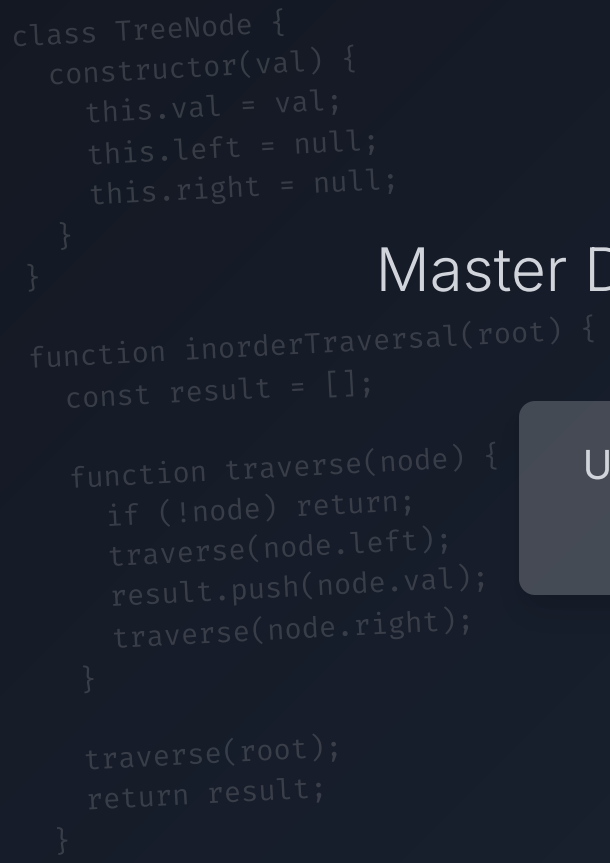
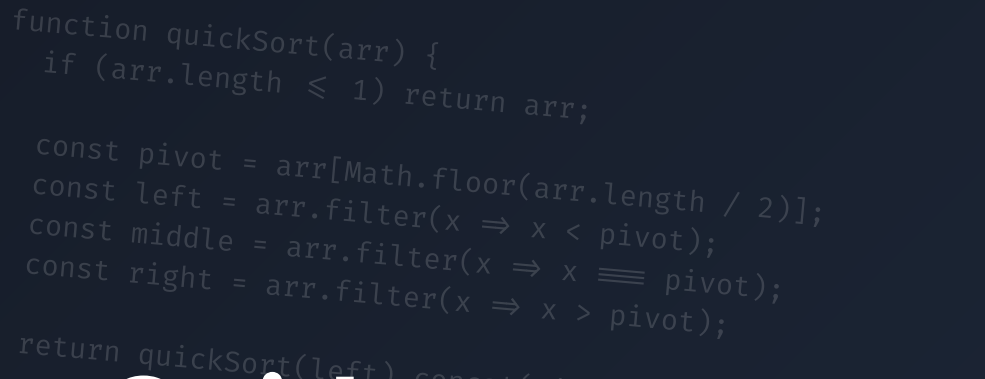
THE ULTIMATE GUIDE TO





DSA Revision Guide

Master Data Structures & Algorithms for **FAANG** Interviews

Unlock your dream tech career with our comprehensive guide to acing the most challenging coding interviews in the industry

2025 Edition

#FAANG_INTERVIEW_PREP

/* Table of Contents */

const sections = [

{ Table of Contents }

]; // End of Contents

Navigate your DSA journey

ESSENTIAL

CHAPTER 01

Introduction

Why every serious tech candidate needs to master Data Structures & Algorithms

Key goals of this guide

Why DSA Dominates FAANG Interviews

> **DSA competency** separates candidates who merely know syntax from those who can build efficient software at scale

- **Algorithmic Thinking**: FAANG companies value engineers who can break down complex problems into solvable components
- **Technical Communication**: DSA interviews test your ability to articulate your thought process and coding decisions
- **Optimization Skills**: Demonstrates your capacity to consider time/space tradeoffs and performance implications

---

The Role of Problem Solving

Problem-solving frameworks enable you to approach any challenge systematically, regardless of whether you've seen a similar problem before

FAANG Interview Process Overview

Process may vary slightly by company

The Multi-Round Process

1. Online Assessment / Resume Screening

Initial automated coding challenges or resume filtering

- 1-2 algorithmic problems (usually easy-medium)
- Typically 60-90 minutes time limit

2. Phone/Virtual Technical Interviews

1-2 rounds with engineers focusing on core DSA

- Live coding in shared document or platform
- Conversational problem-solving expected

3. Onsite/Virtual Loop

4-6 rounds covering various technical areas

- DSA deep dive (2-3 rounds)
- System design (1-2 rounds)
- Behavioral/leadership (1-2 rounds)

The DSA rounds are the most challenging filter - approximately 70% of candidates are eliminated during these rounds. Mastering patterns and practicing communication are key.

Evaluation Criteria

Technical Proficiency
Correct, working solution that handles edge cases

Optimization
Time & space complexity awareness; efficiency

Communication
Clear thought process, asks clarifying questions

Testing & Debugging
Verifies solution, finds & fixes bugs independently

Code Quality
Clean, readable, modular, well-structured code

Most candidates need 2-3 months of dedicated preparation

Success Mindset & Study Strategy

How to excel in your FAANG interview preparation journey

Growth Mindset

Embrace the Challenge
See difficult problems as opportunities to grow rather than obstacles

Learn from Failure
Analyze incorrect solutions and understand the patterns you missed

Smart Study Approach

Quality Over Quantity

Understanding 100 problems thoroughly is better than rushing through 500 problems superficially

- Master core patterns before moving to variations
- Revisit solved problems after 1-2 weeks
- Focus on the thinking process, not just solutions

Deliberate Practice

Timed Sessions
Simulate interview conditions with 30-45 min time blocks

Mock Interviews
Practice with peers to get comfortable explaining your approach

Pattern Mastery
Group problems by pattern to strengthen recognition

Manual Tracing
Work through algorithms by hand before coding

LeetCode Pattern Recognition
Focus on the Blind 75 list and categorize by approach (Two Pointers, DP, DFS, etc.)

Key Takeaway

Consistency beats cramming. Aim for **1-2 hours daily** over 2-3 months rather than intense last-minute preparation.

> *"The FAANG interview is not about testing what you know, but how you think."*

CRITICAL

CHAPTER 02

Time & Space Complexity

Understanding efficiency and optimization in algorithms

Fundamental to FAANG interviews

What is Big O Notation?

> **Big O notation** describes the performance or complexity of an algorithm, focusing on worst-case scenarios as inputs grow

- **Growth Rate Focus**: Big O measures how the runtime or space requirements scale with input size, not the exact time or bytes
- **Asymptotic Analysis**: Examines algorithm efficiency as input size approaches infinity, ignoring constants
- **Comparative Tool**: Provides a standardized way to compare different algorithms and their efficiency

Common Notations

`O(1)` `O(log n)` `O(n)` `O(n log n)` `O(n²)` `O(2ⁿ)` `O(n!)`

Big O Time Complexity Deep Dive

Understanding algorithmic efficiency with practical examples

O(1) - Constant Time

Fastest

Operations that execute in the same time regardless of input size

```
function getFirstElement(array) {
  return array[0]; // O(1) - immediate acc
}
```

Examples: Array index access, Hash map lookups, Stack operations

O(log n) - Logarithmic

Very efficient

Algorithms that divide the problem in half each step

```
function binarySearch(array, target) {
  let left = 0, right = array.length - 1;
  while (left <= right) {
    let mid = Math.floor((left + right) / 2
    if (array[mid] === target) return mid;
    if (array[mid] < target) left = mid + 1
    else right = mid - 1;
  }
  return -1; // O(log n) - eliminates half
}
```

Examples: Binary search, Balanced BST operations, Divide & conquer

O(n) - Linear Time

Scales with input

Processing time grows linearly with input size

```
function findMax(array) {
  let max = array[0];
  for (let i = 1; i < array.length; i++) {
    if (array[i] > max) max = array[i];
  } // O(n) - examines each element once
  return max;
}
```

Examples: Linear search, Traversals (array, linked list), Counting

O(n log n) - Linearithmic

Efficient for sorting

Common in efficient sorting algorithms

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

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

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

Examples: Merge sort, Quick sort (avg), Heap sort

O(n²) - Quadratic Time

Slow for large inputs

Processing time grows with square of input size

```
function bubbleSort(array) {
  for (let i = 0; i < array.length; i++) {
    for (let j = 0; j < array.length - i -
      if (array[j] > array[j + 1]) {
        [array[j], array[j + 1]] = [array[
      }
    } // O(n²) - nested loops
  }
  return array;
}
```

Examples: Bubble sort, Selection sort, Nested iterations

O(2^n) - Exponential

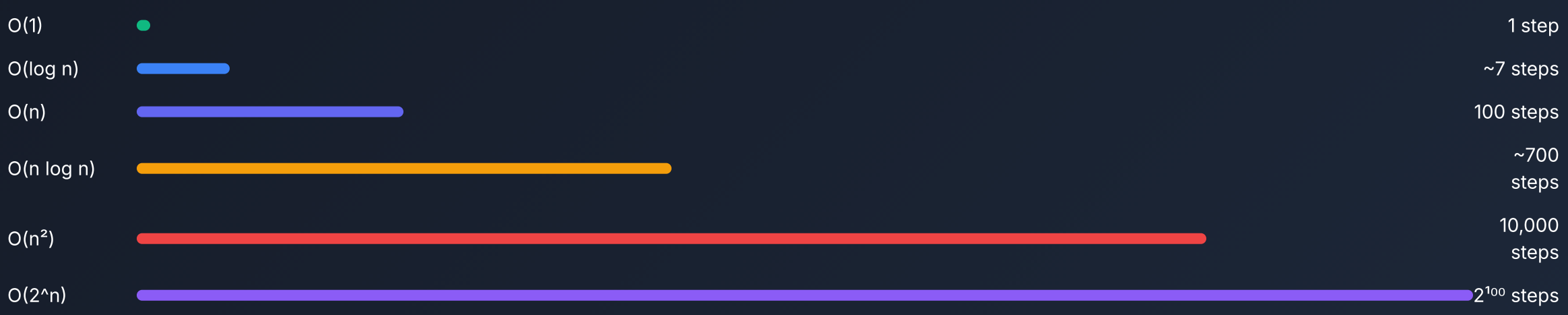
Very expensive

Operations double with each addition to input

```
function fibonacci(n) {
  if (n <= 1) return n;
  return fibonacci(n-1) + fibonacci(n-2);
} // O(2^n) - exponential recursive calls
```

Examples: Naive recursive algorithms, Subsets generation, Brute force

Growth Rate Comparison (n = 100)



FAANG Interview Tip

Always verbalize your time complexity analysis during interviews. When you present a solution, proactively identify its Big O notation and explain why — interviewers specifically look for this awareness.

Space Complexity Essentials

How efficient is your algorithm with memory?

Understanding Space Complexity

Space complexity measures the amount of memory an algorithm uses relative to its input size. It includes:

Auxiliary Space
Extra space used during algorithm execution (variables, data structures)

Input Space
Memory needed to store the input data itself

Common Space Complexities

O(1) Constant Space

Fixed amount of memory regardless of input size

Example: Swapping variables, math operations

O(log n) Logarithmic Space

Memory usage grows logarithmically with input size

Example: Many recursive algorithms like binary search

O(n) Linear Space

Memory usage grows linearly with input size

Example: Creating a new array from input array

Memory Optimization Techniques

In-Place Algorithms

Modify input data structures directly without using extra space

```
// Reversing array in-place: O(1) auxiliary space
function reverseInPlace(arr) {
  let left = 0, right = arr.length - 1;
  while (left < right) {
    [arr[left], arr[right]] = [arr[right], arr[left]];
    left++; right--;
  }
}
```

Space-Time Tradeoffs

Sometimes we can sacrifice memory to gain speed, or vice versa

Memoization (DP)	Fast · Memory heavy
Iterative recalculation	Slower · Memory efficient

Analysis Tips

Identify Variables
Count all auxiliary data structures and their size relative to input

Recursion Stack
Remember to include call stack space for recursive functions

Reuse Memory
Overwrite existing variables/arrays instead of creating new ones

Output Space
Don't count output space if it's a required part of the problem

FAANG Interview Tip
When asked about space complexity, always mention both auxiliary space and total space. This demonstrates deeper understanding of memory analysis.

Practical Complexity: How FAANG Interviewers Think

From the interviewer's perspective

Q: What are interviewers really looking for when discussing Big O?

Interviewers evaluate your ability to reason about efficiency and understand performance implications. They want to see that you can:

- Identify bottlenecks in your algorithm
- Distinguish between average and worst-case scenarios
- Articulate time-space tradeoffs clearly and confidently

Q: How detailed should my complexity analysis be?

Start with high-level complexity, then demonstrate deeper understanding:

```
"This solution is O(n) time complexity because we traverse the array exactly once..."
```

Follow up with nuance: "However, the space complexity could be optimized from O(n) to O(1) by modifying our approach to use two pointers instead..."

Q: What if I'm not sure about the exact complexity?

It's better to reason through it than guess. Walk through your thinking:

"I have nested loops where the inner loop processes at most k elements, so that gives us O(n×k)... since k is bounded by the constant MAX_TRANSACTIONS, we could simplify to O(n)..."

Q: Do I need to optimize every solution to O(1) or O(n)?

No. Interviewers value awareness of tradeoffs more than extreme optimization. Be able to explain:

- Why your current approach has its complexity
- What alternatives exist and their tradeoffs
- Which factors would make you choose one approach over another

Sample Dialogue

O(1)

Interviewer: What's the time complexity of your solution?

Strong Candidate: The time complexity is O(n) where n is the number of elements in the array. We're using a hash map to store values we've seen, which gives us O(1) lookups, and we only need to traverse the array once.

O(log n)

Interviewer: Could we optimize the space complexity?

O(n)

Strong Candidate: Yes, if the array is sorted, we could use a two-pointer approach to achieve O(1) space complexity, though that would require O(n log n) time for sorting if the array isn't already sorted. Let me implement that approach...

O(n log n)

Pro Tip
Always verbalize your complexity analysis as you develop your solution, not just at the end. This demonstrates analytical thinking throughout.

O(n²)

Common Complexities to Know Cold

Sorting Typically O(n log n)	**Binary Search** O(log n)
Graph Traversal O(V + E)	**Dynamic Programming** Often O(n²) or O(n×k)

Time complexity is more important than space in most interviews

ESSENTIAL

CHAPTER 02

Core Data Structures

The building blocks for efficient algorithms and elegant solutions

Foundation for FAANG problems

Why Core Structures Matter

> **90% of interview questions** directly test your understanding of data structure operations and trade-offs

- **Optimized Access Patterns**: Choosing the right structure determines your algorithm's efficiency
- **Problem Classification**: Recognizing which structure fits which problem is a critical interview skill
- **Implementation Details**: Understanding the internals helps you leverage built-in functions effectively

---

Arrays & Strings

Linked Lists

Trees & Graphs

Arrays & Strings – Concepts

Arrays

A contiguous collection of elements of the same type, stored in adjacent memory locations and accessed by numerical indices.

Key Characteristics

- Fixed size in most languages (dynamic in JavaScript, Python)
- Zero-indexed (first element at position 0)
- Elements accessed in O(1) time with index
- Contiguous memory allocation (cache-friendly)

Common Operations

Access: O(1) | Push/Pop end: O(1) | ! Insert/Delete middle: O(n) | Search: O(n)

```
const array = [1, 2, 3, 4, 5]; // Random access array[2];
// ⇒ 3
```

Common Use Cases

- Storing collections of similar items
- Implementing stacks and queues
- Buffer pools and caches
- Matrix/grid representations

Strings

A sequence of characters representing text. In many languages, strings are immutable (cannot be changed after creation).

Key Characteristics

- Immutable in many languages (Java, JavaScript, Python)
- Special operations for text manipulation
- Character encoding matters (ASCII, Unicode/UTF-8)
- Often implemented as character arrays internally

Common Operations

Access: O(1) | Search substring: O(n*m) | ! Concatenation: O(n+m)

```
const str = "hello"; // Substring operation
str.substring(1, 4); // ⇒ "ell"
```

Common Use Cases

- Text processing and manipulation
- Pattern matching and validation
- Data serialization (JSON, XML)
- Representing symbolic data

FAANG Interview Insights

Arrays and strings are the most commonly tested data structures – mastering these is essential. Pay special attention to two-pointer techniques and sliding window patterns.

Be comfortable with string manipulation methods like split, join, substring, and regex. Know when to convert between strings and arrays for efficient problem-solving.

Arrays & Strings form the foundation for many complex data structures

Arrays & Strings – Code Examples

Easy Medium Hard

Example 1: Reverse Array | Example 2: Palindrome | Example 3: Subarray Sum

Reverse an Array

Easy | Basic Array Manipulation

Problem Statement

Write a function that reverses an array in-place without using built-in array reverse methods.

Input/Output Example

Input: [1, 2, 3, 4, 5]
Output: [5, 4, 3, 2, 1]

Approach

- Use two pointers: one at the beginning, one at the end
- Swap elements at both pointers and move inward
- Continue until pointers meet or cross each other

Time: O(n) | Space: O(1) | In-place

Implementation

JavaScript | Copy

```javascript
/**
 * Reverses an array in-place
 * @param {Array} arr - The array to reverse
 * @return {Array} - The reversed array (same reference)
 */
function reverseArray(arr) {
  let left = 0;
  let right = arr.length - 1;

  while (left < right) {
    // Swap elements using destructuring
    [arr[left], arr[right]] = [arr[right], arr[left]];

    // Move pointers inward
    left++;
    right--;
  }

  return arr;
}

// Example usage
const arr = [1, 2, 3, 4, 5];
console.log(reverseArray(arr)); // [5, 4, 3, 2, 1]
```

FAANG Interview Tips

- Always discuss the time and space complexity of your solution
- Consider edge cases like empty arrays or single-element arrays
- If asked about optimizations, mention built-in methods like Array.reverse() but explain why they work the same way internally

Palindrome Check

Check if a string is a palindrome considering only alphanumeric characters...

Subarray Sum

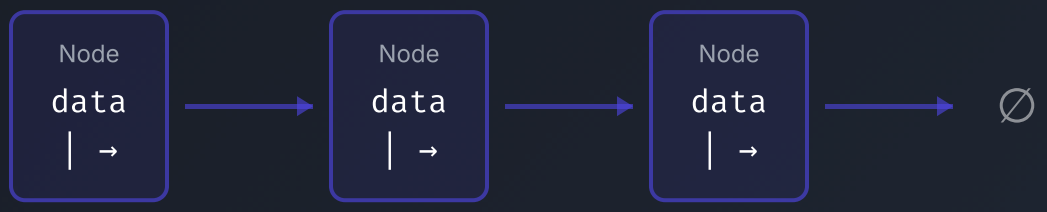
Find a contiguous subarray with a sum equal to a given target value...

Practice implementing these solutions without looking at the code

Linked Lists – Concepts

Core Data Structures

Singly Linked List



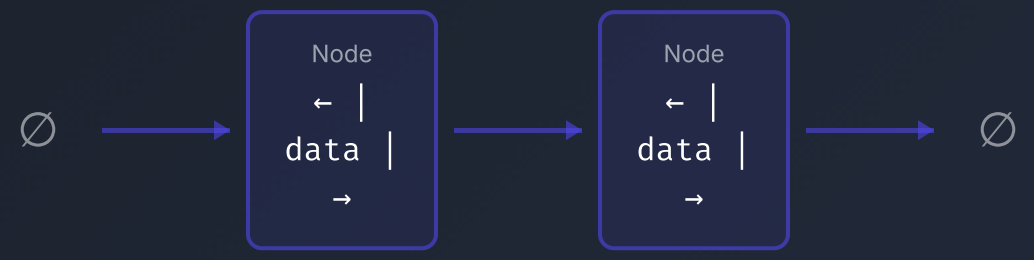
STRUCTURE

- Each node contains **data and a single pointer** to the next node
- Traversal is **one-directional** (forward only)
- Last node points to `null`

Node Implementation

```
class Node {
  constructor(data) {
    this.data = data;
    this.next = null;
  }
}
```

Doubly Linked List



STRUCTURE

- Each node contains **data and two pointers**: next and previous
- Traversal is **bi-directional** (forward and backward)
- First node's prev and last node's next point to `null`

Node Implementation

```
class Node {
  constructor(data) {
    this.data = data;
    this.next = null;
    this.prev = null;
  }
}
```

Key Operations & When to Use

Common Operations

- Insert at head: O(1)
- Delete from head: O(1)
- Insert/Delete at tail: O(1) for doubly, O(n) for singly
- Search by value: O(n)
- Access by index: O(n)

Advantages

- Dynamic size (grow/shrink as needed)
- Efficient insertions/deletions (no shifting)
- Memory efficient (can be allocated anywhere)
- Implementation of stacks/queues
- Doubly: backward traversal & O(1) deletions

When to Use

- **Singly**: Memory-constrained systems, stacks, simple forward traversal
- **Doubly**: Navigational operations (browser history), LRU cache, when frequent deletion is needed
- **Both**: When order matters but random access is rare

Consider space-time tradeoffs when choosing between linked lists vs arrays

Linked Lists – Code Examples

Easy Medium Hard

Example 1: Reverse List | Example 2: Detect Cycle | Example 3: Merge Lists

Reverse a Linked List

Medium | Linked List Manipulation

Problem Statement

Reverse a singly linked list iteratively. Return the new head of the reversed list.

Input/Output Example

Input: 1 → 2 → 3 → 4 → 5 → NULL
Output: 5 → 4 → 3 → 2 → 1 → NULL

Approach

- Track three pointers: previous, current, and next
- Iterate through list, reversing each pointer
- Keep track of next node before changing current's pointer

Time: O(n) | Space: O(1) | Iterative

Implementation

JavaScript | Copy

```javascript
/**
 * Definition for singly-linked list.
 * function ListNode(val, next) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.next = (next===undefined ? null : next)
 * }
 */
/**
 * @param {ListNode} head - Head of the linked list
 * @return {ListNode} - New head of reversed list
 */
function reverseList(head) {
  let prev = null;
  let current = head;

  while (current !== null) {
    // Save next before overwriting
    let nextTemp = current.next;

    // Reverse the pointer
    current.next = prev;

    // Move pointers one position ahead
    prev = current;
    current = nextTemp;
  }

  // New head is the previous pointer
  return prev;
}
```

FAANG Interview Tips

- Be ready to implement both iterative and recursive solutions
- Handle edge cases: empty list and single node list
- Draw out the step-by-step pointer movements to avoid confusion

Detect Cycle

Using Floyd's Cycle-Finding Algorithm (tortoise and hare) to detect if a linked list has a cycle...

Merge Two Lists

Merge two sorted linked lists into a single sorted linked list...

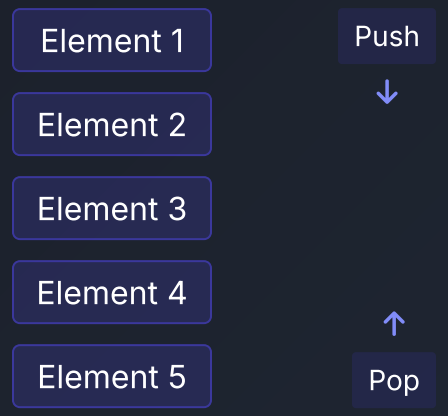
Linked list problems often test pointer manipulation skills

Stacks & Queues – Concepts

Core Linear Data Structures

Stack

Last In, First Out (LIFO)



Key Operations

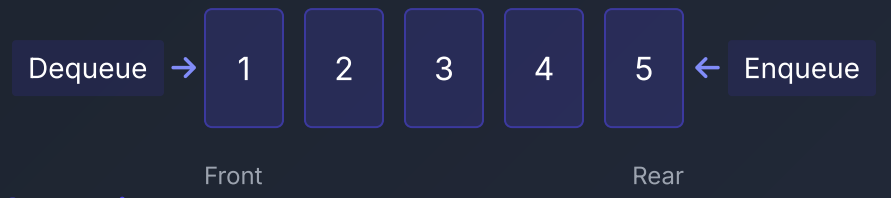
- **push(element)**
  Add to top - O(1)
- **pop()**
  Remove top - O(1)
- **peek()/top()**
  View top - O(1)
- **isEmpty()**
  Check if empty - O(1)

Applications

Function calls · Undo mechanism · Expression evaluation · Parentheses matching

Queue

First In, First Out (FIFO)



Key Operations

- **enqueue(element)**
  Add to rear - O(1)
- **dequeue()**
  Remove from front - O(1)
- **peek()/front()**
  View front - O(1)
- **isEmpty()**
  Check if empty - O(1)

Implementations & Variations

- Array-based
- Linked list-based
- Circular queue
- Priority queue

Applications: BFS, Task scheduling, Printing, Request processing

Stacks vs Queues: Quick Comparison

Stack

- Elements removed in reverse order of insertion
- Only one end is accessible (top)
- Think of a stack of plates - you can only take from the top

Queue

- Elements removed in same order as insertion
- Both ends are used (front and rear)
- Think of people in line - first come, first served

Try implementing a queue using two stacks - a common interview question

Stacks & Queues – Code Examples

Easy Medium Hard

Example 1: Valid Parentheses | Example 2: Min Stack | Example 3: Queue using Stacks

Valid Parentheses

Easy | Stack Implementation

Problem Statement

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

A string is valid if opening brackets are closed by the same type of closing brackets and in the correct order.

Input/Output Example

Input: "([{}])"
Output: true

Input: "([)]"
Output: false

Approach

- Use a stack to keep track of opening brackets
- When encountering an opening bracket, push it to the stack
- When encountering a closing bracket, check if the stack is empty or if the top element matches
- Return true if stack is empty after processing all characters

Time: O(n) Space: O(n)

Implementation

JavaScript Copy

```javascript
/**
 * Checks if a string of brackets is valid
 * @param {string} s - The string containing brackets
 * @return {boolean} - True if valid, false otherwise
 */
function isValid(s) {
  const stack = [];
  const bracketMap = {
    '(': ')',
    '[': ']',
    '{': '}'
  };

  for (let char of s) {
    // If it's an opening bracket
    if (bracketMap[char]) {
      stack.push(char);
    } else {
      // It's a closing bracket
      const lastBracket = stack.pop();

      // Check if the corresponding opening bracket matches
      if (bracketMap[lastBracket] !== char) {
        return false;
      }
    }
  }

  // Stack should be empty for valid string
  return stack.length === 0;
}

// Example usage
console.log(isValid("([{}])")); // true
console.log(isValid("([)]"));   // false
```

FAANG Interview Tips

- Clearly explain why a stack is the ideal data structure for this problem
- Consider edge cases like empty strings or strings with only one bracket
- Be prepared to extend the solution for additional bracket types or different validation rules

Min Stack

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time...

Queue using Stacks

Implement a queue using only two stacks for efficient enqueue and dequeue operations...

String pattern matching is a common application of stack data structures

Trees – Concepts

Hierarchical Data Structures

Tree Types

Binary Tree
A tree where each node has at most two children, referred to as left child and right child

Binary Search Tree (BST)
A binary tree with the key property: for any node, all keys in its left subtree are less than the node's key, and all keys in its right subtree are greater

Balanced Trees
Trees with height close to log(n), including AVL trees and Red-Black trees that maintain balance during operations

Key Properties

Height
The longest path from root to leaf
`O(log n) for balanced trees`

Depth
The distance from a node to the root

Complete Tree
All levels filled except possibly the last

Full Tree
Each node has 0 or 2 children

BSTs provide efficient search, insert and delete operations with average time complexity of O(log n), but can degrade to O(n) if unbalanced

Tree Traversal Types

Inorder
Left → Root → Right
`1, 2, 3, 4, 5`

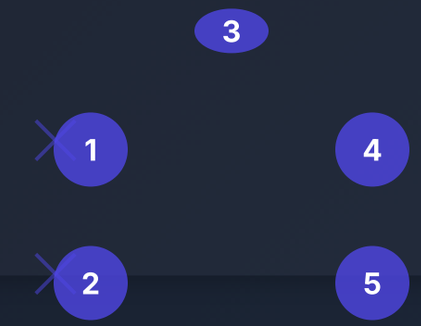
Preorder
Root → Left → Right
`3, 1, 2, 4, 5`

Postorder
Left → Right → Root
`2, 1, 5, 4, 3`

Depth-First Search (DFS)
- Uses stack (or recursion)
- Includes inorder, preorder, postorder
- Better for deep trees

Breadth-First Search (BFS)
- Uses queue
- Level-order traversal
- Better for shallow, wide trees

Binary Search Tree Example



For BST: All nodes to the left < parent, all nodes to the right > parent

BST operations (search, insert, delete) have average O(log n) time complexity

Trees – Code Examples

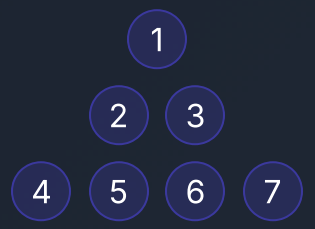
Inorder Preorder Postorder

Tree Traversals | BFS & DFS | Tree Operations

Tree Traversals

Medium | Binary Tree

Example Binary Tree



Traversal Results

- Inorder (Left → Root → Right): 4, 2, 5, 1, 6, 3, 7
- Preorder (Root → Left → Right): 1, 2, 4, 5, 3, 6, 7
- Postorder (Left → Right → Root): 4, 5, 2, 6, 7, 3, 1

Applications

- Inorder: Get elements in ascending order from a BST
- Preorder: Clone or serialize a tree structure
- Postorder: Delete a tree or evaluate expression trees

Time: O(n) | Space: O(h) | h = tree height

Implementation (Recursive)

JavaScript | Copy

```javascript
/**
 * Definition for a binary tree node
 */
class TreeNode {
  constructor(val = 0, left = null, right = null) {
    this.val = val;
    this.left = left;
    this.right = right;
  }
}

/**
 * Inorder traversal: Left → Root → Right
 * @param {TreeNode} root
 * @return {number[]}
 */
function inorderTraversal(root) {
```

FAANG Interview Tips

- Know both recursive and iterative approaches to tree traversals
- Understand the relationship between traversals and real-world applications
- For iterative solutions, practice implementing with stacks (DFS) or queues (BFS)

BFS & DFS

Breadth-first search vs depth-first search traversal methods...

Tree Operations

Insert, delete, and find operations in Binary Search Trees...

Practice identifying which traversal is needed based on problem requirements

Graphs – Concepts

One of the most frequently tested FAANG topics

Graph Representations

Adjacency List

Each vertex stores a list of its adjacent vertices

```
{
  0: [1, 2],
  1: [0, 2, 3],
  2: [0, 1],
  3: [1]
}
```

Space: O(V+E) | Better for sparse graphs

Adjacency Matrix

Boolean matrix where matrix[i][j]=1 if edge exists

```
[
  [0, 1, 1, 0],
  [1, 0, 1, 1],
  [1, 1, 0, 0],
  [0, 1, 0, 0]
]
```

Space: O(V²) | Better for dense graphs

Directed

Edges have direction (one-way)

Undirected

Edges have no direction (two-way)

BFS vs. DFS

Breadth First Search

- Explores layer by layer
- Uses a **queue** (FIFO)
- Finds shortest path in unweighted graphs

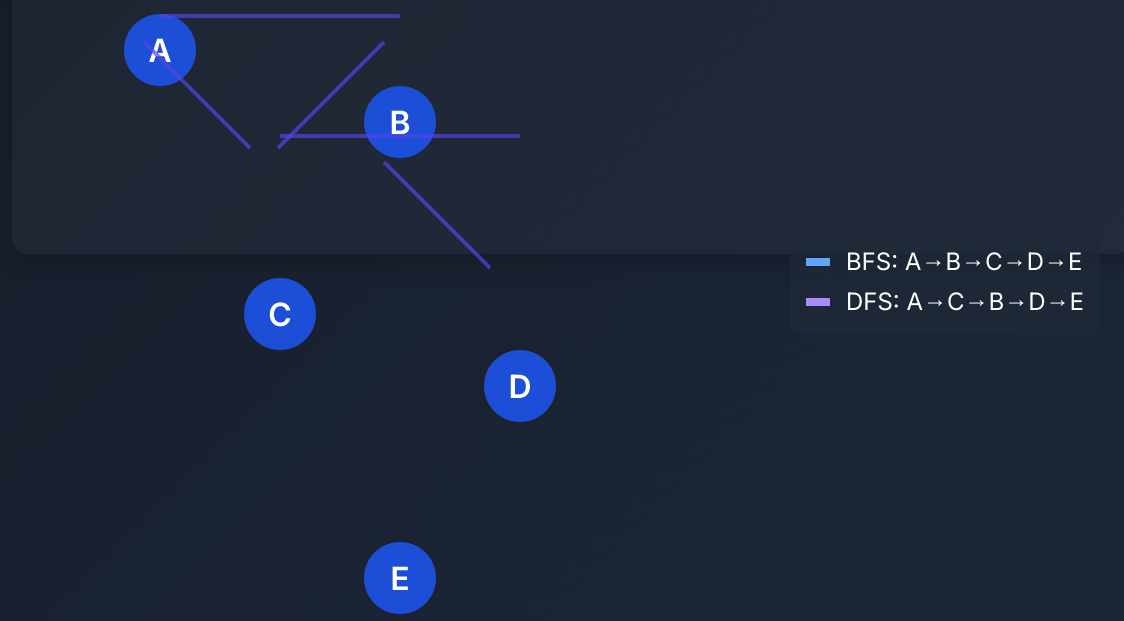
Time & Space: O(V+E)

Depth First Search

- Explores as far as possible first
- Uses a **stack** (LIFO) or recursion
- Useful for topological sorting, cycle detection

Time & Space: O(V+E)

Visual Representation



Common Applications in FAANG Interviews

Social Networks

Finding connections between users

Navigation

Finding shortest paths between locations

Task Scheduling

Using topological sorting to schedule tasks

Cycle Detection

Finding circular dependencies

Game AI

Using search to find optimal moves

Connected Components

Finding groups in networks

Always check for disconnected components in graph problems

Graphs – Code Examples

Medium · Graph Algorithms

Graph Traversal (DFS) | Cycle Detection | Connected Components

Graph Traversal with DFS

Medium | Depth First Search

Problem Statement

Implement a depth-first search (DFS) traversal algorithm for a graph represented as an adjacency list, and use it to detect if the graph has a cycle.

Graph Representation

```
// Graph represented as adjacency list
const graph = {
  0: [1, 2],
  1: [0, 3, 4],
  2: [0, 5],
  3: [1, 4],
  4: [1, 3],
  5: [2]
};
```

Approach

- Use DFS with a visited set to track nodes we've seen
- For cycle detection, also track nodes in the current path
- If we encounter a node that's in the current path, a cycle exists

Time: O(V + E) | Space: O(V) | Recursive

Implementation

JavaScript | Copy

```
/**
 * Performs DFS traversal on a graph
 * @param {Object} graph - Adjacency list representation
 * @param {number} start - Starting vertex
 */
function dfs(graph, start) {
  const visited = new Set();
  const result = [];

  function explore(vertex) {
    visited.add(vertex);
    result.push(vertex);

    for (const neighbor of graph[vertex]) {
      if (!visited.has(neighbor)) {
        explore(neighbor);
      }
    }
  }

  explore(start);
  return result;
}

/**
 * Detects if there's a cycle in an undirected graph
 * @param {Object} graph - Adjacency list
 * @return {boolean} - True if cycle exists
 */
function hasCycleUndirected(graph) {
  const visited = new Set();

  for (const vertex in graph) {
    if (!visited.has(Number(vertex))) {
      if (detectCycle(Number(vertex), -1)) {
        return true;
      }
    }
  }
  return false;

  function detectCycle(current, parent) {
    visited.add(current);

    for (const neighbor of graph[current]) {
      if (!visited.has(neighbor)) {
        if (detectCycle(neighbor, current)) {
          return true;
        }
      } else if (neighbor !== parent) {
        // Found a back edge (cycle)
        return true;
      }
    }
    return false;
  }
}
```

FAANG Interview Tips

- Be clear about directed vs undirected graph cycle detection differences
- If using recursive DFS, be prepared to implement iterative version with stack
- For directed graphs, explain how to use parent tracking vs in-progress set

Graph problems often appear in system design questions too

Hash Tables – Concepts

O(1) average lookup time

Hash Maps & Sets

Definition

A data structure that maps keys to values using a hash function that computes an index into an array where the value is stored

Performance

Lookup: O(1) average, O(n) worst case

Insert: O(1) average, O(n) worst case

Delete: O(1) average, O(n) worst case

Space: O(n)

Hash Function Properties

- Deterministic: Same input always produces same output
- Fast to compute: Should be O(1) time complexity
- Uniform distribution: Minimizes collisions

FAANG interviews often test hash table usage for optimizing solutions from $O(n^2)$ to O(n) time complexity, especially for lookups and existence checks.

Collision Handling

When two keys hash to the same index, a collision occurs. Common strategies to resolve:

Chaining

Each array position contains a linked list of entries that hash to the same index

```
bucket[i] → [k1,v1] → [k2,v2]
```

Open Addressing

Find another open slot in the array using probing techniques:

- Linear probing: Check next slot sequentially
- Quadratic probing: Check slots at quadratic distances
- Double hashing: Use second hash function to determine step size

Robin Hood Hashing

Variant of open addressing that minimizes the maximum distance of any key from its ideal position

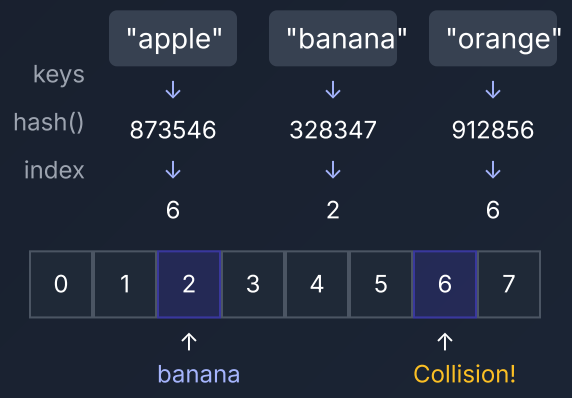
Common Use Cases

Fast Lookup
Two Sum, Dictionary

Deduplication
Remove duplicates, HashSet

Frequency Counting
Anagrams, Character count

Caching
Memoization, LRU Cache



HashMap & HashSet are built-in hash table implementations in most languages

Hash Tables – Code Examples

Easy Medium Hard

Example 1: Two Sum | Example 2: Valid Anagrams

Two Sum

Easy | Hash Table

Problem Statement

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

Input/Output Example

Input: nums = [2, 7, 11, 15], target = 9

Output: [0, 1]

Explanation: nums[0] + nums[1] = 2 + 7 = 9

Approach

- Use a hash table to store numbers we've seen and their indices
- For each number, check if its complement (target - number) exists in the hash table
- Return both indices when found

Time: O(n) | Space: O(n) | One-pass

Implementation

JavaScript | Copy

```javascript
/**
 * @param {number[]} nums - Array of integers
 * @param {number} target - Target sum
 * @return {number[]} - Indices of the two numbers
 */
function twoSum(nums, target) {
  // Hash map to store values we've seen and their indices
  const map = new Map();

  for (let i = 0; i < nums.length; i++) {
    // Calculate the complement we're looking for
    const complement = target - nums[i];

    // If the complement exists in our map, we found our answer
    if (map.has(complement)) {
      return [map.get(complement), i];
    }

    // Otherwise, add the current number to the map
    map.set(nums[i], i);
  }

  return []; // No solution found
}

// Example usage
const nums = [2, 7, 11, 15];
const target = 9;
console.log(twoSum(nums, target)); // [0, 1]
```

FAANG Interview Tips

- Discuss the brute force approach first ($O(n^2)$) to show you understand tradeoffs
- Explain why a hash table gives us O(1) lookups, making this a linear time solution
- Follow-up: How would you solve if the array was sorted? (Two-pointer approach)

Valid Anagrams

Given two strings, determine if one is an anagram of the other using hash tables...

Subarray Sum Equals K

Count the number of continuous subarrays whose sum equals k using prefix sums...

Hash tables are key to solving problems with O(1) lookups - great for pair finding and counting

Heaps & Priority Queues – Concepts

Complete binary tree implementation

Binary Heaps

Max Heap
Parent nodes are always greater than or equal to their children

Min Heap
Parent nodes are always less than or equal to their children

Key Properties

- **Complete Binary Tree**: All levels are fully filled except possibly the last level, which is filled left to right
- **Array Representation**: For zero-based indexing, parent at i has children at 2i+1 and 2i+2
- **Height**: Always log(n) for n elements, ensuring O(log n) operations

Binary heaps are typically implemented using arrays, not pointer-based structures like binary search trees, making them memory-efficient and cache-friendly.

Priority Queue Operations

insert(element)
Add at bottom, then bubble up to correct position
Time: O(log n)

peek()
Return top element (root) without removing
Time: O(1)

extractMax() / extractMin()
Remove root, replace with last element, bubble down
Time: O(log n)

heapify(array)
Convert a regular array into a valid heap structure
Time: O(n) when done optimally

Common Applications

K-th largest elements
Top-K problems, stream processing

Dijkstra's Algorithm
Shortest path in graphs

Heap Sort
O(n log n) in-place sorting

Event schedulers
Task prioritization systems

Python's heapq and Java's PriorityQueue implement min heaps

Heaps & Priority Queues – Code Examples

Easy Medium Hard

Example 1: Kth Largest | Example 2: Top K Elements | Example 3: Merge K Sorted Lists

Find Kth Largest Element

Medium | Heap / Priority Queue

Problem Statement

Given an unsorted array of integers, find the kth largest element. Note that it is the kth largest element in the sorted order, not the kth distinct element.

Input/Output Example

Input: [3,2,1,5,6,4], k = 2
Output: 5

Approach

- Use a min-heap of size k to keep track of k largest elements
- Iterate through the array, adding elements to the heap
- If heap size exceeds k, remove the smallest element
- The top of the heap will be the kth largest element

Time: O(n log k) | Space: O(k) | Common in FAANG

Implementation

JavaScript | Copy

```javascript
/**
 * Find the kth largest element in an unsorted array
 * @param {number[]} nums - The array of numbers
 * @param {number} k - The k value
 * @return {number} - The kth largest element
 */
function findKthLargest(nums, k) {
  // Create min heap (priority queue)
  const minHeap = [];

  for (let i = 0; i < nums.length; i++) {
    // Add current element to heap
    insertIntoMinHeap(minHeap, nums[i]);

    // If heap size exceeds k, remove smallest element
    if (minHeap.length > k) {
      extractMin(minHeap);
    }
  }

  // Top of min heap is kth largest
  return minHeap[0];
}

// Helper functions for min heap operations
function insertIntoMinHeap(heap, val) {
  heap.push(val);
  heapifyUp(heap, heap.length - 1);
}

function extractMin(heap) {
  const min = heap[0];
  heap[0] = heap[heap.length - 1];
  heap.pop();
  heapifyDown(heap, 0);
  return min;
}
```

FAANG Interview Tips

- In JavaScript, consider using a library or implementing a custom MinHeap class for cleaner code
- Mention the alternative QuickSelect approach (O(n) average time)
- For large arrays with small k, heap solution is more efficient; for small arrays, sorting might be simpler

Top K Frequent Elements

Given an array, return the k most frequent elements using heap-based approach...

Merge K Sorted Lists

Efficiently merge k sorted linked lists using min heap to track head nodes...

Heap problems appear in ~15% of FAANG interviews - master them!

⚠ Core Data Structures – Common Pitfalls & Optimization

Key insights to avoid common mistakes and optimize your solutions

Common Pitfalls to Avoid

Arrays & Strings
High Frequency

- Off-by-one errors in index handling
- Forgetting to handle empty array edge cases
- Inefficient string concatenation in loops

Linked Lists

- Missing null pointer checks
- Losing reference to head during manipulation
- Not handling single node edge cases

Trees & Graphs
High Frequency

- Forgetting to track visited nodes (infinite loops)
- Incorrect base case in recursive traversals
- Not accounting for disconnected components

Optimization Strategies

Hash Tables
Interview Favorite

- Use for O(1) lookups when order doesn't matter
- Store computed results to avoid recalculation
- Carefully choose hash key to reduce collisions

Time-Space Tradeoffs

- Consider preprocessing data for faster queries
- Use constant extra space techniques (in-place)
- Avoid unnecessary copies of large structures

Data Structure Selection

- Choose based on operations needed, not familiarity
- Consider specialized structures (Tries for strings)
- Be ready to combine multiple structures

FAANG Interview Insights

When in Doubt

Prefer clarity over cleverness. Explain tradeoffs explicitly rather than using an approach you can't articulate well.

Interviewer Signals

Watch for hints when they ask about performance – it's often a signal that your current approach can be optimized.

Final Verification

Always walk through your solution with test cases before declaring you're done – especially with edge cases.

"The right data structure can make a hard problem easy."

CRUCIAL

CHAPTER 04

Key Algorithms

Mastering the core algorithmic techniques most frequently tested in FAANG interviews

</> Beyond data structures to algorithmic thinking

Algorithm Categories Most Tested

Algorithmic problem-solving demonstrates your ability to recognize patterns and efficiently solve complex computational challenges

Sorting & Searching
From quicksort to binary search, these are fundamental building blocks

Dynamic Programming
Breaking problems into overlapping subproblems for optimal solutions

Greedy Algorithms
Making locally optimal choices for global optimization

Backtracking
Incrementally building solutions and abandoning failing paths

---

Algorithmic Fluency is Non-Negotiable

Interviewers assess whether you can identify the right algorithm for a problem, implement it correctly, and optimize it effectively

Sorting Algorithms – Concepts

Comparison of common sorting approaches

O(n) / O(n log n) O(n log n) O(n²) O(n²)

Bubble Sort

- Best: O(n)
- Avg: O(n²)
- Worst: O(n²)
- Space: O(1)

When to use: Educational purposes, nearly sorted data with small datasets

Selection Sort

- Best: O(n²)
- Avg: O(n²)
- Worst: O(n²)
- Space: O(1)

When to use: Small datasets, minimizing memory writes

Insertion Sort

- Best: O(n)
- Avg: O(n²)
- Worst: O(n²)
- Space: O(1)

When to use: Nearly sorted data, online algorithms (sorting as data arrives)

Merge Sort

- Best: O(n log n)
- Avg: O(n log n)
- Worst: O(n log n)
- Space: O(n)

When to use: Stable sort needed, linked lists, external sorting

Quick Sort

- Best: O(n log n)
- Avg: O(n log n)
- Worst: O(n²)
- Space: O(log n)

When to use: General purpose, arrays, average-case optimal

Heap Sort

- Best: O(n log n)
- Avg: O(n log n)
- Worst: O(n log n)
- Space: O(1)

When to use: Systems with memory constraints, worst-case guarantees

Counting Sort

- Best: O(n+k)
- Avg: O(n+k)
- Worst: O(n+k)
- Space: O(n+k)

When to use: Small range of integers, non-comparative sorting needed

Radix Sort

- Best: O(nk)
- Avg: O(nk)
- Worst: O(nk)
- Space: O(n+k)

When to use: Fixed-length integers, strings with known max length

FAANG Interview Focus

In FAANG interviews, you should know at least Quick Sort and Merge Sort implementations in detail. Be prepared to analyze tradeoffs between different algorithms and know when to choose which sorting approach for different data characteristics.

Most languages have built-in sort functions that use hybrid algorithms

Sorting Algorithms – Code Examples

Medium Difficulty

Merge Sort | Quick Sort

Merge Sort

Divide & Conquer Algorithm

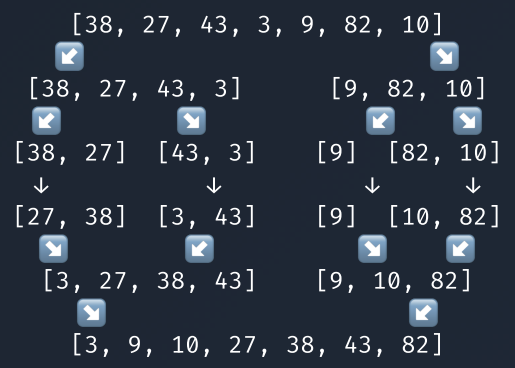
Algorithm Overview

Merge sort is a stable, comparison-based sorting algorithm that uses the divide-and-conquer principle.

How It Works

- Divide the array into two halves
- Recursively sort both halves
- Merge the sorted halves to produce a sorted array

Visual Representation



Time: O(n log n) | Space: O(n) | Stable

FAANG Interview Note

Merge sort is preferred when a stable sort is needed or when dealing with linked lists, as it minimizes the number of comparisons.

Implementation

JavaScript | Copy

```javascript
/**
 * Merge sort implementation
 * @param {Array} arr - The array to sort
 * @return {Array} - The sorted array
 */
function mergeSort(arr) {
  // Base case: arrays with 0 or 1 element are already sorted
  if (arr.length <= 1) {
    return arr;
  }

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

  // Recursive calls to sort the two halves
  return merge(
    mergeSort(left),
```

Key Interview Points

- Guaranteed O(n log n) performance in all cases
- Used in Java's Arrays.sort() for objects
- Additional O(n) space required
- Ideal for external sorting with large datasets

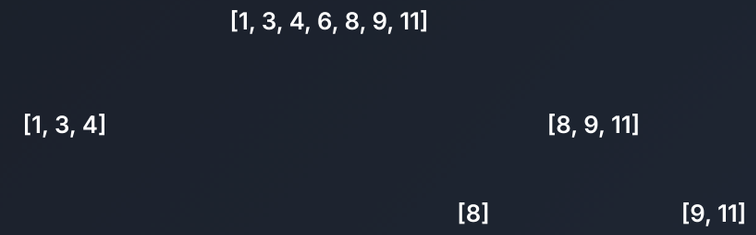
For linked lists, merge sort is often preferred over quick sort

Searching Algorithms – Concepts

Find with optimal complexity

Binary Search Foundation

Binary search is a divide-and-conquer algorithm that finds a target value in a **sorted array** by repeatedly dividing the search interval in half.



Key Property: Reduces search space by half with each comparison

Binary search is often buggy due to boundary conditions. Always check:

- Array boundaries (prevent out-of-bounds)
- Handling of duplicates
- Proper midpoint calculation (`mid = left + (right - left) / 2` to prevent overflow)

Binary Search Variations

Find First Occurrence

When target is found, continue searching left half

```
if (arr[mid] == target) right = mid - 1;
```

Find Last Occurrence

When target is found, continue searching right half

```
if (arr[mid] == target) left = mid + 1;
```

Rotated Sorted Array Search

Find pivot point, then apply binary search on the correct half

2D Matrix Search

Treat matrix as a 1D array using index transformation or apply binary search twice

Applications & Special Cases

- **Lower/Upper Bounds:** Find insertion position for a value (C++ STL equivalent functions)
- **Infinite Array Search:** Use doubling technique to find search bounds
- **Peak Element:** Find local maximum in an unsorted array

O(log n) time complexity - exceptionally efficient for large datasets

Searching Algorithms – Code Examples

Easy Medium Hard

Classic Binary Search | Lower/Upper Bound | Rotated Array Search

Binary Search

Easy | Searching Algorithm

Problem Statement

Implement the binary search algorithm to find the position of a target value in a sorted array. Return -1 if the target is not found.

Input/Output Example

Input: nums = [1, 3, 5, 7, 9], target = 5
Output: 2 (index of target)

Input: nums = [2, 4, 6, 8], target = 5
Output: -1 (target not found)

Approach

- Initialize left and right pointers at start and end
- Find middle element and compare with target value
- If target is found, return its index
- If target is smaller, search left half
- If target is larger, search right half

Time: O(log n) | Space: O(1)

Implementation

JavaScript | Copy

```javascript
/**
 * Binary search implementation
 * @param {number[]} nums - Sorted array of numbers
 * @param {number} target - Target value to find
 * @return {number} - Index of target or -1 if not found
 */
function binarySearch(nums, target) {
  let left = 0;
  let right = nums.length - 1;

  while (left <= right) {
    // Calculate middle index (prevents integer overflow)
    const mid = left + Math.floor((right - left) / 2);

    // Found target
    if (nums[mid] === target) {
      return mid;
    }
    // Search left half
    if (nums[mid] > target) {
      right = mid - 1;
    }
    // Search right half
    else {
      left = mid + 1;
    }
  }

  // Target not found
  return -1;
}
```

FAANG Interview Tips

- Be careful with boundary conditions: use left <= right to include the last element
- Calculate mid as left + Math.floor((right - left) / 2) to avoid integer overflow
- Binary search can be applied to any monotonic function, not just arrays

Lower/Upper Bound

Find the first or last position of a target value in a sorted array with duplicates...

Rotated Array Search

Search for a target value in a sorted array that has been rotated at some pivot point...

Binary search reduces the search space by half in each step

Recursion vs Iteration

When to use each approach in algorithm design

Common FAANG interview topic

Recursion

Function calls itself with modified parameters

When to Use

- **Tree-based algorithms**: Tree/graph traversals (DFS) and divide-and-conquer problems
- **Recursive data structures**: Trees, graphs, and nested objects
- **Backtracking problems**: Permutations, combinations, subsets
- **Elegant solution**: When the recursive solution is significantly more readable

Example: Factorial

```
function factorial(n) {
  // Base case
  if (n <= 1) {
    return 1;
  }

  // Recursive case
  return n * factorial(n - 1);
}
```

Tradeoffs

- ✓ Clear, concise code
- ✓ Natural for tree structures
- ✗ Stack overflow risk
- ✗ Overhead of function calls

Iteration

Using loops (for, while) to repeat operations

When to Use

- **Linear processing**: Arrays, lists, strings, and sequential data
- **Performance-critical code**: When optimizing for speed and memory usage
- **Large input sizes**: Avoids stack overflow with massive datasets
- **Simple counting/accumulation**: When tracking state through loops

Example: Factorial

```
function factorialIterative(n) {
  let result = 1;

  for (let i = 2; i <= n; i++) {
    result *= i;
  }

  return result;
}
```

Tradeoffs

- ✓ More memory efficient
- ✓ No stack overflow risk
- ✗ Can be harder to read
- ✗ Manual state tracking

Interview Tips

- ✓ Consider both approaches and discuss tradeoffs
- ✓ Consider stack space in complexity analysis
- ✓ Know how to convert between recursion and iteration
- ✓ Tail recursion can optimize recursive solutions

Remember: Most recursive solutions can be rewritten iteratively with an explicit stack

Dynamic Programming – Introduction

A powerful optimization technique

What is Dynamic Programming?

A technique for solving complex problems by breaking them down into simpler overlapping subproblems:

Optimal Substructure

The optimal solution can be constructed from optimal solutions of its subproblems

```
shortestPath(A→C) = shortestPath(A→B) +
shortestPath(B→C)
```

Overlapping Subproblems

Same subproblems are solved multiple times when finding the solution

```
Fibonacci: F(4) needs F(3) and F(2)
F(3) also needs F(2) → Overlapping!
```

When to consider using Dynamic Programming:

- Problems asking for optimization (max/min/longest/shortest)
- Counting the total number of ways or combinations
- Problems with future decisions depending on earlier ones

Implementation Approaches

Top-Down: Memoization

Recursive approach with caching of results

```
function fib(n, memo = {}) {
  // Check if already computed
  if (n in memo) return memo[n];
  if (n ⩽ 1) return n;
  return memo[n] = fib(n-1, memo) + fib(n-2,
memo);
}
```

Bottom-Up: Tabulation

Iterative approach building from base cases

```
function fib(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];
}
```

Memoization vs Tabulation Tradeoffs

Memoization	Tabulation
+ Easier to implement	+ Usually more efficient
+ Computes only what's needed	+ No recursion overhead
- Stack overflow risk	- May compute unneeded values
- Call stack overhead	- Sometimes harder to implement

FAANG tip: Practice recognizing when a problem can be solved with DP

Dynamic Programming – Code Patterns

Memoization Tabulation

Fibonacci Sequence | Coin Change

Fibonacci Sequence

Medium | Top-down | Bottom-up

Problem Statement

Calculate the n^{th} Fibonacci number in the sequence where F(0) = 0, F(1) = 1, and F(n) = F(n-1) + F(n-2) for n > 1.

Input/Output Example

Input: n = 6

Output: 8

Explanation: 0, 1, 1, 2, 3, 5, 8

DP Pattern

Top-down with Memoization

Cache results of subproblems to avoid redundant calculations

Bottom-up with Tabulation

Build solution iteratively from base cases

Time: O(n) Space: O(n)

Implementation

JavaScript Copy

```javascript
// Top-down approach (Memoization)
function fibonacci(n, memo = {}) {
  // Base cases
  if (n === 0) return 0;
  if (n === 1) return 1;

  // Check if we've already computed this value
  if (memo[n] !== undefined) {
    return memo[n];
  }

  // Calculate and store result
  memo[n] = fibonacci(n - 1, memo) + fibonacci(n - 2, memo);
  return memo[n];
}

// Bottom-up approach (Tabulation)
function fibonacciTabulation(n) {
  if (n === 0) return 0;

  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];
}
```

FAANG Interview Tips

- Recognize when a problem has overlapping subproblems (key DP indicator)
- Explain both approaches: recursive with memoization and iterative tabulation
- Optimize space with tabulation by only storing last two values instead of entire array

Coin Change Problem

Find the minimum number of coins needed to make up a given amount

Tabulation

DP problems are common in FAANG interviews; master both memoization and tabulation

Greedy Algorithms – Concepts

Make locally optimal choices for global solutions

Greedy Approach Rationale

A greedy algorithm builds up a solution by choosing the option that looks the best at every step:

- Make the locally optimal choice at each step
- Hope that these local choices lead to a globally optimal solution
- Never reconsider previous choices (unlike dynamic programming)

> ⚠ Greedy algorithms don't always yield optimal solutions. You must prove the greedy choice property holds for your problem.

When to Use Greedy?

Optimal Substructure
Optimal solution contains optimal sub-solutions

Greedy Choice Property
Local optimal choices lead to global optimal solution

$ Coin Change (Greedy works in some currencies)

For US coins [25¢, 10¢, 5¢, 1¢], always picking the largest coin first works to make optimal change.

```
41¢ = 25¢ + 10¢ + 5¢ + 1¢ (4 coins - optimal!)
```

Common Greedy Patterns

Activity Selection
Choose activities that finish earliest to maximize the number of activities performed.

</> Sort by end time, select non-overlapping activities

Fractional Knapsack
Take items with highest value-to-weight ratio first, can take fractions of items.

</> Sort by value/weight, take greedily from highest ratio

Huffman Coding
Build optimal prefix codes for data compression based on frequency.

</> Build tree bottom-up using priority queue

Interval Scheduling
Maximize the number of non-overlapping intervals or minimize the number of required resources.

</> Sort by end time or start time depending on problem

Identifying Greedy Problems

FAANG Frequently Tests!

- Can the problem be solved by making a sequence of choices?
- Does making the "best" local choice lead to an optimal solution?
- Can you prove the greedy choice is always safe?

Interview Tip
Always verbalize why a greedy approach works for the specific problem and consider counter-examples to test your solution.

! Not all problems with optimal substructure can be solved greedily (e.g., 0/1 Knapsack)

Greedy Algorithms – Code Examples

Easy Medium Hard

Interval Scheduling | Activity Selection | Minimum Platforms

Interval Scheduling

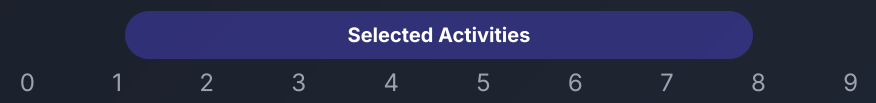
Medium | Classic Greedy Problem

Problem Statement

Given N activities with their start and finish times, select the maximum number of activities that can be performed by a single person, assuming only one activity can be worked on at a time.

Input/Output Example

Input: [(1,4), (3,5), (0,6), (5,7), (8,9), (5,9)]
Output: [(1,4), (5,7), (8,9)]



Greedy Approach

- Sort all activities by finish time
- Select the first activity (earliest finish)
- For remaining activities, select if start time ≥ previous activity's finish time

Time: O(n log n) | Space: O(1) | Sorting dominates

Implementation

JavaScript | Copy

```javascript
/**
 * Finds maximum number of activities one person can perform
 * @param {Array} activities - Array of [start, finish] pairs
 * @return {Array} - Selected activities
 */
function maxActivities(activities) {
  // Sort by finish time (second element in pair)
  activities.sort((a, b) => a[1] - b[1]);

  let selected = [activities[0]]; // First activity
  let lastFinishTime = activities[0][1];

  for (let i = 1; i < activities.length; i++) {
    // If current activity starts after previous one finishes
    if (activities[i][0] >= lastFinishTime) {
      selected.push(activities[i]);
      lastFinishTime = activities[i][1];
    }
  }

  return selected;
}

// Example usage
const activities = [[1,4], [3,5], [0,6], [5,7], [8,9], [5,9]];
console.log(maxActivities(activities));
```

FAANG Interview Tips

- Mention why sorting by finish time is optimal (enables more future activities)
- Be prepared to prove the greedy choice (exchange argument)
- Variations: weighted activities (profit maximization), room allocation problem

Meeting Rooms II
Given meeting intervals, find the minimum number of rooms required...

Merge Intervals
Merge all overlapping intervals and return non-overlapping intervals...

The earliest-finish-time rule guarantees optimal solution for this problem

Backtracking – Concepts

Popular in FAANG algorithm interviews

What is Backtracking?

A general algorithmic technique that incrementally builds candidates to solutions, and abandons a candidate ("backtracks") as soon as it determines the candidate cannot be extended to a valid solution.

The Backtracking Framework

1. Define State Space

Identify all possible configurations of your solution

2. Define Decision Space

At each step, determine the choices available to explore

3. Define Constraints

Set rules that determine valid solutions

4. Implement the Algorithm

- Make a choice and explore recursively
- If it leads to a valid solution, add to result
- If not valid or done exploring, undo the choice (backtrack) and try the next option

The key to backtracking is recognizing when to abandon a path. Pruning the search space early is critical for performance, especially during interviews.

Typical Interview Scenarios

Permutations & Combinations
Generate all possible arrangements or selections of elements

N-Queens Problem
Place N queens on an N×N chessboard so no two queens attack each other

Sudoku Solver
Fill a 9×9 grid with digits so each column, row, and 3×3 subgrid contains 1-9

Subset/Combination Sum
Find all subsets that meet specific criteria (e.g., sum to target)

Word Search
Find a word in a 2D grid by connecting adjacent characters

Interview Tip

When you recognize a backtracking problem, verbalize the template approach first. Interviewers want to see that you understand the pattern before diving into implementation details.

Time complexity is often O(N!) or O(2^N) — exponential in nature

Backtracking – Code Examples

Easy Medium Hard

Example 1: Permutations | Example 2: N-Queens | Example 3: Sudoku Solver

Generate Permutations

Medium | Classic Backtracking

Problem Statement

Given an array of distinct integers, return all possible permutations. You can return the answer in any order.

Input/Output Example

Input: [1, 2, 3]
Output: [[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1]]

Backtracking Approach

- Build permutations incrementally by choosing one element at a time
- Use a temporary array to track current permutation being built
- Use an auxiliary data structure to track used elements
- When a permutation is complete, add it to the result

Time: O(n!) | Space: O(n) | Decision Tree Depth: n

Implementation

JavaScript | Copy

```javascript
/**
 * @param {number[]} nums
 * @return {number[][]}
 */
function permute(nums) {
  const result = [];
  const path = [];
  const used = new Array(nums.length).fill(false);

  backtrack(nums, result, path, used);
  return result;
}

function backtrack(nums, result, path, used) {
```

FAANG Interview Tips

- Explain the backtracking template: choose → explore → unchoose
- Draw a decision tree to visualize the recursion and explain backtracking pruning
- Mention optimization options like Heap's algorithm for generating permutations

N-Queens

Place N queens on an N×N chessboard so no two queens attack each other...

Sudoku Solver

Solve a 9×9 Sudoku puzzle by filling empty cells with digits 1-9 following the rules...

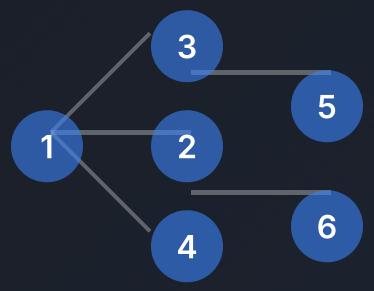
Visualize the recursion tree to better understand the backtracking process

Graph Algorithms – Concepts

Key traversal strategies for connected data

Breadth-First Search (BFS)

Time: O(V + E) Space: O(V)



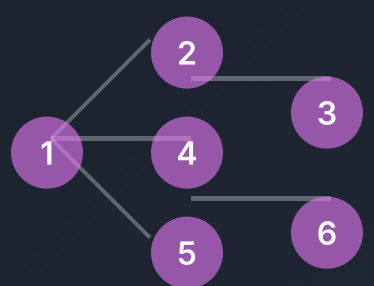
Visit Order: 1→2→3→4→5→6

Key Concept: Explores all neighbors at current depth before moving to nodes at next depth level

- **Implementation:** Uses a queue (FIFO) to track nodes to visit
- **Applications:** Shortest path in unweighted graphs, level-order traversal, connected components
- **FAANG Interview:** Commonly used for level-based problems, nearest neighbor searches

Depth-First Search (DFS)

Time: O(V + E) Space: O(V)



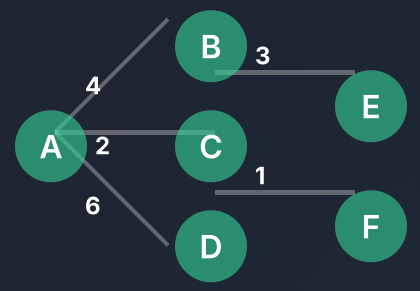
Visit Order: 1→2→3→4→5→6

Key Concept: Explores as far as possible along a branch before backtracking

- **Implementation:** Uses a stack (LIFO) or recursive function calls
- **Applications:** Cycle detection, topological sorting, path finding, maze generation
- **FAANG Interview:** Used for problems involving paths, connected regions, or exhaustive searches

Dijkstra's Algorithm

Time: O(V² or E log V) Space: O(V)



Shortest paths from A: B(4), C(2), D(6), E(7), F(7)

Key Concept: Finds shortest paths from source to all vertices in weighted graph with non-negative edges

- **Implementation:** Uses a priority queue to select next vertex with minimum distance
- **Optimizations:** Binary heap implementation reduces time complexity from O(V²) to O(E log V)

Core Implementation Steps

1. Initialize distances from source to all vertices as infinite
2. Set distance to source vertex as 0
3. Create a priority queue and enqueue source vertex
4. While priority queue is not empty:
   - Extract vertex with minimum distance
   - For each adjacent vertex, update distance if a shorter path is found

When to Use Each Algorithm

BFS	DFS	Dijkstra's
Shortest path in unweighted graphs	Cycle detection, exploring all paths	Shortest path in weighted graphs

Graph algorithms appear in ~15% of FAANG interview questions

Graph Algorithms – Code Examples

Medium Hard

Example 1: Dijkstra's Algorithm | Example 2: Topological Sort

Dijkstra's Algorithm

Medium | Shortest Path Finding

Problem Statement

Implement Dijkstra's algorithm to find the shortest path from a source vertex to all other vertices in a weighted graph.

Input/Output Example

Input: Graph with vertices [0,1,2,3,4], edges with weights, source=0
Output: Shortest distances [0,4,12,19,21]

Approach

- Use a priority queue (min-heap) to store vertices by distance
- Initialize distance to source as 0, all others as infinity
- For each vertex, relax all adjacent vertices
- Extract minimum distance vertex until queue is empty

Time: O((V+E)log V) | Space: O(V)

Implementation

JavaScript | Copy

```javascript
/**
 * Dijkstra's algorithm implementation
 * @param {Array>} graph - Adjacency list with [neighbor, weight]
 * @param {number} start - Starting vertex
 * @return {Array} - Array of shortest distances from start
 */
function dijkstra(graph, start) {
  const distances = new Array(graph.length).fill(Infinity);
  distances[start] = 0;

  // Min-heap priority queue [[vertex, distance]]
  const pq = [[start, 0]];

  while (pq.length > 0) {
    // Extract vertex with minimum distance
    pq.sort((a, b) => a[1] - b[1]);
    const [current, dist] = pq.shift();

    // Skip if we've found a better path already
    if (dist > distances[current]) continue;

    // Process all neighbors of current vertex
    for (const [neighbor, weight] of graph[current]) {
      const newDistance = dist + weight;

      // Found a better path
      if (newDistance < distances[neighbor]) {
        distances[neighbor] = newDistance;
        pq.push([neighbor, newDistance]);
      }
    }
  }

  return distances;
}
```

FAANG Interview Tips

- Mention that a real priority queue implementation would improve efficiency
- Explain that Dijkstra doesn't work with negative weights (use Bellman-Ford instead)
- Be ready to modify the algorithm to return the actual path, not just distances

Topological Sort

Medium

Implementation for DAGs to find a linear ordering of vertices such that for every directed edge (u,v), vertex u comes before v in the ordering.

Graph algorithms frequently appear in system design interviews as well

Algorithm Optimization & Interview Tips

How to refactor for performance and communicate effectively during interviews

Optimization Techniques

Space-Time Tradeoffs

Use additional data structures (hash maps, sets) to reduce time complexity at the expense of space

Early Termination

Add exit conditions to break loops once goal is achieved rather than processing all elements

Algorithmic Upgrades

Replace bubble sort ($O(n^2)$) with merge sort ($O(n \log n)$), or linear search ($O(n)$) with binary search ($O(\log n)$)

Code Refactoring

Refactoring Checklist

- [x] Eliminate unnecessary nested loops when possible
- [x] Use appropriate data structures for operations (e.g., sets for lookups)
- [x] Precompute values instead of recalculating repeatedly
- [x] Replace recursive solutions with iterative ones to avoid stack overflow
- [x] Use dynamic programming to avoid redundant calculations

Interview Communication

Start Simple

Begin with a working solution, then discuss optimizations

Think Aloud

Verbalize your optimization thought process

Discuss Tradeoffs

Explain pros and cons of different approaches

Be Receptive

Listen to interviewer hints and incorporate feedback

Common Pitfalls

- **Premature Optimization:** Focus on correctness first, then optimize
- **Over-engineering:** Don't make solutions more complex than needed
- **Silence:** Avoid long periods without communication

Interviewer Expectations

FAANG interviewers don't just want optimal solutions; they want to see your **problem-solving approach** and how you **iteratively improve** your algorithm.

> *"Perfect is the enemy of good enough. A working solution that you can optimize is better than an incomplete optimal one."*

ESSENTIAL

CHAPTER 06

Problem-Solving Patterns

Recognizing and applying algorithmic patterns to unlock FAANG interview challenges

From individual techniques to patterns

Why Patterns Matter

> **Pattern recognition** is the meta-skill that separates average programmers from exceptional problem-solvers at FAANG companies

- **Mental Models**: Patterns provide templates that can be applied to unfamiliar problems
- **Efficiency**: Pattern recognition dramatically reduces time-to-solution in high-pressure interviews
- **Transferability**: Core patterns work across different domains and problem types

Two Pointers

Sliding Window

Dynamic Programming

Two Pointer & Sliding Window Patterns

High-frequency interview patterns

Two Pointer Pattern

Uses two pointers to iterate through data structures efficiently, typically moving toward or away from each other.

- **Common Applications**
  - Reversing arrays/strings
  - Finding pairs with target sum
  - Removing duplicates
  - Palindrome verification

Visual Example: Two Sum (sorted array)

Target = 12, check if L+R = target, then move pointers

```
function twoSum(arr, target) {
  let left = 0;
  let right = arr.length - 1;

  while (left < right) {
    const sum = arr[left] + arr[right];
    if (sum === target) return [left, right];
    if (sum < target) left++;
    else right--;
  }
  return [-1, -1];
}
```

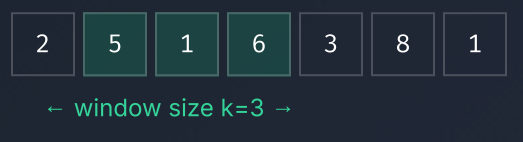
Time: O(n) Space: O(1)

Sliding Window Pattern

Maintains a "window" of elements that expands or contracts based on conditions, avoiding redundant computations.

- **Common Applications**
  - Maximum/minimum subarray of size k
  - Longest substring with distinct chars
  - Subarray with given sum
  - String pattern matching

Visual Example: Max Sum Subarray of Size K



Slide window by adding next element and subtracting leftmost element

```
function maxSubarraySum(arr, k) {
  let maxSum = 0;
  let windowSum = 0;

  // First window sum
  for (let i = 0; i < k; i++) {
    windowSum += arr[i];
  }
  maxSum = windowSum;

  // Slide window
  for (let i = k; i < arr.length; i++) {
    windowSum = windowSum + arr[i] - arr[i-k];
    maxSum = Math.max(maxSum, windowSum);
  }
  return maxSum;
}
```

Time: O(n)

Two Pointer FAANG Examples

Container With Most Water · 3Sum · Valid Palindrome · Remove Duplicates

Sliding Window FAANG Examples

Longest Substring Without Repeating Characters · Minimum Window Substring · Permutation in String

Many sliding window problems can be optimized from $O(n^2)$ to $O(n)$

Fast & Slow Pointer / Cycle Detection

Also known as Tortoise & Hare Algorithm

Technique Overview

The Fast & Slow pointer technique uses two pointers moving at different speeds through a sequence to solve problems related to cycles and sequence patterns.

Key Principles

- **Two Pointers**: One moves at single speed (slow), another at double speed (fast)
- **Intersection**: If pointers meet, a cycle exists in the structure
- **Finding Cycle Start**: Reset slow pointer to head and move both at same speed

Common Applications

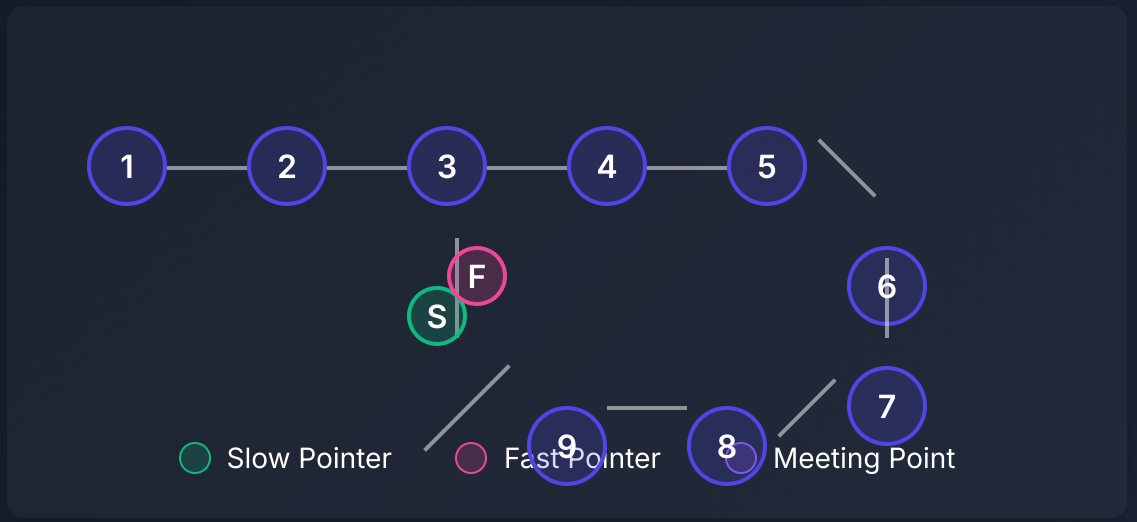
Cycle Detection
Linked Lists, Arrays

Middle Element
Linked Lists

Happy Number
Numeric Sequence

Find Duplicate
Floyd's Algorithm

Cycle Detection Visualization



Implementation Pattern

```
function hasCycle(head) {
  let slow = head;
  let fast = head;

  while (fast && fast.next) {
    slow = slow.next;        // move one step
    fast = fast.next.next;   // move two steps

    if (slow === fast) {
      return true;  // cycle detected!
    }
  }

  return false;  // reached end, no cycle
}
```

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

Classic FAANG interview question: Detect a cycle in a linked list

Prefix Sum / Difference Array

Optimize Range Queries

Q: What are Prefix Sums and why are they important?

A prefix sum array stores cumulative sums of elements, allowing O(1) range sum queries after O(n) preprocessing:

```
prefixSum[i] = arr[0] + arr[1] + ... + arr[i]
```

This transforms expensive repeated summation operations into simple subtraction: sum(i,j) = prefixSum[j] - prefixSum[i-1]

Q: How do Difference Arrays work for range updates?

A difference array enables O(1) range updates with O(n) reconstruction:

```
diff[i] = arr[i] - arr[i-1] // with arr[-1] = 0
```

To add value val to range [i,j]: Set diff[i]+=val and diff[j+1]-=val, then reconstruct the array.

Q: When should I use these techniques in interviews?

Look for these signals in problem descriptions:

- Questions about "range sums" or "subarrays with sum equal to X"
- Multiple range queries on a static array
- Multiple range updates followed by array value lookups
- Problems involving cumulative values or running totals

Q: What common interview problems use these techniques?

Prefix Sums are used in:

- Subarray Sum Equals K
- Maximum Subarray
- Range Sum Query - Immutable

Difference Arrays are used in:

- Corporate Flight Bookings
- Range Addition
- Car Pooling

Prefix Sum Visualization

Original Array:

3	1	4	1	5	9	2

Prefix Sum Array:

3	4	8	9	14	23	25

Sum of range [2,4] = ?

prefixSum[4] - prefixSum[1] = 14 - 4 = 10

3	1	4	1	5	9	2

Interview Tip
When implementing a prefix sum, include a 0 at the beginning of your array (prefixSum[0] = 0) to simplify range calculations.

Common Use Cases

Count Subarrays with Sum K
Use prefix sum + hash map: O(n) time

2D Prefix Sums for Matrix Queries
Compute sums for rectangles in O(1)

```
function createPrefixSum(arr) {
  const prefix = [0];  // Start with 0
  for (let i = 0; i < arr.length; i++) {
    prefix.push(prefix[i] + arr[i]);
  }
  return prefix;
}
```


Prefix sums are especially powerful when combined with hash maps

Divide and Conquer

Framework & Applications

The Framework

1. Divide

Break the original problem into smaller sub-problems of the same type

2. Conquer

Recursively solve these sub-problems independently

3. Combine

Merge the solutions of sub-problems to form the solution to the original problem

When to use Divide & Conquer:

- Problem can be broken down into identical sub-problems
- Sub-problems are independent (no overlapping)
- Solutions can be efficiently combined

Recursive Structure:

```
function divideAndConquer(problem) {
  // Base case
  if (problem is small enough)
    return solve(problem)

  // Divide
  subproblems = divide(problem)

  // Conquer (recursive call)
  results = []
  for each subproblem in subproblems:
    results.add(divideAndConquer(subproblem))

  // Combine
  return combine(results)
}
```

Common Applications

Merge Sort

Divide array in half, sort each half, then merge sorted halves

Time: O(n log n) | Space: O(n)

Binary Search

Divide search space in half based on comparison with middle element

Time: O(log n) | Space: O(1) or O(log n) recursive

Strassen's Matrix Multiplication

Divide matrices into submatrices and use recursive formulas

Time: O(n^2.81) | Better than naive O(n^3)

Closest Pair of Points

Find closest pair among points in a plane by dividing plane

Time: O(n log n) | Better than brute force O(n^2)

FAANG Interview Strategy

- Always explain the recursive tree and its depth
- Analyze base case carefully
- Derive time complexity using recurrence relation
- Identify problems where D&C gives optimal performance

Master the divide & conquer mindset to tackle complex problems systematically

Bit Manipulation

Key tricks and patterns used in coding interviews

Often reduces time complexity

Essential Bit Operations

AND (&)

Sets bit to 1 if both bits are 1

```
if (num & 1) == 1) // Check if odd
```

OR (|)

Sets bit to 1 if either bit is 1

```
num |= (1 << 4); // Set 4th bit to 1
```

XOR (^)

Sets bit to 1 if bits are different

```
x ^= y; y ^= x; x ^= y; // Swap variables
```

Left Shift (<<)

Shifts bits left by specified positions

```
int mask = 1 << 3; // Create mask with bit at position 3
```

Right Shift (>>)

Shifts bits right by specified positions

```
int div4 = num >> 2; // Divide by 4 (2^2)
```

NOT (~)

Flips all bits (0→1, 1→0)

```
int allButKthBit = ~(1 << k); // All 1s except kth bit
```

Common Bit Manipulation Patterns

Set/Check/Clear Individual Bits

Set i-th bit:	`num |= (1 << i)`
Check i-th bit:	`if (num & (1 << i))`
Clear i-th bit:	`num &= ~(1 << i)`
Toggle i-th bit:	`num ^= (1 << i)`

Common Interview Techniques

▸ Find single number among duplicates

XOR all numbers; duplicates cancel out

```
for (int num : nums) result ^= num;
```

▸ Count number of set bits (1's)

Brian Kernighan's Algorithm

```
while (n) { count++; n &= (n-1); }
```

Power of 2 Check

`0 0 0 0 1 0 0 0` Powers of 2 have exactly one bit set

```
boolean isPowerOfTwo(int n) {
  return n > 0 && (n & (n - 1)) == 0;
}
```

FAANG Interview Tip

Bit manipulation often provides optimal O(1) space solutions to problems involving flags, states, or sets. Always consider if a bit manipulation approach could simplify your solution.

CRUCIAL

CHAPTER 06

System Design Fundamentals

Beyond algorithms: designing scalable systems for FAANG-level technical interviews

Senior+ level interviews

Why System Design Matters at FAANG

> System design interviews evaluate your ability to architect complex, scalable solutions that would work in real-world production environments

- **Scalability & Performance**: Can your design handle millions of users and petabytes of data?
- **Trade-off Analysis**: How well do you navigate consistency, availability, and partition tolerance (CAP theorem)?
- **High-Level Thinking**: Can you zoom out from code-level details to architectural considerations?

Core Components to Master

System design builds upon DSA by applying efficient algorithms within distributed systems, focusing on load balancing, caching strategies, database sharding, and microservice architecture

ESSENTIAL

BEYOND CODE

Behavioral Interview Strategies

Showcasing your soft skills and cultural fit is equally important as your technical abilities

Complementing your technical skills

The STAR Method Framework

Structured storytelling helps interviewers assess your past behaviors as predictors of future performance

S: Situation
Set the context with specific background details

T: Task
Describe your responsibility or challenge

A: Action
Explain the steps you personally took

R: Result
Share outcomes with metrics when possible

- **Values Alignment**: FAANG companies assess your alignment with their core values like ownership, innovation, and customer focus
- **Leadership Principles**: Prepare stories that demonstrate leadership even without formal authority
- **Conflict Resolution**: Show how you've successfully navigated disagreements with data-driven approaches

Best Practice Resources & Mock Interview Platforms

Use these platforms strategically in your preparation

Top Platforms for DSA Practice

LeetCode — Essential

Gold standard for technical interview prep with 2000+ problems and company-specific question banks

Pro Tip: Focus on the "Blind 75" list first, then company-specific questions

HackerRank — OA Prep

Common platform for online assessments with structured track-based learning paths

Benefit: Mimics actual online assessment environments used by companies

Pramp — Mock Interviews

Free peer-to-peer mock interviews with structured feedback and rotation system

Key Feature: Take turns as interviewer/interviewee to understand both perspectives

Exponent — Paid Coaching

Expert-led mock interviews with FAANG engineers and comprehensive feedback

Value: Real interview experience with actionable, personalized insights

Blind — Community

Anonymous tech community with insider information on current interview questions

Insider Info: Recent interview experiences and company-specific patterns

AlgoExpert — Structured

Curated selection of 160+ hand-picked questions with video explanations

Best For: Visual learners who prefer guided, in-depth explanations

How to Use Efficiently

3-Phase Approach

1. **Learning Phase**
   Master fundamentals through structured problems, videos, and reading solutions
2. **Practice Phase**
   Solve 100-150 problems across different patterns, starting with easy then medium
3. **Mock Phase**
   Conduct 10-15 mock interviews in real-world conditions with timed practice

Effective Strategy
Instead of random problem solving, group questions by pattern (Two Pointers, DP, Trees, etc.) to recognize underlying themes

Problem Counts by Experience Level

Beginner: ~50 problems	Intermediate: 100-150 problems
Advanced: 200+ problems	Expert: 300+ problems

Final Tips & Success Checklist

Your final preparation roadmap for FAANG interview success

Interview Preparation Checklist

- [x] Review your top 20 most common DSA problems (sorted by frequency)
- [x] Practice verbally explaining your thought process out loud
- [x] Revisit system design fundamentals for senior roles
- [x] Prepare 3-5 personal stories using the STAR method
- [x] Test your code using multiple test cases including edge cases
- [x] Research the specific company's interview format and values
- [x] Sleep well and arrive early (virtually or in-person)

During The Interview

1. **Clarify the problem** - Ask questions about constraints, edge cases, and expected output
2. **Think aloud** - Verbalize your thought process as you consider different approaches
3. **Start with brute force** - Then optimize step-by-step (shows problem-solving)
4. **Test your solution** - Using multiple examples including edge cases
5. **Analyze complexity** - Discuss time and space complexity confidently

Last-Minute DSA Review Focus

Graph Traversals
BFS/DFS implementations and use cases

Dynamic Programming
Memoization vs tabulation approaches

Two-Pointers
For array/string manipulation problems

Hash Tables
O(1) lookup optimization techniques

Remember This

Your DSA preparation journey has already put you ahead of **90%** of candidates. Trust your preparation, stay calm, and showcase your problem-solving approach.

Your Progress:

- [x] Core DSA fundamentals mastered
- [x] Common patterns recognized
- [x] Problem-solving approach structured
- [ ] Continuous improvement mindset activated

"The best way to predict your future is to create it."

Best of luck on your FAANG interview journey!