>
<button onclick="startScheduling()">Start Interval Scheduling</button>
<canvas id="canvas" width="800" height="300"></canvas>
<div id="explanation">Press "Start Interval Scheduling" to begin.</div>

<script>
  const canvas = document.getElementById('canvas');
  const ctx = canvas.getContext('2d');
  const explanationDiv = document.getElementById('explanation');

  // Sample intervals: [start, end]
  const intervals = [
    [1, 4],
    [3, 5],
    [0, 6],
    [5, 7],
    [3, 9],
    [5, 9],
    [6, 10],
    [8, 11],
    [8, 12],
    [2, 14],
    [12, 16],
  ];

  // Sort intervals by finish time (end)
  const sortedIntervals = intervals.slice().sort((a,b) => a[1] - b[1]);

  let selected = [];
  let stepIndex = 0;

  // Constants for drawing
  const timelineStart = 0;
  const timelineEnd = 16;
  const marginLeft = 50;
  const marginRight = 50;
  const marginTop = 40;
  const intervalHeight = 25;
  const gap = 10;
  const totalHeight = (intervalHeight + gap) * sortedIntervals.length + marginTop;

  canvas.height = totalHeight;

  function timeToX(t) {
    const usableWidth = canvas.width - marginLeft - marginRight;
    return marginLeft + (t - timelineStart) / (timelineEnd - timelineStart) * usableWidth;
  }

  function drawIntervals(currentIndex = -1) {
    ctx.clearRect(0, 0, canvas.width, canvas.height);

    // Draw timeline axis
    ctx.strokeStyle = '#333';
    ctx.lineWidth = 2;
    ctx.beginPath();
    ctx.moveTo(marginLeft, marginTop - 20);
    ctx.lineTo(canvas.width - marginRight, marginTop - 20);
    ctx.stroke();
```

```
  // Draw timeline ticks and labels
  ctx.fillStyle = '#333';
  ctx.font = '14px monospace';
  ctx.textAlign = 'center';
  for (let t = timelineStart; t <= timelineEnd; t++) {
    let x = timeToX(t);
    ctx.beginPath();
    ctx.moveTo(x, marginTop - 25);
    ctx.lineTo(x, marginTop - 15);
    ctx.stroke();
    ctx.fillText(t, x, marginTop - 35);
  }

  // Draw all intervals as bars
  for (let i = 0; i < sortedIntervals.length; i++) {
    const [start, end] = sortedIntervals[i];
    const x1 = timeToX(start);
    const x2 = timeToX(end);
    const y = marginTop + i * (intervalHeight + gap);

    // Determine bar color based on selection and current step
    if (selected.includes(i)) {
      ctx.fillStyle = '#27ae60'; // selected green
    } else if (i === currentIndex) {
      ctx.fillStyle = '#f39c12'; // current orange
    } else {
      ctx.fillStyle = '#7f8c8d'; // default gray
    }

    // Draw rectangle for interval
    ctx.fillRect(x1, y, x2 - x1, intervalHeight);

    // Draw border
    ctx.strokeStyle = '#333';
    ctx.lineWidth = 1;
    ctx.strokeRect(x1, y, x2 - x1, intervalHeight);

    // Draw text label [start,end]
    ctx.fillStyle = '#fff';
    ctx.font = '14px monospace';
    ctx.textAlign = 'center';
    ctx.textBaseline = 'middle';
    ctx.fillText(`[${start},${end}]`, (x1 + x2)/2, y + intervalHeight/2);
  }
}

function startScheduling() {
  selected = [];
  stepIndex = 0;
  explanationDiv.textContent = "Intervals sorted by earliest finish time.\n" +
    "We select intervals greedily, choosing the earliest finishing compatible interval at each step."
  drawIntervals();
  setTimeout(stepScheduling, 1500);
}

function stepScheduling() {
  if (stepIndex >= sortedIntervals.length) {
    explanationDiv.textContent += "\n\nScheduling complete.\nSelected intervals:\n" +
```

```
        selected.map(i => `[${sortedIntervals[i][0]},${sortedIntervals[i][1]}]`).join(', ');
      drawIntervals(-1);
      return;
    }

    const [curStart, curEnd] = sortedIntervals[stepIndex];
    let canSelect = true;

    if (selected.length > 0) {
      const lastSelected = sortedIntervals[selected[selected.length-1]];
      if (curStart < lastSelected[1]) {
        canSelect = false;
      }
    }

    explanationDiv.textContent =
      `Step ${stepIndex+1}:\n` +
      `Considering interval [${curStart}, ${curEnd}]\n` +
      (canSelect
        ? `This interval does not overlap with previously selected intervals.\nSelecting it.`
        : `This interval overlaps with previously selected interval [${sortedIntervals[selected[selecte

    if (canSelect) {
      selected.push(stepIndex);
    }

    drawIntervals(stepIndex);
    stepIndex++;
    setTimeout(stepScheduling, 2000);
  }

  // Initial draw
  drawIntervals();
</script>

</body>
</html>
```

19.4.2 Greedy Strategy: Earliest Finish Time

This problem can be solved efficiently with a greedy strategy:

1. **Sort the intervals by their end times**.
2. **Select the first interval** (the one that ends earliest).
3. For each subsequent interval:
    - If its start time is greater than or equal to the end time of the last selected interval, select it.

Why does this work? Choosing the interval that ends earliest **leaves more room** for upcoming intervals, maximizing the total number that can be scheduled. This strategy has

both the **greedy-choice property** and **optimal substructure**, making it ideal for greedy algorithms.

19.4.3 JavaScript Implementation

Let's implement a basic interval scheduling algorithm in JavaScript:

```javascript
function scheduleIntervals(intervals) {
  // Sort intervals by end time
  intervals.sort((a, b) => a.end - b.end);

  const result = [];
  let currentEnd = 0;

  for (const interval of intervals) {
    if (interval.start >= currentEnd) {
      result.push(interval);
      currentEnd = interval.end;
    }
  }

  return result;
}

// Example usage:
const intervals = [
  { start: 1, end: 3 },
  { start: 2, end: 5 },
  { start: 4, end: 7 },
  { start: 6, end: 9 },
  { start: 8, end: 10 }
];

const scheduled = scheduleIntervals(intervals);
console.log("Scheduled intervals:", scheduled);
```

Output:

```
Scheduled intervals: [
  { start: 1, end: 3 },
  { start: 4, end: 7 },
  { start: 8, end: 10 }
]
```

This implementation has a time complexity of **O(n log n)** due to the sorting step and runs in **O(n)** for interval selection.

19.4.4 Variations and Constraints

Not all interval scheduling problems use the same greedy rule. Different constraints can change the optimal strategy:

1. **Weighted Interval Scheduling**: Each interval has a weight (e.g., value or profit), and the goal is to maximize the total weight rather than the number of intervals. Greedy methods don't work here — dynamic programming is needed.
2. **Multiple Resources (k rooms)**: Instead of selecting a subset, the goal may be to assign all intervals to the fewest number of rooms such that no intervals in a room overlap. This leads to the **Interval Partitioning** problem, which also involves sorting and uses a **priority queue** to assign rooms efficiently.
3. **Minimizing Idle Time**: Some scheduling problems aim to minimize the total idle time between jobs. These require more sophisticated approaches and often cannot be solved greedily.

19.4.5 Real-World Applications

Interval scheduling is widely applicable in modern systems:

- **Calendar management**: Scheduling as many meetings as possible without conflict.
- **Classroom or lab usage**: Assigning events to shared resources without overlaps.
- **Job scheduling on a processor**: Running as many jobs as possible without overlaps.
- **Television broadcast**: Scheduling non-overlapping shows to maximize viewership slots.

In many calendar and scheduling apps (like Google Calendar or Microsoft Outlook), the underlying logic for finding available times or suggesting optimal meeting slots is based on interval scheduling principles.

19.4.6 Summary

Interval scheduling is a classic greedy problem where selecting intervals with the **earliest finish time** leads to an optimal solution. This technique is powerful and widely applicable, but the correct greedy strategy depends on the problem's constraints. By identifying when greedy algorithms are appropriate and understanding how to adapt them for variations, you can effectively solve many real-world scheduling challenges using simple and efficient logic in JavaScript.

Chapter 20.

Backtracking and Recursion

1. Combinatorial Problems
2. Subsets, Permutations, and N-Queens
3. Solving Puzzles Recursively

20 Backtracking and Recursion

20.1 Combinatorial Problems

Combinatorial problems are a rich and challenging class of problems where the goal is to explore **all possible configurations** of a given set — such as all subsets, all permutations, or all ways to arrange objects under certain constraints. These problems often involve generating combinations, selecting elements in different orders, or arranging pieces to meet specific goals.

Such problems are inherently **exponential in nature**. For example, the number of subsets of a set with `n` elements is `2^n`, and the number of permutations is `n!`. This makes brute-force approaches inefficient unless guided by smart techniques that reduce redundant computation.

One powerful method for solving combinatorial problems is **backtracking** — a form of controlled recursion that builds solutions step-by-step and **abandons paths** that can't possibly lead to a valid solution.

20.1.1 What Is Backtracking?

Backtracking is a **recursive algorithmic paradigm** used for problems where we build candidates for solutions incrementally, and discard a candidate ("backtrack") as soon as we determine it cannot possibly be part of the final solution.

Backtracking can be seen as a **depth-first search** through a decision tree:

- At each level, make a choice from available options.
- Recursively explore that path.
- If the path is invalid or complete, **undo** the choice and try the next.

This technique significantly reduces the number of recursive calls in problems with constraints or invalid states.

20.1.2 Example 1: Generating All Subsets

Let's start with a classic problem: generating all subsets (also called the power set) of a given array.

Full runnable code:

```
function generateSubsets(nums) {
  const result = [];

  function backtrack(start, path) {
    result.push([...path]); // Add current subset to result
```

```
    for (let i = start; i < nums.length; i++) {
      path.push(nums[i]);             // Choose
      backtrack(i + 1, path);         // Explore
      path.pop();                     // Un-choose (backtrack)
    }
  }

  backtrack(0, []);
  return result;
}

// Example
console.log(generateSubsets([1, 2, 3]));
```

Output:

```
[
  [], [1], [1, 2], [1, 2, 3],
  [1, 3], [2], [2, 3], [3]
]
```

How it works:

- At each step, you choose whether to include the current number.
- You recurse forward, and at each level, explore the remaining elements.
- Backtracking allows the algorithm to discard each choice and explore other branches.

20.1.3 Example 2: Generating All Combinations (k-length)

Let's say we want all combinations of size `k` from a given array.

Full runnable code:

```
function combine(nums, k) {
  const result = [];

  function backtrack(start, path) {
    if (path.length === k) {
      result.push([...path]);
      return;
    }

    for (let i = start; i < nums.length; i++) {
      path.push(nums[i]);
      backtrack(i + 1, path);
      path.pop();
    }
  }

  backtrack(0, []);
  return result;
```

```
}

// Example
console.log(combine([1, 2, 3, 4], 2));
```

Output:

```
[
  [1, 2], [1, 3], [1, 4],
  [2, 3], [2, 4], [3, 4]
]
```

This is very similar to subset generation, but with an added **size constraint**. The recursive tree is pruned whenever the current combination reaches the desired length.

20.1.4 Backtracking Efficiency and Complexity

Although backtracking is more efficient than brute-force enumeration, its time complexity is still **exponential** in most cases:

- Subsets of `n` elements: O(2^n)
- Combinations of `n` elements of length `k`: O(n choose k)
- Permutations of `n` elements: O(n!)

Backtracking improves performance **by pruning** — skipping paths that don't meet the criteria early — but it cannot reduce the fundamental explosion of possibilities when exhaustive search is required.

20.1.5 Real-World Use Cases

Combinatorial and backtracking problems are everywhere:

- **Password generation** or brute-force guessing
- **Subset sum and knapsack**-type decision problems
- **Game solving**: Sudoku, crossword fill-ins, chess move trees
- **Search engines**: Generating keyword permutations
- **Bioinformatics**: DNA sequence alignment

Backtracking is especially useful when the solution space is large, but only a small portion of it is valid or interesting.

20.1.6 Summary

Combinatorial problems require exploring all combinations, permutations, or subsets of a dataset. These problems grow exponentially with input size, but backtracking provides a structured and often more efficient way to explore valid configurations. In JavaScript, recursive functions using arrays and call stacks make backtracking intuitive and powerful. While not a silver bullet, backtracking is a key tool for solving problems that demand a complete search under constraints.

20.2 Subsets, Permutations, and N-Queens

Backtracking shines in problems that involve **exploring all possible arrangements** under a set of constraints. This section dives deeper into three foundational problems that illustrate how backtracking operates in practice: generating subsets, generating permutations, and solving the N-Queens puzzle.

Each of these problems involves recursive decision-making, state tracking, and pruning of invalid paths. Understanding how to structure these recursive calls and manage state cleanly is key to mastering backtracking in JavaScript.

20.2.1 Generating All Subsets

The **subset problem** asks us to generate all possible combinations of elements in an array. This is a basic backtracking problem with no constraints.

JavaScript Example:

Full runnable code:

```
function subsets(nums) {
  const result = [];

  function backtrack(start, path) {
    result.push([...path]); // Add current combination

    for (let i = start; i < nums.length; i++) {
      path.push(nums[i]);          // Choose
      backtrack(i + 1, path);      // Explore
      path.pop();                  // Undo
    }
  }

  backtrack(0, []);
  return result;
}
```

```
// Example
console.log(subsets([1, 2, 3]));
```

State Management Tips:

- `start` prevents revisiting earlier elements.
- `path` is a dynamic list representing the current subset.
- Always `pop()` after recursion to restore the previous state.

Complexity: O(2^n) — Each element can either be included or excluded.

20.2.2 Generating All Permutations

Permutations differ from subsets in that **order matters**, and we use **each element exactly once**.

JavaScript Example:

Full runnable code:

```
function permutations(nums) {
  const result = [];

  function backtrack(path, used) {
    if (path.length === nums.length) {
      result.push([...path]);
      return;
    }

    for (let i = 0; i < nums.length; i++) {
      if (used[i]) continue;

      used[i] = true;
      path.push(nums[i]);
      backtrack(path, used);
      path.pop();
      used[i] = false;
    }
  }

  backtrack([], Array(nums.length).fill(false));
  return result;
}

// Example
console.log(permutations([1, 2, 3]));
```

State Management Tips:

- Use a `used[]` array to track which elements have been included.
- Restore state after each recursive call by resetting `used[i]` and popping from `path`.

Pruning Strategies:

- For arrays with duplicates, skip repeated elements with additional checks to avoid duplicate permutations.

Complexity: O(n!) — The number of permutations of `n` distinct elements.

20.2.3 The N-Queens Problem

The **N-Queens** puzzle asks: Place `n` queens on an `n×n` chessboard so that no two queens threaten each other. That means:

- No two queens in the same row, column, or diagonal.

This is a **constraint satisfaction problem** that's perfect for backtracking.

JavaScript Example (solving and printing solutions):

Full runnable code:

```
function solveNQueens(n) {
  const results = [];
  const board = Array(n).fill().map(() => Array(n).fill('.'));

  const cols = new Set();
  const diag1 = new Set(); // row - col
  const diag2 = new Set(); // row + col

  function backtrack(row) {
    if (row === n) {
      results.push(board.map(r => r.join('')));
      return;
    }

    for (let col = 0; col < n; col++) {
      if (cols.has(col) || diag1.has(row - col) || diag2.has(row + col)) continue;

      board[row][col] = 'Q';
      cols.add(col);
      diag1.add(row - col);
      diag2.add(row + col);

      backtrack(row + 1);

      board[row][col] = '.';
      cols.delete(col);
      diag1.delete(row - col);
      diag2.delete(row + col);
    }
  }

  backtrack(0);
  return results;
```

```
}

// Example
console.log(solveNQueens(4));
```

Output (formatted):

```
[
  [".Q..", "...Q", "Q...", "..Q."],
  ["..Q.", "Q...", "...Q", ".Q.."]
]
```

State Management:

- Use sets to track:
  - Columns where queens are placed.
  - Diagonals: `row - col` ( ) and `row + col` ( ).
- Clean up sets after each recursive call to undo state changes.

Visualization Tip: Print the board at each recursive step to trace queen placement and backtracking:

```
console.log(board.map(row => row.join('')).join('\n') + '\n');
```

Complexity: Exponential, but pruned significantly using sets to skip unsafe positions.

20.2.4 Backtracking Pattern and Debugging Tips

Common structure of backtracking:

```
function backtrack(state) {
  if (solution found) return;
  for (choices) {
    if (valid) {
      make choice;
      backtrack(updated state);
      undo choice;
    }
  }
}
```

Debugging Tips:

- Add `console.log()` at each level to visualize recursion.
- Track the call depth or current path to understand state changes.
- Use indentation based on recursion depth for easier tracing.

20.2.5 Relevance to Constraint Satisfaction

Backtracking is a foundational approach in **constraint satisfaction problems (CSPs)**, which require finding valid configurations that satisfy multiple conditions — such as:

- Sudoku solvers
- Word search puzzles
- Resource assignment with restrictions (e.g., course scheduling)

Each problem defines:

- **Variables** (e.g., board cells, positions)
- **Domains** (e.g., valid values or moves)
- **Constraints** (e.g., no duplicates, no conflicts)

Backtracking efficiently navigates the solution space while pruning invalid paths early — making it ideal for complex CSPs.

20.2.6 Summary

Subsets, permutations, and N-Queens are classic examples that demonstrate the power and flexibility of backtracking. With careful state management, pruning, and recursive logic, you can efficiently explore vast search spaces. As problems grow in complexity, mastering this pattern will help you tackle everything from algorithm puzzles to real-world constraint-driven systems in JavaScript.

20.3 Solving Puzzles Recursively

Recursive algorithms, especially when combined with **backtracking**, provide a natural and powerful approach to solving complex puzzles. Many logic-based challenges—such as **Sudoku**, **crossword filling**, and **maze navigation**—can be modeled as **state exploration problems**. The goal is to recursively try possible moves, prune invalid paths, and backtrack when a dead end is reached.

This section explores how to use recursion to solve simplified versions of these puzzles in JavaScript, along with best practices for managing state, recursion depth, and performance.

20.3.1 Recursive Exploration and Base Cases

At the heart of solving any puzzle recursively is the idea of **recursive state exploration**:

1. **Recursive calls** explore possible choices from the current state.
2. A **base case** defines when a solution has been reached (e.g., the puzzle is completely and correctly filled).
3. Invalid or impossible states are **pruned early** to avoid unnecessary computation.
4. Once all options are exhausted, the algorithm **backtracks** to try other possibilities.

20.3.2 Example 1: Maze Solving

Let's start with a basic puzzle: finding a path through a maze. Represent the maze as a 2D grid with `0`s (open paths) and `1`s (walls).

```
function solveMaze(maze, x = 0, y = 0, path = []) {
  const n = maze.length;
  const m = maze[0].length;

  if (x < 0 || y < 0 || x >= n || y >= m || maze[x][y] !== 0) return false;

  path.push([x, y]);

  if (x === n - 1 && y === m - 1) return true; // Reached the goal

  maze[x][y] = 2; // Mark visited

  const dirs = [[0, 1], [1, 0], [0, -1], [-1, 0]];
  for (const [dx, dy] of dirs) {
    if (solveMaze(maze, x + dx, y + dy, path)) return true;
  }

  path.pop(); // Backtrack
  return false;
}

// Example usage
const maze = [
  [0, 1, 0, 0],
  [0, 0, 0, 1],
  [1, 1, 0, 1],
  [0, 0, 0, 0]
];

const path = [];
if (solveMaze(maze, 0, 0, path)) {
  console.log("Path found:", path);
} else {
  console.log("No path found.");
}
```

Key Points:

- Avoid revisiting by marking cells as visited (`2`).
- Backtrack by removing the last step from the path.
- Base case: reach the goal cell (bottom-right corner).

20.3.3 Example 2: Sudoku Solver (Simplified 44)

Solving a full 9×9 Sudoku is possible with backtracking, but for clarity, here's a simpler 4×4 version using digits `1-4`.

Full runnable code:

```
function isValid(board, row, col, num) {
  for (let i = 0; i < 4; i++) {
    if (board[row][i] === num || board[i][col] === num) return false;
  }

  const boxRow = Math.floor(row / 2) * 2;
  const boxCol = Math.floor(col / 2) * 2;

  for (let i = 0; i < 2; i++) {
    for (let j = 0; j < 2; j++) {
      if (board[boxRow + i][boxCol + j] === num) return false;
    }
  }

  return true;
}

function solveSudoku(board) {
  for (let row = 0; row < 4; row++) {
    for (let col = 0; col < 4; col++) {
      if (board[row][col] === 0) {
        for (let num = 1; num <= 4; num++) {
          if (isValid(board, row, col, num)) {
            board[row][col] = num;
            if (solveSudoku(board)) return true;
            board[row][col] = 0; // Backtrack
          }
        }
        return false; // No valid number found
      }
    }
  }
  return true; // Board is filled correctly
}

// Example board (0 = empty)
const board = [
  [1, 0, 0, 0],
  [0, 0, 2, 1],
  [0, 1, 0, 0],
  [0, 0, 0, 2]
];

if (solveSudoku(board)) {
  console.log("Solved board:");
  console.table(board);
} else {
  console.log("No solution exists.");
}
```

State and Pruning:

- Check constraints before placing a number.
- Backtrack when no valid number can be placed.
- Each recursive call modifies the board in-place.

20.3.4 Managing Recursion Depth and Performance

Recursive puzzle solvers can run deep, especially in more complex puzzles like:

- Full-size Sudoku (up to depth 81)
- Crossword or word-placement problems
- Chessboard problems like N-Queens (depth `n`)

Performance Tips:

- **Limit state copies**: Modify state in place and restore it after backtracking.
- **Use memoization** if subproblems repeat (e.g., in word puzzles).
- **Track visited states** using sets or maps when necessary.
- **Tail recursion optimization** isn't available in all JS engines, so avoid very deep recursion unless necessary.

If recursion depth becomes a concern (e.g., >1000), iterative solutions using a stack may be safer in JavaScript.

20.3.5 Real-World Relevance

Recursive puzzle solving patterns map directly to real-world problems such as:

- **AI planning** (pathfinding, decision trees)
- **Constraint satisfaction** (scheduling, layout, assignment)
- **Interactive games** (solvers for crosswords, puzzles, logic challenges)
- **Validation engines** (form field logic, legal rule applications)

Any problem where you need to **search through options** under **rules and constraints** can likely benefit from backtracking and recursive logic.

20.3.6 Summary

Recursion and backtracking offer a flexible, elegant way to solve constraint-based puzzles like mazes, Sudoku, and word grids. These approaches explore the full space of valid moves and prune dead ends efficiently. With careful management of state, recursion depth, and validity

checks, you can solve even complex puzzles effectively in JavaScript. These strategies aren't just academic — they form the backbone of many real-world problem solvers, games, and AI systems.

Chapter 21.

Real-World Applications

1. Autocomplete with Tries
2. Debouncing/Throttling as Algorithmic Patterns
3. Scheduling Algorithms in UIs
4. Caching Strategies (LRU, LFU) in JS

21 Real-World Applications

21.1 Autocomplete with Tries

Autocomplete features have become essential in modern user interfaces—powering search boxes, text editors, and command palettes. A key data structure enabling fast and responsive autocomplete is the **trie** (pronounced "try"), also known as a **prefix tree**.

Tries are designed for efficient **prefix-based retrieval**, allowing fast lookups for all words that start with a given prefix. Unlike traditional arrays or hash maps, which search whole strings, tries focus on character-by-character structure, making them especially suitable for autocomplete systems.

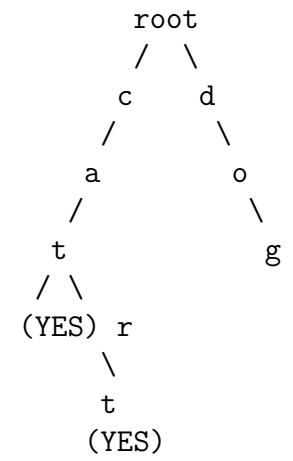
21.1.1 What Is a Trie?

A **trie** is a **tree-like structure** where:

- Each node represents a character in a string.
- The path from the root to a node spells out a prefix.
- Full words are marked explicitly (often with an `isEndOfWord` flag).

For example, storing the words `["cat", "car", "cart", "dog"]` results in a tree where common prefixes like `"ca"` are shared.

`Tree:`



- `YES` indicates end of a word.
- Nodes share common prefixes, making memory usage efficient and searches fast.

21.1.2 Why Use a Trie for Autocomplete?

Consider a search bar where the user types `"ca"` and expects results like `"cat"`, `"car"`, `"cart"`, etc. A naive solution might:

- Loop over an entire array of words.
- Check if each word starts with `"ca"` using `.startsWith()`.

This has $\mathbf{O(n \cdot m)}$ time complexity, where `n` is the number of words and `m` is the prefix length.

A trie, in contrast, allows:

- **Insertion**: O(k) time, where `k` is the length of the word.
- **Prefix search**: O(p) time to find the prefix, then O(r) to collect matching results.

21.1.3 JavaScript Trie Implementation

Let's implement a basic trie in JavaScript with:

- Insertion of words.
- Searching for all words with a given prefix.

```
class TrieNode {
  constructor() {
    this.children = {};       // Map of character -> TrieNode
    this.isEndOfWord = false;
  }
}

class Trie {
  constructor() {
    this.root = new TrieNode();
  }

  // Insert a word into the trie
  insert(word) {
    let node = this.root;
    for (const char of word) {
      if (!node.children[char]) {
        node.children[char] = new TrieNode();
      }
      node = node.children[char];
    }
    node.isEndOfWord = true;
  }

  // Helper to collect all words from a node
  _collect(node, prefix, results) {
    if (node.isEndOfWord) results.push(prefix);

    for (const char in node.children) {
      this._collect(node.children[char], prefix + char, results);
```

```
    }
  }

  // Find all words with a given prefix
  autocomplete(prefix) {
    let node = this.root;
    for (const char of prefix) {
      if (!node.children[char]) return [];
      node = node.children[char];
    }

    const results = [];
    this._collect(node, prefix, results);
    return results;
  }
}
```

21.1.4 Example Usage

```
const trie = new Trie();
const words = ["cat", "car", "cart", "carbon", "dog", "dove", "dot"];

words.forEach(word => trie.insert(word));

console.log(trie.autocomplete("ca")); // ['cat', 'car', 'cart', 'carbon']
console.log(trie.autocomplete("do")); // ['dog', 'dove', 'dot']
console.log(trie.autocomplete("z"));  // []
```

21.1.5 Performance Benefits

Operation	Naive Approach	Trie-Based Approach
Insert Word	O(1) per word	O(k) per word
Search Prefix	O(n · p)	O(p)
Autocomplete Fetch	O(n · p) + filter + collect	O(p) + O(r)

- `n`: number of words
- `p`: prefix length
- `r`: number of results returned

With tries, prefix search is **independent of the total number of words**—once the prefix node is reached, traversal only considers valid completions.

21.1.6 Real-World Applications

- **Search engines**: Efficiently suggest queries as users type.
- **Code editors**: Autocomplete functions, variables, and keywords.
- **Command palettes**: Filter matching actions in tools like VS Code or Figma.
- **Text prediction**: Suggest next words based on context.

Tries scale well for large dictionaries and can be combined with scoring (e.g. frequency, recency) to prioritize results.

21.1.7 Summary

Tries offer a structured and efficient solution to the autocomplete problem. By sharing prefixes and avoiding full-word scans, they reduce lookup time dramatically—especially in large word sets. With simple recursive logic and clear performance advantages, tries are a practical and elegant tool to implement fast prefix search and intelligent text suggestions in JavaScript applications.

21.2 Debouncing/Throttling as Algorithmic Patterns

In user interface programming, it's common to respond to events like `scroll`, `resize`, or `input`—which can fire **dozens of times per second**. If you attach a function that performs heavy computation or updates the DOM frequently, your application can quickly become **sluggish or unresponsive**.

This is where **debouncing** and **throttling** come in. These two **execution control patterns** help regulate how often a function is called in response to rapid-fire events. They are essential tools in frontend development, but they're also rooted in core algorithmic principles: **timing**, **state retention**, and **resource optimization**.

21.2.1 Why Do We Need These Patterns?

Consider this example:

```
window.addEventListener('resize', () => {
  console.log('Window resized!');
});
```

The `resize` event can fire **dozens of times per second** as the user drags the window. If your handler updates a layout, triggers animations, or fetches data, calling it repeatedly could degrade performance. Instead, we want to **delay or limit** the execution.

21.2.2 Debouncing: Wait Until the User Stops

Debouncing ensures a function is called **only after a certain period of inactivity**. It's like saying, "Wait until the user stops typing/resizing/scrolling, then run the function."

Use Case: Search Input

You don't want to send an API request after every keystroke—only when the user **pauses** typing.

JavaScript Implementation:

```
function debounce(fn, delay) {
  let timer = null;

  return function (...args) {
    clearTimeout(timer);
    timer = setTimeout(() => {
      fn.apply(this, args);
    }, delay);
  };
}

// Example usage:
const onSearch = debounce((e) => {
  console.log("Searching for:", e.target.value);
}, 300);

document.getElementById("searchInput").addEventListener("input", onSearch);
```

Behavior:

- The function is **delayed by `delay` ms** each time it's triggered.
- If new input comes in before the delay expires, the timer resets.
- The function only runs **after the user pauses input**.

21.2.3 Throttling: Limit Call Frequency

Throttling ensures a function is **called at most once every `x` milliseconds**, no matter how often the event occurs.

Use Case: Scroll Handler

When tracking scroll position for a sticky header or infinite loading, you want updates at fixed intervals—not dozens per second.

JavaScript Implementation:

```
function throttle(fn, limit) {
  let lastCall = 0;
```

```
  return function (...args) {
    const now = Date.now();
    if (now - lastCall >= limit) {
      lastCall = now;
      fn.apply(this, args);
    }
  };
}

// Example usage:
const onScroll = throttle(() => {
  console.log("Scroll event at:", window.scrollY);
}, 200);

window.addEventListener("scroll", onScroll);
```

Behavior:

- The function runs **immediately**, then waits `limit` ms before it can run again.
- If events continue, only the **first allowed execution** in each time window is performed.

21.2.4 Comparing Debounce vs Throttle

Feature	Debounce	Throttle
Trigger timing	After a pause	At regular intervals
Use case	Input fields, resize	Scroll, drag, mousemove
Example behavior	Run *once* after 300ms of silence	Run *every* 200ms while scrolling

21.2.5 Common Pitfalls

1. **Binding context (`this`) incorrectly** Use `fn.apply(this, args)` or arrow functions to preserve context.
2. **Multiple timers or state leaks** Ensure the timer is scoped properly (e.g., in closures or objects).
3. **Missing cleanup on unmount** In React or similar frameworks, clear timers on component unmount to prevent memory leaks.
4. **Incorrect delay** Test different values to balance responsiveness and performance. Too long = laggy; too short = noisy.

21.2.6 Best Practices

- **Use debounce** when:
  - Waiting for user to finish typing or resizing
  - Validating input or filtering content
- **Use throttle** when:
  - Tracking scroll position
  - Handling rapid mouse movement
  - Updating animations at regular intervals
- **Use libraries** like Lodash (`_.debounce`, `_.throttle`) in production—they're battle-tested and handle edge cases like leading/trailing invocations.

21.2.7 Summary

Debouncing and throttling are crucial for building high-performance, user-friendly web applications. By controlling how frequently functions execute in response to rapid events, these patterns prevent performance issues and improve responsiveness. Understanding how to implement them manually in JavaScript builds your confidence and lets you adapt them to any scenario—whether you're optimizing a search box or handling live window resizes.

21.3 Scheduling Algorithms in UIs

User interface (UI) programming often involves juggling many tasks at once: rendering animations, handling user input, processing data, and updating the DOM. Doing all of this without freezing the interface or making it feel sluggish requires **efficient scheduling**. Scheduling algorithms in UI development aim to **prioritize, defer, or balance tasks** to maintain a smooth and responsive experience.

In this section, we'll explore **`requestAnimationFrame`**, **`setTimeout` prioritization**, and **cooperative multitasking**. These tools and strategies form the backbone of modern UI scheduling in JavaScript, helping manage work in the browser's **single-threaded event loop**.

21.3.1 Why Scheduling Matters in the UI

JavaScript in the browser runs on a **single thread**, which means only one task can run at a time. Long-running computations or frequent DOM updates can block user interactions,

leading to jank, lag, or unresponsiveness.

Smart scheduling ensures that:

- High-priority tasks (like animations or input) happen first.
- Background tasks (like data processing) are delayed.
- No single task monopolizes the main thread.

21.3.2 `requestAnimationFrame`: UI-Friendly Animations

`requestAnimationFrame(fn)` schedules a function to run **right before the next repaint**—roughly every 16ms for 60fps. It's ideal for smooth animations and avoids overloading the browser with redundant frame updates.

Example: Smooth Animation Loop

```
function moveBox() {
  const box = document.getElementById('box');
  let x = 0;

  function animate() {
    x += 2;
    box.style.transform = `translateX(${x}px)`;
    if (x < 300) requestAnimationFrame(animate);
  }

  requestAnimationFrame(animate);
}
```

Benefits:

- Syncs with the browser's refresh rate.
- Skips frames when the tab is inactive, saving resources.
- Prevents layout thrashing from uncontrolled `setInterval` or `setTimeout`.

21.3.3 `setTimeout` and Task Prioritization

`setTimeout` schedules tasks to run **after a delay**, but actual execution is queued behind other tasks in the event loop. It's useful for deferring lower-priority tasks and breaking up long-running operations.

Example: Chunking Large Work

```
function processLargeList(items) {
  let index = 0;

  function processChunk() {
```

```
    const chunkSize = 100;
    for (let i = 0; i < chunkSize && index < items.length; i++) {
      // Simulate work
      console.log("Processing", items[index++]);
    }

    if (index < items.length) {
      setTimeout(processChunk, 0); // Yield control
    }
  }

  processChunk();
}
```

This breaks the work into smaller pieces, preventing UI freezing.

Trade-offs:

- No guarantee on exact delay—timing depends on browser load.
- Delays can compound if the queue is long (e.g. input handling, rendering).

21.3.4 Cooperative Multitasking with `yield` and Microtasks

JavaScript doesn't support preemptive multitasking (automatic pausing of long tasks), but **cooperative multitasking** lets us split tasks into units and yield back to the event loop manually.

Using Promises to Yield Control

```
async function performTasks() {
  for (let i = 0; i < 1000; i++) {
    console.log("Task", i);
    if (i % 100 === 0) {
      await new Promise(resolve => setTimeout(resolve, 0)); // Yield
    }
  }
}
```

When to use:

- Heavy processing that would otherwise block the UI.
- Long loops (e.g. parsing large files or rendering items).

21.3.5 Task Queues and Idle Callbacks

Task queues simulate prioritized execution using queues and timers.

```
const taskQueue = [];

function scheduleTask(task) {
  taskQueue.push(task);
  if (taskQueue.length === 1) {
    setTimeout(runTasks, 0);
  }
}

function runTasks() {
  while (taskQueue.length) {
    const task = taskQueue.shift();
    task();
  }
}
```

For less urgent tasks, you can use `requestIdleCallback()` (not universally supported), which runs when the browser is idle.

```
requestIdleCallback(() => {
  console.log("Running low-priority task");
});
```

21.3.6 Real-World Constraints and Trade-Offs

Technique	Best For	Trade-Offs
`requestAnimationFrame`	Smooth UI animations	Doesn't work for logic-heavy tasks
`setTimeout`	Delayed or chunked work	Lower accuracy, slow under heavy load
`await` with Promises	Cooperative multitasking	Requires async structure
`requestIdleCallback`	Non-urgent background tasks	Not supported in all browsers

In complex UIs, you may combine these techniques. For example:

- Use `requestAnimationFrame` for visual updates.
- Chunk background data work with `setTimeout` or Promises.
- Schedule logging or cleanup with `requestIdleCallback`.

21.3.7 Summary

Efficient scheduling is key to smooth, responsive UI behavior in JavaScript applications. Using tools like `requestAnimationFrame`, `setTimeout`, and cooperative multitasking techniques, you can prioritize user-centric tasks and delay non-essential ones. These scheduling strategies

act like lightweight algorithms—helping distribute workload intelligently while avoiding main-thread congestion. By mastering them, you'll build interfaces that feel fast, fluid, and resilient even under heavy interaction.

21.4 Caching Strategies (LRU, LFU) in JS

Caching is a fundamental technique to **optimize performance** by storing the results of expensive operations—like API calls, computations, or database queries—so they can be quickly retrieved later. However, caches have **limited capacity**, and deciding **which items to evict** when the cache is full is critical to maintaining efficiency. This is where **eviction policies** like **Least Recently Used (LRU)** and **Least Frequently Used (LFU)** come into play.

21.4.1 Understanding Cache Hits and Misses

- **Cache hit:** The requested data is found in the cache, so retrieval is fast.
- **Cache miss:** Data is not in the cache, so it must be fetched or computed, which takes longer.

An effective caching strategy **maximizes cache hits** while using limited memory.

21.4.2 Least Recently Used (LRU) Cache

The LRU policy evicts the **least recently accessed** item when the cache reaches its capacity. The intuition: data used recently is more likely to be used again soon.

Theory Behind LRU

- Track the order of usage.
- When an item is accessed or inserted, it becomes the **most recently used**.
- Evict the **oldest item** (least recently used) to make space.

JavaScript LRU Implementation

Using a `Map` helps since it **preserves insertion order**, which can be manipulated to track usage.

```
class LRUCache {
  constructor(capacity) {
    this.capacity = capacity;
    this.cache = new Map();
  }
```

```
  get(key) {
    if (!this.cache.has(key)) return -1;

    // Move key to the end to mark as recently used
    const value = this.cache.get(key);
    this.cache.delete(key);
    this.cache.set(key, value);

    return value;
  }

  put(key, value) {
    if (this.cache.has(key)) {
      this.cache.delete(key);
    } else if (this.cache.size === this.capacity) {
      // Evict least recently used (first item)
      const oldestKey = this.cache.keys().next().value;
      this.cache.delete(oldestKey);
    }

    this.cache.set(key, value);
  }
}
```

Example Usage: Memoization

```
const lru = new LRUCache(3);

function memoizedFib(n) {
  if (n <= 1) return n;

  const cached = lru.get(n);
  if (cached !== -1) return cached;

  const result = memoizedFib(n - 1) + memoizedFib(n - 2);
  lru.put(n, result);

  return result;
}

console.log(memoizedFib(10));  // Computes and caches
console.log(lru.cache);        // Cache state
```

21.4.3 Least Frequently Used (LFU) Cache

The LFU strategy evicts the item used the **least number of times**. This approach suits workloads where frequency of access is a better predictor of future use than recency.

Theory Behind LFU

- Track how often each cache item is accessed.
- When eviction is needed, remove the item with the **lowest frequency count**.

- If multiple items share the same frequency, evict the least recently used among them.

JavaScript LFU Implementation

Implementing LFU efficiently requires careful data structures to track frequency and usage order.

```
class LFUCache {
  constructor(capacity) {
    this.capacity = capacity;
    this.cache = new Map();          // key -> {value, freq}
    this.freqMap = new Map();        // freq -> keys ordered by usage
    this.minFreq = 0;
  }

  get(key) {
    if (!this.cache.has(key)) return -1;

    const { value, freq } = this.cache.get(key);
    this._updateFreq(key, value, freq);
    return value;
  }

  put(key, value) {
    if (this.capacity === 0) return;

    if (this.cache.has(key)) {
      const { freq } = this.cache.get(key);
      this._updateFreq(key, value, freq);
    } else {
      if (this.cache.size === this.capacity) {
        // Evict LFU key
        const keys = this.freqMap.get(this.minFreq);
        const oldestKey = keys.keys().next().value;
        keys.delete(oldestKey);
        if (keys.size === 0) this.freqMap.delete(this.minFreq);
        this.cache.delete(oldestKey);
      }

      this.cache.set(key, { value, freq: 1 });
      if (!this.freqMap.has(1)) this.freqMap.set(1, new Set());
      this.freqMap.get(1).add(key);
      this.minFreq = 1;
    }
  }

  _updateFreq(key, value, freq) {
    // Remove from old freq set
    const keys = this.freqMap.get(freq);
    keys.delete(key);
    if (keys.size === 0) {
      this.freqMap.delete(freq);
      if (this.minFreq === freq) this.minFreq++;
    }

    // Add to new freq set
    const newFreq = freq + 1;
    if (!this.freqMap.has(newFreq)) this.freqMap.set(newFreq, new Set());
```

```
    this.freqMap.get(newFreq).add(key);

    this.cache.set(key, { value, freq: newFreq });
  }
}
```

Example Usage: API Response Caching

Imagine caching API responses to avoid repeated requests:

```
const cache = new LFUCache(2);

function fetchData(key) {
  const cached = cache.get(key);
  if (cached !== -1) return Promise.resolve(cached);

  return fetch(`https://api.example.com/data/${key}`)
    .then(res => res.json())
    .then(data => {
      cache.put(key, data);
      return data;
    });
}
```

21.4.4 Choosing Between LRU and LFU

Strategy	Best For	Characteristics
LRU	Workloads with **recency locality** (e.g., UI navigation, session data)	Simple to implement, removes oldest unused
LFU	Workloads with **frequency locality** (e.g., hot API endpoints, popular items)	More complex, keeps frequently used items longer

21.4.5 Memory Considerations

- **Cache size:** Choose capacity based on available memory and expected usage.
- **Eviction overhead:** LFU requires extra bookkeeping, possibly increasing memory use.
- **Garbage collection:** Keep references minimal to allow unused data to be freed.

21.4.6 Summary

Caching with eviction policies like LRU and LFU helps JavaScript applications **balance speed and memory constraints**. By intelligently deciding what to keep and what to discard, these algorithms improve responsiveness, reduce redundant computation, and optimize resource usage.

Implementing caches in JavaScript with maps and sets allows you to apply these patterns to memoization, API response storage, or any scenario where repeated access to data benefits from caching. Understanding their trade-offs and use cases helps you build faster, more efficient applications that scale gracefully.

Chapter 22.

Algorithmic Thinking in Practice

1. How to Approach a Problem
2. Brute Force → Greedy → DP
3. Thinking in Constraints

22 Algorithmic Thinking in Practice

22.1 How to Approach a Problem

Effective problem-solving is a cornerstone of algorithmic thinking and programming. Whether you're tackling a coding challenge, debugging, or designing a new feature, having a clear, structured approach helps you break down complexity and build confidence. Here's a step-by-step guide to approaching algorithmic problems, illustrated with a simple JavaScript example.

22.1.1 Step 1: Understand the Problem

Begin by carefully reading the problem statement. Ask yourself:

- **What is the problem asking?**
- **What are the inputs?** (types, constraints, sizes)
- **What are the expected outputs?**
- **Are there any edge cases or special conditions?**

Understanding the problem thoroughly helps avoid wasted effort on irrelevant details.

22.1.2 Step 2: Identify Inputs and Outputs

Write down exactly what the function or program will receive and what it should return.

For example, consider this simple problem:

> Given an array of integers, return the sum of all positive numbers.

- **Input:** An array of integers (e.g., `[-2, 3, 5, -1]`)
- **Output:** A single number (e.g., `8`)

22.1.3 Step 3: Brainstorm Ideas

Think about different ways to solve the problem. Don't worry about efficiency yet—focus on correctness first.

For the sum of positives:

- Loop through the array, add numbers greater than zero.
- Use built-in functions like `.filter()` and `.reduce()`.

22.1.4 Step 4: Choose an Algorithmic Strategy

Decide which approach fits best based on:

- Problem size and constraints
- Complexity requirements
- Clarity and maintainability

For our problem, a simple iteration is efficient and clear.

22.1.5 Step 5: Write Pseudocode

Writing pseudocode before actual coding clarifies logic and helps identify gaps.

Example pseudocode for summing positives:

```
initialize sum to 0
for each number in array
    if number > 0
        add number to sum
return sum
```

22.1.6 Step 6: Implement the Solution

Translate pseudocode to JavaScript:

```javascript
function sumPositiveNumbers(arr) {
  let sum = 0;
  for (const num of arr) {
    if (num > 0) {
      sum += num;
    }
  }
  return sum;
}
```

22.1.7 Step 7: Validate and Test

Test your code with various inputs:

```javascript
console.log(sumPositiveNumbers([-2, 3, 5, -1])); // 8
console.log(sumPositiveNumbers([0, -5, -10]));   // 0
console.log(sumPositiveNumbers([1, 2, 3, 4]));   // 10
```

Consider edge cases such as an empty array or all negative numbers.

22.1.8 Additional Tips for Breaking Down Problems

- **Divide and conquer:** Break complex problems into smaller, manageable subproblems.
- **Draw diagrams:** Visualizing data or flow can reveal insights.
- **Ask "why?" repeatedly:** Understand the purpose of each requirement.
- **Write test cases first:** Clarifies expectations and guides development.

22.1.9 Emphasizing Reasoning Over Coding

Good programmers spend more time thinking about *how* to solve a problem than on writing code immediately. Reasoning includes:

- Verifying assumptions
- Considering time/space complexity
- Ensuring correctness under edge cases

Remember: clear logic leads to clean code.

22.1.10 Summary

A structured problem-solving approach—understand, identify, brainstorm, strategize, plan, implement, and validate—builds a solid foundation for tackling challenges effectively. Using pseudocode bridges thinking and coding, helping to focus on logic before syntax. Practicing this approach sharpens your algorithmic intuition and leads to more robust, maintainable solutions.

22.2 Brute Force → Greedy → DP

When approaching algorithmic problems, you often start with a straightforward but inefficient solution and then refine it to improve performance. Three common stages in this evolution are **brute force**, **greedy algorithms**, and **dynamic programming (DP)**. Each offers different trade-offs between simplicity, speed, and optimality.

22.2.1 Stage 1: Brute Force The Exhaustive Search

Brute force tries all possible solutions to find the best one. It guarantees correctness but often at a high cost.

Example Problem: Coin Change (Minimum Coins)

> Given coin denominations `[1, 3, 4]` and a target amount `6`, find the minimum number of coins needed.

Brute Force Approach

Try every combination of coins to see which sums to the target with the fewest coins.

```
function coinChangeBrute(coins, amount) {
  if (amount === 0) return 0;
  if (amount < 0) return Infinity;

  let minCoins = Infinity;
  for (const coin of coins) {
    const res = coinChangeBrute(coins, amount - coin);
    if (res !== Infinity) {
      minCoins = Math.min(minCoins, res + 1);
    }
  }
  return minCoins;
}

console.log(coinChangeBrute([1, 3, 4], 6)); // Output: 2 (e.g., 3+3)
```

Drawbacks:

- Exponential time complexity (repeated calculations).
- Impractical for large inputs.

22.2.2 Stage 2: Greedy Quick Heuristic

Greedy algorithms make the **locally optimal choice** at each step, hoping it leads to a globally optimal solution. They are faster but **not always correct**.

Greedy Approach to Coin Change

Pick the largest coin possible repeatedly until you reach the amount.

```
function coinChangeGreedy(coins, amount) {
  coins.sort((a, b) => b - a); // Sort descending
  let count = 0;

  for (const coin of coins) {
    while (amount >= coin) {
      amount -= coin;
      count++;
    }
  }

  return amount === 0 ? count : -1;
}

console.log(coinChangeGreedy([1, 3, 4], 6)); // Output: 2 (3+3) - works here
```

Note: Greedy works for some coin sets (like `[1, 5, 10]`), but not all. For example, for `[1, 3, 4]` and amount `6`, greedy yields correct answer here, but for amount `7`, it picks 4 + 1 + 1 = 3 coins, whereas optimal is 3 + 4 = 2 coins.

22.2.3 Stage 3: Dynamic Programming Optimal and Efficient

Dynamic programming builds solutions **bottom-up**, storing intermediate results to avoid repeated work. It guarantees the **optimal solution** with better efficiency than brute force.

DP Solution for Coin Change

```
function coinChangeDP(coins, amount) {
  const dp = Array(amount + 1).fill(Infinity);
  dp[0] = 0;

  for (let i = 1; i <= amount; i++) {
    for (const coin of coins) {
      if (i - coin >= 0) {
        dp[i] = Math.min(dp[i], dp[i - coin] + 1);
      }
    }
  }

  return dp[amount] === Infinity ? -1 : dp[amount];
}

console.log(coinChangeDP([1, 3, 4], 6)); // Output: 2
console.log(coinChangeDP([1, 3, 4], 7)); // Output: 2 (3+4)
```

How it works:

- `dp[i]` stores minimum coins for amount `i`.
- Builds up from smaller subproblems.
- Runs in O(n * m) time (n = amount, m = number of coins).

22.2.4 Recognizing Which Approach to Use

Approach	When to Use	Pros	Cons
Brute Force	For very small input or initial thinking	Simple, guarantees correctness	Exponential time, inefficient

Approach	When to Use	Pros	Cons
Greedy	When problem exhibits **greedy-choice property** (e.g., interval scheduling)	Fast, simple	May produce suboptimal solutions
Dynamic Programming	When overlapping subproblems and optimal substructure exist	Optimal and efficient	Requires memory, more complex

22.2.5 Summary

- **Brute force** is the first step: simple, exhaustive, and slow.
- **Greedy algorithms** speed up by making local choices but risk wrong answers.
- **Dynamic programming** combines correctness with efficiency by remembering past results.

Understanding this progression helps you refine solutions systematically. Start by writing brute force to validate logic, test greedy heuristics for quick wins, then apply DP when optimality and scale matter. This mindset shapes your algorithmic thinking and problem-solving skills in JavaScript and beyond.

22.3 Thinking in Constraints

When designing algorithms, understanding and respecting **constraints**—such as input size, time limits, and memory availability—is essential. Constraints shape your choice of data structures, influence algorithmic complexity, and ultimately determine whether a solution is practical or not. Learning to think within these boundaries is a critical skill for writing efficient, robust JavaScript code.

22.3.1 Why Constraints Matter

Constraints define the environment in which your code runs. For example:

- **Input size:** How large can the input be? Arrays of 10 elements or millions?
- **Time limits:** How quickly must your solution respond? Interactive UIs often need responses under milliseconds, while batch jobs might tolerate minutes.
- **Memory limits:** How much memory can your program consume? Browsers and devices vary widely.

Ignoring constraints leads to code that may work for small examples but fails in real-world use or larger datasets.

22.3.2 Constraints Guide Algorithm Choices

Consider two common algorithms for searching in a list:

- **Linear search:** O(n) time
- **Binary search:** O(log n) time (requires sorted data)

For a small array (e.g., 10 elements), linear search might be fine. But for an array with millions of items, binary search is necessary to meet time constraints.

22.3.3 Example: Handling Large Input Efficiently

Suppose you need to check if many user IDs exist in a database.

Naive Approach

```
const database = [/* large list of user IDs */];

function userExists(id) {
  return database.includes(id);  // O(n) per lookup
}
```

For millions of lookups, this is inefficient (O(n) per query).

Optimized Approach Using a Set

```
const userSet = new Set(database);  // O(n) to build

function userExists(id) {
  return userSet.has(id);            // O(1) per lookup
}
```

By trading some memory to hold a set, you reduce lookup time drastically, meeting stricter time constraints.

22.3.4 Memory vs. Speed Trade-offs

Sometimes, optimizing for speed means using more memory. For example, memoization caches intermediate results for faster retrieval but consumes additional space.

```javascript
const memo = {};

function fibonacci(n) {
  if (n <= 1) return n;
  if (memo[n]) return memo[n];
  memo[n] = fibonacci(n - 1) + fibonacci(n - 2);
  return memo[n];
}
```

This technique drastically improves performance (from exponential to linear time) but requires extra memory.

22.3.5 When Constraints Force Algorithmic Changes

Imagine a problem with **tight memory limits**, such as running code in a browser with limited RAM. You might:

- Use **streaming algorithms** that process data in chunks instead of loading everything at once.
- Avoid large auxiliary data structures.
- Choose in-place algorithms that modify input arrays directly.

Or, if time is critical and you must respond instantly, you may need to:

- Use **approximate algorithms** or heuristics.
- Sacrifice some accuracy for speed.

22.3.6 Edge Cases and Constraints

Constraints often expose edge cases. For example, if an input array might be empty or contain duplicate values, your algorithm must handle these gracefully to avoid errors or incorrect results.

Always test:

- Minimal inputs (empty arrays, zeros).
- Maximal inputs (very large arrays or strings).
- Special inputs (null, undefined, unexpected types).

22.3.7 Summary: Critical Thinking in Constraints

- **Analyze constraints early.** They should guide your initial design, not be an afterthought.
- **Choose data structures wisely.** For example, use sets or maps for fast lookup, linked lists for dynamic data, arrays for indexed access.
- **Balance time and space.** Faster algorithms often require more memory; smaller memory usage may cost speed.
- **Consider edge cases.** Think about how constraints might expose unusual or extreme inputs.
- **Iterate and optimize.** Start simple, then profile and refine your approach to fit constraints.

22.3.8 Final Thought

Thinking in constraints is not just about writing faster code; it's about **writing smart, practical, and reliable solutions**. As you develop your JavaScript skills, keeping these real-world boundaries in mind will make your algorithms both powerful and applicable to the challenges you'll face in production environments.