Data Structures & Algorithms Learning Path

🎯 Complete Basic to Medium-Level DSA Resource

Welcome to your comprehensive journey through Data Structures and Algorithms! This learning resource is designed to take you from beginner to intermediate level with hands-on examples, practical implementations, and real-world applications.

🗂️ Course Structure

Fundamental Data Structures

1. **Arrays & Strings** - Foundation of data storage and manipulation
2. **Linked Lists** - Dynamic data structures and pointer manipulation
3. **Stacks & Queues** - LIFO and FIFO data structures
4. **Hash Tables/Maps** - Fast lookups and key-value storage
5. **Trees** - Hierarchical data structures and binary search trees

Essential Algorithms

6. **Recursion Basics** - Fundamental recursive thinking and implementation
7. **Sorting Algorithms** - Comparison and non-comparison sorting techniques
8. **Searching Algorithms** - Linear, binary, and advanced search methods
9. **Tree Traversals** - DFS, BFS, and specialized tree traversal techniques
10. **Basic Graph Theory** - Graph representations and fundamental algorithms

Medium Complexity Topics

11. **Two Pointers Technique** - Efficient array and string manipulation
12. **Sliding Window** - Optimizing subarray and substring problems
13. **Basic Dynamic Programming** - Optimization through memoization
14. **Backtracking** - Systematic solution space exploration
15. **Simple Greedy Algorithms** - Local optimization strategies

Capstone Project

16. **Mini-Project: Task Management System** - Comprehensive application combining all concepts

---

🎓 Learning Objectives

By the end of this course, you will:

- ✅ **Master fundamental data structures** and their optimal use cases
- ✅ **Implement essential algorithms** in both JavaScript and C++
- ✅ **Analyze time and space complexity** for performance optimization
- ✅ **Solve medium-level coding problems** with confidence

- ✅ **Apply DSA concepts** to real-world software development
- ✅ **Prepare effectively** for technical interviews
- ✅ **Build a comprehensive project** showcasing your skills

🛠️ What You'll Learn

Programming Languages

- **JavaScript**: Modern ES6+ syntax with practical examples
- **C++**: Efficient implementations with STL usage
- **Comparative Analysis**: Understanding language-specific optimizations

Core Concepts

- **Time Complexity**: Big O notation and performance analysis
- **Space Complexity**: Memory usage optimization
- **Algorithm Design**: Problem-solving strategies and patterns
- **Data Structure Selection**: Choosing the right tool for the job
- **Code Optimization**: Writing efficient and maintainable code

Practical Skills

- **Problem Decomposition**: Breaking complex problems into manageable parts
- **Pattern Recognition**: Identifying common algorithmic patterns
- **Testing Strategies**: Validating implementations with edge cases
- **Performance Benchmarking**: Measuring and comparing algorithm efficiency
- **System Design**: Applying DSA concepts to larger systems

📖 Chapter Format

Each chapter follows a consistent, comprehensive format:

📋 Structure

1. **Clear Explanations** - Conceptual understanding with visual aids
2. **Use Cases** - Real-world applications and when to use each concept
3. **Step-by-Step Examples** - Detailed walkthroughs with illustrations
4. **Code Implementations** - Complete JavaScript and C++ examples
5. **Performance Analysis** - Time/space complexity with detailed explanations
6. **Common Pitfalls** - Mistakes to avoid and debugging tips
7. **Practice Problems** - Graduated difficulty with detailed solutions
8. **Interview Tips** - Common questions and optimal approaches

💻 Code Features

- **Comprehensive Comments** - Every line explained for learning
- **Multiple Implementations** - Different approaches for comparison
- **Performance Testing** - Benchmarking code included

- **Error Handling** - Robust implementations with edge case handling
- **Best Practices** - Industry-standard coding conventions

🎯 Target Audience

Perfect for:

- **Beginner Programmers** looking to build strong foundations
- **Self-taught Developers** wanting structured DSA knowledge
- **Computer Science Students** seeking practical implementations
- **Interview Candidates** preparing for technical assessments
- **Professional Developers** refreshing algorithmic knowledge

Prerequisites:

- Basic programming knowledge in any language
- Understanding of variables, loops, and functions
- Familiarity with object-oriented programming concepts
- Willingness to practice and experiment with code

📈 Learning Path Recommendations

🚀 Fast Track (4-6 weeks)

For experienced programmers or intensive study:

- Week 1: Chapters 1-4 (Fundamental Data Structures)
- Week 2: Chapters 5-7 (Trees, Recursion, Sorting)
- Week 3: Chapters 8-10 (Searching, Traversals, Graphs)
- Week 4: Chapters 11-13 (Two Pointers, Sliding Window, DP)
- Week 5: Chapters 14-15 (Backtracking, Greedy)
- Week 6: Chapter 16 (Mini-Project)

🎯 Standard Track (8-10 weeks)

For balanced learning with practice time:

- Weeks 1-2: Chapters 1-3 (Arrays, Strings, Linked Lists, Stacks, Queues)
- Weeks 3-4: Chapters 4-6 (Hash Tables, Trees, Recursion)
- Weeks 5-6: Chapters 7-9 (Sorting, Searching, Traversals)
- Weeks 7-8: Chapters 10-12 (Graphs, Two Pointers, Sliding Window)
- Weeks 9-10: Chapters 13-16 (DP, Backtracking, Greedy, Project)

🌱 Beginner Track (12-16 weeks)

For thorough understanding and mastery:

- Weeks 1-3: Chapters 1-2 (Arrays, Strings, Linked Lists)
- Weeks 4-6: Chapters 3-4 (Stacks, Queues, Hash Tables)

- Weeks 7-9: Chapters 5-6 (Trees, Recursion)
- Weeks 10-12: Chapters 7-9 (Sorting, Searching, Traversals)
- Weeks 13-14: Chapters 10-12 (Graphs, Two Pointers, Sliding Window)
- Weeks 15-16: Chapters 13-16 (DP, Backtracking, Greedy, Project)

🏆 Progress Tracking

Completion Checklist

Fundamental Data Structures

- ☐ **Arrays & Strings** - Master array manipulation and string algorithms
- ☐ **Linked Lists** - Implement singly and doubly linked lists
- ☐ **Stacks & Queues** - Build LIFO and FIFO data structures
- ☐ **Hash Tables** - Create efficient key-value storage systems
- ☐ **Trees** - Construct and manipulate binary trees and BSTs

Essential Algorithms

- ☐ **Recursion** - Master recursive thinking and implementation
- ☐ **Sorting** - Implement and compare various sorting algorithms
- ☐ **Searching** - Build efficient search algorithms
- ☐ **Tree Traversals** - Master all traversal techniques
- ☐ **Graph Basics** - Understand graph representations and algorithms

Medium Complexity

- ☐ **Two Pointers** - Optimize array and string problems
- ☐ **Sliding Window** - Solve subarray optimization problems
- ☐ **Dynamic Programming** - Master memoization and optimization
- ☐ **Backtracking** - Implement constraint satisfaction algorithms
- ☐ **Greedy Algorithms** - Apply local optimization strategies

Capstone Project

- ☐ **Task Management System** - Build comprehensive application
- ☐ **Performance Analysis** - Benchmark and optimize implementations
- ☐ **Testing Suite** - Create comprehensive test coverage
- ☐ **Documentation** - Write clear technical documentation

💡 Study Tips

📝 Effective Learning Strategies

1. **Code Along** - Don't just read, implement every example
2. **Modify Examples** - Change parameters and see what happens
3. **Solve Practice Problems** - Apply concepts immediately

4. **Explain to Others** - Teaching reinforces understanding
5. **Build Projects** - Create your own applications using learned concepts

🔧 Debugging and Problem-Solving

1. **Start Simple** - Begin with basic cases before complex scenarios
2. **Use Debuggers** - Step through code to understand execution
3. **Draw Diagrams** - Visualize data structures and algorithm flow
4. **Test Edge Cases** - Always consider boundary conditions
5. **Optimize Later** - Get it working first, then make it efficient

📊 Performance Analysis

1. **Measure Everything** - Use timing functions to benchmark code
2. **Compare Approaches** - Implement multiple solutions and compare
3. **Understand Trade-offs** - Time vs. space complexity decisions
4. **Profile Memory Usage** - Monitor memory consumption patterns
5. **Scale Testing** - Test with increasingly large datasets

🎯 Interview Preparation

📋 Common Interview Topics

- **Array Manipulation** - Two pointers, sliding window, sorting
- **String Processing** - Pattern matching, parsing, transformation
- **Linked List Operations** - Reversal, cycle detection, merging
- **Tree Problems** - Traversals, path finding, validation
- **Graph Algorithms** - BFS, DFS, shortest path, cycle detection
- **Dynamic Programming** - Optimization problems, memoization
- **System Design** - Scalability, data structure selection

💼 Interview Success Strategies

1. **Practice Regularly** - Solve problems daily to maintain skills
2. **Explain Your Thinking** - Verbalize your problem-solving process
3. **Start with Brute Force** - Get a working solution first
4. **Optimize Iteratively** - Improve time and space complexity
5. **Handle Edge Cases** - Consider null inputs, empty arrays, etc.
6. **Test Your Code** - Walk through examples to verify correctness

---

🛠️ Development Environment Setup

JavaScript Environment

```
# Node.js for running JavaScript
npm install -g node
```

```
# Optional: Testing framework
npm install -g jest

# Optional: Performance benchmarking
npm install benchmark
```

C++ Environment

```
# GCC compiler (Linux/Mac)
sudo apt-get install g++  # Ubuntu
brew install gcc          # macOS

# Visual Studio (Windows)
# Download from Microsoft website

# Compilation example
g++ -std=c++17 -O2 -o program program.cpp
```

Recommended Tools

- **Code Editor**: VS Code, CLion, or your preferred IDE
- **Version Control**: Git for tracking progress
- **Online Judges**: LeetCode, HackerRank for additional practice
- **Visualization Tools**: VisuAlgo, Algorithm Visualizer

📚 Additional Resources

📖 Recommended Books

- "Introduction to Algorithms" by Cormen, Leiserson, Rivest, Stein
- "Algorithms" by Robert Sedgewick and Kevin Wayne
- "Cracking the Coding Interview" by Gayle Laakmann McDowell
- "Elements of Programming Interviews" by Aziz, Lee, and Prakash

🌐 Online Resources

- **LeetCode**: Practice problems with community solutions
- **HackerRank**: Structured challenges and competitions
- **GeeksforGeeks**: Comprehensive algorithm explanations
- **VisuAlgo**: Interactive algorithm visualizations
- **Big-O Cheat Sheet**: Quick complexity reference

📺 Video Content

- **YouTube Channels**: Abdul Bari, MIT OpenCourseWare, Tushar Roy
- **Online Courses**: Coursera, edX, Udemy algorithm courses
- **Coding Streams**: Watch experienced developers solve problems

🤝 Community and Support

💬 Getting Help

- **Stack Overflow**: Technical questions and debugging help
- **Reddit**: r/algorithms, r/programming, r/learnprogramming
- **Discord/Slack**: Programming communities and study groups
- **GitHub**: Open source implementations and discussions

🎯 Practice Platforms

- **LeetCode**: 2000+ problems with detailed solutions
- **HackerRank**: Structured learning paths and competitions
- **CodeSignal**: Interview practice and skill assessment
- **AtCoder**: Competitive programming contests
- **Codeforces**: Advanced competitive programming

🎉 Getting Started

🚀 Quick Start Guide

1. **Choose Your Track** - Select learning pace based on your schedule
2. **Set Up Environment** - Install necessary tools and compilers
3. **Start with Chapter 1** - Begin with Arrays & Strings
4. **Code Every Example** - Don't skip the implementation practice
5. **Solve Practice Problems** - Apply concepts immediately
6. **Track Your Progress** - Use the completion checklist
7. **Build the Final Project** - Demonstrate your comprehensive skills

📆 Study Schedule Template

```
Week X: [Chapter Name]
├── Day 1: Read theory and examples
├── Day 2: Implement JavaScript solutions
├── Day 3: Implement C++ solutions
├── Day 4: Solve practice problems
├── Day 5: Review and optimize solutions
├── Day 6: Additional practice/projects
└── Day 7: Rest or review previous chapters
```

🎯 Success Metrics

📊 How to Measure Progress

- **Completion Rate**: Percentage of chapters finished

- **Problem Solving**: Number of practice problems solved
- **Code Quality**: Clean, efficient, well-commented implementations
- **Performance Understanding**: Ability to analyze time/space complexity
- **Application Skills**: Successfully building the final project
- **Interview Readiness**: Confidence in explaining and implementing solutions

🏆 Milestones

- 🥉 **Bronze**: Complete fundamental data structures (Chapters 1-5)
- 🥈 **Silver**: Master essential algorithms (Chapters 6-10)
- 🥇 **Gold**: Conquer medium complexity topics (Chapters 11-15)
- 💎 **Diamond**: Build and optimize the final project (Chapter 16)

📝 Final Notes

This learning resource is designed to be:

- **Comprehensive**: Covering all essential DSA topics
- **Practical**: Focus on real-world applications
- **Progressive**: Building complexity gradually
- **Interactive**: Hands-on coding and problem-solving
- **Interview-Ready**: Preparing you for technical assessments

Remember: The key to mastering data structures and algorithms is consistent practice and application. Don't rush through the material—take time to understand each concept deeply and implement every example.

Good luck on your DSA learning journey! 🚀

"The best way to learn algorithms is to implement them yourself. This resource gives you the structured path and practical examples to do exactly that."

Happy Coding! 💻✨

📄 License

This educational resource is provided for learning purposes. Feel free to use, modify, and share for educational and non-commercial purposes.

Last Updated: December 2024
Version: 1.0
Maintainer: DSA Learning Community

Chapter 1: Arrays & Strings - Foundation of Data Structures

🎯 What Are Arrays & Strings?

Arrays are collections of elements stored in contiguous memory locations, where each element can be accessed using an index. **Strings** are essentially arrays of characters with additional operations for text manipulation.

Why Arrays & Strings Matter:

- **Foundation**: Building blocks for more complex data structures
- **Performance**: O(1) random access to elements
- **Memory Efficiency**: Contiguous storage reduces memory overhead
- **Ubiquitous**: Used in almost every programming problem

🔍 Core Operations & Concepts

1. Array Traversal

Visiting each element in the array sequentially.

2. Insertion & Deletion

Adding or removing elements at specific positions.

3. Searching

Finding elements or patterns within arrays/strings.

4. String Manipulation

Operations like concatenation, substring extraction, and pattern matching.

💻 JavaScript Implementation

```
class ArrayOperations {
  constructor() {
    this.arr = [];
  }

  // Insert element at specific index
  // Time: O(n), Space: O(1)
  insert(index, element) {
    if (index < 0 || index > this.arr.length) {
      throw new Error("Index out of bounds");
    }

    // Shift elements to the right
    for (let i = this.arr.length; i > index; i--) {
      this.arr[i] = this.arr[i - 1];
    }
```

```
    this.arr[index] = element;
    return this.arr;
  }

  // Delete element at specific index
  // Time: O(n), Space: O(1)
  delete(index) {
    if (index < 0 || index >= this.arr.length) {
      throw new Error("Index out of bounds");
    }

    const deletedElement = this.arr[index];

    // Shift elements to the left
    for (let i = index; i < this.arr.length - 1; i++) {
      this.arr[i] = this.arr[i + 1];
    }

    this.arr.length--; // Reduce array size
    return deletedElement;
  }

  // Linear search for element
  // Time: O(n), Space: O(1)
  search(element) {
    for (let i = 0; i < this.arr.length; i++) {
      if (this.arr[i] === element) {
        return i; // Return index if found
      }
    }
    return -1; // Element not found
  }

  // Reverse array in-place
  // Time: O(n), Space: O(1)
  reverse() {
    let left = 0;
    let right = this.arr.length - 1;

    while (left < right) {
      // Swap elements
      [this.arr[left], this.arr[right]] = [this.arr[right], this.arr[left]];
      left++;
      right--;
    }

    return this.arr;
  }
}

// String Operations
class StringOperations {
  // Check if string is palindrome
```

```
  // Time: O(n), Space: O(1)
  static isPalindrome(str) {
    const cleaned = str.toLowerCase().replace(/[^a-z0-9]/g, "");
    let left = 0;
    let right = cleaned.length - 1;

    while (left < right) {
      if (cleaned[left] !== cleaned[right]) {
        return false;
      }
      left++;
      right--;
    }

    return true;
  }

  // Find all anagrams of pattern in text
  // Time: O(n), Space: O(1) - assuming fixed alphabet size
  static findAnagrams(text, pattern) {
    const result = [];
    const patternCount = new Array(26).fill(0);
    const windowCount = new Array(26).fill(0);

    // Count characters in pattern
    for (let char of pattern) {
      patternCount[char.charCodeAt(0) - "a".charCodeAt(0)]++;
    }

    // Sliding window approach
    for (let i = 0; i < text.length; i++) {
      // Add current character to window
      windowCount[text[i].charCodeAt(0) - "a".charCodeAt(0)]++;

      // Remove character that's out of window
      if (i >= pattern.length) {
        windowCount[
          text[i - pattern.length].charCodeAt(0) - "a".charCodeAt(0)
        ]--;
      }

      // Check if current window is an anagram
      if (
        i >= pattern.length - 1 &&
        this.arraysEqual(patternCount, windowCount)
      ) {
        result.push(i - pattern.length + 1);
      }
    }

    return result;
  }

  static arraysEqual(arr1, arr2) {
```

```
    return arr1.every((val, index) => val === arr2[index]);
  }
}

// Example Usage
const arrayOps = new ArrayOperations();
arrayOps.arr = [1, 2, 3, 4, 5];
console.log(arrayOps.insert(2, 10)); // [1, 2, 10, 3, 4, 5]
console.log(arrayOps.search(10)); // 2
console.log(arrayOps.reverse()); // [5, 4, 3, 10, 2, 1]

console.log(StringOperations.isPalindrome("A man a plan a canal Panama")); // true
console.log(StringOperations.findAnagrams("abab", "ab")); // [0, 2]
```

C++ Implementation

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <unordered_map>
using namespace std;

class ArrayOperations {
private:
    vector<int> arr;

public:
    // Insert element at specific index
    // Time: O(n), Space: O(1)
    void insert(int index, int element) {
        if (index < 0 || index > arr.size()) {
            throw out_of_range("Index out of bounds");
        }

        arr.insert(arr.begin() + index, element);
    }

    // Delete element at specific index
    // Time: O(n), Space: O(1)
    int deleteAt(int index) {
        if (index < 0 || index >= arr.size()) {
            throw out_of_range("Index out of bounds");
        }

        int deletedElement = arr[index];
        arr.erase(arr.begin() + index);
        return deletedElement;
    }
```

```
    // Linear search for element
    // Time: O(n), Space: O(1)
    int search(int element) {
        for (int i = 0; i < arr.size(); i++) {
            if (arr[i] == element) {
                return i; // Return index if found
            }
        }
        return -1; // Element not found
    }

    // Reverse array in-place
    // Time: O(n), Space: O(1)
    void reverse() {
        int left = 0;
        int right = arr.size() - 1;

        while (left < right) {
            swap(arr[left], arr[right]);
            left++;
            right--;
        }
    }

    // Display array
    void display() {
        cout << "[";
        for (int i = 0; i < arr.size(); i++) {
            cout << arr[i];
            if (i < arr.size() - 1) cout << ", ";
        }
        cout << "]" << endl;
    }

    // Initialize array
    void setArray(vector<int> newArr) {
        arr = newArr;
    }
};

class StringOperations {
public:
    // Check if string is palindrome
    // Time: O(n), Space: O(1)
    static bool isPalindrome(string str) {
        // Clean string: remove non-alphanumeric and convert to lowercase
        string cleaned = "";
        for (char c : str) {
            if (isalnum(c)) {
                cleaned += tolower(c);
            }
        }

        int left = 0;
```

```
        int right = cleaned.length() - 1;

        while (left < right) {
            if (cleaned[left] != cleaned[right]) {
                return false;
            }
            left++;
            right--;
        }

        return true;
    }

    // Find all anagrams of pattern in text
    // Time: O(n), Space: O(1) - assuming fixed alphabet size
    static vector<int> findAnagrams(string text, string pattern) {
        vector<int> result;
        if (text.length() < pattern.length()) return result;

        vector<int> patternCount(26, 0);
        vector<int> windowCount(26, 0);

        // Count characters in pattern
        for (char c : pattern) {
            patternCount[c - 'a']++;
        }

        // Sliding window approach
        for (int i = 0; i < text.length(); i++) {
            // Add current character to window
            windowCount[text[i] - 'a']++;

            // Remove character that's out of window
            if (i >= pattern.length()) {
                windowCount[text[i - pattern.length()] - 'a']--;
            }

            // Check if current window is an anagram
            if (i >= pattern.length() - 1 && patternCount == windowCount) {
                result.push_back(i - pattern.length() + 1);
            }
        }

        return result;
    }

    // Longest substring without repeating characters
    // Time: O(n), Space: O(min(m,n)) where m is charset size
    static int longestUniqueSubstring(string s) {
        unordered_map<char, int> charIndex;
        int maxLength = 0;
        int start = 0;

        for (int end = 0; end < s.length(); end++) {
```

```
            char currentChar = s[end];

            // If character is already in current window, move start
            if (charIndex.find(currentChar) != charIndex.end() &&
                charIndex[currentChar] >= start) {
                start = charIndex[currentChar] + 1;
            }

            charIndex[currentChar] = end;
            maxLength = max(maxLength, end - start + 1);
        }

        return maxLength;
    }
};

// Example Usage
int main() {
    // Array operations
    ArrayOperations arrayOps;
    arrayOps.setArray({1, 2, 3, 4, 5});

    cout << "Original array: ";
    arrayOps.display();

    arrayOps.insert(2, 10);
    cout << "After inserting 10 at index 2: ";
    arrayOps.display();

    cout << "Search for 10: " << arrayOps.search(10) << endl;

    arrayOps.reverse();
    cout << "After reversing: ";
    arrayOps.display();

    // String operations
    cout << "\nString Operations:" << endl;
    cout << "Is 'A man a plan a canal Panama' palindrome? "
         << (StringOperations::isPalindrome("A man a plan a canal Panama") ? "Yes"
: "No") << endl;

    vector<int> anagrams = StringOperations::findAnagrams("abab", "ab");
    cout << "Anagram positions of 'ab' in 'abab': ";
    for (int pos : anagrams) {
        cout << pos << " ";
    }
    cout << endl;

    cout << "Longest unique substring in 'abcabcbb': "
         << StringOperations::longestUniqueSubstring("abcabcbb") << endl;

    return 0;
}
```

⚡ Performance Analysis

Time Complexity:

- **Access**: O(1) - Direct indexing
- **Search**: O(n) - Linear search through elements
- **Insertion**: O(n) - May need to shift elements
- **Deletion**: O(n) - May need to shift elements
- **Traversal**: O(n) - Visit each element once

Space Complexity:

- **Fixed-size arrays**: O(1) additional space
- **Dynamic arrays**: O(n) for the array itself
- **String operations**: Often O(1) additional space with in-place algorithms

Common Pitfalls:

1. **Index out of bounds**: Always validate array indices
2. **Off-by-one errors**: Be careful with loop boundaries
3. **Memory management**: In C++, be mindful of dynamic allocation
4. **String immutability**: In some languages, strings are immutable

🧩 Practice Problems

Problem 1: Two Sum

Question: Given an array of integers and a target sum, return indices of two numbers that add up to the target.

Hint: Use a hash map to store complements as you iterate.

Solution Approach:

```
function twoSum(nums, target) {
  const map = new Map();

  for (let i = 0; i < nums.length; i++) {
    const complement = target - nums[i];
    if (map.has(complement)) {
      return [map.get(complement), i];
    }
    map.set(nums[i], i);
  }

  return [];
}
```

Problem 2: Valid Anagram

Question: Given two strings, determine if they are anagrams of each other.

Hint: Count character frequencies or sort both strings.

Problem 3: Remove Duplicates

Question: Remove duplicates from a sorted array in-place.

Hint: Use two pointers - one for reading, one for writing.

🎯 Interview Tips

What Interviewers Look For:

1. **Edge case handling**: Empty arrays, single elements, null inputs
2. **Optimization awareness**: Can you improve from $O(n^2)$ to $O(n)$?
3. **Space-time tradeoffs**: When to use extra space for better time complexity
4. **Clean code**: Readable, well-commented implementations

Common Interview Patterns:

- **Two Pointers**: For problems involving pairs or reversing
- **Sliding Window**: For substring/subarray problems
- **Hash Maps**: For frequency counting and lookups
- **In-place operations**: Modifying arrays without extra space

Red Flags to Avoid:

- Not asking about input constraints
- Jumping to code without explaining approach
- Ignoring edge cases
- Not discussing time/space complexity

🚀 Key Takeaways

1. **Arrays are fundamental** - Master them before moving to complex structures
2. **Index management is crucial** - Most bugs come from incorrect indexing
3. **Consider both time and space** - Sometimes trading space for time is worth it
4. **Practice string manipulation** - Very common in interviews
5. **Learn common patterns** - Two pointers, sliding window, etc.

Next Chapter: We'll explore Linked Lists and see how they differ from arrays in memory layout and operations.

Chapter 2: Linked Lists - Dynamic Memory Management

🎯 What Are Linked Lists?

Linked Lists are linear data structures where elements (nodes) are stored in sequence, but unlike arrays, they don't require contiguous memory. Each node contains data and a reference (pointer) to the next node in the sequence.

Why Linked Lists Matter:

- **Dynamic Size**: Can grow or shrink during runtime
- **Efficient Insertion/Deletion**: O(1) at known positions
- **Memory Efficiency**: Only allocate what you need
- **Foundation**: Building block for stacks, queues, and graphs

Types of Linked Lists:

1. **Singly Linked List**: Each node points to the next node
2. **Doubly Linked List**: Each node has pointers to both next and previous nodes
3. **Circular Linked List**: Last node points back to the first node

---

🔍 Core Operations

1. Insertion

- At the beginning (head)
- At the end (tail)
- At a specific position

2. Deletion

- From the beginning
- From the end
- From a specific position
- By value

3. Traversal

- Forward traversal
- Backward traversal (doubly linked)

4. Search

- Finding a node by value
- Finding a node by position

💻 JavaScript Implementation

```
// Node class for singly linked list
class ListNode {
  constructor(data) {
    this.data = data;
    this.next = null;
  }
}

// Singly Linked List Implementation
class SinglyLinkedList {
  constructor() {
    this.head = null;
    this.size = 0;
  }

  // Insert at the beginning
  // Time: O(1), Space: O(1)
  insertAtHead(data) {
    const newNode = new ListNode(data);
    newNode.next = this.head;
    this.head = newNode;
    this.size++;
    return this;
  }

  // Insert at the end
  // Time: O(n), Space: O(1)
  insertAtTail(data) {
    const newNode = new ListNode(data);

    if (!this.head) {
      this.head = newNode;
    } else {
      let current = this.head;
      while (current.next) {
        current = current.next;
      }
      current.next = newNode;
    }

    this.size++;
    return this;
  }

  // Insert at specific index
  // Time: O(n), Space: O(1)
  insertAt(index, data) {
    if (index < 0 || index > this.size) {
      throw new Error("Index out of bounds");
    }

    if (index === 0) {
      return this.insertAtHead(data);
```

```
    }

    const newNode = new ListNode(data);
    let current = this.head;

    // Traverse to position index-1
    for (let i = 0; i < index - 1; i++) {
      current = current.next;
    }

    newNode.next = current.next;
    current.next = newNode;
    this.size++;
    return this;
  }

  // Delete from head
  // Time: O(1), Space: O(1)
  deleteHead() {
    if (!this.head) {
      return null;
    }

    const deletedData = this.head.data;
    this.head = this.head.next;
    this.size--;
    return deletedData;
  }

  // Delete from tail
  // Time: O(n), Space: O(1)
  deleteTail() {
    if (!this.head) {
      return null;
    }

    if (!this.head.next) {
      const deletedData = this.head.data;
      this.head = null;
      this.size--;
      return deletedData;
    }

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

    const deletedData = current.next.data;
    current.next = null;
    this.size--;
    return deletedData;
  }
```

```
    // Delete by value
    // Time: O(n), Space: O(1)
    deleteByValue(value) {
      if (!this.head) {
        return false;
      }

      if (this.head.data === value) {
        this.deleteHead();
        return true;
      }

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

      if (current.next) {
        current.next = current.next.next;
        this.size--;
        return true;
      }

      return false;
    }

    // Search for value
    // Time: O(n), Space: O(1)
    search(value) {
      let current = this.head;
      let index = 0;

      while (current) {
        if (current.data === value) {
          return index;
        }
        current = current.next;
        index++;
      }

      return -1;
    }

    // Reverse the linked list
    // Time: O(n), Space: O(1)
    reverse() {
      let prev = null;
      let current = this.head;
      let next = null;

      while (current) {
        next = current.next; // Store next node
        current.next = prev; // Reverse the link
        prev = current; // Move prev forward
```

```
      current = next; // Move current forward
    }

    this.head = prev;
    return this;
  }

  // Find middle element (Floyd's Cycle Detection)
  // Time: O(n), Space: O(1)
  findMiddle() {
    if (!this.head) return null;

    let slow = this.head;
    let fast = this.head;

    while (fast && fast.next) {
      slow = slow.next;
      fast = fast.next.next;
    }

    return slow.data;
  }

  // Detect cycle in linked list
  // Time: O(n), Space: O(1)
  hasCycle() {
    if (!this.head) return false;

    let slow = this.head;
    let fast = this.head;

    while (fast && fast.next) {
      slow = slow.next;
      fast = fast.next.next;

      if (slow === fast) {
        return true;
      }
    }

    return false;
  }

  // Convert to array for easy display
  toArray() {
    const result = [];
    let current = this.head;

    while (current) {
      result.push(current.data);
      current = current.next;
    }

    return result;
```

```
    }

    // Get size
    getSize() {
      return this.size;
    }
  }

  // Doubly Linked List Node
  class DoublyListNode {
    constructor(data) {
      this.data = data;
      this.next = null;
      this.prev = null;
    }
  }

  // Doubly Linked List Implementation
  class DoublyLinkedList {
    constructor() {
      this.head = null;
      this.tail = null;
      this.size = 0;
    }

    // Insert at head
    // Time: O(1), Space: O(1)
    insertAtHead(data) {
      const newNode = new DoublyListNode(data);

      if (!this.head) {
        this.head = this.tail = newNode;
      } else {
        newNode.next = this.head;
        this.head.prev = newNode;
        this.head = newNode;
      }

      this.size++;
      return this;
    }

    // Insert at tail
    // Time: O(1), Space: O(1)
    insertAtTail(data) {
      const newNode = new DoublyListNode(data);

      if (!this.tail) {
        this.head = this.tail = newNode;
      } else {
        this.tail.next = newNode;
        newNode.prev = this.tail;
        this.tail = newNode;
      }
```

```
    this.size++;
    return this;
  }

  // Delete from head
  // Time: O(1), Space: O(1)
  deleteHead() {
    if (!this.head) return null;

    const deletedData = this.head.data;

    if (this.head === this.tail) {
      this.head = this.tail = null;
    } else {
      this.head = this.head.next;
      this.head.prev = null;
    }

    this.size--;
    return deletedData;
  }

  // Delete from tail
  // Time: O(1), Space: O(1)
  deleteTail() {
    if (!this.tail) return null;

    const deletedData = this.tail.data;

    if (this.head === this.tail) {
      this.head = this.tail = null;
    } else {
      this.tail = this.tail.prev;
      this.tail.next = null;
    }

    this.size--;
    return deletedData;
  }

  // Convert to array (forward)
  toArray() {
    const result = [];
    let current = this.head;

    while (current) {
      result.push(current.data);
      current = current.next;
    }

    return result;
  }
```

```
  // Convert to array (backward)
  toArrayReverse() {
    const result = [];
    let current = this.tail;

    while (current) {
      result.push(current.data);
      current = current.prev;
    }

    return result;
  }
}

// Example Usage
const sll = new SinglyLinkedList();
sll.insertAtHead(1).insertAtHead(2).insertAtTail(3).insertAt(1, 5);
console.log("Singly Linked List:", sll.toArray()); // [2, 5, 1, 3]
console.log("Middle element:", sll.findMiddle()); // 5
sll.reverse();
console.log("After reverse:", sll.toArray()); // [3, 1, 5, 2]

const dll = new DoublyLinkedList();
dll.insertAtHead(1).insertAtTail(2).insertAtHead(0);
console.log("Doubly Linked List:", dll.toArray()); // [0, 1, 2]
console.log("Reverse traversal:", dll.toArrayReverse()); // [2, 1, 0]
```

🔧 C++ Implementation

```
#include <iostream>
#include <vector>
using namespace std;

// Node structure for singly linked list
struct ListNode {
    int data;
    ListNode* next;

    ListNode(int val) : data(val), next(nullptr) {}
};

// Singly Linked List Class
class SinglyLinkedList {
private:
    ListNode* head;
    int size;

public:
    SinglyLinkedList() : head(nullptr), size(0) {}
```

```
    // Destructor to prevent memory leaks
    ~SinglyLinkedList() {
        clear();
    }

    // Insert at the beginning
    // Time: O(1), Space: O(1)
    void insertAtHead(int data) {
        ListNode* newNode = new ListNode(data);
        newNode->next = head;
        head = newNode;
        size++;
    }

    // Insert at the end
    // Time: O(n), Space: O(1)
    void insertAtTail(int data) {
        ListNode* newNode = new ListNode(data);

        if (!head) {
            head = newNode;
        } else {
            ListNode* current = head;
            while (current->next) {
                current = current->next;
            }
            current->next = newNode;
        }

        size++;
    }

    // Insert at specific index
    // Time: O(n), Space: O(1)
    void insertAt(int index, int data) {
        if (index < 0 || index > size) {
            throw out_of_range("Index out of bounds");
        }

        if (index == 0) {
            insertAtHead(data);
            return;
        }

        ListNode* newNode = new ListNode(data);
        ListNode* current = head;

        // Traverse to position index-1
        for (int i = 0; i < index - 1; i++) {
            current = current->next;
        }

        newNode->next = current->next;
        current->next = newNode;
```

```
        size++;
    }

    // Delete from head
    // Time: O(1), Space: O(1)
    bool deleteHead() {
        if (!head) {
            return false;
        }

        ListNode* temp = head;
        head = head->next;
        delete temp;
        size--;
        return true;
    }

    // Delete by value
    // Time: O(n), Space: O(1)
    bool deleteByValue(int value) {
        if (!head) {
            return false;
        }

        if (head->data == value) {
            return deleteHead();
        }

        ListNode* current = head;
        while (current->next && current->next->data != value) {
            current = current->next;
        }

        if (current->next) {
            ListNode* temp = current->next;
            current->next = current->next->next;
            delete temp;
            size--;
            return true;
        }

        return false;
    }

    // Search for value
    // Time: O(n), Space: O(1)
    int search(int value) {
        ListNode* current = head;
        int index = 0;

        while (current) {
            if (current->data == value) {
                return index;
            }
```

```
            current = current->next;
            index++;
        }

        return -1;
    }

    // Reverse the linked list
    // Time: O(n), Space: O(1)
    void reverse() {
        ListNode* prev = nullptr;
        ListNode* current = head;
        ListNode* next = nullptr;

        while (current) {
            next = current->next;  // Store next node
            current->next = prev;  // Reverse the link
            prev = current;        // Move prev forward
            current = next;        // Move current forward
        }

        head = prev;
    }

    // Find middle element using Floyd's algorithm
    // Time: O(n), Space: O(1)
    int findMiddle() {
        if (!head) {
            throw runtime_error("List is empty");
        }

        ListNode* slow = head;
        ListNode* fast = head;

        while (fast && fast->next) {
            slow = slow->next;
            fast = fast->next->next;
        }

        return slow->data;
    }

    // Detect cycle in linked list
    // Time: O(n), Space: O(1)
    bool hasCycle() {
        if (!head) return false;

        ListNode* slow = head;
        ListNode* fast = head;

        while (fast && fast->next) {
            slow = slow->next;
            fast = fast->next->next;
```

```
            if (slow == fast) {
                return true;
            }
        }

        return false;
    }

    // Merge two sorted linked lists
    // Time: O(m + n), Space: O(1)
    static ListNode* mergeSorted(ListNode* l1, ListNode* l2) {
        ListNode dummy(0);
        ListNode* current = &dummy;

        while (l1 && l2) {
            if (l1->data <= l2->data) {
                current->next = l1;
                l1 = l1->next;
            } else {
                current->next = l2;
                l2 = l2->next;
            }
            current = current->next;
        }

        // Attach remaining nodes
        current->next = l1 ? l1 : l2;

        return dummy.next;
    }

    // Display the list
    void display() {
        ListNode* current = head;
        cout << "[";

        while (current) {
            cout << current->data;
            if (current->next) cout << " -> ";
            current = current->next;
        }

        cout << "]" << endl;
    }

    // Get size
    int getSize() {
        return size;
    }

    // Clear all nodes
    void clear() {
        while (head) {
            ListNode* temp = head;
```

```
            head = head->next;
            delete temp;
        }
        size = 0;
    }

    // Get head for external operations
    ListNode* getHead() {
        return head;
    }
};

// Doubly Linked List Node
struct DoublyListNode {
    int data;
    DoublyListNode* next;
    DoublyListNode* prev;

    DoublyListNode(int val) : data(val), next(nullptr), prev(nullptr) {}
};

// Doubly Linked List Class
class DoublyLinkedList {
private:
    DoublyListNode* head;
    DoublyListNode* tail;
    int size;

public:
    DoublyLinkedList() : head(nullptr), tail(nullptr), size(0) {}

    // Destructor
    ~DoublyLinkedList() {
        clear();
    }

    // Insert at head
    // Time: O(1), Space: O(1)
    void insertAtHead(int data) {
        DoublyListNode* newNode = new DoublyListNode(data);

        if (!head) {
            head = tail = newNode;
        } else {
            newNode->next = head;
            head->prev = newNode;
            head = newNode;
        }

        size++;
    }

    // Insert at tail
    // Time: O(1), Space: O(1)
```

```
    void insertAtTail(int data) {
        DoublyListNode* newNode = new DoublyListNode(data);

        if (!tail) {
            head = tail = newNode;
        } else {
            tail->next = newNode;
            newNode->prev = tail;
            tail = newNode;
        }

        size++;
    }

    // Delete from head
    // Time: O(1), Space: O(1)
    bool deleteHead() {
        if (!head) return false;

        if (head == tail) {
            delete head;
            head = tail = nullptr;
        } else {
            DoublyListNode* temp = head;
            head = head->next;
            head->prev = nullptr;
            delete temp;
        }

        size--;
        return true;
    }

    // Delete from tail
    // Time: O(1), Space: O(1)
    bool deleteTail() {
        if (!tail) return false;

        if (head == tail) {
            delete tail;
            head = tail = nullptr;
        } else {
            DoublyListNode* temp = tail;
            tail = tail->prev;
            tail->next = nullptr;
            delete temp;
        }

        size--;
        return true;
    }

    // Display forward
    void displayForward() {
```

```
        DoublyListNode* current = head;
        cout << "Forward: [";

        while (current) {
            cout << current->data;
            if (current->next) cout << " <-> ";
            current = current->next;
        }

        cout << "]" << endl;
    }

    // Display backward
    void displayBackward() {
        DoublyListNode* current = tail;
        cout << "Backward: [";

        while (current) {
            cout << current->data;
            if (current->prev) cout << " <-> ";
            current = current->prev;
        }

        cout << "]" << endl;
    }

    // Clear all nodes
    void clear() {
        while (head) {
            DoublyListNode* temp = head;
            head = head->next;
            delete temp;
        }
        head = tail = nullptr;
        size = 0;
    }

    int getSize() {
        return size;
    }
};

// Example Usage
int main() {
    cout << "=== Singly Linked List Demo ===" << endl;
    SinglyLinkedList sll;

    sll.insertAtHead(1);
    sll.insertAtHead(2);
    sll.insertAtTail(3);
    sll.insertAt(1, 5);

    cout << "Original list: ";
    sll.display();
```

```
    cout << "Middle element: " << sll.findMiddle() << endl;
    cout << "Search for 5: " << sll.search(5) << endl;

    sll.reverse();
    cout << "After reverse: ";
    sll.display();

    cout << "\n=== Doubly Linked List Demo ===" << endl;
    DoublyLinkedList dll;

    dll.insertAtHead(1);
    dll.insertAtTail(2);
    dll.insertAtHead(0);

    dll.displayForward();
    dll.displayBackward();

    return 0;
}
```

⚡ Performance Analysis

Time Complexity Comparison:

Operation	Array	Singly Linked List	Doubly Linked List
Access	O(1)	O(n)	O(n)
Search	O(n)	O(n)	O(n)
Insert at head	O(n)	O(1)	O(1)
Insert at tail	O(1)	O(n)	O(1)
Delete at head	O(n)	O(1)	O(1)
Delete at tail	O(1)	O(n)	O(1)

Space Complexity:

- **Singly Linked List**: O(n) - one pointer per node
- **Doubly Linked List**: O(n) - two pointers per node
- **Additional space per operation**: O(1)

Common Pitfalls:

1. **Memory leaks**: Always delete nodes in C++
2. **Null pointer access**: Check for null before dereferencing
3. **Lost references**: Keep track of nodes during operations
4. **Infinite loops**: Be careful with cycle detection

Practice Problems

Problem 1: Remove Nth Node from End

Question: Given a linked list, remove the nth node from the end.

Hint: Use two pointers with n distance between them.

Solution Approach:

```
function removeNthFromEnd(head, n) {
  const dummy = new ListNode(0);
  dummy.next = head;
  let first = dummy;
  let second = dummy;

  // Move first pointer n+1 steps ahead
  for (let i = 0; i <= n; i++) {
    first = first.next;
  }

  // Move both pointers until first reaches end
  while (first) {
    first = first.next;
    second = second.next;
  }

  // Remove the nth node
  second.next = second.next.next;
  return dummy.next;
}
```

Problem 2: Merge Two Sorted Lists

Question: Merge two sorted linked lists into one sorted list.

Hint: Use a dummy node and compare values iteratively.

Problem 3: Linked List Cycle II

Question: Find the starting node of a cycle in a linked list.

Hint: Use Floyd's algorithm, then find the intersection point.

Problem 4: Palindrome Linked List

Question: Check if a linked list is a palindrome.

Hint: Find middle, reverse second half, compare with first half.

Interview Tips

What Interviewers Look For:

1. **Pointer manipulation**: Can you handle next/prev pointers correctly?
2. **Edge cases**: Empty lists, single nodes, cycles
3. **Memory management**: Proper allocation/deallocation in C++
4. **Algorithm optimization**: Using techniques like Floyd's cycle detection

Common Interview Patterns:

- **Two Pointers**: Fast/slow pointers for cycle detection, finding middle
- **Dummy Nodes**: Simplify edge cases in insertion/deletion
- **Recursion**: For problems like reversing or merging
- **Stack**: For problems requiring backtracking

Red Flags to Avoid:

- Modifying input without permission
- Not handling null pointers
- Creating memory leaks in C++
- Not considering edge cases (empty list, single node)

Pro Tips:

1. **Draw it out**: Visualize pointer movements
2. **Use dummy nodes**: Simplifies many operations
3. **Check for cycles**: Always consider if cycles are possible
4. **Practice pointer arithmetic**: Master the fundamentals

🚀 Key Takeaways

1. **Linked lists excel at insertion/deletion** - O(1) at known positions
2. **Trade-off with arrays** - Dynamic size vs. random access
3. **Pointer manipulation is crucial** - Practice makes perfect
4. **Memory management matters** - Especially in C++
5. **Many algorithms use two pointers** - Fast/slow, leading/trailing

Next Chapter: We'll explore Stacks & Queues and see how they can be implemented using linked lists and arrays.

Chapter 3: Stacks & Queues - LIFO and FIFO Data Structures

🎯 What Are Stacks & Queues?

Stack is a linear data structure that follows the Last In, First Out (LIFO) principle. Think of it like a stack of plates - you can only add or remove plates from the top.

Queue is a linear data structure that follows the First In, First Out (FIFO) principle. Think of it like a line at a store - the first person in line is the first to be served.

Why Stacks & Queues Matter:

- **Function calls**: Call stack in programming languages
- **Undo operations**: Text editors, browsers
- **Expression evaluation**: Parsing mathematical expressions
- **BFS/DFS**: Graph traversal algorithms
- **Task scheduling**: Operating systems, print queues

🔍 Stack Operations

Core Operations:

1. **Push**: Add element to the top
2. **Pop**: Remove element from the top
3. **Peek/Top**: View the top element without removing
4. **isEmpty**: Check if stack is empty
5. **Size**: Get number of elements

Applications:

- **Expression evaluation** (infix to postfix)
- **Parentheses matching**
- **Function call management**
- **Undo/Redo operations**
- **Depth-First Search (DFS)**

🔍 Queue Operations

Core Operations:

1. **Enqueue**: Add element to the rear
2. **Dequeue**: Remove element from the front
3. **Front**: View the front element
4. **Rear**: View the rear element
5. **isEmpty**: Check if queue is empty
6. **Size**: Get number of elements

Types of Queues:

- **Simple Queue**: Basic FIFO queue
- **Circular Queue**: Rear connects back to front
- **Priority Queue**: Elements have priorities
- **Deque**: Double-ended queue (insertion/deletion at both ends)

💻 JavaScript Implementation

```
// Stack Implementation using Array
class Stack {
  constructor() {
    this.items = [];
  }

  // Push element to top
  // Time: O(1), Space: O(1)
  push(element) {
    this.items.push(element);
    return this.size();
  }

  // Pop element from top
  // Time: O(1), Space: O(1)
  pop() {
    if (this.isEmpty()) {
      throw new Error("Stack is empty");
    }
    return this.items.pop();
  }

  // Peek at top element
  // Time: O(1), Space: O(1)
  peek() {
    if (this.isEmpty()) {
      throw new Error("Stack is empty");
    }
    return this.items[this.items.length - 1];
  }

  // Check if stack is empty
  isEmpty() {
    return this.items.length === 0;
  }

  // Get stack size
  size() {
    return this.items.length;
  }

  // Clear stack
  clear() {
    this.items = [];
  }

  // Convert to array for display
  toArray() {
    return [...this.items];
  }
```

```
}

// Stack Implementation using Linked List
class StackNode {
  constructor(data) {
    this.data = data;
    this.next = null;
  }
}

class LinkedStack {
  constructor() {
    this.top = null;
    this.count = 0;
  }

  // Push element
  // Time: O(1), Space: O(1)
  push(data) {
    const newNode = new StackNode(data);
    newNode.next = this.top;
    this.top = newNode;
    this.count++;
    return this.count;
  }

  // Pop element
  // Time: O(1), Space: O(1)
  pop() {
    if (this.isEmpty()) {
      throw new Error("Stack is empty");
    }

    const poppedData = this.top.data;
    this.top = this.top.next;
    this.count--;
    return poppedData;
  }

  // Peek at top
  peek() {
    if (this.isEmpty()) {
      throw new Error("Stack is empty");
    }
    return this.top.data;
  }

  isEmpty() {
    return this.top === null;
  }

  size() {
    return this.count;
  }
```

```
}

// Queue Implementation using Array
class Queue {
  constructor() {
    this.items = [];
  }

  // Add element to rear
  // Time: O(1), Space: O(1)
  enqueue(element) {
    this.items.push(element);
    return this.size();
  }

  // Remove element from front
  // Time: O(n) due to array shift, Space: O(1)
  dequeue() {
    if (this.isEmpty()) {
      throw new Error("Queue is empty");
    }
    return this.items.shift();
  }

  // View front element
  front() {
    if (this.isEmpty()) {
      throw new Error("Queue is empty");
    }
    return this.items[0];
  }

  // View rear element
  rear() {
    if (this.isEmpty()) {
      throw new Error("Queue is empty");
    }
    return this.items[this.items.length - 1];
  }

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

  size() {
    return this.items.length;
  }

  toArray() {
    return [...this.items];
  }
}

// Optimized Queue using Two Pointers
```

```
class OptimizedQueue {
  constructor() {
    this.items = [];
    this.frontIndex = 0;
    this.rearIndex = 0;
  }

  // Enqueue operation
  // Time: O(1), Space: O(1)
  enqueue(element) {
    this.items[this.rearIndex] = element;
    this.rearIndex++;
    return this.size();
  }

  // Dequeue operation
  // Time: O(1), Space: O(1)
  dequeue() {
    if (this.isEmpty()) {
      throw new Error("Queue is empty");
    }

    const dequeuedElement = this.items[this.frontIndex];
    delete this.items[this.frontIndex];
    this.frontIndex++;

    // Reset pointers when queue becomes empty
    if (this.frontIndex === this.rearIndex) {
      this.frontIndex = 0;
      this.rearIndex = 0;
    }

    return dequeuedElement;
  }

  front() {
    if (this.isEmpty()) {
      throw new Error("Queue is empty");
    }
    return this.items[this.frontIndex];
  }

  isEmpty() {
    return this.frontIndex === this.rearIndex;
  }

  size() {
    return this.rearIndex - this.frontIndex;
  }
}

// Circular Queue Implementation
class CircularQueue {
  constructor(capacity) {
```

```
    this.capacity = capacity;
    this.items = new Array(capacity);
    this.front = -1;
    this.rear = -1;
    this.count = 0;
  }

  // Check if queue is full
  isFull() {
    return this.count === this.capacity;
  }

  isEmpty() {
    return this.count === 0;
  }

  // Enqueue operation
  // Time: O(1), Space: O(1)
  enqueue(element) {
    if (this.isFull()) {
      throw new Error("Queue is full");
    }

    if (this.isEmpty()) {
      this.front = 0;
      this.rear = 0;
    } else {
      this.rear = (this.rear + 1) % this.capacity;
    }

    this.items[this.rear] = element;
    this.count++;
    return this.count;
  }

  // Dequeue operation
  // Time: O(1), Space: O(1)
  dequeue() {
    if (this.isEmpty()) {
      throw new Error("Queue is empty");
    }

    const dequeuedElement = this.items[this.front];
    this.items[this.front] = undefined;

    if (this.count === 1) {
      this.front = -1;
      this.rear = -1;
    } else {
      this.front = (this.front + 1) % this.capacity;
    }

    this.count--;
    return dequeuedElement;
```

```
  }

  getFront() {
    if (this.isEmpty()) {
      throw new Error("Queue is empty");
    }
    return this.items[this.front];
  }

  getRear() {
    if (this.isEmpty()) {
      throw new Error("Queue is empty");
    }
    return this.items[this.rear];
  }

  size() {
    return this.count;
  }

  display() {
    if (this.isEmpty()) {
      return [];
    }

    const result = [];
    let i = this.front;
    let itemsAdded = 0;

    while (itemsAdded < this.count) {
      result.push(this.items[i]);
      i = (i + 1) % this.capacity;
      itemsAdded++;
    }

    return result;
  }
}

// Stack Applications
class StackApplications {
  // Check balanced parentheses
  // Time: O(n), Space: O(n)
  static isBalanced(expression) {
    const stack = new Stack();
    const pairs = {
      ")": "(",
      "}": "{",
      "]": "[",
    };

    for (let char of expression) {
      if (char === "(" || char === "{" || char === "[") {
        stack.push(char);
```

```
      } else if (char === ")" || char === "}" || char === "]") {
        if (stack.isEmpty() || stack.pop() !== pairs[char]) {
          return false;
        }
      }
    }

    return stack.isEmpty();
  }

  // Convert infix to postfix
  // Time: O(n), Space: O(n)
  static infixToPostfix(infix) {
    const stack = new Stack();
    let postfix = "";
    const precedence = {
      "+": 1,
      "-": 1,
      "*": 2,
      "/": 2,
      "^": 3,
    };

    for (let char of infix) {
      if (/[a-zA-Z0-9]/.test(char)) {
        postfix += char;
      } else if (char === "(") {
        stack.push(char);
      } else if (char === ")") {
        while (!stack.isEmpty() && stack.peek() !== "(") {
          postfix += stack.pop();
        }
        stack.pop(); // Remove '('
      } else if (precedence[char]) {
        while (
          !stack.isEmpty() &&
          stack.peek() !== "(" &&
          precedence[stack.peek()] >= precedence[char]
        ) {
          postfix += stack.pop();
        }
        stack.push(char);
      }
    }

    while (!stack.isEmpty()) {
      postfix += stack.pop();
    }

    return postfix;
  }

  // Evaluate postfix expression
  // Time: O(n), Space: O(n)
```

```
  static evaluatePostfix(postfix) {
    const stack = new Stack();

    for (let char of postfix) {
      if (/\d/.test(char)) {
        stack.push(parseInt(char));
      } else {
        const b = stack.pop();
        const a = stack.pop();

        switch (char) {
          case "+":
            stack.push(a + b);
            break;
          case "-":
            stack.push(a - b);
            break;
          case "*":
            stack.push(a * b);
            break;
          case "/":
            stack.push(Math.floor(a / b));
            break;
        }
      }
    }

    return stack.pop();
  }
}

// Example Usage
console.log("=== Stack Demo ===");
const stack = new Stack();
stack.push(1);
stack.push(2);
stack.push(3);
console.log("Stack:", stack.toArray()); // [1, 2, 3]
console.log("Pop:", stack.pop()); // 3
console.log("Peek:", stack.peek()); // 2

console.log("\n=== Queue Demo ===");
const queue = new Queue();
queue.enqueue("A");
queue.enqueue("B");
queue.enqueue("C");
console.log("Queue:", queue.toArray()); // ['A', 'B', 'C']
console.log("Dequeue:", queue.dequeue()); // 'A'
console.log("Front:", queue.front()); // 'B'

console.log("\n=== Circular Queue Demo ===");
const circularQueue = new CircularQueue(3);
circularQueue.enqueue(1);
circularQueue.enqueue(2);
```

```
circularQueue.enqueue(3);
console.log("Circular Queue:", circularQueue.display()); // [1, 2, 3]
circularQueue.dequeue();
circularQueue.enqueue(4);
console.log("After dequeue and enqueue:", circularQueue.display()); // [2, 3, 4]

console.log("\n=== Stack Applications ===");
console.log('Balanced "({[]})": ', StackApplications.isBalanced("({[]})")); //
true
console.log(
  'Infix "A+B*C" to postfix:',
  StackApplications.infixToPostfix("A+B*C")
); // ABC*+
console.log('Evaluate "23*4+":', StackApplications.evaluatePostfix("23*4+")); //
10
```

🔧 C++ Implementation

```
#include <iostream>
#include <vector>
#include <string>
#include <stdexcept>
#include <stack>
#include <queue>
using namespace std;

// Stack Implementation using Array
template<typename T>
class ArrayStack {
private:
    vector<T> items;

public:
    // Push element to top
    // Time: O(1), Space: O(1)
    void push(const T& element) {
        items.push_back(element);
    }

    // Pop element from top
    // Time: O(1), Space: O(1)
    T pop() {
        if (isEmpty()) {
            throw runtime_error("Stack is empty");
        }

        T topElement = items.back();
        items.pop_back();
        return topElement;
    }
```

```
    // Peek at top element
    // Time: O(1), Space: O(1)
    T peek() const {
        if (isEmpty()) {
            throw runtime_error("Stack is empty");
        }
        return items.back();
    }

    // Check if stack is empty
    bool isEmpty() const {
        return items.empty();
    }

    // Get stack size
    size_t size() const {
        return items.size();
    }

    // Display stack
    void display() const {
        cout << "Stack (top to bottom): [";
        for (int i = items.size() - 1; i >= 0; i--) {
            cout << items[i];
            if (i > 0) cout << ", ";
        }
        cout << "]" << endl;
    }
};

// Stack Implementation using Linked List
template<typename T>
struct StackNode {
    T data;
    StackNode* next;

    StackNode(const T& value) : data(value), next(nullptr) {}
};

template<typename T>
class LinkedStack {
private:
    StackNode<T>* top;
    size_t count;

public:
    LinkedStack() : top(nullptr), count(0) {}

    // Destructor
    ~LinkedStack() {
        clear();
    }
```

```
    // Push element
    // Time: O(1), Space: O(1)
    void push(const T& data) {
        StackNode<T>* newNode = new StackNode<T>(data);
        newNode->next = top;
        top = newNode;
        count++;
    }

    // Pop element
    // Time: O(1), Space: O(1)
    T pop() {
        if (isEmpty()) {
            throw runtime_error("Stack is empty");
        }

        T poppedData = top->data;
        StackNode<T>* temp = top;
        top = top->next;
        delete temp;
        count--;
        return poppedData;
    }

    // Peek at top
    T peek() const {
        if (isEmpty()) {
            throw runtime_error("Stack is empty");
        }
        return top->data;
    }

    bool isEmpty() const {
        return top == nullptr;
    }

    size_t size() const {
        return count;
    }

    // Clear all elements
    void clear() {
        while (!isEmpty()) {
            pop();
        }
    }
};

// Circular Queue Implementation
template<typename T>
class CircularQueue {
private:
    vector<T> items;
    int front;
```

```
    int rear;
    int capacity;
    int count;

public:
    CircularQueue(int cap) : capacity(cap), front(-1), rear(-1), count(0) {
        items.resize(capacity);
    }

    // Check if queue is full
    bool isFull() const {
        return count == capacity;
    }

    // Check if queue is empty
    bool isEmpty() const {
        return count == 0;
    }

    // Enqueue operation
    // Time: O(1), Space: O(1)
    void enqueue(const T& element) {
        if (isFull()) {
            throw runtime_error("Queue is full");
        }

        if (isEmpty()) {
            front = 0;
            rear = 0;
        } else {
            rear = (rear + 1) % capacity;
        }

        items[rear] = element;
        count++;
    }

    // Dequeue operation
    // Time: O(1), Space: O(1)
    T dequeue() {
        if (isEmpty()) {
            throw runtime_error("Queue is empty");
        }

        T dequeuedElement = items[front];

        if (count == 1) {
            front = -1;
            rear = -1;
        } else {
            front = (front + 1) % capacity;
        }

        count--;
```

```
        return dequeuedElement;
    }

    // Get front element
    T getFront() const {
        if (isEmpty()) {
            throw runtime_error("Queue is empty");
        }
        return items[front];
    }

    // Get rear element
    T getRear() const {
        if (isEmpty()) {
            throw runtime_error("Queue is empty");
        }
        return items[rear];
    }

    // Get size
    int size() const {
        return count;
    }

    // Display queue
    void display() const {
        if (isEmpty()) {
            cout << "Queue is empty" << endl;
            return;
        }

        cout << "Queue: [";
        int i = front;
        int itemsDisplayed = 0;

        while (itemsDisplayed < count) {
            cout << items[i];
            if (itemsDisplayed < count - 1) cout << ", ";
            i = (i + 1) % capacity;
            itemsDisplayed++;
        }
        cout << "]" << endl;
    }
};

// Stack Applications
class StackApplications {
public:
    // Check balanced parentheses
    // Time: O(n), Space: O(n)
    static bool isBalanced(const string& expression) {
        stack<char> s;

        for (char ch : expression) {
```

```
            if (ch == '(' || ch == '{' || ch == '[') {
                s.push(ch);
            } else if (ch == ')' || ch == '}' || ch == ']') {
                if (s.empty()) return false;

                char top = s.top();
                s.pop();

                if ((ch == ')' && top != '(') ||
                    (ch == '}' && top != '{') ||
                    (ch == ']' && top != '[')) {
                    return false;
                }
            }
        }

        return s.empty();
    }

    // Get precedence of operator
    static int getPrecedence(char op) {
        switch (op) {
            case '+': case '-': return 1;
            case '*': case '/': return 2;
            case '^': return 3;
            default: return 0;
        }
    }

    // Convert infix to postfix
    // Time: O(n), Space: O(n)
    static string infixToPostfix(const string& infix) {
        stack<char> s;
        string postfix = "";

        for (char ch : infix) {
            if (isalnum(ch)) {
                postfix += ch;
            } else if (ch == '(') {
                s.push(ch);
            } else if (ch == ')') {
                while (!s.empty() && s.top() != '(') {
                    postfix += s.top();
                    s.pop();
                }
                if (!s.empty()) s.pop(); // Remove '('
            } else if (getPrecedence(ch) > 0) {
                while (!s.empty() &&
                       s.top() != '(' &&
                       getPrecedence(s.top()) >= getPrecedence(ch)) {
                    postfix += s.top();
                    s.pop();
                }
                s.push(ch);
```

```
            }
        }

        while (!s.empty()) {
            postfix += s.top();
            s.pop();
        }

        return postfix;
    }

    // Evaluate postfix expression
    // Time: O(n), Space: O(n)
    static int evaluatePostfix(const string& postfix) {
        stack<int> s;

        for (char ch : postfix) {
            if (isdigit(ch)) {
                s.push(ch - '0');
            } else {
                int b = s.top(); s.pop();
                int a = s.top(); s.pop();

                switch (ch) {
                    case '+': s.push(a + b); break;
                    case '-': s.push(a - b); break;
                    case '*': s.push(a * b); break;
                    case '/': s.push(a / b); break;
                }
            }
        }

        return s.top();
    }
};

// Example Usage
int main() {
    cout << "=== Array Stack Demo ===" << endl;
    ArrayStack<int> arrayStack;
    arrayStack.push(1);
    arrayStack.push(2);
    arrayStack.push(3);
    arrayStack.display();
    cout << "Pop: " << arrayStack.pop() << endl;
    cout << "Peek: " << arrayStack.peek() << endl;

    cout << "\n=== Linked Stack Demo ===" << endl;
    LinkedStack<string> linkedStack;
    linkedStack.push("First");
    linkedStack.push("Second");
    linkedStack.push("Third");
    cout << "Size: " << linkedStack.size() << endl;
    cout << "Pop: " << linkedStack.pop() << endl;
```

```
    cout << "\n=== Circular Queue Demo ===" << endl;
    CircularQueue<int> cq(3);
    cq.enqueue(1);
    cq.enqueue(2);
    cq.enqueue(3);
    cq.display();
    cout << "Dequeue: " << cq.dequeue() << endl;
    cq.enqueue(4);
    cq.display();

    cout << "\n=== Stack Applications ===" << endl;
    cout << "Balanced \"({[]})\": " << (StackApplications::isBalanced("({[]})") ?
"Yes" : "No") << endl;
    cout << "Infix \"A+B*C\" to postfix: " <<
StackApplications::infixToPostfix("A+B*C") << endl;
    cout << "Evaluate \"23*4+\": " << StackApplications::evaluatePostfix("23*4+")
<< endl;

    return 0;
}
```

⚡ Performance Analysis

Time Complexity:

Operation	Array Stack	Linked Stack	Array Queue	Circular Queue
Push/Enqueue	O(1)	O(1)	O(1)	O(1)
Pop/Dequeue	O(1)	O(1)	O(n)*	O(1)
Peek/Front	O(1)	O(1)	O(1)	O(1)
Size	O(1)	O(1)	O(1)	O(1)

*Array queue dequeue is O(n) due to shifting elements

Space Complexity:

- **Array-based**: O(n) where n is the number of elements
- **Linked-based**: O(n) + pointer overhead
- **Circular Queue**: O(k) where k is the fixed capacity

Common Pitfalls:

1. **Stack overflow**: Pushing too many elements
2. **Queue overflow**: In fixed-size implementations
3. **Underflow**: Popping from empty structures
4. **Memory leaks**: Not properly deallocating in C++

🧩 Practice Problems

Problem 1: Valid Parentheses

Question: Given a string containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

Solution:

```
function isValid(s) {
  const stack = [];
  const map = { ")": "(", "}": "{", "]": "[" };

  for (let char of s) {
    if (char in map) {
      if (stack.pop() !== map[char]) return false;
    } else {
      stack.push(char);
    }
  }

  return stack.length === 0;
}
```

Problem 2: Implement Queue using Stacks

Question: Implement a first in first out (FIFO) queue using only two stacks.

Hint: Use one stack for enqueue and another for dequeue operations.

Problem 3: Next Greater Element

Question: Find the next greater element for each element in an array.

Hint: Use a stack to keep track of elements for which we haven't found the next greater element yet.

Problem 4: Sliding Window Maximum

Question: Find the maximum element in each sliding window of size k.

Hint: Use a deque to maintain elements in decreasing order.

🎯 Interview Tips

What Interviewers Look For:

1. **Understanding of LIFO/FIFO**: Can you explain the principles clearly?
2. **Implementation choices**: When to use array vs. linked list?
3. **Edge case handling**: Empty structures, overflow/underflow
4. **Application knowledge**: Real-world uses of stacks and queues

Common Interview Patterns:

- **Monotonic Stack**: For next greater/smaller element problems
- **Two Stacks**: To implement queue or undo/redo functionality
- **BFS with Queue**: Level-order traversal, shortest path
- **DFS with Stack**: Tree/graph traversal, backtracking

Red Flags to Avoid:

- Not checking for empty structures before operations
- Confusing LIFO and FIFO principles
- Not considering space complexity in recursive solutions
- Implementing inefficient queue operations

Pro Tips:

1. **Visualize the operations**: Draw the stack/queue state
2. **Consider circular arrays**: For efficient queue implementation
3. **Think about applications**: Expression evaluation, BFS/DFS
4. **Practice with constraints**: Fixed size vs. dynamic size

---

🚀 Key Takeaways

1. **Stacks are perfect for LIFO scenarios** - Function calls, undo operations
2. **Queues excel at FIFO processing** - Task scheduling, BFS
3. **Circular queues optimize space** - Better than simple array queues
4. **Choose implementation wisely** - Array vs. linked list trade-offs
5. **Master the applications** - Expression evaluation, parentheses matching

Next Chapter: We'll explore Hash Tables/Maps and learn about efficient key-value storage and retrieval.

Chapter 4: Hash Tables & Maps - Efficient Key-Value Storage

🎯 What Are Hash Tables?

Hash Tables (also called Hash Maps) are data structures that implement an associative array abstract data type, mapping keys to values. They use a hash function to compute an index into an array of buckets or slots, from which the desired value can be found.

Why Hash Tables Matter:

- **Fast Access**: Average O(1) time for search, insert, and delete
- **Flexible Keys**: Can use strings, numbers, or custom objects as keys
- **Memory Efficient**: Direct indexing eliminates need for comparisons
- **Ubiquitous**: Foundation for databases, caches, and many algorithms

Core Concepts:

1. **Hash Function**: Converts keys into array indices
2. **Buckets**: Array slots where key-value pairs are stored
3. **Collision**: When two keys hash to the same index
4. **Load Factor**: Ratio of stored elements to total capacity

---

🔍 Hash Functions & Collision Resolution

Hash Function Properties:

- **Deterministic**: Same key always produces same hash
- **Uniform Distribution**: Spreads keys evenly across buckets
- **Fast Computation**: Should be quick to calculate
- **Avalanche Effect**: Small key changes cause large hash changes

Collision Resolution Techniques:

1. Separate Chaining (Open Hashing)

- Each bucket contains a linked list of key-value pairs
- Multiple elements can exist in the same bucket

2. Open Addressing (Closed Hashing)

- All elements stored in the hash table itself
- When collision occurs, probe for next available slot
- **Linear Probing**: Check next slot sequentially
- **Quadratic Probing**: Check slots at quadratic intervals
- **Double Hashing**: Use second hash function for probing

💻 JavaScript Implementation

```
// Hash Table with Separate Chaining
class HashTableChaining {
  constructor(size = 10) {
    this.size = size;
    this.buckets = new Array(size);
    this.count = 0;

    // Initialize buckets as empty arrays
    for (let i = 0; i < size; i++) {
      this.buckets[i] = [];
    }
  }

  // Simple hash function
  // Time: O(k) where k is key length, Space: O(1)
```

```
    hash(key) {
      let hash = 0;
      const keyStr = String(key);

      for (let i = 0; i < keyStr.length; i++) {
        hash = (hash + keyStr.charCodeAt(i) * (i + 1)) % this.size;
      }

      return hash;
    }

    // Insert or update key-value pair
    // Average Time: O(1), Worst Time: O(n), Space: O(1)
    set(key, value) {
      const index = this.hash(key);
      const bucket = this.buckets[index];

      // Check if key already exists
      for (let i = 0; i < bucket.length; i++) {
        if (bucket[i][0] === key) {
          bucket[i][1] = value; // Update existing
          return;
        }
      }

      // Add new key-value pair
      bucket.push([key, value]);
      this.count++;

      // Resize if load factor exceeds threshold
      if (this.count > this.size * 0.75) {
        this.resize();
      }
    }

    // Get value by key
    // Average Time: O(1), Worst Time: O(n), Space: O(1)
    get(key) {
      const index = this.hash(key);
      const bucket = this.buckets[index];

      for (let i = 0; i < bucket.length; i++) {
        if (bucket[i][0] === key) {
          return bucket[i][1];
        }
      }

      return undefined;
    }

    // Check if key exists
    // Average Time: O(1), Worst Time: O(n), Space: O(1)
    has(key) {
      return this.get(key) !== undefined;
```

```
  }

  // Delete key-value pair
  // Average Time: O(1), Worst Time: O(n), Space: O(1)
  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);
        this.count--;
        return true;
      }
    }

    return false;
  }

  // Get all keys
  keys() {
    const keys = [];
    for (let bucket of this.buckets) {
      for (let [key] of bucket) {
        keys.push(key);
      }
    }
    return keys;
  }

  // Get all values
  values() {
    const values = [];
    for (let bucket of this.buckets) {
      for (let [, value] of bucket) {
        values.push(value);
      }
    }
    return values;
  }

  // Resize hash table when load factor is high
  resize() {
    const oldBuckets = this.buckets;
    this.size *= 2;
    this.buckets = new Array(this.size);
    this.count = 0;

    // Initialize new buckets
    for (let i = 0; i < this.size; i++) {
      this.buckets[i] = [];
    }

    // Rehash all existing elements
```

```
    for (let bucket of oldBuckets) {
      for (let [key, value] of bucket) {
        this.set(key, value);
      }
    }
  }

  // Get load factor
  getLoadFactor() {
    return this.count / this.size;
  }

  // Display hash table structure
  display() {
    console.log("Hash Table Structure:");
    for (let i = 0; i < this.buckets.length; i++) {
      if (this.buckets[i].length > 0) {
        console.log(`Bucket ${i}:`, this.buckets[i]);
      }
    }
    console.log(`Load Factor: ${this.getLoadFactor().toFixed(2)}`);
  }
}

// Hash Table with Linear Probing
class HashTableLinearProbing {
  constructor(size = 10) {
    this.size = size;
    this.keys = new Array(size);
    this.values = new Array(size);
    this.count = 0;
  }

  // Hash function
  hash(key) {
    let hash = 0;
    const keyStr = String(key);

    for (let i = 0; i < keyStr.length; i++) {
      hash = (hash + keyStr.charCodeAt(i) * 31) % this.size;
    }

    return hash;
  }

  // Insert or update key-value pair
  // Average Time: O(1), Worst Time: O(n), Space: O(1)
  set(key, value) {
    if (this.count >= this.size * 0.75) {
      this.resize();
    }

    let index = this.hash(key);
```

```
    // Linear probing to find empty slot or existing key
    while (this.keys[index] !== undefined) {
      if (this.keys[index] === key) {
        this.values[index] = value; // Update existing
        return;
      }
      index = (index + 1) % this.size;
    }

    // Insert new key-value pair
    this.keys[index] = key;
    this.values[index] = value;
    this.count++;
  }

  // Get value by key
  // Average Time: O(1), Worst Time: O(n), Space: O(1)
  get(key) {
    let index = this.hash(key);

    while (this.keys[index] !== undefined) {
      if (this.keys[index] === key) {
        return this.values[index];
      }
      index = (index + 1) % this.size;
    }

    return undefined;
  }

  // Delete key-value pair
  // Average Time: O(1), Worst Time: O(n), Space: O(1)
  delete(key) {
    let index = this.hash(key);

    while (this.keys[index] !== undefined) {
      if (this.keys[index] === key) {
        this.keys[index] = undefined;
        this.values[index] = undefined;
        this.count--;

        // Rehash subsequent elements to maintain clustering
        this.rehashCluster(index);
        return true;
      }
      index = (index + 1) % this.size;
    }

    return false;
  }

  // Rehash elements after deletion to maintain proper clustering
  rehashCluster(deletedIndex) {
    let index = (deletedIndex + 1) % this.size;
```

```
    while (this.keys[index] !== undefined) {
      const keyToRehash = this.keys[index];
      const valueToRehash = this.values[index];

      this.keys[index] = undefined;
      this.values[index] = undefined;
      this.count--;

      this.set(keyToRehash, valueToRehash);
      index = (index + 1) % this.size;
    }
  }

  // Resize hash table
  resize() {
    const oldKeys = this.keys;
    const oldValues = this.values;

    this.size *= 2;
    this.keys = new Array(this.size);
    this.values = new Array(this.size);
    this.count = 0;

    // Rehash all existing elements
    for (let i = 0; i < oldKeys.length; i++) {
      if (oldKeys[i] !== undefined) {
        this.set(oldKeys[i], oldValues[i]);
      }
    }
  }

  // Check if key exists
  has(key) {
    return this.get(key) !== undefined;
  }

  // Get all keys
  getKeys() {
    return this.keys.filter((key) => key !== undefined);
  }

  // Get load factor
  getLoadFactor() {
    return this.count / this.size;
  }
}

// Hash Set Implementation
class HashSet {
  constructor() {
    this.map = new HashTableChaining();
  }
```

```
    // Add element to set
    add(element) {
      this.map.set(element, true);
    }

    // Check if element exists
    has(element) {
      return this.map.has(element);
    }

    // Remove element from set
    delete(element) {
      return this.map.delete(element);
    }

    // Get all elements
    values() {
      return this.map.keys();
    }

    // Get size
    size() {
      return this.map.count;
    }

    // Clear all elements
    clear() {
      this.map = new HashTableChaining();
    }
  }

  // Hash Table Applications
  class HashTableApplications {
    // Find two numbers that sum to target
    // Time: O(n), Space: O(n)
    static twoSum(nums, target) {
      const map = new Map();

      for (let i = 0; i < nums.length; i++) {
        const complement = target - nums[i];
        if (map.has(complement)) {
          return [map.get(complement), i];
        }
        map.set(nums[i], i);
      }

      return [];
    }

    // Group anagrams together
    // Time: O(n * k log k), Space: O(n * k)
    static groupAnagrams(strs) {
      const map = new Map();
```

```
    for (let str of strs) {
      const sorted = str.split("").sort().join("");
      if (!map.has(sorted)) {
        map.set(sorted, []);
      }
      map.get(sorted).push(str);
    }

    return Array.from(map.values());
  }

  // Find first non-repeating character
  // Time: O(n), Space: O(1) - limited character set
  static firstUniqueChar(s) {
    const charCount = new Map();

    // Count character frequencies
    for (let char of s) {
      charCount.set(char, (charCount.get(char) || 0) + 1);
    }

    // Find first character with count 1
    for (let i = 0; i < s.length; i++) {
      if (charCount.get(s[i]) === 1) {
        return i;
      }
    }

    return -1;
  }

  // Implement LRU Cache
  static createLRUCache(capacity) {
    return new LRUCache(capacity);
  }
}

// LRU Cache Implementation
class LRUCache {
  constructor(capacity) {
    this.capacity = capacity;
    this.cache = new Map();
  }

  // Get value and mark as recently used
  // Time: O(1), Space: O(1)
  get(key) {
    if (this.cache.has(key)) {
      const value = this.cache.get(key);
      // Move to end (most recently used)
      this.cache.delete(key);
      this.cache.set(key, value);
      return value;
    }
```

```
    return -1;
  }

  // Put key-value pair
  // Time: O(1), Space: O(1)
  put(key, value) {
    if (this.cache.has(key)) {
      // Update existing key
      this.cache.delete(key);
    } else if (this.cache.size >= this.capacity) {
      // Remove least recently used (first item)
      const firstKey = this.cache.keys().next().value;
      this.cache.delete(firstKey);
    }

    this.cache.set(key, value);
  }

  // Display cache state
  display() {
    console.log("LRU Cache:", Array.from(this.cache.entries()));
  }
}

// Example Usage
console.log("=== Hash Table with Chaining Demo ===");
const hashTable = new HashTableChaining(5);
hashTable.set("name", "John");
hashTable.set("age", 30);
hashTable.set("city", "New York");
hashTable.set("country", "USA");
console.log("Get name:", hashTable.get("name")); // John
console.log("Has age:", hashTable.has("age")); // true
hashTable.display();

console.log("\n=== Hash Table with Linear Probing Demo ===");
const hashTableLP = new HashTableLinearProbing(7);
hashTableLP.set("apple", 5);
hashTableLP.set("banana", 3);
hashTableLP.set("orange", 8);
console.log("Get apple:", hashTableLP.get("apple")); // 5
console.log("Load factor:", hashTableLP.getLoadFactor().toFixed(2));

console.log("\n=== Hash Set Demo ===");
const hashSet = new HashSet();
hashSet.add("red");
hashSet.add("blue");
hashSet.add("green");
console.log("Has red:", hashSet.has("red")); // true
console.log("Set values:", hashSet.values());

console.log("\n=== Hash Table Applications ===");
console.log(
  "Two Sum [2,7,11,15], target 9:",
```

```
  HashTableApplications.twoSum([2, 7, 11, 15], 9)
); // [0,1]
console.log(
  "Group Anagrams:",
  HashTableApplications.groupAnagrams([
    "eat",
    "tea",
    "tan",
    "ate",
    "nat",
    "bat",
  ])
);
console.log(
  'First unique char in "leetcode":',
  HashTableApplications.firstUniqueChar("leetcode")
); // 0

console.log("\n=== LRU Cache Demo ===");
const lru = new LRUCache(2);
lru.put(1, 1);
lru.put(2, 2);
console.log("Get 1:", lru.get(1)); // 1
lru.put(3, 3); // Evicts key 2
console.log("Get 2:", lru.get(2)); // -1 (not found)
lru.display();
```

🔧 C++ Implementation

```
#include <iostream>
#include <vector>
#include <string>
#include <list>
#include <unordered_map>
#include <algorithm>
using namespace std;

// Hash Table with Separate Chaining
template<typename K, typename V>
class HashTableChaining {
private:
    struct KeyValue {
        K key;
        V value;
        KeyValue(const K& k, const V& v) : key(k), value(v) {}
    };

    vector<list<KeyValue>> buckets;
    size_t bucket_count;
    size_t size;
```

```
    // Hash function for different types
    size_t hash(const K& key) const {
        return std::hash<K>{}(key) % bucket_count;
    }

public:
    HashTableChaining(size_t initial_bucket_count = 10)
        : bucket_count(initial_bucket_count), size(0) {
        buckets.resize(bucket_count);
    }

    // Insert or update key-value pair
    // Average Time: O(1), Worst Time: O(n), Space: O(1)
    void set(const K& key, const V& value) {
        size_t index = hash(key);
        auto& bucket = buckets[index];

        // Check if key already exists
        for (auto& kv : bucket) {
            if (kv.key == key) {
                kv.value = value; // Update existing
                return;
            }
        }

        // Add new key-value pair
        bucket.emplace_back(key, value);
        size++;

        // Resize if load factor exceeds threshold
        if (static_cast<double>(size) / bucket_count > 0.75) {
            resize();
        }
    }

    // Get value by key
    // Average Time: O(1), Worst Time: O(n), Space: O(1)
    bool get(const K& key, V& value) const {
        size_t index = hash(key);
        const auto& bucket = buckets[index];

        for (const auto& kv : bucket) {
            if (kv.key == key) {
                value = kv.value;
                return true;
            }
        }

        return false;
    }

    // Check if key exists
    bool has(const K& key) const {
```

```
        V dummy;
        return get(key, dummy);
    }

    // Delete key-value pair
    // Average Time: O(1), Worst Time: O(n), Space: O(1)
    bool remove(const K& key) {
        size_t index = hash(key);
        auto& bucket = buckets[index];

        for (auto it = bucket.begin(); it != bucket.end(); ++it) {
            if (it->key == key) {
                bucket.erase(it);
                size--;
                return true;
            }
        }

        return false;
    }

    // Get all keys
    vector<K> keys() const {
        vector<K> result;
        for (const auto& bucket : buckets) {
            for (const auto& kv : bucket) {
                result.push_back(kv.key);
            }
        }
        return result;
    }

    // Get load factor
    double getLoadFactor() const {
        return static_cast<double>(size) / bucket_count;
    }

    // Get size
    size_t getSize() const {
        return size;
    }

    // Display hash table structure
    void display() const {
        cout << "Hash Table Structure:" << endl;
        for (size_t i = 0; i < buckets.size(); i++) {
            if (!buckets[i].empty()) {
                cout << "Bucket " << i << ": ";
                for (const auto& kv : buckets[i]) {
                    cout << "(" << kv.key << ", " << kv.value << ") ";
                }
                cout << endl;
            }
        }
```

```
        cout << "Load Factor: " << getLoadFactor() << endl;
    }

private:
    // Resize hash table when load factor is high
    void resize() {
        vector<list<KeyValue>> old_buckets = move(buckets);
        bucket_count *= 2;
        buckets.clear();
        buckets.resize(bucket_count);
        size = 0;

        // Rehash all existing elements
        for (const auto& bucket : old_buckets) {
            for (const auto& kv : bucket) {
                set(kv.key, kv.value);
            }
        }
    }
};

// Hash Table with Linear Probing
template<typename K, typename V>
class HashTableLinearProbing {
private:
    struct Entry {
        K key;
        V value;
        bool deleted;

        Entry() : deleted(true) {}
        Entry(const K& k, const V& v) : key(k), value(v), deleted(false) {}
    };

    vector<Entry> table;
    size_t capacity;
    size_t size;

    size_t hash(const K& key) const {
        return std::hash<K>{}(key) % capacity;
    }

public:
    HashTableLinearProbing(size_t initial_capacity = 10)
        : capacity(initial_capacity), size(0) {
        table.resize(capacity);
    }

    // Insert or update key-value pair
    void set(const K& key, const V& value) {
        if (static_cast<double>(size) / capacity > 0.75) {
            resize();
        }
```

```
        size_t index = hash(key);

        // Linear probing to find empty slot or existing key
        while (!table[index].deleted) {
            if (table[index].key == key) {
                table[index].value = value; // Update existing
                return;
            }
            index = (index + 1) % capacity;
        }

        // Insert new key-value pair
        table[index] = Entry(key, value);
        size++;
    }

    // Get value by key
    bool get(const K& key, V& value) const {
        size_t index = hash(key);
        size_t original_index = index;

        do {
            if (table[index].deleted) {
                return false; // Empty slot found, key doesn't exist
            }
            if (table[index].key == key) {
                value = table[index].value;
                return true;
            }
            index = (index + 1) % capacity;
        } while (index != original_index);

        return false;
    }

    // Check if key exists
    bool has(const K& key) const {
        V dummy;
        return get(key, dummy);
    }

    // Delete key-value pair
    bool remove(const K& key) {
        size_t index = hash(key);
        size_t original_index = index;

        do {
            if (table[index].deleted) {
                return false; // Empty slot found, key doesn't exist
            }
            if (table[index].key == key) {
                table[index].deleted = true;
                size--;
                return true;
```

```
            }
            index = (index + 1) % capacity;
        } while (index != original_index);

        return false;
    }

    // Get load factor
    double getLoadFactor() const {
        return static_cast<double>(size) / capacity;
    }

    // Get size
    size_t getSize() const {
        return size;
    }

private:
    // Resize hash table
    void resize() {
        vector<Entry> old_table = move(table);
        capacity *= 2;
        table.clear();
        table.resize(capacity);
        size = 0;

        // Rehash all existing elements
        for (const auto& entry : old_table) {
            if (!entry.deleted) {
                set(entry.key, entry.value);
            }
        }
    }
};

// Hash Set Implementation
template<typename T>
class HashSet {
private:
    HashTableChaining<T, bool> table;

public:
    void add(const T& element) {
        table.set(element, true);
    }

    bool has(const T& element) const {
        return table.has(element);
    }

    bool remove(const T& element) {
        return table.remove(element);
    }
```

```
    vector<T> values() const {
        return table.keys();
    }

    size_t size() const {
        return table.getSize();
    }
};

// Hash Table Applications
class HashTableApplications {
public:
    // Find two numbers that sum to target
    // Time: O(n), Space: O(n)
    static vector<int> twoSum(const vector<int>& nums, int target) {
        unordered_map<int, int> map;

        for (int i = 0; i < nums.size(); i++) {
            int complement = target - nums[i];
            if (map.find(complement) != map.end()) {
                return {map[complement], i};
            }
            map[nums[i]] = i;
        }

        return {};
    }

    // Group anagrams together
    // Time: O(n * k log k), Space: O(n * k)
    static vector<vector<string>> groupAnagrams(const vector<string>& strs) {
        unordered_map<string, vector<string>> map;

        for (const string& str : strs) {
            string sorted = str;
            sort(sorted.begin(), sorted.end());
            map[sorted].push_back(str);
        }

        vector<vector<string>> result;
        for (const auto& pair : map) {
            result.push_back(pair.second);
        }

        return result;
    }

    // Find first non-repeating character
    // Time: O(n), Space: O(1) - limited character set
    static int firstUniqueChar(const string& s) {
        unordered_map<char, int> charCount;

        // Count character frequencies
        for (char c : s) {
```

```
            charCount[c]++;
        }

        // Find first character with count 1
        for (int i = 0; i < s.length(); i++) {
            if (charCount[s[i]] == 1) {
                return i;
            }
        }

        return -1;
    }
};

// Example Usage
int main() {
    cout << "=== Hash Table with Chaining Demo ===" << endl;
    HashTableChaining<string, int> hashTable;
    hashTable.set("apple", 5);
    hashTable.set("banana", 3);
    hashTable.set("orange", 8);

    int value;
    if (hashTable.get("apple", value)) {
        cout << "Apple: " << value << endl;
    }

    hashTable.display();

    cout << "\n=== Hash Table with Linear Probing Demo ===" << endl;
    HashTableLinearProbing<string, int> hashTableLP;
    hashTableLP.set("red", 1);
    hashTableLP.set("green", 2);
    hashTableLP.set("blue", 3);

    cout << "Load factor: " << hashTableLP.getLoadFactor() << endl;

    cout << "\n=== Hash Set Demo ===" << endl;
    HashSet<string> hashSet;
    hashSet.add("cat");
    hashSet.add("dog");
    hashSet.add("bird");

    cout << "Has cat: " << (hashSet.has("cat") ? "Yes" : "No") << endl;
    cout << "Set size: " << hashSet.size() << endl;

    cout << "\n=== Hash Table Applications ===" << endl;
    vector<int> nums = {2, 7, 11, 15};
    vector<int> result = HashTableApplications::twoSum(nums, 9);
    cout << "Two Sum result: [" << result[0] << ", " << result[1] << "]" << endl;

    cout << "First unique char in 'leetcode': "
         << HashTableApplications::firstUniqueChar("leetcode") << endl;
```

```
    return 0;
}
```

⚡ Performance Analysis

Time Complexity:

Operation	Average Case	Worst Case	Best Case
Search	O(1)	O(n)	O(1)
Insert	O(1)	O(n)	O(1)
Delete	O(1)	O(n)	O(1)

Space Complexity:

- **Hash Table**: O(n) where n is the number of key-value pairs
- **Additional space per operation**: O(1)

Load Factor Impact:

- **Low load factor (< 0.5)**: Fewer collisions, more memory usage
- **High load factor (> 0.75)**: More collisions, less memory usage
- **Optimal range**: 0.5 - 0.75 for good balance

Collision Resolution Comparison:

Aspect	Separate Chaining	Linear Probing
Memory	Higher (pointers)	Lower (no pointers)
Cache Performance	Poor	Better
Deletion	Easy	Complex
Load Factor Tolerance	Higher	Lower

🏵 Practice Problems

Problem 1: Two Sum

Question: Given an array of integers and a target, return indices of two numbers that add up to target.

Solution:

```
function twoSum(nums, target) {
  const map = new Map();
  for (let i = 0; i < nums.length; i++) {
    const complement = target - nums[i];
```

```
      if (map.has(complement)) {
        return [map.get(complement), i];
      }
      map.set(nums[i], i);
    }
    return [];
  }
```

Problem 2: Group Anagrams

Question: Group strings that are anagrams of each other.

Hint: Use sorted string as key in hash map.

Problem 3: Longest Substring Without Repeating Characters

Question: Find the length of the longest substring without repeating characters.

Hint: Use sliding window with hash set to track characters.

Problem 4: Design HashMap

Question: Design a HashMap without using built-in hash table libraries.

Hint: Implement with array of buckets and handle collisions.

🎯 Interview Tips

What Interviewers Look For:

1. **Hash function understanding**: Can you explain how hashing works?
2. **Collision handling**: Knowledge of different resolution techniques
3. **Load factor awareness**: Understanding of performance implications
4. **Real-world applications**: When to use hash tables vs. other structures

Common Interview Patterns:

- **Frequency counting**: Character/element frequency problems
- **Two-pointer with hash**: Finding pairs, triplets with specific sums
- **Caching**: LRU cache, memoization problems
- **Grouping**: Anagrams, similar strings, etc.

Red Flags to Avoid:

- Not considering hash collisions
- Ignoring load factor and resizing
- Using hash tables when order matters
- Not handling edge cases (empty keys, null values)

Pro Tips:

1. **Understand trade-offs**: Hash tables vs. trees vs. arrays
2. **Consider hash quality**: Good distribution reduces collisions
3. **Think about resizing**: When and how to resize
4. **Practice common patterns**: Two sum, anagrams, frequency counting

---

🚀 Key Takeaways

1. **Hash tables provide O(1) average access** - Excellent for lookups
2. **Hash function quality matters** - Good distribution prevents clustering
3. **Handle collisions properly** - Choose appropriate resolution method
4. **Monitor load factor** - Resize when necessary for performance
5. **Perfect for many algorithms** - Frequency counting, caching, grouping

Next Chapter: We'll explore Trees and see how they provide hierarchical data organization with efficient search, insertion, and deletion operations.

Chapter 5: Trees - Hierarchical Data Structures

🎯 What Are Trees?

Trees are hierarchical data structures consisting of nodes connected by edges. Unlike linear structures (arrays, linked lists), trees represent hierarchical relationships with a root node at the top and child nodes branching downward.

Why Trees Matter:

- **Hierarchical Organization**: Natural representation of hierarchical data
- **Efficient Search**: O(log n) operations in balanced trees
- **Flexible Structure**: Can represent many real-world relationships
- **Foundation**: Basis for databases, file systems, and decision trees

Tree Terminology:

- **Root**: Top node with no parent
- **Parent**: Node with child nodes
- **Child**: Node with a parent
- **Leaf**: Node with no children
- **Height**: Longest path from root to leaf
- **Depth**: Distance from root to a specific node
- **Subtree**: Tree formed by a node and its descendants

🔍 Types of Trees

1. Binary Tree

- Each node has at most two children (left and right)
- Foundation for many other tree types

2. Binary Search Tree (BST)

- Binary tree with ordering property
- Left subtree values < parent < right subtree values
- Enables efficient searching

3. Complete Binary Tree

- All levels filled except possibly the last
- Last level filled from left to right

4. Full Binary Tree

- Every node has either 0 or 2 children
- No node has exactly one child

5. Perfect Binary Tree

- All internal nodes have two children
- All leaves are at the same level

💻 JavaScript Implementation

```
// Binary Tree Node
class TreeNode {
  constructor(data) {
    this.data = data;
    this.left = null;
    this.right = null;
  }
}

// Binary Tree Implementation
class BinaryTree {
  constructor() {
    this.root = null;
  }

  // Insert node (level-order insertion for complete tree)
  // Time: O(n), Space: O(n)
  insert(data) {
    const newNode = new TreeNode(data);

    if (!this.root) {
      this.root = newNode;
      return;
    }

    // Use queue for level-order insertion
    const queue = [this.root];
```

```
    while (queue.length > 0) {
      const current = queue.shift();

      if (!current.left) {
        current.left = newNode;
        return;
      } else if (!current.right) {
        current.right = newNode;
        return;
      } else {
        queue.push(current.left);
        queue.push(current.right);
      }
    }
  }

  // Search for a value
  // Time: O(n), Space: O(h) where h is height
  search(data, node = this.root) {
    if (!node) {
      return false;
    }

    if (node.data === data) {
      return true;
    }

    return this.search(data, node.left) || this.search(data, node.right);
  }

  // Get height of tree
  // Time: O(n), Space: O(h)
  getHeight(node = this.root) {
    if (!node) {
      return -1;
    }

    return 1 + Math.max(this.getHeight(node.left), this.getHeight(node.right));
  }

  // Count total nodes
  // Time: O(n), Space: O(h)
  countNodes(node = this.root) {
    if (!node) {
      return 0;
    }

    return 1 + this.countNodes(node.left) + this.countNodes(node.right);
  }

  // Check if tree is balanced
  // Time: O(n), Space: O(h)
  isBalanced(node = this.root) {
```

```
    if (!node) {
      return true;
    }

    const leftHeight = this.getHeight(node.left);
    const rightHeight = this.getHeight(node.right);

    return (
      Math.abs(leftHeight - rightHeight) <= 1 &&
      this.isBalanced(node.left) &&
      this.isBalanced(node.right)
    );
  }

  // Find maximum value
  // Time: O(n), Space: O(h)
  findMax(node = this.root) {
    if (!node) {
      return null;
    }

    let max = node.data;
    const leftMax = this.findMax(node.left);
    const rightMax = this.findMax(node.right);

    if (leftMax !== null && leftMax > max) {
      max = leftMax;
    }
    if (rightMax !== null && rightMax > max) {
      max = rightMax;
    }

    return max;
  }

  // Find minimum value
  // Time: O(n), Space: O(h)
  findMin(node = this.root) {
    if (!node) {
      return null;
    }

    let min = node.data;
    const leftMin = this.findMin(node.left);
    const rightMin = this.findMin(node.right);

    if (leftMin !== null && leftMin < min) {
      min = leftMin;
    }
    if (rightMin !== null && rightMin < min) {
      min = rightMin;
    }

    return min;
```

```
    }
  }

  // Binary Search Tree Implementation
  class BinarySearchTree {
    constructor() {
      this.root = null;
    }

    // Insert node maintaining BST property
    // Time: O(log n) average, O(n) worst, Space: O(h)
    insert(data) {
      this.root = this.insertNode(this.root, data);
    }

    insertNode(node, data) {
      if (!node) {
        return new TreeNode(data);
      }

      if (data < node.data) {
        node.left = this.insertNode(node.left, data);
      } else if (data > node.data) {
        node.right = this.insertNode(node.right, data);
      }
      // Ignore duplicates

      return node;
    }

    // Search for a value
    // Time: O(log n) average, O(n) worst, Space: O(h)
    search(data, node = this.root) {
      if (!node) {
        return false;
      }

      if (data === node.data) {
        return true;
      } else if (data < node.data) {
        return this.search(data, node.left);
      } else {
        return this.search(data, node.right);
      }
    }

    // Find minimum value (leftmost node)
    // Time: O(log n) average, O(n) worst, Space: O(h)
    findMin(node = this.root) {
      if (!node) {
        return null;
      }

      while (node.left) {
```

```
      node = node.left;
    }

    return node.data;
  }

  // Find maximum value (rightmost node)
  // Time: O(log n) average, O(n) worst, Space: O(h)
  findMax(node = this.root) {
    if (!node) {
      return null;
    }

    while (node.right) {
      node = node.right;
    }

    return node.data;
  }

  // Delete node
  // Time: O(log n) average, O(n) worst, Space: O(h)
  delete(data) {
    this.root = this.deleteNode(this.root, data);
  }

  deleteNode(node, data) {
    if (!node) {
      return null;
    }

    if (data < node.data) {
      node.left = this.deleteNode(node.left, data);
    } else if (data > node.data) {
      node.right = this.deleteNode(node.right, data);
    } else {
      // Node to be deleted found

      // Case 1: No children (leaf node)
      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
      // Find inorder successor (smallest in right subtree)
      const successor = this.findMinNode(node.right);
```

```
      node.data = successor.data;
      node.right = this.deleteNode(node.right, successor.data);
    }

    return node;
  }

  // Helper method to find minimum node
  findMinNode(node) {
    while (node.left) {
      node = node.left;
    }
    return node;
  }

  // Validate if tree is a valid BST
  // Time: O(n), Space: O(h)
  isValidBST(node = this.root, min = null, max = null) {
    if (!node) {
      return true;
    }

    if (
      (min !== null && node.data <= min) ||
      (max !== null && node.data >= max)
    ) {
      return false;
    }

    return (
      this.isValidBST(node.left, min, node.data) &&
      this.isValidBST(node.right, node.data, max)
    );
  }

  // Find kth smallest element
  // Time: O(k), Space: O(h)
  kthSmallest(k) {
    const result = { count: 0, value: null };
    this.inorderKth(this.root, k, result);
    return result.value;
  }

  inorderKth(node, k, result) {
    if (!node || result.count >= k) {
      return;
    }

    this.inorderKth(node.left, k, result);

    result.count++;
    if (result.count === k) {
      result.value = node.data;
      return;
```

```
      }

      this.inorderKth(node.right, k, result);
    }

    // Find lowest common ancestor
    // Time: O(log n) average, O(n) worst, Space: O(h)
    findLCA(node1Data, node2Data, node = this.root) {
      if (!node) {
        return null;
      }

      // If both nodes are smaller, LCA is in left subtree
      if (node1Data < node.data && node2Data < node.data) {
        return this.findLCA(node1Data, node2Data, node.left);
      }

      // If both nodes are larger, LCA is in right subtree
      if (node1Data > node.data && node2Data > node.data) {
        return this.findLCA(node1Data, node2Data, node.right);
      }

      // If one is smaller and one is larger, current node is LCA
      return node.data;
    }
  }

  // Tree Traversal Methods
  class TreeTraversal {
    // Inorder Traversal (Left, Root, Right)
    // Time: O(n), Space: O(h)
    static inorder(node, result = []) {
      if (node) {
        TreeTraversal.inorder(node.left, result);
        result.push(node.data);
        TreeTraversal.inorder(node.right, result);
      }
      return result;
    }

    // Preorder Traversal (Root, Left, Right)
    // Time: O(n), Space: O(h)
    static preorder(node, result = []) {
      if (node) {
        result.push(node.data);
        TreeTraversal.preorder(node.left, result);
        TreeTraversal.preorder(node.right, result);
      }
      return result;
    }

    // Postorder Traversal (Left, Right, Root)
    // Time: O(n), Space: O(h)
    static postorder(node, result = []) {
```

```
    if (node) {
      TreeTraversal.postorder(node.left, result);
      TreeTraversal.postorder(node.right, result);
      result.push(node.data);
    }
    return result;
  }

  // Level Order Traversal (Breadth-First)
  // Time: O(n), Space: O(w) where w is maximum width
  static levelOrder(root) {
    if (!root) {
      return [];
    }

    const result = [];
    const queue = [root];

    while (queue.length > 0) {
      const levelSize = queue.length;
      const currentLevel = [];

      for (let i = 0; i < levelSize; i++) {
        const node = queue.shift();
        currentLevel.push(node.data);

        if (node.left) queue.push(node.left);
        if (node.right) queue.push(node.right);
      }

      result.push(currentLevel);
    }

    return result;
  }

  // Iterative Inorder Traversal
  // Time: O(n), Space: O(h)
  static inorderIterative(root) {
    const result = [];
    const stack = [];
    let current = root;

    while (current || stack.length > 0) {
      // Go to leftmost node
      while (current) {
        stack.push(current);
        current = current.left;
      }

      // Process current node
      current = stack.pop();
      result.push(current.data);
```

```
        // Move to right subtree
        current = current.right;
      }

      return result;
    }
  }

  // Tree Applications
  class TreeApplications {
    // Build tree from inorder and preorder traversals
    // Time: O(n), Space: O(n)
    static buildTreeFromTraversals(preorder, inorder) {
      if (preorder.length === 0 || inorder.length === 0) {
        return null;
      }

      const rootVal = preorder[0];
      const root = new TreeNode(rootVal);
      const rootIndex = inorder.indexOf(rootVal);

      const leftInorder = inorder.slice(0, rootIndex);
      const rightInorder = inorder.slice(rootIndex + 1);

      const leftPreorder = preorder.slice(1, 1 + leftInorder.length);
      const rightPreorder = preorder.slice(1 + leftInorder.length);

      root.left = TreeApplications.buildTreeFromTraversals(
        leftPreorder,
        leftInorder
      );
      root.right = TreeApplications.buildTreeFromTraversals(
        rightPreorder,
        rightInorder
      );

      return root;
    }

    // Convert sorted array to balanced BST
    // Time: O(n), Space: O(log n)
    static sortedArrayToBST(nums) {
      if (nums.length === 0) {
        return null;
      }

      const mid = Math.floor(nums.length / 2);
      const root = new TreeNode(nums[mid]);

      root.left = TreeApplications.sortedArrayToBST(nums.slice(0, mid));
      root.right = TreeApplications.sortedArrayToBST(nums.slice(mid + 1));

      return root;
    }
```

```
  // Find diameter of binary tree
  // Time: O(n), Space: O(h)
  static diameterOfBinaryTree(root) {
    let diameter = 0;

    function height(node) {
      if (!node) return 0;

      const leftHeight = height(node.left);
      const rightHeight = height(node.right);

      // Update diameter if path through current node is longer
      diameter = Math.max(diameter, leftHeight + rightHeight);

      return 1 + Math.max(leftHeight, rightHeight);
    }

    height(root);
    return diameter;
  }

  // Check if two trees are identical
  // Time: O(n), Space: O(h)
  static isSameTree(p, q) {
    if (!p && !q) return true;
    if (!p || !q) return false;

    return (
      p.data === q.data &&
      TreeApplications.isSameTree(p.left, q.left) &&
      TreeApplications.isSameTree(p.right, q.right)
    );
  }
}

// Example Usage
console.log("=== Binary Tree Demo ===");
const bt = new BinaryTree();
bt.insert(1);
bt.insert(2);
bt.insert(3);
bt.insert(4);
bt.insert(5);

console.log("Tree height:", bt.getHeight()); // 2
console.log("Node count:", bt.countNodes()); // 5
console.log("Is balanced:", bt.isBalanced()); // true
console.log("Search 4:", bt.search(4)); // true

console.log("\n=== Binary Search Tree Demo ===");
const bst = new BinarySearchTree();
[50, 30, 70, 20, 40, 60, 80].forEach((val) => bst.insert(val));
```

```
console.log("Search 40:", bst.search(40)); // true
console.log("Min value:", bst.findMin()); // 20
console.log("Max value:", bst.findMax()); // 80
console.log("Is valid BST:", bst.isValidBST()); // true
console.log("3rd smallest:", bst.kthSmallest(3)); // 40
console.log("LCA of 20 and 40:", bst.findLCA(20, 40)); // 30

console.log("\n=== Tree Traversals ===");
console.log("Inorder:", TreeTraversal.inorder(bst.root)); // [20, 30, 40, 50, 60,
70, 80]
console.log("Preorder:", TreeTraversal.preorder(bst.root)); // [50, 30, 20, 40,
70, 60, 80]
console.log("Postorder:", TreeTraversal.postorder(bst.root)); // [20, 40, 30, 60,
80, 70, 50]
console.log("Level order:", TreeTraversal.levelOrder(bst.root));

console.log("\n=== Tree Applications ===");
const sortedArray = [1, 2, 3, 4, 5, 6, 7];
const balancedBST = TreeApplications.sortedArrayToBST(sortedArray);
console.log(
  "Balanced BST from sorted array:",
  TreeTraversal.inorder(balancedBST)
);
console.log(
  "Diameter of tree:",
  TreeApplications.diameterOfBinaryTree(bst.root)
);
```

🔧 C++ Implementation

```
#include <iostream>
#include <vector>
#include <queue>
#include <stack>
#include <algorithm>
#include <climits>
using namespace std;

// Binary Tree Node
struct TreeNode {
    int data;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int val) : data(val), left(nullptr), right(nullptr) {}
};

// Binary Tree Class
class BinaryTree {
public:
```

```
    TreeNode* root;

    BinaryTree() : root(nullptr) {}

    // Destructor to prevent memory leaks
    ~BinaryTree() {
        destroyTree(root);
    }

    // Insert node (level-order insertion)
    // Time: O(n), Space: O(n)
    void insert(int data) {
        TreeNode* newNode = new TreeNode(data);

        if (!root) {
            root = newNode;
            return;
        }

        queue<TreeNode*> q;
        q.push(root);

        while (!q.empty()) {
            TreeNode* current = q.front();
            q.pop();

            if (!current->left) {
                current->left = newNode;
                return;
            } else if (!current->right) {
                current->right = newNode;
                return;
            } else {
                q.push(current->left);
                q.push(current->right);
            }
        }
    }

    // Search for a value
    // Time: O(n), Space: O(h)
    bool search(int data, TreeNode* node = nullptr) {
        if (node == nullptr) node = root;
        if (!node) return false;

        if (node->data == data) return true;

        return search(data, node->left) || search(data, node->right);
    }

    // Get height of tree
    // Time: O(n), Space: O(h)
    int getHeight(TreeNode* node = nullptr) {
        if (node == nullptr) node = root;
```

```
        if (!node) return -1;

        return 1 + max(getHeight(node->left), getHeight(node->right));
    }

    // Count total nodes
    // Time: O(n), Space: O(h)
    int countNodes(TreeNode* node = nullptr) {
        if (node == nullptr) node = root;
        if (!node) return 0;

        return 1 + countNodes(node->left) + countNodes(node->right);
    }

    // Check if tree is balanced
    // Time: O(n), Space: O(h)
    bool isBalanced(TreeNode* node = nullptr) {
        if (node == nullptr) node = root;
        if (!node) return true;

        int leftHeight = getHeight(node->left);
        int rightHeight = getHeight(node->right);

        return abs(leftHeight - rightHeight) <= 1 &&
               isBalanced(node->left) &&
               isBalanced(node->right);
    }

private:
    // Helper function to destroy tree
    void destroyTree(TreeNode* node) {
        if (node) {
            destroyTree(node->left);
            destroyTree(node->right);
            delete node;
        }
    }
};

// Binary Search Tree Class
class BinarySearchTree {
public:
    TreeNode* root;

    BinarySearchTree() : root(nullptr) {}

    // Destructor
    ~BinarySearchTree() {
        destroyTree(root);
    }

    // Insert node maintaining BST property
    // Time: O(log n) average, O(n) worst, Space: O(h)
    void insert(int data) {
```

```
        root = insertNode(root, data);
    }

    // Search for a value
    // Time: O(log n) average, O(n) worst, Space: O(h)
    bool search(int data, TreeNode* node = nullptr) {
        if (node == nullptr) node = root;
        if (!node) return false;

        if (data == node->data) {
            return true;
        } else if (data < node->data) {
            return search(data, node->left);
        } else {
            return search(data, node->right);
        }
    }

    // Find minimum value
    // Time: O(log n) average, O(n) worst, Space: O(1)
    int findMin() {
        if (!root) throw runtime_error("Tree is empty");

        TreeNode* current = root;
        while (current->left) {
            current = current->left;
        }
        return current->data;
    }

    // Find maximum value
    // Time: O(log n) average, O(n) worst, Space: O(1)
    int findMax() {
        if (!root) throw runtime_error("Tree is empty");

        TreeNode* current = root;
        while (current->right) {
            current = current->right;
        }
        return current->data;
    }

    // Delete node
    // Time: O(log n) average, O(n) worst, Space: O(h)
    void deleteNode(int data) {
        root = deleteNodeHelper(root, data);
    }

    // Validate if tree is a valid BST
    // Time: O(n), Space: O(h)
    bool isValidBST() {
        return isValidBSTHelper(root, INT_MIN, INT_MAX);
    }
```

```
    // Find kth smallest element
    // Time: O(k), Space: O(h)
    int kthSmallest(int k) {
        int count = 0;
        int result = -1;
        inorderKth(root, k, count, result);
        return result;
    }

    // Find lowest common ancestor
    // Time: O(log n) average, O(n) worst, Space: O(h)
    int findLCA(int node1, int node2) {
        TreeNode* lca = findLCAHelper(root, node1, node2);
        return lca ? lca->data : -1;
    }

private:
    TreeNode* insertNode(TreeNode* node, int data) {
        if (!node) {
            return new TreeNode(data);
        }

        if (data < node->data) {
            node->left = insertNode(node->left, data);
        } else if (data > node->data) {
            node->right = insertNode(node->right, data);
        }
        // Ignore duplicates

        return node;
    }

    TreeNode* deleteNodeHelper(TreeNode* node, int data) {
        if (!node) return nullptr;

        if (data < node->data) {
            node->left = deleteNodeHelper(node->left, data);
        } else if (data > node->data) {
            node->right = deleteNodeHelper(node->right, data);
        } else {
            // Node to be deleted found

            // Case 1: No children
            if (!node->left && !node->right) {
                delete node;
                return nullptr;
            }

            // Case 2: One child
            if (!node->left) {
                TreeNode* temp = node->right;
                delete node;
                return temp;
            }
```

```
            if (!node->right) {
                TreeNode* temp = node->left;
                delete node;
                return temp;
            }

            // Case 3: Two children
            TreeNode* successor = findMinNode(node->right);
            node->data = successor->data;
            node->right = deleteNodeHelper(node->right, successor->data);
        }

        return node;
    }

    TreeNode* findMinNode(TreeNode* node) {
        while (node->left) {
            node = node->left;
        }
        return node;
    }

    bool isValidBSTHelper(TreeNode* node, int minVal, int maxVal) {
        if (!node) return true;

        if (node->data <= minVal || node->data >= maxVal) {
            return false;
        }

        return isValidBSTHelper(node->left, minVal, node->data) &&
               isValidBSTHelper(node->right, node->data, maxVal);
    }

    void inorderKth(TreeNode* node, int k, int& count, int& result) {
        if (!node || count >= k) return;

        inorderKth(node->left, k, count, result);

        count++;
        if (count == k) {
            result = node->data;
            return;
        }

        inorderKth(node->right, k, count, result);
    }

    TreeNode* findLCAHelper(TreeNode* node, int node1, int node2) {
        if (!node) return nullptr;

        if (node1 < node->data && node2 < node->data) {
            return findLCAHelper(node->left, node1, node2);
        }
```

```
        if (node1 > node->data && node2 > node->data) {
            return findLCAHelper(node->right, node1, node2);
        }

        return node;
    }

    void destroyTree(TreeNode* node) {
        if (node) {
            destroyTree(node->left);
            destroyTree(node->right);
            delete node;
        }
    }
};

// Tree Traversal Class
class TreeTraversal {
public:
    // Inorder Traversal (Left, Root, Right)
    // Time: O(n), Space: O(h)
    static void inorder(TreeNode* node, vector<int>& result) {
        if (node) {
            inorder(node->left, result);
            result.push_back(node->data);
            inorder(node->right, result);
        }
    }

    // Preorder Traversal (Root, Left, Right)
    // Time: O(n), Space: O(h)
    static void preorder(TreeNode* node, vector<int>& result) {
        if (node) {
            result.push_back(node->data);
            preorder(node->left, result);
            preorder(node->right, result);
        }
    }

    // Postorder Traversal (Left, Right, Root)
    // Time: O(n), Space: O(h)
    static void postorder(TreeNode* node, vector<int>& result) {
        if (node) {
            postorder(node->left, result);
            postorder(node->right, result);
            result.push_back(node->data);
        }
    }

    // Level Order Traversal
    // Time: O(n), Space: O(w)
    static vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> result;
        if (!root) return result;
```

```
        queue<TreeNode*> q;
        q.push(root);

        while (!q.empty()) {
            int levelSize = q.size();
            vector<int> currentLevel;

            for (int i = 0; i < levelSize; i++) {
                TreeNode* node = q.front();
                q.pop();
                currentLevel.push_back(node->data);

                if (node->left) q.push(node->left);
                if (node->right) q.push(node->right);
            }

            result.push_back(currentLevel);
        }

        return result;
    }

    // Iterative Inorder Traversal
    // Time: O(n), Space: O(h)
    static vector<int> inorderIterative(TreeNode* root) {
        vector<int> result;
        stack<TreeNode*> stk;
        TreeNode* current = root;

        while (current || !stk.empty()) {
            while (current) {
                stk.push(current);
                current = current->left;
            }

            current = stk.top();
            stk.pop();
            result.push_back(current->data);

            current = current->right;
        }

        return result;
    }
};

// Example Usage
int main() {
    cout << "=== Binary Tree Demo ===" << endl;
    BinaryTree bt;
    bt.insert(1);
    bt.insert(2);
    bt.insert(3);
```

```
    bt.insert(4);
    bt.insert(5);

    cout << "Tree height: " << bt.getHeight() << endl;
    cout << "Node count: " << bt.countNodes() << endl;
    cout << "Is balanced: " << (bt.isBalanced() ? "Yes" : "No") << endl;
    cout << "Search 4: " << (bt.search(4) ? "Found" : "Not found") << endl;

    cout << "\n=== Binary Search Tree Demo ===" << endl;
    BinarySearchTree bst;
    vector<int> values = {50, 30, 70, 20, 40, 60, 80};
    for (int val : values) {
        bst.insert(val);
    }

    cout << "Search 40: " << (bst.search(40) ? "Found" : "Not found") << endl;
    cout << "Min value: " << bst.findMin() << endl;
    cout << "Max value: " << bst.findMax() << endl;
    cout << "Is valid BST: " << (bst.isValidBST() ? "Yes" : "No") << endl;
    cout << "3rd smallest: " << bst.kthSmallest(3) << endl;
    cout << "LCA of 20 and 40: " << bst.findLCA(20, 40) << endl;

    cout << "\n=== Tree Traversals ===" << endl;
    vector<int> inorderResult, preorderResult, postorderResult;

    TreeTraversal::inorder(bst.root, inorderResult);
    TreeTraversal::preorder(bst.root, preorderResult);
    TreeTraversal::postorder(bst.root, postorderResult);

    cout << "Inorder: ";
    for (int val : inorderResult) cout << val << " ";
    cout << endl;

    cout << "Preorder: ";
    for (int val : preorderResult) cout << val << " ";
    cout << endl;

    cout << "Postorder: ";
    for (int val : postorderResult) cout << val << " ";
    cout << endl;

    vector<vector<int>> levelOrderResult = TreeTraversal::levelOrder(bst.root);
    cout << "Level order: ";
    for (const auto& level : levelOrderResult) {
        cout << "[";
        for (int i = 0; i < level.size(); i++) {
            cout << level[i];
            if (i < level.size() - 1) cout << ", ";
        }
        cout << "] ";
    }
    cout << endl;
```

```
    return 0;
}
```

⚡ Performance Analysis

Time Complexity Comparison:

Operation	Binary Tree	Binary Search Tree (Balanced)	Binary Search Tree (Worst)
Search	O(n)	O(log n)	O(n)
Insert	O(n)	O(log n)	O(n)
Delete	O(n)	O(log n)	O(n)
Traversal	O(n)	O(n)	O(n)

Space Complexity:

- **Tree storage**: O(n) where n is number of nodes
- **Recursive operations**: O(h) where h is height
- **Balanced tree height**: O(log n)
- **Worst case height**: O(n) - skewed tree

Common Pitfalls:

1. **Memory leaks**: Not properly deallocating nodes in C++
2. **Stack overflow**: Deep recursion in skewed trees
3. **BST property violation**: Incorrect insertion/deletion
4. **Null pointer access**: Not checking for null nodes

🧩 Practice Problems

Problem 1: Validate Binary Search Tree

Question: Determine if a binary tree is a valid BST.

Solution:

```
function isValidBST(root, min = null, max = null) {
  if (!root) return true;

  if ((min !== null && root.val <= min) || (max !== null && root.val >= max)) {
    return false;
  }

  return (
    isValidBST(root.left, min, root.val) &&
    isValidBST(root.right, root.val, max)
```

```
    );
}
```

Problem 2: Lowest Common Ancestor

Question: Find the lowest common ancestor of two nodes in a BST.

Hint: Use BST property to navigate efficiently.

Problem 3: Binary Tree Level Order Traversal

Question: Return the level order traversal of a binary tree.

Hint: Use a queue for breadth-first traversal.

Problem 4: Convert Sorted Array to BST

Question: Convert a sorted array to a height-balanced BST.

Hint: Use divide and conquer with middle element as root.

🎯 Interview Tips

What Interviewers Look For:

1. **Tree traversal mastery**: Can you implement all traversal methods?
2. **BST property understanding**: Do you maintain ordering during operations?
3. **Recursion skills**: Comfortable with recursive tree algorithms?
4. **Edge case handling**: Empty trees, single nodes, null pointers

Common Interview Patterns:

- **Tree traversal**: Inorder, preorder, postorder, level-order
- **Tree construction**: From traversals, from sorted arrays
- **Tree validation**: BST validation, balanced tree checks
- **Path problems**: Root to leaf paths, path sums

Red Flags to Avoid:

- Not handling null/empty trees
- Confusing tree traversal orders
- Violating BST property during modifications
- Not considering tree balance and performance

Pro Tips:

1. **Draw the tree**: Visualize the problem
2. **Think recursively**: Most tree problems have recursive solutions
3. **Consider base cases**: Empty nodes, leaf nodes

4. **Practice traversals**: Master all four traversal methods

---

🚀 Key Takeaways

1. **Trees provide hierarchical organization** - Natural for many data relationships
2. **BSTs enable efficient searching** - O(log n) in balanced trees
3. **Traversals are fundamental** - Master inorder, preorder, postorder, level-order
4. **Balance matters for performance** - Avoid skewed trees
5. **Recursion is your friend** - Most tree algorithms are naturally recursive

Next Chapter: We'll explore Recursion in detail and see how it's the foundation for many tree and other algorithms.

Chapter 6: Recursion Basics - The Art of Self-Reference

🎯 What is Recursion?

Recursion is a programming technique where a function calls itself to solve a smaller version of the same problem. It's like looking into two mirrors facing each other - you see infinite reflections, each smaller than the last.

Why Recursion Matters:

- **Natural problem-solving**: Many problems have recursive structure
- **Elegant solutions**: Often simpler and more readable than iterative approaches
- **Tree and graph algorithms**: Essential for hierarchical data structures
- **Divide and conquer**: Foundation for many efficient algorithms

Recursion Components:

1. **Base Case**: Condition that stops the recursion
2. **Recursive Case**: Function calls itself with modified parameters
3. **Progress**: Each call moves closer to the base case

🔍 How Recursion Works

The Call Stack:

When a function calls itself, each call is added to the **call stack**:

```
factorial(4)
├── factorial(3)
│   ├── factorial(2)
│   │   ├── factorial(1)
│   │   │   └── factorial(0) → returns 1
│   │   └── returns 1 * 1 = 1
│   └── returns 2 * 1 = 2
```

```
└── returns 3 * 2 = 6
returns 4 * 6 = 24
```

Recursion vs Iteration:

Aspect	Recursion	Iteration
Readability	Often cleaner	Can be verbose
Memory	Uses call stack	Uses variables
Performance	Function call overhead	Generally faster
Stack overflow	Possible with deep recursion	Not applicable

💻 JavaScript Implementation

```javascript
// Basic Recursion Examples

// 1. Factorial
// Time: O(n), Space: O(n)
function factorial(n) {
  // Base case: factorial of 0 or 1 is 1
  if (n <= 1) {
    return 1;
  }

  // Recursive case: n! = n * (n-1)!
  return n * factorial(n - 1);
}

// 2. Fibonacci Sequence
// Time: O(2^n), Space: O(n) - naive approach
function fibonacci(n) {
  // Base cases
  if (n <= 1) {
    return n;
  }

  // Recursive case: F(n) = F(n-1) + F(n-2)
  return fibonacci(n - 1) + fibonacci(n - 2);
}

// Optimized Fibonacci with Memoization
// Time: O(n), Space: O(n)
function fibonacciMemo(n, memo = {}) {
  // Check if already computed
  if (n in memo) {
    return memo[n];
  }
```

```
  // Base cases
  if (n <= 1) {
    return n;
  }

  // Store result in memo and return
  memo[n] = fibonacciMemo(n - 1, memo) + fibonacciMemo(n - 2, memo);
  return memo[n];
}

// 3. Power Function
// Time: O(log n), Space: O(log n)
function power(base, exponent) {
  // Base case
  if (exponent === 0) {
    return 1;
  }

  // Handle negative exponents
  if (exponent < 0) {
    return 1 / power(base, -exponent);
  }

  // Optimize using divide and conquer
  if (exponent % 2 === 0) {
    const half = power(base, exponent / 2);
    return half * half;
  } else {
    return base * power(base, exponent - 1);
  }
}

// 4. Sum of Array
// Time: O(n), Space: O(n)
function arraySum(arr, index = 0) {
  // Base case: reached end of array
  if (index >= arr.length) {
    return 0;
  }

  // Recursive case: current element + sum of rest
  return arr[index] + arraySum(arr, index + 1);
}

// 5. Reverse String
// Time: O(n), Space: O(n)
function reverseString(str) {
  // Base case: empty or single character
  if (str.length <= 1) {
    return str;
  }

  // Recursive case: last char + reverse of rest
  return str[str.length - 1] + reverseString(str.slice(0, -1));
```

```
}

// 6. Check if String is Palindrome
// Time: O(n), Space: O(n)
function isPalindrome(str, start = 0, end = str.length - 1) {
  // Base case: single character or empty
  if (start >= end) {
    return true;
  }

  // Check if characters match
  if (str[start] !== str[end]) {
    return false;
  }

  // Recursive case: check inner substring
  return isPalindrome(str, start + 1, end - 1);
}

// 7. Binary Search (Recursive)
// Time: O(log n), Space: O(log n)
function binarySearch(arr, target, left = 0, right = arr.length - 1) {
  // Base case: element not found
  if (left > right) {
    return -1;
  }

  const mid = Math.floor((left + right) / 2);

  // Base case: element found
  if (arr[mid] === target) {
    return mid;
  }

  // Recursive cases
  if (target < arr[mid]) {
    return binarySearch(arr, target, left, mid - 1);
  } else {
    return binarySearch(arr, target, mid + 1, right);
  }
}

// 8. Generate All Subsets
// Time: O(2^n), Space: O(2^n)
function generateSubsets(nums) {
  const result = [];

  function backtrack(index, currentSubset) {
    // Base case: processed all elements
    if (index === nums.length) {
      result.push([...currentSubset]);
      return;
    }
```

```
    // Include current element
    currentSubset.push(nums[index]);
    backtrack(index + 1, currentSubset);

    // Exclude current element (backtrack)
    currentSubset.pop();
    backtrack(index + 1, currentSubset);
  }

  backtrack(0, []);
  return result;
}

// 9. Tower of Hanoi
// Time: O(2^n), Space: O(n)
function towerOfHanoi(n, source, destination, auxiliary) {
  const moves = [];

  function solve(disks, src, dest, aux) {
    // Base case: only one disk
    if (disks === 1) {
      moves.push(`Move disk 1 from ${src} to ${dest}`);
      return;
    }

    // Move n-1 disks from source to auxiliary
    solve(disks - 1, src, aux, dest);

    // Move the largest disk from source to destination
    moves.push(`Move disk ${disks} from ${src} to ${dest}`);

    // Move n-1 disks from auxiliary to destination
    solve(disks - 1, aux, dest, src);
  }

  solve(n, source, destination, auxiliary);
  return moves;
}

// 10. Count Paths in Grid
// Time: O(2^(m+n)), Space: O(m+n) - naive
// Time: O(m*n), Space: O(m*n) - with memoization
function countPaths(m, n, memo = {}) {
  const key = `${m},${n}`;

  // Check memo
  if (key in memo) {
    return memo[key];
  }

  // Base cases
  if (m === 1 || n === 1) {
    return 1;
  }
```

```
  // Recursive case: paths from top + paths from left
  memo[key] = countPaths(m - 1, n, memo) + countPaths(m, n - 1, memo);
  return memo[key];
}

// 11. Merge Sort (Recursive)
// Time: O(n log n), Space: O(n)
function mergeSort(arr) {
  // Base case: array with 0 or 1 element
  if (arr.length <= 1) {
    return arr;
  }

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

  // Conquer (merge)
  return merge(left, right);
}

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

  while (i < left.length && j < right.length) {
    if (left[i] <= right[j]) {
      result.push(left[i]);
      i++;
    } else {
      result.push(right[j]);
      j++;
    }
  }

  // Add remaining elements
  return result.concat(left.slice(i)).concat(right.slice(j));
}

// 12. Quick Sort (Recursive)
// Time: O(n log n) average, O(n^2) worst, Space: O(log n)
function quickSort(arr, low = 0, high = arr.length - 1) {
  if (low < high) {
    // Partition and get pivot index
    const pivotIndex = partition(arr, low, high);

    // Recursively sort elements before and after partition
    quickSort(arr, low, pivotIndex - 1);
    quickSort(arr, pivotIndex + 1, high);
  }
```

```
    return arr;
  }

  function partition(arr, low, high) {
    const pivot = arr[high];
    let i = low - 1;

    for (let j = low; j < high; j++) {
      if (arr[j] <= pivot) {
        i++;
        [arr[i], arr[j]] = [arr[j], arr[i]];
      }
    }

    [arr[i + 1], arr[high]] = [arr[high], arr[i + 1]];
    return i + 1;
  }

  // Recursion Helper Functions
  class RecursionHelpers {
    // Convert recursion to iteration using explicit stack
    static factorialIterative(n) {
      if (n <= 1) return 1;

      let result = 1;
      for (let i = 2; i <= n; i++) {
        result *= i;
      }
      return result;
    }

    // Tail recursion optimization (JavaScript doesn't optimize, but concept)
    static factorialTailRecursive(n, accumulator = 1) {
      if (n <= 1) {
        return accumulator;
      }
      return RecursionHelpers.factorialTailRecursive(n - 1, n * accumulator);
    }

    // Mutual recursion example
    static isEven(n) {
      if (n === 0) return true;
      return RecursionHelpers.isOdd(n - 1);
    }

    static isOdd(n) {
      if (n === 0) return false;
      return RecursionHelpers.isEven(n - 1);
    }

    // Recursion with multiple base cases
    static tribonacci(n) {
      if (n === 0) return 0;
      if (n === 1 || n === 2) return 1;
```

```
    return (
      RecursionHelpers.tribonacci(n - 1) +
      RecursionHelpers.tribonacci(n - 2) +
      RecursionHelpers.tribonacci(n - 3)
    );
  }
}

// Example Usage and Testing
console.log("=== Basic Recursion Examples ===");
console.log("Factorial of 5:", factorial(5)); // 120
console.log("Fibonacci of 10:", fibonacci(10)); // 55
console.log("Fibonacci of 10 (memoized):", fibonacciMemo(10)); // 55
console.log("2^10:", power(2, 10)); // 1024
console.log("Sum of [1,2,3,4,5]:", arraySum([1, 2, 3, 4, 5])); // 15
console.log('Reverse "hello":', reverseString("hello")); // "olleh"
console.log('Is "racecar" palindrome?', isPalindrome("racecar")); // true

console.log("\n=== Advanced Recursion Examples ===");
const sortedArray = [1, 3, 5, 7, 9, 11, 13];
console.log("Binary search for 7:", binarySearch(sortedArray, 7)); // 3

const subsets = generateSubsets([1, 2, 3]);
console.log("Subsets of [1,2,3]:", subsets);

const hanoi = towerOfHanoi(3, "A", "C", "B");
console.log("Tower of Hanoi (3 disks):");
hanoi.forEach((move) => console.log(move));

console.log("Paths in 3x3 grid:", countPaths(3, 3)); // 6

const unsortedArray = [64, 34, 25, 12, 22, 11, 90];
console.log("Original array:", unsortedArray);
console.log("Merge sorted:", mergeSort([...unsortedArray]));
console.log("Quick sorted:", quickSort([...unsortedArray]));

console.log("\n=== Helper Functions ===");
console.log("Factorial iterative:", RecursionHelpers.factorialIterative(5));
console.log(
  "Factorial tail recursive:",
  RecursionHelpers.factorialTailRecursive(5)
);
console.log("Is 4 even?", RecursionHelpers.isEven(4)); // true
console.log("Is 5 odd?", RecursionHelpers.isOdd(5)); // true
console.log("Tribonacci of 7:", RecursionHelpers.tribonacci(7)); // 24
```

🔗 C++ Implementation

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <unordered_map>
#include <algorithm>
using namespace std;

// Basic Recursion Examples

// 1. Factorial
// Time: O(n), Space: O(n)
long long factorial(int n) {
    // Base case
    if (n <= 1) {
        return 1;
    }

    // Recursive case
    return n * factorial(n - 1);
}

// 2. Fibonacci (naive)
// Time: O(2^n), Space: O(n)
long long fibonacci(int n) {
    // Base cases
    if (n <= 1) {
        return n;
    }

    // Recursive case
    return fibonacci(n - 1) + fibonacci(n - 2);
}

// Fibonacci with memoization
// Time: O(n), Space: O(n)
unordered_map<int, long long> fiboMemo;

long long fibonacciMemo(int n) {
    // Check if already computed
    if (fiboMemo.find(n) != fiboMemo.end()) {
        return fiboMemo[n];
    }

    // Base cases
    if (n <= 1) {
        return n;
    }

    // Compute and store
    fiboMemo[n] = fibonacciMemo(n - 1) + fibonacciMemo(n - 2);
    return fiboMemo[n];
}
```

```cpp
// 3. Power function
// Time: O(log n), Space: O(log n)
double power(double base, int exponent) {
    // Base case
    if (exponent == 0) {
        return 1.0;
    }

    // Handle negative exponents
    if (exponent < 0) {
        return 1.0 / power(base, -exponent);
    }

    // Optimize using divide and conquer
    if (exponent % 2 == 0) {
        double half = power(base, exponent / 2);
        return half * half;
    } else {
        return base * power(base, exponent - 1);
    }
}

// 4. Sum of array
// Time: O(n), Space: O(n)
int arraySum(const vector<int>& arr, int index = 0) {
    // Base case
    if (index >= arr.size()) {
        return 0;
    }

    // Recursive case
    return arr[index] + arraySum(arr, index + 1);
}

// 5. Reverse string
// Time: O(n), Space: O(n)
string reverseString(const string& str) {
    // Base case
    if (str.length() <= 1) {
        return str;
    }

    // Recursive case
    return str.back() + reverseString(str.substr(0, str.length() - 1));
}

// 6. Check palindrome
// Time: O(n), Space: O(n)
bool isPalindrome(const string& str, int start = 0, int end = -1) {
    if (end == -1) end = str.length() - 1;

    // Base case
    if (start >= end) {
        return true;
```

```
    }

    // Check characters
    if (str[start] != str[end]) {
        return false;
    }

    // Recursive case
    return isPalindrome(str, start + 1, end - 1);
}

// 7. Binary search
// Time: O(log n), Space: O(log n)
int binarySearch(const vector<int>& arr, int target, int left = 0, int right = -1)
{
    if (right == -1) right = arr.size() - 1;

    // Base case: not found
    if (left > right) {
        return -1;
    }

    int mid = left + (right - left) / 2;

    // Base case: found
    if (arr[mid] == target) {
        return mid;
    }

    // Recursive cases
    if (target < arr[mid]) {
        return binarySearch(arr, target, left, mid - 1);
    } else {
        return binarySearch(arr, target, mid + 1, right);
    }
}

// 8. Generate subsets
// Time: O(2^n), Space: O(2^n)
void generateSubsets(const vector<int>& nums, int index, vector<int>& current,
vector<vector<int>>& result) {
    // Base case: processed all elements
    if (index == nums.size()) {
        result.push_back(current);
        return;
    }

    // Include current element
    current.push_back(nums[index]);
    generateSubsets(nums, index + 1, current, result);

    // Exclude current element (backtrack)
    current.pop_back();
    generateSubsets(nums, index + 1, current, result);
```

```
}

vector<vector<int>> getAllSubsets(const vector<int>& nums) {
    vector<vector<int>> result;
    vector<int> current;
    generateSubsets(nums, 0, current, result);
    return result;
}

// 9. Tower of Hanoi
// Time: O(2^n), Space: O(n)
void towerOfHanoi(int n, char source, char destination, char auxiliary,
vector<string>& moves) {
    // Base case
    if (n == 1) {
        moves.push_back("Move disk 1 from " + string(1, source) + " to " +
string(1, destination));
        return;
    }

    // Move n-1 disks from source to auxiliary
    towerOfHanoi(n - 1, source, auxiliary, destination, moves);

    // Move the largest disk
    moves.push_back("Move disk " + to_string(n) + " from " + string(1, source) + "
to " + string(1, destination));

    // Move n-1 disks from auxiliary to destination
    towerOfHanoi(n - 1, auxiliary, destination, source, moves);
}

// 10. Count paths in grid
// Time: O(m*n) with memoization, Space: O(m*n)
unordered_map<string, int> pathMemo;

int countPaths(int m, int n) {
    string key = to_string(m) + "," + to_string(n);

    // Check memo
    if (pathMemo.find(key) != pathMemo.end()) {
        return pathMemo[key];
    }

    // Base cases
    if (m == 1 || n == 1) {
        return 1;
    }

    // Recursive case
    pathMemo[key] = countPaths(m - 1, n) + countPaths(m, n - 1);
    return pathMemo[key];
}

// 11. Merge Sort
```

```cpp
// Time: O(n log n), Space: O(n)
vector<int> merge(const vector<int>& left, const vector<int>& right) {
    vector<int> result;
    int i = 0, j = 0;

    while (i < left.size() && j < right.size()) {
        if (left[i] <= right[j]) {
            result.push_back(left[i]);
            i++;
        } else {
            result.push_back(right[j]);
            j++;
        }
    }

    // Add remaining elements
    while (i < left.size()) {
        result.push_back(left[i]);
        i++;
    }
    while (j < right.size()) {
        result.push_back(right[j]);
        j++;
    }

    return result;
}

vector<int> mergeSort(const vector<int>& arr) {
    // Base case
    if (arr.size() <= 1) {
        return arr;
    }

    // Divide
    int mid = arr.size() / 2;
    vector<int> left(arr.begin(), arr.begin() + mid);
    vector<int> right(arr.begin() + mid, arr.end());

    // Conquer
    vector<int> sortedLeft = mergeSort(left);
    vector<int> sortedRight = mergeSort(right);

    // Merge
    return merge(sortedLeft, sortedRight);
}

// 12. Quick Sort
// Time: O(n log n) average, O(n^2) worst, Space: O(log n)
int partition(vector<int>& arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j < high; j++) {
```

```
            if (arr[j] <= pivot) {
                i++;
                swap(arr[i], arr[j]);
            }
        }

        swap(arr[i + 1], arr[high]);
        return i + 1;
    }

    void quickSort(vector<int>& arr, int low, int high) {
        if (low < high) {
            int pivotIndex = partition(arr, low, high);

            quickSort(arr, low, pivotIndex - 1);
            quickSort(arr, pivotIndex + 1, high);
        }
    }

    // Helper Functions
    class RecursionHelpers {
    public:
        // Tail recursion (C++ may optimize)
        static long long factorialTailRecursive(int n, long long accumulator = 1) {
            if (n <= 1) {
                return accumulator;
            }
            return factorialTailRecursive(n - 1, n * accumulator);
        }

        // Mutual recursion
        static bool isEven(int n) {
            if (n == 0) return true;
            return isOdd(n - 1);
        }

        static bool isOdd(int n) {
            if (n == 0) return false;
            return isEven(n - 1);
        }

        // Greatest Common Divisor (Euclidean algorithm)
        static int gcd(int a, int b) {
            if (b == 0) return a;
            return gcd(b, a % b);
        }

        // Convert decimal to binary
        static string decimalToBinary(int n) {
            if (n == 0) return "0";
            if (n == 1) return "1";

            return decimalToBinary(n / 2) + to_string(n % 2);
        }
```

```
};

// Example Usage
int main() {
    cout << "=== Basic Recursion Examples ===" << endl;
    cout << "Factorial of 5: " << factorial(5) << endl;
    cout << "Fibonacci of 10: " << fibonacci(10) << endl;
    cout << "Fibonacci of 10 (memoized): " << fibonacciMemo(10) << endl;
    cout << "2^10: " << power(2, 10) << endl;

    vector<int> arr = {1, 2, 3, 4, 5};
    cout << "Sum of array: " << arraySum(arr) << endl;

    string str = "hello";
    cout << "Reverse of \"" << str << "\": " << reverseString(str) << endl;

    string palindrome = "racecar";
    cout << "Is \"" << palindrome << "\" palindrome? " <<
(isPalindrome(palindrome) ? "Yes" : "No") << endl;

    cout << "\n=== Advanced Recursion Examples ===" << endl;
    vector<int> sortedArr = {1, 3, 5, 7, 9, 11, 13};
    cout << "Binary search for 7: " << binarySearch(sortedArr, 7) << endl;

    vector<int> nums = {1, 2, 3};
    vector<vector<int>> subsets = getAllSubsets(nums);
    cout << "Subsets of [1,2,3]: ";
    for (const auto& subset : subsets) {
        cout << "[";
        for (int i = 0; i < subset.size(); i++) {
            cout << subset[i];
            if (i < subset.size() - 1) cout << ",";
        }
        cout << "] ";
    }
    cout << endl;

    vector<string> hanoiMoves;
    towerOfHanoi(3, 'A', 'C', 'B', hanoiMoves);
    cout << "Tower of Hanoi (3 disks):" << endl;
    for (const string& move : hanoiMoves) {
        cout << move << endl;
    }

    cout << "Paths in 3x3 grid: " << countPaths(3, 3) << endl;

    vector<int> unsorted = {64, 34, 25, 12, 22, 11, 90};
    cout << "Original array: ";
    for (int x : unsorted) cout << x << " ";
    cout << endl;

    vector<int> mergeSorted = mergeSort(unsorted);
    cout << "Merge sorted: ";
    for (int x : mergeSorted) cout << x << " ";
```

```
    cout << endl;

    vector<int> quickSorted = unsorted;
    quickSort(quickSorted, 0, quickSorted.size() - 1);
    cout << "Quick sorted: ";
    for (int x : quickSorted) cout << x << " ";
    cout << endl;

    cout << "\n=== Helper Functions ===" << endl;
    cout << "Factorial tail recursive: " <<
RecursionHelpers::factorialTailRecursive(5) << endl;
    cout << "Is 4 even? " << (RecursionHelpers::isEven(4) ? "Yes" : "No") << endl;
    cout << "Is 5 odd? " << (RecursionHelpers::isOdd(5) ? "Yes" : "No") << endl;
    cout << "GCD of 48 and 18: " << RecursionHelpers::gcd(48, 18) << endl;
    cout << "Binary of 10: " << RecursionHelpers::decimalToBinary(10) << endl;

    return 0;
}
```

⚡ Performance Analysis

Time Complexity Patterns:

Pattern	Example	Time Complexity	Space Complexity
Linear Recursion	Factorial, Sum	O(n)	O(n)
Binary Recursion	Fibonacci (naive)	O(2^n)	O(n)
Logarithmic	Binary Search, Power	O(log n)	O(log n)
Divide & Conquer	Merge Sort	O(n log n)	O(n)

Common Pitfalls:

1. **Stack Overflow**: Deep recursion can exhaust call stack

```
// Bad: Will cause stack overflow for large n
function badFactorial(n) {
  if (n === 0) return 1;
  return n * badFactorial(n - 1); // No tail call optimization
}
```

2. **Exponential Time**: Naive recursive solutions

```
// Bad: O(2^n) time complexity
function slowFibonacci(n) {
  if (n <= 1) return n;
  return slowFibonacci(n - 1) + slowFibonacci(n - 2); // Recalculates same
```

```
values
}
```

3. **Missing Base Case**: Infinite recursion

```
// Bad: No base case
function infiniteRecursion(n) {
  return infiniteRecursion(n - 1); // Will never stop
}
```

4. **Incorrect Progress**: Not moving toward base case

```
// Bad: n never decreases
function noProgress(n) {
  if (n === 0) return 0;
  return noProgress(n); // Same value passed
}
```

Optimization Techniques:

1. **Memoization**: Store computed results
2. **Tail Recursion**: Last operation is recursive call
3. **Iterative Conversion**: Convert to loops when possible
4. **Dynamic Programming**: Bottom-up approach

Practice Problems

Problem 1: Sum of Digits

Question: Write a recursive function to find the sum of digits of a number.

Example: `sumDigits(1234)` should return `10`

Solution:

```
function sumDigits(n) {
  if (n === 0) return 0;
  return (n % 10) + sumDigits(Math.floor(n / 10));
}
```

Problem 2: Count Occurrences

Question: Count occurrences of a character in a string recursively.

Hint: Process one character at a time.

Problem 3: Flatten Nested Array

Question: Flatten a nested array using recursion.

Example: `[1, [2, 3], [4, [5, 6]]]` → `[1, 2, 3, 4, 5, 6]`

Hint: Check if element is array, recurse if true.

Problem 4: Generate Permutations

Question: Generate all permutations of a string.

Hint: Fix first character, permute rest, then swap.

🎯 Interview Tips

What Interviewers Look For:

1. **Base case identification**: Can you identify when to stop?
2. **Recursive case logic**: Do you break down the problem correctly?
3. **Complexity analysis**: Can you analyze time/space complexity?
4. **Optimization awareness**: Do you know when recursion isn't optimal?

Common Interview Patterns:

- **Tree traversals**: Inorder, preorder, postorder
- **Divide and conquer**: Merge sort, quick sort, binary search
- **Backtracking**: Permutations, combinations, N-Queens
- **Dynamic programming**: Fibonacci, coin change, longest subsequence

Red Flags to Avoid:

- Forgetting base cases
- Not making progress toward base case
- Ignoring stack overflow for large inputs
- Not considering iterative alternatives

Pro Tips:

1. **Start with base case**: Always identify stopping condition first
2. **Trust the recursion**: Assume recursive calls work correctly
3. **Draw the call stack**: Visualize for complex problems
4. **Consider memoization**: For overlapping subproblems
5. **Think iteratively too**: Sometimes loops are better

🚀 Key Takeaways

1. **Recursion is powerful** - Elegant solutions for many problems
2. **Base cases are crucial** - Always define stopping conditions
3. **Watch the complexity** - Naive recursion can be exponential
4. **Memoization helps** - Cache results for overlapping subproblems
5. **Not always optimal** - Sometimes iteration is better
6. **Practice makes perfect** - Start simple, build complexity

Next Chapter: We'll explore Sorting Algorithms and see how recursion powers divide-and-conquer sorting methods like merge sort and quick sort.

Chapter 7: Sorting Algorithms - Organizing Data Efficiently

🎯 What is Sorting?

Sorting is the process of arranging data in a particular order (ascending or descending). It's one of the most fundamental operations in computer science and forms the basis for many other algorithms.

Why Sorting Matters:

- **Search optimization**: Sorted data enables binary search (O(log n))
- **Data organization**: Makes data easier to understand and process
- **Algorithm foundation**: Many algorithms require sorted input
- **Database operations**: Crucial for joins, indexing, and queries
- **User experience**: Organized data is more user-friendly

Sorting Categories:

1. **Comparison-based**: Compare elements to determine order
2. **Non-comparison**: Use element properties (counting, radix)
3. **Stable**: Preserve relative order of equal elements
4. **In-place**: Sort with O(1) extra space
5. **Adaptive**: Perform better on partially sorted data

🔍 Sorting Algorithm Overview

Algorithm	Time (Best)	Time (Average)	Time (Worst)	Space	Stable	In-place
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	✅	✅
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	❌	✅
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	✅	✅
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	✅	❌
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$	O(log n)	❌	✅
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)	❌	✅

💻 JavaScript Implementation

```
// Sorting Algorithms Implementation

// 1. Bubble Sort
// Time: O(n²), Space: O(1)
// Stable: Yes, In-place: Yes
function bubbleSort(arr) {
  const n = arr.length;
  const result = [...arr]; // Create copy to avoid mutation

  // Outer loop for number of passes
  for (let i = 0; i < n - 1; i++) {
    let swapped = false; // Optimization: track if any swaps occurred

    // Inner loop for comparisons in current pass
    // Last i elements are already sorted
    for (let j = 0; j < n - i - 1; j++) {
      // Compare adjacent elements
      if (result[j] > result[j + 1]) {
        // Swap if they're in wrong order
        [result[j], result[j + 1]] = [result[j + 1], result[j]];
        swapped = true;
      }
    }

    // If no swaps occurred, array is sorted
    if (!swapped) {
      break;
    }
  }

  return result;
}

// 2. Selection Sort
// Time: O(n²), Space: O(1)
// Stable: No, In-place: Yes
function selectionSort(arr) {
  const n = arr.length;
  const result = [...arr];

  // Move boundary of unsorted subarray
  for (let i = 0; i < n - 1; i++) {
    // Find minimum element in unsorted array
    let minIndex = i;

    for (let j = i + 1; j < n; j++) {
      if (result[j] < result[minIndex]) {
        minIndex = j;
      }
```

```
    }

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

  return result;
}

// 3. Insertion Sort
// Time: O(n²), Space: O(1)
// Stable: Yes, In-place: Yes
function insertionSort(arr) {
  const n = arr.length;
  const result = [...arr];

  // Start from second element (first is considered sorted)
  for (let i = 1; i < n; i++) {
    const key = result[i]; // Current element to be positioned
    let j = i - 1;

    // Move elements greater than key one position ahead
    while (j >= 0 && result[j] > key) {
      result[j + 1] = result[j];
      j--;
    }

    // Place key at its correct position
    result[j + 1] = key;
  }

  return result;
}

// 4. Merge Sort
// Time: O(n log n), Space: O(n)
// Stable: Yes, In-place: No
function mergeSort(arr) {
  // Base case: arrays with 0 or 1 element are already sorted
  if (arr.length <= 1) {
    return arr;
  }

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

  // Conquer: merge sorted halves
  return merge(left, right);
}
```

```
// Helper function to merge two sorted arrays
function merge(left, right) {
  const result = [];
  let i = 0,
    j = 0;

  // Compare elements from both arrays and merge in sorted order
  while (i < left.length && j < right.length) {
    if (left[i] <= right[j]) {
      result.push(left[i]);
      i++;
    } else {
      result.push(right[j]);
      j++;
    }
  }

  // Add remaining elements (if any)
  while (i < left.length) {
    result.push(left[i]);
    i++;
  }

  while (j < right.length) {
    result.push(right[j]);
    j++;
  }

  return result;
}

// 5. Quick Sort
// Time: O(n log n) average, O(n²) worst, Space: O(log n)
// Stable: No, In-place: Yes
function quickSort(arr, low = 0, high = arr.length - 1) {
  const result = [...arr]; // Create copy for non-destructive sorting
  quickSortHelper(result, low, high);
  return result;
}

function quickSortHelper(arr, low, high) {
  if (low < high) {
    // Partition array and get pivot index
    const pivotIndex = partition(arr, low, high);

    // Recursively sort elements before and after partition
    quickSortHelper(arr, low, pivotIndex - 1);
    quickSortHelper(arr, pivotIndex + 1, high);
  }
}

// Partition function for Quick Sort
function partition(arr, low, high) {
  // Choose rightmost element as pivot
```

```
  const pivot = arr[high];
  let i = low - 1; // Index of smaller element

  for (let j = low; j < high; j++) {
    // If current element is smaller than or equal to pivot
    if (arr[j] <= pivot) {
      i++;
      [arr[i], arr[j]] = [arr[j], arr[i]];
    }
  }

  // Place pivot in correct position
  [arr[i + 1], arr[high]] = [arr[high], arr[i + 1]];
  return i + 1;
}

// 6. Heap Sort
// Time: O(n log n), Space: O(1)
// Stable: No, In-place: Yes
function heapSort(arr) {
  const result = [...arr];
  const n = result.length;

  // Build max heap
  for (let i = Math.floor(n / 2) - 1; i >= 0; i--) {
    heapify(result, n, i);
  }

  // Extract elements from heap one by one
  for (let i = n - 1; i > 0; i--) {
    // Move current root to end
    [result[0], result[i]] = [result[i], result[0]];

    // Call heapify on reduced heap
    heapify(result, i, 0);
  }

  return result;
}

// Heapify a subtree rooted at index i
function heapify(arr, n, i) {
  let largest = i; // Initialize largest as root
  const left = 2 * i + 1;
  const right = 2 * i + 2;

  // If left child is larger than root
  if (left < n && arr[left] > arr[largest]) {
    largest = left;
  }

  // If right child is larger than largest so far
  if (right < n && arr[right] > arr[largest]) {
    largest = right;
```

```
    }

    // If largest is not root
    if (largest !== i) {
      [arr[i], arr[largest]] = [arr[largest], arr[i]];

      // Recursively heapify the affected sub-tree
      heapify(arr, n, largest);
    }
  }

  // Advanced Sorting Algorithms

  // 7. Counting Sort (for integers in limited range)
  // Time: O(n + k), Space: O(k) where k is range
  // Stable: Yes, In-place: No
  function countingSort(arr, maxValue = null) {
    if (arr.length === 0) return arr;

    // Find maximum value if not provided
    if (maxValue === null) {
      maxValue = Math.max(...arr);
    }

    // Create count array
    const count = new Array(maxValue + 1).fill(0);

    // Count occurrences of each element
    for (let num of arr) {
      count[num]++;
    }

    // Build result array
    const result = [];
    for (let i = 0; i <= maxValue; i++) {
      while (count[i] > 0) {
        result.push(i);
        count[i]--;
      }
    }

    return result;
  }

  // 8. Radix Sort (for non-negative integers)
  // Time: O(d * (n + k)), Space: O(n + k)
  // where d is number of digits, k is range of digits (0-9)
  function radixSort(arr) {
    if (arr.length === 0) return arr;

    // Find maximum number to know number of digits
    const max = Math.max(...arr);

    // Do counting sort for every digit
```

```
  let result = [...arr];
  for (let exp = 1; Math.floor(max / exp) > 0; exp *= 10) {
    result = countingSortByDigit(result, exp);
  }

  return result;
}

// Counting sort for radix sort
function countingSortByDigit(arr, exp) {
  const n = arr.length;
  const output = new Array(n);
  const count = new Array(10).fill(0);

  // Count occurrences of each digit
  for (let i = 0; i < n; i++) {
    const digit = Math.floor(arr[i] / exp) % 10;
    count[digit]++;
  }

  // Change count[i] to actual position
  for (let i = 1; i < 10; i++) {
    count[i] += count[i - 1];
  }

  // Build output array
  for (let i = n - 1; i >= 0; i--) {
    const digit = Math.floor(arr[i] / exp) % 10;
    output[count[digit] - 1] = arr[i];
    count[digit]--;
  }

  return output;
}

// Sorting Utilities
class SortingUtils {
  // Check if array is sorted
  static isSorted(arr, ascending = true) {
    for (let i = 1; i < arr.length; i++) {
      if (ascending && arr[i] < arr[i - 1]) {
        return false;
      }
      if (!ascending && arr[i] > arr[i - 1]) {
        return false;
      }
    }
    return true;
  }

  // Generate random array for testing
  static generateRandomArray(size, min = 0, max = 100) {
    return Array.from(
      { length: size },
```

```
      () => Math.floor(Math.random() * (max - min + 1)) + min
    );
  }

  // Measure sorting performance
  static measurePerformance(sortFunction, arr, name) {
    const start = performance.now();
    const sorted = sortFunction([...arr]);
    const end = performance.now();

    console.log(`${name}: ${(end - start).toFixed(2)}ms`);
    console.log(`Sorted correctly: ${SortingUtils.isSorted(sorted)}`);
    return sorted;
  }

  // Sort with custom comparator
  static customSort(arr, compareFn) {
    return [...arr].sort(compareFn);
  }

  // Stable sort check
  static isStableSort(sortFunction) {
    // Test with objects having same keys
    const testData = [
      { key: 3, id: "a" },
      { key: 1, id: "b" },
      { key: 3, id: "c" },
      { key: 2, id: "d" },
      { key: 3, id: "e" },
    ];

    const sorted = sortFunction(testData, (a, b) => a.key - b.key);

    // Check if relative order of equal elements is preserved
    const threes = sorted.filter((item) => item.key === 3);
    return threes[0].id === "a" && threes[1].id === "c" && threes[2].id === "e";
  }

  // Find kth smallest element (QuickSelect)
  static quickSelect(arr, k) {
    if (k < 1 || k > arr.length) {
      throw new Error("k is out of bounds");
    }

    const result = [...arr];
    return quickSelectHelper(result, 0, result.length - 1, k - 1);
  }
}

// QuickSelect helper function
function quickSelectHelper(arr, low, high, k) {
  if (low === high) {
    return arr[low];
  }
```

```
  const pivotIndex = partition(arr, low, high);

  if (k === pivotIndex) {
    return arr[k];
  } else if (k < pivotIndex) {
    return quickSelectHelper(arr, low, pivotIndex - 1, k);
  } else {
    return quickSelectHelper(arr, pivotIndex + 1, high, k);
  }
}

// Hybrid Sorting Algorithm (Introsort-like)
function hybridSort(arr) {
  if (arr.length <= 10) {
    // Use insertion sort for small arrays
    return insertionSort(arr);
  } else if (arr.length <= 1000) {
    // Use quick sort for medium arrays
    return quickSort(arr);
  } else {
    // Use merge sort for large arrays
    return mergeSort(arr);
  }
}

// Example Usage and Performance Testing
console.log("=== Sorting Algorithms Demo ===");

// Test data
const testArray = [64, 34, 25, 12, 22, 11, 90, 5, 77, 30];
console.log("Original array:", testArray);

// Test all sorting algorithms
console.log("\n=== Basic Sorting Algorithms ===");
console.log("Bubble Sort:", bubbleSort(testArray));
console.log("Selection Sort:", selectionSort(testArray));
console.log("Insertion Sort:", insertionSort(testArray));

console.log("\n=== Advanced Sorting Algorithms ===");
console.log("Merge Sort:", mergeSort(testArray));
console.log("Quick Sort:", quickSort(testArray));
console.log("Heap Sort:", heapSort(testArray));

console.log("\n=== Specialized Sorting Algorithms ===");
const integerArray = [4, 2, 2, 8, 3, 3, 1];
console.log("Original integers:", integerArray);
console.log("Counting Sort:", countingSort(integerArray));
console.log("Radix Sort:", radixSort(integerArray));

// Performance comparison
console.log("\n=== Performance Comparison ===");
const largeArray = SortingUtils.generateRandomArray(1000);
```

```
SortingUtils.measurePerformance(bubbleSort, largeArray, "Bubble Sort");
SortingUtils.measurePerformance(selectionSort, largeArray, "Selection Sort");
SortingUtils.measurePerformance(insertionSort, largeArray, "Insertion Sort");
SortingUtils.measurePerformance(mergeSort, largeArray, "Merge Sort");
SortingUtils.measurePerformance(quickSort, largeArray, "Quick Sort");
SortingUtils.measurePerformance(heapSort, largeArray, "Heap Sort");

// Utility demonstrations
console.log("\n=== Utility Functions ===");
console.log("Is [1,2,3,4,5] sorted?", SortingUtils.isSorted([1, 2, 3, 4, 5]));
console.log(
  "3rd smallest in [3,1,4,1,5,9,2,6]:",
  SortingUtils.quickSelect([3, 1, 4, 1, 5, 9, 2, 6], 3)
);

// Custom sorting
const people = [
  { name: "Alice", age: 30 },
  { name: "Bob", age: 25 },
  { name: "Charlie", age: 35 },
];
const sortedByAge = SortingUtils.customSort(people, (a, b) => a.age - b.age);
console.log("People sorted by age:", sortedByAge);

console.log("\n=== Hybrid Sort Demo ===");
console.log("Hybrid Sort result:", hybridSort(testArray));
```

🔧 C++ Implementation

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <chrono>
#include <random>
#include <climits>
using namespace std;
using namespace std::chrono;

// Sorting Algorithms Implementation

// 1. Bubble Sort
// Time: O(n²), Space: O(1)
void bubbleSort(vector<int>& arr) {
    int n = arr.size();

    for (int i = 0; i < n - 1; i++) {
        bool swapped = false;

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

```
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }

        // If no swapping occurred, array is sorted
        if (!swapped) {
            break;
        }
    }
}

// 2. Selection Sort
// Time: O(n²), Space: O(1)
void selectionSort(vector<int>& arr) {
    int n = arr.size();

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

        // Find minimum element in unsorted portion
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }

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

// 3. Insertion Sort
// Time: O(n²), Space: O(1)
void insertionSort(vector<int>& arr) {
    int n = arr.size();

    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;

        // Move elements greater than key one position ahead
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }

        arr[j + 1] = key;
    }
}

// 4. Merge Sort
```

```cpp
// Time: O(n log n), Space: O(n)
void merge(vector<int>& arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Create temporary arrays
    vector<int> leftArr(n1), rightArr(n2);

    // Copy data to temporary arrays
    for (int i = 0; i < n1; i++) {
        leftArr[i] = arr[left + i];
    }
    for (int j = 0; j < n2; j++) {
        rightArr[j] = arr[mid + 1 + j];
    }

    // Merge temporary arrays back
    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {
        if (leftArr[i] <= rightArr[j]) {
            arr[k] = leftArr[i];
            i++;
        } else {
            arr[k] = rightArr[j];
            j++;
        }
        k++;
    }

    // Copy remaining elements
    while (i < n1) {
        arr[k] = leftArr[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = rightArr[j];
        j++;
        k++;
    }
}

void mergeSort(vector<int>& arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
```

```cpp
// 5. Quick Sort
// Time: O(n log n) average, O(n²) worst, Space: O(log n)
int partition(vector<int>& arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

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

    swap(arr[i + 1], arr[high]);
    return i + 1;
}

void quickSort(vector<int>& arr, int low, int high) {
    if (low < high) {
        int pivotIndex = partition(arr, low, high);

        quickSort(arr, low, pivotIndex - 1);
        quickSort(arr, pivotIndex + 1, high);
    }
}

// 6. Heap Sort
// Time: O(n log n), Space: O(1)
void heapify(vector<int>& arr, int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] > arr[largest]) {
        largest = left;
    }

    if (right < n && arr[right] > arr[largest]) {
        largest = right;
    }

    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

void heapSort(vector<int>& arr) {
    int n = arr.size();

    // Build max heap
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }
```

```
    // Extract elements from heap
    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}

// 7. Counting Sort
// Time: O(n + k), Space: O(k)
vector<int> countingSort(const vector<int>& arr, int maxVal) {
    vector<int> count(maxVal + 1, 0);
    vector<int> result;

    // Count occurrences
    for (int num : arr) {
        count[num]++;
    }

    // Build result
    for (int i = 0; i <= maxVal; i++) {
        while (count[i] > 0) {
            result.push_back(i);
            count[i]--;
        }
    }

    return result;
}

// 8. Radix Sort
// Time: O(d * (n + k)), Space: O(n + k)
vector<int> countingSortForRadix(vector<int> arr, int exp) {
    int n = arr.size();
    vector<int> output(n);
    vector<int> count(10, 0);

    // Count occurrences of each digit
    for (int i = 0; i < n; i++) {
        count[(arr[i] / exp) % 10]++;
    }

    // Change count[i] to actual position
    for (int i = 1; i < 10; i++) {
        count[i] += count[i - 1];
    }

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

    return output;
```

```
}

vector<int> radixSort(vector<int> arr) {
    int maxVal = *max_element(arr.begin(), arr.end());

    for (int exp = 1; maxVal / exp > 0; exp *= 10) {
        arr = countingSortForRadix(arr, exp);
    }

    return arr;
}

// Utility Functions
class SortingUtils {
public:
    // Check if array is sorted
    static bool isSorted(const vector<int>& arr) {
        for (int i = 1; i < arr.size(); i++) {
            if (arr[i] < arr[i - 1]) {
                return false;
            }
        }
        return true;
    }

    // Generate random array
    static vector<int> generateRandomArray(int size, int minVal = 0, int maxVal =
100) {
        vector<int> arr(size);
        random_device rd;
        mt19937 gen(rd());
        uniform_int_distribution<> dis(minVal, maxVal);

        for (int i = 0; i < size; i++) {
            arr[i] = dis(gen);
        }

        return arr;
    }

    // Measure performance
    static void measurePerformance(void (*sortFunc)(vector<int>&), vector<int>
arr, const string& name) {
        auto start = high_resolution_clock::now();
        sortFunc(arr);
        auto end = high_resolution_clock::now();

        auto duration = duration_cast<microseconds>(end - start);
        cout << name << ": " << duration.count() << " microseconds" << endl;
        cout << "Sorted correctly: " << (isSorted(arr) ? "Yes" : "No") << endl;
    }

    // QuickSelect for kth smallest element
    static int quickSelect(vector<int> arr, int k) {
```

```
        if (k < 1 || k > arr.size()) {
            throw invalid_argument("k is out of bounds");
        }

        return quickSelectHelper(arr, 0, arr.size() - 1, k - 1);
    }

private:
    static int quickSelectHelper(vector<int>& arr, int low, int high, int k) {
        if (low == high) {
            return arr[low];
        }

        int pivotIndex = partition(arr, low, high);

        if (k == pivotIndex) {
            return arr[k];
        } else if (k < pivotIndex) {
            return quickSelectHelper(arr, low, pivotIndex - 1, k);
        } else {
            return quickSelectHelper(arr, pivotIndex + 1, high, k);
        }
    }
};

// Print array utility
void printArray(const vector<int>& arr, const string& label = "") {
    if (!label.empty()) {
        cout << label << ": ";
    }
    for (int x : arr) {
        cout << x << " ";
    }
    cout << endl;
}

// Example Usage
int main() {
    cout << "=== Sorting Algorithms Demo ===" << endl;

    // Test data
    vector<int> testArray = {64, 34, 25, 12, 22, 11, 90, 5, 77, 30};
    printArray(testArray, "Original array");

    cout << "\n=== Basic Sorting Algorithms ===" << endl;

    vector<int> bubbleArr = testArray;
    bubbleSort(bubbleArr);
    printArray(bubbleArr, "Bubble Sort");

    vector<int> selectionArr = testArray;
    selectionSort(selectionArr);
    printArray(selectionArr, "Selection Sort");
```

```
    vector<int> insertionArr = testArray;
    insertionSort(insertionArr);
    printArray(insertionArr, "Insertion Sort");

    cout << "\n=== Advanced Sorting Algorithms ===" << endl;

    vector<int> mergeArr = testArray;
    mergeSort(mergeArr, 0, mergeArr.size() - 1);
    printArray(mergeArr, "Merge Sort");

    vector<int> quickArr = testArray;
    quickSort(quickArr, 0, quickArr.size() - 1);
    printArray(quickArr, "Quick Sort");

    vector<int> heapArr = testArray;
    heapSort(heapArr);
    printArray(heapArr, "Heap Sort");

    cout << "\n=== Specialized Sorting Algorithms ===" << endl;

    vector<int> integerArray = {4, 2, 2, 8, 3, 3, 1};
    printArray(integerArray, "Original integers");

    vector<int> countingSorted = countingSort(integerArray, 8);
    printArray(countingSorted, "Counting Sort");

    vector<int> radixSorted = radixSort(integerArray);
    printArray(radixSorted, "Radix Sort");

    // Performance comparison
    cout << "\n=== Performance Comparison (1000 elements) ===" << endl;
    vector<int> largeArray = SortingUtils::generateRandomArray(1000);

    SortingUtils::measurePerformance(bubbleSort, largeArray, "Bubble Sort");
    SortingUtils::measurePerformance(selectionSort, largeArray, "Selection Sort");
    SortingUtils::measurePerformance(insertionSort, largeArray, "Insertion Sort");

    // For merge and quick sort, we need wrapper functions
    auto mergeSortWrapper = [](vector<int>& arr) {
        mergeSort(arr, 0, arr.size() - 1);
    };

    auto quickSortWrapper = [](vector<int>& arr) {
        quickSort(arr, 0, arr.size() - 1);
    };

    SortingUtils::measurePerformance(mergeSortWrapper, largeArray, "Merge Sort");
    SortingUtils::measurePerformance(quickSortWrapper, largeArray, "Quick Sort");
    SortingUtils::measurePerformance(heapSort, largeArray, "Heap Sort");

    // Utility demonstrations
    cout << "\n=== Utility Functions ===" << endl;
    vector<int> sortedTest = {1, 2, 3, 4, 5};
    cout << "Is [1,2,3,4,5] sorted? " << (SortingUtils::isSorted(sortedTest) ?
```

```
"Yes" : "No") << endl;

    vector<int> selectTest = {3, 1, 4, 1, 5, 9, 2, 6};
    cout << "3rd smallest in [3,1,4,1,5,9,2,6]: " <<
SortingUtils::quickSelect(selectTest, 3) << endl;

    return 0;
}
```

⚡ Performance Analysis

When to Use Each Algorithm:

1. **Bubble Sort**:
   - ✅ Educational purposes, very small datasets
   - ❌ Never for production code
2. **Selection Sort**:
   - ✅ When memory writes are expensive
   - ❌ Generally poor performance
3. **Insertion Sort**:
   - ✅ Small arrays (< 50 elements)
   - ✅ Nearly sorted data
   - ✅ Online algorithm (can sort as data arrives)
4. **Merge Sort**:
   - ✅ Stable sorting required
   - ✅ Guaranteed O(n log n) performance
   - ✅ External sorting (large datasets)
   - ❌ Extra memory required
5. **Quick Sort**:
   - ✅ Average case performance
   - ✅ In-place sorting
   - ❌ Worst case $O(n^2)$
   - ❌ Not stable
6. **Heap Sort**:
   - ✅ Guaranteed O(n log n)
   - ✅ In-place sorting
   - ❌ Not stable
   - ❌ Poor cache performance

Common Pitfalls:

1. **Choosing wrong algorithm**: Using bubble sort for large data
2. **Ignoring stability**: When relative order matters
3. **Memory constraints**: Not considering space complexity
4. **Worst-case scenarios**: Quick sort on already sorted data
5. **Integer overflow**: In partition calculations

Practice Problems

Problem 1: Sort Colors (Dutch Flag)

Question: Sort an array containing only 0s, 1s, and 2s.

Example: `[2,0,2,1,1,0]` → `[0,0,1,1,2,2]`

Solution:

```
function sortColors(nums) {
  let low = 0,
    mid = 0,
    high = nums.length - 1;

  while (mid <= high) {
    if (nums[mid] === 0) {
      [nums[low], nums[mid]] = [nums[mid], nums[low]];
      low++;
      mid++;
    } else if (nums[mid] === 1) {
      mid++;
    } else {
      [nums[mid], nums[high]] = [nums[high], nums[mid]];
      high--;
    }
  }
}
```

Problem 2: Merge Sorted Arrays

Question: Merge two sorted arrays into one sorted array.

Hint: Use two pointers approach.

Problem 3: Find Kth Largest Element

Question: Find the kth largest element in an unsorted array.

Hint: Use QuickSelect algorithm.

Problem 4: Sort Array by Frequency

Question: Sort array elements by their frequency of occurrence.

Hint: Use hash map to count frequencies, then custom sort.

🎯 Interview Tips

What Interviewers Look For:

1. **Algorithm selection**: Can you choose the right sorting algorithm?
2. **Implementation skills**: Can you code the algorithm correctly?
3. **Complexity analysis**: Do you understand time/space trade-offs?
4. **Edge cases**: Empty arrays, single elements, duplicates

Common Interview Questions:

- "Sort an array of 0s, 1s, and 2s"
- "Merge k sorted arrays"
- "Find the kth largest element"
- "Sort array with custom comparator"
- "Implement merge sort/quick sort"

Red Flags to Avoid:

- Using bubble sort for large datasets
- Not handling edge cases (empty arrays)
- Incorrect complexity analysis
- Not considering stability when required

Pro Tips:

1. **Know the classics**: Master merge sort and quick sort
2. **Understand trade-offs**: Time vs space, stable vs unstable
3. **Practice implementation**: Code without looking up
4. **Consider constraints**: Array size, memory limits, stability
5. **Optimize for the problem**: Sometimes counting sort is better

---

🚀 Key Takeaways

1. **No universal best algorithm** - Choice depends on constraints
2. **Merge sort for stability** - When relative order matters
3. **Quick sort for average performance** - Good general-purpose choice
4. **Insertion sort for small arrays** - Simple and efficient
5. **Specialized algorithms exist** - Counting/radix for specific data
6. **Practice makes perfect** - Implement algorithms from scratch

Next Chapter: We'll explore Searching Algorithms and see how sorted data enables efficient search techniques like binary search.

Chapter 8: Searching Algorithms - Finding Data Efficiently

🎯 What is Searching?

Searching is the process of finding a specific element or determining if it exists in a collection of data. It's one of the most fundamental operations in computer science and forms the backbone of many applications.

Why Searching Matters:

- **Data retrieval**: Core operation in databases and file systems
- **User experience**: Fast search improves application responsiveness
- **Algorithm foundation**: Many algorithms rely on efficient searching
- **Real-world applications**: Web search, autocomplete, recommendation systems
- **Problem solving**: Essential for many coding interview questions

Search Categories:

1. **Linear vs Binary**: Sequential vs divide-and-conquer approaches
2. **Exact vs Approximate**: Finding exact matches vs similar items
3. **Single vs Multiple**: Finding one occurrence vs all occurrences
4. **Static vs Dynamic**: Searching in fixed vs changing datasets

---

🔍 Search Algorithm Overview

Algorithm	Data Structure	Time (Best)	Time (Average)	Time (Worst)	Space	Prerequisites
Linear Search	Any	O(1)	O(n)	O(n)	O(1)	None
Binary Search	Sorted Array	O(1)	O(log n)	O(log n)	O(1)	Sorted data
Jump Search	Sorted Array	O(1)	O(√n)	O(√n)	O(1)	Sorted data
Interpolation Search	Uniformly Distributed	O(1)	O(log log n)	O(n)	O(1)	Sorted, uniform
Exponential Search	Sorted Array	O(1)	O(log n)	O(log n)	O(1)	Sorted data
Ternary Search	Sorted Array	O(1)	$O(\log_3 n)$	$O(\log_3 n)$	O(1)	Sorted data

---

💻 JavaScript Implementation

```
// Searching Algorithms Implementation

// 1. Linear Search (Sequential Search)
// Time: O(n), Space: O(1)
```

```
// Works on: Any data structure
function linearSearch(arr, target) {
  // Search through each element sequentially
  for (let i = 0; i < arr.length; i++) {
    if (arr[i] === target) {
      return i; // Return index if found
    }
  }
  return -1; // Return -1 if not found
}

// Linear Search - Find All Occurrences
// Time: O(n), Space: O(k) where k is number of occurrences
function linearSearchAll(arr, target) {
  const indices = [];

  for (let i = 0; i < arr.length; i++) {
    if (arr[i] === target) {
      indices.push(i);
    }
  }

  return indices;
}

// 2. Binary Search (Iterative)
// Time: O(log n), Space: O(1)
// Prerequisite: Array must be sorted
function binarySearch(arr, target) {
  let left = 0;
  let right = arr.length - 1;

  while (left <= right) {
    // Calculate middle index (avoid overflow)
    const mid = Math.floor(left + (right - left) / 2);

    // Check if target is at middle
    if (arr[mid] === target) {
      return mid;
    }

    // If target is smaller, search left half
    if (arr[mid] > target) {
      right = mid - 1;
    }
    // If target is larger, search right half
    else {
      left = mid + 1;
    }
  }

  return -1; // Target not found
}
```

```javascript
// Binary Search (Recursive)
// Time: O(log n), Space: O(log n)
function binarySearchRecursive(arr, target, left = 0, right = arr.length - 1) {
  // Base case: element not found
  if (left > right) {
    return -1;
  }

  const mid = Math.floor(left + (right - left) / 2);

  // Base case: element found
  if (arr[mid] === target) {
    return mid;
  }

  // Recursive cases
  if (arr[mid] > target) {
    return binarySearchRecursive(arr, target, left, mid - 1);
  } else {
    return binarySearchRecursive(arr, target, mid + 1, right);
  }
}

// Binary Search Variations

// Find First Occurrence (Leftmost)
// Time: O(log n), Space: O(1)
function findFirstOccurrence(arr, target) {
  let left = 0;
  let right = arr.length - 1;
  let result = -1;

  while (left <= right) {
    const mid = Math.floor(left + (right - left) / 2);

    if (arr[mid] === target) {
      result = mid;
      right = mid - 1; // Continue searching in left half
    } else if (arr[mid] < target) {
      left = mid + 1;
    } else {
      right = mid - 1;
    }
  }

  return result;
}

// Find Last Occurrence (Rightmost)
// Time: O(log n), Space: O(1)
function findLastOccurrence(arr, target) {
  let left = 0;
  let right = arr.length - 1;
  let result = -1;
```

```
  while (left <= right) {
    const mid = Math.floor(left + (right - left) / 2);

    if (arr[mid] === target) {
      result = mid;
      left = mid + 1; // Continue searching in right half
    } else if (arr[mid] < target) {
      left = mid + 1;
    } else {
      right = mid - 1;
    }
  }

  return result;
}

// Find Range of Target (First and Last Occurrence)
// Time: O(log n), Space: O(1)
function findRange(arr, target) {
  const first = findFirstOccurrence(arr, target);
  if (first === -1) {
    return [-1, -1];
  }

  const last = findLastOccurrence(arr, target);
  return [first, last];
}

// Find Insert Position
// Time: O(log n), Space: O(1)
function findInsertPosition(arr, target) {
  let left = 0;
  let right = arr.length - 1;

  while (left <= right) {
    const mid = Math.floor(left + (right - left) / 2);

    if (arr[mid] < target) {
      left = mid + 1;
    } else {
      right = mid - 1;
    }
  }

  return left;
}

// 3. Jump Search (Block Search)
// Time: O(√n), Space: O(1)
// Prerequisite: Array must be sorted
function jumpSearch(arr, target) {
  const n = arr.length;
  const step = Math.floor(Math.sqrt(n));
```

```
  let prev = 0;

  // Find the block where element is present
  while (arr[Math.min(step, n) - 1] < target) {
    prev = step;
    step += Math.floor(Math.sqrt(n));

    // If we reached end of array
    if (prev >= n) {
      return -1;
    }
  }

  // Linear search in the identified block
  while (arr[prev] < target) {
    prev++;

    // If we reached next block or end of array
    if (prev === Math.min(step, n)) {
      return -1;
    }
  }

  // If element is found
  if (arr[prev] === target) {
    return prev;
  }

  return -1;
}

// 4. Interpolation Search
// Time: O(log log n) average, O(n) worst, Space: O(1)
// Prerequisite: Array must be sorted and uniformly distributed
function interpolationSearch(arr, target) {
  let left = 0;
  let right = arr.length - 1;

  while (left <= right && target >= arr[left] && target <= arr[right]) {
    // If array has only one element
    if (left === right) {
      return arr[left] === target ? left : -1;
    }

    // Calculate position using interpolation formula
    const pos =
      left +
      Math.floor(
        ((target - arr[left]) * (right - left)) / (arr[right] - arr[left])
      );

    // Target found
    if (arr[pos] === target) {
      return pos;
```

```
    }

    // If target is larger, search right subarray
    if (arr[pos] < target) {
      left = pos + 1;
    }
    // If target is smaller, search left subarray
    else {
      right = pos - 1;
    }
  }

  return -1;
}

// 5. Exponential Search (Doubling Search)
// Time: O(log n), Space: O(1)
// Prerequisite: Array must be sorted
function exponentialSearch(arr, target) {
  const n = arr.length;

  // If target is at first position
  if (arr[0] === target) {
    return 0;
  }

  // Find range for binary search by repeated doubling
  let bound = 1;
  while (bound < n && arr[bound] <= target) {
    bound *= 2;
  }

  // Perform binary search in the found range
  return binarySearchRange(arr, target, bound / 2, Math.min(bound, n - 1));
}

// Helper function for exponential search
function binarySearchRange(arr, target, left, right) {
  while (left <= right) {
    const mid = Math.floor(left + (right - left) / 2);

    if (arr[mid] === target) {
      return mid;
    }

    if (arr[mid] > target) {
      right = mid - 1;
    } else {
      left = mid + 1;
    }
  }

  return -1;
}
```

```
// 6. Ternary Search
// Time: O(log₃ n), Space: O(1)
// Prerequisite: Array must be sorted
function ternarySearch(arr, target, left = 0, right = arr.length - 1) {
  if (left > right) {
    return -1;
  }

  // Divide array into three parts
  const mid1 = left + Math.floor((right - left) / 3);
  const mid2 = right - Math.floor((right - left) / 3);

  // Check if target is at either midpoint
  if (arr[mid1] === target) {
    return mid1;
  }
  if (arr[mid2] === target) {
    return mid2;
  }

  // Determine which third to search
  if (target < arr[mid1]) {
    return ternarySearch(arr, target, left, mid1 - 1);
  } else if (target > arr[mid2]) {
    return ternarySearch(arr, target, mid2 + 1, right);
  } else {
    return ternarySearch(arr, target, mid1 + 1, mid2 - 1);
  }
}

// Advanced Search Algorithms

// 7. Search in Rotated Sorted Array
// Time: O(log n), Space: O(1)
function searchRotatedArray(arr, target) {
  let left = 0;
  let right = arr.length - 1;

  while (left <= right) {
    const mid = Math.floor(left + (right - left) / 2);

    if (arr[mid] === target) {
      return mid;
    }

    // Check which half is sorted
    if (arr[left] <= arr[mid]) {
      // Left half is sorted
      if (target >= arr[left] && target < arr[mid]) {
        right = mid - 1;
      } else {
        left = mid + 1;
      }
```

```
    } else {
      // Right half is sorted
      if (target > arr[mid] && target <= arr[right]) {
        left = mid + 1;
      } else {
        right = mid - 1;
      }
    }
  }

  return -1;
}

// 8. Search in 2D Matrix
// Time: O(log(m*n)), Space: O(1)
function searchMatrix(matrix, target) {
  if (!matrix || matrix.length === 0 || matrix[0].length === 0) {
    return false;
  }

  const m = matrix.length;
  const n = matrix[0].length;
  let left = 0;
  let right = m * n - 1;

  while (left <= right) {
    const mid = Math.floor(left + (right - left) / 2);
    const midValue = matrix[Math.floor(mid / n)][mid % n];

    if (midValue === target) {
      return true;
    } else if (midValue < target) {
      left = mid + 1;
    } else {
      right = mid - 1;
    }
  }

  return false;
}

// 9. Find Peak Element
// Time: O(log n), Space: O(1)
function findPeakElement(arr) {
  let left = 0;
  let right = arr.length - 1;

  while (left < right) {
    const mid = Math.floor(left + (right - left) / 2);

    if (arr[mid] > arr[mid + 1]) {
      // Peak is in left half (including mid)
      right = mid;
    } else {
```

```
      // Peak is in right half
      left = mid + 1;
    }
  }

  return left;
}

// 10. Find Minimum in Rotated Sorted Array
// Time: O(log n), Space: O(1)
function findMinRotated(arr) {
  let left = 0;
  let right = arr.length - 1;

  while (left < right) {
    const mid = Math.floor(left + (right - left) / 2);

    if (arr[mid] > arr[right]) {
      // Minimum is in right half
      left = mid + 1;
    } else {
      // Minimum is in left half (including mid)
      right = mid;
    }
  }

  return arr[left];
}

// Search Utilities
class SearchUtils {
  // Binary search for closest element
  static findClosest(arr, target) {
    if (arr.length === 0) return -1;

    let left = 0;
    let right = arr.length - 1;
    let closest = 0;

    while (left <= right) {
      const mid = Math.floor(left + (right - left) / 2);

      // Update closest if current element is closer
      if (Math.abs(arr[mid] - target) < Math.abs(arr[closest] - target)) {
        closest = mid;
      }

      if (arr[mid] === target) {
        return mid;
      } else if (arr[mid] < target) {
        left = mid + 1;
      } else {
        right = mid - 1;
      }
```

```
    }

    return closest;
  }

  // Count occurrences using binary search
  static countOccurrences(arr, target) {
    const range = findRange(arr, target);
    if (range[0] === -1) {
      return 0;
    }
    return range[1] - range[0] + 1;
  }

  // Search in infinite array (simulated with large array)
  static searchInfinite(arr, target) {
    let bound = 1;

    // Find upper bound
    while (bound < arr.length && arr[bound] < target) {
      bound *= 2;
    }

    // Binary search in the range
    return binarySearchRange(
      arr,
      target,
      bound / 2,
      Math.min(bound, arr.length - 1)
    );
  }

  // Find floor and ceiling
  static findFloorCeiling(arr, target) {
    let floor = -1;
    let ceiling = -1;

    for (let i = 0; i < arr.length; i++) {
      if (arr[i] <= target) {
        floor = arr[i];
      }
      if (arr[i] >= target && ceiling === -1) {
        ceiling = arr[i];
      }
    }

    return { floor, ceiling };
  }

  // Performance measurement
  static measureSearchPerformance(searchFunc, arr, target, name) {
    const start = performance.now();
    const result = searchFunc(arr, target);
    const end = performance.now();
```

```
    console.log(`${name}: ${(end - start).toFixed(4)}ms, Result: ${result}`);
    return result;
  }

  // Generate test data
  static generateSortedArray(size, min = 0, max = 1000) {
    const arr = [];
    for (let i = 0; i < size; i++) {
      arr.push(Math.floor(Math.random() * (max - min + 1)) + min);
    }
    return arr.sort((a, b) => a - b);
  }

  // Fuzzy search (simple implementation)
  static fuzzySearch(arr, target, threshold = 2) {
    const results = [];

    for (let i = 0; i < arr.length; i++) {
      if (typeof arr[i] === "string" && typeof target === "string") {
        const distance = SearchUtils.levenshteinDistance(arr[i], target);
        if (distance <= threshold) {
          results.push({ index: i, value: arr[i], distance });
        }
      }
    }

    return results.sort((a, b) => a.distance - b.distance);
  }

  // Levenshtein distance for fuzzy search
  static levenshteinDistance(str1, str2) {
    const matrix = [];

    for (let i = 0; i <= str2.length; i++) {
      matrix[i] = [i];
    }

    for (let j = 0; j <= str1.length; j++) {
      matrix[0][j] = j;
    }

    for (let i = 1; i <= str2.length; i++) {
      for (let j = 1; j <= str1.length; j++) {
        if (str2.charAt(i - 1) === str1.charAt(j - 1)) {
          matrix[i][j] = matrix[i - 1][j - 1];
        } else {
          matrix[i][j] = Math.min(
            matrix[i - 1][j - 1] + 1, // substitution
            matrix[i][j - 1] + 1, // insertion
            matrix[i - 1][j] + 1 // deletion
          );
        }
      }
```

```
    }

    return matrix[str2.length][str1.length];
  }
}

// Example Usage and Testing
console.log("=== Searching Algorithms Demo ===");

// Test data
const sortedArray = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25];
const target = 13;

console.log("Sorted array:", sortedArray);
console.log("Target:", target);

console.log("\n=== Basic Search Algorithms ===");
console.log("Linear Search:", linearSearch(sortedArray, target));
console.log("Binary Search (Iterative):", binarySearch(sortedArray, target));
console.log(
  "Binary Search (Recursive):",
  binarySearchRecursive(sortedArray, target)
);

console.log("\n=== Advanced Search Algorithms ===");
console.log("Jump Search:", jumpSearch(sortedArray, target));
console.log("Interpolation Search:", interpolationSearch(sortedArray, target));
console.log("Exponential Search:", exponentialSearch(sortedArray, target));
console.log("Ternary Search:", ternarySearch(sortedArray, target));

// Test with duplicates
const arrayWithDuplicates = [1, 2, 2, 2, 3, 4, 4, 5, 6, 6, 6, 7];
console.log("\n=== Binary Search Variations ===");
console.log("Array with duplicates:", arrayWithDuplicates);
console.log(
  "First occurrence of 2:",
  findFirstOccurrence(arrayWithDuplicates, 2)
);
console.log(
  "Last occurrence of 6:",
  findLastOccurrence(arrayWithDuplicates, 6)
);
console.log("Range of 4:", findRange(arrayWithDuplicates, 4));
console.log(
  "Insert position for 3.5:",
  findInsertPosition(arrayWithDuplicates, 3.5)
);

// Advanced problems
console.log("\n=== Advanced Search Problems ===");
const rotatedArray = [4, 5, 6, 7, 0, 1, 2];
console.log("Rotated array:", rotatedArray);
console.log("Search 0 in rotated array:", searchRotatedArray(rotatedArray, 0));
console.log("Find minimum in rotated array:", findMinRotated(rotatedArray));
```

```
const peakArray = [1, 2, 3, 1];
console.log("Peak array:", peakArray);
console.log("Peak element index:", findPeakElement(peakArray));

// 2D Matrix search
const matrix = [
  [1, 4, 7, 11],
  [2, 5, 8, 12],
  [3, 6, 9, 16],
  [10, 13, 14, 17],
];
console.log("\n=== 2D Matrix Search ===");
console.log("Search 5 in matrix:", searchMatrix(matrix, 5));
console.log("Search 13 in matrix:", searchMatrix(matrix, 13));

// Performance comparison
console.log("\n=== Performance Comparison ===");
const largeArray = SearchUtils.generateSortedArray(10000);
const searchTarget = largeArray[Math.floor(Math.random() * largeArray.length)];

SearchUtils.measureSearchPerformance(
  linearSearch,
  largeArray,
  searchTarget,
  "Linear Search"
);
SearchUtils.measureSearchPerformance(
  binarySearch,
  largeArray,
  searchTarget,
  "Binary Search"
);
SearchUtils.measureSearchPerformance(
  jumpSearch,
  largeArray,
  searchTarget,
  "Jump Search"
);
SearchUtils.measureSearchPerformance(
  interpolationSearch,
  largeArray,
  searchTarget,
  "Interpolation Search"
);

// Utility demonstrations
console.log("\n=== Search Utilities ===");
console.log(
  "Closest to 14 in sorted array:",
  SearchUtils.findClosest(sortedArray, 14)
);
console.log(
  "Count of 2 in duplicates array:",
```

```
  SearchUtils.countOccurrences(arrayWithDuplicates, 2)
);

const floorCeiling = SearchUtils.findFloorCeiling(sortedArray, 12);
console.log("Floor and ceiling of 12:", floorCeiling);

// Fuzzy search demo
const names = ["Alice", "Bob", "Charlie", "David", "Eve", "Frank"];
const fuzzyResults = SearchUtils.fuzzySearch(names, "Charli", 2);
console.log("\n=== Fuzzy Search ===");
console.log('Fuzzy search for "Charli":', fuzzyResults);
```

🔧 C++ Implementation

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>
#include <chrono>
#include <random>
using namespace std;
using namespace std::chrono;

// Searching Algorithms Implementation

// 1. Linear Search
// Time: O(n), Space: O(1)
int linearSearch(const vector<int>& arr, int target) {
    for (int i = 0; i < arr.size(); i++) {
        if (arr[i] == target) {
            return i;
        }
    }
    return -1;
}

// Linear Search - Find All Occurrences
vector<int> linearSearchAll(const vector<int>& arr, int target) {
    vector<int> indices;

    for (int i = 0; i < arr.size(); i++) {
        if (arr[i] == target) {
            indices.push_back(i);
        }
    }

    return indices;
}

// 2. Binary Search (Iterative)
```

```cpp
// Time: O(log n), Space: O(1)
int binarySearch(const vector<int>& arr, int target) {
    int left = 0;
    int right = arr.size() - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == target) {
            return mid;
        }

        if (arr[mid] > target) {
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }

    return -1;
}

// Binary Search (Recursive)
// Time: O(log n), Space: O(log n)
int binarySearchRecursive(const vector<int>& arr, int target, int left, int right)
{
    if (left > right) {
        return -1;
    }

    int mid = left + (right - left) / 2;

    if (arr[mid] == target) {
        return mid;
    }

    if (arr[mid] > target) {
        return binarySearchRecursive(arr, target, left, mid - 1);
    } else {
        return binarySearchRecursive(arr, target, mid + 1, right);
    }
}

// Find First Occurrence
int findFirstOccurrence(const vector<int>& arr, int target) {
    int left = 0;
    int right = arr.size() - 1;
    int result = -1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == target) {
            result = mid;
```

```
            right = mid - 1; // Continue searching left
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return result;
}

// Find Last Occurrence
int findLastOccurrence(const vector<int>& arr, int target) {
    int left = 0;
    int right = arr.size() - 1;
    int result = -1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == target) {
            result = mid;
            left = mid + 1; // Continue searching right
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return result;
}

// Find Range
pair<int, int> findRange(const vector<int>& arr, int target) {
    int first = findFirstOccurrence(arr, target);
    if (first == -1) {
        return {-1, -1};
    }

    int last = findLastOccurrence(arr, target);
    return {first, last};
}

// 3. Jump Search
// Time: O(√n), Space: O(1)
int jumpSearch(const vector<int>& arr, int target) {
    int n = arr.size();
    int step = sqrt(n);
    int prev = 0;

    // Find the block where element is present
    while (arr[min(step, n) - 1] < target) {
        prev = step;
```

```
        step += sqrt(n);

        if (prev >= n) {
            return -1;
        }
    }

    // Linear search in the block
    while (arr[prev] < target) {
        prev++;

        if (prev == min(step, n)) {
            return -1;
        }
    }

    if (arr[prev] == target) {
        return prev;
    }

    return -1;
}

// 4. Interpolation Search
// Time: O(log log n) average, O(n) worst
int interpolationSearch(const vector<int>& arr, int target) {
    int left = 0;
    int right = arr.size() - 1;

    while (left <= right && target >= arr[left] && target <= arr[right]) {
        if (left == right) {
            return arr[left] == target ? left : -1;
        }

        // Calculate position using interpolation formula
        int pos = left + ((double)(target - arr[left]) * (right - left)) /
(arr[right] - arr[left]);

        if (arr[pos] == target) {
            return pos;
        }

        if (arr[pos] < target) {
            left = pos + 1;
        } else {
            right = pos - 1;
        }
    }

    return -1;
}

// 5. Exponential Search
// Time: O(log n), Space: O(1)
```

```cpp
int binarySearchRange(const vector<int>& arr, int target, int left, int right) {
    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == target) {
            return mid;
        }

        if (arr[mid] > target) {
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }

    return -1;
}

int exponentialSearch(const vector<int>& arr, int target) {
    int n = arr.size();

    if (arr[0] == target) {
        return 0;
    }

    // Find range for binary search
    int bound = 1;
    while (bound < n && arr[bound] <= target) {
        bound *= 2;
    }

    return binarySearchRange(arr, target, bound / 2, min(bound, n - 1));
}

// 6. Ternary Search
// Time: O(log₃ n), Space: O(log n)
int ternarySearch(const vector<int>& arr, int target, int left, int right) {
    if (left > right) {
        return -1;
    }

    int mid1 = left + (right - left) / 3;
    int mid2 = right - (right - left) / 3;

    if (arr[mid1] == target) {
        return mid1;
    }
    if (arr[mid2] == target) {
        return mid2;
    }

    if (target < arr[mid1]) {
        return ternarySearch(arr, target, left, mid1 - 1);
    } else if (target > arr[mid2]) {
```

```
        return ternarySearch(arr, target, mid2 + 1, right);
    } else {
        return ternarySearch(arr, target, mid1 + 1, mid2 - 1);
    }
}

// Advanced Search Problems

// Search in Rotated Sorted Array
int searchRotatedArray(const vector<int>& arr, int target) {
    int left = 0;
    int right = arr.size() - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == target) {
            return mid;
        }

        if (arr[left] <= arr[mid]) {
            // Left half is sorted
            if (target >= arr[left] && target < arr[mid]) {
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        } else {
            // Right half is sorted
            if (target > arr[mid] && target <= arr[right]) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
    }

    return -1;
}

// Find Peak Element
int findPeakElement(const vector<int>& arr) {
    int left = 0;
    int right = arr.size() - 1;

    while (left < right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] > arr[mid + 1]) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }
```

```
    return left;
}

// Find Minimum in Rotated Sorted Array
int findMinRotated(const vector<int>& arr) {
    int left = 0;
    int right = arr.size() - 1;

    while (left < right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] > arr[right]) {
            left = mid + 1;
        } else {
            right = mid;
        }
    }

    return arr[left];
}

// Search Utilities
class SearchUtils {
public:
    // Generate sorted array for testing
    static vector<int> generateSortedArray(int size, int minVal = 0, int maxVal =
1000) {
        vector<int> arr(size);
        random_device rd;
        mt19937 gen(rd());
        uniform_int_distribution<> dis(minVal, maxVal);

        for (int i = 0; i < size; i++) {
            arr[i] = dis(gen);
        }

        sort(arr.begin(), arr.end());
        return arr;
    }

    // Measure search performance
    static void measurePerformance(int (*searchFunc)(const vector<int>&, int),
                                   const vector<int>& arr, int target, const
string& name) {
        auto start = high_resolution_clock::now();
        int result = searchFunc(arr, target);
        auto end = high_resolution_clock::now();

        auto duration = duration_cast<microseconds>(end - start);
        cout << name << ": " << duration.count() << " microseconds, Result: " <<
result << endl;
    }
```

```
    // Find closest element
    static int findClosest(const vector<int>& arr, int target) {
        if (arr.empty()) return -1;

        int left = 0;
        int right = arr.size() - 1;
        int closest = 0;

        while (left <= right) {
            int mid = left + (right - left) / 2;

            if (abs(arr[mid] - target) < abs(arr[closest] - target)) {
                closest = mid;
            }

            if (arr[mid] == target) {
                return mid;
            } else if (arr[mid] < target) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }

        return closest;
    }

    // Count occurrences
    static int countOccurrences(const vector<int>& arr, int target) {
        auto range = findRange(arr, target);
        if (range.first == -1) {
            return 0;
        }
        return range.second - range.first + 1;
    }
};

// Print vector utility
void printVector(const vector<int>& arr, const string& label = "") {
    if (!label.empty()) {
        cout << label << ": ";
    }
    for (int x : arr) {
        cout << x << " ";
    }
    cout << endl;
}

// Example Usage
int main() {
    cout << "=== Searching Algorithms Demo ===" << endl;

    // Test data
    vector<int> sortedArray = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25};
```

```
    int target = 13;

    printVector(sortedArray, "Sorted array");
    cout << "Target: " << target << endl;

    cout << "\n=== Basic Search Algorithms ===" << endl;
    cout << "Linear Search: " << linearSearch(sortedArray, target) << endl;
    cout << "Binary Search (Iterative): " << binarySearch(sortedArray, target) <<
endl;
    cout << "Binary Search (Recursive): " << binarySearchRecursive(sortedArray,
target, 0, sortedArray.size() - 1) << endl;

    cout << "\n=== Advanced Search Algorithms ===" << endl;
    cout << "Jump Search: " << jumpSearch(sortedArray, target) << endl;
    cout << "Interpolation Search: " << interpolationSearch(sortedArray, target)
<< endl;
    cout << "Exponential Search: " << exponentialSearch(sortedArray, target) <<
endl;
    cout << "Ternary Search: " << ternarySearch(sortedArray, target, 0,
sortedArray.size() - 1) << endl;

    // Test with duplicates
    vector<int> arrayWithDuplicates = {1, 2, 2, 2, 3, 4, 4, 5, 6, 6, 6, 7};
    cout << "\n=== Binary Search Variations ===" << endl;
    printVector(arrayWithDuplicates, "Array with duplicates");
    cout << "First occurrence of 2: " << findFirstOccurrence(arrayWithDuplicates,
2) << endl;
    cout << "Last occurrence of 6: " << findLastOccurrence(arrayWithDuplicates, 6)
<< endl;

    auto range = findRange(arrayWithDuplicates, 4);
    cout << "Range of 4: [" << range.first << ", " << range.second << "]" << endl;

    // Advanced problems
    cout << "\n=== Advanced Search Problems ===" << endl;
    vector<int> rotatedArray = {4, 5, 6, 7, 0, 1, 2};
    printVector(rotatedArray, "Rotated array");
    cout << "Search 0 in rotated array: " << searchRotatedArray(rotatedArray, 0)
<< endl;
    cout << "Find minimum in rotated array: " << findMinRotated(rotatedArray) <<
endl;

    vector<int> peakArray = {1, 2, 3, 1};
    printVector(peakArray, "Peak array");
    cout << "Peak element index: " << findPeakElement(peakArray) << endl;

    // Performance comparison
    cout << "\n=== Performance Comparison (10000 elements) ===" << endl;
    vector<int> largeArray = SearchUtils::generateSortedArray(10000);
    int searchTarget = largeArray[rand() % largeArray.size()];

    SearchUtils::measurePerformance(linearSearch, largeArray, searchTarget,
"Linear Search");
    SearchUtils::measurePerformance(binarySearch, largeArray, searchTarget,
```

```
"Binary Search");
    SearchUtils::measurePerformance(jumpSearch, largeArray, searchTarget, "Jump
Search");
    SearchUtils::measurePerformance(interpolationSearch, largeArray, searchTarget,
"Interpolation Search");

    // Utility demonstrations
    cout << "\n=== Search Utilities ===" << endl;
    cout << "Closest to 14 in sorted array: " <<
SearchUtils::findClosest(sortedArray, 14) << endl;
    cout << "Count of 2 in duplicates array: " <<
SearchUtils::countOccurrences(arrayWithDuplicates, 2) << endl;

    return 0;
}
```

⚡ Performance Analysis

Algorithm Selection Guide:

1. **Linear Search**:
   - ✅ Unsorted data
   - ✅ Small datasets (< 100 elements)
   - ✅ Simple implementation needed
   - ❌ Large datasets
2. **Binary Search**:
   - ✅ Sorted data
   - ✅ Large datasets
   - ✅ Guaranteed O(log n) performance
   - ❌ Unsorted data
3. **Jump Search**:
   - ✅ Sorted data, better than linear
   - ✅ When binary search overhead is concern
   - ❌ Generally worse than binary search
4. **Interpolation Search**:
   - ✅ Uniformly distributed sorted data
   - ✅ Very large datasets
   - ❌ Non-uniform distribution

Common Pitfalls:

1. **Integer overflow**: In mid calculation

```
// Bad: Can overflow
const mid = Math.floor((left + right) / 2);

// Good: Prevents overflow
const mid = Math.floor(left + (right - left) / 2);
```

2. **Infinite loops**: Incorrect boundary updates

```
// Bad: Can cause infinite loop
if (arr[mid] < target) {
  left = mid; // Should be mid + 1
}
```

3. **Off-by-one errors**: Incorrect loop conditions

```
// Correct condition for binary search
while (left <= right) { // Note: <=, not <
```

4. **Unsorted data**: Using binary search on unsorted array

Practice Problems

Problem 1: Search Insert Position

Question: Find the index where target should be inserted in sorted array.

Example: `nums = [1,3,5,6], target = 5` → `2`

Solution:

```
function searchInsert(nums, target) {
  let left = 0,
    right = nums.length - 1;

  while (left <= right) {
    const mid = Math.floor(left + (right - left) / 2);

    if (nums[mid] === target) {
      return mid;
    } else if (nums[mid] < target) {
      left = mid + 1;
    } else {
      right = mid - 1;
    }
  }
```

```
    return left;
}
```

Problem 2: Find First and Last Position

Question: Find first and last position of target in sorted array.

Hint: Use modified binary search to find leftmost and rightmost occurrences.

Problem 3: Search in 2D Matrix II

Question: Search target in matrix where each row and column is sorted.

Hint: Start from top-right or bottom-left corner.

Problem 4: Find Minimum in Rotated Sorted Array

Question: Find minimum element in rotated sorted array.

Hint: Use binary search with rotation logic.

🎯 Interview Tips

What Interviewers Look For:

1. **Algorithm selection**: Can you choose the right search algorithm?
2. **Implementation correctness**: Handle edge cases and boundaries
3. **Complexity analysis**: Understand time/space trade-offs
4. **Problem variations**: Adapt to different constraints

Common Interview Patterns:

- **Binary search variations**: First/last occurrence, insert position
- **Rotated arrays**: Search in rotated sorted arrays
- **2D searching**: Matrix search problems
- **Peak finding**: Local maxima/minima
- **Range queries**: Finding ranges and counts

Red Flags to Avoid:

- Using linear search when binary search is possible
- Integer overflow in mid calculation
- Infinite loops due to incorrect boundaries
- Not handling empty arrays or edge cases

Pro Tips:

1. **Clarify requirements**: Sorted? Duplicates? Return index or boolean?
2. **Start with brute force**: Then optimize to binary search

3. **Draw the search space**: Visualize the problem
4. **Test edge cases**: Empty array, single element, target not found
5. **Practice variations**: Master the binary search template

---

🚀 Key Takeaways

1. **Binary search is powerful** - O(log n) vs O(n) is huge for large data
2. **Sorted data enables efficiency** - Always consider if sorting first helps
3. **Master the template** - Binary search has many variations
4. **Handle edge cases** - Empty arrays, duplicates, boundaries
5. **Choose wisely** - Linear search is fine for small datasets
6. **Practice variations** - First/last occurrence, rotated arrays, 2D matrices

Next Chapter: We'll explore Tree Traversals and see how searching principles apply to hierarchical data structures.

Chapter 9: Tree Traversals - Navigating Hierarchical Data

🎯 What are Tree Traversals?

Tree traversal is the process of visiting each node in a tree data structure exactly once in a systematic way. Unlike linear data structures (arrays, linked lists), trees are hierarchical, so there are multiple ways to traverse them.

Why Tree Traversals Matter:

- **Data processing**: Process all nodes in a specific order
- **Tree algorithms**: Foundation for search, insertion, deletion
- **Expression evaluation**: Parse mathematical/logical expressions
- **File systems**: Navigate directory structures
- **Database indexing**: B-tree operations
- **Compiler design**: Abstract syntax tree processing

Traversal Categories:

1. **Depth-First Search (DFS)**: Go deep before going wide
    - Inorder (Left → Root → Right)
    - Preorder (Root → Left → Right)
    - Postorder (Left → Right → Root)
2. **Breadth-First Search (BFS)**: Go wide before going deep
    - Level-order traversal

🌳 Tree Traversal Overview

Traversal	Order	Use Cases	Time	Space	Stack Usage
Inorder	L→R→R	BST sorted output, expression evaluation	O(n)	O(h)	Recursive: O(h)
Preorder	R→L→R	Tree copying, prefix expressions	O(n)	O(h)	Recursive: O(h)
Postorder	L→R→R	Tree deletion, postfix expressions	O(n)	O(h)	Recursive: O(h)
Level-order	Level by level	Tree printing, shortest path	O(n)	O(w)	Queue: O(w)

h = height of tree, w = maximum width of tree

💻 JavaScript Implementation

```
// Tree Node Definition
class TreeNode {
  constructor(val, left = null, right = null) {
    this.val = val;
    this.left = left;
    this.right = right;
  }
}

// Tree Traversal Implementations

// ===== DEPTH-FIRST SEARCH (DFS) TRAVERSALS =====

// 1. INORDER TRAVERSAL (Left → Root → Right)
// Use case: Get sorted sequence from BST

// Recursive Inorder
// Time: O(n), Space: O(h) where h is height
function inorderRecursive(root, result = []) {
  if (root === null) {
    return result;
  }

  // Traverse left subtree
  inorderRecursive(root.left, result);

  // Visit root
  result.push(root.val);

  // Traverse right subtree
  inorderRecursive(root.right, result);

  return result;
```

```
}

// Iterative Inorder using Stack
// Time: O(n), Space: O(h)
function inorderIterative(root) {
  const result = [];
  const stack = [];
  let current = root;

  while (current !== null || stack.length > 0) {
    // Go to the leftmost node
    while (current !== null) {
      stack.push(current);
      current = current.left;
    }

    // Current is null, so we backtrack
    current = stack.pop();
    result.push(current.val);

    // Visit right subtree
    current = current.right;
  }

  return result;
}

// Morris Inorder Traversal (No extra space)
// Time: O(n), Space: O(1)
function inorderMorris(root) {
  const result = [];
  let current = root;

  while (current !== null) {
    if (current.left === null) {
      // No left child, visit current and go right
      result.push(current.val);
      current = current.right;
    } else {
      // Find inorder predecessor
      let predecessor = current.left;
      while (predecessor.right !== null && predecessor.right !== current) {
        predecessor = predecessor.right;
      }

      if (predecessor.right === null) {
        // Make current the right child of predecessor
        predecessor.right = current;
        current = current.left;
      } else {
        // Revert the changes
        predecessor.right = null;
        result.push(current.val);
        current = current.right;
```

```
      }
    }
  }

  return result;
}

// 2. PREORDER TRAVERSAL (Root → Left → Right)
// Use case: Tree copying, prefix expressions

// Recursive Preorder
// Time: O(n), Space: O(h)
function preorderRecursive(root, result = []) {
  if (root === null) {
    return result;
  }

  // Visit root
  result.push(root.val);

  // Traverse left subtree
  preorderRecursive(root.left, result);

  // Traverse right subtree
  preorderRecursive(root.right, result);

  return result;
}

// Iterative Preorder using Stack
// Time: O(n), Space: O(h)
function preorderIterative(root) {
  if (root === null) return [];

  const result = [];
  const stack = [root];

  while (stack.length > 0) {
    const node = stack.pop();
    result.push(node.val);

    // Push right first, then left (stack is LIFO)
    if (node.right !== null) {
      stack.push(node.right);
    }
    if (node.left !== null) {
      stack.push(node.left);
    }
  }

  return result;
}

// Morris Preorder Traversal
```

```
// Time: O(n), Space: O(1)
function preorderMorris(root) {
  const result = [];
  let current = root;

  while (current !== null) {
    if (current.left === null) {
      result.push(current.val);
      current = current.right;
    } else {
      let predecessor = current.left;
      while (predecessor.right !== null && predecessor.right !== current) {
        predecessor = predecessor.right;
      }

      if (predecessor.right === null) {
        result.push(current.val); // Visit before going left
        predecessor.right = current;
        current = current.left;
      } else {
        predecessor.right = null;
        current = current.right;
      }
    }
  }

  return result;
}

// 3. POSTORDER TRAVERSAL (Left → Right → Root)
// Use case: Tree deletion, postfix expressions

// Recursive Postorder
// Time: O(n), Space: O(h)
function postorderRecursive(root, result = []) {
  if (root === null) {
    return result;
  }

  // Traverse left subtree
  postorderRecursive(root.left, result);

  // Traverse right subtree
  postorderRecursive(root.right, result);

  // Visit root
  result.push(root.val);

  return result;
}

// Iterative Postorder using Two Stacks
// Time: O(n), Space: O(h)
function postorderIterativeTwoStacks(root) {
```

```
  if (root === null) return [];

  const stack1 = [root];
  const stack2 = [];
  const result = [];

  // First stack for traversal, second for result order
  while (stack1.length > 0) {
    const node = stack1.pop();
    stack2.push(node);

    if (node.left !== null) {
      stack1.push(node.left);
    }
    if (node.right !== null) {
      stack1.push(node.right);
    }
  }

  // Pop from second stack to get postorder
  while (stack2.length > 0) {
    result.push(stack2.pop().val);
  }

  return result;
}

// Iterative Postorder using One Stack
// Time: O(n), Space: O(h)
function postorderIterativeOneStack(root) {
  if (root === null) return [];

  const result = [];
  const stack = [];
  let lastVisited = null;
  let current = root;

  while (stack.length > 0 || current !== null) {
    if (current !== null) {
      stack.push(current);
      current = current.left;
    } else {
      const peekNode = stack[stack.length - 1];

      // If right child exists and hasn't been processed yet
      if (peekNode.right !== null && lastVisited !== peekNode.right) {
        current = peekNode.right;
      } else {
        result.push(peekNode.val);
        lastVisited = stack.pop();
      }
    }
  }
```

```
  return result;
}

// ===== BREADTH-FIRST SEARCH (BFS) TRAVERSAL =====

// 4. LEVEL-ORDER TRAVERSAL (Breadth-First)
// Use case: Tree printing, shortest path in unweighted trees

// Basic Level-order using Queue
// Time: O(n), Space: O(w) where w is maximum width
function levelOrder(root) {
  if (root === null) return [];

  const result = [];
  const queue = [root];

  while (queue.length > 0) {
    const node = queue.shift();
    result.push(node.val);

    if (node.left !== null) {
      queue.push(node.left);
    }
    if (node.right !== null) {
      queue.push(node.right);
    }
  }

  return result;
}

// Level-order with Level Separation
// Returns array of arrays, each representing a level
function levelOrderByLevels(root) {
  if (root === null) return [];

  const result = [];
  const queue = [root];

  while (queue.length > 0) {
    const levelSize = queue.length;
    const currentLevel = [];

    for (let i = 0; i < levelSize; i++) {
      const node = queue.shift();
      currentLevel.push(node.val);

      if (node.left !== null) {
        queue.push(node.left);
      }
      if (node.right !== null) {
        queue.push(node.right);
      }
    }
```

```
    result.push(currentLevel);
  }

  return result;
}

// Zigzag Level-order Traversal
// Alternate between left-to-right and right-to-left
function zigzagLevelOrder(root) {
  if (root === null) return [];

  const result = [];
  const queue = [root];
  let leftToRight = true;

  while (queue.length > 0) {
    const levelSize = queue.length;
    const currentLevel = [];

    for (let i = 0; i < levelSize; i++) {
      const node = queue.shift();

      if (leftToRight) {
        currentLevel.push(node.val);
      } else {
        currentLevel.unshift(node.val);
      }

      if (node.left !== null) {
        queue.push(node.left);
      }
      if (node.right !== null) {
        queue.push(node.right);
      }
    }

    result.push(currentLevel);
    leftToRight = !leftToRight;
  }

  return result;
}

// Reverse Level-order Traversal (Bottom-up)
function reverseLevelOrder(root) {
  if (root === null) return [];

  const result = [];
  const queue = [root];

  while (queue.length > 0) {
    const levelSize = queue.length;
    const currentLevel = [];
```

```
    for (let i = 0; i < levelSize; i++) {
      const node = queue.shift();
      currentLevel.push(node.val);

      if (node.left !== null) {
        queue.push(node.left);
      }
      if (node.right !== null) {
        queue.push(node.right);
      }
    }

    result.unshift(currentLevel); // Add to beginning
  }

  return result;
}

// ===== ADVANCED TRAVERSAL TECHNIQUES =====

// Vertical Order Traversal
// Group nodes by their horizontal distance from root
function verticalOrder(root) {
  if (root === null) return [];

  const columnMap = new Map();
  const queue = [[root, 0]]; // [node, column]

  while (queue.length > 0) {
    const [node, col] = queue.shift();

    if (!columnMap.has(col)) {
      columnMap.set(col, []);
    }
    columnMap.get(col).push(node.val);

    if (node.left !== null) {
      queue.push([node.left, col - 1]);
    }
    if (node.right !== null) {
      queue.push([node.right, col + 1]);
    }
  }

  // Sort by column and return values
  const sortedColumns = Array.from(columnMap.keys()).sort((a, b) => a - b);
  return sortedColumns.map((col) => columnMap.get(col));
}

// Boundary Traversal
// Traverse the boundary of the tree (anticlockwise)
function boundaryTraversal(root) {
  if (root === null) return [];
```

```
    const result = [];

    // Add root
    result.push(root.val);

    if (root.left === null && root.right === null) {
      return result; // Single node
    }

    // Add left boundary (excluding leaves)
    function addLeftBoundary(node) {
      if (node === null || (node.left === null && node.right === null)) {
        return;
      }

      result.push(node.val);

      if (node.left !== null) {
        addLeftBoundary(node.left);
      } else {
        addLeftBoundary(node.right);
      }
    }

    // Add leaves
    function addLeaves(node) {
      if (node === null) return;

      if (node.left === null && node.right === null) {
        result.push(node.val);
        return;
      }

      addLeaves(node.left);
      addLeaves(node.right);
    }

    // Add right boundary (excluding leaves, in reverse)
    function addRightBoundary(node) {
      if (node === null || (node.left === null && node.right === null)) {
        return;
      }

      if (node.right !== null) {
        addRightBoundary(node.right);
      } else {
        addRightBoundary(node.left);
      }

      result.push(node.val);
    }

    addLeftBoundary(root.left);
```

```
  addLeaves(root);
  addRightBoundary(root.right);

  return result;
}

// Diagonal Traversal
// Traverse diagonally (nodes at same diagonal distance)
function diagonalTraversal(root) {
  if (root === null) return [];

  const diagonalMap = new Map();

  function traverse(node, diagonal) {
    if (node === null) return;

    if (!diagonalMap.has(diagonal)) {
      diagonalMap.set(diagonal, []);
    }
    diagonalMap.get(diagonal).push(node.val);

    // Left child increases diagonal distance
    traverse(node.left, diagonal + 1);
    // Right child maintains same diagonal distance
    traverse(node.right, diagonal);
  }

  traverse(root, 0);

  // Convert map to array
  const result = [];
  for (let i = 0; i < diagonalMap.size; i++) {
    if (diagonalMap.has(i)) {
      result.push(...diagonalMap.get(i));
    }
  }

  return result;
}

// ===== TRAVERSAL UTILITIES =====

class TraversalUtils {
  // Build tree from array (level-order)
  static buildTreeFromArray(arr) {
    if (!arr || arr.length === 0) return null;

    const root = new TreeNode(arr[0]);
    const queue = [root];
    let i = 1;

    while (queue.length > 0 && i < arr.length) {
      const node = queue.shift();
```

```
      if (i < arr.length && arr[i] !== null) {
        node.left = new TreeNode(arr[i]);
        queue.push(node.left);
      }
      i++;

      if (i < arr.length && arr[i] !== null) {
        node.right = new TreeNode(arr[i]);
        queue.push(node.right);
      }
      i++;
    }

    return root;
  }

  // Convert tree to array (level-order)
  static treeToArray(root) {
    if (root === null) return [];

    const result = [];
    const queue = [root];

    while (queue.length > 0) {
      const node = queue.shift();

      if (node === null) {
        result.push(null);
      } else {
        result.push(node.val);
        queue.push(node.left);
        queue.push(node.right);
      }
    }

    // Remove trailing nulls
    while (result.length > 0 && result[result.length - 1] === null) {
      result.pop();
    }

    return result;
  }

  // Print tree structure
  static printTree(root, prefix = "", isLast = true) {
    if (root === null) return;

    console.log(prefix + (isLast ? "└── " : "├── ") + root.val);

    const children = [root.left, root.right].filter((child) => child !== null);

    children.forEach((child, index) => {
      const isLastChild = index === children.length - 1;
      const newPrefix = prefix + (isLast ? "    " : "│   ");
```

```
      TraversalUtils.printTree(child, newPrefix, isLastChild);
    });
  }

  // Get tree height
  static getHeight(root) {
    if (root === null) return 0;
    return (
      1 +
      Math.max(
        TraversalUtils.getHeight(root.left),
        TraversalUtils.getHeight(root.right)
      )
    );
  }

  // Get tree width (maximum nodes at any level)
  static getWidth(root) {
    if (root === null) return 0;

    let maxWidth = 0;
    const queue = [root];

    while (queue.length > 0) {
      const levelSize = queue.length;
      maxWidth = Math.max(maxWidth, levelSize);

      for (let i = 0; i < levelSize; i++) {
        const node = queue.shift();

        if (node.left !== null) {
          queue.push(node.left);
        }
        if (node.right !== null) {
          queue.push(node.right);
        }
      }
    }

    return maxWidth;
  }

  // Compare all traversal methods
  static compareTraversals(root) {
    console.log("=== Tree Traversal Comparison ===");
    console.log("Inorder (Recursive):", inorderRecursive(root));
    console.log("Inorder (Iterative):", inorderIterative(root));
    console.log("Inorder (Morris):", inorderMorris(root));
    console.log("Preorder (Recursive):", preorderRecursive(root));
    console.log("Preorder (Iterative):", preorderIterative(root));
    console.log("Postorder (Recursive):", postorderRecursive(root));
    console.log("Postorder (Iterative):", postorderIterativeTwoStacks(root));
    console.log("Level-order:", levelOrder(root));
    console.log("Level-order by levels:", levelOrderByLevels(root));
```

```
    console.log("Zigzag level-order:", zigzagLevelOrder(root));
  }

  // Performance measurement
  static measureTraversalPerformance(root, traversalFunc, name) {
    const start = performance.now();
    const result = traversalFunc(root);
    const end = performance.now();

    console.log(
      `${name}: ${(end - start).toFixed(4)}ms, Nodes: ${result.length}`
    );
    return result;
  }

  // Generate random binary tree
  static generateRandomTree(maxDepth, probability = 0.7) {
    function buildRandom(depth) {
      if (depth > maxDepth || Math.random() > probability) {
        return null;
      }

      const val = Math.floor(Math.random() * 100) + 1;
      const node = new TreeNode(val);
      node.left = buildRandom(depth + 1);
      node.right = buildRandom(depth + 1);

      return node;
    }

    return buildRandom(0);
  }

  // Generate BST from sorted array
  static generateBSTFromSorted(arr) {
    function buildBST(start, end) {
      if (start > end) return null;

      const mid = Math.floor((start + end) / 2);
      const node = new TreeNode(arr[mid]);

      node.left = buildBST(start, mid - 1);
      node.right = buildBST(mid + 1, end);

      return node;
    }

    return buildBST(0, arr.length - 1);
  }
}

// ===== EXAMPLE USAGE AND TESTING =====

console.log("=== Tree Traversals Demo ===");
```

```
// Create example tree:
//       1
//      / \
//     2   3
//    / \
//   4   5
const root = new TreeNode(1);
root.left = new TreeNode(2);
root.right = new TreeNode(3);
root.left.left = new TreeNode(4);
root.left.right = new TreeNode(5);

console.log("\n=== Tree Structure ===");
TraversalUtils.printTree(root);

console.log("\n=== Basic Traversals ===");
console.log("Inorder:", inorderRecursive(root)); // [4, 2, 5, 1, 3]
console.log("Preorder:", preorderRecursive(root)); // [1, 2, 4, 5, 3]
console.log("Postorder:", postorderRecursive(root)); // [4, 5, 2, 3, 1]
console.log("Level-order:", levelOrder(root)); // [1, 2, 3, 4, 5]

console.log("\n=== Advanced Traversals ===");
console.log("Level-order by levels:", levelOrderByLevels(root));
console.log("Zigzag level-order:", zigzagLevelOrder(root));
console.log("Reverse level-order:", reverseLevelOrder(root));
console.log("Vertical order:", verticalOrder(root));
console.log("Boundary traversal:", boundaryTraversal(root));
console.log("Diagonal traversal:", diagonalTraversal(root));

// Create BST for demonstration
const bstArray = [1, 2, 3, 4, 5, 6, 7];
const bst = TraversalUtils.generateBSTFromSorted(bstArray);

console.log("\n=== BST Traversals ===");
console.log("BST Structure:");
TraversalUtils.printTree(bst);
console.log("BST Inorder (should be sorted):", inorderRecursive(bst));

// Performance comparison
console.log("\n=== Performance Comparison ===");
const largeTree = TraversalUtils.generateRandomTree(10);

TraversalUtils.measureTraversalPerformance(
  largeTree,
  inorderRecursive,
  "Inorder Recursive"
);
TraversalUtils.measureTraversalPerformance(
  largeTree,
  inorderIterative,
  "Inorder Iterative"
);
TraversalUtils.measureTraversalPerformance(
```

```
  largeTree,
  inorderMorris,
  "Inorder Morris"
);
TraversalUtils.measureTraversalPerformance(
  largeTree,
  levelOrder,
  "Level-order"
);

// Tree statistics
console.log("\n=== Tree Statistics ===");
console.log("Tree height:", TraversalUtils.getHeight(root));
console.log("Tree width:", TraversalUtils.getWidth(root));

// All traversals comparison
console.log("\n=== Complete Comparison ===");
TraversalUtils.compareTraversals(root);
```

🔧 C++ Implementation

```
#include <iostream>
#include <vector>
#include <queue>
#include <stack>
#include <map>
#include <algorithm>
#include <chrono>
using namespace std;
using namespace std::chrono;

// Tree Node Definition
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

// ===== DEPTH-FIRST SEARCH (DFS) TRAVERSALS =====

// 1. INORDER TRAVERSAL (Left → Root → Right)

// Recursive Inorder
void inorderRecursive(TreeNode* root, vector<int>& result) {
    if (root == nullptr) return;

    inorderRecursive(root->left, result);
    result.push_back(root->val);
```

```
    inorderRecursive(root->right, result);
}

vector<int> inorderRecursive(TreeNode* root) {
    vector<int> result;
    inorderRecursive(root, result);
    return result;
}

// Iterative Inorder
vector<int> inorderIterative(TreeNode* root) {
    vector<int> result;
    stack<TreeNode*> stk;
    TreeNode* current = root;

    while (current != nullptr || !stk.empty()) {
        // Go to leftmost node
        while (current != nullptr) {
            stk.push(current);
            current = current->left;
        }

        // Backtrack
        current = stk.top();
        stk.pop();
        result.push_back(current->val);

        // Visit right subtree
        current = current->right;
    }

    return result;
}

// Morris Inorder (O(1) space)
vector<int> inorderMorris(TreeNode* root) {
    vector<int> result;
    TreeNode* current = root;

    while (current != nullptr) {
        if (current->left == nullptr) {
            result.push_back(current->val);
            current = current->right;
        } else {
            // Find inorder predecessor
            TreeNode* predecessor = current->left;
            while (predecessor->right != nullptr && predecessor->right != current)
{
                predecessor = predecessor->right;
            }

            if (predecessor->right == nullptr) {
                // Make current the right child of predecessor
                predecessor->right = current;
```

```
                current = current->left;
            } else {
                // Revert changes
                predecessor->right = nullptr;
                result.push_back(current->val);
                current = current->right;
            }
        }
    }

    return result;
}

// 2. PREORDER TRAVERSAL (Root → Left → Right)

// Recursive Preorder
void preorderRecursive(TreeNode* root, vector<int>& result) {
    if (root == nullptr) return;

    result.push_back(root->val);
    preorderRecursive(root->left, result);
    preorderRecursive(root->right, result);
}

vector<int> preorderRecursive(TreeNode* root) {
    vector<int> result;
    preorderRecursive(root, result);
    return result;
}

// Iterative Preorder
vector<int> preorderIterative(TreeNode* root) {
    if (root == nullptr) return {};

    vector<int> result;
    stack<TreeNode*> stk;
    stk.push(root);

    while (!stk.empty()) {
        TreeNode* node = stk.top();
        stk.pop();
        result.push_back(node->val);

        // Push right first, then left
        if (node->right != nullptr) {
            stk.push(node->right);
        }
        if (node->left != nullptr) {
            stk.push(node->left);
        }
    }

    return result;
}
```

```cpp
// 3. POSTORDER TRAVERSAL (Left → Right → Root)

// Recursive Postorder
void postorderRecursive(TreeNode* root, vector<int>& result) {
    if (root == nullptr) return;

    postorderRecursive(root->left, result);
    postorderRecursive(root->right, result);
    result.push_back(root->val);
}

vector<int> postorderRecursive(TreeNode* root) {
    vector<int> result;
    postorderRecursive(root, result);
    return result;
}

// Iterative Postorder (Two Stacks)
vector<int> postorderIterativeTwoStacks(TreeNode* root) {
    if (root == nullptr) return {};

    vector<int> result;
    stack<TreeNode*> stk1, stk2;
    stk1.push(root);

    while (!stk1.empty()) {
        TreeNode* node = stk1.top();
        stk1.pop();
        stk2.push(node);

        if (node->left != nullptr) {
            stk1.push(node->left);
        }
        if (node->right != nullptr) {
            stk1.push(node->right);
        }
    }

    while (!stk2.empty()) {
        result.push_back(stk2.top()->val);
        stk2.pop();
    }

    return result;
}

// Iterative Postorder (One Stack)
vector<int> postorderIterativeOneStack(TreeNode* root) {
    if (root == nullptr) return {};

    vector<int> result;
    stack<TreeNode*> stk;
    TreeNode* lastVisited = nullptr;
```

```
    TreeNode* current = root;

    while (!stk.empty() || current != nullptr) {
        if (current != nullptr) {
            stk.push(current);
            current = current->left;
        } else {
            TreeNode* peekNode = stk.top();

            if (peekNode->right != nullptr && lastVisited != peekNode->right) {
                current = peekNode->right;
            } else {
                result.push_back(peekNode->val);
                lastVisited = stk.top();
                stk.pop();
            }
        }
    }

    return result;
}

// ===== BREADTH-FIRST SEARCH (BFS) TRAVERSAL =====

// Level-order Traversal
vector<int> levelOrder(TreeNode* root) {
    if (root == nullptr) return {};

    vector<int> result;
    queue<TreeNode*> q;
    q.push(root);

    while (!q.empty()) {
        TreeNode* node = q.front();
        q.pop();
        result.push_back(node->val);

        if (node->left != nullptr) {
            q.push(node->left);
        }
        if (node->right != nullptr) {
            q.push(node->right);
        }
    }

    return result;
}

// Level-order by Levels
vector<vector<int>> levelOrderByLevels(TreeNode* root) {
    if (root == nullptr) return {};

    vector<vector<int>> result;
    queue<TreeNode*> q;
```

```
    q.push(root);

    while (!q.empty()) {
        int levelSize = q.size();
        vector<int> currentLevel;

        for (int i = 0; i < levelSize; i++) {
            TreeNode* node = q.front();
            q.pop();
            currentLevel.push_back(node->val);

            if (node->left != nullptr) {
                q.push(node->left);
            }
            if (node->right != nullptr) {
                q.push(node->right);
            }
        }

        result.push_back(currentLevel);
    }

    return result;
}

// Zigzag Level-order
vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
    if (root == nullptr) return {};

    vector<vector<int>> result;
    queue<TreeNode*> q;
    q.push(root);
    bool leftToRight = true;

    while (!q.empty()) {
        int levelSize = q.size();
        vector<int> currentLevel(levelSize);

        for (int i = 0; i < levelSize; i++) {
            TreeNode* node = q.front();
            q.pop();

            int index = leftToRight ? i : levelSize - 1 - i;
            currentLevel[index] = node->val;

            if (node->left != nullptr) {
                q.push(node->left);
            }
            if (node->right != nullptr) {
                q.push(node->right);
            }
        }

        result.push_back(currentLevel);
```

```
            leftToRight = !leftToRight;
        }

        return result;
    }

    // ===== ADVANCED TRAVERSALS =====

    // Vertical Order Traversal
    vector<vector<int>> verticalOrder(TreeNode* root) {
        if (root == nullptr) return {};

        map<int, vector<int>> columnMap;
        queue<pair<TreeNode*, int>> q;
        q.push({root, 0});

        while (!q.empty()) {
            auto [node, col] = q.front();
            q.pop();

            columnMap[col].push_back(node->val);

            if (node->left != nullptr) {
                q.push({node->left, col - 1});
            }
            if (node->right != nullptr) {
                q.push({node->right, col + 1});
            }
        }

        vector<vector<int>> result;
        for (auto& [col, values] : columnMap) {
            result.push_back(values);
        }

        return result;
    }

    // Boundary Traversal
    vector<int> boundaryTraversal(TreeNode* root) {
        if (root == nullptr) return {};

        vector<int> result;
        result.push_back(root->val);

        if (root->left == nullptr && root->right == nullptr) {
            return result;
        }

        // Left boundary (excluding leaves)
        function<void(TreeNode*)> addLeftBoundary = [&](TreeNode* node) {
            if (node == nullptr || (node->left == nullptr && node->right == nullptr))
    {
                return;
```

```
        }

        result.push_back(node->val);

        if (node->left != nullptr) {
            addLeftBoundary(node->left);
        } else {
            addLeftBoundary(node->right);
        }
    };

    // Leaves
    function<void(TreeNode*)> addLeaves = [&](TreeNode* node) {
        if (node == nullptr) return;

        if (node->left == nullptr && node->right == nullptr) {
            result.push_back(node->val);
            return;
        }

        addLeaves(node->left);
        addLeaves(node->right);
    };

    // Right boundary (excluding leaves, in reverse)
    function<void(TreeNode*)> addRightBoundary = [&](TreeNode* node) {
        if (node == nullptr || (node->left == nullptr && node->right == nullptr))
{
            return;
        }

        if (node->right != nullptr) {
            addRightBoundary(node->right);
        } else {
            addRightBoundary(node->left);
        }

        result.push_back(node->val);
    };

    addLeftBoundary(root->left);
    addLeaves(root);
    addRightBoundary(root->right);

    return result;
}

// ===== UTILITY FUNCTIONS =====

class TraversalUtils {
public:
    // Build tree from array
    static TreeNode* buildTreeFromArray(const vector<int>& arr) {
        if (arr.empty()) return nullptr;
```

```
        TreeNode* root = new TreeNode(arr[0]);
        queue<TreeNode*> q;
        q.push(root);
        int i = 1;

        while (!q.empty() && i < arr.size()) {
            TreeNode* node = q.front();
            q.pop();

            if (i < arr.size()) {
                node->left = new TreeNode(arr[i]);
                q.push(node->left);
                i++;
            }

            if (i < arr.size()) {
                node->right = new TreeNode(arr[i]);
                q.push(node->right);
                i++;
            }
        }

        return root;
    }

    // Print tree structure
    static void printTree(TreeNode* root, string prefix = "", bool isLast = true)
{
        if (root == nullptr) return;

        cout << prefix << (isLast ? "└── " : "├── ") << root->val << endl;

        vector<TreeNode*> children;
        if (root->left != nullptr) children.push_back(root->left);
        if (root->right != nullptr) children.push_back(root->right);

        for (int i = 0; i < children.size(); i++) {
            bool isLastChild = (i == children.size() - 1);
            string newPrefix = prefix + (isLast ? "    " : "│   ");
            printTree(children[i], newPrefix, isLastChild);
        }
    }

    // Get tree height
    static int getHeight(TreeNode* root) {
        if (root == nullptr) return 0;
        return 1 + max(getHeight(root->left), getHeight(root->right));
    }

    // Performance measurement
    static void measurePerformance(function<vector<int>(TreeNode*)> traversalFunc,
                                   TreeNode* root, const string& name) {
        auto start = high_resolution_clock::now();
```

```
        vector<int> result = traversalFunc(root);
        auto end = high_resolution_clock::now();

        auto duration = duration_cast<microseconds>(end - start);
        cout << name << ": " << duration.count() << " microseconds, Nodes: " <<
result.size() << endl;
    }

    // Generate BST from sorted array
    static TreeNode* generateBSTFromSorted(const vector<int>& arr) {
        function<TreeNode*(int, int)> buildBST = [&](int start, int end) ->
TreeNode* {
            if (start > end) return nullptr;

            int mid = start + (end - start) / 2;
            TreeNode* node = new TreeNode(arr[mid]);

            node->left = buildBST(start, mid - 1);
            node->right = buildBST(mid + 1, end);

            return node;
        };

        return buildBST(0, arr.size() - 1);
    }
};

// Print vector utility
template<typename T>
void printVector(const vector<T>& vec, const string& label = "") {
    if (!label.empty()) {
        cout << label << ": ";
    }
    for (const auto& x : vec) {
        cout << x << " ";
    }
    cout << endl;
}

// Print 2D vector utility
void print2DVector(const vector<vector<int>>& vec, const string& label = "") {
    if (!label.empty()) {
        cout << label << ": ";
    }
    cout << "[";
    for (int i = 0; i < vec.size(); i++) {
        cout << "[";
        for (int j = 0; j < vec[i].size(); j++) {
            cout << vec[i][j];
            if (j < vec[i].size() - 1) cout << ", ";
        }
        cout << "]";
        if (i < vec.size() - 1) cout << ", ";
    }
```

```
    cout << "]" << endl;
}

// Example Usage
int main() {
    cout << "=== Tree Traversals Demo ===" << endl;

    // Create example tree:
    //       1
    //      / \
    //     2   3
    //    / \
    //   4   5
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);
    root->left->right = new TreeNode(5);

    cout << "\n=== Tree Structure ===" << endl;
    TraversalUtils::printTree(root);

    cout << "\n=== Basic Traversals ===" << endl;
    printVector(inorderRecursive(root), "Inorder");
    printVector(preorderRecursive(root), "Preorder");
    printVector(postorderRecursive(root), "Postorder");
    printVector(levelOrder(root), "Level-order");

    cout << "\n=== Advanced Traversals ===" << endl;
    print2DVector(levelOrderByLevels(root), "Level-order by levels");
    print2DVector(zigzagLevelOrder(root), "Zigzag level-order");
    print2DVector(verticalOrder(root), "Vertical order");
    printVector(boundaryTraversal(root), "Boundary traversal");

    // Create BST
    vector<int> bstArray = {1, 2, 3, 4, 5, 6, 7};
    TreeNode* bst = TraversalUtils::generateBSTFromSorted(bstArray);

    cout << "\n=== BST Traversals ===" << endl;
    cout << "BST Structure:" << endl;
    TraversalUtils::printTree(bst);
    printVector(inorderRecursive(bst), "BST Inorder (should be sorted)");

    // Performance comparison
    cout << "\n=== Performance Comparison ===" << endl;
    TraversalUtils::measurePerformance(inorderRecursive, root, "Inorder
Recursive");
    TraversalUtils::measurePerformance(inorderIterative, root, "Inorder
Iterative");
    TraversalUtils::measurePerformance(inorderMorris, root, "Inorder Morris");
    TraversalUtils::measurePerformance(levelOrder, root, "Level-order");

    cout << "\n=== Tree Statistics ===" << endl;
    cout << "Tree height: " << TraversalUtils::getHeight(root) << endl;
```

```
    return 0;
}
```

⚡ Performance Analysis

Time & Space Complexity:

Traversal	Time	Space (Recursive)	Space (Iterative)	Space (Morris)
Inorder	O(n)	O(h)	O(h)	O(1)
Preorder	O(n)	O(h)	O(h)	O(1)
Postorder	O(n)	O(h)	O(h)	O(1)
Level-order	O(n)	N/A	O(w)	N/A

h = height, w = maximum width, n = number of nodes

When to Use Each:

1. **Inorder**:
    - ✅ BST: Get sorted sequence
    - ✅ Expression trees: Infix notation
    - ✅ Binary tree validation
2. **Preorder**:
    - ✅ Tree copying/serialization
    - ✅ Prefix expressions
    - ✅ Directory listing
3. **Postorder**:
    - ✅ Tree deletion (children first)
    - ✅ Postfix expressions
    - ✅ Calculate tree properties
4. **Level-order**:
    - ✅ Tree printing by levels
    - ✅ Shortest path in unweighted trees
    - ✅ Complete tree operations

Common Pitfalls:

1. **Stack overflow**: Deep recursion in unbalanced trees
2. **Memory leaks**: Not deallocating tree nodes in C++
3. **Null pointer access**: Not checking for null nodes

4. **Incorrect Morris implementation**: Breaking tree structure permanently

---

🧩 Practice Problems

Problem 1: Binary Tree Right Side View

Question: Return values of nodes you can see from the right side.

Example: `[1,2,3,null,5,null,4]` → `[1,3,4]`

Hint: Use level-order traversal, take last node of each level.

Problem 2: Binary Tree Vertical Order Traversal

Question: Return vertical order traversal of binary tree.

Hint: Use BFS with column tracking.

Problem 3: Serialize and Deserialize Binary Tree

Question: Design algorithm to serialize/deserialize binary tree.

Hint: Use preorder traversal with null markers.

Problem 4: Binary Tree Maximum Path Sum

Question: Find maximum path sum between any two nodes.

Hint: Use postorder traversal with path sum calculation.

🎯 Interview Tips

What Interviewers Look For:

1. **Traversal mastery**: Know all four basic traversals
2. **Implementation choice**: Recursive vs iterative vs Morris
3. **Space optimization**: When to use Morris traversal
4. **Problem adaptation**: Apply traversals to solve problems

Common Interview Patterns:

- **Tree validation**: Use inorder for BST validation
- **Tree construction**: Use preorder/postorder with inorder
- **Path problems**: Use DFS traversals
- **Level problems**: Use BFS traversal
- **Tree views**: Right/left/top/bottom views

Red Flags to Avoid:

- Confusing traversal orders

- Not handling null nodes
- Stack overflow in deep trees
- Inefficient space usage when O(1) is possible

Pro Tips:

1. **Master the basics**: Know all four traversals by heart
2. **Practice iterative**: Interviewers often prefer iterative solutions
3. **Consider Morris**: For space-constrained problems
4. **Draw examples**: Visualize traversal order
5. **Handle edge cases**: Empty tree, single node, skewed tree

---

🚀 Key Takeaways

1. **Four fundamental traversals** - Each has specific use cases
2. **Recursive vs Iterative** - Trade-off between simplicity and space
3. **Morris traversal** - Achieves O(1) space complexity
4. **Level-order uses queue** - BFS pattern for tree problems
5. **BST inorder is sorted** - Key property for many algorithms
6. **Practice variations** - Zigzag, vertical, boundary traversals

Next Chapter: We'll explore Basic Graph Theory and see how traversal concepts extend to graph structures.

Chapter 10: Basic Graph Theory - Understanding Connected Data

---

🎯 What are Graphs?

Graphs are non-linear data structures consisting of vertices (nodes) connected by edges. They represent relationships between entities and are fundamental to modeling real-world problems like social networks, transportation systems, and computer networks.

Why Graphs Matter:

- **Social networks**: Friends, followers, connections
- **Transportation**: Roads, flights, routes
- **Computer networks**: Internet, LANs, routing
- **Dependencies**: Task scheduling, package management
- **Recommendation systems**: User preferences, item relationships
- **Game development**: Pathfinding, AI decision trees

Graph Components:

1. **Vertices (V)**: Nodes or points in the graph
2. **Edges (E)**: Connections between vertices
3. **Weight**: Optional value associated with edges
4. **Degree**: Number of edges connected to a vertex

📊 Graph Types & Properties

Graph Classifications:

Type	Description	Example	Use Cases
Directed	Edges have direction	Twitter follows	Web links, dependencies
Undirected	Edges are bidirectional	Facebook friends	Social networks, maps
Weighted	Edges have values	Road distances	Shortest path, cost optimization
Unweighted	All edges equal	Simple connections	Basic relationships
Cyclic	Contains cycles	Road networks	Most real-world graphs
Acyclic	No cycles	Family trees	Hierarchies, DAGs
Connected	Path exists between all vertices	Single network	Communication systems
Disconnected	Some vertices unreachable	Multiple networks	Isolated components

Special Graph Types:

- **Complete Graph**: Every vertex connected to every other vertex
- **Bipartite Graph**: Vertices can be divided into two disjoint sets
- **Tree**: Connected acyclic graph with V-1 edges
- **DAG (Directed Acyclic Graph)**: Directed graph with no cycles
- **Planar Graph**: Can be drawn without edge crossings

💻 JavaScript Implementation

```
// Graph Representations and Basic Operations

// ===== GRAPH REPRESENTATIONS =====

// 1. ADJACENCY LIST REPRESENTATION
// Most common and space-efficient for sparse graphs
class GraphAdjacencyList {
  constructor(isDirected = false) {
    this.vertices = new Map(); // vertex -> list of neighbors
    this.isDirected = isDirected;
    this.vertexCount = 0;
    this.edgeCount = 0;
  }

  // Add vertex to graph
  addVertex(vertex) {
    if (!this.vertices.has(vertex)) {
      this.vertices.set(vertex, []);
      this.vertexCount++;
```

```
      return true;
    }
    return false; // Vertex already exists
  }

  // Remove vertex and all its edges
  removeVertex(vertex) {
    if (!this.vertices.has(vertex)) {
      return false;
    }

    // Remove all edges to this vertex
    for (let [v, neighbors] of this.vertices) {
      const index = neighbors.findIndex(
        (neighbor) =>
          (typeof neighbor === "object" ? neighbor.vertex : neighbor) === vertex
      );
      if (index !== -1) {
        neighbors.splice(index, 1);
        this.edgeCount--;
      }
    }

    // Remove the vertex itself
    this.edgeCount -= this.vertices.get(vertex).length;
    this.vertices.delete(vertex);
    this.vertexCount--;
    return true;
  }

  // Add edge between two vertices
  addEdge(vertex1, vertex2, weight = 1) {
    // Ensure both vertices exist
    this.addVertex(vertex1);
    this.addVertex(vertex2);

    // Add edge from vertex1 to vertex2
    const neighbors1 = this.vertices.get(vertex1);
    if (weight === 1) {
      neighbors1.push(vertex2);
    } else {
      neighbors1.push({ vertex: vertex2, weight });
    }

    // If undirected, add reverse edge
    if (!this.isDirected) {
      const neighbors2 = this.vertices.get(vertex2);
      if (weight === 1) {
        neighbors2.push(vertex1);
      } else {
        neighbors2.push({ vertex: vertex1, weight });
      }
    }
```

```
    this.edgeCount++;
    return true;
  }

  // Remove edge between two vertices
  removeEdge(vertex1, vertex2) {
    if (!this.vertices.has(vertex1) || !this.vertices.has(vertex2)) {
      return false;
    }

    // Remove edge from vertex1 to vertex2
    const neighbors1 = this.vertices.get(vertex1);
    const index1 = neighbors1.findIndex(
      (neighbor) =>
        (typeof neighbor === "object" ? neighbor.vertex : neighbor) === vertex2
    );

    if (index1 === -1) {
      return false; // Edge doesn't exist
    }

    neighbors1.splice(index1, 1);

    // If undirected, remove reverse edge
    if (!this.isDirected) {
      const neighbors2 = this.vertices.get(vertex2);
      const index2 = neighbors2.findIndex(
        (neighbor) =>
          (typeof neighbor === "object" ? neighbor.vertex : neighbor) ===
          vertex1
      );
      if (index2 !== -1) {
        neighbors2.splice(index2, 1);
      }
    }

    this.edgeCount--;
    return true;
  }

  // Check if edge exists
  hasEdge(vertex1, vertex2) {
    if (!this.vertices.has(vertex1)) {
      return false;
    }

    const neighbors = this.vertices.get(vertex1);
    return neighbors.some(
      (neighbor) =>
        (typeof neighbor === "object" ? neighbor.vertex : neighbor) === vertex2
    );
  }

  // Get neighbors of a vertex
```

```
    getNeighbors(vertex) {
      return this.vertices.get(vertex) || [];
    }

    // Get all vertices
    getVertices() {
      return Array.from(this.vertices.keys());
    }

    // Get vertex degree
    getDegree(vertex) {
      if (!this.vertices.has(vertex)) {
        return 0;
      }
      return this.vertices.get(vertex).length;
    }

    // Print graph
    printGraph() {
      console.log(`Graph (${this.isDirected ? "Directed" : "Undirected"}):`);
      for (let [vertex, neighbors] of this.vertices) {
        const neighborStr = neighbors
          .map((neighbor) =>
            typeof neighbor === "object"
              ? `${neighbor.vertex}(${neighbor.weight})`
              : neighbor
          )
          .join(", ");
        console.log(`${vertex} -> [${neighborStr}]`);
      }
      console.log(`Vertices: ${this.vertexCount}, Edges: ${this.edgeCount}`);
    }
  }

  // 2. ADJACENCY MATRIX REPRESENTATION
  // Good for dense graphs and when you need O(1) edge lookup
  class GraphAdjacencyMatrix {
    constructor(maxVertices = 100, isDirected = false) {
      this.maxVertices = maxVertices;
      this.isDirected = isDirected;
      this.matrix = Array(maxVertices)
        .fill(null)
        .map(() => Array(maxVertices).fill(0));
      this.vertexMap = new Map(); // vertex -> index
      this.indexMap = new Map(); // index -> vertex
      this.vertexCount = 0;
      this.edgeCount = 0;
    }

    // Add vertex
    addVertex(vertex) {
      if (this.vertexMap.has(vertex) || this.vertexCount >= this.maxVertices) {
        return false;
      }
```

```
    const index = this.vertexCount;
    this.vertexMap.set(vertex, index);
    this.indexMap.set(index, vertex);
    this.vertexCount++;
    return true;
  }

  // Add edge
  addEdge(vertex1, vertex2, weight = 1) {
    if (!this.vertexMap.has(vertex1) || !this.vertexMap.has(vertex2)) {
      return false;
    }

    const index1 = this.vertexMap.get(vertex1);
    const index2 = this.vertexMap.get(vertex2);

    // Add edge
    if (this.matrix[index1][index2] === 0) {
      this.edgeCount++;
    }
    this.matrix[index1][index2] = weight;

    // If undirected, add reverse edge
    if (!this.isDirected) {
      this.matrix[index2][index1] = weight;
    }

    return true;
  }

  // Remove edge
  removeEdge(vertex1, vertex2) {
    if (!this.vertexMap.has(vertex1) || !this.vertexMap.has(vertex2)) {
      return false;
    }

    const index1 = this.vertexMap.get(vertex1);
    const index2 = this.vertexMap.get(vertex2);

    if (this.matrix[index1][index2] !== 0) {
      this.matrix[index1][index2] = 0;
      if (!this.isDirected) {
        this.matrix[index2][index1] = 0;
      }
      this.edgeCount--;
      return true;
    }

    return false;
  }

  // Check if edge exists
  hasEdge(vertex1, vertex2) {
```

```
    if (!this.vertexMap.has(vertex1) || !this.vertexMap.has(vertex2)) {
      return false;
    }

    const index1 = this.vertexMap.get(vertex1);
    const index2 = this.vertexMap.get(vertex2);
    return this.matrix[index1][index2] !== 0;
  }

  // Get neighbors
  getNeighbors(vertex) {
    if (!this.vertexMap.has(vertex)) {
      return [];
    }

    const index = this.vertexMap.get(vertex);
    const neighbors = [];

    for (let i = 0; i < this.vertexCount; i++) {
      if (this.matrix[index][i] !== 0) {
        const neighborVertex = this.indexMap.get(i);
        const weight = this.matrix[index][i];
        if (weight === 1) {
          neighbors.push(neighborVertex);
        } else {
          neighbors.push({ vertex: neighborVertex, weight });
        }
      }
    }

    return neighbors;
  }

  // Print matrix
  printMatrix() {
    console.log("Adjacency Matrix:");
    const vertices = Array.from(this.indexMap.values());

    // Print header
    console.log("   ", vertices.join("  "));

    // Print rows
    for (let i = 0; i < this.vertexCount; i++) {
      const row = this.matrix[i].slice(0, this.vertexCount);
      console.log(`${vertices[i]}:  ${row.join("  ")}`);
    }
  }
}

// ===== GRAPH TRAVERSAL ALGORITHMS =====

// 1. DEPTH-FIRST SEARCH (DFS)
// Explores as far as possible before backtracking
class GraphDFS {
```

```
  // Recursive DFS
  static dfsRecursive(graph, startVertex, visited = new Set(), result = []) {
    visited.add(startVertex);
    result.push(startVertex);

    const neighbors = graph.getNeighbors(startVertex);
    for (let neighbor of neighbors) {
      const vertex = typeof neighbor === "object" ? neighbor.vertex : neighbor;
      if (!visited.has(vertex)) {
        GraphDFS.dfsRecursive(graph, vertex, visited, result);
      }
    }

    return result;
  }

  // Iterative DFS using stack
  static dfsIterative(graph, startVertex) {
    const visited = new Set();
    const result = [];
    const stack = [startVertex];

    while (stack.length > 0) {
      const vertex = stack.pop();

      if (!visited.has(vertex)) {
        visited.add(vertex);
        result.push(vertex);

        // Add neighbors to stack (in reverse order for consistent traversal)
        const neighbors = graph.getNeighbors(vertex);
        for (let i = neighbors.length - 1; i >= 0; i--) {
          const neighbor = neighbors[i];
          const neighborVertex =
            typeof neighbor === "object" ? neighbor.vertex : neighbor;
          if (!visited.has(neighborVertex)) {
            stack.push(neighborVertex);
          }
        }
      }
    }

    return result;
  }

  // DFS to find path between two vertices
  static findPath(graph, start, end, visited = new Set(), path = []) {
    visited.add(start);
    path.push(start);

    if (start === end) {
      return [...path]; // Return copy of path
    }
```

```
    const neighbors = graph.getNeighbors(start);
    for (let neighbor of neighbors) {
      const vertex = typeof neighbor === "object" ? neighbor.vertex : neighbor;
      if (!visited.has(vertex)) {
        const result = GraphDFS.findPath(graph, vertex, end, visited, path);
        if (result) {
          return result;
        }
      }
    }

    path.pop(); // Backtrack
    return null;
  }

  // DFS to find all paths between two vertices
  static findAllPaths(
    graph,
    start,
    end,
    visited = new Set(),
    path = [],
    allPaths = []
  ) {
    visited.add(start);
    path.push(start);

    if (start === end) {
      allPaths.push([...path]);
    } else {
      const neighbors = graph.getNeighbors(start);
      for (let neighbor of neighbors) {
        const vertex =
          typeof neighbor === "object" ? neighbor.vertex : neighbor;
        if (!visited.has(vertex)) {
          GraphDFS.findAllPaths(graph, vertex, end, visited, path, allPaths);
        }
      }
    }

    path.pop();
    visited.delete(start);
    return allPaths;
  }

  // Check if graph is connected (for undirected graphs)
  static isConnected(graph) {
    const vertices = graph.getVertices();
    if (vertices.length === 0) return true;

    const visited = GraphDFS.dfsRecursive(graph, vertices[0]);
    return visited.length === vertices.length;
  }
```

```
  // Detect cycle in undirected graph
  static hasCycleUndirected(graph) {
    const visited = new Set();
    const vertices = graph.getVertices();

    function dfsCheckCycle(vertex, parent) {
      visited.add(vertex);

      const neighbors = graph.getNeighbors(vertex);
      for (let neighbor of neighbors) {
        const neighborVertex =
          typeof neighbor === "object" ? neighbor.vertex : neighbor;

        if (!visited.has(neighborVertex)) {
          if (dfsCheckCycle(neighborVertex, vertex)) {
            return true;
          }
        } else if (neighborVertex !== parent) {
          return true; // Back edge found
        }
      }

      return false;
    }

    for (let vertex of vertices) {
      if (!visited.has(vertex)) {
        if (dfsCheckCycle(vertex, null)) {
          return true;
        }
      }
    }

    return false;
  }

  // Detect cycle in directed graph
  static hasCycleDirected(graph) {
    const visited = new Set();
    const recursionStack = new Set();
    const vertices = graph.getVertices();

    function dfsCheckCycle(vertex) {
      visited.add(vertex);
      recursionStack.add(vertex);

      const neighbors = graph.getNeighbors(vertex);
      for (let neighbor of neighbors) {
        const neighborVertex =
          typeof neighbor === "object" ? neighbor.vertex : neighbor;

        if (!visited.has(neighborVertex)) {
          if (dfsCheckCycle(neighborVertex)) {
            return true;
```

```
            }
          } else if (recursionStack.has(neighborVertex)) {
            return true; // Back edge in recursion stack
          }
        }

        recursionStack.delete(vertex);
        return false;
      }

      for (let vertex of vertices) {
        if (!visited.has(vertex)) {
          if (dfsCheckCycle(vertex)) {
            return true;
          }
        }
      }

      return false;
    }
  }

  // 2. BREADTH-FIRST SEARCH (BFS)
  // Explores neighbors level by level
  class GraphBFS {
    // Basic BFS traversal
    static bfs(graph, startVertex) {
      const visited = new Set();
      const result = [];
      const queue = [startVertex];

      visited.add(startVertex);

      while (queue.length > 0) {
        const vertex = queue.shift();
        result.push(vertex);

        const neighbors = graph.getNeighbors(vertex);
        for (let neighbor of neighbors) {
          const neighborVertex =
            typeof neighbor === "object" ? neighbor.vertex : neighbor;
          if (!visited.has(neighborVertex)) {
            visited.add(neighborVertex);
            queue.push(neighborVertex);
          }
        }
      }

      return result;
    }

    // BFS to find shortest path (unweighted graph)
    static shortestPath(graph, start, end) {
      if (start === end) {
```

```
        return [start];
      }

      const visited = new Set();
      const queue = [[start]];
      visited.add(start);

      while (queue.length > 0) {
        const path = queue.shift();
        const vertex = path[path.length - 1];

        const neighbors = graph.getNeighbors(vertex);
        for (let neighbor of neighbors) {
          const neighborVertex =
            typeof neighbor === "object" ? neighbor.vertex : neighbor;

          if (neighborVertex === end) {
            return [...path, neighborVertex];
          }

          if (!visited.has(neighborVertex)) {
            visited.add(neighborVertex);
            queue.push([...path, neighborVertex]);
          }
        }
      }

      return null; // No path found
    }

    // BFS to find shortest distance
    static shortestDistance(graph, start, end) {
      if (start === end) {
        return 0;
      }

      const visited = new Set();
      const queue = [{ vertex: start, distance: 0 }];
      visited.add(start);

      while (queue.length > 0) {
        const { vertex, distance } = queue.shift();

        const neighbors = graph.getNeighbors(vertex);
        for (let neighbor of neighbors) {
          const neighborVertex =
            typeof neighbor === "object" ? neighbor.vertex : neighbor;

          if (neighborVertex === end) {
            return distance + 1;
          }

          if (!visited.has(neighborVertex)) {
            visited.add(neighborVertex);
```

```
            queue.push({ vertex: neighborVertex, distance: distance + 1 });
          }
        }
      }

      return -1; // No path found
    }

    // BFS level-order traversal
    static levelOrder(graph, startVertex) {
      const visited = new Set();
      const result = [];
      const queue = [{ vertex: startVertex, level: 0 }];
      visited.add(startVertex);

      while (queue.length > 0) {
        const { vertex, level } = queue.shift();

        // Ensure result array has enough levels
        while (result.length <= level) {
          result.push([]);
        }

        result[level].push(vertex);

        const neighbors = graph.getNeighbors(vertex);
        for (let neighbor of neighbors) {
          const neighborVertex =
            typeof neighbor === "object" ? neighbor.vertex : neighbor;
          if (!visited.has(neighborVertex)) {
            visited.add(neighborVertex);
            queue.push({ vertex: neighborVertex, level: level + 1 });
          }
        }
      }

      return result;
    }

    // Check if graph is bipartite
    static isBipartite(graph) {
      const color = new Map();
      const vertices = graph.getVertices();

      for (let startVertex of vertices) {
        if (color.has(startVertex)) continue;

        const queue = [startVertex];
        color.set(startVertex, 0);

        while (queue.length > 0) {
          const vertex = queue.shift();
          const currentColor = color.get(vertex);
```

```
          const neighbors = graph.getNeighbors(vertex);
          for (let neighbor of neighbors) {
            const neighborVertex =
              typeof neighbor === "object" ? neighbor.vertex : neighbor;

            if (!color.has(neighborVertex)) {
              color.set(neighborVertex, 1 - currentColor);
              queue.push(neighborVertex);
            } else if (color.get(neighborVertex) === currentColor) {
              return false; // Same color adjacent vertices
            }
          }
        }
      }

      return true;
    }
  }

  // ===== GRAPH UTILITIES =====

  class GraphUtils {
    // Create graph from edge list
    static fromEdgeList(edges, isDirected = false) {
      const graph = new GraphAdjacencyList(isDirected);

      for (let edge of edges) {
        if (edge.length === 2) {
          graph.addEdge(edge[0], edge[1]);
        } else if (edge.length === 3) {
          graph.addEdge(edge[0], edge[1], edge[2]);
        }
      }

      return graph;
    }

    // Convert to edge list
    static toEdgeList(graph) {
      const edges = [];
      const vertices = graph.getVertices();
      const visited = new Set();

      for (let vertex of vertices) {
        const neighbors = graph.getNeighbors(vertex);
        for (let neighbor of neighbors) {
          const neighborVertex =
            typeof neighbor === "object" ? neighbor.vertex : neighbor;
          const weight = typeof neighbor === "object" ? neighbor.weight : 1;

          const edgeKey = graph.isDirected
            ? `${vertex}->${neighborVertex}`
            : [vertex, neighborVertex].sort().join("-");
```

```
        if (!visited.has(edgeKey)) {
          visited.add(edgeKey);
          if (weight === 1) {
            edges.push([vertex, neighborVertex]);
          } else {
            edges.push([vertex, neighborVertex, weight]);
          }
        }
      }
    }

    return edges;
  }

  // Find connected components
  static findConnectedComponents(graph) {
    const visited = new Set();
    const components = [];
    const vertices = graph.getVertices();

    for (let vertex of vertices) {
      if (!visited.has(vertex)) {
        const component = GraphDFS.dfsRecursive(graph, vertex, visited);
        components.push(component);
      }
    }

    return components;
  }

  // Calculate graph density
  static calculateDensity(graph) {
    const V = graph.vertexCount;
    const E = graph.edgeCount;

    if (V <= 1) return 0;

    const maxEdges = graph.isDirected ? V * (V - 1) : (V * (V - 1)) / 2;
    return E / maxEdges;
  }

  // Generate random graph
  static generateRandomGraph(
    numVertices,
    edgeProbability = 0.3,
    isDirected = false
  ) {
    const graph = new GraphAdjacencyList(isDirected);

    // Add vertices
    for (let i = 0; i < numVertices; i++) {
      graph.addVertex(i);
    }
```

```
    // Add random edges
    for (let i = 0; i < numVertices; i++) {
      for (let j = isDirected ? 0 : i + 1; j < numVertices; j++) {
        if (i !== j && Math.random() < edgeProbability) {
          graph.addEdge(i, j);
        }
      }
    }

    return graph;
  }

  // Performance comparison
  static compareRepresentations(numVertices, numEdges) {
    console.log("=== Graph Representation Comparison ===");

    // Create same graph in both representations
    const adjList = new GraphAdjacencyList();
    const adjMatrix = new GraphAdjacencyMatrix(numVertices);

    // Add vertices
    for (let i = 0; i < numVertices; i++) {
      adjList.addVertex(i);
      adjMatrix.addVertex(i);
    }

    // Add random edges
    const edges = [];
    for (let i = 0; i < numEdges; i++) {
      const v1 = Math.floor(Math.random() * numVertices);
      const v2 = Math.floor(Math.random() * numVertices);
      if (v1 !== v2) {
        edges.push([v1, v2]);
      }
    }

    // Measure edge addition time
    console.time("Adjacency List - Add Edges");
    for (let [v1, v2] of edges) {
      adjList.addEdge(v1, v2);
    }
    console.timeEnd("Adjacency List - Add Edges");

    console.time("Adjacency Matrix - Add Edges");
    for (let [v1, v2] of edges) {
      adjMatrix.addEdge(v1, v2);
    }
    console.timeEnd("Adjacency Matrix - Add Edges");

    // Measure edge lookup time
    const testEdge = edges[0];

    console.time("Adjacency List - Edge Lookup");
    for (let i = 0; i < 1000; i++) {
```

```
      adjList.hasEdge(testEdge[0], testEdge[1]);
    }
    console.timeEnd("Adjacency List - Edge Lookup");

    console.time("Adjacency Matrix - Edge Lookup");
    for (let i = 0; i < 1000; i++) {
      adjMatrix.hasEdge(testEdge[0], testEdge[1]);
    }
    console.timeEnd("Adjacency Matrix - Edge Lookup");

    // Space usage estimation
    const listSpace = numVertices + 2 * numEdges; // Rough estimate
    const matrixSpace = numVertices * numVertices;

    console.log(`\nSpace Usage Estimation:`);
    console.log(`Adjacency List: ~${listSpace} units`);
    console.log(`Adjacency Matrix: ~${matrixSpace} units`);
    console.log(
      `Density: ${(
        (numEdges / ((numVertices * (numVertices - 1)) / 2)) *
        100
      ).toFixed(2)}%`
    );
  }
}

// ===== EXAMPLE USAGE AND TESTING =====

console.log("=== Basic Graph Theory Demo ===");

// Create undirected graph
const graph = new GraphAdjacencyList(false);

// Add vertices
["A", "B", "C", "D", "E"].forEach((v) => graph.addVertex(v));

// Add edges
graph.addEdge("A", "B");
graph.addEdge("A", "C");
graph.addEdge("B", "D");
graph.addEdge("C", "D");
graph.addEdge("D", "E");

console.log("\n=== Graph Structure ===");
graph.printGraph();

console.log("\n=== Graph Traversals ===");
console.log("DFS (Recursive):", GraphDFS.dfsRecursive(graph, "A"));
console.log("DFS (Iterative):", GraphDFS.dfsIterative(graph, "A"));
console.log("BFS:", GraphBFS.bfs(graph, "A"));
console.log("BFS Level Order:", GraphBFS.levelOrder(graph, "A"));

console.log("\n=== Path Finding ===");
console.log("Path A to E:", GraphDFS.findPath(graph, "A", "E"));
```

```
console.log("All paths A to E:", GraphDFS.findAllPaths(graph, "A", "E"));
console.log("Shortest path A to E:", GraphBFS.shortestPath(graph, "A", "E"));
console.log(
  "Shortest distance A to E:",
  GraphBFS.shortestDistance(graph, "A", "E")
);

console.log("\n=== Graph Properties ===");
console.log("Is connected:", GraphDFS.isConnected(graph));
console.log("Has cycle:", GraphDFS.hasCycleUndirected(graph));
console.log("Is bipartite:", GraphBFS.isBipartite(graph));
console.log("Connected components:", GraphUtils.findConnectedComponents(graph));
console.log("Graph density:", GraphUtils.calculateDensity(graph).toFixed(3));

// Create directed graph for cycle detection
const directedGraph = new GraphAdjacencyList(true);
["X", "Y", "Z"].forEach((v) => directedGraph.addVertex(v));
directedGraph.addEdge("X", "Y");
directedGraph.addEdge("Y", "Z");
directedGraph.addEdge("Z", "X"); // Creates cycle

console.log("\n=== Directed Graph ===");
directedGraph.printGraph();
console.log("Has cycle (directed):", GraphDFS.hasCycleDirected(directedGraph));

// Weighted graph example
const weightedGraph = new GraphAdjacencyList(false);
["P", "Q", "R", "S"].forEach((v) => weightedGraph.addVertex(v));
weightedGraph.addEdge("P", "Q", 5);
weightedGraph.addEdge("P", "R", 3);
weightedGraph.addEdge("Q", "S", 2);
weightedGraph.addEdge("R", "S", 7);

console.log("\n=== Weighted Graph ===");
weightedGraph.printGraph();

// Adjacency matrix example
const matrixGraph = new GraphAdjacencyMatrix(5, false);
[0, 1, 2, 3, 4].forEach((v) => matrixGraph.addVertex(v));
matrixGraph.addEdge(0, 1);
matrixGraph.addEdge(0, 2);
matrixGraph.addEdge(1, 3);
matrixGraph.addEdge(2, 3);
matrixGraph.addEdge(3, 4);

console.log("\n=== Adjacency Matrix ===");
matrixGraph.printMatrix();

// Edge list conversion
console.log("\n=== Edge List Conversion ===");
const edgeList = GraphUtils.toEdgeList(graph);
console.log("Edge list:", edgeList);

const graphFromEdges = GraphUtils.fromEdgeList(edgeList);
```

```
console.log("Graph from edge list:");
graphFromEdges.printGraph();

// Performance comparison
console.log("\n=== Performance Comparison ===");
GraphUtils.compareRepresentations(100, 200);

// Random graph generation
console.log("\n=== Random Graph ===");
const randomGraph = GraphUtils.generateRandomGraph(6, 0.4);
randomGraph.printGraph();
console.log("Random graph properties:");
console.log(
  "- Connected components:",
  GraphUtils.findConnectedComponents(randomGraph).length
);
console.log("- Density:", GraphUtils.calculateDensity(randomGraph).toFixed(3));
console.log("- Has cycle:", GraphDFS.hasCycleUndirected(randomGraph));
```

🔧 C++ Implementation

```
#include <iostream>
#include <vector>
#include <list>
#include <queue>
#include <stack>
#include <unordered_set>
#include <unordered_map>
#include <algorithm>
#include <chrono>
using namespace std;
using namespace std::chrono;

// ===== ADJACENCY LIST REPRESENTATION =====

class GraphAdjacencyList {
private:
    unordered_map<int, list<pair<int, int>>> adjList; // vertex -> list of
(neighbor, weight)
    bool isDirected;
    int vertexCount;
    int edgeCount;

public:
    GraphAdjacencyList(bool directed = false) : isDirected(directed),
vertexCount(0), edgeCount(0) {}

    // Add vertex
    bool addVertex(int vertex) {
        if (adjList.find(vertex) == adjList.end()) {
```

```
            adjList[vertex] = list<pair<int, int>>();
            vertexCount++;
            return true;
        }
        return false;
    }

    // Add edge
    bool addEdge(int vertex1, int vertex2, int weight = 1) {
        addVertex(vertex1);
        addVertex(vertex2);

        adjList[vertex1].push_back({vertex2, weight});

        if (!isDirected) {
            adjList[vertex2].push_back({vertex1, weight});
        }

        edgeCount++;
        return true;
    }

    // Remove edge
    bool removeEdge(int vertex1, int vertex2) {
        if (adjList.find(vertex1) == adjList.end() || adjList.find(vertex2) ==
adjList.end()) {
            return false;
        }

        auto& neighbors1 = adjList[vertex1];
        auto it1 = find_if(neighbors1.begin(), neighbors1.end(),
                           [vertex2](const pair<int, int>& p) { return p.first ==
vertex2; });

        if (it1 == neighbors1.end()) {
            return false;
        }

        neighbors1.erase(it1);

        if (!isDirected) {
            auto& neighbors2 = adjList[vertex2];
            auto it2 = find_if(neighbors2.begin(), neighbors2.end(),
                               [vertex1](const pair<int, int>& p) { return p.first
== vertex1; });
            if (it2 != neighbors2.end()) {
                neighbors2.erase(it2);
            }
        }

        edgeCount--;
        return true;
    }
```

```
    // Check if edge exists
    bool hasEdge(int vertex1, int vertex2) {
        if (adjList.find(vertex1) == adjList.end()) {
            return false;
        }

        const auto& neighbors = adjList[vertex1];
        return find_if(neighbors.begin(), neighbors.end(),
                       [vertex2](const pair<int, int>& p) { return p.first ==
vertex2; }) != neighbors.end();
    }

    // Get neighbors
    vector<pair<int, int>> getNeighbors(int vertex) {
        if (adjList.find(vertex) == adjList.end()) {
            return {};
        }

        vector<pair<int, int>> neighbors;
        for (const auto& neighbor : adjList[vertex]) {
            neighbors.push_back(neighbor);
        }
        return neighbors;
    }

    // Get all vertices
    vector<int> getVertices() {
        vector<int> vertices;
        for (const auto& pair : adjList) {
            vertices.push_back(pair.first);
        }
        return vertices;
    }

    // Print graph
    void printGraph() {
        cout << "Graph (" << (isDirected ? "Directed" : "Undirected") << "):" <<
endl;
        for (const auto& vertex : adjList) {
            cout << vertex.first << " -> [";
            bool first = true;
            for (const auto& neighbor : vertex.second) {
                if (!first) cout << ", ";
                if (neighbor.second == 1) {
                    cout << neighbor.first;
                } else {
                    cout << neighbor.first << "(" << neighbor.second << ")";
                }
                first = false;
            }
            cout << "]" << endl;
        }
        cout << "Vertices: " << vertexCount << ", Edges: " << edgeCount << endl;
    }
```

```
    // Getters
    int getVertexCount() const { return vertexCount; }
    int getEdgeCount() const { return edgeCount; }
    bool getIsDirected() const { return isDirected; }
};

// ===== ADJACENCY MATRIX REPRESENTATION =====

class GraphAdjacencyMatrix {
private:
    vector<vector<int>> matrix;
    unordered_map<int, int> vertexToIndex;
    unordered_map<int, int> indexToVertex;
    bool isDirected;
    int maxVertices;
    int vertexCount;
    int edgeCount;

public:
    GraphAdjacencyMatrix(int maxVert = 100, bool directed = false)
        : maxVertices(maxVert), isDirected(directed), vertexCount(0), edgeCount(0)
{
        matrix.resize(maxVertices, vector<int>(maxVertices, 0));
    }

    // Add vertex
    bool addVertex(int vertex) {
        if (vertexToIndex.find(vertex) != vertexToIndex.end() || vertexCount >=
maxVertices) {
            return false;
        }

        int index = vertexCount;
        vertexToIndex[vertex] = index;
        indexToVertex[index] = vertex;
        vertexCount++;
        return true;
    }

    // Add edge
    bool addEdge(int vertex1, int vertex2, int weight = 1) {
        if (vertexToIndex.find(vertex1) == vertexToIndex.end() ||
            vertexToIndex.find(vertex2) == vertexToIndex.end()) {
            return false;
        }

        int index1 = vertexToIndex[vertex1];
        int index2 = vertexToIndex[vertex2];

        if (matrix[index1][index2] == 0) {
            edgeCount++;
        }
```

```
        matrix[index1][index2] = weight;

        if (!isDirected) {
            matrix[index2][index1] = weight;
        }

        return true;
    }

    // Check if edge exists
    bool hasEdge(int vertex1, int vertex2) {
        if (vertexToIndex.find(vertex1) == vertexToIndex.end() ||
            vertexToIndex.find(vertex2) == vertexToIndex.end()) {
            return false;
        }

        int index1 = vertexToIndex[vertex1];
        int index2 = vertexToIndex[vertex2];
        return matrix[index1][index2] != 0;
    }

    // Print matrix
    void printMatrix() {
        cout << "Adjacency Matrix:" << endl;

        // Print header
        cout << "   ";
        for (int i = 0; i < vertexCount; i++) {
            cout << indexToVertex[i] << "  ";
        }
        cout << endl;

        // Print rows
        for (int i = 0; i < vertexCount; i++) {
            cout << indexToVertex[i] << ": ";
            for (int j = 0; j < vertexCount; j++) {
                cout << matrix[i][j] << "  ";
            }
            cout << endl;
        }
    }
};

// ===== GRAPH TRAVERSAL ALGORITHMS =====

// DFS Implementation
class GraphDFS {
public:
    // Recursive DFS
    static vector<int> dfsRecursive(GraphAdjacencyList& graph, int startVertex) {
        unordered_set<int> visited;
        vector<int> result;
        dfsRecursiveHelper(graph, startVertex, visited, result);
        return result;
```

```
    }

private:
    static void dfsRecursiveHelper(GraphAdjacencyList& graph, int vertex,
                                   unordered_set<int>& visited, vector<int>&
result) {
        visited.insert(vertex);
        result.push_back(vertex);

        auto neighbors = graph.getNeighbors(vertex);
        for (const auto& neighbor : neighbors) {
            if (visited.find(neighbor.first) == visited.end()) {
                dfsRecursiveHelper(graph, neighbor.first, visited, result);
            }
        }
    }

public:
    // Iterative DFS
    static vector<int> dfsIterative(GraphAdjacencyList& graph, int startVertex) {
        unordered_set<int> visited;
        vector<int> result;
        stack<int> stk;

        stk.push(startVertex);

        while (!stk.empty()) {
            int vertex = stk.top();
            stk.pop();

            if (visited.find(vertex) == visited.end()) {
                visited.insert(vertex);
                result.push_back(vertex);

                auto neighbors = graph.getNeighbors(vertex);
                for (auto it = neighbors.rbegin(); it != neighbors.rend(); ++it) {
                    if (visited.find(it->first) == visited.end()) {
                        stk.push(it->first);
                    }
                }
            }
        }

        return result;
    }

    // Find path between two vertices
    static vector<int> findPath(GraphAdjacencyList& graph, int start, int end) {
        unordered_set<int> visited;
        vector<int> path;

        if (findPathHelper(graph, start, end, visited, path)) {
            return path;
        }
```

```
        return {}; // No path found
    }

private:
    static bool findPathHelper(GraphAdjacencyList& graph, int current, int end,
                               unordered_set<int>& visited, vector<int>& path) {
        visited.insert(current);
        path.push_back(current);

        if (current == end) {
            return true;
        }

        auto neighbors = graph.getNeighbors(current);
        for (const auto& neighbor : neighbors) {
            if (visited.find(neighbor.first) == visited.end()) {
                if (findPathHelper(graph, neighbor.first, end, visited, path)) {
                    return true;
                }
            }
        }

        path.pop_back(); // Backtrack
        return false;
    }

public:
    // Check if graph has cycle (undirected)
    static bool hasCycleUndirected(GraphAdjacencyList& graph) {
        unordered_set<int> visited;
        auto vertices = graph.getVertices();

        for (int vertex : vertices) {
            if (visited.find(vertex) == visited.end()) {
                if (hasCycleUndirectedHelper(graph, vertex, -1, visited)) {
                    return true;
                }
            }
        }

        return false;
    }

private:
    static bool hasCycleUndirectedHelper(GraphAdjacencyList& graph, int vertex,
int parent,
                                         unordered_set<int>& visited) {
        visited.insert(vertex);

        auto neighbors = graph.getNeighbors(vertex);
        for (const auto& neighbor : neighbors) {
            if (visited.find(neighbor.first) == visited.end()) {
                if (hasCycleUndirectedHelper(graph, neighbor.first, vertex,
```

```
visited)) {
                    return true;
                }
            } else if (neighbor.first != parent) {
                return true; // Back edge found
            }
        }

        return false;
    }
};

// BFS Implementation
class GraphBFS {
public:
    // Basic BFS
    static vector<int> bfs(GraphAdjacencyList& graph, int startVertex) {
        unordered_set<int> visited;
        vector<int> result;
        queue<int> q;

        visited.insert(startVertex);
        q.push(startVertex);

        while (!q.empty()) {
            int vertex = q.front();
            q.pop();
            result.push_back(vertex);

            auto neighbors = graph.getNeighbors(vertex);
            for (const auto& neighbor : neighbors) {
                if (visited.find(neighbor.first) == visited.end()) {
                    visited.insert(neighbor.first);
                    q.push(neighbor.first);
                }
            }
        }

        return result;
    }

    // Find shortest path (unweighted)
    static vector<int> shortestPath(GraphAdjacencyList& graph, int start, int end)
{
        if (start == end) {
            return {start};
        }

        unordered_set<int> visited;
        queue<vector<int>> q;

        visited.insert(start);
        q.push({start});
```

```
        while (!q.empty()) {
            vector<int> path = q.front();
            q.pop();

            int vertex = path.back();
            auto neighbors = graph.getNeighbors(vertex);

            for (const auto& neighbor : neighbors) {
                if (neighbor.first == end) {
                    path.push_back(neighbor.first);
                    return path;
                }

                if (visited.find(neighbor.first) == visited.end()) {
                    visited.insert(neighbor.first);
                    vector<int> newPath = path;
                    newPath.push_back(neighbor.first);
                    q.push(newPath);
                }
            }
        }

        return {}; // No path found
    }

    // Find shortest distance
    static int shortestDistance(GraphAdjacencyList& graph, int start, int end) {
        if (start == end) {
            return 0;
        }

        unordered_set<int> visited;
        queue<pair<int, int>> q; // vertex, distance

        visited.insert(start);
        q.push({start, 0});

        while (!q.empty()) {
            auto [vertex, distance] = q.front();
            q.pop();

            auto neighbors = graph.getNeighbors(vertex);
            for (const auto& neighbor : neighbors) {
                if (neighbor.first == end) {
                    return distance + 1;
                }

                if (visited.find(neighbor.first) == visited.end()) {
                    visited.insert(neighbor.first);
                    q.push({neighbor.first, distance + 1});
                }
            }
        }
```

```
        return -1; // No path found
    }
};

// Utility functions
void printVector(const vector<int>& vec, const string& label = "") {
    if (!label.empty()) {
        cout << label << ": ";
    }
    for (int x : vec) {
        cout << x << " ";
    }
    cout << endl;
}

// Example Usage
int main() {
    cout << "=== Basic Graph Theory Demo ===" << endl;

    // Create undirected graph
    GraphAdjacencyList graph(false);

    // Add vertices and edges
    for (int i = 1; i <= 5; i++) {
        graph.addVertex(i);
    }

    graph.addEdge(1, 2);
    graph.addEdge(1, 3);
    graph.addEdge(2, 4);
    graph.addEdge(3, 4);
    graph.addEdge(4, 5);

    cout << "\n=== Graph Structure ===" << endl;
    graph.printGraph();

    cout << "\n=== Graph Traversals ===" << endl;
    printVector(GraphDFS::dfsRecursive(graph, 1), "DFS (Recursive)");
    printVector(GraphDFS::dfsIterative(graph, 1), "DFS (Iterative)");
    printVector(GraphBFS::bfs(graph, 1), "BFS");

    cout << "\n=== Path Finding ===" << endl;
    printVector(GraphDFS::findPath(graph, 1, 5), "Path 1 to 5");
    printVector(GraphBFS::shortestPath(graph, 1, 5), "Shortest path 1 to 5");
    cout << "Shortest distance 1 to 5: " << GraphBFS::shortestDistance(graph, 1,
5) << endl;

    cout << "\n=== Graph Properties ===" << endl;
    cout << "Has cycle: " << (GraphDFS::hasCycleUndirected(graph) ? "Yes" : "No")
<< endl;

    // Adjacency matrix example
    cout << "\n=== Adjacency Matrix ===" << endl;
    GraphAdjacencyMatrix matrixGraph(10, false);
```

```
    for (int i = 0; i < 4; i++) {
        matrixGraph.addVertex(i);
    }

    matrixGraph.addEdge(0, 1);
    matrixGraph.addEdge(0, 2);
    matrixGraph.addEdge(1, 3);
    matrixGraph.addEdge(2, 3);

    matrixGraph.printMatrix();

    return 0;
}
```

⚡ Performance Analysis

Representation Comparison:

Operation	Adjacency List	Adjacency Matrix
Add Vertex	O(1)	O(1)
Add Edge	O(1)	O(1)
Remove Edge	O(V)	O(1)
Check Edge	O(V)	O(1)
Get Neighbors	O(degree)	O(V)
Space	O(V + E)	$O(V^2)$

When to Use Each:

Adjacency List:

- ✅ Sparse graphs ($E << V^2$)
- ✅ Memory-constrained environments
- ✅ Frequent traversals
- ❌ Frequent edge lookups

Adjacency Matrix:

- ✅ Dense graphs ($E \approx V^2$)
- ✅ Frequent edge lookups
- ✅ Simple implementation
- ❌ Memory-constrained environments

Traversal Complexity:

Algorithm	Time	Space	Use Case
DFS	O(V + E)	O(V)	Path finding, cycle detection
BFS	O(V + E)	O(V)	Shortest path, level-order

🧩 Practice Problems

Problem 1: Number of Islands

Question: Count number of islands in 2D grid (1=land, 0=water).

Example:

```
11110
11010
11000
00000
```

Output: 1

Hint: Use DFS/BFS to explore connected components.

Problem 2: Clone Graph

Question: Deep clone an undirected graph.

Hint: Use DFS/BFS with hash map to track cloned nodes.

Problem 3: Course Schedule

Question: Determine if you can finish all courses given prerequisites.

Hint: Model as directed graph, detect cycles.

Problem 4: Word Ladder

Question: Find shortest transformation sequence from start to end word.

Hint: Model as graph where words are vertices, edges connect words differing by one character.

🎯 Interview Tips

What Interviewers Look For:

1. **Representation choice**: Can you choose appropriate representation?
2. **Traversal mastery**: Know DFS and BFS implementations
3. **Problem modeling**: Can you model real problems as graphs?
4. **Complexity analysis**: Understand time/space trade-offs

Common Interview Patterns:

- **Connected components**: Use DFS/BFS to find groups
- **Cycle detection**: DFS with recursion stack (directed) or parent tracking (undirected)
- **Shortest path**: BFS for unweighted, Dijkstra for weighted
- **Topological sort**: DFS-based ordering for DAGs
- **Bipartite checking**: BFS with 2-coloring

Red Flags to Avoid:

- Confusing directed vs undirected graphs
- Not handling disconnected components
- Infinite loops in traversal
- Wrong complexity analysis

Pro Tips:

1. **Clarify graph type**: Directed? Weighted? Connected?
2. **Choose representation wisely**: Consider space/time trade-offs
3. **Handle edge cases**: Empty graph, single vertex, disconnected components
4. **Practice both traversals**: DFS and BFS have different use cases
5. **Think in terms of problems**: Many problems are graph problems in disguise

---

🚀 Key Takeaways

1. **Graphs model relationships** - Fundamental for many real-world problems
2. **Two main representations** - Adjacency list vs matrix trade-offs
3. **DFS goes deep** - Good for path finding and cycle detection
4. **BFS goes wide** - Good for shortest paths and level-order processing
5. **Many problems are graphs** - Social networks, dependencies, maps
6. **Practice recognition** - Learn to identify graph problems

Next Chapter: We'll explore Two Pointers technique and see how it optimizes array and string problems.

Chapter 11: Two Pointers - Efficient Array & String Processing

---

🎯 What is the Two Pointers Technique?

Two Pointers is an algorithmic pattern that uses two pointers to traverse data structures, typically arrays or strings, to solve problems more efficiently than brute force approaches. Instead of nested loops ($O(n^2)$), we often achieve O(n) time complexity.

Why Two Pointers Matter:

- **Optimization**: Reduces time complexity from $O(n^2)$ to O(n)
- **Space efficiency**: Usually O(1) extra space

- **Versatile**: Works on arrays, strings, linked lists
- **Interview favorite**: Common in coding interviews
- **Real applications**: Data processing, algorithms, system design

Core Concept:

Use two pointers that move through the data structure according to specific rules to find solutions without examining all possible pairs.

📊 Two Pointers Patterns

Pattern Classifications:

Pattern	Description	Movement	Use Cases
Opposite Direction	Start from ends, move toward center	`left++`, `right--`	Palindromes, Two Sum (sorted)
Same Direction	Both start from beginning	`slow++`, `fast++`	Remove duplicates, sliding window
Fast & Slow	Different speeds	`slow++`, `fast += 2`	Cycle detection, middle element
Sliding Window	Maintain window size	Expand/contract window	Subarray problems

When to Use Two Pointers:

- ✅ **Sorted arrays**: Take advantage of ordering
- ✅ **Palindrome problems**: Check from both ends
- ✅ **Pair/triplet finding**: Avoid nested loops
- ✅ **Subarray problems**: Maintain window efficiently
- ✅ **Linked list problems**: Fast/slow pointer techniques

💻 JavaScript Implementation

```
// Two Pointers Technique - Comprehensive Implementation

// ===== PATTERN 1: OPPOSITE DIRECTION POINTERS =====

/**
 * Two Sum - Find pair that sums to target (sorted array)
 * Time: O(n), Space: O(1)
 */
function twoSumSorted(nums, target) {
    let left = 0;
    let right = nums.length - 1;
```

```
    while (left < right) {
        const sum = nums[left] + nums[right];

        if (sum === target) {
            return [left, right]; // Return indices
        } else if (sum < target) {
            left++; // Need larger sum
        } else {
            right--; // Need smaller sum
        }
    }

    return [-1, -1]; // No solution found
}

/**
 * Valid Palindrome - Check if string is palindrome
 * Time: O(n), Space: O(1)
 */
function isPalindrome(s) {
    // Clean string: remove non-alphanumeric, convert to lowercase
    const cleaned = s.replace(/[^a-zA-Z0-9]/g, '').toLowerCase();

    let left = 0;
    let right = cleaned.length - 1;

    while (left < right) {
        if (cleaned[left] !== cleaned[right]) {
            return false;
        }
        left++;
        right--;
    }

    return true;
}

/**
 * Three Sum - Find all unique triplets that sum to zero
 * Time: O(n²), Space: O(1) excluding output
 */
function threeSum(nums) {
    const result = [];
    nums.sort((a, b) => a - b); // Sort array first

    for (let i = 0; i < nums.length - 2; i++) {
        // Skip duplicates for first element
        if (i > 0 && nums[i] === nums[i - 1]) {
            continue;
        }

        let left = i + 1;
        let right = nums.length - 1;
        const target = -nums[i]; // We want nums[i] + nums[left] + nums[right] = 0
```

```
        while (left < right) {
            const sum = nums[left] + nums[right];

            if (sum === target) {
                result.push([nums[i], nums[left], nums[right]]);

                // Skip duplicates for second element
                while (left < right && nums[left] === nums[left + 1]) {
                    left++;
                }
                // Skip duplicates for third element
                while (left < right && nums[right] === nums[right - 1]) {
                    right--;
                }

                left++;
                right--;
            } else if (sum < target) {
                left++;
            } else {
                right--;
            }
        }
    }

    return result;
}

/**
 * Container With Most Water - Find maximum area
 * Time: O(n), Space: O(1)
 */
function maxArea(height) {
    let left = 0;
    let right = height.length - 1;
    let maxWater = 0;

    while (left < right) {
        // Calculate current area
        const width = right - left;
        const currentHeight = Math.min(height[left], height[right]);
        const area = width * currentHeight;

        maxWater = Math.max(maxWater, area);

        // Move pointer with smaller height
        if (height[left] < height[right]) {
            left++;
        } else {
            right--;
        }
    }
```

```
    return maxWater;
}

/**
 * Reverse Array In-Place
 * Time: O(n), Space: O(1)
 */
function reverseArray(arr) {
    let left = 0;
    let right = arr.length - 1;

    while (left < right) {
        // Swap elements
        [arr[left], arr[right]] = [arr[right], arr[left]];
        left++;
        right--;
    }

    return arr;
}

// ===== PATTERN 2: SAME DIRECTION POINTERS =====

/**
 * Remove Duplicates from Sorted Array
 * Time: O(n), Space: O(1)
 */
function removeDuplicates(nums) {
    if (nums.length <= 1) return nums.length;

    let writeIndex = 1; // Position to write next unique element

    for (let readIndex = 1; readIndex < nums.length; readIndex++) {
        if (nums[readIndex] !== nums[readIndex - 1]) {
            nums[writeIndex] = nums[readIndex];
            writeIndex++;
        }
    }

    return writeIndex; // New length
}

/**
 * Remove Element - Remove all instances of val
 * Time: O(n), Space: O(1)
 */
function removeElement(nums, val) {
    let writeIndex = 0;

    for (let readIndex = 0; readIndex < nums.length; readIndex++) {
        if (nums[readIndex] !== val) {
            nums[writeIndex] = nums[readIndex];
            writeIndex++;
        }
```

```
    }

    return writeIndex;
}

/**
 * Move Zeros - Move all zeros to end
 * Time: O(n), Space: O(1)
 */
function moveZeroes(nums) {
    let writeIndex = 0; // Position for next non-zero element

    // Move all non-zero elements to front
    for (let readIndex = 0; readIndex < nums.length; readIndex++) {
        if (nums[readIndex] !== 0) {
            nums[writeIndex] = nums[readIndex];
            writeIndex++;
        }
    }

    // Fill remaining positions with zeros
    while (writeIndex < nums.length) {
        nums[writeIndex] = 0;
        writeIndex++;
    }

    return nums;
}

/**
 * Partition Array - Rearrange so elements < x come before elements >= x
 * Time: O(n), Space: O(1)
 */
function partition(nums, x) {
    let writeIndex = 0;

    // First pass: move elements < x to front
    for (let readIndex = 0; readIndex < nums.length; readIndex++) {
        if (nums[readIndex] < x) {
            [nums[writeIndex], nums[readIndex]] = [nums[readIndex],
nums[writeIndex]];
            writeIndex++;
        }
    }

    return nums;
}

// ===== PATTERN 3: FAST & SLOW POINTERS =====

/**
 * Find Middle of Array/Linked List
 * Time: O(n), Space: O(1)
 */
```

```javascript
function findMiddle(arr) {
    let slow = 0;
    let fast = 0;

    // Fast pointer moves 2 steps, slow moves 1 step
    while (fast < arr.length - 1 && fast < arr.length - 2) {
        slow++;
        fast += 2;
    }

    return arr[slow];
}

/**
 * Detect Cycle in Array (using indices as next pointers)
 * Time: O(n), Space: O(1)
 */
function hasCycle(nums) {
    if (nums.length <= 1) return false;

    let slow = 0;
    let fast = 0;

    // Phase 1: Detect if cycle exists
    do {
        slow = Math.abs(nums[slow]) % nums.length;
        fast = Math.abs(nums[Math.abs(nums[fast]) % nums.length]) % nums.length;
    } while (slow !== fast);

    // Phase 2: Find cycle start (if needed)
    slow = 0;
    while (slow !== fast) {
        slow = Math.abs(nums[slow]) % nums.length;
        fast = Math.abs(nums[fast]) % nums.length;
    }

    return true; // Cycle detected
}

// ===== PATTERN 4: SLIDING WINDOW WITH TWO POINTERS =====

/**
 * Longest Substring Without Repeating Characters
 * Time: O(n), Space: O(min(m,n)) where m is charset size
 */
function lengthOfLongestSubstring(s) {
    const charSet = new Set();
    let left = 0;
    let maxLength = 0;

    for (let right = 0; right < s.length; right++) {
        // Shrink window until no duplicates
        while (charSet.has(s[right])) {
            charSet.delete(s[left]);
```

```
            left++;
        }

        charSet.add(s[right]);
        maxLength = Math.max(maxLength, right - left + 1);
    }

    return maxLength;
}

/**
 * Minimum Window Substring
 * Time: O(|s| + |t|), Space: O(|s| + |t|)
 */
function minWindow(s, t) {
    if (s.length < t.length) return "";

    // Count characters in t
    const targetCount = new Map();
    for (let char of t) {
        targetCount.set(char, (targetCount.get(char) || 0) + 1);
    }

    let left = 0;
    let minLength = Infinity;
    let minStart = 0;
    let required = targetCount.size;
    let formed = 0;
    const windowCount = new Map();

    for (let right = 0; right < s.length; right++) {
        const char = s[right];
        windowCount.set(char, (windowCount.get(char) || 0) + 1);

        if (targetCount.has(char) && windowCount.get(char) ===
targetCount.get(char)) {
            formed++;
        }

        // Try to shrink window
        while (left <= right && formed === required) {
            if (right - left + 1 < minLength) {
                minLength = right - left + 1;
                minStart = left;
            }

            const leftChar = s[left];
            windowCount.set(leftChar, windowCount.get(leftChar) - 1);

            if (targetCount.has(leftChar) && windowCount.get(leftChar) <
targetCount.get(leftChar)) {
                formed--;
            }
```

```
            left++;
        }
    }

    return minLength === Infinity ? "" : s.substring(minStart, minStart +
minLength);
}

/**
 * Subarray Sum Equals K (for positive numbers)
 * Time: O(n), Space: O(1)
 */
function subarraySum(nums, k) {
    let left = 0;
    let currentSum = 0;
    let count = 0;

    for (let right = 0; right < nums.length; right++) {
        currentSum += nums[right];

        // Shrink window if sum exceeds k
        while (currentSum > k && left <= right) {
            currentSum -= nums[left];
            left++;
        }

        if (currentSum === k) {
            count++;
        }
    }

    return count;
}

// ===== ADVANCED TWO POINTERS TECHNIQUES =====

/**
 * Four Sum - Find all unique quadruplets that sum to target
 * Time: O(n³), Space: O(1) excluding output
 */
function fourSum(nums, target) {
    const result = [];
    nums.sort((a, b) => a - b);

    for (let i = 0; i < nums.length - 3; i++) {
        if (i > 0 && nums[i] === nums[i - 1]) continue;

        for (let j = i + 1; j < nums.length - 2; j++) {
            if (j > i + 1 && nums[j] === nums[j - 1]) continue;

            let left = j + 1;
            let right = nums.length - 1;

            while (left < right) {
```

```
                const sum = nums[i] + nums[j] + nums[left] + nums[right];

                if (sum === target) {
                    result.push([nums[i], nums[j], nums[left], nums[right]]);

                    while (left < right && nums[left] === nums[left + 1]) left++;
                    while (left < right && nums[right] === nums[right - 1]) right-
-;

                    left++;
                    right--;
                } else if (sum < target) {
                    left++;
                } else {
                    right--;
                }
            }
        }
    }

    return result;
}

/**
 * Trapping Rain Water
 * Time: O(n), Space: O(1)
 */
function trap(height) {
    if (height.length <= 2) return 0;

    let left = 0;
    let right = height.length - 1;
    let leftMax = 0;
    let rightMax = 0;
    let water = 0;

    while (left < right) {
        if (height[left] < height[right]) {
            if (height[left] >= leftMax) {
                leftMax = height[left];
            } else {
                water += leftMax - height[left];
            }
            left++;
        } else {
            if (height[right] >= rightMax) {
                rightMax = height[right];
            } else {
                water += rightMax - height[right];
            }
            right--;
        }
    }
```

```
    return water;
}

/**
 * Sort Colors (Dutch National Flag)
 * Time: O(n), Space: O(1)
 */
function sortColors(nums) {
    let left = 0;    // Boundary for 0s
    let current = 0; // Current element
    let right = nums.length - 1; // Boundary for 2s

    while (current <= right) {
        if (nums[current] === 0) {
            [nums[left], nums[current]] = [nums[current], nums[left]];
            left++;
            current++;
        } else if (nums[current] === 2) {
            [nums[current], nums[right]] = [nums[right], nums[current]];
            right--;
            // Don't increment current as we need to check swapped element
        } else {
            current++; // nums[current] === 1
        }
    }

    return nums;
}

// ===== UTILITY FUNCTIONS =====

/**
 * Two Pointers Template for Custom Problems
 */
function twoPointersTemplate(arr, condition) {
    let left = 0;
    let right = arr.length - 1;

    while (left < right) {
        if (condition(arr[left], arr[right])) {
            // Found solution or move both pointers
            return [left, right];
        } else if (/* need to increase something */) {
            left++;
        } else {
            right--;
        }
    }

    return null; // No solution found
}

/**
 * Performance Testing
```

```
 */
function performanceTest() {
    console.log('=== Two Pointers Performance Test ===');

    // Generate test data
    const sizes = [1000, 10000, 100000];

    sizes.forEach(size => {
        const arr = Array.from({length: size}, (_, i) => i);
        const target = size - 1;

        console.time(`Two Sum (size: ${size})`);
        twoSumSorted(arr, target);
        console.timeEnd(`Two Sum (size: ${size})`);

        console.time(`Remove Duplicates (size: ${size})`);
        removeDuplicates([...arr, ...arr]); // Create duplicates
        console.timeEnd(`Remove Duplicates (size: ${size})`);
    });
}

// ===== EXAMPLE USAGE AND TESTING =====

console.log('=== Two Pointers Technique Demo ===');

// Test Opposite Direction Pattern
console.log('\n=== Opposite Direction Pattern ===');
const sortedArray = [2, 7, 11, 15];
console.log('Two Sum:', twoSumSorted(sortedArray, 9)); // [0, 1]
console.log('Is Palindrome "racecar":', isPalindrome('racecar')); // true
console.log('Is Palindrome "race a car":', isPalindrome('race a car')); // false
console.log('Three Sum:', threeSum([-1, 0, 1, 2, -1, -4])); // [[-1,-1,2],
[-1,0,1]]

const heights = [1, 8, 6, 2, 5, 4, 8, 3, 7];
console.log('Max Area:', maxArea(heights)); // 49

const testArray = [1, 2, 3, 4, 5];
console.log('Original:', testArray);
console.log('Reversed:', reverseArray([...testArray])); // [5, 4, 3, 2, 1]

// Test Same Direction Pattern
console.log('\n=== Same Direction Pattern ===');
const duplicatesArray = [1, 1, 2, 2, 3, 4, 4, 5];
console.log('Remove Duplicates:', removeDuplicates([...duplicatesArray])); // 5

const elementArray = [3, 2, 2, 3, 4, 5];
console.log('Remove Element 3:', removeElement([...elementArray], 3)); // 4

const zerosArray = [0, 1, 0, 3, 12];
console.log('Move Zeros:', moveZeroes([...zerosArray])); // [1, 3, 12, 0, 0]

const partitionArray = [3, 1, 4, 1, 5, 9, 2, 6];
console.log('Partition around 5:', partition([...partitionArray], 5)); // [3, 1,
```

```
4, 1, 2, 9, 5, 6]

// Test Fast & Slow Pattern
console.log('\n=== Fast & Slow Pattern ===');
const middleArray = [1, 2, 3, 4, 5, 6, 7];
console.log('Find Middle:', findMiddle(middleArray)); // 4

// Test Sliding Window Pattern
console.log('\n=== Sliding Window Pattern ===');
console.log('Longest Substring:', lengthOfLongestSubstring('abcabcbb')); // 3
console.log('Min Window:', minWindow('ADOBECODEBANC', 'ABC')); // 'BANC'

const sumArray = [1, 2, 3, 4, 5];
console.log('Subarray Sum (k=5):', subarraySum(sumArray, 5)); // 2

// Test Advanced Techniques
console.log('\n=== Advanced Techniques ===');
const fourSumArray = [1, 0, -1, 0, -2, 2];
console.log('Four Sum (target=0):', fourSum(fourSumArray, 0));

const rainArray = [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1];
console.log('Trapped Rain Water:', trap(rainArray)); // 6

const colorsArray = [2, 0, 2, 1, 1, 0];
console.log('Sort Colors:', sortColors([...colorsArray])); // [0, 0, 1, 1, 2, 2]

// Performance test
performanceTest();

// Pattern Recognition Examples
console.log('\n=== Pattern Recognition ===');
console.log('When to use each pattern:');
console.log('1. Opposite Direction: Sorted arrays, palindromes, pair finding');
console.log('2. Same Direction: Remove elements, partitioning');
console.log('3. Fast & Slow: Cycle detection, finding middle');
console.log('4. Sliding Window: Substring problems, subarray sums');
```

🔧 C++ Implementation

```
#include <iostream>
#include <vector>
#include <string>
#include <unordered_set>
#include <unordered_map>
#include <algorithm>
#include <climits>
using namespace std;

// ===== OPPOSITE DIRECTION POINTERS =====
```

```cpp
// Two Sum in Sorted Array
vector<int> twoSumSorted(vector<int>& nums, int target) {
    int left = 0, right = nums.size() - 1;

    while (left < right) {
        int sum = nums[left] + nums[right];

        if (sum == target) {
            return {left, right};
        } else if (sum < target) {
            left++;
        } else {
            right--;
        }
    }

    return {-1, -1};
}

// Valid Palindrome
bool isPalindrome(string s) {
    int left = 0, right = s.length() - 1;

    while (left < right) {
        // Skip non-alphanumeric characters
        while (left < right && !isalnum(s[left])) left++;
        while (left < right && !isalnum(s[right])) right--;

        if (tolower(s[left]) != tolower(s[right])) {
            return false;
        }

        left++;
        right--;
    }

    return true;
}

// Three Sum
vector<vector<int>> threeSum(vector<int>& nums) {
    vector<vector<int>> result;
    sort(nums.begin(), nums.end());

    for (int i = 0; i < nums.size() - 2; i++) {
        if (i > 0 && nums[i] == nums[i - 1]) continue;

        int left = i + 1, right = nums.size() - 1;
        int target = -nums[i];

        while (left < right) {
            int sum = nums[left] + nums[right];

            if (sum == target) {
```

```
                result.push_back({nums[i], nums[left], nums[right]});

                while (left < right && nums[left] == nums[left + 1]) left++;
                while (left < right && nums[right] == nums[right - 1]) right--;

                left++;
                right--;
            } else if (sum < target) {
                left++;
            } else {
                right--;
            }
        }
    }

    return result;
}

// Container With Most Water
int maxArea(vector<int>& height) {
    int left = 0, right = height.size() - 1;
    int maxWater = 0;

    while (left < right) {
        int width = right - left;
        int currentHeight = min(height[left], height[right]);
        int area = width * currentHeight;

        maxWater = max(maxWater, area);

        if (height[left] < height[right]) {
            left++;
        } else {
            right--;
        }
    }

    return maxWater;
}

// ===== SAME DIRECTION POINTERS =====

// Remove Duplicates from Sorted Array
int removeDuplicates(vector<int>& nums) {
    if (nums.size() <= 1) return nums.size();

    int writeIndex = 1;

    for (int readIndex = 1; readIndex < nums.size(); readIndex++) {
        if (nums[readIndex] != nums[readIndex - 1]) {
            nums[writeIndex] = nums[readIndex];
            writeIndex++;
        }
    }
```

```
    return writeIndex;
}

// Remove Element
int removeElement(vector<int>& nums, int val) {
    int writeIndex = 0;

    for (int readIndex = 0; readIndex < nums.size(); readIndex++) {
        if (nums[readIndex] != val) {
            nums[writeIndex] = nums[readIndex];
            writeIndex++;
        }
    }

    return writeIndex;
}

// Move Zeros
void moveZeroes(vector<int>& nums) {
    int writeIndex = 0;

    // Move non-zero elements to front
    for (int readIndex = 0; readIndex < nums.size(); readIndex++) {
        if (nums[readIndex] != 0) {
            nums[writeIndex] = nums[readIndex];
            writeIndex++;
        }
    }

    // Fill remaining with zeros
    while (writeIndex < nums.size()) {
        nums[writeIndex] = 0;
        writeIndex++;
    }
}

// ===== FAST & SLOW POINTERS =====

// Find Middle Element
int findMiddle(vector<int>& arr) {
    int slow = 0, fast = 0;

    while (fast < arr.size() - 1 && fast < arr.size() - 2) {
        slow++;
        fast += 2;
    }

    return arr[slow];
}

// ===== SLIDING WINDOW =====

// Longest Substring Without Repeating Characters
```

```
int lengthOfLongestSubstring(string s) {
    unordered_set<char> charSet;
    int left = 0, maxLength = 0;

    for (int right = 0; right < s.length(); right++) {
        while (charSet.count(s[right])) {
            charSet.erase(s[left]);
            left++;
        }

        charSet.insert(s[right]);
        maxLength = max(maxLength, right - left + 1);
    }

    return maxLength;
}

// Minimum Window Substring
string minWindow(string s, string t) {
    if (s.length() < t.length()) return "";

    unordered_map<char, int> targetCount;
    for (char c : t) {
        targetCount[c]++;
    }

    int left = 0, minLength = INT_MAX, minStart = 0;
    int required = targetCount.size(), formed = 0;
    unordered_map<char, int> windowCount;

    for (int right = 0; right < s.length(); right++) {
        char c = s[right];
        windowCount[c]++;

        if (targetCount.count(c) && windowCount[c] == targetCount[c]) {
            formed++;
        }

        while (left <= right && formed == required) {
            if (right - left + 1 < minLength) {
                minLength = right - left + 1;
                minStart = left;
            }

            char leftChar = s[left];
            windowCount[leftChar]--;

            if (targetCount.count(leftChar) && windowCount[leftChar] <
targetCount[leftChar]) {
                formed--;
            }

            left++;
        }
```

```
    }

    return minLength == INT_MAX ? "" : s.substr(minStart, minLength);
}

// ===== ADVANCED TECHNIQUES =====

// Trapping Rain Water
int trap(vector<int>& height) {
    if (height.size() <= 2) return 0;

    int left = 0, right = height.size() - 1;
    int leftMax = 0, rightMax = 0, water = 0;

    while (left < right) {
        if (height[left] < height[right]) {
            if (height[left] >= leftMax) {
                leftMax = height[left];
            } else {
                water += leftMax - height[left];
            }
            left++;
        } else {
            if (height[right] >= rightMax) {
                rightMax = height[right];
            } else {
                water += rightMax - height[right];
            }
            right--;
        }
    }

    return water;
}

// Sort Colors (Dutch National Flag)
void sortColors(vector<int>& nums) {
    int left = 0, current = 0, right = nums.size() - 1;

    while (current <= right) {
        if (nums[current] == 0) {
            swap(nums[left], nums[current]);
            left++;
            current++;
        } else if (nums[current] == 2) {
            swap(nums[current], nums[right]);
            right--;
            // Don't increment current
        } else {
            current++;
        }
    }
}
```

```cpp
// ===== UTILITY FUNCTIONS =====

template<typename T>
void printVector(const vector<T>& vec, const string& label = "") {
    if (!label.empty()) {
        cout << label << ": ";
    }
    for (const auto& x : vec) {
        cout << x << " ";
    }
    cout << endl;
}

void printMatrix(const vector<vector<int>>& matrix, const string& label = "") {
    if (!label.empty()) {
        cout << label << ":" << endl;
    }
    for (const auto& row : matrix) {
        cout << "[";
        for (int i = 0; i < row.size(); i++) {
            cout << row[i];
            if (i < row.size() - 1) cout << ", ";
        }
        cout << "]" << endl;
    }
}

// Example Usage
int main() {
    cout << "=== Two Pointers Technique Demo ===" << endl;

    // Test Opposite Direction
    cout << "\n=== Opposite Direction Pattern ===" << endl;
    vector<int> sortedArray = {2, 7, 11, 15};
    auto twoSumResult = twoSumSorted(sortedArray, 9);
    printVector(twoSumResult, "Two Sum");

    cout << "Is Palindrome 'racecar': " << (isPalindrome("racecar") ? "true" :
"false") << endl;
    cout << "Is Palindrome 'race a car': " << (isPalindrome("race a car") ? "true"
: "false") << endl;

    vector<int> threeSumArray = {-1, 0, 1, 2, -1, -4};
    auto threeSumResult = threeSum(threeSumArray);
    printMatrix(threeSumResult, "Three Sum");

    vector<int> heights = {1, 8, 6, 2, 5, 4, 8, 3, 7};
    cout << "Max Area: " << maxArea(heights) << endl;

    // Test Same Direction
    cout << "\n=== Same Direction Pattern ===" << endl;
    vector<int> duplicatesArray = {1, 1, 2, 2, 3, 4, 4, 5};
    cout << "Remove Duplicates: " << removeDuplicates(duplicatesArray) << endl;
```

```
    vector<int> elementArray = {3, 2, 2, 3, 4, 5};
    cout << "Remove Element 3: " << removeElement(elementArray, 3) << endl;

    vector<int> zerosArray = {0, 1, 0, 3, 12};
    moveZeroes(zerosArray);
    printVector(zerosArray, "Move Zeros");

    // Test Fast & Slow
    cout << "\n=== Fast & Slow Pattern ===" << endl;
    vector<int> middleArray = {1, 2, 3, 4, 5, 6, 7};
    cout << "Find Middle: " << findMiddle(middleArray) << endl;

    // Test Sliding Window
    cout << "\n=== Sliding Window Pattern ===" << endl;
    cout << "Longest Substring: " << lengthOfLongestSubstring("abcabcbb") << endl;
    cout << "Min Window: " << minWindow("ADOBECODEBANC", "ABC") << endl;

    // Test Advanced
    cout << "\n=== Advanced Techniques ===" << endl;
    vector<int> rainArray = {0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1};
    cout << "Trapped Rain Water: " << trap(rainArray) << endl;

    vector<int> colorsArray = {2, 0, 2, 1, 1, 0};
    sortColors(colorsArray);
    printVector(colorsArray, "Sort Colors");

    return 0;
}
```

⚡ Performance Analysis

Time Complexity Improvements:

Problem	Brute Force	Two Pointers	Improvement
Two Sum (sorted)	$O(n^2)$	O(n)	n times faster
Three Sum	$O(n^3)$	$O(n^2)$	n times faster
Palindrome Check	O(n)	O(n)	Same, but O(1) space
Remove Duplicates	$O(n^2)$	O(n)	n times faster
Container Water	$O(n^2)$	O(n)	n times faster

Space Complexity:

- Most two-pointer solutions use **O(1) extra space**
- Sliding window may use **O(k)** where k is window size
- Much better than hash-based solutions that use **O(n) space**

When Two Pointers Wins:

- ✅ **Sorted arrays**: Take advantage of ordering
- ✅ **Pair/triplet problems**: Avoid nested loops
- ✅ **In-place operations**: Minimize space usage
- ✅ **Optimization problems**: Find optimal solutions efficiently

---

🧩 Practice Problems

Problem 1: Squares of Sorted Array

Question: Given sorted array, return squares in sorted order. **Example**: `[-4,-1,0,3,10]` → `[0,1,9,16,100]` **Hint**: Use two pointers from ends, compare absolute values.

Problem 2: Intersection of Two Arrays

Question: Find intersection of two sorted arrays. **Example**: `[1,2,2,1]`, `[2,2]` → `[2]` **Hint**: Use two pointers, advance smaller element.

Problem 3: Merge Sorted Array

Question: Merge two sorted arrays in-place. **Example**: `[1,2,3,0,0,0]`, `[2,5,6]` → `[1,2,2,3,5,6]` **Hint**: Start from the end to avoid overwriting.

Problem 4: Backspace String Compare

Question: Compare strings with backspaces (`#`). **Example**: `"ab#c"`, `"ad#c"` → `true` **Hint**: Process from end using two pointers.

🎯 Interview Tips

What Interviewers Look For:

1. **Pattern recognition**: Can you identify when to use two pointers?
2. **Pointer movement logic**: Do you move pointers correctly?
3. **Edge case handling**: Empty arrays, single elements, no solution
4. **Optimization thinking**: Can you improve from brute force?

Common Interview Patterns:

- **Sorted array problems**: Almost always consider two pointers
- **Palindrome problems**: Start from ends
- **Sum problems**: Use sorting + two pointers
- **Sliding window**: Expand/contract window with two pointers
- **In-place operations**: Use read/write pointers

Red Flags to Avoid:

- Moving pointers incorrectly (infinite loops)
- Not handling duplicates properly
- Forgetting edge cases (empty input, single element)

- Using extra space when O(1) is possible

Pro Tips:

1. **Draw it out**: Visualize pointer movements
2. **Start simple**: Get basic case working first
3. **Handle duplicates**: Often requires special logic
4. **Check boundaries**: Ensure pointers don't go out of bounds
5. **Consider sorting**: Sometimes preprocessing helps
6. **Think about invariants**: What should be true at each step?

🚀 Key Takeaways

1. **Two pointers optimize nested loops** - Reduce $O(n^2)$ to O(n)
2. **Four main patterns** - Opposite, same direction, fast/slow, sliding window
3. **Sorted arrays are key** - Ordering enables efficient pointer movement
4. **Space efficient** - Usually O(1) extra space
5. **Interview favorite** - Master the patterns for coding interviews
6. **Think before coding** - Identify the pattern first

Next Chapter: We'll explore Sliding Window technique in detail and see how it optimizes subarray and substring problems.

Chapter 12: Sliding Window - Optimizing Subarray & Substring Problems

---

🎯 What is the Sliding Window Technique?

Sliding Window is an algorithmic pattern that maintains a "window" (subarray/substring) that slides through the data structure to solve problems efficiently. Instead of checking all possible subarrays ($O(n^2)$ or $O(n^3)$), we maintain a window and adjust its size dynamically to achieve O(n) time complexity.

Why Sliding Window Matters:

- **Optimization**: Reduces time complexity from $O(n^2/n^3)$ to O(n)
- **Space efficiency**: Usually O(1) or O(k) extra space
- **Versatile**: Works on arrays, strings, linked lists
- **Real applications**: Data streaming, network protocols, analytics
- **Interview favorite**: Common pattern in coding interviews

Core Concept:

Maintain a window with two pointers (left and right) and expand/contract the window based on problem constraints while tracking the optimal solution.

📊 Sliding Window Patterns

Pattern Classifications:

Pattern	Window Size	Expansion/Contraction	Use Cases
Fixed Size	Constant	Slide by 1 position	Max sum of k elements
Variable Size	Dynamic	Expand/contract based on condition	Longest substring
Shrinkable	Grows then shrinks	Expand until invalid, then shrink	Min window substring
Non-shrinkable	Only grows	Expand and slide	Longest valid window

When to Use Sliding Window:

- ✅ **Contiguous subarray/substring problems**
- ✅ **Optimization problems** (min/max/longest/shortest)
- ✅ **Problems with constraints** (sum, distinct characters, etc.)
- ✅ **Streaming data** where you process elements sequentially

---

💻 JavaScript Implementation

```
// Sliding Window Technique - Comprehensive Implementation

// ===== PATTERN 1: FIXED SIZE SLIDING WINDOW =====

/**
 * Maximum Sum of Subarray of Size K
 * Time: O(n), Space: O(1)
 */
function maxSumSubarray(arr, k) {
  if (arr.length < k) return -1;

  // Calculate sum of first window
  let windowSum = 0;
  for (let i = 0; i < k; i++) {
    windowSum += arr[i];
  }

  let maxSum = windowSum;

  // Slide the window
  for (let i = k; i < arr.length; i++) {
    windowSum = windowSum - arr[i - k] + arr[i];
    maxSum = Math.max(maxSum, windowSum);
  }

  return maxSum;
}

/**
 * Average of All Subarrays of Size K
```

```
 * Time: O(n), Space: O(n-k+1)
 */
function findAverages(arr, k) {
  if (arr.length < k) return [];

  const result = [];
  let windowSum = 0;

  // Calculate sum of first window
  for (let i = 0; i < k; i++) {
    windowSum += arr[i];
  }
  result.push(windowSum / k);

  // Slide the window
  for (let i = k; i < arr.length; i++) {
    windowSum = windowSum - arr[i - k] + arr[i];
    result.push(windowSum / k);
  }

  return result;
}

/**
 * Maximum of All Subarrays of Size K
 * Time: O(n), Space: O(k) using deque
 */
function maxSlidingWindow(nums, k) {
  const result = [];
  const deque = []; // Store indices

  for (let i = 0; i < nums.length; i++) {
    // Remove indices outside current window
    while (deque.length > 0 && deque[0] <= i - k) {
      deque.shift();
    }

    // Remove indices of smaller elements
    while (deque.length > 0 && nums[deque[deque.length - 1]] <= nums[i]) {
      deque.pop();
    }

    deque.push(i);

    // Add to result if window is complete
    if (i >= k - 1) {
      result.push(nums[deque[0]]);
    }
  }

  return result;
}

/**
```

```
 * First Negative Number in Every Window of Size K
 * Time: O(n), Space: O(k)
 */
function firstNegativeInWindow(arr, k) {
  const result = [];
  const negatives = []; // Queue of negative number indices

  for (let i = 0; i < arr.length; i++) {
    // Remove indices outside current window
    while (negatives.length > 0 && negatives[0] <= i - k) {
      negatives.shift();
    }

    // Add current negative number
    if (arr[i] < 0) {
      negatives.push(i);
    }

    // Add result for current window
    if (i >= k - 1) {
      result.push(negatives.length > 0 ? arr[negatives[0]] : 0);
    }
  }

  return result;
}

// ===== PATTERN 2: VARIABLE SIZE SLIDING WINDOW =====

/**
 * Longest Substring Without Repeating Characters
 * Time: O(n), Space: O(min(m,n)) where m is charset size
 */
function lengthOfLongestSubstring(s) {
  const charSet = new Set();
  let left = 0;
  let maxLength = 0;

  for (let right = 0; right < s.length; right++) {
    // Shrink window until no duplicates
    while (charSet.has(s[right])) {
      charSet.delete(s[left]);
      left++;
    }

    charSet.add(s[right]);
    maxLength = Math.max(maxLength, right - left + 1);
  }

  return maxLength;
}

/**
 * Longest Substring with At Most K Distinct Characters
```

```
 * Time: O(n), Space: O(k)
 */
function lengthOfLongestSubstringKDistinct(s, k) {
  if (k === 0) return 0;

  const charCount = new Map();
  let left = 0;
  let maxLength = 0;

  for (let right = 0; right < s.length; right++) {
    // Add character to window
    charCount.set(s[right], (charCount.get(s[right]) || 0) + 1);

    // Shrink window if more than k distinct characters
    while (charCount.size > k) {
      const leftChar = s[left];
      charCount.set(leftChar, charCount.get(leftChar) - 1);
      if (charCount.get(leftChar) === 0) {
        charCount.delete(leftChar);
      }
      left++;
    }

    maxLength = Math.max(maxLength, right - left + 1);
  }

  return maxLength;
}

/**
 * Longest Substring with At Most 2 Distinct Characters
 * Time: O(n), Space: O(1)
 */
function lengthOfLongestSubstringTwoDistinct(s) {
  return lengthOfLongestSubstringKDistinct(s, 2);
}

/**
 * Longest Repeating Character Replacement
 * Time: O(n), Space: O(1) - at most 26 characters
 */
function characterReplacement(s, k) {
  const charCount = new Map();
  let left = 0;
  let maxLength = 0;
  let maxCount = 0; // Count of most frequent character in current window

  for (let right = 0; right < s.length; right++) {
    charCount.set(s[right], (charCount.get(s[right]) || 0) + 1);
    maxCount = Math.max(maxCount, charCount.get(s[right]));

    // If window size - maxCount > k, shrink window
    if (right - left + 1 - maxCount > k) {
      charCount.set(s[left], charCount.get(s[left]) - 1);
```

```
      left++;
    }

    maxLength = Math.max(maxLength, right - left + 1);
  }

  return maxLength;
}

/**
 * Subarray with Given Sum (Positive Numbers)
 * Time: O(n), Space: O(1)
 */
function subarraySum(arr, targetSum) {
  let left = 0;
  let currentSum = 0;

  for (let right = 0; right < arr.length; right++) {
    currentSum += arr[right];

    // Shrink window if sum exceeds target
    while (currentSum > targetSum && left <= right) {
      currentSum -= arr[left];
      left++;
    }

    if (currentSum === targetSum) {
      return [left, right]; // Return indices
    }
  }

  return [-1, -1]; // No subarray found
}

/**
 * Smallest Subarray with Sum Greater Than X
 * Time: O(n), Space: O(1)
 */
function smallestSubarrayWithSum(arr, x) {
  let left = 0;
  let currentSum = 0;
  let minLength = Infinity;

  for (let right = 0; right < arr.length; right++) {
    currentSum += arr[right];

    // Shrink window while sum > x
    while (currentSum > x && left <= right) {
      minLength = Math.min(minLength, right - left + 1);
      currentSum -= arr[left];
      left++;
    }
  }
```

```
  return minLength === Infinity ? 0 : minLength;
}

// ===== PATTERN 3: SHRINKABLE SLIDING WINDOW =====

/**
 * Minimum Window Substring
 * Time: O(|s| + |t|), Space: O(|s| + |t|)
 */
function minWindow(s, t) {
  if (s.length < t.length) return "";

  // Count characters in t
  const targetCount = new Map();
  for (let char of t) {
    targetCount.set(char, (targetCount.get(char) || 0) + 1);
  }

  let left = 0;
  let minLength = Infinity;
  let minStart = 0;
  let required = targetCount.size;
  let formed = 0;
  const windowCount = new Map();

  for (let right = 0; right < s.length; right++) {
    const char = s[right];
    windowCount.set(char, (windowCount.get(char) || 0) + 1);

    if (
      targetCount.has(char) &&
      windowCount.get(char) === targetCount.get(char)
    ) {
      formed++;
    }

    // Try to shrink window
    while (left <= right && formed === required) {
      if (right - left + 1 < minLength) {
        minLength = right - left + 1;
        minStart = left;
      }

      const leftChar = s[left];
      windowCount.set(leftChar, windowCount.get(leftChar) - 1);

      if (
        targetCount.has(leftChar) &&
        windowCount.get(leftChar) < targetCount.get(leftChar)
      ) {
        formed--;
      }

      left++;
```

```
    }
  }

  return minLength === Infinity
    ? ""
    : s.substring(minStart, minStart + minLength);
}

/**
 * Find All Anagrams in a String
 * Time: O(|s| + |p|), Space: O(1) - at most 26 characters
 */
function findAnagrams(s, p) {
  if (s.length < p.length) return [];

  const result = [];
  const pCount = new Map();
  const windowCount = new Map();

  // Count characters in p
  for (let char of p) {
    pCount.set(char, (pCount.get(char) || 0) + 1);
  }

  let left = 0;
  let right = 0;

  while (right < s.length) {
    // Expand window
    const rightChar = s[right];
    windowCount.set(rightChar, (windowCount.get(rightChar) || 0) + 1);

    // Shrink window if size exceeds p.length
    if (right - left + 1 > p.length) {
      const leftChar = s[left];
      windowCount.set(leftChar, windowCount.get(leftChar) - 1);
      if (windowCount.get(leftChar) === 0) {
        windowCount.delete(leftChar);
      }
      left++;
    }

    // Check if current window is anagram
    if (right - left + 1 === p.length && mapsEqual(windowCount, pCount)) {
      result.push(left);
    }

    right++;
  }

  return result;
}

// Helper function to compare maps
```

```
function mapsEqual(map1, map2) {
  if (map1.size !== map2.size) return false;

  for (let [key, value] of map1) {
    if (map2.get(key) !== value) return false;
  }

  return true;
}

/**
 * Permutation in String
 * Time: O(|s1| + |s2|), Space: O(1)
 */
function checkInclusion(s1, s2) {
  if (s1.length > s2.length) return false;

  const s1Count = new Map();
  const windowCount = new Map();

  // Count characters in s1
  for (let char of s1) {
    s1Count.set(char, (s1Count.get(char) || 0) + 1);
  }

  let left = 0;

  for (let right = 0; right < s2.length; right++) {
    // Expand window
    const rightChar = s2[right];
    windowCount.set(rightChar, (windowCount.get(rightChar) || 0) + 1);

    // Shrink window if size exceeds s1.length
    if (right - left + 1 > s1.length) {
      const leftChar = s2[left];
      windowCount.set(leftChar, windowCount.get(leftChar) - 1);
      if (windowCount.get(leftChar) === 0) {
        windowCount.delete(leftChar);
      }
      left++;
    }

    // Check if current window is permutation
    if (right - left + 1 === s1.length && mapsEqual(windowCount, s1Count)) {
      return true;
    }
  }

  return false;
}

// ===== PATTERN 4: NON-SHRINKABLE SLIDING WINDOW =====

/**
```

```
 * Longest Subarray with Ones after Replacement
 * Time: O(n), Space: O(1)
 */
function longestOnes(nums, k) {
  let left = 0;
  let maxLength = 0;
  let zeroCount = 0;

  for (let right = 0; right < nums.length; right++) {
    if (nums[right] === 0) {
      zeroCount++;
    }

    // Shrink window if zero count exceeds k
    while (zeroCount > k) {
      if (nums[left] === 0) {
        zeroCount--;
      }
      left++;
    }

    maxLength = Math.max(maxLength, right - left + 1);
  }

  return maxLength;
}

/**
 * Fruits into Baskets (At Most 2 Types)
 * Time: O(n), Space: O(1)
 */
function totalFruit(fruits) {
  const fruitCount = new Map();
  let left = 0;
  let maxFruits = 0;

  for (let right = 0; right < fruits.length; right++) {
    fruitCount.set(fruits[right], (fruitCount.get(fruits[right]) || 0) + 1);

    // Shrink window if more than 2 types
    while (fruitCount.size > 2) {
      const leftFruit = fruits[left];
      fruitCount.set(leftFruit, fruitCount.get(leftFruit) - 1);
      if (fruitCount.get(leftFruit) === 0) {
        fruitCount.delete(leftFruit);
      }
      left++;
    }

    maxFruits = Math.max(maxFruits, right - left + 1);
  }

  return maxFruits;
}
```

```javascript
// ===== ADVANCED SLIDING WINDOW TECHNIQUES =====

/**
 * Sliding Window Maximum with Custom Comparator
 * Time: O(n log k), Space: O(k)
 */
function slidingWindowMaximumCustom(nums, k, compareFn) {
  const result = [];
  const window = [];

  for (let i = 0; i < nums.length; i++) {
    // Remove elements outside window
    while (window.length > 0 && window[0].index <= i - k) {
      window.shift();
    }

    // Maintain decreasing order
    while (
      window.length > 0 &&
      compareFn(nums[i], window[window.length - 1].value) >= 0
    ) {
      window.pop();
    }

    window.push({ value: nums[i], index: i });

    if (i >= k - 1) {
      result.push(window[0].value);
    }
  }

  return result;
}

/**
 * Count Number of Nice Subarrays
 * Time: O(n), Space: O(1)
 */
function numberOfSubarrays(nums, k) {
  return atMostK(nums, k) - atMostK(nums, k - 1);

  function atMostK(nums, k) {
    let left = 0;
    let count = 0;
    let oddCount = 0;

    for (let right = 0; right < nums.length; right++) {
      if (nums[right] % 2 === 1) {
        oddCount++;
      }

      while (oddCount > k) {
        if (nums[left] % 2 === 1) {
```

```
            oddCount--;
          }
          left++;
        }

        count += right - left + 1;
      }

      return count;
    }
  }

  /**
   * Sliding Window Median
   * Time: O(n log k), Space: O(k)
   */
  function medianSlidingWindow(nums, k) {
    const result = [];
    const window = [];

    for (let i = 0; i < nums.length; i++) {
      // Add current element
      insertSorted(window, nums[i]);

      // Remove element outside window
      if (window.length > k) {
        const toRemove = nums[i - k];
        const index = window.indexOf(toRemove);
        window.splice(index, 1);
      }

      // Calculate median
      if (window.length === k) {
        const median =
          k % 2 === 1
            ? window[Math.floor(k / 2)]
            : (window[k / 2 - 1] + window[k / 2]) / 2;
        result.push(median);
      }
    }

    return result;
  }

  function insertSorted(arr, val) {
    let left = 0;
    let right = arr.length;

    while (left < right) {
      const mid = Math.floor((left + right) / 2);
      if (arr[mid] < val) {
        left = mid + 1;
      } else {
        right = mid;
```

```
    }
  }

  arr.splice(left, 0, val);
}

// ===== UTILITY FUNCTIONS =====

/**
 * Generic Sliding Window Template
 */
function slidingWindowTemplate(arr, k, operation) {
  const result = [];
  let windowState = null;

  for (let i = 0; i < arr.length; i++) {
    // Add element to window
    windowState = operation.add(windowState, arr[i], i);

    // Remove element if window too large
    if (i >= k) {
      windowState = operation.remove(windowState, arr[i - k], i - k);
    }

    // Process window if complete
    if (i >= k - 1) {
      result.push(operation.getResult(windowState));
    }
  }

  return result;
}

/**
 * Performance Testing
 */
function performanceTest() {
  console.log("=== Sliding Window Performance Test ===");

  const sizes = [1000, 10000, 100000];

  sizes.forEach((size) => {
    const arr = Array.from({ length: size }, (_, i) =>
      Math.floor(Math.random() * 100)
    );
    const k = Math.min(100, size);

    console.time(`Max Sum Subarray (size: ${size})`);
    maxSumSubarray(arr, k);
    console.timeEnd(`Max Sum Subarray (size: ${size})`);

    const str = Array.from({ length: size }, () =>
      String.fromCharCode(97 + Math.floor(Math.random() * 26))
    ).join("");
```

```
    console.time(`Longest Substring (size: ${size})`);
    lengthOfLongestSubstring(str);
    console.timeEnd(`Longest Substring (size: ${size})`);
  });
}

// ===== EXAMPLE USAGE AND TESTING =====

console.log("=== Sliding Window Technique Demo ===");

// Test Fixed Size Window
console.log("\n=== Fixed Size Window ===");
const arr1 = [2, 1, 5, 1, 3, 2];
console.log("Max Sum (k=3):", maxSumSubarray(arr1, 3)); // 9
console.log("Averages (k=3):", findAverages(arr1, 3)); // [2.67, 2.33, 3, 2]

const arr2 = [1, 3, -1, -3, 5, 3, 6, 7];
console.log("Max Sliding Window (k=3):", maxSlidingWindow(arr2, 3)); // [3, 3, 5,
5, 6, 7]

const arr3 = [12, -1, -7, 8, -15, 30, 16, 28];
console.log("First Negative (k=3):", firstNegativeInWindow(arr3, 3)); // [-1, -1,
-7, -15, -15, 0]

// Test Variable Size Window
console.log("\n=== Variable Size Window ===");
console.log("Longest Substring:", lengthOfLongestSubstring("abcabcbb")); // 3
console.log(
  "Longest K Distinct:",
  lengthOfLongestSubstringKDistinct("araaci", 2)
); // 4
console.log(
  "Longest 2 Distinct:",
  lengthOfLongestSubstringTwoDistinct("eceba")
); // 3
console.log("Character Replacement:", characterReplacement("AABABBA", 1)); // 4

const arr4 = [1, 4, 4];
console.log("Subarray Sum (target=9):", subarraySum(arr4, 9)); // [0, 2]

const arr5 = [1, 4, 4];
console.log("Smallest Subarray (x=6):", smallestSubarrayWithSum(arr5, 6)); // 2

// Test Shrinkable Window
console.log("\n=== Shrinkable Window ===");
console.log("Min Window:", minWindow("ADOBECODEBANC", "ABC")); // 'BANC'
console.log("Find Anagrams:", findAnagrams("abab", "ab")); // [0, 2]
console.log("Check Inclusion:", checkInclusion("ab", "eidbaooo")); // true

// Test Non-shrinkable Window
console.log("\n=== Non-shrinkable Window ===");
const arr6 = [1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0];
console.log("Longest Ones (k=2):", longestOnes(arr6, 2)); // 6
```

```
const fruits = [1, 2, 1];
console.log("Total Fruit:", totalFruit(fruits)); // 3

// Test Advanced Techniques
console.log("\n=== Advanced Techniques ===");
const arr7 = [1, 1, 2, 1, 1];
console.log("Nice Subarrays (k=3):", numberOfSubarrays(arr7, 3)); // 2

const arr8 = [1, 3, -1, -3, 5, 3, 6, 7];
console.log("Sliding Window Median (k=3):", medianSlidingWindow(arr8, 3)); // [1,
-1, -1, 3, 5, 6]

// Custom comparator example
const customMax = slidingWindowMaximumCustom(arr2, 3, (a, b) => a - b);
console.log("Custom Max:", customMax);

// Performance test
performanceTest();

// Pattern Recognition
console.log("\n=== Pattern Recognition ===");
console.log("Choose the right pattern:");
console.log("1. Fixed Size: When window size is constant");
console.log("2. Variable Size: When optimizing window size");
console.log("3. Shrinkable: When finding minimum valid window");
console.log("4. Non-shrinkable: When finding maximum valid window");
```

🔧 C++ Implementation

```
#include <iostream>
#include <vector>
#include <string>
#include <unordered_map>
#include <unordered_set>
#include <deque>
#include <algorithm>
#include <climits>
using namespace std;

// ===== FIXED SIZE SLIDING WINDOW =====

// Maximum Sum of Subarray of Size K
int maxSumSubarray(vector<int>& arr, int k) {
    if (arr.size() < k) return -1;

    int windowSum = 0;
    for (int i = 0; i < k; i++) {
        windowSum += arr[i];
    }
```

```
    int maxSum = windowSum;

    for (int i = k; i < arr.size(); i++) {
        windowSum = windowSum - arr[i - k] + arr[i];
        maxSum = max(maxSum, windowSum);
    }

    return maxSum;
}

// Maximum Sliding Window
vector<int> maxSlidingWindow(vector<int>& nums, int k) {
    vector<int> result;
    deque<int> dq; // Store indices

    for (int i = 0; i < nums.size(); i++) {
        // Remove indices outside current window
        while (!dq.empty() && dq.front() <= i - k) {
            dq.pop_front();
        }

        // Remove indices of smaller elements
        while (!dq.empty() && nums[dq.back()] <= nums[i]) {
            dq.pop_back();
        }

        dq.push_back(i);

        if (i >= k - 1) {
            result.push_back(nums[dq.front()]);
        }
    }

    return result;
}

// ===== VARIABLE SIZE SLIDING WINDOW =====

// Longest Substring Without Repeating Characters
int lengthOfLongestSubstring(string s) {
    unordered_set<char> charSet;
    int left = 0, maxLength = 0;

    for (int right = 0; right < s.length(); right++) {
        while (charSet.count(s[right])) {
            charSet.erase(s[left]);
            left++;
        }

        charSet.insert(s[right]);
        maxLength = max(maxLength, right - left + 1);
    }
```

```
    return maxLength;
}

// Longest Substring with At Most K Distinct Characters
int lengthOfLongestSubstringKDistinct(string s, int k) {
    if (k == 0) return 0;

    unordered_map<char, int> charCount;
    int left = 0, maxLength = 0;

    for (int right = 0; right < s.length(); right++) {
        charCount[s[right]]++;

        while (charCount.size() > k) {
            charCount[s[left]]--;
            if (charCount[s[left]] == 0) {
                charCount.erase(s[left]);
            }
            left++;
        }

        maxLength = max(maxLength, right - left + 1);
    }

    return maxLength;
}

// Longest Repeating Character Replacement
int characterReplacement(string s, int k) {
    unordered_map<char, int> charCount;
    int left = 0, maxLength = 0, maxCount = 0;

    for (int right = 0; right < s.length(); right++) {
        charCount[s[right]]++;
        maxCount = max(maxCount, charCount[s[right]]);

        if (right - left + 1 - maxCount > k) {
            charCount[s[left]]--;
            left++;
        }

        maxLength = max(maxLength, right - left + 1);
    }

    return maxLength;
}

// Subarray with Given Sum
vector<int> subarraySum(vector<int>& arr, int targetSum) {
    int left = 0, currentSum = 0;

    for (int right = 0; right < arr.size(); right++) {
        currentSum += arr[right];
```

```
        while (currentSum > targetSum && left <= right) {
            currentSum -= arr[left];
            left++;
        }

        if (currentSum == targetSum) {
            return {left, right};
        }
    }

    return {-1, -1};
}

// ===== SHRINKABLE SLIDING WINDOW =====

// Minimum Window Substring
string minWindow(string s, string t) {
    if (s.length() < t.length()) return "";

    unordered_map<char, int> targetCount;
    for (char c : t) {
        targetCount[c]++;
    }

    int left = 0, minLength = INT_MAX, minStart = 0;
    int required = targetCount.size(), formed = 0;
    unordered_map<char, int> windowCount;

    for (int right = 0; right < s.length(); right++) {
        char c = s[right];
        windowCount[c]++;

        if (targetCount.count(c) && windowCount[c] == targetCount[c]) {
            formed++;
        }

        while (left <= right && formed == required) {
            if (right - left + 1 < minLength) {
                minLength = right - left + 1;
                minStart = left;
            }

            char leftChar = s[left];
            windowCount[leftChar]--;

            if (targetCount.count(leftChar) && windowCount[leftChar] <
targetCount[leftChar]) {
                formed--;
            }

            left++;
        }
    }
```

```
    return minLength == INT_MAX ? "" : s.substr(minStart, minLength);
}

// Find All Anagrams in a String
vector<int> findAnagrams(string s, string p) {
    if (s.length() < p.length()) return {};

    vector<int> result;
    unordered_map<char, int> pCount, windowCount;

    for (char c : p) {
        pCount[c]++;
    }

    int left = 0;

    for (int right = 0; right < s.length(); right++) {
        windowCount[s[right]]++;

        if (right - left + 1 > p.length()) {
            windowCount[s[left]]--;
            if (windowCount[s[left]] == 0) {
                windowCount.erase(s[left]);
            }
            left++;
        }

        if (right - left + 1 == p.length() && windowCount == pCount) {
            result.push_back(left);
        }
    }

    return result;
}

// ===== NON-SHRINKABLE SLIDING WINDOW =====

// Longest Subarray with Ones after Replacement
int longestOnes(vector<int>& nums, int k) {
    int left = 0, maxLength = 0, zeroCount = 0;

    for (int right = 0; right < nums.size(); right++) {
        if (nums[right] == 0) {
            zeroCount++;
        }

        while (zeroCount > k) {
            if (nums[left] == 0) {
                zeroCount--;
            }
            left++;
        }

        maxLength = max(maxLength, right - left + 1);
```

```
    }

    return maxLength;
}

// Fruits into Baskets
int totalFruit(vector<int>& fruits) {
    unordered_map<int, int> fruitCount;
    int left = 0, maxFruits = 0;

    for (int right = 0; right < fruits.size(); right++) {
        fruitCount[fruits[right]]++;

        while (fruitCount.size() > 2) {
            fruitCount[fruits[left]]--;
            if (fruitCount[fruits[left]] == 0) {
                fruitCount.erase(fruits[left]);
            }
            left++;
        }

        maxFruits = max(maxFruits, right - left + 1);
    }

    return maxFruits;
}

// ===== UTILITY FUNCTIONS =====

template<typename T>
void printVector(const vector<T>& vec, const string& label = "") {
    if (!label.empty()) {
        cout << label << ": ";
    }
    for (const auto& x : vec) {
        cout << x << " ";
    }
    cout << endl;
}

// Example Usage
int main() {
    cout << "=== Sliding Window Technique Demo ===" << endl;

    // Test Fixed Size Window
    cout << "\n=== Fixed Size Window ===" << endl;
    vector<int> arr1 = {2, 1, 5, 1, 3, 2};
    cout << "Max Sum (k=3): " << maxSumSubarray(arr1, 3) << endl;

    vector<int> arr2 = {1, 3, -1, -3, 5, 3, 6, 7};
    auto maxWindow = maxSlidingWindow(arr2, 3);
    printVector(maxWindow, "Max Sliding Window (k=3)");

    // Test Variable Size Window
```

```
    cout << "\n=== Variable Size Window ===" << endl;
    cout << "Longest Substring: " << lengthOfLongestSubstring("abcabcbb") << endl;
    cout << "Longest K Distinct: " << lengthOfLongestSubstringKDistinct("araaci",
2) << endl;
    cout << "Character Replacement: " << characterReplacement("AABABBA", 1) <<
endl;

    vector<int> arr4 = {1, 4, 4};
    auto subarrayResult = subarraySum(arr4, 9);
    printVector(subarrayResult, "Subarray Sum (target=9)");

    // Test Shrinkable Window
    cout << "\n=== Shrinkable Window ===" << endl;
    cout << "Min Window: " << minWindow("ADOBECODEBANC", "ABC") << endl;

    auto anagrams = findAnagrams("abab", "ab");
    printVector(anagrams, "Find Anagrams");

    // Test Non-shrinkable Window
    cout << "\n=== Non-shrinkable Window ===" << endl;
    vector<int> arr6 = {1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0};
    cout << "Longest Ones (k=2): " << longestOnes(arr6, 2) << endl;

    vector<int> fruits = {1, 2, 1};
    cout << "Total Fruit: " << totalFruit(fruits) << endl;

    return 0;
}
```

⚡ Performance Analysis

Time Complexity Improvements:

Problem Type	Brute Force	Sliding Window	Improvement
Fixed Size Problems	$O(n \times k)$	$O(n)$	k times faster
Variable Size Problems	$O(n^2)$ or $O(n^3)$	$O(n)$	n times faster
Substring Problems	$O(n^2 \times m)$	$O(n+m)$	Exponential improvement
Subarray Sum	$O(n^2)$	$O(n)$	n times faster

Space Complexity:

- **Fixed Size**: O(1) or O(k)
- **Variable Size**: O(k) where k is window size
- **Character Problems**: O(1) for ASCII (at most 256 characters)
- **Much better than generating all subarrays**: $O(n^2)$ space

When Sliding Window Excels:

- ✅ **Contiguous elements**: Problems involving subarrays/substrings
- ✅ **Optimization**: Finding min/max/longest/shortest
- ✅ **Constraints**: Problems with specific conditions
- ✅ **Streaming data**: Processing data sequentially

🧩 Practice Problems

Problem 1: Minimum Size Subarray Sum

Question: Find minimum length subarray with sum ≥ target. **Example**: `[2,3,1,2,4,3]`, target=7 → 2 (subarray `[4,3]`) **Hint**: Use variable size window, expand until sum ≥ target, then shrink.

Problem 2: Longest Substring with At Most K Distinct Characters

Question: Find longest substring with at most k distinct characters. **Example**: `"eceba"`, k=2 → 3 (`"ece"`) **Hint**: Use hash map to count characters, shrink when count > k.

Problem 3: Sliding Window Maximum

Question: Find maximum in each window of size k. **Example**: `[1,3,-1,-3,5,3,6,7]`, k=3 → `[3,3,5,5,6,7]` **Hint**: Use deque to maintain decreasing order of elements.

Problem 4: Count Subarrays with K Odd Numbers

Question: Count subarrays with exactly k odd numbers. **Example**: `[1,1,2,1,1]`, k=3 → 2 **Hint**: Use "at most k" - "at most k-1" technique.

🎯 Interview Tips

What Interviewers Look For:

1. **Pattern recognition**: Can you identify sliding window problems?
2. **Window management**: Do you expand/contract correctly?
3. **Edge case handling**: Empty input, window larger than array
4. **Optimization**: Can you achieve O(n) time complexity?

Common Interview Patterns:

- **Fixed size**: "Find max/min in every window of size k"
- **Variable size**: "Find longest/shortest subarray with condition"
- **Character counting**: "Find substring with specific character constraints"
- **Two pointers**: "Maintain window with left and right pointers"

Red Flags to Avoid:

- Using nested loops for subarray problems
- Not maintaining window invariants correctly
- Forgetting to handle edge cases
- Inefficient window expansion/contraction

Pro Tips:

1. **Identify the pattern**: Look for "subarray", "substring", "window", "contiguous"
2. **Choose right variant**: Fixed vs variable size
3. **Maintain invariants**: What should be true about the window?
4. **Handle edge cases**: Empty input, k > array size
5. **Optimize data structures**: Use appropriate containers for tracking
6. **Practice templates**: Master the basic patterns

🚀 Key Takeaways

1. **Sliding window optimizes subarray problems** - From $O(n^2)$ to $O(n)$
2. **Four main patterns** - Fixed, variable, shrinkable, non-shrinkable
3. **Two pointers manage window** - Left and right boundaries
4. **Maintain window invariants** - What conditions must the window satisfy?
5. **Choose right data structures** - Hash maps, sets, deques as needed
6. **Master the templates** - Practice until pattern recognition is automatic

Next Chapter: We'll explore Basic Dynamic Programming and see how to solve optimization problems by breaking them into subproblems.

Chapter 13: Basic Dynamic Programming - Optimizing Through Subproblems

🎯 What is Dynamic Programming?

Dynamic Programming (DP) is an algorithmic technique that solves complex problems by breaking them down into simpler subproblems and storing the results to avoid redundant calculations. It's particularly effective for optimization problems where we need to find the best solution among many possibilities.

Why Dynamic Programming Matters:

- **Optimization**: Reduces exponential time complexity to polynomial
- **Efficiency**: Avoids redundant calculations through memoization
- **Versatile**: Solves many types of optimization problems
- **Real applications**: Resource allocation, scheduling, game theory
- **Interview favorite**: Essential for technical interviews

Core Principles:

1. **Optimal Substructure**: Optimal solution contains optimal solutions to subproblems
2. **Overlapping Subproblems**: Same subproblems are solved multiple times
3. **Memoization**: Store results to avoid recomputation

📊 Dynamic Programming Approaches

Approach Classifications:

Approach	Description	Implementation	Use Cases
Top-Down (Memoization)	Recursive with caching	Recursion + memo table	Natural recursive problems
Bottom-Up (Tabulation)	Iterative table filling	Loops + DP table	When iteration is clearer
Space Optimized	Reduce space complexity	Rolling arrays	When only recent states needed

When to Use Dynamic Programming:

- ✅ **Optimization problems**: Find min/max/count/best solution
- ✅ **Overlapping subproblems**: Same calculations repeated
- ✅ **Optimal substructure**: Optimal solution built from optimal subsolutions
- ✅ **Decision problems**: Make choices that affect future options

---

💻 JavaScript Implementation

```
// Dynamic Programming - Comprehensive Implementation

// ===== CLASSIC DP PROBLEMS =====

/**
 * Fibonacci Sequence - Classic DP Introduction
 * Problem: Find nth Fibonacci number
 * Recurrence: F(n) = F(n-1) + F(n-2)
 */

// Naive Recursive (Exponential Time)
function fibonacciNaive(n) {
  if (n <= 1) return n;
  return fibonacciNaive(n - 1) + fibonacciNaive(n - 2);
}

// Top-Down DP (Memoization)
function fibonacciMemo(n, memo = {}) {
  if (n in memo) return memo[n];
  if (n <= 1) return n;

  memo[n] = fibonacciMemo(n - 1, memo) + fibonacciMemo(n - 2, memo);
  return memo[n];
}

// Bottom-Up DP (Tabulation)
function fibonacciDP(n) {
  if (n <= 1) return n;
```

```
  const dp = new Array(n + 1);
  dp[0] = 0;
  dp[1] = 1;

  for (let i = 2; i <= n; i++) {
    dp[i] = dp[i - 1] + dp[i - 2];
  }

  return dp[n];
}

// Space Optimized DP
function fibonacciOptimized(n) {
  if (n <= 1) return n;

  let prev2 = 0;
  let prev1 = 1;

  for (let i = 2; i <= n; i++) {
    const current = prev1 + prev2;
    prev2 = prev1;
    prev1 = current;
  }

  return prev1;
}

/**
 * Climbing Stairs
 * Problem: Count ways to climb n stairs (1 or 2 steps at a time)
 * Recurrence: ways(n) = ways(n-1) + ways(n-2)
 */
function climbStairs(n) {
  if (n <= 2) return n;

  let prev2 = 1; // ways(1)
  let prev1 = 2; // ways(2)

  for (let i = 3; i <= n; i++) {
    const current = prev1 + prev2;
    prev2 = prev1;
    prev1 = current;
  }

  return prev1;
}

/**
 * House Robber
 * Problem: Rob houses to maximize money without robbing adjacent houses
 * Recurrence: rob(i) = max(rob(i-1), rob(i-2) + nums[i])
 */
function rob(nums) {
  if (nums.length === 0) return 0;
```

```
  if (nums.length === 1) return nums[0];

  let prev2 = 0; // rob(i-2)
  let prev1 = nums[0]; // rob(i-1)

  for (let i = 1; i < nums.length; i++) {
    const current = Math.max(prev1, prev2 + nums[i]);
    prev2 = prev1;
    prev1 = current;
  }

  return prev1;
}

/**
 * Coin Change
 * Problem: Find minimum coins needed to make amount
 * Recurrence: dp[amount] = min(dp[amount - coin] + 1) for all coins
 */
function coinChange(coins, amount) {
  const dp = new Array(amount + 1).fill(Infinity);
  dp[0] = 0;

  for (let i = 1; i <= amount; i++) {
    for (let coin of coins) {
      if (coin <= i) {
        dp[i] = Math.min(dp[i], dp[i - coin] + 1);
      }
    }
  }

  return dp[amount] === Infinity ? -1 : dp[amount];
}

/**
 * Coin Change II - Count Ways
 * Problem: Count number of ways to make amount
 * Recurrence: dp[amount] = sum(dp[amount - coin]) for all coins
 */
function change(amount, coins) {
  const dp = new Array(amount + 1).fill(0);
  dp[0] = 1;

  // Process coins one by one to avoid counting permutations
  for (let coin of coins) {
    for (let i = coin; i <= amount; i++) {
      dp[i] += dp[i - coin];
    }
  }

  return dp[amount];
}

/**
```

```
 * Longest Increasing Subsequence (LIS)
 * Problem: Find length of longest increasing subsequence
 * Recurrence: dp[i] = max(dp[j] + 1) where j < i and nums[j] < nums[i]
 */
function lengthOfLIS(nums) {
  if (nums.length === 0) return 0;

  const dp = new Array(nums.length).fill(1);
  let maxLength = 1;

  for (let i = 1; i < nums.length; i++) {
    for (let j = 0; j < i; j++) {
      if (nums[j] < nums[i]) {
        dp[i] = Math.max(dp[i], dp[j] + 1);
      }
    }
    maxLength = Math.max(maxLength, dp[i]);
  }

  return maxLength;
}

/**
 * Maximum Subarray (Kadane's Algorithm)
 * Problem: Find maximum sum of contiguous subarray
 * Recurrence: maxEndingHere = max(nums[i], maxEndingHere + nums[i])
 */
function maxSubArray(nums) {
  let maxSoFar = nums[0];
  let maxEndingHere = nums[0];

  for (let i = 1; i < nums.length; i++) {
    maxEndingHere = Math.max(nums[i], maxEndingHere + nums[i]);
    maxSoFar = Math.max(maxSoFar, maxEndingHere);
  }

  return maxSoFar;
}

/**
 * Unique Paths
 * Problem: Count unique paths from top-left to bottom-right in grid
 * Recurrence: dp[i][j] = dp[i-1][j] + dp[i][j-1]
 */
function uniquePaths(m, n) {
  const dp = Array(m)
    .fill(null)
    .map(() => 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];
}

// Space Optimized Version
function uniquePathsOptimized(m, n) {
  let dp = new Array(n).fill(1);

  for (let i = 1; i < m; i++) {
    for (let j = 1; j < n; j++) {
      dp[j] += dp[j - 1];
    }
  }

  return dp[n - 1];
}

/**
 * Unique Paths II (with obstacles)
 * Problem: Count unique paths with obstacles
 * Recurrence: dp[i][j] = dp[i-1][j] + dp[i][j-1] if no obstacle
 */
function uniquePathsWithObstacles(obstacleGrid) {
  const m = obstacleGrid.length;
  const n = obstacleGrid[0].length;

  if (obstacleGrid[0][0] === 1) return 0;

  const dp = Array(m)
    .fill(null)
    .map(() => Array(n).fill(0));
  dp[0][0] = 1;

  // Fill first row
  for (let j = 1; j < n; j++) {
    dp[0][j] = obstacleGrid[0][j] === 1 ? 0 : dp[0][j - 1];
  }

  // Fill first column
  for (let i = 1; i < m; i++) {
    dp[i][0] = obstacleGrid[i][0] === 1 ? 0 : dp[i - 1][0];
  }

  // Fill rest of the grid
  for (let i = 1; i < m; i++) {
    for (let j = 1; j < n; j++) {
      if (obstacleGrid[i][j] === 1) {
        dp[i][j] = 0;
      } else {
        dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
      }
    }
  }
```

```
  return dp[m - 1][n - 1];
}

/**
 * Minimum Path Sum
 * Problem: Find minimum sum path from top-left to bottom-right
 * Recurrence: dp[i][j] = grid[i][j] + min(dp[i-1][j], dp[i][j-1])
 */
function minPathSum(grid) {
  const m = grid.length;
  const n = grid[0].length;

  const dp = Array(m)
    .fill(null)
    .map(() => Array(n).fill(0));
  dp[0][0] = grid[0][0];

  // Fill first row
  for (let j = 1; j < n; j++) {
    dp[0][j] = dp[0][j - 1] + grid[0][j];
  }

  // Fill first column
  for (let i = 1; i < m; i++) {
    dp[i][0] = dp[i - 1][0] + grid[i][0];
  }

  // Fill rest of the grid
  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];
}

// ===== STRING DP PROBLEMS =====

/**
 * Longest Common Subsequence (LCS)
 * Problem: Find length of longest common subsequence
 * Recurrence: dp[i][j] = dp[i-1][j-1] + 1 if chars match, else max(dp[i-1][j],
dp[i][j-1])
 */
function longestCommonSubsequence(text1, text2) {
  const m = text1.length;
  const n = text2.length;

  const dp = Array(m + 1)
    .fill(null)
    .map(() => Array(n + 1).fill(0));

  for (let i = 1; i <= m; i++) {
```

```
    for (let j = 1; j <= n; j++) {
      if (text1[i - 1] === text2[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];
}

/**
 * Edit Distance (Levenshtein Distance)
 * Problem: Minimum operations to convert word1 to word2
 * Operations: insert, delete, replace
 */
function minDistance(word1, word2) {
  const m = word1.length;
  const n = word2.length;

  const dp = Array(m + 1)
    .fill(null)
    .map(() => Array(n + 1).fill(0));

  // Base cases
  for (let i = 0; i <= m; i++) dp[i][0] = i; // Delete all
  for (let j = 0; j <= n; j++) dp[0][j] = j; // Insert all

  for (let i = 1; i <= m; i++) {
    for (let j = 1; j <= n; j++) {
      if (word1[i - 1] === word2[j - 1]) {
        dp[i][j] = dp[i - 1][j - 1]; // No operation needed
      } else {
        dp[i][j] =
          1 +
          Math.min(
            dp[i - 1][j], // Delete
            dp[i][j - 1], // Insert
            dp[i - 1][j - 1] // Replace
          );
      }
    }
  }

  return dp[m][n];
}

/**
 * Palindromic Substrings
 * Problem: Count number of palindromic substrings
 * Approach: Expand around centers
 */
function countSubstrings(s) {
```

```
  let count = 0;

  function expandAroundCenter(left, right) {
    while (left >= 0 && right < s.length && s[left] === s[right]) {
      count++;
      left--;
      right++;
    }
  }

  for (let i = 0; i < s.length; i++) {
    expandAroundCenter(i, i); // Odd length palindromes
    expandAroundCenter(i, i + 1); // Even length palindromes
  }

  return count;
}

/**
 * Longest Palindromic Substring
 * Problem: Find longest palindromic substring
 * Approach: DP table or expand around centers
 */
function longestPalindrome(s) {
  if (s.length <= 1) return s;

  let start = 0;
  let maxLength = 1;

  function expandAroundCenter(left, right) {
    while (left >= 0 && right < s.length && s[left] === s[right]) {
      const currentLength = right - left + 1;
      if (currentLength > maxLength) {
        start = left;
        maxLength = currentLength;
      }
      left--;
      right++;
    }
  }

  for (let i = 0; i < s.length; i++) {
    expandAroundCenter(i, i); // Odd length
    expandAroundCenter(i, i + 1); // Even length
  }

  return s.substring(start, start + maxLength);
}

// ===== KNAPSACK PROBLEMS =====

/**
 * 0/1 Knapsack
 * Problem: Maximum value with weight constraint
```

```
 * Recurrence: dp[i][w] = max(dp[i-1][w], dp[i-1][w-weight[i]] + value[i])
 */
function knapsack(weights, values, capacity) {
  const n = weights.length;
  const dp = Array(n + 1)
    .fill(null)
    .map(() => Array(capacity + 1).fill(0));

  for (let i = 1; i <= n; i++) {
    for (let w = 1; w <= capacity; w++) {
      if (weights[i - 1] <= w) {
        dp[i][w] = Math.max(
          dp[i - 1][w], // Don't take item
          dp[i - 1][w - weights[i - 1]] + values[i - 1] // Take item
        );
      } else {
        dp[i][w] = dp[i - 1][w]; // Can't take item
      }
    }
  }

  return dp[n][capacity];
}

// Space Optimized Knapsack
function knapsackOptimized(weights, values, capacity) {
  let dp = new Array(capacity + 1).fill(0);

  for (let i = 0; i < weights.length; i++) {
    // Traverse backwards to avoid using updated values
    for (let w = capacity; w >= weights[i]; w--) {
      dp[w] = Math.max(dp[w], dp[w - weights[i]] + values[i]);
    }
  }

  return dp[capacity];
}

/**
 * Partition Equal Subset Sum
 * Problem: Check if array can be partitioned into two equal sum subsets
 * Approach: Knapsack variant - find subset with sum = total/2
 */
function canPartition(nums) {
  const sum = nums.reduce((a, b) => a + b, 0);
  if (sum % 2 !== 0) return false;

  const target = sum / 2;
  const dp = new Array(target + 1).fill(false);
  dp[0] = true;

  for (let num of nums) {
    for (let j = target; j >= num; j--) {
      dp[j] = dp[j] || dp[j - num];
```

```
    }
  }

  return dp[target];
}

// ===== UTILITY FUNCTIONS =====

/**
 * Generic DP Template with Memoization
 */
function dpWithMemo(problem, ...args) {
  const memo = new Map();

  function solve(...params) {
    const key = JSON.stringify(params);
    if (memo.has(key)) return memo.get(key);

    const result = problem(...params, solve);
    memo.set(key, result);
    return result;
  }

  return solve(...args);
}

/**
 * Performance Comparison
 */
function performanceComparison() {
  console.log("=== DP Performance Comparison ===");

  const n = 35;

  console.time("Fibonacci Naive");
  // fibonacciNaive(n); // Too slow for large n
  console.timeEnd("Fibonacci Naive");

  console.time("Fibonacci Memoization");
  fibonacciMemo(n);
  console.timeEnd("Fibonacci Memoization");

  console.time("Fibonacci DP");
  fibonacciDP(n);
  console.timeEnd("Fibonacci DP");

  console.time("Fibonacci Optimized");
  fibonacciOptimized(n);
  console.timeEnd("Fibonacci Optimized");
}

/**
 * DP Problem Classifier
 */
```

```
function classifyDPProblem(description) {
  const patterns = {
    sequence: ["fibonacci", "climbing", "house robber"],
    grid: ["unique paths", "minimum path", "dungeon"],
    string: ["edit distance", "lcs", "palindrome"],
    knapsack: ["subset sum", "coin change", "partition"],
    interval: ["matrix chain", "burst balloons"],
    tree: ["binary tree", "diameter", "path sum"],
  };

  for (let [category, keywords] of Object.entries(patterns)) {
    if (
      keywords.some((keyword) => description.toLowerCase().includes(keyword))
    ) {
      return category;
    }
  }

  return "unknown";
}

// ===== EXAMPLE USAGE AND TESTING =====

console.log("=== Dynamic Programming Demo ===");

// Test Classic Problems
console.log("\n=== Classic DP Problems ===");
console.log("Fibonacci(10):", fibonacciDP(10)); // 55
console.log("Climb Stairs(5):", climbStairs(5)); // 8
console.log("House Robber([2,7,9,3,1]):", rob([2, 7, 9, 3, 1])); // 12
console.log("Coin Change([1,3,4], 6):", coinChange([1, 3, 4], 6)); // 2
console.log("Coin Change Ways([1,2,5], 5):", change(5, [1, 2, 5])); // 4
console.log(
  "LIS([10,9,2,5,3,7,101,18]):",
  lengthOfLIS([10, 9, 2, 5, 3, 7, 101, 18])
); // 4
console.log(
  "Max Subarray([-2,1,-3,4,-1,2,1,-5,4]):",
  maxSubArray([-2, 1, -3, 4, -1, 2, 1, -5, 4])
); // 6

// Test Grid Problems
console.log("\n=== Grid DP Problems ===");
console.log("Unique Paths(3,7):", uniquePaths(3, 7)); // 28
const obstacleGrid = [
  [0, 0, 0],
  [0, 1, 0],
  [0, 0, 0],
];
console.log(
  "Unique Paths with Obstacles:",
  uniquePathsWithObstacles(obstacleGrid)
); // 2
const grid = [
```

```
  [1, 3, 1],
  [1, 5, 1],
  [4, 2, 1],
];
console.log("Min Path Sum:", minPathSum(grid)); // 7

// Test String Problems
console.log("\n=== String DP Problems ===");
console.log('LCS("abcde", "ace"):', longestCommonSubsequence("abcde", "ace")); //
3
console.log('Edit Distance("horse", "ros"):', minDistance("horse", "ros")); // 3
console.log('Palindromic Substrings("abc"):', countSubstrings("abc")); // 3
console.log('Longest Palindrome("babad"):', longestPalindrome("babad")); // "bab"
or "aba"

// Test Knapsack Problems
console.log("\n=== Knapsack DP Problems ===");
const weights = [1, 3, 4, 5];
const values = [1, 4, 5, 7];
console.log("Knapsack(capacity=7):", knapsack(weights, values, 7)); // 9
console.log("Can Partition([1,5,11,5]):", canPartition([1, 5, 11, 5])); // true

// Performance comparison
performanceComparison();

// Problem classification
console.log("\n=== Problem Classification ===");
console.log(
  '"Find fibonacci number":',
  classifyDPProblem("Find fibonacci number")
);
console.log(
  '"Count unique paths in grid":',
  classifyDPProblem("Count unique paths in grid")
);
console.log(
  '"Edit distance between strings":',
  classifyDPProblem("Edit distance between strings")
);
console.log('"Subset sum problem":', classifyDPProblem("Subset sum problem"));

console.log("\n=== DP Strategy Guide ===");
console.log("1. Identify optimal substructure");
console.log("2. Find overlapping subproblems");
console.log("3. Define recurrence relation");
console.log("4. Choose memoization vs tabulation");
console.log("5. Optimize space if possible");
```

🔧 C++ Implementation

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <unordered_map>
#include <climits>
using namespace std;

// ===== CLASSIC DP PROBLEMS =====

// Fibonacci with Memoization
class FibonacciMemo {
private:
    unordered_map<int, long long> memo;

public:
    long long fib(int n) {
        if (memo.find(n) != memo.end()) {
            return memo[n];
        }

        if (n <= 1) {
            return memo[n] = n;
        }

        return memo[n] = fib(n - 1) + fib(n - 2);
    }
};

// Fibonacci Bottom-Up
long long fibonacciDP(int n) {
    if (n <= 1) return n;

    vector<long long> dp(n + 1);
    dp[0] = 0;
    dp[1] = 1;

    for (int i = 2; i <= n; i++) {
        dp[i] = dp[i - 1] + dp[i - 2];
    }

    return dp[n];
}

// Climbing Stairs
int climbStairs(int n) {
    if (n <= 2) return n;

    int prev2 = 1, prev1 = 2;

    for (int i = 3; i <= n; i++) {
        int current = prev1 + prev2;
        prev2 = prev1;
```

```
            prev1 = current;
        }

        return prev1;
    }

    // House Robber
    int rob(vector<int>& nums) {
        if (nums.empty()) return 0;
        if (nums.size() == 1) return nums[0];

        int prev2 = 0, prev1 = nums[0];

        for (int i = 1; i < nums.size(); i++) {
            int current = max(prev1, prev2 + nums[i]);
            prev2 = prev1;
            prev1 = current;
        }

        return prev1;
    }

    // Coin Change
    int coinChange(vector<int>& coins, int amount) {
        vector<int> dp(amount + 1, INT_MAX);
        dp[0] = 0;

        for (int i = 1; i <= amount; i++) {
            for (int coin : coins) {
                if (coin <= i && dp[i - coin] != INT_MAX) {
                    dp[i] = min(dp[i], dp[i - coin] + 1);
                }
            }
        }

        return dp[amount] == INT_MAX ? -1 : dp[amount];
    }

    // Longest Increasing Subsequence
    int lengthOfLIS(vector<int>& nums) {
        if (nums.empty()) return 0;

        vector<int> dp(nums.size(), 1);
        int maxLength = 1;

        for (int i = 1; i < nums.size(); i++) {
            for (int j = 0; j < i; j++) {
                if (nums[j] < nums[i]) {
                    dp[i] = max(dp[i], dp[j] + 1);
                }
            }
            maxLength = max(maxLength, dp[i]);
        }
```

```
    return maxLength;
}

// Maximum Subarray (Kadane's Algorithm)
int maxSubArray(vector<int>& nums) {
    int maxSoFar = nums[0];
    int maxEndingHere = nums[0];

    for (int i = 1; i < nums.size(); i++) {
        maxEndingHere = max(nums[i], maxEndingHere + nums[i]);
        maxSoFar = max(maxSoFar, maxEndingHere);
    }

    return maxSoFar;
}

// ===== GRID DP PROBLEMS =====

// Unique Paths
int uniquePaths(int m, int n) {
    vector<vector<int>> dp(m, vector<int>(n, 1));

    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
        }
    }

    return dp[m - 1][n - 1];
}

// Minimum Path Sum
int minPathSum(vector<vector<int>>& grid) {
    int m = grid.size(), n = grid[0].size();
    vector<vector<int>> dp(m, vector<int>(n));

    dp[0][0] = grid[0][0];

    // Fill first row
    for (int j = 1; j < n; j++) {
        dp[0][j] = dp[0][j - 1] + grid[0][j];
    }

    // Fill first column
    for (int i = 1; i < m; i++) {
        dp[i][0] = dp[i - 1][0] + grid[i][0];
    }

    // Fill rest of grid
    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            dp[i][j] = grid[i][j] + min(dp[i - 1][j], dp[i][j - 1]);
        }
    }
```

```
    return dp[m - 1][n - 1];
}

// ===== STRING DP PROBLEMS =====

// Longest Common Subsequence
int longestCommonSubsequence(string text1, string text2) {
    int m = text1.length(), n = text2.length();
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (text1[i - 1] == text2[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }

    return dp[m][n];
}

// Edit Distance
int minDistance(string word1, string word2) {
    int m = word1.length(), n = word2.length();
    vector<vector<int>> dp(m + 1, vector<int>(n + 1));

    // Base cases
    for (int i = 0; i <= m; i++) dp[i][0] = i;
    for (int j = 0; j <= n; j++) dp[0][j] = j;

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (word1[i - 1] == word2[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1];
            } else {
                dp[i][j] = 1 + min({dp[i - 1][j], dp[i][j - 1], dp[i - 1][j -
1]});
            }
        }
    }

    return dp[m][n];
}

// ===== KNAPSACK PROBLEMS =====

// 0/1 Knapsack
int knapsack(vector<int>& weights, vector<int>& values, int capacity) {
    int n = weights.size();
    vector<vector<int>> dp(n + 1, vector<int>(capacity + 1, 0));
```

```
    for (int i = 1; i <= n; i++) {
        for (int w = 1; w <= capacity; w++) {
            if (weights[i - 1] <= w) {
                dp[i][w] = max(dp[i - 1][w],
                               dp[i - 1][w - weights[i - 1]] + values[i - 1]);
            } else {
                dp[i][w] = dp[i - 1][w];
            }
        }
    }

    return dp[n][capacity];
}

// Partition Equal Subset Sum
bool canPartition(vector<int>& nums) {
    int sum = 0;
    for (int num : nums) sum += num;

    if (sum % 2 != 0) return false;

    int target = sum / 2;
    vector<bool> dp(target + 1, false);
    dp[0] = true;

    for (int num : nums) {
        for (int j = target; j >= num; j--) {
            dp[j] = dp[j] || dp[j - num];
        }
    }

    return dp[target];
}

// ===== UTILITY FUNCTIONS =====

template<typename T>
void printVector(const vector<T>& vec, const string& label = "") {
    if (!label.empty()) {
        cout << label << ": ";
    }
    for (const auto& x : vec) {
        cout << x << " ";
    }
    cout << endl;
}

void printMatrix(const vector<vector<int>>& matrix, const string& label = "") {
    if (!label.empty()) {
        cout << label << ":" << endl;
    }
    for (const auto& row : matrix) {
        for (int val : row) {
            cout << val << " ";
```

```
            }
            cout << endl;
        }
    }

    // Example Usage
    int main() {
        cout << "=== Dynamic Programming Demo ===" << endl;

        // Test Classic Problems
        cout << "\n=== Classic DP Problems ===" << endl;
        cout << "Fibonacci(10): " << fibonacciDP(10) << endl;
        cout << "Climb Stairs(5): " << climbStairs(5) << endl;

        vector<int> houses = {2, 7, 9, 3, 1};
        cout << "House Robber: " << rob(houses) << endl;

        vector<int> coins = {1, 3, 4};
        cout << "Coin Change(6): " << coinChange(coins, 6) << endl;

        vector<int> nums = {10, 9, 2, 5, 3, 7, 101, 18};
        cout << "LIS: " << lengthOfLIS(nums) << endl;

        vector<int> subarray = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
        cout << "Max Subarray: " << maxSubArray(subarray) << endl;

        // Test Grid Problems
        cout << "\n=== Grid DP Problems ===" << endl;
        cout << "Unique Paths(3,7): " << uniquePaths(3, 7) << endl;

        vector<vector<int>> grid = {{1, 3, 1}, {1, 5, 1}, {4, 2, 1}};
        cout << "Min Path Sum: " << minPathSum(grid) << endl;

        // Test String Problems
        cout << "\n=== String DP Problems ===" << endl;
        cout << "LCS('abcde', 'ace'): " << longestCommonSubsequence("abcde", "ace") <<
    endl;
        cout << "Edit Distance('horse', 'ros'): " << minDistance("horse", "ros") <<
    endl;

        // Test Knapsack Problems
        cout << "\n=== Knapsack DP Problems ===" << endl;
        vector<int> weights = {1, 3, 4, 5};
        vector<int> values = {1, 4, 5, 7};
        cout << "Knapsack(capacity=7): " << knapsack(weights, values, 7) << endl;

        vector<int> partition = {1, 5, 11, 5};
        cout << "Can Partition: " << (canPartition(partition) ? "true" : "false") <<
    endl;

        return 0;
    }
```

⚡ Performance Analysis

Time Complexity Improvements:

Problem	Naive Approach	DP Approach	Improvement
Fibonacci	$O(2^n)$	O(n)	Exponential to linear
Coin Change	O(amount^coins)	O(amount × coins)	Exponential to polynomial
LCS	O(2^(m+n))	O(m × n)	Exponential to polynomial
Knapsack	$O(2^n)$	O(n × capacity)	Exponential to polynomial
Edit Distance	O(3^max(m,n))	O(m × n)	Exponential to polynomial

Space Complexity:

- **2D DP**: O(m × n) - can often be optimized to O(min(m, n))
- **1D DP**: O(n) - can often be optimized to O(1)
- **Memoization**: O(recursion depth + memo size)

When DP Excels:

- ✅ **Optimization problems**: Min/max/count solutions
- ✅ **Overlapping subproblems**: Same calculations repeated
- ✅ **Optimal substructure**: Optimal solution contains optimal subsolutions
- ✅ **Decision problems**: Choices affect future options

🧩 Practice Problems

Problem 1: Triangle Minimum Path Sum

Question: Find minimum path sum from top to bottom of triangle. **Example**: `[[2],[3,4],[6,5,7],[4,1,8,3]]` → 11 (path: 2+3+5+1) **Hint**: Bottom-up DP, dp[i][j] = triangle[i][j] + min(dp[i+1][j], dp[i+1][j+1])

Problem 2: Word Break

Question: Check if string can be segmented into dictionary words. **Example**: `"leetcode"`, dict=["leet","code"] → true **Hint**: dp[i] = true if substring(0,i) can be segmented

Problem 3: Decode Ways

Question: Count ways to decode numeric string to letters (A=1, B=2, ..., Z=26). **Example**: `"226"` → 3 ("BZ", "VF", "BBF") **Hint**: dp[i] = dp[i-1] + dp[i-2] (if valid single/double digit)

Problem 4: Maximum Product Subarray

Question: Find contiguous subarray with maximum product. **Example**: `[2,3,-2,4]` → 6 (subarray [2,3]) **Hint**: Track both max and min products (negative × negative = positive)

🎯 Interview Tips

What Interviewers Look For:

1. **Problem recognition**: Can you identify DP problems?
2. **Recurrence relation**: Can you define the recursive structure?
3. **Base cases**: Do you handle edge cases correctly?
4. **Optimization**: Can you optimize space complexity?

Common Interview Patterns:

- **1D DP**: Fibonacci-like problems, house robber, climbing stairs
- **2D DP**: Grid problems, string matching, knapsack
- **State machines**: Problems with different states/modes
- **Interval DP**: Problems involving ranges or intervals

Red Flags to Avoid:

- Not identifying optimal substructure
- Incorrect recurrence relation
- Missing base cases
- Not considering space optimization
- Confusing top-down vs bottom-up

Pro Tips:

1. **Start with recursion**: Write naive recursive solution first
2. **Identify overlapping subproblems**: Look for repeated calculations
3. **Define state clearly**: What parameters uniquely identify a subproblem?
4. **Choose approach**: Memoization (top-down) vs tabulation (bottom-up)
5. **Optimize space**: Can you reduce dimensions?
6. **Practice patterns**: Master common DP patterns

🚀 Key Takeaways

1. **DP optimizes overlapping subproblems** - Avoid redundant calculations
2. **Two main approaches** - Top-down (memoization) vs bottom-up (tabulation)
3. **Optimal substructure required** - Optimal solution contains optimal subsolutions
4. **State definition is crucial** - What parameters define a subproblem?
5. **Space optimization often possible** - Reduce dimensions when only recent states needed
6. **Practice pattern recognition** - Learn to identify DP problems quickly

Next Chapter: We'll explore Backtracking and see how to systematically explore solution spaces by making and undoing choices.

Chapter 14: Backtracking - Systematic Solution Space Exploration

🎯 What is Backtracking?

Backtracking is an algorithmic technique that systematically explores all possible solutions to a problem by making choices, exploring their consequences, and undoing (backtracking) when a choice leads to a dead end. It's essentially a refined brute force approach that prunes invalid paths early.

Why Backtracking Matters:

- **Systematic exploration**: Explores all possible solutions methodically
- **Early pruning**: Eliminates invalid paths to improve efficiency
- **Constraint satisfaction**: Solves problems with multiple constraints
- **Combinatorial problems**: Generates permutations, combinations, subsets
- **Game solving**: Chess, Sudoku, N-Queens, maze solving
- **Interview favorite**: Tests problem-solving and recursion skills

Core Principles:

1. **Choose**: Make a choice from available options
2. **Explore**: Recursively explore the consequences
3. **Unchoose**: Backtrack if the path doesn't lead to a solution

---

📊 Backtracking Framework

General Template:

```
function backtrack(state, choices):
    if (isComplete(state)):
        processResult(state)
        return

    for choice in choices:
        if (isValid(choice, state)):
            makeChoice(choice, state)
            backtrack(state, getNextChoices(state))
            undoChoice(choice, state)  // Backtrack
```

Problem Classifications:

Type	Description	Examples	Characteristics
Decision Problems	Find if solution exists	N-Queens, Sudoku	Boolean result
Optimization Problems	Find best solution	Traveling Salesman	Compare solutions
Enumeration Problems	Find all solutions	All permutations	Generate all results
Construction Problems	Build valid solution	Generate parentheses	Construct step by step

When to Use Backtracking:

- ✅ **Constraint satisfaction**: Multiple rules to satisfy
- ✅ **Combinatorial enumeration**: Generate all possibilities
- ✅ **Puzzle solving**: Sudoku, crosswords, mazes
- ✅ **Game tree search**: Chess, tic-tac-toe
- ✅ **Path finding**: With constraints or obstacles

💻 JavaScript Implementation

```
// Backtracking - Comprehensive Implementation

// ===== CLASSIC BACKTRACKING PROBLEMS =====

/**
 * N-Queens Problem
 * Problem: Place N queens on NxN chessboard so none attack each other
 * Constraints: No two queens in same row, column, or diagonal
 */
function solveNQueens(n) {
  const result = [];
  const board = Array(n)
    .fill(null)
    .map(() => Array(n).fill("."));

  function isValid(row, col) {
    // Check column
    for (let i = 0; i < row; i++) {
      if (board[i][col] === "Q") return false;
    }

    // Check diagonal (top-left to bottom-right)
    for (let i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--) {
      if (board[i][j] === "Q") return false;
    }

    // Check diagonal (top-right to bottom-left)
    for (let i = row - 1, j = col + 1; i >= 0 && j < n; i--, j++) {
      if (board[i][j] === "Q") return false;
    }

    return true;
  }

  function backtrack(row) {
    // Base case: all queens placed
    if (row === n) {
      result.push(board.map((row) => row.join("")));
      return;
    }

    // Try placing queen in each column of current row
    for (let col = 0; col < n; col++) {
```

```
      if (isValid(row, col)) {
        board[row][col] = "Q"; // Choose
        backtrack(row + 1); // Explore
        board[row][col] = "."; // Unchoose
      }
    }
  }

  backtrack(0);
  return result;
}

/**
 * Generate Parentheses
 * Problem: Generate all valid combinations of n pairs of parentheses
 * Constraints: Balanced parentheses
 */
function generateParenthesis(n) {
  const result = [];

  function backtrack(current, open, close) {
    // Base case: used all parentheses
    if (current.length === 2 * n) {
      result.push(current);
      return;
    }

    // Add opening parenthesis if we haven't used all
    if (open < n) {
      backtrack(current + "(", open + 1, close);
    }

    // Add closing parenthesis if it won't make string invalid
    if (close < open) {
      backtrack(current + ")", open, close + 1);
    }
  }

  backtrack("", 0, 0);
  return result;
}

/**
 * Permutations
 * Problem: Generate all permutations of given array
 * Approach: Choose element, recurse on remaining, backtrack
 */
function permute(nums) {
  const result = [];
  const current = [];
  const used = new Array(nums.length).fill(false);

  function backtrack() {
    // Base case: permutation complete
```

```
    if (current.length === nums.length) {
      result.push([...current]); // Copy array
      return;
    }

    // Try each unused number
    for (let i = 0; i < nums.length; i++) {
      if (!used[i]) {
        current.push(nums[i]); // Choose
        used[i] = true;
        backtrack(); // Explore
        current.pop(); // Unchoose
        used[i] = false;
      }
    }
  }

  backtrack();
  return result;
}

/**
 * Permutations II (with duplicates)
 * Problem: Generate unique permutations from array with duplicates
 * Approach: Sort array and skip duplicates intelligently
 */
function permuteUnique(nums) {
  const result = [];
  const current = [];
  const used = new Array(nums.length).fill(false);

  nums.sort((a, b) => a - b); // Sort to group duplicates

  function backtrack() {
    if (current.length === nums.length) {
      result.push([...current]);
      return;
    }

    for (let i = 0; i < nums.length; i++) {
      if (used[i]) continue;

      // Skip duplicates: if current element equals previous
      // and previous is not used, skip current
      if (i > 0 && nums[i] === nums[i - 1] && !used[i - 1]) {
        continue;
      }

      current.push(nums[i]);
      used[i] = true;
      backtrack();
      current.pop();
      used[i] = false;
    }
```

```
  }

  backtrack();
  return result;
}

/**
 * Combinations
 * Problem: Generate all combinations of k numbers from 1 to n
 * Approach: Choose number, recurse with remaining choices
 */
function combine(n, k) {
  const result = [];
  const current = [];

  function backtrack(start) {
    // Base case: combination complete
    if (current.length === k) {
      result.push([...current]);
      return;
    }

    // Try numbers from start to n
    for (let i = start; i <= n; i++) {
      current.push(i); // Choose
      backtrack(i + 1); // Explore (i+1 to avoid duplicates)
      current.pop(); // Unchoose
    }
  }

  backtrack(1);
  return result;
}

/**
 * Combination Sum
 * Problem: Find all combinations that sum to target
 * Approach: Choose number, subtract from target, recurse
 */
function combinationSum(candidates, target) {
  const result = [];
  const current = [];

  function backtrack(start, remaining) {
    // Base case: found valid combination
    if (remaining === 0) {
      result.push([...current]);
      return;
    }

    // Base case: exceeded target
    if (remaining < 0) {
      return;
    }
```

```
    // Try each candidate from start index
    for (let i = start; i < candidates.length; i++) {
      current.push(candidates[i]);
      // Can reuse same number, so pass i (not i+1)
      backtrack(i, remaining - candidates[i]);
      current.pop();
    }
  }

  backtrack(0, target);
  return result;
}

/**
 * Combination Sum II (no duplicates)
 * Problem: Find combinations that sum to target, each number used once
 * Approach: Sort array, skip duplicates at same level
 */
function combinationSum2(candidates, target) {
  const result = [];
  const current = [];

  candidates.sort((a, b) => a - b);

  function backtrack(start, remaining) {
    if (remaining === 0) {
      result.push([...current]);
      return;
    }

    for (let i = start; i < candidates.length; i++) {
      // Skip duplicates at same recursion level
      if (i > start && candidates[i] === candidates[i - 1]) {
        continue;
      }

      if (candidates[i] > remaining) break; // Optimization

      current.push(candidates[i]);
      backtrack(i + 1, remaining - candidates[i]); // i+1: use each number once
      current.pop();
    }
  }

  backtrack(0, target);
  return result;
}

/**
 * Subsets
 * Problem: Generate all possible subsets (power set)
 * Approach: For each element, choose to include or exclude
 */
```

```javascript
function subsets(nums) {
  const result = [];
  const current = [];

  function backtrack(start) {
    // Add current subset to result
    result.push([...current]);

    // Try adding each remaining number
    for (let i = start; i < nums.length; i++) {
      current.push(nums[i]); // Choose
      backtrack(i + 1); // Explore
      current.pop(); // Unchoose
    }
  }

  backtrack(0);
  return result;
}

/**
 * Subsets II (with duplicates)
 * Problem: Generate unique subsets from array with duplicates
 * Approach: Sort and skip duplicates at same level
 */
function subsetsWithDup(nums) {
  const result = [];
  const current = [];

  nums.sort((a, b) => a - b);

  function backtrack(start) {
    result.push([...current]);

    for (let i = start; i < nums.length; i++) {
      // Skip duplicates at same level
      if (i > start && nums[i] === nums[i - 1]) {
        continue;
      }

      current.push(nums[i]);
      backtrack(i + 1);
      current.pop();
    }
  }

  backtrack(0);
  return result;
}

/**
 * Palindrome Partitioning
 * Problem: Partition string into palindromic substrings
 * Approach: Try all possible cuts, check if substring is palindrome
```

```
 */
function partition(s) {
  const result = [];
  const current = [];

  function isPalindrome(str, start, end) {
    while (start < end) {
      if (str[start] !== str[end]) return false;
      start++;
      end--;
    }
    return true;
  }

  function backtrack(start) {
    // Base case: processed entire string
    if (start === s.length) {
      result.push([...current]);
      return;
    }

    // Try all possible end positions
    for (let end = start; end < s.length; end++) {
      if (isPalindrome(s, start, end)) {
        current.push(s.substring(start, end + 1)); // Choose
        backtrack(end + 1); // Explore
        current.pop(); // Unchoose
      }
    }
  }

  backtrack(0);
  return result;
}

/**
 * Word Search
 * Problem: Find if word exists in 2D board
 * Approach: DFS with backtracking, mark visited cells
 */
function exist(board, word) {
  const rows = board.length;
  const cols = board[0].length;
  const directions = [
    [0, 1],
    [1, 0],
    [0, -1],
    [-1, 0],
  ];

  function backtrack(row, col, index) {
    // Base case: found complete word
    if (index === word.length) return true;
```

```
    // Check bounds and character match
    if (
      row < 0 ||
      row >= rows ||
      col < 0 ||
      col >= cols ||
      board[row][col] !== word[index] ||
      board[row][col] === "#"
    ) {
      return false;
    }

    // Mark cell as visited
    const temp = board[row][col];
    board[row][col] = "#";

    // Explore all directions
    for (let [dr, dc] of directions) {
      if (backtrack(row + dr, col + dc, index + 1)) {
        board[row][col] = temp; // Restore
        return true;
      }
    }

    // Backtrack: restore cell
    board[row][col] = temp;
    return false;
  }

  // Try starting from each cell
  for (let i = 0; i < rows; i++) {
    for (let j = 0; j < cols; j++) {
      if (backtrack(i, j, 0)) {
        return true;
      }
    }
  }

  return false;
}

/**
 * Sudoku Solver
 * Problem: Solve 9x9 Sudoku puzzle
 * Approach: Try numbers 1-9 in empty cells, backtrack if invalid
 */
function solveSudoku(board) {
  function isValid(board, row, col, num) {
    // Check row
    for (let j = 0; j < 9; j++) {
      if (board[row][j] === num) return false;
    }

    // Check column
```

```
    for (let i = 0; i < 9; i++) {
      if (board[i][col] === num) return false;
    }

    // Check 3x3 box
    const boxRow = Math.floor(row / 3) * 3;
    const boxCol = Math.floor(col / 3) * 3;
    for (let i = boxRow; i < boxRow + 3; i++) {
      for (let j = boxCol; j < boxCol + 3; j++) {
        if (board[i][j] === num) return false;
      }
    }

    return true;
  }

  function backtrack() {
    for (let i = 0; i < 9; i++) {
      for (let j = 0; j < 9; j++) {
        if (board[i][j] === ".") {
          // Try numbers 1-9
          for (let num = "1"; num <= "9"; num++) {
            if (isValid(board, i, j, num)) {
              board[i][j] = num; // Choose
              if (backtrack()) {
                // Explore
                return true;
              }
              board[i][j] = "."; // Unchoose
            }
          }
          return false; // No valid number found
        }
      }
    }
    return true; // All cells filled
  }

  backtrack();
}

/**
 * Letter Combinations of Phone Number
 * Problem: Generate all letter combinations from phone number
 * Approach: Map digits to letters, backtrack through combinations
 */
function letterCombinations(digits) {
  if (digits.length === 0) return [];

  const phoneMap = {
    2: "abc",
    3: "def",
    4: "ghi",
    5: "jkl",
```

```
    6: "mno",
    7: "pqrs",
    8: "tuv",
    9: "wxyz",
  };

  const result = [];

  function backtrack(index, current) {
    // Base case: processed all digits
    if (index === digits.length) {
      result.push(current);
      return;
    }

    // Get letters for current digit
    const letters = phoneMap[digits[index]];

    // Try each letter
    for (let letter of letters) {
      backtrack(index + 1, current + letter);
    }
  }

  backtrack(0, "");
  return result;
}

// ===== ADVANCED BACKTRACKING PROBLEMS =====

/**
 * Restore IP Addresses
 * Problem: Generate all valid IP addresses from string
 * Approach: Try all possible segment lengths, validate each segment
 */
function restoreIpAddresses(s) {
  const result = [];
  const segments = [];

  function isValidSegment(segment) {
    if (segment.length === 0 || segment.length > 3) return false;
    if (segment[0] === "0" && segment.length > 1) return false; // No leading
zeros
    const num = parseInt(segment);
    return num >= 0 && num <= 255;
  }

  function backtrack(start) {
    // Base case: 4 segments formed
    if (segments.length === 4) {
      if (start === s.length) {
        result.push(segments.join("."));
      }
      return;
```

```
    }

    // Try segment lengths 1, 2, 3
    for (let len = 1; len <= 3 && start + len <= s.length; len++) {
      const segment = s.substring(start, start + len);
      if (isValidSegment(segment)) {
        segments.push(segment); // Choose
        backtrack(start + len); // Explore
        segments.pop(); // Unchoose
      }
    }
  }

  backtrack(0);
  return result;
}

/**
 * Expression Add Operators
 * Problem: Add +, -, * operators to make target
 * Approach: Try all operator placements, handle precedence
 */
function addOperators(num, target) {
  const result = [];

  function backtrack(index, expression, value, prev) {
    // Base case: processed all digits
    if (index === num.length) {
      if (value === target) {
        result.push(expression);
      }
      return;
    }

    // Try all possible number lengths from current position
    for (let i = index; i < num.length; i++) {
      const numStr = num.substring(index, i + 1);

      // Skip numbers with leading zeros (except single '0')
      if (numStr.length > 1 && numStr[0] === "0") break;

      const numVal = parseInt(numStr);

      if (index === 0) {
        // First number, no operator needed
        backtrack(i + 1, numStr, numVal, numVal);
      } else {
        // Try addition
        backtrack(i + 1, expression + "+" + numStr, value + numVal, numVal);

        // Try subtraction
        backtrack(i + 1, expression + "-" + numStr, value - numVal, -numVal);

        // Try multiplication (handle precedence)
```

```
          backtrack(
            i + 1,
            expression + "*" + numStr,
            value - prev + prev * numVal,
            prev * numVal
          );
        }
      }
    }

    backtrack(0, "", 0, 0);
    return result;
}

// ===== UTILITY FUNCTIONS =====

/**
 * Generic Backtracking Template
 */
function genericBacktrack(
    choices,
    isComplete,
    isValid,
    makeChoice,
    undoChoice,
    processResult
) {
    const state = [];

    function backtrack() {
        if (isComplete(state)) {
            processResult([...state]);
            return;
        }

        for (let choice of choices(state)) {
            if (isValid(choice, state)) {
                makeChoice(choice, state);
                backtrack();
                undoChoice(choice, state);
            }
        }
    }

    backtrack();
}

/**
 * Performance Measurement
 */
function measureBacktrackingPerformance(func, ...args) {
    const start = performance.now();
    const result = func(...args);
    const end = performance.now();
```

```
  console.log(`Function: ${func.name}`);
  console.log(`Time: ${(end - start).toFixed(2)}ms`);
  console.log(`Results: ${Array.isArray(result) ? result.length : result}`);

  return result;
}

/**
 * Backtracking Problem Classifier
 */
function classifyBacktrackingProblem(description) {
  const patterns = {
    permutation: ["permute", "arrange", "order"],
    combination: ["choose", "select", "subset"],
    partition: ["split", "divide", "partition"],
    constraint: ["sudoku", "queens", "valid"],
    path: ["maze", "word search", "path"],
    expression: ["operators", "parentheses", "formula"],
  };

  for (let [category, keywords] of Object.entries(patterns)) {
    if (
      keywords.some((keyword) => description.toLowerCase().includes(keyword))
    ) {
      return category;
    }
  }

  return "unknown";
}

// ===== EXAMPLE USAGE AND TESTING =====

console.log("=== Backtracking Demo ===");

// Test Classic Problems
console.log("\n=== Classic Backtracking Problems ===");
console.log("N-Queens(4):");
solveNQueens(4).forEach((solution, i) => {
  console.log(`Solution ${i + 1}:`);
  solution.forEach((row) => console.log(row));
  console.log();
});

console.log("Generate Parentheses(3):", generateParenthesis(3));
console.log("Permutations([1,2,3]):", permute([1, 2, 3]));
console.log("Combinations(4,2):", combine(4, 2));
console.log("Combination Sum([2,3,6,7], 7):", combinationSum([2, 3, 6, 7], 7));
console.log("Subsets([1,2,3]):", subsets([1, 2, 3]));

// Test String Problems
console.log("\n=== String Backtracking Problems ===");
console.log('Palindrome Partitioning("aab"):', partition("aab"));
```

```
console.log('Letter Combinations("23"):', letterCombinations("23"));
console.log('Restore IP("25525511135"):', restoreIpAddresses("25525511135"));

// Test Board Problems
console.log("\n=== Board Backtracking Problems ===");
const board = [
  ["A", "B", "C", "E"],
  ["S", "F", "C", "S"],
  ["A", "D", "E", "E"],
];
console.log('Word Search("ABCCED"):', exist(board, "ABCCED")); // true
console.log('Word Search("SEE"):', exist(board, "SEE")); // true
console.log('Word Search("ABCB"):', exist(board, "ABCB")); // false

// Performance measurement
console.log("\n=== Performance Measurement ===");
measureBacktrackingPerformance(permute, [1, 2, 3, 4]);
measureBacktrackingPerformance(subsets, [1, 2, 3, 4, 5]);
measureBacktrackingPerformance(combine, 10, 3);

// Problem classification
console.log("\n=== Problem Classification ===");
console.log(
  '"Generate all permutations":',
  classifyBacktrackingProblem("Generate all permutations")
);
console.log(
  '"Choose k elements":',
  classifyBacktrackingProblem("Choose k elements")
);
console.log(
  '"Solve sudoku puzzle":',
  classifyBacktrackingProblem("Solve sudoku puzzle")
);
console.log(
  '"Find path in maze":',
  classifyBacktrackingProblem("Find path in maze")
);

console.log("\n=== Backtracking Strategy Guide ===");
console.log("1. Identify choices at each step");
console.log("2. Define constraints/validity checks");
console.log("3. Implement choose-explore-unchoose pattern");
console.log("4. Handle base cases (complete/invalid states)");
console.log("5. Optimize with early pruning");
```

C++ Implementation

```
#include <iostream>
#include <vector>
```

```cpp
#include <string>
#include <algorithm>
#include <unordered_set>
using namespace std;

// ===== CLASSIC BACKTRACKING PROBLEMS =====

// N-Queens Problem
class NQueens {
public:
    vector<vector<string>> solveNQueens(int n) {
        vector<vector<string>> result;
        vector<string> board(n, string(n, '.'));
        backtrack(result, board, 0, n);
        return result;
    }

private:
    void backtrack(vector<vector<string>>& result, vector<string>& board, int row,
int n) {
        if (row == n) {
            result.push_back(board);
            return;
        }

        for (int col = 0; col < n; col++) {
            if (isValid(board, row, col, n)) {
                board[row][col] = 'Q';        // Choose
                backtrack(result, board, row + 1, n); // Explore
                board[row][col] = '.';        // Unchoose
            }
        }
    }

    bool isValid(vector<string>& board, int row, int col, int n) {
        // Check column
        for (int i = 0; i < row; i++) {
            if (board[i][col] == 'Q') return false;
        }

        // Check diagonal (top-left to bottom-right)
        for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--) {
            if (board[i][j] == 'Q') return false;
        }

        // Check diagonal (top-right to bottom-left)
        for (int i = row - 1, j = col + 1; i >= 0 && j < n; i--, j++) {
            if (board[i][j] == 'Q') return false;
        }

        return true;
    }
};
```

```cpp
// Generate Parentheses
class GenerateParentheses {
public:
    vector<string> generateParenthesis(int n) {
        vector<string> result;
        backtrack(result, "", 0, 0, n);
        return result;
    }

private:
    void backtrack(vector<string>& result, string current, int open, int close,
int n) {
        if (current.length() == 2 * n) {
            result.push_back(current);
            return;
        }

        if (open < n) {
            backtrack(result, current + "(", open + 1, close, n);
        }

        if (close < open) {
            backtrack(result, current + ")", open, close + 1, n);
        }
    }
};

// Permutations
class Permutations {
public:
    vector<vector<int>> permute(vector<int>& nums) {
        vector<vector<int>> result;
        vector<int> current;
        vector<bool> used(nums.size(), false);
        backtrack(result, current, nums, used);
        return result;
    }

private:
    void backtrack(vector<vector<int>>& result, vector<int>& current,
                   vector<int>& nums, vector<bool>& used) {
        if (current.size() == nums.size()) {
            result.push_back(current);
            return;
        }

        for (int i = 0; i < nums.size(); i++) {
            if (!used[i]) {
                current.push_back(nums[i]);  // Choose
                used[i] = true;
                backtrack(result, current, nums, used); // Explore
                current.pop_back();          // Unchoose
                used[i] = false;
            }
```

```
        }
    }
};

// Combinations
class Combinations {
public:
    vector<vector<int>> combine(int n, int k) {
        vector<vector<int>> result;
        vector<int> current;
        backtrack(result, current, 1, n, k);
        return result;
    }

private:
    void backtrack(vector<vector<int>>& result, vector<int>& current,
                   int start, int n, int k) {
        if (current.size() == k) {
            result.push_back(current);
            return;
        }

        for (int i = start; i <= n; i++) {
            current.push_back(i);          // Choose
            backtrack(result, current, i + 1, n, k); // Explore
            current.pop_back();            // Unchoose
        }
    }
};

// Combination Sum
class CombinationSum {
public:
    vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
        vector<vector<int>> result;
        vector<int> current;
        backtrack(result, current, candidates, target, 0);
        return result;
    }

private:
    void backtrack(vector<vector<int>>& result, vector<int>& current,
                   vector<int>& candidates, int remaining, int start) {
        if (remaining == 0) {
            result.push_back(current);
            return;
        }

        if (remaining < 0) return;

        for (int i = start; i < candidates.size(); i++) {
            current.push_back(candidates[i]);
            backtrack(result, current, candidates, remaining - candidates[i], i);
            current.pop_back();
```

```
        }
    }
};

// Subsets
class Subsets {
public:
    vector<vector<int>> subsets(vector<int>& nums) {
        vector<vector<int>> result;
        vector<int> current;
        backtrack(result, current, nums, 0);
        return result;
    }

private:
    void backtrack(vector<vector<int>>& result, vector<int>& current,
                   vector<int>& nums, int start) {
        result.push_back(current);

        for (int i = start; i < nums.size(); i++) {
            current.push_back(nums[i]);   // Choose
            backtrack(result, current, nums, i + 1); // Explore
            current.pop_back();           // Unchoose
        }
    }
};

// Palindrome Partitioning
class PalindromePartitioning {
public:
    vector<vector<string>> partition(string s) {
        vector<vector<string>> result;
        vector<string> current;
        backtrack(result, current, s, 0);
        return result;
    }

private:
    void backtrack(vector<vector<string>>& result, vector<string>& current,
                   string& s, int start) {
        if (start == s.length()) {
            result.push_back(current);
            return;
        }

        for (int end = start; end < s.length(); end++) {
            if (isPalindrome(s, start, end)) {
                current.push_back(s.substr(start, end - start + 1));
                backtrack(result, current, s, end + 1);
                current.pop_back();
            }
        }
    }
```

```
    bool isPalindrome(string& s, int start, int end) {
        while (start < end) {
            if (s[start] != s[end]) return false;
            start++;
            end--;
        }
        return true;
    }
};

// Word Search
class WordSearch {
public:
    bool exist(vector<vector<char>>& board, string word) {
        int rows = board.size(), cols = board[0].size();

        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                if (backtrack(board, word, i, j, 0)) {
                    return true;
                }
            }
        }

        return false;
    }

private:
    bool backtrack(vector<vector<char>>& board, string& word,
                   int row, int col, int index) {
        if (index == word.length()) return true;

        if (row < 0 || row >= board.size() || col < 0 || col >= board[0].size() ||
            board[row][col] != word[index] || board[row][col] == '#') {
            return false;
        }

        char temp = board[row][col];
        board[row][col] = '#';  // Mark as visited

        // Explore all directions
        vector<vector<int>> directions = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};
        for (auto& dir : directions) {
            if (backtrack(board, word, row + dir[0], col + dir[1], index + 1)) {
                board[row][col] = temp; // Restore
                return true;
            }
        }

        board[row][col] = temp; // Backtrack
        return false;
    }
};
```

```cpp
// Sudoku Solver
class SudokuSolver {
public:
    void solveSudoku(vector<vector<char>>& board) {
        backtrack(board);
    }

private:
    bool backtrack(vector<vector<char>>& board) {
        for (int i = 0; i < 9; i++) {
            for (int j = 0; j < 9; j++) {
                if (board[i][j] == '.') {
                    for (char num = '1'; num <= '9'; num++) {
                        if (isValid(board, i, j, num)) {
                            board[i][j] = num;    // Choose
                            if (backtrack(board)) { // Explore
                                return true;
                            }
                            board[i][j] = '.';    // Unchoose
                        }
                    }
                    return false;
                }
            }
        }
        return true;
    }

    bool isValid(vector<vector<char>>& board, int row, int col, char num) {
        // Check row
        for (int j = 0; j < 9; j++) {
            if (board[row][j] == num) return false;
        }

        // Check column
        for (int i = 0; i < 9; i++) {
            if (board[i][col] == num) return false;
        }

        // Check 3x3 box
        int boxRow = (row / 3) * 3;
        int boxCol = (col / 3) * 3;
        for (int i = boxRow; i < boxRow + 3; i++) {
            for (int j = boxCol; j < boxCol + 3; j++) {
                if (board[i][j] == num) return false;
            }
        }

        return true;
    }
};

// ===== UTILITY FUNCTIONS =====
```

```cpp
template<typename T>
void printVector(const vector<T>& vec, const string& label = "") {
    if (!label.empty()) {
        cout << label << ": ";
    }
    for (const auto& x : vec) {
        cout << x << " ";
    }
    cout << endl;
}

void printMatrix(const vector<vector<int>>& matrix, const string& label = "") {
    if (!label.empty()) {
        cout << label << ":" << endl;
    }
    for (const auto& row : matrix) {
        for (int val : row) {
            cout << val << " ";
        }
        cout << endl;
    }
}

void printStringMatrix(const vector<vector<string>>& matrix, const string& label =
"") {
    if (!label.empty()) {
        cout << label << ":" << endl;
    }
    for (const auto& row : matrix) {
        for (const string& val : row) {
            cout << val << " ";
        }
        cout << endl;
    }
}

// Example Usage
int main() {
    cout << "=== Backtracking Demo ===" << endl;

    // Test N-Queens
    cout << "\n=== N-Queens Problem ===" << endl;
    NQueens nq;
    auto queens = nq.solveNQueens(4);
    cout << "N-Queens(4) solutions: " << queens.size() << endl;
    for (int i = 0; i < queens.size(); i++) {
        cout << "Solution " << i + 1 << ":" << endl;
        for (const string& row : queens[i]) {
            cout << row << endl;
        }
        cout << endl;
    }

    // Test Generate Parentheses
```

```
    cout << "\n=== Generate Parentheses ===" << endl;
    GenerateParentheses gp;
    auto parentheses = gp.generateParenthesis(3);
    cout << "Generate Parentheses(3): ";
    for (const string& p : parentheses) {
        cout << p << " ";
    }
    cout << endl;

    // Test Permutations
    cout << "\n=== Permutations ===" << endl;
    Permutations perm;
    vector<int> nums = {1, 2, 3};
    auto perms = perm.permute(nums);
    cout << "Permutations([1,2,3]):" << endl;
    printMatrix(perms);

    // Test Combinations
    cout << "\n=== Combinations ===" << endl;
    Combinations comb;
    auto combs = comb.combine(4, 2);
    cout << "Combinations(4,2):" << endl;
    printMatrix(combs);

    // Test Combination Sum
    cout << "\n=== Combination Sum ===" << endl;
    CombinationSum cs;
    vector<int> candidates = {2, 3, 6, 7};
    auto combSums = cs.combinationSum(candidates, 7);
    cout << "Combination Sum([2,3,6,7], 7):" << endl;
    printMatrix(combSums);

    // Test Subsets
    cout << "\n=== Subsets ===" << endl;
    Subsets sub;
    vector<int> subNums = {1, 2, 3};
    auto subsets = sub.subsets(subNums);
    cout << "Subsets([1,2,3]):" << endl;
    printMatrix(subsets);

    // Test Palindrome Partitioning
    cout << "\n=== Palindrome Partitioning ===" << endl;
    PalindromePartitioning pp;
    auto partitions = pp.partition("aab");
    cout << "Palindrome Partitioning('aab'):" << endl;
    printStringMatrix(partitions);

    // Test Word Search
    cout << "\n=== Word Search ===" << endl;
    WordSearch ws;
    vector<vector<char>> board = {
        {'A', 'B', 'C', 'E'},
        {'S', 'F', 'C', 'S'},
        {'A', 'D', 'E', 'E'}
```

```
    };
    cout << "Word Search('ABCCED'): " << (ws.exist(board, "ABCCED") ? "true" :
"false") << endl;
    cout << "Word Search('SEE'): " << (ws.exist(board, "SEE") ? "true" : "false")
<< endl;
    cout << "Word Search('ABCB'): " << (ws.exist(board, "ABCB") ? "true" :
"false") << endl;

    return 0;
}
```

⚡ Performance Analysis

Time Complexity:

Problem	Time Complexity	Space Complexity	Notes
N-Queens	O(N!)	$O(N^2)$	N! possible arrangements
Permutations	O(N! × N)	O(N)	N! permutations, N to copy
Combinations	O(C(n,k))	O(k)	Binomial coefficient
Subsets	$O(2^n \times N)$	O(N)	2^n subsets, N to copy
Sudoku	O(9^(empty cells))	O(1)	Worst case: try all numbers
Word Search	O(M×N×4^L)	O(L)	M×N starting points, 4^L paths

Space Complexity Factors:

- **Recursion stack**: O(depth of recursion)
- **Current state**: O(size of partial solution)
- **Result storage**: O(number of solutions × solution size)

Optimization Techniques:

1. **Early pruning**: Eliminate invalid paths quickly
2. **Constraint propagation**: Use constraints to reduce search space
3. **Ordering heuristics**: Try most promising choices first
4. **Memoization**: Cache results of subproblems (when applicable)

🧩 Practice Problems

Problem 1: Beautiful Arrangement

Question: Count arrangements where nums[i] is divisible by i or i is divisible by nums[i]. **Example**: N=2 → 2 (arrangements: [1,2] and [2,1]) **Hint**: Use backtracking with position-based choices

Problem 2: Partition to K Equal Sum Subsets

Question: Check if array can be partitioned into k subsets with equal sum. **Example**: [4,3,2,3,5,2,1], k=4 → true **Hint**: Backtrack by trying to fill each subset to target sum

Problem 3: Remove Invalid Parentheses

Question: Remove minimum parentheses to make string valid. **Example**: "()())()" → ["()()()", "(())()"] **Hint**: BFS or backtracking with pruning

Problem 4: Word Search II

Question: Find all words from dictionary that exist in 2D board. **Example**: Board + ["oath","pea","eat","rain"] → ["eat","oath"] **Hint**: Use Trie + backtracking for efficiency

---

🎯 Interview Tips

What Interviewers Look For:

1. **Problem decomposition**: Can you break down the problem?
2. **Constraint identification**: Do you understand the rules?
3. **Backtracking pattern**: Can you implement choose-explore-unchoose?
4. **Base cases**: Do you handle termination correctly?
5. **Optimization**: Can you prune invalid paths early?

Common Interview Patterns:

- **Generate all**: Permutations, combinations, subsets
- **Find valid**: N-Queens, Sudoku, valid arrangements
- **Path finding**: Word search, maze solving
- **Constraint satisfaction**: Scheduling, assignment problems

Red Flags to Avoid:

- Not implementing proper backtracking (missing unchoose step)
- Incorrect base cases or termination conditions
- Not handling duplicates properly
- Missing constraint validation
- Inefficient pruning or no pruning at all

Pro Tips:

1. **Start with brute force**: Identify all possible choices
2. **Add constraints**: Implement validity checks
3. **Implement backtracking**: Choose-explore-unchoose pattern
4. **Optimize with pruning**: Eliminate invalid paths early
5. **Handle edge cases**: Empty inputs, single elements
6. **Practice templates**: Master the backtracking framework

🚀 Key Takeaways

1. **Backtracking explores all possibilities systematically** - Choose, explore, unchoose
2. **Early pruning is crucial** - Eliminate invalid paths to improve efficiency
3. **State management matters** - Properly track and restore state
4. **Constraint validation is key** - Check validity before exploring
5. **Template approach works** - Master the general backtracking pattern
6. **Practice problem recognition** - Learn to identify backtracking problems

Next Chapter: We'll explore Simple Greedy Algorithms and learn how to make locally optimal choices that lead to globally optimal solutions.

Chapter 15: Simple Greedy Algorithms - Making Locally Optimal Choices

🎯 What are Greedy Algorithms?

Greedy Algorithms make locally optimal choices at each step, hoping to find a globally optimal solution. The key insight is that for certain problems, making the best choice at each moment leads to the best overall solution.

Why Greedy Algorithms Matter:

- **Simplicity**: Often easier to understand and implement than DP
- **Efficiency**: Usually have better time complexity than exhaustive search
- **Real-world applications**: Scheduling, resource allocation, networking
- **Building intuition**: Helps develop algorithmic thinking
- **Interview frequency**: Common in coding interviews
- **Foundation**: Basis for more complex optimization algorithms

Core Principles:

1. **Greedy Choice Property**: Local optimum leads to global optimum
2. **Optimal Substructure**: Optimal solution contains optimal subsolutions
3. **No backtracking**: Once a choice is made, it's never reconsidered

---

📊 Greedy vs Other Approaches

Comparison Table:

Aspect	Greedy	Dynamic Programming	Backtracking
Strategy	Local optimum	Global optimum via subproblems	Explore all possibilities
Backtracking	Never	No (bottom-up)	Always
Time Complexity	Usually O(n log n)	Usually $O(n^2)$ or $O(n^3)$	Usually exponential
Space Complexity	Usually O(1)	Usually O(n) or $O(n^2)$	Usually O(depth)
Guarantee	Not always optimal	Always optimal (if applicable)	Always finds solution

Aspect	Greedy	Dynamic Programming	Backtracking
Examples	Activity selection	Knapsack	N-Queens

When Greedy Works:

- ✅ **Greedy choice property holds**: Local optimum → global optimum
- ✅ **Optimal substructure exists**: Problem can be broken down
- ✅ **No dependencies**: Current choice doesn't affect future optimality
- ✅ **Sorting helps**: Often involves sorting by some criteria

When Greedy Fails:

- ❌ **Coin change with arbitrary denominations**: [1,3,4] for amount 6
- ❌ **0/1 Knapsack**: Need to consider all combinations
- ❌ **Shortest path with negative weights**: Need to consider all paths

---

💻 JavaScript Implementation

```
// Greedy Algorithms - Comprehensive Implementation

// ===== CLASSIC GREEDY PROBLEMS =====

/**
 * Activity Selection Problem
 * Problem: Select maximum number of non-overlapping activities
 * Greedy Strategy: Always pick activity that finishes earliest
 */
function activitySelection(activities) {
  // Sort by finish time
  activities.sort((a, b) => a.finish - b.finish);

  const selected = [];
  let lastFinishTime = 0;

  for (let activity of activities) {
    // If activity starts after last selected activity finishes
    if (activity.start >= lastFinishTime) {
      selected.push(activity);
      lastFinishTime = activity.finish;
    }
  }

  return selected;
}

/**
 * Fractional Knapsack
 * Problem: Maximize value with weight constraint (can take fractions)
 * Greedy Strategy: Sort by value-to-weight ratio, take highest ratios first
```

```
 */
function fractionalKnapsack(items, capacity) {
  // Calculate value-to-weight ratio and sort
  const itemsWithRatio = items.map((item, index) => ({
    ...item,
    ratio: item.value / item.weight,
    index,
  }));

  itemsWithRatio.sort((a, b) => b.ratio - a.ratio);

  let totalValue = 0;
  let remainingCapacity = capacity;
  const solution = [];

  for (let item of itemsWithRatio) {
    if (remainingCapacity === 0) break;

    if (item.weight <= remainingCapacity) {
      // Take entire item
      solution.push({ ...item, fraction: 1 });
      totalValue += item.value;
      remainingCapacity -= item.weight;
    } else {
      // Take fraction of item
      const fraction = remainingCapacity / item.weight;
      solution.push({ ...item, fraction });
      totalValue += item.value * fraction;
      remainingCapacity = 0;
    }
  }

  return { totalValue, solution };
}

/**
 * Coin Change (Greedy - works for standard denominations)
 * Problem: Make change using minimum number of coins
 * Greedy Strategy: Use largest denomination first
 * Note: Only works for canonical coin systems (like US coins)
 */
function coinChangeGreedy(coins, amount) {
  coins.sort((a, b) => b - a); // Sort in descending order

  const result = [];
  let remaining = amount;

  for (let coin of coins) {
    while (remaining >= coin) {
      result.push(coin);
      remaining -= coin;
    }
  }
```

```
  return remaining === 0 ? result : null; // null if can't make exact change
}

/**
 * Job Scheduling with Deadlines
 * Problem: Schedule jobs to maximize profit within deadlines
 * Greedy Strategy: Sort by profit, schedule highest profit jobs first
 */
function jobScheduling(jobs) {
  // Sort jobs by profit in descending order
  jobs.sort((a, b) => b.profit - a.profit);

  const maxDeadline = Math.max(...jobs.map((job) => job.deadline));
  const schedule = new Array(maxDeadline).fill(null);
  const selectedJobs = [];
  let totalProfit = 0;

  for (let job of jobs) {
    // Find latest available slot before deadline
    for (
      let slot = Math.min(job.deadline - 1, maxDeadline - 1);
      slot >= 0;
      slot--
    ) {
      if (schedule[slot] === null) {
        schedule[slot] = job;
        selectedJobs.push(job);
        totalProfit += job.profit;
        break;
      }
    }
  }

  return { selectedJobs, totalProfit, schedule };
}

/**
 * Huffman Coding
 * Problem: Create optimal prefix-free binary codes
 * Greedy Strategy: Always merge two nodes with smallest frequencies
 */
class HuffmanNode {
  constructor(char, freq, left = null, right = null) {
    this.char = char;
    this.freq = freq;
    this.left = left;
    this.right = right;
  }

  isLeaf() {
    return this.left === null && this.right === null;
  }
}
```

```
class MinHeap {
  constructor() {
    this.heap = [];
  }

  insert(node) {
    this.heap.push(node);
    this.heapifyUp(this.heap.length - 1);
  }

  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.heapifyDown(0);
    return min;
  }

  heapifyUp(index) {
    while (index > 0) {
      const parentIndex = Math.floor((index - 1) / 2);
      if (this.heap[parentIndex].freq <= this.heap[index].freq) break;

      [this.heap[parentIndex], this.heap[index]] = [
        this.heap[index],
        this.heap[parentIndex],
      ];
      index = parentIndex;
    }
  }

  heapifyDown(index) {
    while (true) {
      let smallest = index;
      const leftChild = 2 * index + 1;
      const rightChild = 2 * index + 2;

      if (
        leftChild < this.heap.length &&
        this.heap[leftChild].freq < this.heap[smallest].freq
      ) {
        smallest = leftChild;
      }

      if (
        rightChild < this.heap.length &&
        this.heap[rightChild].freq < this.heap[smallest].freq
      ) {
        smallest = rightChild;
      }

      if (smallest === index) break;
```

```
      [this.heap[index], this.heap[smallest]] = [
        this.heap[smallest],
        this.heap[index],
      ];
      index = smallest;
    }
  }

  size() {
    return this.heap.length;
  }
}

function huffmanCoding(text) {
  // Count character frequencies
  const freqMap = {};
  for (let char of text) {
    freqMap[char] = (freqMap[char] || 0) + 1;
  }

  // Create min heap with character nodes
  const heap = new MinHeap();
  for (let [char, freq] of Object.entries(freqMap)) {
    heap.insert(new HuffmanNode(char, freq));
  }

  // Build Huffman tree
  while (heap.size() > 1) {
    const left = heap.extractMin();
    const right = heap.extractMin();
    const merged = new HuffmanNode(null, left.freq + right.freq, left, right);
    heap.insert(merged);
  }

  const root = heap.extractMin();

  // Generate codes
  const codes = {};

  function generateCodes(node, code = "") {
    if (node.isLeaf()) {
      codes[node.char] = code || "0"; // Handle single character case
      return;
    }

    if (node.left) generateCodes(node.left, code + "0");
    if (node.right) generateCodes(node.right, code + "1");
  }

  generateCodes(root);

  // Encode text
  const encoded = text
```

```
    .split("")
    .map((char) => codes[char])
    .join("");

  return { codes, encoded, root };
}

/**
 * Minimum Spanning Tree - Kruskal's Algorithm
 * Problem: Find minimum cost to connect all vertices
 * Greedy Strategy: Sort edges by weight, add edge if it doesn't create cycle
 */
class UnionFind {
  constructor(n) {
    this.parent = Array.from({ length: n }, (_, i) => i);
    this.rank = new Array(n).fill(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; // Already connected

    // 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;
  }
}

function kruskalMST(vertices, edges) {
  // Sort edges by weight
  edges.sort((a, b) => a.weight - b.weight);

  const uf = new UnionFind(vertices);
  const mst = [];
  let totalWeight = 0;

  for (let edge of edges) {
```

```
      if (uf.union(edge.from, edge.to)) {
        mst.push(edge);
        totalWeight += edge.weight;

        // MST complete when we have V-1 edges
        if (mst.length === vertices - 1) break;
      }
    }

    return { mst, totalWeight };
}

/**
 * Minimum Spanning Tree - Prim's Algorithm
 * Problem: Find minimum cost to connect all vertices
 * Greedy Strategy: Start from vertex, always add minimum weight edge to new
vertex
 */
function primMST(graph) {
    const vertices = graph.length;
    const visited = new Array(vertices).fill(false);
    const minEdge = new Array(vertices).fill(Infinity);
    const parent = new Array(vertices).fill(-1);

    minEdge[0] = 0; // Start from vertex 0
    const mst = [];
    let totalWeight = 0;

    for (let count = 0; count < vertices; count++) {
      // Find minimum weight edge to unvisited vertex
      let u = -1;
      for (let v = 0; v < vertices; v++) {
        if (!visited[v] && (u === -1 || minEdge[v] < minEdge[u])) {
          u = v;
        }
      }

      visited[u] = true;

      if (parent[u] !== -1) {
        mst.push({ from: parent[u], to: u, weight: minEdge[u] });
        totalWeight += minEdge[u];
      }

      // Update minimum edges to adjacent vertices
      for (let v = 0; v < vertices; v++) {
        if (!visited[v] && graph[u][v] < minEdge[v]) {
          minEdge[v] = graph[u][v];
          parent[v] = u;
        }
      }
    }

    return { mst, totalWeight };
```

```
}

/**
 * Dijkstra's Shortest Path
 * Problem: Find shortest path from source to all vertices
 * Greedy Strategy: Always process vertex with minimum distance
 */
function dijkstra(graph, source) {
  const vertices = graph.length;
  const dist = new Array(vertices).fill(Infinity);
  const visited = new Array(vertices).fill(false);
  const parent = new Array(vertices).fill(-1);

  dist[source] = 0;

  for (let count = 0; count < vertices; count++) {
    // Find unvisited vertex with minimum distance
    let u = -1;
    for (let v = 0; v < vertices; v++) {
      if (!visited[v] && (u === -1 || dist[v] < dist[u])) {
        u = v;
      }
    }

    visited[u] = true;

    // Update distances to adjacent vertices
    for (let v = 0; v < vertices; v++) {
      if (!visited[v] && graph[u][v] !== 0 && dist[u] + graph[u][v] < dist[v]) {
        dist[v] = dist[u] + graph[u][v];
        parent[v] = u;
      }
    }
  }

  return { distances: dist, parents: parent };
}

// ===== INTERVAL PROBLEMS =====

/**
 * Meeting Rooms II
 * Problem: Find minimum number of meeting rooms needed
 * Greedy Strategy: Sort by start time, use heap to track end times
 */
function minMeetingRooms(intervals) {
  if (intervals.length === 0) return 0;

  // Sort by start time
  intervals.sort((a, b) => a.start - b.start);

  // Min heap to track end times of ongoing meetings
  const endTimes = [];
```

```
  function insertHeap(time) {
    endTimes.push(time);
    let i = endTimes.length - 1;
    while (i > 0) {
      const parent = Math.floor((i - 1) / 2);
      if (endTimes[parent] <= endTimes[i]) break;
      [endTimes[parent], endTimes[i]] = [endTimes[i], endTimes[parent]];
      i = parent;
    }
  }

  function extractMin() {
    if (endTimes.length === 1) return endTimes.pop();
    const min = endTimes[0];
    endTimes[0] = endTimes.pop();

    let i = 0;
    while (true) {
      let smallest = i;
      const left = 2 * i + 1;
      const right = 2 * i + 2;

      if (left < endTimes.length && endTimes[left] < endTimes[smallest]) {
        smallest = left;
      }
      if (right < endTimes.length && endTimes[right] < endTimes[smallest]) {
        smallest = right;
      }

      if (smallest === i) break;
      [endTimes[i], endTimes[smallest]] = [endTimes[smallest], endTimes[i]];
      i = smallest;
    }

    return min;
  }

  for (let interval of intervals) {
    // If current meeting starts after earliest ending meeting
    if (endTimes.length > 0 && interval.start >= endTimes[0]) {
      extractMin(); // Free up a room
    }
    insertHeap(interval.end); // Allocate room for current meeting
  }

  return endTimes.length;
}

/**
 * Non-overlapping Intervals
 * Problem: Remove minimum intervals to make rest non-overlapping
 * Greedy Strategy: Sort by end time, keep intervals that end earliest
 */
function eraseOverlapIntervals(intervals) {
```

```
  if (intervals.length <= 1) return 0;

  // Sort by end time
  intervals.sort((a, b) => a[1] - b[1]);

  let count = 0;
  let lastEnd = intervals[0][1];

  for (let i = 1; i < intervals.length; i++) {
    if (intervals[i][0] < lastEnd) {
      // Overlapping interval, remove it
      count++;
    } else {
      // Non-overlapping, update last end time
      lastEnd = intervals[i][1];
    }
  }

  return count;
}

// ===== STRING PROBLEMS =====

/**
 * Remove K Digits
 * Problem: Remove k digits to make smallest possible number
 * Greedy Strategy: Remove digits that make number larger (monotonic stack)
 */
function removeKdigits(num, k) {
  const stack = [];
  let toRemove = k;

  for (let digit of num) {
    // Remove larger digits from stack
    while (
      toRemove > 0 &&
      stack.length > 0 &&
      stack[stack.length - 1] > digit
    ) {
      stack.pop();
      toRemove--;
    }
    stack.push(digit);
  }

  // Remove remaining digits from end
  while (toRemove > 0) {
    stack.pop();
    toRemove--;
  }

  // Build result, removing leading zeros
  const result = stack.join("").replace(/^0+/, "");
  return result === "" ? "0" : result;
```

```
}

/**
 * Gas Station
 * Problem: Find starting gas station to complete circular route
 * Greedy Strategy: If total gas >= total cost, solution exists
 */
function canCompleteCircuit(gas, cost) {
  let totalGas = 0;
  let totalCost = 0;
  let currentGas = 0;
  let start = 0;

  for (let i = 0; i < gas.length; i++) {
    totalGas += gas[i];
    totalCost += cost[i];
    currentGas += gas[i] - cost[i];

    // If we can't reach next station, start from next station
    if (currentGas < 0) {
      start = i + 1;
      currentGas = 0;
    }
  }

  return totalGas >= totalCost ? start : -1;
}

// ===== UTILITY FUNCTIONS =====

/**
 * Greedy Algorithm Validator
 * Checks if greedy choice property holds for a problem
 */
function validateGreedyChoice(problem, testCases) {
  console.log(`Validating greedy choice for: ${problem}`);

  for (let testCase of testCases) {
    const greedyResult = testCase.greedyFunction(...testCase.input);
    const optimalResult = testCase.optimalFunction(...testCase.input);

    const isValid =
      JSON.stringify(greedyResult) === JSON.stringify(optimalResult);
    console.log(`Test case ${testCase.name}: ${isValid ? "PASS" : "FAIL"}`);

    if (!isValid) {
      console.log(`  Greedy: ${JSON.stringify(greedyResult)}`);
      console.log(`  Optimal: ${JSON.stringify(optimalResult)}`);
    }
  }
}

/**
 * Performance Comparison
```

```
 */
function compareAlgorithmPerformance(algorithms, input) {
  console.log("=== Algorithm Performance Comparison ===");

  for (let algo of algorithms) {
    console.time(algo.name);
    const result = algo.function(...input);
    console.timeEnd(algo.name);
    console.log(`${algo.name} result:`, result);
    console.log();
  }
}

/**
 * Greedy Problem Classifier
 */
function classifyGreedyProblem(description) {
  const patterns = {
    scheduling: ["activity", "job", "meeting", "interval"],
    graph: ["spanning tree", "shortest path", "minimum cost"],
    optimization: ["maximum", "minimum", "optimal"],
    selection: ["choose", "select", "pick"],
    partitioning: ["partition", "divide", "split"],
    encoding: ["huffman", "compression", "coding"],
  };

  for (let [category, keywords] of Object.entries(patterns)) {
    if (
      keywords.some((keyword) => description.toLowerCase().includes(keyword))
    ) {
      return category;
    }
  }

  return "unknown";
}

// ===== EXAMPLE USAGE AND TESTING =====

console.log("=== Greedy Algorithms Demo ===");

// Test Activity Selection
console.log("\n=== Activity Selection ===");
const activities = [
  { name: "A1", start: 1, finish: 4 },
  { name: "A2", start: 3, finish: 5 },
  { name: "A3", start: 0, finish: 6 },
  { name: "A4", start: 5, finish: 7 },
  { name: "A5", start: 8, finish: 9 },
  { name: "A6", start: 5, finish: 9 },
];
const selectedActivities = activitySelection(activities);
console.log(
  "Selected activities:",
```

```
  selectedActivities.map((a) => a.name)
);

// Test Fractional Knapsack
console.log("\n=== Fractional Knapsack ===");
const items = [
  { value: 60, weight: 10 },
  { value: 100, weight: 20 },
  { value: 120, weight: 30 },
];
const knapsackResult = fractionalKnapsack(items, 50);
console.log("Fractional Knapsack result:", knapsackResult);

// Test Coin Change
console.log("\n=== Coin Change (Greedy) ===");
const coins = [25, 10, 5, 1];
console.log("Change for 67:", coinChangeGreedy(coins, 67));
console.log("Change for 43:", coinChangeGreedy(coins, 43));

// Test Job Scheduling
console.log("\n=== Job Scheduling ===");
const jobs = [
  { id: "J1", deadline: 4, profit: 20 },
  { id: "J2", deadline: 1, profit: 10 },
  { id: "J3", deadline: 1, profit: 40 },
  { id: "J4", deadline: 1, profit: 30 },
];
const jobResult = jobScheduling(jobs);
console.log("Job scheduling result:", jobResult);

// Test Huffman Coding
console.log("\n=== Huffman Coding ===");
const text = "hello world";
const huffmanResult = huffmanCoding(text);
console.log("Huffman codes:", huffmanResult.codes);
console.log("Encoded text:", huffmanResult.encoded);
console.log(
  "Compression ratio:",
  (huffmanResult.encoded.length / (text.length * 8)).toFixed(2)
);

// Test MST
console.log("\n=== Minimum Spanning Tree ===");
const edges = [
  { from: 0, to: 1, weight: 10 },
  { from: 0, to: 2, weight: 6 },
  { from: 0, to: 3, weight: 5 },
  { from: 1, to: 3, weight: 15 },
  { from: 2, to: 3, weight: 4 },
];
const mstResult = kruskalMST(4, edges);
console.log("Kruskal MST:", mstResult);

// Test Dijkstra
```

```
console.log("\n=== Dijkstra Shortest Path ===");
const graph = [
  [0, 4, 0, 0, 0, 0, 0, 8, 0],
  [4, 0, 8, 0, 0, 0, 0, 11, 0],
  [0, 8, 0, 7, 0, 4, 0, 0, 2],
  [0, 0, 7, 0, 9, 14, 0, 0, 0],
  [0, 0, 0, 9, 0, 10, 0, 0, 0],
  [0, 0, 4, 14, 10, 0, 2, 0, 0],
  [0, 0, 0, 0, 0, 2, 0, 1, 6],
  [8, 11, 0, 0, 0, 0, 1, 0, 7],
  [0, 0, 2, 0, 0, 0, 6, 7, 0],
];
const dijkstraResult = dijkstra(graph, 0);
console.log("Shortest distances from vertex 0:", dijkstraResult.distances);

// Test Interval Problems
console.log("\n=== Interval Problems ===");
const meetings = [
  { start: 0, end: 30 },
  { start: 5, end: 10 },
  { start: 15, end: 20 },
];
console.log("Minimum meeting rooms needed:", minMeetingRooms(meetings));

const intervals = [
  [1, 2],
  [2, 3],
  [3, 4],
  [1, 3],
];
console.log("Intervals to remove:", eraseOverlapIntervals(intervals));

// Test String Problems
console.log("\n=== String Problems ===");
console.log('Remove 3 digits from "1432219":', removeKdigits("1432219", 3));
console.log('Remove 1 digit from "10200":', removeKdigits("10200", 1));

const gas = [1, 2, 3, 4, 5];
const cost = [3, 4, 5, 1, 2];
console.log("Gas station starting point:", canCompleteCircuit(gas, cost));

// Problem classification
console.log("\n=== Problem Classification ===");
console.log(
  '"Activity selection problem":',
  classifyGreedyProblem("Activity selection problem")
);
console.log(
  '"Find minimum spanning tree":',
  classifyGreedyProblem("Find minimum spanning tree")
);
console.log(
  '"Choose maximum profit jobs":',
  classifyGreedyProblem("Choose maximum profit jobs")
```

```
);
console.log(
  '"Huffman encoding algorithm":',
  classifyGreedyProblem("Huffman encoding algorithm")
);

console.log("\n=== Greedy Strategy Guide ===");
console.log("1. Identify the greedy choice property");
console.log("2. Prove that greedy choice leads to optimal solution");
console.log("3. Sort input by appropriate criteria");
console.log("4. Make locally optimal choices");
console.log("5. Verify solution optimality");
```

🔧 C++ Implementation

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <queue>
#include <unordered_map>
#include <string>
#include <climits>
using namespace std;

// ===== CLASSIC GREEDY PROBLEMS =====

// Activity Selection
struct Activity {
    string name;
    int start, finish;
};

vector<Activity> activitySelection(vector<Activity> activities) {
    // Sort by finish time
    sort(activities.begin(), activities.end(),
         [](const Activity& a, const Activity& b) {
             return a.finish < b.finish;
         });

    vector<Activity> selected;
    int lastFinishTime = 0;

    for (const auto& activity : activities) {
        if (activity.start >= lastFinishTime) {
            selected.push_back(activity);
            lastFinishTime = activity.finish;
        }
    }

    return selected;
```

```
}

// Fractional Knapsack
struct Item {
    int value, weight;
    double ratio;
};

struct KnapsackResult {
    double totalValue;
    vector<pair<Item, double>> solution; // item and fraction taken
};

KnapsackResult fractionalKnapsack(vector<Item> items, int capacity) {
    // Calculate ratios
    for (auto& item : items) {
        item.ratio = (double)item.value / item.weight;
    }

    // Sort by ratio in descending order
    sort(items.begin(), items.end(),
         [](const Item& a, const Item& b) {
             return a.ratio > b.ratio;
         });

    KnapsackResult result;
    result.totalValue = 0;
    int remainingCapacity = capacity;

    for (const auto& item : items) {
        if (remainingCapacity == 0) break;

        if (item.weight <= remainingCapacity) {
            // Take entire item
            result.solution.push_back({item, 1.0});
            result.totalValue += item.value;
            remainingCapacity -= item.weight;
        } else {
            // Take fraction
            double fraction = (double)remainingCapacity / item.weight;
            result.solution.push_back({item, fraction});
            result.totalValue += item.value * fraction;
            remainingCapacity = 0;
        }
    }

    return result;
}

// Job Scheduling
struct Job {
    string id;
    int deadline, profit;
};
```

```
struct JobResult {
    vector<Job> selectedJobs;
    int totalProfit;
    vector<Job> schedule;
};

JobResult jobScheduling(vector<Job> jobs) {
    // Sort by profit in descending order
    sort(jobs.begin(), jobs.end(),
         [](const Job& a, const Job& b) {
             return a.profit > b.profit;
         });

    int maxDeadline = 0;
    for (const auto& job : jobs) {
        maxDeadline = max(maxDeadline, job.deadline);
    }

    vector<Job> schedule(maxDeadline);
    vector<bool> occupied(maxDeadline, false);
    JobResult result;
    result.totalProfit = 0;

    for (const auto& job : jobs) {
        // Find latest available slot before deadline
        for (int slot = min(job.deadline - 1, maxDeadline - 1); slot >= 0; slot--)
{
            if (!occupied[slot]) {
                schedule[slot] = job;
                occupied[slot] = true;
                result.selectedJobs.push_back(job);
                result.totalProfit += job.profit;
                break;
            }
        }
    }

    result.schedule = schedule;
    return result;
}

// Union-Find for Kruskal's Algorithm
class UnionFind {
public:
    vector<int> parent, rank;

    UnionFind(int n) : parent(n), rank(n, 0) {
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
    }

    int find(int x) {
```

```
        if (parent[x] != x) {
            parent[x] = find(parent[x]); // Path compression
        }
        return parent[x];
    }

    bool unite(int x, int y) {
        int rootX = find(x), rootY = find(y);
        if (rootX == rootY) return false;

        // Union by rank
        if (rank[rootX] < rank[rootY]) {
            parent[rootX] = rootY;
        } else if (rank[rootX] > rank[rootY]) {
            parent[rootY] = rootX;
        } else {
            parent[rootY] = rootX;
            rank[rootX]++;
        }

        return true;
    }
};

// Kruskal's MST
struct Edge {
    int from, to, weight;
};

struct MSTResult {
    vector<Edge> mst;
    int totalWeight;
};

MSTResult kruskalMST(int vertices, vector<Edge> edges) {
    // Sort edges by weight
    sort(edges.begin(), edges.end(),
         [](const Edge& a, const Edge& b) {
             return a.weight < b.weight;
         });

    UnionFind uf(vertices);
    MSTResult result;
    result.totalWeight = 0;

    for (const auto& edge : edges) {
        if (uf.unite(edge.from, edge.to)) {
            result.mst.push_back(edge);
            result.totalWeight += edge.weight;

            if (result.mst.size() == vertices - 1) break;
        }
    }
```

```
    return result;
}

// Dijkstra's Algorithm
vector<int> dijkstra(vector<vector<int>>& graph, int source) {
    int vertices = graph.size();
    vector<int> dist(vertices, INT_MAX);
    vector<bool> visited(vertices, false);

    dist[source] = 0;

    for (int count = 0; count < vertices; count++) {
        // Find unvisited vertex with minimum distance
        int u = -1;
        for (int v = 0; v < vertices; v++) {
            if (!visited[v] && (u == -1 || dist[v] < dist[u])) {
                u = v;
            }
        }

        visited[u] = true;

        // Update distances
        for (int v = 0; v < vertices; v++) {
            if (!visited[v] && graph[u][v] != 0 &&
                dist[u] != INT_MAX && dist[u] + graph[u][v] < dist[v]) {
                dist[v] = dist[u] + graph[u][v];
            }
        }
    }

    return dist;
}

// Meeting Rooms II
struct Interval {
    int start, end;
};

int minMeetingRooms(vector<Interval> intervals) {
    if (intervals.empty()) return 0;

    // Sort by start time
    sort(intervals.begin(), intervals.end(),
         [](const Interval& a, const Interval& b) {
             return a.start < b.start;
         });

    // Min heap for end times
    priority_queue<int, vector<int>, greater<int>> endTimes;

    for (const auto& interval : intervals) {
        // If current meeting starts after earliest ending meeting
        if (!endTimes.empty() && interval.start >= endTimes.top()) {
```

```
            endTimes.pop(); // Free up a room
        }
        endTimes.push(interval.end); // Allocate room
    }

    return endTimes.size();
}

// Remove K Digits
string removeKdigits(string num, int k) {
    string stack;
    int toRemove = k;

    for (char digit : num) {
        // Remove larger digits
        while (toRemove > 0 && !stack.empty() && stack.back() > digit) {
            stack.pop_back();
            toRemove--;
        }
        stack.push_back(digit);
    }

    // Remove remaining digits from end
    while (toRemove > 0) {
        stack.pop_back();
        toRemove--;
    }

    // Remove leading zeros
    int start = 0;
    while (start < stack.length() && stack[start] == '0') {
        start++;
    }

    string result = stack.substr(start);
    return result.empty() ? "0" : result;
}

// Gas Station
int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {
    int totalGas = 0, totalCost = 0, currentGas = 0, start = 0;

    for (int i = 0; i < gas.size(); i++) {
        totalGas += gas[i];
        totalCost += cost[i];
        currentGas += gas[i] - cost[i];

        if (currentGas < 0) {
            start = i + 1;
            currentGas = 0;
        }
    }

    return totalGas >= totalCost ? start : -1;
```

```
}

// ===== UTILITY FUNCTIONS =====

template<typename T>
void printVector(const vector<T>& vec, const string& label = "") {
    if (!label.empty()) {
        cout << label << ": ";
    }
    for (const auto& x : vec) {
        cout << x << " ";
    }
    cout << endl;
}

void printActivities(const vector<Activity>& activities) {
    for (const auto& activity : activities) {
        cout << activity.name << "(" << activity.start << "-" << activity.finish
<< ") ";
    }
    cout << endl;
}

void printJobs(const vector<Job>& jobs) {
    for (const auto& job : jobs) {
        cout << job.id << "(deadline:" << job.deadline << ", profit:" <<
job.profit << ") ";
    }
    cout << endl;
}

void printEdges(const vector<Edge>& edges) {
    for (const auto& edge : edges) {
        cout << edge.from << "-" << edge.to << "(" << edge.weight << ") ";
    }
    cout << endl;
}

// Example Usage
int main() {
    cout << "=== Greedy Algorithms Demo ===" << endl;

    // Test Activity Selection
    cout << "\n=== Activity Selection ===" << endl;
    vector<Activity> activities = {
        {"A1", 1, 4}, {"A2", 3, 5}, {"A3", 0, 6},
        {"A4", 5, 7}, {"A5", 8, 9}, {"A6", 5, 9}
    };
    auto selected = activitySelection(activities);
    cout << "Selected activities: ";
    printActivities(selected);

    // Test Fractional Knapsack
    cout << "\n=== Fractional Knapsack ===" << endl;
```

```
    vector<Item> items = {{60, 10, 0}, {100, 20, 0}, {120, 30, 0}};
    auto knapsackResult = fractionalKnapsack(items, 50);
    cout << "Total value: " << knapsackResult.totalValue << endl;

    // Test Job Scheduling
    cout << "\n=== Job Scheduling ===" << endl;
    vector<Job> jobs = {
        {"J1", 4, 20}, {"J2", 1, 10}, {"J3", 1, 40}, {"J4", 1, 30}
    };
    auto jobResult = jobScheduling(jobs);
    cout << "Selected jobs: ";
    printJobs(jobResult.selectedJobs);
    cout << "Total profit: " << jobResult.totalProfit << endl;

    // Test Kruskal's MST
    cout << "\n=== Kruskal's MST ===" << endl;
    vector<Edge> edges = {
        {0, 1, 10}, {0, 2, 6}, {0, 3, 5}, {1, 3, 15}, {2, 3, 4}
    };
    auto mstResult = kruskalMST(4, edges);
    cout << "MST edges: ";
    printEdges(mstResult.mst);
    cout << "Total weight: " << mstResult.totalWeight << endl;

    // Test Dijkstra
    cout << "\n=== Dijkstra's Algorithm ===" << endl;
    vector<vector<int>> graph = {
        {0, 4, 0, 0, 0, 0, 0, 8, 0},
        {4, 0, 8, 0, 0, 0, 0, 11, 0},
        {0, 8, 0, 7, 0, 4, 0, 0, 2},
        {0, 0, 7, 0, 9, 14, 0, 0, 0},
        {0, 0, 0, 9, 0, 10, 0, 0, 0},
        {0, 0, 4, 14, 10, 0, 2, 0, 0},
        {0, 0, 0, 0, 0, 2, 0, 1, 6},
        {8, 11, 0, 0, 0, 0, 1, 0, 7},
        {0, 0, 2, 0, 0, 0, 6, 7, 0}
    };
    auto distances = dijkstra(graph, 0);
    cout << "Shortest distances from vertex 0: ";
    printVector(distances);

    // Test Meeting Rooms
    cout << "\n=== Meeting Rooms ===" << endl;
    vector<Interval> meetings = {{0, 30}, {5, 10}, {15, 20}};
    cout << "Minimum meeting rooms: " << minMeetingRooms(meetings) << endl;

    // Test Remove K Digits
    cout << "\n=== Remove K Digits ===" << endl;
    cout << "Remove 3 digits from '1432219': " << removeKdigits("1432219", 3) <<
endl;
    cout << "Remove 1 digit from '10200': " << removeKdigits("10200", 1) << endl;

    // Test Gas Station
    cout << "\n=== Gas Station ===" << endl;
```

```
    vector<int> gas = {1, 2, 3, 4, 5};
    vector<int> cost = {3, 4, 5, 1, 2};
    cout << "Starting gas station: " << canCompleteCircuit(gas, cost) << endl;

    return 0;
}
```

⚡ Performance Analysis

Time Complexity Comparison:

Problem	Greedy Time	Optimal Time	Space	Notes
Activity Selection	O(n log n)	$O(2^n)$	O(1)	Sorting dominates
Fractional Knapsack	O(n log n)	O(n log n)	O(1)	Greedy is optimal
Coin Change	O(n)	O(amount × coins)	O(1)	Only for canonical systems
Job Scheduling	O(n log n)	$O(2^n)$	O(n)	Sorting + scheduling
MST (Kruskal)	O(E log E)	O(E log E)	O(V)	Greedy is optimal
MST (Prim)	$O(V^2)$	$O(V^2)$	O(V)	With adjacency matrix
Shortest Path	$O(V^2)$	$O(V^2)$	O(V)	Dijkstra is optimal

Space Complexity:

- **Most greedy algorithms**: O(1) to O(n) auxiliary space
- **Graph algorithms**: O(V + E) for adjacency representation
- **Heap-based algorithms**: O(n) for priority queue

When Greedy is Optimal:

1. **Greedy choice property**: Local optimum leads to global optimum
2. **Optimal substructure**: Problem exhibits optimal substructure
3. **No future dependencies**: Current choice doesn't affect future optimality

---

🧩 Practice Problems

Problem 1: Jump Game II

Question: Find minimum jumps to reach end of array. **Example**: `[2,3,1,1,4]` → 2 (jump 1 step from index 0 to 1, then 3 steps to last index) **Hint**: Greedy - always jump to position that allows furthest reach

Problem 2: Candy Distribution

Question: Distribute minimum candies to children with rating constraints. **Example**: `[1,0,2]` → 5 (give [2,1,2] candies) **Hint**: Two passes - left to right, then right to left

Problem 3: Task Scheduler

Question: Find minimum time to execute tasks with cooldown period. **Example**: `['A','A','A','B','B','B']`, n=2 → 8 **Hint**: Greedy scheduling with most frequent tasks first

Problem 4: Minimum Number of Arrows

Question: Find minimum arrows to burst all balloons. **Example**: `[[10,16],[2,8],[1,6],[7,12]]` → 2 **Hint**: Sort by end position, shoot arrow at earliest end

---

🎯 Interview Tips

What Interviewers Look For:

1. **Problem recognition**: Can you identify when greedy works?
2. **Proof of correctness**: Can you prove greedy choice property?
3. **Implementation**: Clean, efficient code
4. **Edge cases**: Handle empty inputs, single elements
5. **Optimization**: Can you optimize time/space complexity?

Common Interview Patterns:

- **Interval problems**: Activity selection, meeting rooms
- **Graph problems**: MST, shortest path
- **Optimization problems**: Knapsack variants, scheduling
- **String problems**: Parentheses, digit removal
- **Array problems**: Jump game, candy distribution

Red Flags to Avoid:

- Using greedy when it doesn't guarantee optimal solution
- Not proving greedy choice property
- Incorrect sorting criteria
- Missing edge cases
- Inefficient implementation

Pro Tips:

1. **Identify the greedy choice**: What's the locally optimal decision?
2. **Prove correctness**: Show that greedy choice leads to optimal solution
3. **Sort when needed**: Many greedy algorithms require sorting
4. **Consider counterexamples**: Test if greedy always works
5. **Optimize implementation**: Use appropriate data structures
6. **Practice pattern recognition**: Learn common greedy problem types

---

🚀 Key Takeaways

1. **Greedy makes locally optimal choices** - Hope for globally optimal solution

2. **Not always optimal** - Must prove greedy choice property
3. **Often involves sorting** - Sort by appropriate criteria first
4. **Efficient when applicable** - Usually better time complexity than DP
5. **Pattern recognition crucial** - Learn to identify greedy problems
6. **Proof is important** - Always verify correctness

Congratulations! You've completed the Medium Complexity section of the DSA learning path. You now have a solid foundation in fundamental data structures and essential algorithms. The next step would be to work on the final mini-project that combines multiple concepts you've learned.

Next Steps:

- Review all chapters and practice problems
- Work on the comprehensive mini-project
- Apply these concepts to real-world problems
- Continue with advanced topics like advanced graph algorithms, advanced DP, and system design

Mini-Project: Task Management System with Advanced Data Structures

🎯 Project Overview

This comprehensive mini-project combines multiple data structures and algorithms concepts learned throughout the course. You'll build a **Task Management System** that demonstrates practical applications of arrays, linked lists, stacks, queues, hash tables, trees, graphs, sorting, searching, dynamic programming, backtracking, and greedy algorithms.

Learning Objectives:

- **Integration**: Combine multiple DSA concepts in a real-world application
- **Problem-solving**: Apply algorithmic thinking to practical problems
- **Optimization**: Use appropriate data structures for different operations
- **Scalability**: Design systems that can handle growing data
- **Performance**: Analyze and optimize time/space complexity

Project Features:

1. **Task Management**: Create, update, delete, and organize tasks
2. **Priority Scheduling**: Implement priority-based task scheduling
3. **Dependency Management**: Handle task dependencies using graphs
4. **Search & Filter**: Efficient searching and filtering capabilities
5. **Analytics**: Generate insights using various algorithms
6. **Optimization**: Resource allocation and scheduling optimization

---

🏗️ System Architecture

Core Components:

```
TaskManagementSystem
├── TaskManager (Hash Table + Linked List)
├── PriorityScheduler (Heap + Queue)
├── DependencyGraph (Graph + Topological Sort)
├── SearchEngine (Trie + Binary Search)
├── Analytics (Dynamic Programming)
└── Optimizer (Greedy Algorithms)
```

Data Structures Used:

- **Hash Table**: Fast task lookup and storage
- **Linked List**: Task history and undo operations
- **Stack**: Undo/Redo functionality
- **Queue**: Task processing pipeline
- **Heap**: Priority-based scheduling
- **Tree**: Hierarchical task organization
- **Graph**: Task dependencies
- **Trie**: Efficient text search

💻 Implementation

JavaScript Implementation

```javascript
// Task Management System - Comprehensive Implementation

// ===== CORE DATA STRUCTURES =====

/**
 * Task Entity
 */
class Task {
  constructor(
    id,
    title,
    description,
    priority = 1,
    estimatedTime = 1,
    tags = []
  ) {
    this.id = id;
    this.title = title;
    this.description = description;
    this.priority = priority; // 1-5 (5 = highest)
    this.estimatedTime = estimatedTime; // in hours
    this.actualTime = 0;
    this.status = "pending"; // pending, in-progress, completed, cancelled
    this.tags = new Set(tags);
    this.dependencies = new Set();
```

```
    this.dependents = new Set();
    this.createdAt = new Date();
    this.updatedAt = new Date();
    this.completedAt = null;
    this.assignee = null;
    this.deadline = null;
  }

  addDependency(taskId) {
    this.dependencies.add(taskId);
  }

  addDependent(taskId) {
    this.dependents.add(taskId);
  }

  updateStatus(status) {
    this.status = status;
    this.updatedAt = new Date();
    if (status === "completed") {
      this.completedAt = new Date();
    }
  }

  addTag(tag) {
    this.tags.add(tag);
  }

  removeTag(tag) {
    this.tags.delete(tag);
  }

  isBlocked() {
    return this.dependencies.size > 0;
  }

  getEfficiency() {
    if (this.actualTime === 0) return 1;
    return this.estimatedTime / this.actualTime;
  }
}

/**
 * Custom Hash Table for Task Storage
 */
class TaskHashTable {
  constructor(initialSize = 16) {
    this.size = initialSize;
    this.count = 0;
    this.buckets = new Array(this.size).fill(null).map(() => []);
  }

  hash(key) {
    let hash = 0;
```

```
    for (let i = 0; i < key.length; i++) {
      hash = (hash * 31 + key.charCodeAt(i)) % this.size;
    }
    return hash;
  }

  put(key, value) {
    const index = this.hash(key);
    const bucket = this.buckets[index];

    // Check if key already exists
    for (let i = 0; i < bucket.length; i++) {
      if (bucket[i][0] === key) {
        bucket[i][1] = value;
        return;
      }
    }

    // Add new key-value pair
    bucket.push([key, value]);
    this.count++;

    // Resize if load factor > 0.75
    if (this.count > this.size * 0.75) {
      this.resize();
    }
  }

  get(key) {
    const index = this.hash(key);
    const bucket = this.buckets[index];

    for (let [k, v] of bucket) {
      if (k === key) return v;
    }

    return null;
  }

  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);
        this.count--;
        return true;
      }
    }

    return false;
  }
```

```
    resize() {
      const oldBuckets = this.buckets;
      this.size *= 2;
      this.count = 0;
      this.buckets = new Array(this.size).fill(null).map(() => []);

      for (let bucket of oldBuckets) {
        for (let [key, value] of bucket) {
          this.put(key, value);
        }
      }
    }

    getAllValues() {
      const values = [];
      for (let bucket of this.buckets) {
        for (let [key, value] of bucket) {
          values.push(value);
        }
      }
      return values;
    }
  }

  /**
   * Priority Queue using Min/Max Heap
   */
  class PriorityQueue {
    constructor(compareFn = (a, b) => a.priority - b.priority) {
      this.heap = [];
      this.compare = compareFn;
    }

    enqueue(item) {
      this.heap.push(item);
      this.heapifyUp(this.heap.length - 1);
    }

    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.heapifyDown(0);
      return root;
    }

    peek() {
      return this.heap.length > 0 ? this.heap[0] : null;
    }

    heapifyUp(index) {
      while (index > 0) {
```

```
      const parentIndex = Math.floor((index - 1) / 2);
      if (this.compare(this.heap[index], this.heap[parentIndex]) >= 0) break;

      [this.heap[index], this.heap[parentIndex]] = [
        this.heap[parentIndex],
        this.heap[index],
      ];
      index = parentIndex;
    }
  }

  heapifyDown(index) {
    while (true) {
      let smallest = index;
      const leftChild = 2 * index + 1;
      const rightChild = 2 * index + 2;

      if (
        leftChild < this.heap.length &&
        this.compare(this.heap[leftChild], this.heap[smallest]) < 0
      ) {
        smallest = leftChild;
      }

      if (
        rightChild < this.heap.length &&
        this.compare(this.heap[rightChild], this.heap[smallest]) < 0
      ) {
        smallest = rightChild;
      }

      if (smallest === index) break;

      [this.heap[index], this.heap[smallest]] = [
        this.heap[smallest],
        this.heap[index],
      ];
      index = smallest;
    }
  }

  size() {
    return this.heap.length;
  }

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

/**
 * Trie for Efficient Text Search
 */
class TrieNode {
```

```
    constructor() {
      this.children = {};
      this.isEndOfWord = false;
      this.taskIds = new Set();
    }
  }

  class Trie {
    constructor() {
      this.root = new TrieNode();
    }

    insert(word, taskId) {
      let node = this.root;

      for (let char of word.toLowerCase()) {
        if (!node.children[char]) {
          node.children[char] = new TrieNode();
        }
        node = node.children[char];
        node.taskIds.add(taskId);
      }

      node.isEndOfWord = true;
    }

    search(prefix) {
      let node = this.root;

      for (let char of prefix.toLowerCase()) {
        if (!node.children[char]) {
          return [];
        }
        node = node.children[char];
      }

      return Array.from(node.taskIds);
    }

    delete(word, taskId) {
      this.deleteHelper(this.root, word.toLowerCase(), 0, taskId);
    }

    deleteHelper(node, word, index, taskId) {
      if (index === word.length) {
        node.isEndOfWord = false;
        node.taskIds.delete(taskId);
        return node.taskIds.size === 0 && Object.keys(node.children).length === 0;
      }

      const char = word[index];
      const childNode = node.children[char];

      if (!childNode) return false;
```

```
    const shouldDeleteChild = this.deleteHelper(
      childNode,
      word,
      index + 1,
      taskId
    );

    if (shouldDeleteChild) {
      delete node.children[char];
    }

    childNode.taskIds.delete(taskId);

    return (
      node.taskIds.size === 0 &&
      Object.keys(node.children).length === 0 &&
      !node.isEndOfWord
    );
  }
}

/**
 * Dependency Graph for Task Dependencies
 */
class DependencyGraph {
  constructor() {
    this.adjacencyList = new Map();
    this.inDegree = new Map();
  }

  addTask(taskId) {
    if (!this.adjacencyList.has(taskId)) {
      this.adjacencyList.set(taskId, []);
      this.inDegree.set(taskId, 0);
    }
  }

  addDependency(fromTask, toTask) {
    this.addTask(fromTask);
    this.addTask(toTask);

    this.adjacencyList.get(fromTask).push(toTask);
    this.inDegree.set(toTask, this.inDegree.get(toTask) + 1);
  }

  removeDependency(fromTask, toTask) {
    if (this.adjacencyList.has(fromTask)) {
      const neighbors = this.adjacencyList.get(fromTask);
      const index = neighbors.indexOf(toTask);
      if (index > -1) {
        neighbors.splice(index, 1);
        this.inDegree.set(toTask, this.inDegree.get(toTask) - 1);
      }
```

```
    }
  }

  getTopologicalOrder() {
    const result = [];
    const queue = [];
    const inDegreeMap = new Map(this.inDegree);

    // Find all tasks with no dependencies
    for (let [taskId, degree] of inDegreeMap) {
      if (degree === 0) {
        queue.push(taskId);
      }
    }

    while (queue.length > 0) {
      const current = queue.shift();
      result.push(current);

      // Process all dependent tasks
      for (let neighbor of this.adjacencyList.get(current) || []) {
        inDegreeMap.set(neighbor, inDegreeMap.get(neighbor) - 1);
        if (inDegreeMap.get(neighbor) === 0) {
          queue.push(neighbor);
        }
      }
    }

    // Check for cycles
    if (result.length !== this.adjacencyList.size) {
      throw new Error("Circular dependency detected!");
    }

    return result;
  }

  hasCycle() {
    try {
      this.getTopologicalOrder();
      return false;
    } catch (error) {
      return true;
    }
  }

  getReadyTasks() {
    const readyTasks = [];
    for (let [taskId, degree] of this.inDegree) {
      if (degree === 0) {
        readyTasks.push(taskId);
      }
    }
    return readyTasks;
  }
```

```
}

/**
 * Undo/Redo Stack
 */
class UndoRedoStack {
  constructor(maxSize = 50) {
    this.undoStack = [];
    this.redoStack = [];
    this.maxSize = maxSize;
  }

  execute(command) {
    command.execute();
    this.undoStack.push(command);
    this.redoStack = []; // Clear redo stack

    // Limit stack size
    if (this.undoStack.length > this.maxSize) {
      this.undoStack.shift();
    }
  }

  undo() {
    if (this.undoStack.length === 0) return false;

    const command = this.undoStack.pop();
    command.undo();
    this.redoStack.push(command);
    return true;
  }

  redo() {
    if (this.redoStack.length === 0) return false;

    const command = this.redoStack.pop();
    command.execute();
    this.undoStack.push(command);
    return true;
  }

  canUndo() {
    return this.undoStack.length > 0;
  }

  canRedo() {
    return this.redoStack.length > 0;
  }
}

// ===== COMMAND PATTERN FOR UNDO/REDO =====

class Command {
  execute() {
```

```
    throw new Error("Execute method must be implemented");
  }

  undo() {
    throw new Error("Undo method must be implemented");
  }
}

class CreateTaskCommand extends Command {
  constructor(taskManager, task) {
    super();
    this.taskManager = taskManager;
    this.task = task;
  }

  execute() {
    this.taskManager.addTaskDirect(this.task);
  }

  undo() {
    this.taskManager.deleteTaskDirect(this.task.id);
  }
}

class UpdateTaskCommand extends Command {
  constructor(taskManager, taskId, oldData, newData) {
    super();
    this.taskManager = taskManager;
    this.taskId = taskId;
    this.oldData = oldData;
    this.newData = newData;
  }

  execute() {
    this.taskManager.updateTaskDirect(this.taskId, this.newData);
  }

  undo() {
    this.taskManager.updateTaskDirect(this.taskId, this.oldData);
  }
}

class DeleteTaskCommand extends Command {
  constructor(taskManager, task) {
    super();
    this.taskManager = taskManager;
    this.task = task;
  }

  execute() {
    this.taskManager.deleteTaskDirect(this.task.id);
  }

  undo() {
```

```
      this.taskManager.addTaskDirect(this.task);
    }
  }

  // ===== MAIN TASK MANAGEMENT SYSTEM =====

  class TaskManagementSystem {
    constructor() {
      this.tasks = new TaskHashTable();
      this.priorityQueue = new PriorityQueue((a, b) => b.priority - a.priority);
      this.searchTrie = new Trie();
      this.dependencyGraph = new DependencyGraph();
      this.undoRedoStack = new UndoRedoStack();
      this.taskIdCounter = 1;
      this.analytics = new TaskAnalytics();
    }

    // ===== TASK CRUD OPERATIONS =====

    createTask(title, description, priority = 1, estimatedTime = 1, tags = []) {
      const taskId = `task_${this.taskIdCounter++}`;
      const task = new Task(
        taskId,
        title,
        description,
        priority,
        estimatedTime,
        tags
      );

      const command = new CreateTaskCommand(this, task);
      this.undoRedoStack.execute(command);

      return task;
    }

    addTaskDirect(task) {
      this.tasks.put(task.id, task);
      this.priorityQueue.enqueue(task);
      this.dependencyGraph.addTask(task.id);

      // Index for search
      this.indexTaskForSearch(task);

      // Update analytics
      this.analytics.addTask(task);
    }

    updateTask(taskId, updates) {
      const task = this.tasks.get(taskId);
      if (!task) throw new Error(`Task ${taskId} not found`);

      const oldData = { ...task };
      const command = new UpdateTaskCommand(this, taskId, oldData, updates);
```

```
    this.undoRedoStack.execute(command);

    return task;
  }

  updateTaskDirect(taskId, updates) {
    const task = this.tasks.get(taskId);
    if (!task) return;

    // Remove old search indices
    this.removeTaskFromSearch(task);

    // Update task properties
    Object.assign(task, updates);
    task.updatedAt = new Date();

    // Re-index for search
    this.indexTaskForSearch(task);

    // Update analytics
    this.analytics.updateTask(task);
  }

  deleteTask(taskId) {
    const task = this.tasks.get(taskId);
    if (!task) throw new Error(`Task ${taskId} not found`);

    const command = new DeleteTaskCommand(this, task);
    this.undoRedoStack.execute(command);

    return true;
  }

  deleteTaskDirect(taskId) {
    const task = this.tasks.get(taskId);
    if (!task) return;

    this.tasks.delete(taskId);
    this.removeTaskFromSearch(task);

    // Remove from dependency graph
    this.dependencyGraph.adjacencyList.delete(taskId);
    this.dependencyGraph.inDegree.delete(taskId);

    // Update analytics
    this.analytics.removeTask(task);
  }

  getTask(taskId) {
    return this.tasks.get(taskId);
  }

  getAllTasks() {
    return this.tasks.getAllValues();
```

```
  }

  // ===== SEARCH AND FILTERING =====

  indexTaskForSearch(task) {
    // Index title and description words
    const words = [
      ...task.title.split(/\s+/),
      ...task.description.split(/\s+/),
    ];
    for (let word of words) {
      if (word.length > 2) {
        this.searchTrie.insert(word, task.id);
      }
    }

    // Index tags
    for (let tag of task.tags) {
      this.searchTrie.insert(tag, task.id);
    }
  }

  removeTaskFromSearch(task) {
    const words = [
      ...task.title.split(/\s+/),
      ...task.description.split(/\s+/),
    ];
    for (let word of words) {
      if (word.length > 2) {
        this.searchTrie.delete(word, task.id);
      }
    }

    for (let tag of task.tags) {
      this.searchTrie.delete(tag, task.id);
    }
  }

  searchTasks(query) {
    const taskIds = this.searchTrie.search(query);
    return taskIds
      .map((id) => this.tasks.get(id))
      .filter((task) => task !== null);
  }

  filterTasks(criteria) {
    const allTasks = this.getAllTasks();

    return allTasks.filter((task) => {
      if (criteria.status && task.status !== criteria.status) return false;
      if (criteria.priority && task.priority !== criteria.priority)
        return false;
      if (criteria.assignee && task.assignee !== criteria.assignee)
        return false;
```

```
      if (criteria.tag && !task.tags.has(criteria.tag)) return false;
      if (criteria.minPriority && task.priority < criteria.minPriority)
        return false;
      if (criteria.maxPriority && task.priority > criteria.maxPriority)
        return false;

      return true;
    });
  }

  // ===== DEPENDENCY MANAGEMENT =====

  addDependency(fromTaskId, toTaskId) {
    const fromTask = this.tasks.get(fromTaskId);
    const toTask = this.tasks.get(toTaskId);

    if (!fromTask || !toTask) {
      throw new Error("One or both tasks not found");
    }

    this.dependencyGraph.addDependency(fromTaskId, toTaskId);

    // Check for cycles
    if (this.dependencyGraph.hasCycle()) {
      this.dependencyGraph.removeDependency(fromTaskId, toTaskId);
      throw new Error("Adding this dependency would create a cycle");
    }

    fromTask.addDependent(toTaskId);
    toTask.addDependency(fromTaskId);
  }

  removeDependency(fromTaskId, toTaskId) {
    this.dependencyGraph.removeDependency(fromTaskId, toTaskId);

    const fromTask = this.tasks.get(fromTaskId);
    const toTask = this.tasks.get(toTaskId);

    if (fromTask) fromTask.dependents.delete(toTaskId);
    if (toTask) toTask.dependencies.delete(fromTaskId);
  }

  getTaskExecutionOrder() {
    return this.dependencyGraph.getTopologicalOrder();
  }

  getReadyTasks() {
    const readyTaskIds = this.dependencyGraph.getReadyTasks();
    return readyTaskIds
      .map((id) => this.tasks.get(id))
      .filter((task) => task && task.status === "pending");
  }

  // ===== SCHEDULING AND OPTIMIZATION =====
```

```
  getNextTaskByPriority() {
    const readyTasks = this.getReadyTasks();
    if (readyTasks.length === 0) return null;

    // Sort by priority (descending) then by creation time (ascending)
    readyTasks.sort((a, b) => {
      if (a.priority !== b.priority) {
        return b.priority - a.priority;
      }
      return a.createdAt - b.createdAt;
    });

    return readyTasks[0];
  }

  scheduleTasksGreedy(availableHours) {
    const readyTasks = this.getReadyTasks();

    // Sort by value-to-time ratio (priority / estimated time)
    readyTasks.sort((a, b) => {
      const ratioA = a.priority / a.estimatedTime;
      const ratioB = b.priority / b.estimatedTime;
      return ratioB - ratioA;
    });

    const schedule = [];
    let remainingHours = availableHours;

    for (let task of readyTasks) {
      if (task.estimatedTime <= remainingHours) {
        schedule.push(task);
        remainingHours -= task.estimatedTime;
      }
    }

    return {
      schedule,
      totalTime: availableHours - remainingHours,
      remainingTime: remainingHours,
    };
  }

  optimizeTaskOrder() {
    // Use dynamic programming to find optimal task completion order
    const tasks = this.getAllTasks().filter(
      (task) => task.status === "pending"
    );
    const n = tasks.length;

    if (n === 0) return [];

    // DP state: dp[mask] = minimum time to complete tasks in mask
    const dp = new Array(1 << n).fill(Infinity);
```

```
    const parent = new Array(1 << n).fill(-1);
    dp[0] = 0;

    for (let mask = 0; mask < 1 << n; mask++) {
      if (dp[mask] === Infinity) continue;

      for (let i = 0; i < n; i++) {
        if (mask & (1 << i)) continue; // Task already completed

        // Check if dependencies are satisfied
        let canExecute = true;
        for (let j = 0; j < n; j++) {
          if (tasks[i].dependencies.has(tasks[j].id) && !(mask & (1 << j))) {
            canExecute = false;
            break;
          }
        }

        if (canExecute) {
          const newMask = mask | (1 << i);
          const newTime = dp[mask] + tasks[i].estimatedTime;

          if (newTime < dp[newMask]) {
            dp[newMask] = newTime;
            parent[newMask] = mask;
          }
        }
      }
    }

    // Reconstruct optimal order
    const order = [];
    let currentMask = (1 << n) - 1;

    while (currentMask > 0) {
      const prevMask = parent[currentMask];
      const taskIndex = Math.log2(currentMask ^ prevMask);
      order.unshift(tasks[taskIndex]);
      currentMask = prevMask;
    }

    return order;
  }

  // ===== UNDO/REDO OPERATIONS =====

  undo() {
    return this.undoRedoStack.undo();
  }

  redo() {
    return this.undoRedoStack.redo();
  }
```

```
  canUndo() {
    return this.undoRedoStack.canUndo();
  }

  canRedo() {
    return this.undoRedoStack.canRedo();
  }

  // ===== ANALYTICS AND REPORTING =====

  getAnalytics() {
    return this.analytics.generateReport();
  }

  getTaskStatistics() {
    const tasks = this.getAllTasks();
    const stats = {
      total: tasks.length,
      pending: 0,
      inProgress: 0,
      completed: 0,
      cancelled: 0,
      averagePriority: 0,
      totalEstimatedTime: 0,
      totalActualTime: 0,
    };

    let prioritySum = 0;

    for (let task of tasks) {
      stats[task.status]++;
      prioritySum += task.priority;
      stats.totalEstimatedTime += task.estimatedTime;
      stats.totalActualTime += task.actualTime;
    }

    stats.averagePriority = tasks.length > 0 ? prioritySum / tasks.length : 0;
    stats.efficiency =
      stats.totalActualTime > 0
        ? stats.totalEstimatedTime / stats.totalActualTime
        : 1;

    return stats;
  }
}

// ===== ANALYTICS MODULE =====

class TaskAnalytics {
  constructor() {
    this.taskHistory = [];
    this.completionTimes = [];
    this.priorityDistribution = new Map();
  }
```

```
  addTask(task) {
    this.taskHistory.push({
      action: "created",
      taskId: task.id,
      timestamp: new Date(),
      data: { ...task },
    });

    this.updatePriorityDistribution(task.priority, 1);
  }

  updateTask(task) {
    this.taskHistory.push({
      action: "updated",
      taskId: task.id,
      timestamp: new Date(),
      data: { ...task },
    });

    if (task.status === "completed" && task.completedAt) {
      const completionTime = task.completedAt - task.createdAt;
      this.completionTimes.push(completionTime);
    }
  }

  removeTask(task) {
    this.taskHistory.push({
      action: "deleted",
      taskId: task.id,
      timestamp: new Date(),
      data: { ...task },
    });

    this.updatePriorityDistribution(task.priority, -1);
  }

  updatePriorityDistribution(priority, delta) {
    const current = this.priorityDistribution.get(priority) || 0;
    this.priorityDistribution.set(priority, current + delta);
  }

  generateReport() {
    const report = {
      totalActions: this.taskHistory.length,
      averageCompletionTime: this.getAverageCompletionTime(),
      priorityDistribution: Object.fromEntries(this.priorityDistribution),
      productivityTrend: this.getProductivityTrend(),
      recentActivity: this.getRecentActivity(10),
    };

    return report;
  }
```

```
  getAverageCompletionTime() {
    if (this.completionTimes.length === 0) return 0;

    const sum = this.completionTimes.reduce((a, b) => a + b, 0);
    return sum / this.completionTimes.length;
  }

  getProductivityTrend() {
    // Calculate tasks completed per day for last 30 days
    const thirtyDaysAgo = new Date(Date.now() - 30 * 24 * 60 * 60 * 1000);
    const completedTasks = this.taskHistory.filter(
      (entry) =>
        entry.action === "updated" &&
        entry.data.status === "completed" &&
        entry.timestamp >= thirtyDaysAgo
    );

    const dailyCompletion = new Map();

    for (let task of completedTasks) {
      const date = task.timestamp.toDateString();
      dailyCompletion.set(date, (dailyCompletion.get(date) || 0) + 1);
    }

    return Object.fromEntries(dailyCompletion);
  }

  getRecentActivity(limit = 10) {
    return this.taskHistory
      .slice(-limit)
      .reverse()
      .map((entry) => ({
        action: entry.action,
        taskId: entry.taskId,
        timestamp: entry.timestamp,
        title: entry.data.title,
      }));
  }
}

// ===== EXAMPLE USAGE AND TESTING =====

console.log("=== Task Management System Demo ===");

// Create task management system
const tms = new TaskManagementSystem();

// Create sample tasks
console.log("\n=== Creating Tasks ===");
const task1 = tms.createTask(
  "Design Database Schema",
  "Create ERD and table structures",
  4,
  8,
```

```
    ["database", "design"]
);
const task2 = tms.createTask(
    "Implement User Authentication",
    "Build login/logout functionality",
    5,
    12,
    ["backend", "security"]
);
const task3 = tms.createTask(
    "Create API Endpoints",
    "Build REST API for user management",
    4,
    16,
    ["backend", "api"]
);
const task4 = tms.createTask(
    "Design UI Mockups",
    "Create wireframes and mockups",
    3,
    6,
    ["frontend", "design"]
);
const task5 = tms.createTask(
    "Write Unit Tests",
    "Create comprehensive test suite",
    3,
    10,
    ["testing", "quality"]
);

console.log(
    "Created tasks:",
    [task1, task2, task3, task4, task5].map((t) => t.title)
);

// Add dependencies
console.log("\n=== Adding Dependencies ===");
try {
    tms.addDependency(task1.id, task2.id); // Auth depends on DB
    tms.addDependency(task1.id, task3.id); // API depends on DB
    tms.addDependency(task2.id, task3.id); // API depends on Auth
    tms.addDependency(task4.id, task5.id); // Tests depend on UI
    console.log("Dependencies added successfully");
} catch (error) {
    console.error("Error adding dependencies:", error.message);
}

// Get execution order
console.log("\n=== Task Execution Order ===");
const executionOrder = tms.getTaskExecutionOrder();
console.log("Optimal execution order:", executionOrder);

// Get ready tasks
```

```
console.log("\n=== Ready Tasks ===");
const readyTasks = tms.getReadyTasks();
console.log(
  "Tasks ready to start:",
  readyTasks.map((t) => t.title)
);

// Search functionality
console.log("\n=== Search Functionality ===");
const searchResults = tms.searchTasks("design");
console.log(
  'Search results for "design":',
  searchResults.map((t) => t.title)
);

// Filter tasks
console.log("\n=== Filter Tasks ===");
const highPriorityTasks = tms.filterTasks({ minPriority: 4 });
console.log(
  "High priority tasks:",
  highPriorityTasks.map((t) => `${t.title} (Priority: ${t.priority})`)
);

const backendTasks = tms.filterTasks({ tag: "backend" });
console.log(
  "Backend tasks:",
  backendTasks.map((t) => t.title)
);

// Schedule tasks with greedy algorithm
console.log("\n=== Greedy Scheduling ===");
const schedule = tms.scheduleTasksGreedy(20); // 20 hours available
console.log("Greedy schedule for 20 hours:");
console.log(
  "Selected tasks:",
  schedule.schedule.map((t) => `${t.title} (${t.estimatedTime}h)`)
);
console.log("Total time used:", schedule.totalTime, "hours");
console.log("Remaining time:", schedule.remainingTime, "hours");

// Get next task by priority
console.log("\n=== Priority-based Next Task ===");
const nextTask = tms.getNextTaskByPriority();
console.log(
  "Next task to work on:",
  nextTask ? nextTask.title : "No tasks available"
);

// Update task status
console.log("\n=== Update Task Status ===");
tms.updateTask(task1.id, { status: "completed", actualTime: 7 });
console.log("Updated task1 status to completed");

// Check ready tasks after completion
```

```
const newReadyTasks = tms.getReadyTasks();
console.log(
  "New ready tasks after task1 completion:",
  newReadyTasks.map((t) => t.title)
);

// Undo/Redo functionality
console.log("\n=== Undo/Redo Functionality ===");
console.log("Can undo:", tms.canUndo());
console.log("Can redo:", tms.canRedo());

if (tms.canUndo()) {
  tms.undo();
  console.log("Undid last action");
  console.log("Task1 status after undo:", tms.getTask(task1.id).status);
}

if (tms.canRedo()) {
  tms.redo();
  console.log("Redid last action");
  console.log("Task1 status after redo:", tms.getTask(task1.id).status);
}

// Get statistics
console.log("\n=== Task Statistics ===");
const stats = tms.getTaskStatistics();
console.log("Task statistics:", stats);

// Get analytics
console.log("\n=== Analytics Report ===");
const analytics = tms.getAnalytics();
console.log("Analytics report:", analytics);

// Optimize task order using DP
console.log("\n=== Optimal Task Order (DP) ===");
const optimalOrder = tms.optimizeTaskOrder();
console.log(
  "DP-optimized task order:",
  optimalOrder.map((t) => t.title)
);

// Performance testing
console.log("\n=== Performance Testing ===");
console.time("Create 1000 tasks");
for (let i = 0; i < 1000; i++) {
  tms.createTask(
    `Task ${i}`,
    `Description for task ${i}`,
    Math.floor(Math.random() * 5) + 1
  );
}
console.timeEnd("Create 1000 tasks");

console.time("Search in 1000+ tasks");
```

```
const largeSearchResults = tms.searchTasks("task");
console.timeEnd("Search in 1000+ tasks");
console.log("Search results count:", largeSearchResults.length);

console.time("Filter 1000+ tasks");
const filteredResults = tms.filterTasks({ minPriority: 4 });
console.timeEnd("Filter 1000+ tasks");
console.log("Filtered results count:", filteredResults.length);

console.log("\n=== Task Management System Demo Complete ===");
```

C++ Implementation

```
#include <iostream>
#include <vector>
#include <string>
#include <unordered_map>
#include <unordered_set>
#include <queue>
#include <stack>
#include <algorithm>
#include <chrono>
#include <memory>
using namespace std;
using namespace std::chrono;

// ===== TASK ENTITY =====

class Task {
public:
    string id;
    string title;
    string description;
    int priority;
    int estimatedTime;
    int actualTime;
    string status; // "pending", "in-progress", "completed", "cancelled"
    unordered_set<string> tags;
    unordered_set<string> dependencies;
    unordered_set<string> dependents;
    time_point<system_clock> createdAt;
    time_point<system_clock> updatedAt;
    time_point<system_clock> completedAt;
    string assignee;

    Task(const string& id, const string& title, const string& description,
         int priority = 1, int estimatedTime = 1)
        : id(id), title(title), description(description), priority(priority),
          estimatedTime(estimatedTime), actualTime(0), status("pending"),
          createdAt(system_clock::now()), updatedAt(system_clock::now()) {}
```

```
    void addDependency(const string& taskId) {
        dependencies.insert(taskId);
    }

    void addDependent(const string& taskId) {
        dependents.insert(taskId);
    }

    void updateStatus(const string& newStatus) {
        status = newStatus;
        updatedAt = system_clock::now();
        if (newStatus == "completed") {
            completedAt = system_clock::now();
        }
    }

    void addTag(const string& tag) {
        tags.insert(tag);
    }

    bool isBlocked() const {
        return !dependencies.empty();
    }

    double getEfficiency() const {
        if (actualTime == 0) return 1.0;
        return static_cast<double>(estimatedTime) / actualTime;
    }
};

// ===== PRIORITY QUEUE FOR TASK SCHEDULING =====

struct TaskComparator {
    bool operator()(const shared_ptr<Task>& a, const shared_ptr<Task>& b) {
        if (a->priority != b->priority) {
            return a->priority < b->priority; // Max heap by priority
        }
        return a->createdAt > b->createdAt; // Earlier tasks first
    }
};

// ===== DEPENDENCY GRAPH =====

class DependencyGraph {
private:
    unordered_map<string, vector<string>> adjacencyList;
    unordered_map<string, int> inDegree;

public:
    void addTask(const string& taskId) {
        if (adjacencyList.find(taskId) == adjacencyList.end()) {
            adjacencyList[taskId] = vector<string>();
            inDegree[taskId] = 0;
```

```
        }
    }

    void addDependency(const string& fromTask, const string& toTask) {
        addTask(fromTask);
        addTask(toTask);

        adjacencyList[fromTask].push_back(toTask);
        inDegree[toTask]++;
    }

    void removeDependency(const string& fromTask, const string& toTask) {
        auto& neighbors = adjacencyList[fromTask];
        auto it = find(neighbors.begin(), neighbors.end(), toTask);
        if (it != neighbors.end()) {
            neighbors.erase(it);
            inDegree[toTask]--;
        }
    }

    vector<string> getTopologicalOrder() {
        vector<string> result;
        queue<string> q;
        unordered_map<string, int> tempInDegree = inDegree;

        // Find all tasks with no dependencies
        for (const auto& pair : tempInDegree) {
            if (pair.second == 0) {
                q.push(pair.first);
            }
        }

        while (!q.empty()) {
            string current = q.front();
            q.pop();
            result.push_back(current);

            // Process all dependent tasks
            for (const string& neighbor : adjacencyList[current]) {
                tempInDegree[neighbor]--;
                if (tempInDegree[neighbor] == 0) {
                    q.push(neighbor);
                }
            }
        }

        // Check for cycles
        if (result.size() != adjacencyList.size()) {
            throw runtime_error("Circular dependency detected!");
        }

        return result;
    }
```

```
    vector<string> getReadyTasks() {
        vector<string> readyTasks;
        for (const auto& pair : inDegree) {
            if (pair.second == 0) {
                readyTasks.push_back(pair.first);
            }
        }
        return readyTasks;
    }

    bool hasCycle() {
        try {
            getTopologicalOrder();
            return false;
        } catch (const runtime_error&) {
            return true;
        }
    }
};

// ===== TRIE FOR TEXT SEARCH =====

struct TrieNode {
    unordered_map<char, unique_ptr<TrieNode>> children;
    bool isEndOfWord;
    unordered_set<string> taskIds;

    TrieNode() : isEndOfWord(false) {}
};

class Trie {
private:
    unique_ptr<TrieNode> root;

    void toLowerCase(string& str) {
        transform(str.begin(), str.end(), str.begin(), ::tolower);
    }

public:
    Trie() : root(make_unique<TrieNode>()) {}

    void insert(string word, const string& taskId) {
        toLowerCase(word);
        TrieNode* node = root.get();

        for (char c : word) {
            if (node->children.find(c) == node->children.end()) {
                node->children[c] = make_unique<TrieNode>();
            }
            node = node->children[c].get();
            node->taskIds.insert(taskId);
        }

        node->isEndOfWord = true;
```

```
    }

    vector<string> search(string prefix) {
        toLowerCase(prefix);
        TrieNode* node = root.get();

        for (char c : prefix) {
            if (node->children.find(c) == node->children.end()) {
                return vector<string>();
            }
            node = node->children[c].get();
        }

        return vector<string>(node->taskIds.begin(), node->taskIds.end());
    }
};

// ===== MAIN TASK MANAGEMENT SYSTEM =====

class TaskManagementSystem {
private:
    unordered_map<string, shared_ptr<Task>> tasks;
    priority_queue<shared_ptr<Task>, vector<shared_ptr<Task>>, TaskComparator>
priorityQueue;
    Trie searchTrie;
    DependencyGraph dependencyGraph;
    int taskIdCounter;

    void indexTaskForSearch(shared_ptr<Task> task) {
        // Index title words
        istringstream titleStream(task->title);
        string word;
        while (titleStream >> word) {
            if (word.length() > 2) {
                searchTrie.insert(word, task->id);
            }
        }

        // Index description words
        istringstream descStream(task->description);
        while (descStream >> word) {
            if (word.length() > 2) {
                searchTrie.insert(word, task->id);
            }
        }

        // Index tags
        for (const string& tag : task->tags) {
            searchTrie.insert(tag, task->id);
        }
    }

public:
    TaskManagementSystem() : taskIdCounter(1) {}
```

```
    // ===== TASK CRUD OPERATIONS =====

    shared_ptr<Task> createTask(const string& title, const string& description,
                                int priority = 1, int estimatedTime = 1) {
        string taskId = "task_" + to_string(taskIdCounter++);
        auto task = make_shared<Task>(taskId, title, description, priority,
estimatedTime);

        tasks[taskId] = task;
        priorityQueue.push(task);
        dependencyGraph.addTask(taskId);
        indexTaskForSearch(task);

        return task;
    }

    shared_ptr<Task> getTask(const string& taskId) {
        auto it = tasks.find(taskId);
        return (it != tasks.end()) ? it->second : nullptr;
    }

    vector<shared_ptr<Task>> getAllTasks() {
        vector<shared_ptr<Task>> allTasks;
        for (const auto& pair : tasks) {
            allTasks.push_back(pair.second);
        }
        return allTasks;
    }

    bool updateTask(const string& taskId, const string& field, const string&
value) {
        auto task = getTask(taskId);
        if (!task) return false;

        if (field == "status") {
            task->updateStatus(value);
        } else if (field == "title") {
            task->title = value;
        } else if (field == "description") {
            task->description = value;
        } else if (field == "assignee") {
            task->assignee = value;
        }

        task->updatedAt = system_clock::now();
        return true;
    }

    bool deleteTask(const string& taskId) {
        auto it = tasks.find(taskId);
        if (it == tasks.end()) return false;

        tasks.erase(it);
```

```
        return true;
    }

    // ===== SEARCH AND FILTERING =====

    vector<shared_ptr<Task>> searchTasks(const string& query) {
        vector<string> taskIds = searchTrie.search(query);
        vector<shared_ptr<Task>> results;

        for (const string& id : taskIds) {
            auto task = getTask(id);
            if (task) {
                results.push_back(task);
            }
        }

        return results;
    }

    vector<shared_ptr<Task>> filterTasksByStatus(const string& status) {
        vector<shared_ptr<Task>> filtered;
        for (const auto& pair : tasks) {
            if (pair.second->status == status) {
                filtered.push_back(pair.second);
            }
        }
        return filtered;
    }

    vector<shared_ptr<Task>> filterTasksByPriority(int minPriority) {
        vector<shared_ptr<Task>> filtered;
        for (const auto& pair : tasks) {
            if (pair.second->priority >= minPriority) {
                filtered.push_back(pair.second);
            }
        }
        return filtered;
    }

    // ===== DEPENDENCY MANAGEMENT =====

    bool addDependency(const string& fromTaskId, const string& toTaskId) {
        auto fromTask = getTask(fromTaskId);
        auto toTask = getTask(toTaskId);

        if (!fromTask || !toTask) return false;

        dependencyGraph.addDependency(fromTaskId, toTaskId);

        // Check for cycles
        if (dependencyGraph.hasCycle()) {
            dependencyGraph.removeDependency(fromTaskId, toTaskId);
            return false;
        }
```

```
        fromTask->addDependent(toTaskId);
        toTask->addDependency(fromTaskId);
        return true;
    }

    vector<string> getTaskExecutionOrder() {
        return dependencyGraph.getTopologicalOrder();
    }

    vector<shared_ptr<Task>> getReadyTasks() {
        vector<string> readyTaskIds = dependencyGraph.getReadyTasks();
        vector<shared_ptr<Task>> readyTasks;

        for (const string& id : readyTaskIds) {
            auto task = getTask(id);
            if (task && task->status == "pending") {
                readyTasks.push_back(task);
            }
        }

        return readyTasks;
    }

    // ===== SCHEDULING =====

    shared_ptr<Task> getNextTaskByPriority() {
        auto readyTasks = getReadyTasks();
        if (readyTasks.empty()) return nullptr;

        // Sort by priority (descending) then by creation time (ascending)
        sort(readyTasks.begin(), readyTasks.end(),
             [](const shared_ptr<Task>& a, const shared_ptr<Task>& b) {
                 if (a->priority != b->priority) {
                     return a->priority > b->priority;
                 }
                 return a->createdAt < b->createdAt;
             });

        return readyTasks[0];
    }

    struct ScheduleResult {
        vector<shared_ptr<Task>> schedule;
        int totalTime;
        int remainingTime;
    };

    ScheduleResult scheduleTasksGreedy(int availableHours) {
        auto readyTasks = getReadyTasks();

        // Sort by value-to-time ratio (priority / estimated time)
        sort(readyTasks.begin(), readyTasks.end(),
             [](const shared_ptr<Task>& a, const shared_ptr<Task>& b) {
```

```
                double ratioA = static_cast<double>(a->priority) / a-
>estimatedTime;
                double ratioB = static_cast<double>(b->priority) / b-
>estimatedTime;
                return ratioA > ratioB;
            });

        ScheduleResult result;
        result.totalTime = 0;
        result.remainingTime = availableHours;

        for (auto task : readyTasks) {
            if (task->estimatedTime <= result.remainingTime) {
                result.schedule.push_back(task);
                result.totalTime += task->estimatedTime;
                result.remainingTime -= task->estimatedTime;
            }
        }

        return result;
    }

    // ===== STATISTICS =====

    struct TaskStatistics {
        int total;
        int pending;
        int inProgress;
        int completed;
        int cancelled;
        double averagePriority;
        int totalEstimatedTime;
        int totalActualTime;
        double efficiency;
    };

    TaskStatistics getTaskStatistics() {
        TaskStatistics stats = {0, 0, 0, 0, 0, 0.0, 0, 0, 1.0};
        int prioritySum = 0;

        for (const auto& pair : tasks) {
            auto task = pair.second;
            stats.total++;

            if (task->status == "pending") stats.pending++;
            else if (task->status == "in-progress") stats.inProgress++;
            else if (task->status == "completed") stats.completed++;
            else if (task->status == "cancelled") stats.cancelled++;

            prioritySum += task->priority;
            stats.totalEstimatedTime += task->estimatedTime;
            stats.totalActualTime += task->actualTime;
        }
```

```
        if (stats.total > 0) {
            stats.averagePriority = static_cast<double>(prioritySum) /
stats.total;
        }

        if (stats.totalActualTime > 0) {
            stats.efficiency = static_cast<double>(stats.totalEstimatedTime) /
stats.totalActualTime;
        }

        return stats;
    }

    void printStatistics() {
        auto stats = getTaskStatistics();
        cout << "=== Task Statistics ===" << endl;
        cout << "Total tasks: " << stats.total << endl;
        cout << "Pending: " << stats.pending << endl;
        cout << "In Progress: " << stats.inProgress << endl;
        cout << "Completed: " << stats.completed << endl;
        cout << "Cancelled: " << stats.cancelled << endl;
        cout << "Average Priority: " << stats.averagePriority << endl;
        cout << "Total Estimated Time: " << stats.totalEstimatedTime << " hours"
<< endl;
        cout << "Total Actual Time: " << stats.totalActualTime << " hours" <<
endl;
        cout << "Efficiency: " << stats.efficiency << endl;
    }
};

// ===== EXAMPLE USAGE =====

int main() {
    cout << "=== Task Management System Demo ===" << endl;

    TaskManagementSystem tms;

    // Create sample tasks
    cout << "\n=== Creating Tasks ===" << endl;
    auto task1 = tms.createTask("Design Database Schema", "Create ERD and table
structures", 4, 8);
    auto task2 = tms.createTask("Implement User Authentication", "Build
login/logout functionality", 5, 12);
    auto task3 = tms.createTask("Create API Endpoints", "Build REST API for user
management", 4, 16);
    auto task4 = tms.createTask("Design UI Mockups", "Create wireframes and
mockups", 3, 6);
    auto task5 = tms.createTask("Write Unit Tests", "Create comprehensive test
suite", 3, 10);

    task1->addTag("database");
    task1->addTag("design");
    task2->addTag("backend");
    task2->addTag("security");
```

```
    task3->addTag("backend");
    task3->addTag("api");
    task4->addTag("frontend");
    task4->addTag("design");
    task5->addTag("testing");
    task5->addTag("quality");

    cout << "Created 5 tasks successfully" << endl;

    // Add dependencies
    cout << "\n=== Adding Dependencies ===" << endl;
    if (tms.addDependency(task1->id, task2->id)) {
        cout << "Added dependency: Auth depends on DB" << endl;
    }
    if (tms.addDependency(task1->id, task3->id)) {
        cout << "Added dependency: API depends on DB" << endl;
    }
    if (tms.addDependency(task2->id, task3->id)) {
        cout << "Added dependency: API depends on Auth" << endl;
    }
    if (tms.addDependency(task4->id, task5->id)) {
        cout << "Added dependency: Tests depend on UI" << endl;
    }

    // Get execution order
    cout << "\n=== Task Execution Order ===" << endl;
    try {
        auto executionOrder = tms.getTaskExecutionOrder();
        cout << "Optimal execution order: ";
        for (const string& taskId : executionOrder) {
            auto task = tms.getTask(taskId);
            if (task) {
                cout << task->title << " -> ";
            }
        }
        cout << "END" << endl;
    } catch (const runtime_error& e) {
        cout << "Error: " << e.what() << endl;
    }

    // Get ready tasks
    cout << "\n=== Ready Tasks ===" << endl;
    auto readyTasks = tms.getReadyTasks();
    cout << "Tasks ready to start: ";
    for (auto task : readyTasks) {
        cout << task->title << ", ";
    }
    cout << endl;

    // Search functionality
    cout << "\n=== Search Functionality ===" << endl;
    auto searchResults = tms.searchTasks("design");
    cout << "Search results for 'design': ";
    for (auto task : searchResults) {
```

```
        cout << task->title << ", ";
    }
    cout << endl;

    // Filter tasks
    cout << "\n=== Filter Tasks ===" << endl;
    auto highPriorityTasks = tms.filterTasksByPriority(4);
    cout << "High priority tasks (>=4): ";
    for (auto task : highPriorityTasks) {
        cout << task->title << " (Priority: " << task->priority << "), ";
    }
    cout << endl;

    auto pendingTasks = tms.filterTasksByStatus("pending");
    cout << "Pending tasks: " << pendingTasks.size() << " tasks" << endl;

    // Schedule tasks with greedy algorithm
    cout << "\n=== Greedy Scheduling ===" << endl;
    auto schedule = tms.scheduleTasksGreedy(20); // 20 hours available
    cout << "Greedy schedule for 20 hours:" << endl;
    cout << "Selected tasks: ";
    for (auto task : schedule.schedule) {
        cout << task->title << " (" << task->estimatedTime << "h), ";
    }
    cout << endl;
    cout << "Total time used: " << schedule.totalTime << " hours" << endl;
    cout << "Remaining time: " << schedule.remainingTime << " hours" << endl;

    // Get next task by priority
    cout << "\n=== Priority-based Next Task ===" << endl;
    auto nextTask = tms.getNextTaskByPriority();
    if (nextTask) {
        cout << "Next task to work on: " << nextTask->title << endl;
    } else {
        cout << "No tasks available" << endl;
    }

    // Update task status
    cout << "\n=== Update Task Status ===" << endl;
    if (tms.updateTask(task1->id, "status", "completed")) {
        cout << "Updated task1 status to completed" << endl;
    }

    // Check ready tasks after completion
    auto newReadyTasks = tms.getReadyTasks();
    cout << "New ready tasks after task1 completion: ";
    for (auto task : newReadyTasks) {
        cout << task->title << ", ";
    }
    cout << endl;

    // Get statistics
    cout << "\n=== Task Statistics ===" << endl;
    tms.printStatistics();
```

```
    // Performance testing
    cout << "\n=== Performance Testing ===" << endl;
    auto start = high_resolution_clock::now();

    for (int i = 0; i < 1000; i++) {
        tms.createTask("Task " + to_string(i), "Description for task " +
to_string(i),
                       (rand() % 5) + 1, (rand() % 10) + 1);
    }

    auto end = high_resolution_clock::now();
    auto duration = duration_cast<milliseconds>(end - start);
    cout << "Created 1000 tasks in: " << duration.count() << " ms" << endl;

    start = high_resolution_clock::now();
    auto largeSearchResults = tms.searchTasks("task");
    end = high_resolution_clock::now();
    duration = duration_cast<milliseconds>(end - start);
    cout << "Search in 1000+ tasks took: " << duration.count() << " ms" << endl;
    cout << "Search results count: " << largeSearchResults.size() << endl;

    start = high_resolution_clock::now();
    auto filteredResults = tms.filterTasksByPriority(4);
    end = high_resolution_clock::now();
    duration = duration_cast<milliseconds>(end - start);
    cout << "Filter 1000+ tasks took: " << duration.count() << " ms" << endl;
    cout << "Filtered results count: " << filteredResults.size() << endl;

    cout << "\n=== Task Management System Demo Complete ===" << endl;

    return 0;
}
```

🎯 Project Challenges and Extensions

Phase 1: Basic Implementation (Beginner)

1. **Task CRUD Operations**: Implement basic create, read, update, delete
2. **Simple Search**: Linear search through task titles
3. **Basic Priority Queue**: Use built-in priority queue
4. **Simple Dependencies**: Track dependencies without cycle detection

Phase 2: Intermediate Features (Medium)

1. **Advanced Search**: Implement Trie-based search with autocomplete
2. **Dependency Management**: Add cycle detection and topological sorting
3. **Undo/Redo System**: Implement command pattern with stacks
4. **Performance Optimization**: Add hash tables and efficient data structures

Phase 3: Advanced Features (Advanced)

1. **Dynamic Programming**: Optimize task scheduling with DP
2. **Graph Algorithms**: Implement shortest path for task dependencies
3. **Machine Learning**: Add task time estimation based on historical data
4. **Distributed System**: Scale to multiple users with conflict resolution

Bonus Challenges:

1. **Real-time Collaboration**: Multiple users editing simultaneously
2. **Mobile App**: Create mobile interface with offline sync
3. **AI Assistant**: Natural language task creation and scheduling
4. **Analytics Dashboard**: Visual insights and productivity metrics

---

📊 Performance Analysis

Time Complexity Analysis:

Operation	Hash Table	Priority Queue	Trie Search	Dependency Graph
Insert	O(1) avg	O(log n)	O(m)	O(1)
Search	O(1) avg	O(n)	O(m + k)	O(V + E)
Delete	O(1) avg	O(log n)	O(m)	O(V + E)
Update	O(1) avg	O(log n)	O(m)	O(1)

Where n = number of tasks, m = length of search term, k = number of results, V = vertices, E = edges

Space Complexity:

- **Hash Table**: O(n) for task storage
- **Trie**: O(ALPHABET*SIZE * N _ M) where N is number of words, M is average word length
- **Dependency Graph**: O(V + E) for adjacency list representation
- **Priority Queue**: O(n) for heap storage
- **Undo Stack**: O(k) where k is maximum undo operations

Optimization Techniques Used:

1. **Hash Table Resizing**: Dynamic resizing to maintain load factor
2. **Trie Compression**: Store task IDs at each node for efficient retrieval
3. **Lazy Evaluation**: Compute topological order only when needed
4. **Memoization**: Cache frequently accessed computations
5. **Batch Operations**: Group multiple operations for better performance

---

🧪 Testing Strategy

Unit Tests:

```
// Example test cases
function testTaskCreation() {
  const tms = new TaskManagementSystem();
  const task = tms.createTask("Test Task", "Test Description", 3, 5);

  assert(task.title === "Test Task");
  assert(task.priority === 3);
  assert(task.estimatedTime === 5);
  assert(task.status === "pending");
  console.log("✓ Task creation test passed");
}

function testDependencyManagement() {
  const tms = new TaskManagementSystem();
  const task1 = tms.createTask("Task 1", "First task", 1);
  const task2 = tms.createTask("Task 2", "Second task", 2);

  tms.addDependency(task1.id, task2.id);

  const readyTasks = tms.getReadyTasks();
  assert(readyTasks.length === 1);
  assert(readyTasks[0].id === task1.id);
  console.log("✓ Dependency management test passed");
}

function testSearchFunctionality() {
  const tms = new TaskManagementSystem();
  tms.createTask("Design Database", "Create schema", 4);
  tms.createTask("Implement API", "Build endpoints", 3);

  const results = tms.searchTasks("design");
  assert(results.length === 1);
  assert(results[0].title.includes("Design"));
  console.log("✓ Search functionality test passed");
}

function testGreedyScheduling() {
  const tms = new TaskManagementSystem();
  tms.createTask("High Priority Short", "Quick task", 5, 2);
  tms.createTask("Low Priority Long", "Long task", 1, 10);

  const schedule = tms.scheduleTasksGreedy(5);
  assert(schedule.schedule.length === 1);
  assert(schedule.schedule[0].priority === 5);
  console.log("✓ Greedy scheduling test passed");
}

// Run all tests
testTaskCreation();
testDependencyManagement();
testSearchFunctionality();
testGreedyScheduling();
console.log("\n🎉 All tests passed!");
```

Integration Tests:

1. **End-to-End Workflow**: Create → Add Dependencies → Schedule → Complete
2. **Performance Tests**: Large dataset operations (1000+ tasks)
3. **Stress Tests**: Concurrent operations and edge cases
4. **Memory Tests**: Check for memory leaks in long-running operations

Edge Cases to Test:

1. **Circular Dependencies**: Ensure proper cycle detection
2. **Empty Datasets**: Handle operations on empty task lists
3. **Invalid Inputs**: Graceful handling of malformed data
4. **Boundary Conditions**: Maximum task limits and constraints

---

🎓 Learning Outcomes

After completing this mini-project, you will have:

Data Structures Mastery:

- ✅ **Hash Tables**: Fast lookups and storage
- ✅ **Linked Lists**: Dynamic data management
- ✅ **Stacks**: Undo/Redo functionality
- ✅ **Queues**: Task processing pipelines
- ✅ **Heaps**: Priority-based scheduling
- ✅ **Trees**: Hierarchical organization
- ✅ **Graphs**: Dependency relationships
- ✅ **Tries**: Efficient text searching

Algorithm Proficiency:

- ✅ **Sorting**: Task prioritization and ordering
- ✅ **Searching**: Binary search and text search
- ✅ **Graph Traversal**: DFS/BFS for dependencies
- ✅ **Topological Sort**: Task execution ordering
- ✅ **Dynamic Programming**: Optimal scheduling
- ✅ **Greedy Algorithms**: Resource allocation
- ✅ **Backtracking**: Constraint satisfaction

System Design Skills:

- ✅ **Modular Architecture**: Separation of concerns
- ✅ **Performance Optimization**: Time/space complexity
- ✅ **Error Handling**: Robust error management
- ✅ **Testing Strategy**: Comprehensive test coverage
- ✅ **Scalability**: Design for growth

Real-World Applications:

- ✅ **Project Management**: Task scheduling and tracking
- ✅ **Resource Optimization**: Efficient allocation algorithms
- ✅ **User Experience**: Search and filtering capabilities
- ✅ **Data Analytics**: Performance insights and reporting

🚀 Next Steps and Career Applications

Portfolio Enhancement:

1. **GitHub Repository**: Create a well-documented repo
2. **Live Demo**: Deploy a web version
3. **Technical Blog**: Write about your implementation
4. **Video Walkthrough**: Explain your design decisions

Interview Preparation:

1. **System Design**: Use this as a system design example
2. **Coding Challenges**: Extract individual algorithms for practice
3. **Complexity Analysis**: Discuss optimization decisions
4. **Trade-offs**: Explain design choices and alternatives

Industry Applications:

- **Project Management Tools**: Jira, Asana, Trello
- **Development Workflows**: CI/CD pipelines, build systems
- **Resource Planning**: Manufacturing, logistics, scheduling
- **Game Development**: Quest systems, skill trees
- **AI/ML**: Task scheduling in distributed training

Advanced Extensions:

1. **Microservices**: Split into independent services
2. **Event Sourcing**: Track all state changes
3. **CQRS**: Separate read/write models
4. **Real-time Updates**: WebSocket integration
5. **Machine Learning**: Predictive analytics

📚 Additional Resources

Books:

- "Introduction to Algorithms" by Cormen, Leiserson, Rivest, Stein
- "System Design Interview" by Alex Xu
- "Designing Data-Intensive Applications" by Martin Kleppmann

Online Courses:

- Coursera: "Algorithms Specialization" by Stanford
- edX: "Data Structures and Algorithms" by MIT
- Udemy: "Master the Coding Interview: Data Structures + Algorithms"

Practice Platforms:

- LeetCode: Algorithm practice problems
- HackerRank: Data structures challenges
- CodeSignal: System design questions
- Pramp: Mock interviews

Documentation:

- MDN Web Docs: JavaScript reference
- cppreference.com: C++ standard library
- GeeksforGeeks: Algorithm explanations
- Stack Overflow: Community Q&A

🎉 Conclusion

Congratulations! You've completed a comprehensive mini-project that integrates multiple data structures and algorithms concepts. This Task Management System demonstrates practical applications of:

- **15+ Data Structures**: From basic arrays to complex graphs
- **10+ Algorithms**: Sorting, searching, optimization, and more
- **Real-world Problem Solving**: Practical software development
- **Performance Optimization**: Efficient algorithm selection
- **System Design**: Scalable architecture patterns

This project serves as an excellent portfolio piece and interview preparation tool. You've not only learned individual concepts but also how to combine them effectively to solve complex problems.

Keep building, keep learning, and keep coding! 🚀

"The best way to learn data structures and algorithms is to implement them in real projects. This mini-project gives you that hands-on experience while building something genuinely useful."

Happy Coding! 💻✨