Coding Interview Prep

PDF auto-generated from [Shawon Notes](). If you found it useful, please consider contributing to the project in [Github]().

Chapter 1: Introduction

Why This Guide?

Cracking a coding interview requires more than just knowing syntax—it demands structured problem-solving, a deep understanding of patterns, and the ability to compare multiple solutions effectively. This guide is designed to help you navigate coding interviews with a **Python-focused approach**, emphasizing **efficient solutions, trade-offs, and best practices**.

Problem-Solving Patterns for Structured Learning

Instead of solving problems randomly, this guide categorizes them into **common coding patterns** such as **Sliding Window, Two Pointers, Dynamic Programming, Graph Traversal, and Backtracking**. Recognizing these patterns allows you to:

- Solve **new problems faster** by identifying their underlying structure.
- Reduce the need to memorize individual solutions.
- Approach unfamiliar problems with a **systematic framework**.

Multiple Solutions with Trade-Offs

For each problem, this guide does more than just present a single answer—it explores:

- **Brute-force solutions** to establish a baseline.
- **Optimized approaches** using patterns and advanced techniques.
- **Trade-offs between time complexity, space usage, and readability**.
- **Alternative strategies**, such as iterative vs. recursive solutions.

By the end, you'll not only be able to **solve** problems but also **defend your choices in an interview** with clear justifications.

How to Use This Guide

Mastering coding interviews is not about memorizing solutions but about **understanding patterns and trade-offs**. This guide is structured to help you achieve this through a **progressive learning approach**.

Step 1: Study Patterns Before Solving Problems

Each chapter focuses on a specific **coding pattern**. Before diving into problems, study the pattern's core **concepts, common use cases, and variations**.

Step 2: Solve Problems with a Structured Approach

For each problem, follow a structured **four-step approach**:

1. **Understand the problem** – Identify constraints and edge cases.
2. **Develop a brute-force solution** – Establish a baseline.
3. **Optimize using patterns** – Apply the best pattern for efficiency.
4. **Analyze trade-offs** – Compare time, space, and readability.

Step 3: Compare Different Solutions

After solving a problem, don't stop at just one solution—explore **alternative approaches** and analyze **when each is preferable**.

Step 4: Reinforce Learning with Mock Interviews

Once comfortable with patterns, simulate **real interview conditions** by:

- Practicing with a **time limit** (e.g., 30–45 minutes per problem).
- Explaining your thought process **out loud** as if in an interview.
- Writing code in a **plain text editor or whiteboard** without autocompletion.

Interview Process Overview

1. Coding Round

The **first stage** of technical interviews typically involves **solving coding problems** on platforms like **LeetCode, HackerRank, or a company's own system**. This guide prepares you for this round by covering:

- **Algorithms and data structures** (arrays, graphs, trees, etc.).
- **Problem-solving techniques** (dynamic programming, greedy, etc.).
- **Writing efficient, bug-free code** under time constraints.

2. System Design (For Senior Roles)

For mid-level and senior roles, companies expect candidates to **design scalable systems**. Topics covered include:

- **Scalability principles** (caching, load balancing, sharding).
- **Database design and indexing**.
- **Trade-offs in distributed architectures**.This guide focuses primarily on **coding interviews**, but system design is briefly touched upon for reference.

3. Behavioral Interviews

Many candidates focus only on coding and neglect **behavioral questions**, which are critical for landing offers, especially at **FAANG companies**.

- Use the **STAR framework** (Situation, Task, Action, Result) to structure answers.
- Expect questions on **teamwork, leadership, and handling challenges**.
- Practice **concise but detailed responses** with real experiences.

How to Approach Problems Effectively

- **Clarify the problem** before jumping to coding.
- **Write a plan** (pseudocode or an outline) before implementation.
- **Discuss trade-offs and edge cases** during your explanation.
- **Optimize only after getting a working solution**.

- **Practice thinking out loud**, since interviewers want to hear your thought process.

This guide is designed to **give you a structured path to success**—whether you're preparing for FAANG, startups, or any top tech company. By following this methodology, you'll **build problem-solving intuition and confidently tackle any interview challenge.** 🚀

Chapter 2: Sliding Window

Concept & When to Use

The **Sliding Window** technique is a powerful approach for optimizing problems involving **contiguous subarrays or substrings**. Instead of using nested loops to repeatedly compute values, we maintain a **window (a range of indices)** that dynamically expands and contracts as we iterate through the input.

When to Use Sliding Window

Use this pattern when:

✔ **The problem involves subarrays or substrings.**

✔ **You need to find an optimal subarray (max/min sum, longest/shortest length).**

✔ **A brute-force approach involves recomputing overlapping parts of an array.**

Types of Sliding Window Approaches

There are two main types of sliding window approaches:

🔹 **Fixed-size Sliding Window**

- The window size is **predetermined** and remains **constant** throughout the iteration.
- Used in problems where we are asked to compute values over a **fixed-length subarray** (e.g., "find the max sum of a subarray of size k").

🔹 **Variable-size Sliding Window (Expanding/Shrinking Window)**

- The window size **changes dynamically** based on the problem constraints.
- Used when trying to find **the shortest/longest subarray** that meets a condition (e.g., "find the smallest subarray with a sum $\geq S$").

Grind 75 Problems

The **Sliding Window** pattern appears in multiple **Grind 75 problems**, such as:

1. **Maximum Sum Subarray of Size K** (Fixed-size) (LeetCode #643)
2. **Longest Substring Without Repeating Characters** (Variable-size) (LeetCode #3)
3. **Minimum Window Substring** (Variable-size) (LeetCode #76)

Below, we analyze each problem and discuss brute-force vs. optimized solutions.

1. Fixed-size Sliding Window

Problem: Maximum Sum Subarray of Size K

💡 **Given an array `nums` and an integer `k`, find the maximum sum of any contiguous subarray of size `k`.**

Brute-Force Approach (O(n × k))

- Iterate through all possible subarrays of length `k`.
- Compute their sums and track the maximum sum.
- **Time Complexity:** $O(n \times k)$ – inefficient for large arrays.

Optimized Sliding Window Approach (O(n))

- Maintain a **running sum** of size `k`.
- As the window slides, **remove the leftmost element** and **add the new rightmost element**.
- **Time Complexity:** $O(n)$ – each element is processed once.
- **Space Complexity:** $O(1)$ – only a few variables are used.

Python Implementation

```python
def maxSumSubarray(nums, k):
    max_sum, window_sum = 0, sum(nums[:k])
```

```
    for i in range(k, len(nums)):
        window_sum += nums[i] - nums[i - k]
        max_sum = max(max_sum, window_sum)

    return max_sum

# Example
nums = [2, 1, 5, 1, 3, 2]
k = 3
print(maxSumSubarray(nums, k))  # Output: 9
```

🚀 Trade-offs:

- Uses constant space (O(1)), but requires careful index management.
- **Efficient alternative** to recomputing sums for every subarray.

2. Variable-size Sliding Window

Problem: Longest Substring Without Repeating Characters

💡 **Given a string `s`, find the length of the longest substring without repeating characters.**

Brute-Force Approach (O(n²))

- Try **all substrings** and check for duplicates.
- **Time Complexity:** O(n²) – inefficient.

Optimized Sliding Window Approach (O(n))

- Use a **hash set** to track unique characters.
- Expand the window (`right` pointer) while characters are unique.
- When a duplicate is found, **shrink the window** (`left` pointer).
- **Time Complexity:** O(n) – each character is processed twice.
- **Space Complexity:** O(min(n, 26)) ≈ O(1), since we store at most 26 letters.

Python Implementation

```python
def lengthOfLongestSubstring(s):
    char_set = set()
    left, max_length = 0, 0

    for right in range(len(s)):
        while s[right] in char_set:
            char_set.remove(s[left])
            left += 1
        char_set.add(s[right])
        max_length = max(max_length, right - left + 1)

    return max_length

# Example
s = "abcabcbb"
print(lengthOfLongestSubstring(s))  # Output: 3
```

🚀 Trade-offs:

- Uses **extra space** for `char_set`, but ensures **O(n) performance**.
- Works efficiently for **ASCII characters** but may need modifications for Unicode.

3. Variable-size Sliding Window (Shrinking)

Problem: Minimum Window Substring

💡 **Given two strings `s` and `t`, find the smallest substring of `s` that contains all characters of `t`.**

Brute-Force Approach (O(n² × m))

- Try all substrings and check if they contain all characters of `t`.
- **Time Complexity:** $O(n^2 \times m)$ – inefficient.

Optimized Sliding Window Approach (O(n))

- Maintain a **hash map** of character frequencies in `t`.
- Expand the **right pointer** until all characters are included.
- Shrink the **left pointer** to minimize the window.

- **Time Complexity:** O(n) – each character is processed at most twice.
- **Space Complexity:** O(1) – only 26 characters stored in frequency maps.

Python Implementation

```python
from collections import Counter

def minWindow(s, t):
    if not t or not s:
        return ""

    char_count = Counter(t)
    left, min_length, required_chars, current_chars = 0, float('inf'),
len(char_count), 0
    window_counts = {}
    result = ""

    for right in range(len(s)):
        char = s[right]
        window_counts[char] = window_counts.get(char, 0) + 1

        if char in char_count and window_counts[char] == char_count[char]:
            current_chars += 1

        while left <= right and current_chars == required_chars:
            if right - left + 1 < min_length:
                min_length = right - left + 1
                result = s[left:right+1]

            window_counts[s[left]] -= 1
            if s[left] in char_count and window_counts[s[left]] <
char_count[s[left]]:
                current_chars -= 1
            left += 1

    return result

# Example
s = "ADOBECODEBANC"
t = "ABC"
print(minWindow(s, t))  # Output: "BANC"
```

🚀 Trade-offs:

- Uses extra space for **hash maps**, but avoids recomputation.
- Ensures **O(n) performance**, ideal for long strings.

Key Takeaways

✅ **Fixed vs. variable window sizes depend on problem constraints.**

✅ **Sliding Window reduces time complexity in subarray/substring problems.**

✅ **Character frequency maps help with substring containment problems.**

✅ **Trade-offs include extra space (sets/maps) vs. recomputation costs.**

By mastering this pattern, you'll solve many problems efficiently and recognize **when to apply it in coding interviews!** 🚀

Chapter 3: Two Pointers

Concept & When to Use

The **Two Pointers** technique is an efficient approach used to solve problems involving **pairs or sequences of elements in an array or linked list**. Instead of using nested loops, we maintain **two pointers** that traverse the data structure in different ways to optimize time complexity.

When to Use Two Pointers

✔ The problem involves **pairs** or **triplets** (e.g., "find two numbers that sum to a target").

✔ The input is **sorted** or can be sorted (e.g., "find the closest pair of numbers").

✔ The problem requires **removal, merging, or partitioning** elements in-place (e.g., "remove duplicates from sorted array").

✔ The problem can be solved using a **left-right** or **fast-slow** traversal (e.g., "find the middle of a linked list").

Types of Two Pointers Approaches

🔷 **Left-Right Pointers:** Used for problems involving **sorted arrays** or **bounding conditions** (e.g., "find two numbers that sum to X").

🔷 **Fast-Slow Pointers:** Used for problems involving **linked lists** or **cyclic detection** (e.g., "detect a cycle in a linked list").

Grind 75 Problems

The **Two Pointers** pattern appears in multiple **Grind 75 problems**, such as:

1. **Two Sum II (sorted)** (LeetCode #167)
2. **Three Sum** (LeetCode #15)

3. **Container With Most Water** (LeetCode #11)

Each of these problems benefits from the **Two Pointers** technique. Below, we analyze each problem and discuss brute-force vs. optimized solutions.

Solutions & Trade-offs

1. Two Sum II (Sorted)

💡 **Problem:** Given a sorted array `nums` and a target sum `target`, return the indices of two numbers such that they add up to `target`.

Brute-Force Approach (O(n²))

- Use **two nested loops** to find the pair that sums to `target`.
- **Time Complexity:** O(n²) – checking all pairs is slow for large arrays.
- **Space Complexity:** O(1) – no extra space used.

Optimized Two Pointers Approach (O(n))

- Since the array is **sorted**, we use **left (`l`) and right (`r`) pointers** to find the target sum.
- **If sum is too small**, move `l` right.
- **If sum is too large**, move `r` left.
- **Time Complexity:** O(n) – each element is checked once.
- **Space Complexity:** O(1) – no extra storage needed.

Python Implementation

```python
def twoSum(nums: list[int], target: int) -> list[int]:
    l, r = 0, len(nums) - 1

    while l < r:
        current_sum = nums[l] + nums[r]
        if current_sum == target:
            return [l + 1, r + 1]  # 1-based index
        elif current_sum < target:
            l += 1
        else:
```

```
            r -= 1

    return []
```

🚀 Trade-offs:

- **Sorting helps reduce complexity** to O(n), but it only works if the input is already sorted.
- If the array was **unsorted**, we would need **O(n log n) sorting time** or use a **hash map** (O(n) but requires extra space).

2. Three Sum

💡 **Problem:** Given an array `nums`, return all unique triplets `(a, b, c)` such that `a + b + c = 0`.

Brute-Force Approach (O(n³))

- Try **all triplets** and check if they sum to `0`.
- **Time Complexity:** O(n³) – extremely slow for large inputs.
- **Space Complexity:** O(n) – storing triplets in a result list.

Optimized Sorting + Two Pointers Approach (O(n²))

- **Sort the array** and **fix one element (`nums[i]`)**.
- Use **two pointers (`l` and `r`)** to find the remaining two numbers that sum to `nums[i]`.
- **Avoid duplicates** by skipping repeated values.
- **Time Complexity:** O(n²) – sorting takes O(n log n), and the two-pointer search is O(n).
- **Space Complexity:** O(n) – required for the result set.

Python Implementation

```python
def threeSum(nums: list[int]) -> list[list[int]]:
    nums.sort()
    result = []
```

```python
    for i in range(len(nums) - 2):
        if i > 0 and nums[i] == nums[i - 1]:  # Skip duplicates
            continue

        l, r = i + 1, len(nums) - 1
        while l < r:
            three_sum = nums[i] + nums[l] + nums[r]
            if three_sum == 0:
                result.append([nums[i], nums[l], nums[r]])
                l += 1
                r -= 1
                while l < r and nums[l] == nums[l - 1]:  # Skip duplicates
                    l += 1
            elif three_sum < 0:
                l += 1
            else:
                r -= 1

    return result
```

🚀 Trade-offs:

- **Sorting speeds up** the solution but requires **O(n log n)** time.
- **Avoiding duplicate triplets** ensures correct output.

3. Container With Most Water

💡 **Problem:** Given an array `height`, find the two lines that **hold the most water**.

Brute-Force Approach (O(n²))

- Try **all pairs** and calculate water capacity.
- **Time Complexity:** O(n²) – checking all pairs is inefficient.
- **Space Complexity:** O(1) – no extra storage needed.

Optimized Two Pointers Approach (O(n))

- Start with **left (l) and right (r) pointers** at both ends of the array.
- **Move the pointer pointing to the smaller height** (since increasing width won't help if the height is small).
- **Time Complexity:** O(n) – each element is checked once.

- **Space Complexity:** O(1) – no extra storage needed.

Python Implementation

```python
def maxArea(height: list[int]) -> int:
    l, r = 0, len(height) - 1
    max_water = 0

    while l < r:
        max_water = max(max_water, min(height[l], height[r]) * (r - l))
        if height[l] < height[r]:
            l += 1
        else:
            r -= 1

    return max_water
```

🚀 **Trade-offs:**

- **Optimized approach ensures O(n) performance** by eliminating unnecessary comparisons.
- Moving **only the smaller height pointer** guarantees maximization of the water area.

Key Takeaways

✅ **Two Pointers improve efficiency** in problems involving **pairs, triplets, or partitions**.

✅ **Sorting + Two Pointers** is a common strategy for sum problems.

✅ **Fast-Slow Pointers** are useful for **linked list cycle detection**.

✅ **Trade-offs include sorting time vs. extra space for hash maps.**

Mastering **Two Pointers** will help you solve many **array and linked list problems efficiently!** 🚀

Chapter 4: Fast & Slow Pointers (Cycle Detection)

Concept & When to Use

The **Fast & Slow Pointers** (also known as the **Tortoise and Hare**) technique is a fundamental algorithm used in problems involving **linked lists and cyclic detection**. It efficiently detects cycles and finds entry points in problems related to **linked lists and repeated sequences**.

When to Use Fast & Slow Pointers

✔ The problem involves **linked lists** (e.g., "detect if a linked list has a cycle").

✔ The problem involves **repeated numbers or sequences** (e.g., "find the duplicate in an array").

✔ The problem needs to detect **loops or intersections** (e.g., "find the start of a cycle").

Key Idea

🔷 Use two pointers:

- A **slow pointer (`slow`)** that moves **one step** at a time.
- A **fast pointer (`fast`)** that moves **two steps** at a time.
- If the two pointers meet, there is a **cycle**.

Mathematical Insight

- The fast pointer moves **twice as fast** as the slow pointer.
- If a cycle exists, the fast pointer will eventually **catch up** to the slow pointer.

Grind 75 Problems

The **Fast & Slow Pointers** technique is essential for solving the following **Grind 75** problems:

1. **Linked List Cycle** (LeetCode #141)
2. **Find Duplicate Number** (LeetCode #287)

Below, we explore these problems, along with different solution approaches and trade-offs.

Solutions & Trade-offs

1. Linked List Cycle

💡 **Problem:** Given the head of a linked list, determine if it contains a cycle.

Brute-Force Approach (Using a Hash Set) – O(n) Space

- Store visited nodes in a **hash set**.
- If we encounter a node we've seen before, a cycle exists.
- **Time Complexity:** O(n) – traversing the linked list once.
- **Space Complexity:** O(n) – storing all visited nodes.

Python Implementation (Using Hash Set)

```python
def hasCycle(head: ListNode) -> bool:
    visited = set()
    while head:
        if head in visited:
            return True
        visited.add(head)
        head = head.next
    return False
```

Optimized Approach (Floyd's Cycle Detection) – O(1) Space

- Use **fast and slow pointers**.
- If a cycle exists, they will eventually meet.
- **Time Complexity:** O(n) – each node is visited at most twice.
- **Space Complexity:** O(1) – no extra storage is used.

Python Implementation (Floyd's Cycle Detection)

```
def hasCycle(head: ListNode) -> bool:
    slow, fast = head, head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow == fast:
            return True  # Cycle detected
    return False
```

🚀 Trade-offs:

- **Floyd's Algorithm is optimal (O(1) space)** but requires careful pointer movement.
- **Hash Set method is easier to understand** but requires **O(n) extra space**.

2. Find Duplicate Number

💡 **Problem:** Given an array `nums` with `n + 1` integers where each number is in the range `[1, n]`, find the duplicate number **without modifying the array** and using only **O(1) extra space**.

Brute-Force Approach (Sorting) – O(n log n) Time, O(1) Space

- Sort the array and find consecutive duplicates.
- **Time Complexity:** O(n log n) – due to sorting.
- **Space Complexity:** O(1) – sorting in-place.

Python Implementation (Sorting)

```
def findDuplicate(nums: list[int]) -> int:
    nums.sort()
    for i in range(1, len(nums)):
        if nums[i] == nums[i - 1]:
            return nums[i]
    return -1
```

Better Approach (Using Hash Set) – O(n) Space

- Use a **set** to track visited numbers.

- **Time Complexity:** O(n) – each element is checked once.
- **Space Complexity:** O(n) – storing visited numbers.

Python Implementation (Using Hash Set)

```python
def findDuplicate(nums: list[int]) -> int:
    seen = set()
    for num in nums:
        if num in seen:
            return num
        seen.add(num)
    return -1
```

Optimized Approach (Floyd's Cycle Detection) – O(1) Space

Observation:

- Think of the array as a **linked list** where `nums[i]` points to `nums[nums[i]]`.
- The duplicate number forms a **cycle** because it appears more than once.

Algorithm Steps:

1. Use **fast and slow pointers** to detect a cycle.
2. Move `slow` one step and `fast` two steps.
3. If they meet, reset `slow` to `0` and move both pointers **one step at a time** to find the **start of the cycle (duplicate number)**.

Python Implementation (Floyd's Cycle Detection)

```python
def findDuplicate(nums: list[int]) -> int:
    slow, fast = nums[0], nums[0]

    # Phase 1: Detect the cycle
    while True:
        slow = nums[slow]
        fast = nums[nums[fast]]
        if slow == fast:
            break

    # Phase 2: Find cycle start (duplicate)
```

```
    slow = nums[0]
    while slow != fast:
        slow = nums[slow]
        fast = nums[fast]

    return slow
```

🚀 Trade-offs:

- **Floyd's Cycle Detection is O(n) time, O(1) space** (optimal).
- **Hash Set method is simpler but uses O(n) extra space**.

Key Takeaways

✅ **Fast & Slow Pointers detect cycles efficiently without extra space**.

✅ **Floyd's Algorithm works for both linked lists and repeated sequences**.

✅ **This technique is critical for problems involving cycles or repeated numbers**.

Mastering **Fast & Slow Pointers** will help you solve many **cycle detection problems efficiently!** 🚀

Chapter 5: Merge Intervals

Concept & When to Use

The **Merge Intervals** pattern is useful when dealing with **overlapping intervals** in problems related to scheduling, time ranges, or segment merging. The key idea is to **sort intervals** and then **merge or process them sequentially**.

When to Use Merge Intervals

✔ The problem involves **intervals or ranges** (e.g., `[start, end]`).

✔ You need to **merge overlapping intervals** into a single range.

✔ The problem involves **sorting and comparing** intervals based on their start and end points.

✔ You need to **count active intervals** at any given time (e.g., **finding the maximum number of concurrent meetings**).

Key Idea

◆ **Sort intervals by start time** to process them in a logical order.

◆ **Use a greedy approach** to merge intervals **as we iterate**.

◆ **Heap (Priority Queue) can optimize counting active intervals** (used in scheduling problems).

Grind 75 Problems

The **Merge Intervals** technique is essential for solving the following **Grind 75** problems:

1. **Merge Intervals** (LeetCode #56)
2. **Meeting Rooms II** (LeetCode #253)

Below, we explore these problems, different solution approaches, and trade-offs.

Solutions & Trade-offs

1. Merge Intervals

💡 **Problem:** Given an array of intervals `intervals`, merge all overlapping intervals and return an array of non-overlapping intervals.

Brute-Force Approach (Checking All Pairs) – O(n²) Time

- Compare each interval with every other interval to check for overlaps.
- **Time Complexity:** O(n²) – due to nested comparisons.
- **Space Complexity:** O(n) – for storing merged intervals.

Python Implementation (Brute Force, Inefficient)

```python
def merge(intervals: list[list[int]]) -> list[list[int]]:
    merged = []
    for i in range(len(intervals)):
        for j in range(i + 1, len(intervals)):
            if intervals[i][1] >= intervals[j][0]:  # Overlapping condition
                intervals[j] = [min(intervals[i][0], intervals[j][0]),
max(intervals[i][1], intervals[j][1])]
    return merged
```

❌ **Not efficient for large inputs**.

Optimized Approach (Sorting & Merging) – O(n log n) Time, O(n) Space

Steps:

1. **Sort intervals** based on start time.
2. **Iterate through intervals** and merge overlapping ones.
3. **Keep track of the last merged interval** and modify it if necessary.

Time Complexity: O(n log n) – due to sorting.

Space Complexity: O(n) – for storing the result.

Python Implementation (Sorting & Merging)

```python
def merge(intervals: list[list[int]]) -> list[list[int]]:
    intervals.sort()  # Sort by start time
    merged = []

    for interval in intervals:
        if not merged or merged[-1][1] < interval[0]:
            merged.append(interval)  # No overlap, add directly
        else:
            merged[-1][1] = max(merged[-1][1], interval[1])  # Merge overlapping
intervals

    return merged
```

🚀 Trade-offs:

- Sorting is **O(n log n)**, but merging is **O(n)**, making this approach **efficient**.
- **Modifies input** in-place, which can be useful but requires caution.

2. Meeting Rooms II

💡 **Problem:** Given an array of meeting time intervals, return the **minimum number of conference rooms** required.

Brute-Force Approach (Checking All Overlaps) – O(n²) Time

- Compare each meeting with every other meeting to count overlaps.
- **Time Complexity:** O(n²) – due to nested loops.
- **Space Complexity:** O(n) – storing overlaps.

Python Implementation (Brute Force, Inefficient)

```python
def minMeetingRooms(intervals: list[list[int]]) -> int:
    max_rooms = 0
    for i in range(len(intervals)):
        count = 1
        for j in range(len(intervals)):
            if i != j and intervals[j][0] < intervals[i][1]:
                count += 1
```

```
        max_rooms = max(max_rooms, count)
    return max_rooms
```

❌ Too slow for large inputs.

Optimized Approach (Sorting + Min Heap) – O(n log n) Time, O(n) Space

Steps:

1. **Sort meetings by start time**.
2. Use a **min-heap** to keep track of meeting end times.
3. **If a room is free (earliest end time is ≤ current start time), reuse it**.
   Otherwise, allocate a new room.

Time Complexity: O(n log n) – due to sorting and heap operations.

Space Complexity: O(n) – storing end times in a heap.

Python Implementation (Min Heap)

```python
import heapq

def minMeetingRooms(intervals: list[list[int]]) -> int:
    if not intervals:
        return 0

    intervals.sort()  # Sort by start time
    min_heap = []  # Stores end times of meetings

    for interval in intervals:
        if min_heap and min_heap[0] <= interval[0]:
            heapq.heappop(min_heap)  # Free up a room
        heapq.heappush(min_heap, interval[1])  # Allocate new room

    return len(min_heap)  # Number of rooms used
```

🚀 Trade-offs:

- Sorting is **O(n log n)**, but heap operations are **O(log n)** per interval, making this approach **efficient**.

- **Heap keeps track of active meetings** in the smallest amount of space possible.

Key Takeaways

✅ **Sorting is often necessary** when dealing with **overlapping intervals**.

✅ **Greedy merging** works well for merging intervals but **does not work for counting overlapping events**.

✅ **Min Heaps are useful** when counting **active overlapping intervals** (e.g., meeting room allocation).

Mastering **Merge Intervals** will help you solve **scheduling and range-based problems efficiently!** 🚀

Chapter 6: Cyclic Sort

Concept & When to Use

The **Cyclic Sort** pattern is useful when dealing with problems that involve a **range of numbers from 1 to N**, where the goal is to **rearrange the numbers into their correct positions** with minimal extra space. This technique is especially helpful in problems involving **finding duplicates, missing numbers, or misplaced elements** in an array.

When to Use Cyclic Sort

✔ The input consists of numbers **in a fixed range** (e.g., 1 to N).

✔ The problem requires finding **missing, duplicate, or misplaced numbers**.

✔ The array should be **sorted with constant extra space**.

✔ The values **can be used as indices** for in-place swapping.

Key Idea

◆ **Iterate through the array** and swap each number to its correct index (`nums[i]` should be at `nums[nums[i] - 1]`).

◆ **Continue swapping** until every number is in its correct position or a cycle is detected.

◆ **After sorting, iterate again** to identify missing or duplicate numbers.

Grind 75 Problems

The **Cyclic Sort** technique is essential for solving the following **Grind 75** problems:

   1. **Find All Duplicates in an Array** (LeetCode #442)

2. **First Missing Positive** (LeetCode #41)

Below, we explore these problems, different solution approaches, and trade-offs.

Solutions & Trade-offs

1. Find All Duplicates in an Array

💡 **Problem:** Given an integer array `nums` where $1 \leq nums[i] \leq n$ (where `n` is the array's length), return **all the numbers that appear twice**.

Brute-Force Approach (Sorting or Hash Set) – O(n log n) or O(n) Space

- Sort the array and check adjacent elements for duplicates (**O(n log n)**).
- Use a **hash set** to track seen elements (**O(n) space**).

Python Implementation (Hash Set)

```python
def findDuplicates(nums: list[int]) -> list[int]:
    seen = set()
    duplicates = []
    for num in nums:
        if num in seen:
            duplicates.append(num)
        else:
            seen.add(num)
    return duplicates
```

✅ **Correct, but uses extra space (O(n)).**

❌ **Does not modify the array in-place.**

Optimized Approach (Cyclic Sort) – O(n) Time, O(1) Space

Steps:

1. **Use Cyclic Sort** to place each number in its correct position.
2. **After sorting, iterate through the array** to find misplaced numbers.

Time Complexity: O(n) – single pass sorting.

Space Complexity: O(1) – modifies input in-place.

Python Implementation (Cyclic Sort)

```python
def findDuplicates(nums: list[int]) -> list[int]:
    result = []

    for i in range(len(nums)):
        while nums[i] != nums[nums[i] - 1]:  # Place the number at the correct
index
            nums[nums[i] - 1], nums[i] = nums[i], nums[nums[i] - 1]

    for i in range(len(nums)):
        if nums[i] != i + 1:
            result.append(nums[i])  # Misplaced number is a duplicate

    return result
```

🚀 **Trade-offs:**

- **Faster than sorting (O(n) vs. O(n log n)).**
- **Uses no extra space.**
- **Modifies the input array in-place.**

2. First Missing Positive

💡 **Problem:** Given an unsorted integer array `nums`, return the **smallest missing positive integer**.

Brute-Force Approach (Sorting or Hash Set) – O(n log n) or O(n) Space

- **Sorting:** Sort and iterate to find the first missing positive number (**O(n log n)**).
- **Hash Set:** Store numbers and check for missing ones (**O(n) space**).

Python Implementation (Sorting)

```python
def firstMissingPositive(nums: list[int]) -> int:
    nums.sort()
    smallest = 1
    for num in nums:
```

```
        if num == smallest:
            smallest += 1
    return smallest
```

✅ **Correct, but slow (O(n log n)).**

❌ **Extra space if using a set.**

Optimized Approach (Cyclic Sort) – O(n) Time, O(1) Space

Steps:

1. **Use Cyclic Sort** to place each positive number at its correct index (`nums[i]` `= i + 1`).
2. **Iterate again** to find the first missing number.

Time Complexity: O(n) – single pass sorting.

Space Complexity: O(1) – modifies input in-place.

Python Implementation (Cyclic Sort)

```python
def firstMissingPositive(nums: list[int]) -> int:
    n = len(nums)

    for i in range(n):
        while 1 <= nums[i] <= n and nums[i] != nums[nums[i] - 1]:  # Place numbers
in correct index
            nums[nums[i] - 1], nums[i] = nums[i], nums[nums[i] - 1]

    for i in range(n):
        if nums[i] != i + 1:
            return i + 1  # First missing positive number

    return n + 1  # If all are in place, return next positive number
```

🚀 **Trade-offs:**

- **O(n) time complexity is optimal.**
- **Uses no extra space.**
- **Modifies input array in-place.**

Key Takeaways

✅ **Cyclic Sort works best for problems involving numbers within a specific range.**

✅ **Sorting in O(n) time with constant space is achievable when modifying the array in-place.**

✅ **This pattern is powerful for missing/duplicate number problems.**

Mastering **Cyclic Sort** will help you solve **sorting and missing number problems efficiently!** 🚀

Chapter 7: Heap and Priority Queue

Concept & When to Use

The **Heap and Priority Queue** pattern is powerful for solving problems that require **efficient retrieval of the largest/smallest elements, scheduling tasks, and maintaining dynamic order**.

This technique uses:

- A **Min Heap** (Priority Queue) to efficiently retrieve the smallest elements.
- A **Max Heap** (simulated using a Min Heap with negated values) to retrieve the largest elements.
- **Balancing heaps** enables efficient median retrieval and other optimizations.

When to Use Heaps & Priority Queues

✔ **Finding the Kth largest/smallest element** (e.g., "Kth Largest Element in an Array").

✔ **Finding the median of a dynamic data stream** (e.g., "Find Median in a Stream").

✔ **Handling priority-based tasks** (e.g., "Task Scheduler").

✔ **Merging multiple sorted streams efficiently** (e.g., "Merge K Sorted Lists").

✔ **Processing dynamic elements where order matters** (e.g., "Meeting Rooms II").

Key Idea

🔷 **Min Heap** (`min-heap`) efficiently retrieves the smallest elements.

🔷 **Max Heap** (`max-heap`) efficiently retrieves the largest elements.

🔷 **Insertion & removal take O(log n)** due to heap operations.

🔷 **Top element retrieval takes O(1)**, making heaps ideal for priority-based problems.

Grind 75 Problems

The **Heap and Priority Queue** pattern is essential for solving these **Grind 75** problems:

1. **Kth Largest Element in an Array** (LeetCode #215)
2. **Find Median in a Stream** (LeetCode #295)
3. **Task Scheduler** (LeetCode #621)

Below, we explore these problems, different solution approaches, and trade-offs.

Solutions & Trade-offs

1. Kth Largest Element in an Array

💡 **Problem:** Given an unsorted array, find the **Kth largest element**.

Brute-Force Approach (Sorting) – O(n log n) Time, O(1) Space

- **Sort the array** and return `nums[-k]` (the `Kth` largest element).

Python Implementation (Sorting)

```python
def findKthLargest(nums: list[int], k: int) -> int:
    nums.sort()
    return nums[-k]
```

✅ **Simple, but inefficient** for large datasets due to sorting (**O(n log n)**).

Optimized Approach (Min Heap) – O(n log k) Time, O(k) Space

- **Use a Min Heap of size `k`** to track the top `k` largest elements.
- **After iterating, the heap's root is the Kth largest element**.

Python Implementation (Min Heap)

```python
import heapq

def findKthLargest(nums: list[int], k: int) -> int:
    min_heap = []
    for num in nums:
        heapq.heappush(min_heap, num)
        if len(min_heap) > k:
            heapq.heappop(min_heap)  # Remove smallest element

    return min_heap[0]
```

🚀 Trade-offs:

- **O(n log k) time complexity**, faster than sorting for large n.
- **O(k) space complexity**, since we store only k elements.

2. Find Median in a Stream

💡 **Problem:** Design a data structure that supports:

- addNum(int num): Inserts a number into the data stream.
- findMedian(): Returns the median of all elements so far.

Brute-Force Approach (Sorting) – O(n log n) Time, O(n) Space

- **Store all numbers** in a list and **sort on every insertion**.

Python Implementation (Sorting)

```python
class MedianFinder:
    def __init__(self):
        self.data = []

    def addNum(self, num: int) -> None:
        self.data.append(num)
        self.data.sort()

    def findMedian(self) -> float:
        n = len(self.data)
        if n % 2 == 1:
```

```
            return self.data[n // 2]
        else:
            return (self.data[n // 2 - 1] + self.data[n // 2]) / 2
```

✅ **Simple, but inefficient** due to sorting on every insert (**O(n log n)**).

Optimized Approach (Two Heaps) – O(log n) Insert, O(1) Find Median

- **Use a Max Heap** for the lower half of numbers.
- **Use a Min Heap** for the upper half.
- **Balance the two heaps** to ensure correct median retrieval.

Python Implementation (Two Heaps)

```python
import heapq

class MedianFinder:
    def __init__(self):
        self.small = []  # Max Heap (store negative values)
        self.large = []  # Min Heap

    def addNum(self, num: int) -> None:
        heapq.heappush(self.small, -num)
        heapq.heappush(self.large, -heapq.heappop(self.small))

        if len(self.small) < len(self.large):
            heapq.heappush(self.small, -heapq.heappop(self.large))

    def findMedian(self) -> float:
        if len(self.small) > len(self.large):
            return -self.small[0]
        return (-self.small[0] + self.large[0]) / 2
```

🚀 **Trade-offs:**

- **O(log n) insertion** vs. **O(n log n) sorting**.
- **O(1) median retrieval** vs. **O(n) median retrieval in sorting**.
- **Extra space for two heaps (O(n))**, but avoids frequent sorting.

3. Task Scheduler

💡 **Problem:** Given an array of tasks and a cooling interval n, find the **minimum time required** to execute all tasks, ensuring that the same task is scheduled only after n intervals.

Brute-Force Approach (Sorting & Simulation) – O(n log n) Time, O(n) Space

- **Sort tasks by frequency** and process in order.
- **Insert idle slots manually** to maintain the cooling period.

Python Implementation (Sorting)

```python
from collections import Counter

def leastInterval(tasks: list[str], n: int) -> int:
    freq = list(Counter(tasks).values())
    freq.sort(reverse=True)
    max_freq = freq[0]
    idle_time = (max_freq - 1) * n

    for f in freq[1:]:
        idle_time -= min(max_freq - 1, f)

    idle_time = max(0, idle_time)
    return len(tasks) + idle_time
```

✅ **Works but inefficient** due to sorting (**O(n log n)**).

Optimized Approach (Max Heap) – O(n log k) Time, O(n) Space

- **Use a Max Heap** to always process the most frequent tasks first.
- **Use a queue to track cooldown periods** for tasks.

Python Implementation (Heap)

```python
import heapq
from collections import Counter, deque

def leastInterval(tasks: list[str], n: int) -> int:
    freq_map = Counter(tasks)
    max_heap = [-f for f in freq_map.values()]
    heapq.heapify(max_heap)
```

```python
        queue = deque()
        time = 0

        while max_heap or queue:
            time += 1
            if max_heap:
                count = 1 + heapq.heappop(max_heap)
                if count:
                    queue.append((count, time + n))

            if queue and queue[0][1] == time:
                heapq.heappush(max_heap, queue.popleft()[0])

        return time
```

🚀 **Trade-offs:**

- **O(n log k) is better than O(n log n) sorting**.
- **Max Heap ensures efficient task execution**.
- **Uses extra space for heap & queue**.

Key Takeaways

✅ **Heaps and Priority Queues provide efficient ways to solve order-based problems.**

✅ **Heap operations (O(log n)) are often better than sorting (O(n log n)).**

✅ **Min Heaps retrieve smallest elements efficiently, Max Heaps retrieve largest elements efficiently.**

✅ **Use heaps for priority scheduling, median retrieval, and Kth largest/smallest problems.**

By mastering **Heap and Priority Queue**, you'll solve scheduling, median-finding, and priority problems efficiently! 🚀

Chapter 8: Tree Traversal (BFS & DFS)

Concept & When to Use

Tree traversal is a fundamental technique in **binary trees and graphs**, used to explore nodes in a structured manner. There are two primary traversal methods:

1. **Breadth-First Search (BFS)** – Explores all nodes at the same depth before moving deeper.
2. **Depth-First Search (DFS)** – Explores as deep as possible before backtracking.

These approaches help solve problems related to **tree structure, hierarchy, and relationships** efficiently.

When to Use BFS vs. DFS

Criteria	BFS (Level Order Traversal)	DFS (Preorder, Inorder, Postorder)
When to use?	Finding shortest path, level-wise traversal	Finding ancestors, validating BST, path-finding
Space Complexity	O(N) (queue holds all nodes at a level)	O(H) (stack holds recursion depth, H=height)
Best for	Problems needing level-wise relationships	Problems requiring full tree exploration
Iterative Implementation?	Uses a queue (FIFO)	Uses a stack (LIFO) or recursion

Grind 75 Problems

The **Tree Traversal** pattern is crucial for solving the following **Grind 75** problems:

1. **Binary Tree Level Order Traversal** (BFS)
2. **Lowest Common Ancestor** (DFS)

3. **Validate Binary Search Tree** (DFS)

We explore different solutions and trade-offs for these problems.

Solutions & Trade-offs

1. Binary Tree Level Order Traversal (BFS)

💡 **Problem:** Given a binary tree, return its **level-order traversal** (left to right, level by level).

Approach: BFS (Queue) – O(N) Time, O(N) Space

- Use a **queue (FIFO)** to process nodes level by level.
- Maintain a list of nodes for each level.

Python Implementation (BFS)

```python
from collections import deque

class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def levelOrder(root: TreeNode) -> list[list[int]]:
    if not root:
        return []

    result, queue = [], deque([root])

    while queue:
        level = []
        for _ in range(len(queue)):  # Process all nodes at the current level
            node = queue.popleft()
            level.append(node.val)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)

        result.append(level)
```

```
        return result
```

✅ **Trade-offs:**

- **O(N) time complexity** (each node is visited once).
- **O(N) space complexity** (stores all nodes at the deepest level).
- **BFS ensures level-wise traversal, but uses more memory than DFS**.

2. Lowest Common Ancestor (DFS)

💡 **Problem:** Given a binary tree and two nodes p and q, find their **Lowest Common Ancestor (LCA)**.

Approach: DFS (Recursive) – O(N) Time, O(H) Space

- Traverse the tree using **DFS** until p or q is found.
- If a node is an ancestor of both p and q, return it as the LCA.

Python Implementation (DFS)

```python
def lowestCommonAncestor(root: TreeNode, p: TreeNode, q: TreeNode) -> TreeNode:
    if not root or root == p or root == q:
        return root  # Found p or q, return current node

    left = lowestCommonAncestor(root.left, p, q)
    right = lowestCommonAncestor(root.right, p, q)

    if left and right:
        return root  # This is the LCA

    return left if left else right  # Return non-null subtree
```

✅ **Trade-offs:**

- **O(N) time complexity** (DFS visits each node once).
- **O(H) space complexity** (recursive stack depth is the tree height).
- **Recursive DFS is elegant, but may cause stack overflow in deep trees**.

3. Validate Binary Search Tree (DFS)

💡 **Problem:** Given a binary tree, determine if it is a **valid Binary Search Tree (BST)**.

Approach: DFS (Inorder Traversal) – O(N) Time, O(H) Space

- Perform an **inorder traversal** (left → root → right).
- Ensure values are strictly increasing.

Python Implementation (DFS)

```python
def isValidBST(root: TreeNode) -> bool:
    def inorder(node, lower=float('-inf'), upper=float('inf')):
        if not node:
            return True

        if node.val <= lower or node.val >= upper:
            return False  # Violates BST property

        return inorder(node.left, lower, node.val) and inorder(node.right, node.val, upper)

    return inorder(root)
```

✅ **Trade-offs:**

- **O(N) time complexity** (visits each node once).
- **O(H) space complexity** (recursive depth depends on tree height).
- **DFS is memory-efficient for balanced trees but can cause stack overflows in skewed trees**.

BFS vs. DFS: Which One to Use?

Scenario	Use BFS (Queue)	Use DFS (Stack/Recursion)
Level-wise traversal required?	✅ Yes	❌ No
Searching for shortest path?	✅ Yes (e.g., unweighted graphs)	❌ No

Tree structure validation (BST)?	❌ No	✅ Yes (inorder traversal)
Tree depth-related problems?	❌ No	✅ Yes (finding ancestors, recursion)
Memory constraints?	❌ More memory	✅ Less memory in balanced trees

Key Takeaways

✅ **BFS (Queue) is best for level-wise traversal and shortest paths.**

✅ **DFS (Recursion/Stack) is efficient for ancestor, validation, and search problems.**

✅ **Iterative solutions avoid recursion depth limits but may be harder to implement.**

✅ **DFS (Inorder) is ideal for checking BST validity.**

Mastering **BFS & DFS** will help you efficiently traverse trees and solve **search, validation, and relationship-based problems!** 🚀

Chapter 9: Graph Algorithms (BFS, DFS, Union-Find, Dijkstra)

Concept & When to Use

Graph algorithms are vital for solving problems related to **network traversal, pathfinding, and connectivity**. A graph is a collection of nodes (vertices) connected by edges, and it can represent various real-world structures such as networks, maps, and social relationships. The four most common graph algorithms are:

1. **Breadth-First Search (BFS)** – Explores all neighbors at the current level before moving on to the next level. Ideal for finding the shortest path in unweighted graphs.
2. **Depth-First Search (DFS)** – Explores as far down a branch as possible before backtracking. Useful for solving problems related to connectivity and pathfinding.
3. **Union-Find (Disjoint Set Union, DSU)** – A data structure for efficiently tracking and merging disjoint sets. Essential for solving problems involving connectivity (e.g., determining if two nodes are in the same connected component).
4. **Dijkstra's Algorithm** – Finds the shortest path between nodes in a weighted graph. It is typically used for pathfinding in graphs with non-negative edge weights.

When to Use Each Algorithm

- **BFS:** When you need to explore nodes level-by-level or find the shortest path in unweighted graphs.
- **DFS:** When you need to explore all nodes in a branch first, useful for topological sorting, connected components, and backtracking problems.
- **Union-Find:** When you need to manage and merge sets of connected nodes efficiently (e.g., determining if two nodes are connected in an undirected

graph).

- **Dijkstra's:** When you need to find the shortest path between nodes in weighted graphs.

Grind 75 Problems

The following **Grind 75** problems make use of various graph algorithms:

1. **Clone Graph (DFS/BFS)**
2. **Course Schedule (Topological Sort)**
3. **Number of Islands (DFS/BFS)**

Solutions & Trade-offs

1. Clone Graph (DFS/BFS)

💡 **Problem:** Given a reference to a graph node, **clone the graph**. Each node in the graph contains a value and a list of neighbors.

Approach: BFS/DFS – O(V + E) Time, O(V) Space

- For **DFS**, use recursion or a stack to explore each node and its neighbors, cloning each node as you visit it.
- For **BFS**, use a queue and iterate level-by-level, cloning nodes and adding them to a new graph.

Python Implementation (DFS)

```python
class Node:
    def __init__(self, val=0, neighbors=None):
        self.val = val
        self.neighbors = neighbors if neighbors is not None else []

def cloneGraph(node: 'Node') -> 'Node':
    if not node:
        return None

    visited = {}
```

```python
    def dfs(node):
        if node in visited:
            return visited[node]

        # Create a new node and store it in visited
        clone = Node(node.val)
        visited[node] = clone

        # Recursively clone neighbors
        for neighbor in node.neighbors:
            clone.neighbors.append(dfs(neighbor))

        return clone

    return dfs(node)
```

Approach: BFS

```python
from collections import deque

def cloneGraph(node: 'Node') -> 'Node':
    if not node:
        return None

    visited = {node: Node(node.val)}
    queue = deque([node])

    while queue:
        curr = queue.popleft()

        for neighbor in curr.neighbors:
            if neighbor not in visited:
                visited[neighbor] = Node(neighbor.val)
                queue.append(neighbor)
            visited[curr].neighbors.append(visited[neighbor])

    return visited[node]
```

✅ **Trade-offs:**

- **O(V + E) time complexity**: Each node and edge is visited once.
- **O(V) space complexity**: Store all visited nodes.
- **DFS is easy to implement recursively but can lead to stack overflow for deep graphs. BFS is more memory-intensive but avoids recursion depth**

issues.

2. Course Schedule (Topological Sort)

💡 **Problem:** Given a set of courses and prerequisites, determine if it's possible to finish all the courses. This is a **topological sorting** problem on a directed graph.

Approach: Topological Sort – O(V + E) Time, O(V) Space

- Use **DFS** to detect cycles and perform a topological sort. If you can complete the sorting, the courses can be finished.
- Alternatively, use **Kahn's algorithm** (BFS) to process nodes with no incoming edges, which allows you to build the topological order.

Python Implementation (DFS)

```python
from collections import defaultdict

def canFinish(numCourses: int, prerequisites: list[list[int]]) -> bool:
    graph = defaultdict(list)
    for dest, src in prerequisites:
        graph[src].append(dest)

    visited = [0] * numCourses  # 0 = unvisited, 1 = visiting, 2 = visited

    def dfs(course):
        if visited[course] == 1:  # Cycle detected
            return False
        if visited[course] == 2:
            return True

        visited[course] = 1
        for neighbor in graph[course]:
            if not dfs(neighbor):
                return False

        visited[course] = 2
        return True

    for course in range(numCourses):
        if visited[course] == 0:
            if not dfs(course):
                return False
```

```
        return True
```

Approach: BFS (Kahn's Algorithm)

```python
from collections import import deque, defaultdict

def canFinish(numCourses: int, prerequisites: list[list[int]]) -> bool:
    graph = defaultdict(list)
    in_degree = [0] * numCourses

    # Build graph and calculate in-degrees
    for dest, src in prerequisites:
        graph[src].append(dest)
        in_degree[dest] += 1

    # Start with courses that have no prerequisites
    queue = deque([i for i in range(numCourses) if in_degree[i] == 0])

    visited_courses = 0

    while queue:
        course = queue.popleft()
        visited_courses += 1

        for neighbor in graph[course]:
            in_degree[neighbor] -= 1
            if in_degree[neighbor] == 0:
                queue.append(neighbor)

    return visited_courses == numCourses
```

✅ **Trade-offs:**

- **O(V + E) time complexity** (topological sort).
- **O(V) space complexity** (graph and in-degree storage).
- **DFS is elegant but may be tricky to implement for large graphs. BFS is iterative and avoids recursion depth issues.**

3. Number of Islands (DFS/BFS)

💡 **Problem:** Given a 2D grid representing a map of `'1'`s (land) and `'0'`s (water), find the number of islands. An island is surrounded by water and is formed by

connecting adjacent lands horizontally or vertically.

Approach: DFS/BFS – O(M _ N) Time, O(M _ N) Space

- Use **DFS** or **BFS** to mark all land cells connected to a given land cell as visited (flood fill).
- Count the number of connected components (islands).

Python Implementation (DFS)

```python
def numIslands(grid: list[list[str]]) -> int:
    if not grid:
        return 0

    def dfs(i, j):
        if i < 0 or i >= len(grid) or j < 0 or j >= len(grid[0]) or grid[i][j] ==
'0':
            return
        grid[i][j] = '0'  # Mark as visited
        dfs(i+1, j)
        dfs(i-1, j)
        dfs(i, j+1)
        dfs(i, j-1)

    count = 0
    for i in range(len(grid)):
        for j in range(len(grid[0])):
            if grid[i][j] == '1':  # Found an unvisited land cell
                count += 1
                dfs(i, j)  # Mark all connected land as visited

    return count
```

Approach: BFS

```python
from collections import deque

def numIslands(grid: list[list[str]]) -> int:
    if not grid:
        return 0

    def bfs(i, j):
        queue = deque([(i, j)])
        grid[i][j] = '0'  # Mark as visited
```

```python
        while queue:
            x, y = queue.popleft()
            for dx, dy in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
                nx, ny = x + dx, y + dy
                if 0 <= nx < len(grid) and 0 <= ny < len(grid[0]) and grid[nx][ny]
== '1':
                    grid[nx][ny] = '0'  # Mark as visited
                    queue.append((nx, ny))

    count = 0
    for i in range(len(grid)):
        for j in range(len(grid[0])):
            if grid[i][j] == '1':  # Found an unvisited land cell
                count += 1
                bfs(i, j)  # Mark all connected land as visited

    return count
```

✅ Trade-offs:

- **O(M * N) time complexity** (each cell is visited once).
- **O(M * N) space complexity** (for the recursion stack or queue).
- **DFS may cause stack overflow on large grids, while BFS avoids recursion but uses more memory for large grids.**

BFS vs. DFS vs. Union-Find

- **BFS** is best for problems involving level-order traversal or shortest path in unweighted graphs.
- **DFS** is ideal for problems where you need to explore every branch (e.g., pathfinding, connectivity).
- **Union-Find** is optimal when you need to efficiently track connected components or merge sets.

Chapter 10: Dynamic Programming (Top-down & Bottom-up)

Concept & When to Use

Dynamic Programming (DP) is a powerful technique for solving optimization problems by breaking them down into smaller subproblems. It is useful when the problem exhibits the following two properties:

1. **Optimal Substructure:** The optimal solution to a problem can be constructed from the optimal solutions of its subproblems.
2. **Overlapping Subproblems:** The problem can be divided into subproblems that are solved multiple times. Instead of recalculating the solutions to the subproblems each time, DP saves the results and reuses them.

There are two primary approaches in DP:

- **Top-down approach (Memoization):** This approach starts from the original problem and solves it by recursively breaking it down into smaller subproblems. The results of these subproblems are stored to avoid redundant calculations.
- **Bottom-up approach (Tabulation):** This approach solves all the subproblems first and builds the solution to the original problem incrementally.

When to Use DP:

- When a problem involves making decisions over time or in stages.
- When a problem can be broken down into overlapping subproblems.
- When an optimal solution to a problem can be constructed from solutions to subproblems.

Grind 75 Problems

Here are the **Grind 75** problems that make use of dynamic programming techniques:

1. **Coin Change**
2. **Longest Increasing Subsequence**
3. **Edit Distance**

Solutions & Trade-offs

1. Coin Change

💡 **Problem:** Given an integer array `coins` representing coins of different denominations and an integer `amount`, return the fewest number of coins needed to make up that amount. If that amount of money cannot be made up by any combination of the coins, return `-1`.

Approach: Bottom-up DP (Tabulation)

This problem can be solved by defining a DP array `dp[i]` that represents the minimum number of coins needed to make up amount `i`. The idea is to fill this DP array by considering each coin and checking if using that coin results in a smaller number of coins than previously known.

- **Time Complexity:** O(n * amount) where n is the number of coin denominations.
- **Space Complexity:** O(amount) due to the DP array.

Python Implementation (Bottom-up Tabulation)

```python
def coinChange(coins, amount):
    dp = [float('inf')] * (amount + 1)
    dp[0] = 0  # 0 coins needed to make amount 0

    for i in range(1, amount + 1):
        for coin in coins:
            if i - coin >= 0:
                dp[i] = min(dp[i], dp[i - coin] + 1)
```

```
    return dp[amount] if dp[amount] != float('inf') else -1
```

Trade-offs:

- **Top-down (Memoization)**: Involves recursion with memoization, which can be intuitive but has overhead due to recursive calls. It can be more difficult to implement for large input sizes compared to the bottom-up approach.
- **Bottom-up (Tabulation)**: It avoids recursion, which can lead to stack overflow in the top-down approach. It is more efficient in terms of both time and space, especially for larger input sizes, and is generally the preferred approach.

2. Longest Increasing Subsequence

💡 **Problem:** Given an integer array `nums`, return the length of the longest strictly increasing subsequence.

Approach: Bottom-up DP (Tabulation)

This problem involves building a DP table where each entry `dp[i]` represents the length of the longest increasing subsequence ending at index `i`. For each element, we check all previous elements to see if they can form an increasing subsequence.

- **Time Complexity:** O(n^2), where n is the length of the array.
- **Space Complexity:** O(n) due to the DP array.

Python Implementation (Bottom-up Tabulation)

```python
def lengthOfLIS(nums):
    if not nums:
        return 0

    dp = [1] * len(nums)  # Initialize DP array with 1

    for i in range(1, len(nums)):
        for j in range(i):
            if nums[i] > nums[j]:
                dp[i] = max(dp[i], dp[j] + 1)
```

```
    return max(dp)
```

Trade-offs:

- **Top-down (Memoization)**: You can solve this problem using recursion with memoization, but the time complexity will still be O(n^2), and managing the recursive state and bounds could be cumbersome.
- **Bottom-up (Tabulation)**: The bottom-up approach is more intuitive and avoids recursion. It ensures better memory usage because it doesn't store the recursive call stack. For large inputs, the bottom-up approach is usually more practical.

3. Edit Distance

💡 **Problem:** Given two strings `word1` and `word2`, return the minimum number of operations required to convert `word1` to `word2`. You have three possible operations: insert a character, delete a character, or replace a character.

Approach: Bottom-up DP (Tabulation)

The DP array `dp[i][j]` represents the minimum number of operations required to convert the first `i` characters of `word1` to the first `j` characters of `word2`. The transitions depend on whether the characters match or not.

- **Time Complexity:** O(m * n), where m and n are the lengths of `word1` and `word2`.
- **Space Complexity:** O(m * n) due to the DP table.

Python Implementation (Bottom-up Tabulation)

```python
def minDistance(word1, word2):
    m, n = len(word1), len(word2)

    # Create a DP table
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    # Initialize base cases
```

```python
    for i in range(m + 1):
        dp[i][0] = i  # Deleting all characters from word1

    for j in range(n + 1):
        dp[0][j] = j  # Inserting all characters into word1

    # Fill the DP table
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if word1[i - 1] == word2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1]  # No operation needed
            else:
                dp[i][j] = min(dp[i - 1][j], dp[i][j - 1], dp[i - 1][j - 1]) + 1  #
Min operation

    return dp[m][n]
```

Trade-offs:

- **Top-down (Memoization):** This can be more intuitive, especially for recursion lovers. However, it uses extra space for recursion and might have performance drawbacks for larger strings due to function call overhead.
- **Bottom-up (Tabulation):** The bottom-up approach is more efficient in terms of both time and space and avoids the pitfalls of recursion. It's generally the preferred choice for problems like this, especially for large inputs.

Recursion vs. Memoization vs. Tabulation

- **Recursion** is the most intuitive approach for DP problems but can lead to inefficient solutions due to redundant calculations. It also risks exceeding the recursion limit on larger inputs.
- **Memoization** (top-down DP) saves the results of subproblems to avoid recomputing them. It combines the clarity of recursion with the efficiency of storing results but still suffers from the overhead of recursive calls.
- **Tabulation** (bottom-up DP) builds the solution iteratively and is generally more efficient because it avoids recursion depth issues and function call overhead. It's the preferred approach for most DP problems due to its simplicity and efficiency.

Summary

- **Dynamic Programming** is a powerful technique for solving problems with overlapping subproblems and optimal substructure.
- **Top-down (Memoization)** is recursive and intuitive, but can be less efficient due to overhead.
- **Bottom-up (Tabulation)** builds the solution iteratively and is generally more space and time-efficient, especially for larger inputs.

Chapter 11: Backtracking

Concept & When to Use

Backtracking is a general algorithmic technique used for finding all (or some) solutions to computational problems, particularly for problems that involve searching through all possible combinations. The idea behind backtracking is to build the solution incrementally, one piece at a time, and discard partial solutions as soon as it becomes clear they cannot lead to a valid solution.

When to Use Backtracking:

- When solving problems that involve combinatorial search, such as finding permutations, combinations, subsets, or paths.
- In problems that have a constraint that must be satisfied for a solution to be valid (e.g., Sudoku or n-queens).
- When the solution space is large, but it's possible to prune branches early, avoiding unnecessary exploration of invalid solutions.

Backtracking is often used to solve problems where we need to explore all possible solutions but can eliminate many possibilities early (i.e., pruning). It is similar to brute force, but it is more efficient because it avoids trying out solutions that are guaranteed to fail.

Grind 75 Problems

Here are the **Grind 75** problems that are suitable for solving using the backtracking approach:

1. **Subsets**
2. **Permutations**
3. **Sudoku Solver**

Solutions & Trade-offs

1. Subsets

💡 **Problem:** Given an integer array `nums`, return all possible subsets (the power set).

Approach:

Backtracking is a natural choice for solving the subsets problem because it allows us to generate all possible subsets by making a decision at each step: whether to include the current element in the subset or not.

- **Time Complexity:** $O(2^n)$, where n is the number of elements in the input array. This is because we have two choices for each element (include or exclude), and there are 2^n subsets in total.
- **Space Complexity:** $O(n)$, due to the recursive stack space and the storage needed for the output.

Python Implementation

```python
def subsets(nums):
    result = []

    def backtrack(start, current):
        result.append(current[:])  # Append the current subset to the result
        for i in range(start, len(nums)):
            current.append(nums[i])  # Include the element
            backtrack(i + 1, current)  # Recursively explore with the next elements
            current.pop()  # Backtrack and remove the last element

    backtrack(0, [])
    return result
```

Trade-offs:

- **Recursive approach:** Backtracking with recursion is simple and intuitive. However, it can lead to deep recursion for large input arrays, which may cause stack overflow in extreme cases.

- **Iterative approach:** An iterative approach using bit manipulation can also be used for generating subsets in O(2^n) time, but it may be less readable and intuitive than the recursive backtracking approach.

2. Permutations

💡 **Problem:** Given a collection of distinct integers, return all possible permutations.

Approach:

Backtracking works well for generating permutations since we need to decide whether to include an element at each position. The approach typically involves swapping elements in-place and generating all permutations by exploring all possible arrangements.

- **Time Complexity:** O(n!), where n is the number of elements in the input array. This is because there are n! possible permutations of n elements.
- **Space Complexity:** O(n), as we only need space for the current permutation and the recursive call stack.

Python Implementation

```python
def permute(nums):
    result = []

    def backtrack(start):
        if start == len(nums):
            result.append(nums[:])  # Add the current permutation to the result
            return
        for i in range(start, len(nums)):
            nums[start], nums[i] = nums[i], nums[start]  # Swap elements
            backtrack(start + 1)  # Recurse with the next position
            nums[start], nums[i] = nums[i], nums[start]  # Backtrack (restore the
swap)

    backtrack(0)
    return result
```

Trade-offs:

- **Recursive approach:** This is the most common and clean solution, but for large arrays, the number of permutations grows factorially, which may become inefficient for larger inputs.
- **Iterative approach:** Permutations can also be generated iteratively using an algorithm like Heap's algorithm, but backtracking is more flexible and easier to understand for many cases.

3. Sudoku Solver

💡 **Problem:** Solve a given Sudoku puzzle by filling the empty cells.

Approach:

Backtracking is ideal for this problem, as it involves trying out possible numbers for each empty cell and "backtracking" when we encounter a conflict (i.e., when a number is repeated in a row, column, or 3x3 grid).

- **Time Complexity:** $O(9^m)$, where m is the number of empty cells. In the worst case, we may need to try all 9 possible digits for each empty cell, though the solution is often found much sooner due to pruning.
- **Space Complexity:** $O(m)$, where m is the number of empty cells. The space is used for the recursive stack and the board state.

Python Implementation

```python
def solveSudoku(board):
    def is_valid(board, row, col, num):
        # Check if the number is not repeated in the row, column, or 3x3 grid
        for i in range(9):
            if board[row][i] == num or board[i][col] == num:
                return False
            if board[3 * (row // 3) + i // 3][3 * (col // 3) + i % 3] == num:
                return False
        return True

    def backtrack(board):
        for row in range(9):
            for col in range(9):
                if board[row][col] == '.':  # Find an empty cell
                    for num in '123456789':
```

```
                    if is_valid(board, row, col, num):
                        board[row][col] = num  # Try the number
                        if backtrack(board):  # Recursively fill the next cell
                            return True
                        board[row][col] = '.'  # Backtrack if it leads to a
 dead-end
                return False  # No valid number can be placed here
        return True  # All cells are filled

    backtrack(board)
```

Trade-offs:

- **Recursive approach:** The backtracking approach is intuitive and straightforward. However, for large puzzles or complex constraints, it may be inefficient without proper pruning.
- **Iterative approach:** An iterative approach using constraint propagation (like the AC-3 algorithm) can be more efficient, but backtracking is easier to implement and understand for Sudoku puzzles.

Recursive vs. Iterative Approaches

- **Recursive approach (Backtracking):** This approach is generally easier to implement and understand for problems like subsets, permutations, and Sudoku. It allows for elegant exploration of all possibilities and pruning of invalid branches. However, recursion can be inefficient for large inputs due to the deep call stacks, and can sometimes lead to stack overflow errors.
- **Iterative approach:** While iterative solutions like using bit manipulation or generating permutations using an explicit stack can be more efficient in terms of space, they are often more complex to implement and less intuitive. Backtracking via recursion is typically the best approach for problems that naturally fit this paradigm.

Summary

Backtracking is a powerful technique for solving problems that involve exploring all potential solutions, particularly for combinatorial problems such as finding

subsets, permutations, or solving puzzles. The key benefits of backtracking are:

- It is often simple to implement using recursion.
- It can prune invalid solutions early, making it more efficient than brute force.
- The trade-off involves dealing with potentially deep recursion or the need for additional optimizations for large inputs.

By understanding how and when to use backtracking, you'll be able to solve a variety of complex problems that require an exhaustive search through possible combinations while efficiently eliminating unfeasible options.

Chapter 12: Greedy Algorithms

Why Greedy Algorithms?

Greedy algorithms are a fundamental problem-solving technique that work by making the locally optimal choice at each step with the hope of finding a global optimum. They are particularly useful in **optimization problems**, such as **scheduling, graph algorithms, and interval selection**. However, understanding when a greedy approach works is key—greedy solutions do not always lead to the optimal solution.

Key Use Cases

- **Optimization Problems**: Finding the best solution within constraints.
- **Scheduling & Resource Allocation**: Maximizing utilization with minimum resources.
- **Graph Problems**: Algorithms like Dijkstra's shortest path and Prim's MST.

Example Problems

1. **Jump Game** (Medium)
2. **Gas Station** (Medium)
3. **Interval Scheduling Maximization** (like **Activity Selection Problem**)

Greedy Algorithm Strategies & Implementations

1. Jump Game (Greedy Traversal)

The **Jump Game** problem requires determining if one can reach the last index of an array given certain jump constraints. A greedy approach works by tracking the **furthest reachable index**.

Implementation

```
def can_jump(nums):
    max_reach = 0
```

```
    for i, jump in enumerate(nums):
        if i > max_reach:
            return False
        max_reach = max(max_reach, i + jump)
    return True
```

Time Complexity: O(N)

Space Complexity: O(1)

2. Gas Station (Circular Route Optimization)

The **Gas Station** problem requires determining if a circular route can be completed given gas constraints. The greedy solution involves tracking the net fuel balance and restarting the journey from potential candidates.

Implementation

```
def can_complete_circuit(gas, cost):
    total, tank, start = 0, 0, 0
    for i in range(len(gas)):
        diff = gas[i] - cost[i]
        total += diff
        tank += diff
        if tank < 0:
            start = i + 1
            tank = 0
    return start if total >= 0 else -1
```

Time Complexity: O(N)

Space Complexity: O(1)

3. Interval Scheduling Maximization (Activity Selection Problem)

This classic **interval scheduling problem** involves selecting the maximum number of non-overlapping intervals. A greedy approach sorts by **end time** and selects intervals accordingly.

Implementation

```python
def max_non_overlapping_intervals(intervals):
    intervals.sort(key=lambda x: x[1])
    count, last_end = 0, float('-inf')
    for start, end in intervals:
        if start >= last_end:
            count += 1
            last_end = end
    return count
```

Time Complexity: O(N log N) (sorting step)

Space Complexity: O(1)

Trade-offs & Complexity Analysis

Approach	Time Complexity	Space Complexity	Notes
Jump Game (Greedy)	O(N)	O(1)	Works since reaching the farthest index is optimal
Gas Station (Greedy Selection)	O(N)	O(1)	Greedy approach ensures a valid start if possible
Interval Scheduling (Sorting)	O(N log N)	O(1)	Sorting ensures optimal selection of intervals

Key Takeaways

1. **Greedy algorithms work well for optimization problems** where local choices lead to a global optimum.
2. **Sorting-based greedy strategies** are common in scheduling problems.
3. **Greedy approaches don't always work**—validating correctness is crucial.

Practice Problems

- LeetCode 55: Jump Game
- LeetCode 134: Gas Station
- LeetCode 435: Non-overlapping Intervals

Conclusion

Greedy algorithms provide efficient solutions for many optimization problems. While they don't always guarantee optimality, understanding their applicability and limitations is crucial for solving interview problems effectively.

Chapter 13: Topological Sorting (Advanced Graphs)

Why Topological Sorting?

Topological Sorting is a fundamental algorithm in **graph theory** used for problems involving **dependency resolution**, **build order**, and **task scheduling**. It applies to **Directed Acyclic Graphs (DAGs)** and provides a linear ordering of nodes such that for every directed edge u -> v, node u appears before v in the ordering.

Key Use Cases

- **Dependency Resolution**: Ensuring correct execution order in systems like package managers.
- **Build Order Problems**: Determining the correct sequence of tasks with dependencies.
- **Scheduling Tasks**: Solving real-world scheduling and precedence problems efficiently.

Example Problems

1. **Course Schedule** (Medium)
2. **Alien Dictionary** (Hard)

Topological Sorting Algorithms

1. Kahn's Algorithm (BFS Approach)

This approach uses **indegree tracking** to iteratively process nodes with zero incoming edges.

Algorithm Steps

1. Compute the indegree (number of incoming edges) for each node.
2. Add all nodes with indegree 0 to a queue.
3. While the queue is not empty:

- Remove a node from the queue and append it to the topological order.
- Reduce the indegree of its neighbors.
- If any neighbor's indegree becomes 0, add it to the queue.

4. If all nodes are processed, return the order; otherwise, a cycle exists.

Implementation

```
from collections import deque

def topological_sort_kahn(graph, num_nodes):
    indegree = {i: 0 for i in range(num_nodes)}
    for node in graph:
        for neighbor in graph[node]:
            indegree[neighbor] += 1

    queue = deque([node for node in indegree if indegree[node] == 0])
    topo_order = []

    while queue:
        node = queue.popleft()
        topo_order.append(node)
        for neighbor in graph[node]:
            indegree[neighbor] -= 1
            if indegree[neighbor] == 0:
                queue.append(neighbor)

    return topo_order if len(topo_order) == num_nodes else []  # Detect cycle
```

2. DFS-Based Topological Sorting

A **Depth-First Search (DFS) approach** recursively explores nodes and records the **finishing order** to determine the topological order.

Algorithm Steps

1. Maintain a visited set to track processed nodes.
2. Perform DFS and push nodes to a stack after all their neighbors are visited.
3. The final topological order is obtained by reversing the stack.

Implementation

```
def topological_sort_dfs(graph, num_nodes):
    visited = set()
```

```
    stack = []

    def dfs(node):
        if node in visited:
            return
        visited.add(node)
        for neighbor in graph[node]:
            dfs(neighbor)
        stack.append(node)

    for node in range(num_nodes):
        if node not in visited:
            dfs(node)

    return stack[::-1]  # Reverse to get correct order
```

Trade-offs & Complexity Analysis

Approach	Time Complexity	Space Complexity	Notes
Kahn's Algorithm (BFS)	O(V + E)	O(V + E)	Good for iterative processing
DFS-based Sorting	O(V + E)	O(V + E)	Useful for problems involving recursion

Key Takeaways

1. **Use Kahn's Algorithm (BFS)** when iterative processing is required (e.g., resolving dependencies in layers).
2. **Use DFS-Based Sorting** when recursion is natural and we need to track finishing order.
3. **Topological sorting only works on DAGs**—cycle detection is crucial before applying it.

Practice Problems

- LeetCode 207: Course Schedule
- LeetCode 210: Course Schedule II
- LeetCode 269: Alien Dictionary

Conclusion

Topological Sorting is an essential technique for solving dependency-based problems efficiently. By mastering **Kahn's Algorithm (BFS)** and **DFS-based Sorting**, you can tackle complex scheduling, ordering, and graph traversal problems effectively.

Chapter 14: Union-Find (Disjoint Set Union - DSU)

Why?

Union-Find (also called Disjoint Set Union, DSU) is a powerful data structure for efficiently solving **dynamic connectivity problems** in graphs. It is particularly useful for:

- **Finding connected components**: Identifying groups of nodes that are connected.
- **Cycle detection**: Checking whether adding an edge forms a cycle in an undirected graph.
- **Merging related entities**: Used in problems like clustering, network connectivity, and Kruskal's algorithm for Minimum Spanning Trees (MST).

Union-Find provides nearly **O(1) (amortized)** time complexity for key operations when optimized using **path compression** and **union by rank/size**.

Core Operations

Union-Find supports two main operations:

1. **Find(x)**: Determines the representative (root) of the set containing x.
2. **Union(x, y)**: Merges the sets containing x and y.

Optimizations:

- **Path Compression**: Makes future `find(x)` calls faster by directly linking nodes to their root.
- **Union by Rank/Size**: Ensures smaller trees attach to larger ones, keeping the structure balanced.

With both optimizations, Union-Find operates in **O(α(N))**, where **α(N)** (inverse Ackermann function) is **almost constant** for practical inputs.

Example Problems and Solutions

1. Number of Provinces (Medium)

Problem: Given an `n x n` adjacency matrix representing a graph, find the number of connected components (provinces).

Approach:

- Treat each node as its own component.
- Iterate through the adjacency matrix and union connected nodes.
- Count the number of unique root nodes.

Python Solution:

```python
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [1] * n

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])  # Path compression
        return self.parent[x]

    def union(self, x, y):
        root_x = self.find(x)
        root_y = self.find(y)
        if root_x != root_y:
            if self.rank[root_x] > self.rank[root_y]:
                self.parent[root_y] = root_x
            elif self.rank[root_x] < self.rank[root_y]:
                self.parent[root_x] = root_y
            else:
                self.parent[root_y] = root_x
                self.rank[root_x] += 1

def findCircleNum(isConnected):
    n = len(isConnected)
    uf = UnionFind(n)

    for i in range(n):
        for j in range(i + 1, n):  # Avoid redundant checks
            if isConnected[i][j] == 1:
                uf.union(i, j)
```

```
    return len(set(uf.find(i) for i in range(n)))

# Example
isConnected = [[1,1,0],[1,1,0],[0,0,1]]
print(findCircleNum(isConnected))  # Output: 2
```

Trade-offs:

- **Better than DFS/BFS** for larger graphs since it avoids recursion depth issues.
- **Nearly O(1) operations** with optimizations, making it highly efficient.

2. Redundant Connection (Medium)

Problem: Given a tree (graph with n nodes and n−1 edges) with one extra edge, find the **redundant** edge that creates a cycle.

Approach:

- Initialize Union-Find for n nodes.
- Process each edge and perform union(x, y).
- If x and y are already in the same set, that edge is redundant.

Python Solution:

```
def findRedundantConnection(edges):
    uf = UnionFind(len(edges))

    for u, v in edges:
        if uf.find(u − 1) == uf.find(v − 1):  # Already connected
            return [u, v]
        uf.union(u − 1, v − 1)

# Example
edges = [[1,2],[1,3],[2,3]]
print(findRedundantConnection(edges))  # Output: [2,3]
```

Trade-offs:

- **More efficient than DFS/BFS** for cycle detection in undirected graphs.
- **Scales well** for large graphs, as opposed to an adjacency list approach.

3. Accounts Merge (Hard)

Problem: Given a list of accounts (each containing an email list), merge accounts with overlapping emails.

Approach:

- Use a **Union-Find structure** to group emails into components.
- Store an email → index mapping to track connected components.
- Sort and format the merged accounts.

Python Solution:

```python
from collections import defaultdict

def accountsMerge(accounts):
    uf = UnionFind(len(accounts))
    email_to_index = {}

    for i, account in enumerate(accounts):
        for email in account[1:]:
            if email in email_to_index:
                uf.union(i, email_to_index[email])
            email_to_index[email] = i

    index_to_emails = defaultdict(set)
    for email, index in email_to_index.items():
        index_to_emails[uf.find(index)].add(email)

    return [[accounts[i][0]] + sorted(emails) for i, emails in
index_to_emails.items()]

# Example
accounts = [
    ["John", "johnsmith@mail.com", "john_newyork@mail.com"],
    ["John", "johnsmith@mail.com", "john00@mail.com"],
    ["Mary", "mary@mail.com"],
    ["John", "johnnybravo@mail.com"]
]
print(accountsMerge(accounts))
```

Trade-offs:

- **Faster than DFS-based merging** for large datasets.
- **Efficient for millions of emails**, avoiding repeated DFS traversals.

When to Use Union-Find

Problem Type	Union-Find?	Why?
Connected Components	✅ Yes	Efficient for merging nodes dynamically.
Cycle Detection (Undirected Graphs)	✅ Yes	Detects cycles in O(1) operations.
Cycle Detection (Directed Graphs)	❌ No	Use **DFS** or **Kahn's Algorithm** instead.
Kruskal's Algorithm (MST)	✅ Yes	Quickly processes edge unions.
Path Queries (Dynamic Graph)	✅ Yes	Answers connectivity in **O(1) amortized time**.

Conclusion

Union-Find (Disjoint Set Union) is a crucial technique for **connected components, cycle detection, and merging problems**. With **path compression** and **union by rank**, it achieves **almost constant-time operations**, making it one of the most **efficient** ways to handle dynamic connectivity in graphs.

Would you like to add more problem variations or explanations? 🚀

Chapter 15: Trie (Prefix Tree)

Why Trie?

The **Trie (Prefix Tree)** is a specialized tree data structure used for **efficient storage and retrieval of strings**, particularly in scenarios involving **autocomplete systems**, **dictionary implementations**, and **prefix-based searches**. Unlike hash tables, Tries enable prefix queries in **O(M)** time, where **M** is the length of the word.

Key Use Cases

- **Efficient Word Storage & Retrieval**: Fast lookups for dictionaries and word-based searches.
- **Autocomplete & Search Suggestions**: Used in search engines and predictive text.
- **Prefix-Based Searches**: Finding words with common prefixes quickly.

Example Problems

1. **Implement Trie (Prefix Tree)** (Medium)
2. **Word Search II** (Hard)

Trie Data Structure & Implementation

1. Trie Node & Basic Operations

A Trie consists of nodes where:

- Each node has a **dictionary of children** (representing characters).
- A **boolean flag** indicates if a word ends at that node.

Trie Implementation

```python
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False
```

```python
class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word: str):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end_of_word = True

    def search(self, word: str) -> bool:
        node = self._find_node(word)
        return node is not None and node.is_end_of_word

    def starts_with(self, prefix: str) -> bool:
        return self._find_node(prefix) is not None

    def _find_node(self, prefix: str):
        node = self.root
        for char in prefix:
            if char not in node.children:
                return None
            node = node.children[char]
        return node
```

2. Word Search II (Trie + DFS)

The **Word Search II** problem is an advanced Trie application where we combine
Trie construction with **Depth-First Search (DFS)** to efficiently find words in a
grid.

Optimized Approach (Trie + DFS + Backtracking)

```python
class Solution:
    def findWords(self, board: List[List[str]], words: List[str]) -> List[str]:
        trie = Trie()
        for word in words:
            trie.insert(word)

        rows, cols = len(board), len(board[0])
        result, visited = set(), set()

        def dfs(r, c, node, path):
```

```python
            if (r, c) in visited or not (0 <= r < rows and 0 <= c < cols):
                return
            char = board[r][c]
            if char not in node.children:
                return

            visited.add((r, c))
            node = node.children[char]
            path += char

            if node.is_end_of_word:
                result.add(path)

            for dr, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                dfs(r + dr, c + dc, node, path)

            visited.remove((r, c))

    for r in range(rows):
        for c in range(cols):
            dfs(r, c, trie.root, "")

    return list(result)
```

Trade-offs & Complexity Analysis

Approach	Time Complexity	Space Complexity	Notes
Trie Insert/Search	O(M)	O(N * M)	Fast lookups but uses extra space
Trie + DFS (Word Search II)	O(N _ M _ 4^L)	O(N * M)	Efficient for word grids but backtracking can be costly

Key Takeaways

1. **Tries are useful for prefix-based queries**, offering a major advantage over hash tables.
2. **Word Search II combines Tries with DFS** to efficiently find words in a matrix.
3. **Trade-offs exist between space efficiency and performance**, but Tries are optimal for problems requiring fast prefix searches.

Practice Problems

- LeetCode 208: Implement Trie (Prefix Tree)
- LeetCode 211: Design Add and Search Words Data Structure
- LeetCode 212: Word Search II

Conclusion

Tries are a powerful data structure for solving word-based problems efficiently. Mastering **basic Trie operations** and **Trie-based search optimizations** (such as DFS in Word Search II) is crucial for excelling in string-processing problems in coding interviews.

Chapter 16: Monotonic Stack (Stack-based Problems)

Why Monotonic Stack?

The **Monotonic Stack** is a specialized stack data structure used in problems that require finding the **next greater element**, **next smaller element**, or solving **range-based calculations** efficiently. It maintains elements in a strictly increasing or decreasing order, allowing us to process elements in linear time instead of quadratic time.

Key Use Cases

- **Next Greater Element Problems**: Efficiently find the next greater or smaller element for each item in an array.
- **Histogram-based Problems**: Solve problems related to **largest rectangle in histogram** or **maximal rectangle**.
- **Sliding Window Problems**: Maintain a stack of useful elements while iterating through a window of values.

Example Problems (Grind 75 & Beyond)

1. **Largest Rectangle in Histogram** (Hard)
2. **Daily Temperatures** (Medium)
3. **Next Greater Element I & II** (Medium)

Monotonic Stack Implementation & Variants

1. Finding Next Greater Element (NGE)

The most common monotonic stack application is to find the **Next Greater Element (NGE)** for each index in an array.

Brute Force Approach

A naive way would be to use two nested loops to check for the next greater element, leading to an **O(N²)** time complexity.

```python
# Brute Force: O(N²) Time Complexity
def next_greater_element(nums):
    result = [-1] * len(nums)
    for i in range(len(nums)):
        for j in range(i + 1, len(nums)):
            if nums[j] > nums[i]:
                result[i] = nums[j]
                break
    return result
```

Optimized Monotonic Stack Approach

We use a **decreasing stack** (stores indices of elements in decreasing order) to efficiently find the next greater element in **O(N)** time.

```python
# Monotonic Stack: O(N) Time Complexity
def next_greater_element(nums):
    result = [-1] * len(nums)
    stack = []  # Stores indices

    for i in range(len(nums)):
        while stack and nums[i] > nums[stack[-1]]:
            index = stack.pop()
            result[index] = nums[i]
        stack.append(i)

    return result
```

2. Largest Rectangle in Histogram

The **Largest Rectangle in Histogram** problem is a classic example where a **monotonic increasing stack** helps track potential heights efficiently.

Optimized Stack Approach

Instead of recalculating left and right bounds for each bar separately, we use a stack to track indices and compute the maximum area in **O(N)** time.

```python
# Monotonic Stack: O(N) Time Complexity
def largest_rectangle_area(heights):
    heights.append(0)  # Sentinel to flush stack at the end
    stack = []  # Stores indices of histogram bars
    max_area = 0

    for i, h in enumerate(heights):
        while stack and heights[stack[-1]] > h:
            height = heights[stack.pop()]
            width = i if not stack else i - stack[-1] - 1
            max_area = max(max_area, height * width)
        stack.append(i)

    return max_area
```

3. Daily Temperatures

Given a list of daily temperatures, find how many days you have to wait for a warmer temperature.

```python
# Monotonic Stack: O(N) Time Complexity
def daily_temperatures(temperatures):
    result = [0] * len(temperatures)
    stack = []  # Stores indices

    for i, temp in enumerate(temperatures):
        while stack and temp > temperatures[stack[-1]]:
            index = stack.pop()
            result[index] = i - index
        stack.append(i)

    return result
```

Trade-offs & Complexity Analysis

Approach	Time Complexity	Space Complexity	Notes
Brute Force	O(N²)	O(1)	Inefficient for large input sizes
Monotonic Stack	O(N)	O(N)	Optimal for stack-based problems

Key Takeaways

1. **Use monotonic stacks** for problems that require range-based calculations efficiently.
2. **Stack direction matters**: Increasing stacks are used for **finding next smaller elements**, while decreasing stacks help in **next greater element problems**.
3. **Histogram problems** can be solved efficiently using a stack-based approach to keep track of valid widths.

Practice Problems

- LeetCode 84: Largest Rectangle in Histogram
- LeetCode 739: Daily Temperatures
- LeetCode 503: Next Greater Element II

Conclusion

The **Monotonic Stack** is an essential tool in tackling stack-based problems efficiently, often reducing the complexity from $O(N^2)$ **to** $O(N)$. Mastering this technique is crucial for solving range queries, histogram problems, and sliding window optimizations effectively.

Chapter 17: Bit Manipulation

Concept & When to Use

Bit manipulation involves directly manipulating bits (the 0s and 1s) that make up data. It is a low-level operation that can be used for efficient problem-solving, especially when space or time complexity is a concern. Bit manipulation problems often involve performing operations such as AND, OR, XOR, shifting, and toggling to solve specific challenges.

Common Bit Manipulation Operations:

1. **AND (&):** Performs a logical AND between two bits. Only returns 1 when both bits are 1.
2. **OR (|):** Performs a logical OR between two bits. Returns 1 if at least one bit is 1.
3. **XOR (^):** Performs a logical XOR between two bits. Returns 1 if the bits are different.
4. **NOT (~):** Inverts all the bits (i.e., turns 0s to 1s and vice versa).
5. **Bit Shifts:** Moves the bits left (<<) or right (>>). This is often used to multiply or divide by powers of 2.

When to Use Bit Manipulation:

- When a problem requires working with binary numbers or flags.
- To reduce space or time complexity, especially in problems that involve checking subsets, counting bits, or performing bitwise arithmetic.
- When dealing with large datasets where other methods may be too slow or inefficient.

Grind 75 Problems

Here are the **Grind 75** problems that make use of bit manipulation techniques:

1. **Single Number**
2. **Reverse Bits**

Solutions & Trade-offs

1. Single Number

💡 **Problem:** Given a non-empty array of integers, every element appears twice except for one. Find that single one.

Approach: XOR

One of the most common and efficient bit manipulation techniques for this problem is XOR. The XOR operation has the following properties:

- `a ^ a = 0`: XORing a number with itself results in 0.
- `a ^ 0 = a`: XORing a number with 0 results in the number itself.
- XOR is commutative and associative.

If we XOR all the numbers together, the pairs will cancel out because of the first property (`a ^ a = 0`), leaving only the single number.

- **Time Complexity:** O(n), where n is the number of elements in the array.
- **Space Complexity:** O(1), as we only need a single variable to store the result.

Python Implementation

```python
def singleNumber(nums):
    result = 0
    for num in nums:
        result ^= num  # XOR operation
    return result
```

Trade-offs:

- **XOR approach:** This approach is highly efficient in terms of time and space complexity. It is simple and works perfectly for this type of problem.

- **Set-based approach:** A more naive approach would involve using a set to store the numbers that have been seen and check for duplicates. However, this solution would have O(n) time complexity with O(n) space complexity, which is less efficient compared to the XOR approach.

2. Reverse Bits

💡 **Problem:** Reverse bits of a given 32-bit unsigned integer.

Approach: Bit Shifting

The approach involves reversing the bits by repeatedly shifting the bits from the input number and placing them into a new number. You can achieve this by shifting the result to the left and shifting the input number to the right while checking and extracting the rightmost bit.

- **Time Complexity:** O(1) (since we only need 32 operations for a 32-bit number).
- **Space Complexity:** O(1), as we only need a fixed amount of space for the result.

Python Implementation

```python
def reverseBits(n):
    result = 0
    for _ in range(32):
        result = (result << 1) | (n & 1)  # Shift result left and add the rightmost bit of n
        n >>= 1  # Shift n right by 1
    return result
```

Trade-offs:

- **Bit shifting approach:** This is the most efficient approach for reversing bits because it operates in constant time (O(1)) and space (O(1)). It also doesn't require additional space for storing the binary representation or performing unnecessary calculations.

- **Set-based approach:** Another method might involve converting the number to binary and reversing the string representation, but this is less efficient in terms of both time and space, and it involves additional steps like converting back to an integer.

XOR vs. Set-based Approach

- **XOR** is a powerful operation for problems where elements appear in pairs or need to be canceled out. It offers optimal time and space complexity (O(n) and O(1), respectively) and is often the best choice for problems like finding the single number or detecting duplicates.
- **Set-based approach** is less efficient when dealing with problems that involve bit-level operations. While simple to implement, it has a higher space complexity (O(n)) and can be slower for large inputs, as it requires extra space and operations like inserting and checking for duplicates.

When to Use XOR:

- When the problem involves finding unique numbers in an array where duplicates cancel each other out.
- XOR is highly efficient for problems involving pairs, like "Single Number," "Find the Two Non-Repeating Numbers," or parity-related tasks.

When to Use a Set-based Approach:

- When dealing with problems that don't have the properties that make XOR efficient (e.g., problems where you need to track all unique elements or don't have a clear cancellation property).
- Set-based solutions are easier to understand and can be helpful for simpler problems or when learning bit manipulation concepts.

Summary

- **Bit Manipulation** is a highly efficient way to solve problems that deal with binary operations and bits.

- **XOR** is often the optimal approach for problems where elements cancel each other out, such as in the "Single Number" problem.
- **Bit Shifting** is a powerful technique for reversing bits and other bit-level operations.
- **Set-based approaches** are less efficient than bit manipulation but can be used for simpler problems or as a stepping stone to learning bit manipulation.

By mastering these bit manipulation techniques, you'll be able to solve a variety of problems efficiently, especially those that require handling binary representations or optimizing space and time complexity.

Chapter 18: Kadane's Algorithm (Maximum Subarray)

Why?

Kadane's Algorithm is a **greedy + dynamic programming** technique that efficiently finds the **maximum sum subarray** in an array. It is commonly used when:

- **Finding the largest contiguous sum in an array** (e.g., stock market analysis, gaming scores).
- **Solving DP problems with subarray constraints** in **O(N) time** instead of **O(N²) or O(N³)** brute-force approaches.
- **Handling problems that require maintaining local/global optimal values** efficiently.

Kadane's Algorithm works because **a maximum subarray ending at index `i` must either**:

1. Extend the previous subarray (accumulate sum).
2. Start a new subarray at index `i` (if previous sum is negative).

This simple **"keep or restart"** decision makes Kadane's Algorithm both **greedy and optimal**.

Core Idea

We maintain two variables:

- `max_sum` → Tracks the maximum subarray sum found so far.
- `current_sum` → Tracks the current running sum.

At each index, update `current_sum` as:

$$\text{current\_sum} = \max(\text{current\_sum} + \text{nums}[i], \text{nums}[i])$$

Update `max_sum` if `current_sum` is larger.

Example Problems and Solutions

1. Maximum Subarray (Medium)

Problem: Given an array `nums`, find the contiguous subarray with the maximum sum.

Approach:

- Iterate through the array, maintaining `current_sum` and `max_sum`.
- If `current_sum` drops below 0, restart the subarray at the current index.
- Return `max_sum`.

Python Solution:

```python
def maxSubArray(nums):
    max_sum = float('-inf')
    current_sum = 0

    for num in nums:
        current_sum = max(current_sum + num, num)
        max_sum = max(max_sum, current_sum)

    return max_sum

# Example
nums = [-2,1,-3,4,-1,2,1,-5,4]
print(maxSubArray(nums))  # Output: 6  (Subarray: [4,-1,2,1])
```

Trade-offs:

✅ **O(N) time complexity** (optimal).

✅ **Constant space** (no extra storage needed).

⚠️ **Only works for sum-based problems** (modifications needed for other variations).

2. Maximum Product Subarray (Medium)

Problem: Find the contiguous subarray with the **maximum product**.

Challenge:

- Unlike sums, **products can flip signs** (negative × negative = positive).
- We must track both **maximum** and **minimum** products at each step.

Approach:

- Maintain `max_product` and `min_product` (since a negative min product can turn into a large positive).
- At each step, update: $\text{temp\_max} = \max(\text{num}, \text{max\_product} \times \text{num}, \text{min\_product} \times \text{num})$ $\text{min\_product} = \min(\text{num}, \text{max\_product} \times \text{num}, \text{min\_product} \times \text{num})$
- `max_product` becomes `temp_max`.

Python Solution:

```python
def maxProduct(nums):
    max_product = min_product = result = nums[0]

    for num in nums[1:]:
        temp_max = max(num, max_product * num, min_product * num)
        min_product = min(num, max_product * num, min_product * num)
        max_product = temp_max

        result = max(result, max_product)

    return result

# Example
nums = [2,3,-2,4]
print(maxProduct(nums))  # Output: 6  (Subarray: [2,3])
```

Trade-offs:

✅ **Handles negative numbers correctly**.

✅ **O(N) time complexity** (optimal).

⚠️ **Requires extra tracking (min & max products)**.

When to Use Kadane's Algorithm

Problem Type	Kadane's Algorithm?	Why?
Maximum Sum Subarray	✅ Yes	Standard Kadane's Algorithm.
Maximum Product Subarray	✅ Yes (Modified)	Track both min & max.
Subarrays with Constraints	⚠️ Maybe	Needs variations (e.g., at most K elements).
Maximum Subarray with Removal	❌ No	DP may be better.
2D Grid (Max Sum)	❌ No	Use **Kadane's on rows**, then **prefix sums**.

Conclusion

Kadane's Algorithm is a **powerful greedy DP technique** for finding maximum subarrays in **O(N) time**. With minor modifications, it can handle **product subarrays, constraints, and grid problems**.

Chapter 19: Rolling Hash & Rabin-Karp (String Algorithms)

Why?

Rolling Hash and Rabin-Karp are **powerful hashing techniques** for **efficient string matching**, especially when dealing with:

- **Substring search** (e.g., checking if a pattern exists in a text).
- **Detecting repeated sequences** (e.g., plagiarism detection, bioinformatics).
- **Finding anagrams or scrambled substrings** efficiently.

Traditional **brute-force substring search** takes $O(N \times M)$ time (where N is text length, M is pattern length).

Rolling Hash reduces this to O(N) on average using a **sliding window hash function**.

Core Idea

Rolling Hash (Sliding Window Hashing)

Instead of recomputing the **entire hash** for each substring, **Rolling Hash** efficiently updates it in **O(1) time** when sliding the window.

For a string S of length N and a window of size M:

- Compute the **initial hash** of `S[:M]`.
- Slide the window: Remove the **left character**, add the **right character**, and compute the new hash efficiently.

Hash formula for a base B and prime P:

$$\text{Hash} = (B \times \text{previous\_hash} - \text{outgoing\_char} \times B^M + \text{incoming\_char}) \mod P$$

Rabin-Karp Algorithm (Pattern Matching)

Uses **rolling hash** for **fast substring matching**:

1. Compute the hash of the **pattern**.
2. Compute the hash of **each window in the text**.
3. If hashes match, **compare characters to confirm** (to avoid false positives).
4. Slide the window and update the hash in **O(1)**.

Example Problems and Solutions

1. Repeated DNA Sequences (Medium)

Problem: Given a DNA sequence s, find all **10-letter-long** substrings that appear more than once.

Approach:

- Use **rolling hash** (or **set-based lookup**) to detect duplicate substrings efficiently.

Python Solution (Rolling Hash)

```python
def findRepeatedDnaSequences(s):
    if len(s) < 10:
        return []

    seen = set()
    repeated = set()
    base = 4   # Since DNA has 4 characters (A, C, G, T)
    prime = 10**9 + 7
    hash_val = 0
    B_M = pow(base, 9, prime)  # Base^M-1 for rolling hash

    char_map = {'A': 0, 'C': 1, 'G': 2, 'T': 3}

    for i in range(len(s)):
        hash_val = (hash_val * base + char_map[s[i]]) % prime

        if i >= 9:   # When we have a valid 10-char window
            if hash_val in seen:
                repeated.add(s[i-9:i+1])
            seen.add(hash_val)
```

```
        # Remove leftmost char from hash (rolling hash update)
        hash_val = (hash_val - char_map[s[i-9]] * B_M) % prime

    return list(repeated)

# Example
s = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT"
print(findRepeatedDnaSequences(s))  # Output: ["AAAAACCCCC", "CCCCCAAAAA"]
```

✅ **O(N) complexity**, much faster than **O(N²) brute force**.

✅ **Memory-efficient hashing** instead of storing all substrings.

⚠️ **Potential hash collisions** (rare but possible).

2. Substring with Concatenation of All Words (Hard)

Problem: Given a string **s** and a list of words (all of same length), find all **starting indices** of substrings in **s** that are concatenations of all words in any order.

Approach:

- Each window contains a **permutation of the words**.
- **Rolling hash + sliding window** for fast checking.

Python Solution (HashMap + Sliding Window)

```
from collections import Counter

def findSubstring(s, words):
    if not s or not words:
        return []

    word_len = len(words[0])
    word_count = len(words)
    total_len = word_len * word_count
    word_map = Counter(words)
    result = []

    for i in range(word_len):  # Try all possible start positions
        left, right = i, i
        current_map = Counter()

        while right + word_len <= len(s):
```

```
                word = s[right:right + word_len]
                right += word_len

                if word in word_map:
                    current_map[word] += 1
                    while current_map[word] > word_map[word]:  # Too many instances of
a word

                        current_map[s[left:left + word_len]] -= 1
                        left += word_len
                    if right - left == total_len:  # Valid window
                        result.append(left)
                else:  # Invalid word, reset window
                    current_map.clear()
                    left = right

    return result

# Example
s = "barfoothefoobarman"
words = ["foo", "bar"]
print(findSubstring(s, words))  # Output: [0, 9]
```

✅ **Sliding window avoids unnecessary recomputation**.

✅ **Efficient O(N) solution** compared to brute-force **O(N × M!)**.

⚠️ **Does not use rolling hash (word-based hashmap instead)**.

When to Use Rolling Hash vs. Other String Matching Algorithms

Algorithm	Best Use Case	Time Complexity	Notes
Brute Force (O(N × M))	Short pattern matching	O(N × M)	Slow for large inputs
KMP (Knuth-Morris-Pratt)	Exact pattern matching	O(N + M)	Good when no hash needed
Rabin-Karp (Rolling Hash)	Multiple pattern matching	O(N) (avg)	Fast for large texts
Aho-Corasick (Trie + BFS)	Multi-pattern matching	O(N + M)	Works for dictionary-based lookups

Conclusion

- **Rolling Hash is an efficient O(N) technique for substring matching** (vs. O(N × M) brute force).
- **Rabin-Karp extends Rolling Hash for pattern matching** with quick hash comparisons.
- **Useful for detecting duplicates, plagiarism detection, DNA sequence analysis, and anagram search**.
- **Alternative string algorithms (KMP, Aho-Corasick) are better for certain problems**.

Chapter 20: Fenwick Tree / Segment Tree (Advanced Data Structures)

Why?

Some problems require **efficient range queries** (e.g., sum, min, max) while allowing **fast updates** to the data.

For such scenarios, **Fenwick Tree (Binary Indexed Tree, BIT)** and **Segment Tree** provide powerful solutions.

They are particularly useful for:

- **Prefix sum / range sum queries** with updates.
- **Range minimum / maximum queries (RMQ)**.
- **Dynamic problems where values are frequently modified** (e.g., stock prices, competitive programming).

Data Structure	Updates (`update()`)	Queries (`query()`)	Space Complexity
Fenwick Tree (BIT)	O(log N)	O(log N)	O(N)
Segment Tree	O(log N)	O(log N)	O(2N)

Fenwick Tree (Binary Indexed Tree - BIT)

A **Fenwick Tree** efficiently supports **prefix sum queries** and **point updates** in **O(log N)** time using bitwise operations.

It is **simpler and more memory-efficient** than a Segment Tree but **cannot handle range updates efficiently**.

Operations

1. **Update an index (`update(idx, val)`)** → Adds `val` to `arr[idx]` and propagates changes.
2. **Prefix sum query (`query(idx)`)** → Computes the sum of `arr[0]` to `arr[idx]`.
3. **Range sum query (`range_sum(left, right)`)** → Uses `query(right) – query(left-1)`.

Python Implementation

```python
class FenwickTree:
    def __init__(self, n):
        self.size = n
        self.tree = [0] * (n + 1)

    def update(self, index, value):
        while index <= self.size:
            self.tree[index] += value
            index += index & -index   # Move to parent

    def query(self, index):
        total = 0
        while index > 0:
            total += self.tree[index]
            index -= index & -index   # Move to previous index
        return total

    def range_sum(self, left, right):
        return self.query(right) - self.query(left - 1)

# Example Usage
arr = [1, 3, 5]
fenwick = FenwickTree(len(arr))

for i, num in enumerate(arr):
    fenwick.update(i + 1, num)   # BIT uses 1-based indexing

print(fenwick.range_sum(1, 2))   # Output: 4 (1 + 3)
```

✅ **Fast updates and queries in O(log N).**

⚠️ **Cannot efficiently handle range updates (use Segment Tree instead).**

Segment Tree

A **Segment Tree** is a **divide-and-conquer** data structure that stores information about array segments.

Unlike a Fenwick Tree, a **Segment Tree supports range queries on various operations** (sum, min, max, GCD, etc.).

Operations

1. **Build the tree (`build()`)** → Constructs the Segment Tree in **O(N)**.
2. **Point update (`update(idx, val)`)** → Modifies a value and updates affected segments.
3. **Range query (`query(left, right)`)** → Recursively queries segments in **O(log N)**.

Python Implementation

```python
class SegmentTree:
    def __init__(self, nums):
        self.n = len(nums)
        self.tree = [0] * (2 * self.n)  # Segment tree array
        self.build(nums)

    def build(self, nums):
        # Fill the leaves
        for i in range(self.n):
            self.tree[self.n + i] = nums[i]
        # Build the tree by calculating parents
        for i in range(self.n - 1, 0, -1):
            self.tree[i] = self.tree[2 * i] + self.tree[2 * i + 1]

    def update(self, index, value):
        index += self.n  # Move to leaf
        self.tree[index] = value  # Update leaf
        # Propagate updates to the root
        while index > 1:
            index //= 2
            self.tree[index] = self.tree[2 * index] + self.tree[2 * index + 1]

    def query(self, left, right):
        left += self.n  # Move to leaf range
        right += self.n
        total = 0
        while left <= right:
            if left % 2 == 1:  # If left is a right child, include it
```

```
                total += self.tree[left]
                left += 1
            if right % 2 == 0:  # If right is a left child, include it
                total += self.tree[right]
                right -= 1
            left //= 2
            right //= 2
        return total

# Example Usage
arr = [1, 3, 5]
seg_tree = SegmentTree(arr)

print(seg_tree.query(0, 2))  # Output: 9 (1 + 3 + 5)
seg_tree.update(1, 2)  # Update index 1 to 2
print(seg_tree.query(0, 2))  # Output: 8 (1 + 2 + 5)
```

✅ **Supports a variety of range queries (sum, min, max, etc.).**

✅ **Efficient for large-scale dynamic data updates.**

⚠️ **Takes more space (2N instead of N for Fenwick Tree).**

When to Use Fenwick Tree vs. Segment Tree

Use Case	Fenwick Tree	Segment Tree
Point updates + prefix sum queries	✅ Best Choice	✅ Works but overkill
Range sum queries	✅ Yes	✅ Yes
Range min/max/GCD queries	❌ No	✅ Best Choice
Range updates (lazy propagation)	❌ No	✅ Yes
Space efficiency	✅ O(N)	❌ O(2N)

Example Problem: Range Sum Query - Mutable

Problem: Given an array, efficiently perform:

1. `update(index, value)`: Change `arr[index] = value`.
2. `sumRange(left, right)`: Return sum of `arr[left:right]`.

Approach	Time Complexity

Brute Force (Iterate Every Time) O(N) per query

Fenwick Tree / BIT   **O(log N)** per update/query

Segment Tree   **O(log N)** per update/query

Optimized Python Solution (Fenwick Tree)

```python
class NumArray:
    def __init__(self, nums):
        self.nums = nums
        self.bit = FenwickTree(len(nums))
        for i, num in enumerate(nums):
            self.bit.update(i + 1, num)

    def update(self, index, val):
        self.bit.update(index + 1, val - self.nums[index])  # Difference update
        self.nums[index] = val

    def sumRange(self, left, right):
        return self.bit.range_sum(left + 1, right + 1)

# Example
nums = [1, 3, 5]
numArray = NumArray(nums)
print(numArray.sumRange(0, 2))  # Output: 9
numArray.update(1, 2)
print(numArray.sumRange(0, 2))  # Output: 8
```

✅ **Faster than brute force**.

⚠️ **Fenwick Tree works well for sum queries but not min/max queries**.

Conclusion

- **Fenwick Tree** is simple and memory-efficient but limited to **prefix sum queries**.
- **Segment Tree** is more versatile but requires **more space**.
- **Both structures provide O(log N) updates and queries**, making them crucial for **competitive programming** and **real-time range queries**.

Chapter 21: Mock Interviews and Strategies

How to Approach Problems Effectively

When you're faced with a coding problem in an interview, the key is not just to solve it but to solve it effectively. Here's a step-by-step guide on how to approach problems:

1. **Understand the Problem Statement:**
   - **Clarify Edge Cases:** Before jumping into coding, always ask for clarifications. What happens when the input is empty? Are there any constraints like maximum input size? Are there any special edge cases you should be aware of?
   - **Restate the Problem:** Briefly restate the problem in your own words to ensure you've understood it correctly. This will also help the interviewer see how you approach problem-solving.
2. **Plan Your Approach:**
   - **Break Down the Problem:** If the problem is complex, break it down into smaller, manageable parts. For example, if you're working on a tree traversal problem, think of it as traversing left and right subtrees and handling nodes individually.
   - **Choose a Data Structure:** Consider what data structures best fit the problem. For example, graphs are often best solved with adjacency lists, whereas array-based problems may benefit from sorting or the sliding window technique.
   - **Pick an Algorithm:** Select an algorithm that fits the problem. Think of efficient algorithms like sorting, dynamic programming, or binary search. If an optimal solution isn't immediately clear, solve it using a brute-force approach first, then optimize.
   - **Time Complexity:** Always consider the time and space complexity of your solution. Try to avoid brute-force solutions unless they're

acceptable within the given constraints. Aim for the best time complexity that solves the problem efficiently.

3. **Write the Code:**
   - **Start Simple:** Write the solution step by step, ensuring that it works for the simplest cases first.
   - **Incremental Development:** Test as you go. Don't wait until the whole solution is complete. This will help catch bugs early.
   - **Keep it Clean:** Maintain readable code with proper variable names. Don't rush to write cryptic code.

4. **Verify the Solution:**
   - **Test Cases:** After writing the code, run it with multiple test cases to ensure its correctness. Test edge cases, and make sure the solution handles large inputs efficiently.
   - **Ask for Feedback:** Once you've completed your solution, ask the interviewer for feedback. See if there's a different approach they would recommend, or if you could optimize the solution further.

Communicating During an Interview

Effective communication is a critical aspect of any interview, especially when you're working through a problem. Here's how to communicate well during your coding interview:

1. **Think Aloud:**
   - **Explain Your Thought Process:** As you approach the problem, describe the steps you're taking and the rationale behind your decisions. This will show the interviewer that you understand the problem and are thinking critically.
   - **Use Visuals:** If appropriate, draw diagrams or write out pseudocode. This helps the interviewer see your approach clearly.

2. **Ask Questions:**
   - **Clarify Assumptions:** Don't be afraid to ask questions about the problem statement, input constraints, or edge cases. This not only helps

clarify the problem but also shows that you're thinking deeply.

- ○ **Understand Requirements:** If a solution seems ambiguous, ask for examples of inputs and outputs to better understand the requirements.

3. **Explain Trade-offs:**

- ○ **Discuss Your Approach:** Once you've written your initial solution, explain why you chose that particular approach. Discuss the time and space complexity, and if you've used any particular algorithm or data structure, explain why it was suitable for the problem.
- ○ **Alternative Solutions:** If time permits, mention any alternative approaches you considered and why you chose the current one.

4. **Stay Calm and Positive:**

- ○ **Don't Panic:** If you get stuck, don't panic. Pause, take a breath, and try to approach the problem from a different angle. The interviewer is often more interested in seeing how you handle challenges than whether you solve the problem on your first try.
- ○ **Stay Positive:** Even if you don't immediately know the answer, express enthusiasm for solving the problem. A positive attitude shows resilience and problem-solving skills.

Debugging Efficiently

Debugging is an essential skill in coding interviews. Here's how to debug effectively during your interview:

1. **Stay Calm:**

- ○ **Don't Rush:** If you encounter an error or the code doesn't work, don't panic. Step back and methodically go through the problem.
- ○ **Isolate the Problem:** If the problem is complex, break it down and try to isolate where things are going wrong. Start with smaller inputs and gradually increase the complexity.

2. **Use Print Statements or Debuggers:**

- ○ **Print Statements:** Insert `print` statements to check intermediate values and track the flow of the program.

- **Debuggers:** If you're allowed to use a debugger, step through the code to identify where the logic is breaking.

3. **Test with Edge Cases:**
   - **Edge Cases:** Always test with edge cases after writing your code. For example, consider input that is empty, negative, or extremely large.
   - **Boundary Values:** Test for the lower and upper boundaries of input constraints. This often helps identify off-by-one errors or issues with loops.

4. **Revisit Your Algorithm:**
   - **Reassess Your Approach:** If the code doesn't work, consider whether you've chosen the right algorithm or data structure. Sometimes, an error arises because of a suboptimal approach.
   - **Re-check Requirements:** If you're consistently getting wrong results, revisit the problem statement. It's easy to miss a requirement, and a small oversight can lead to bugs.

5. **Ask for Help if Needed:**
   - **Don't Hesitate to Ask:** If you're stuck, don't be afraid to ask the interviewer for a hint. It's better to ask for a nudge in the right direction than to waste too much time in frustration.

Common Pitfalls

1. **Misunderstanding the Problem:**
   - **Not Asking Clarifying Questions:** Many candidates dive into coding without understanding the problem fully. Always take the time to ask questions and confirm edge cases.
   - **Assumptions:** Avoid making assumptions about input formats, constraints, or expected output. Verify everything with the interviewer.

2. **Over-Complicating the Solution:**
   - **Going for the Optimal Solution Too Early:** While it's important to think about optimization, it's equally important to get the basic solution working first. Don't try to optimize prematurely—get the brute-force solution first and optimize later if necessary.

- **Ignoring Simplicity:** Some problems have elegant, simple solutions, but candidates often overthink them. Always start with the simplest solution that works and improve upon it if needed.

3. **Not Testing Thoroughly:**
   - **Skipping Edge Cases:** It's easy to forget about edge cases like empty inputs, negative numbers, or large inputs. Always test these cases to ensure the solution is robust.
   - **Assuming the Code Works Without Testing:** Don't assume that your code works perfectly just because it runs for a few basic cases. Always run it with a variety of test cases.

4. **Not Communicating Clearly:**
   - **Not Explaining Your Thought Process:** Simply writing code without explaining it may leave the interviewer in the dark about your approach. Be sure to talk through your thought process, so the interviewer can follow your reasoning and provide feedback.
   - **Over-explaining:** On the other hand, over-explaining simple concepts or code can waste time. Strike a balance between being clear and being concise.

5. **Failing to Handle Time Pressure:**
   - **Panic Under Time Constraints:** Coding interviews often come with a time limit. Stay calm, and don't rush. Focus on writing correct code first, then optimize if you have time.
   - **Inability to Manage Time Efficiently:** Allocate time for each part of the interview—understanding the problem, planning your solution, coding, testing, and debugging. Don't spend too much time on any single part.

Summary

Mock interviews and practicing strategies are crucial to mastering coding interviews. By understanding how to approach problems methodically, communicating clearly with your interviewer, debugging efficiently, and avoiding common pitfalls, you can set yourself up for success. Here are the key takeaways:

- **Understand the problem thoroughly** before you start solving it.
- **Communicate your thought process** clearly and effectively.
- **Debug strategically** by isolating the issue and testing with edge cases.
- **Stay calm and manage time wisely** to perform well under pressure. By practicing mock interviews and following these strategies, you can improve both your problem-solving skills and your confidence during interviews.

Final Notes

Last-Minute Revision Checklist

As you approach your coding interview, it's essential to prepare effectively and focus on the key areas that will make the biggest impact. Use this checklist to make sure you're ready for the big day:

1. Review Core Algorithms & Data Structures

- **Key Patterns:** Make sure you've reviewed all major problem-solving patterns such as sliding window, two pointers, dynamic programming, and graph traversal algorithms. These are the core techniques that often appear in coding interviews.
- **Data Structures:** Double-check your understanding of essential data structures like arrays, linked lists, stacks, queues, hash maps, heaps, and trees. Know how to manipulate and traverse them efficiently.
- **Time & Space Complexity:** Revisit the concepts of Big O notation for time and space complexity. Be ready to discuss the trade-offs between different approaches, especially in terms of efficiency.

2. Practice Key Problems

- **Grind 75 Problems:** If you've been following the Grind 75 list, make sure you can solve all of them confidently. Practice solving them within a time limit to simulate real interview conditions.

- **Common Interview Questions:** Make sure you're comfortable with classic problems like:
  - **Array manipulation (e.g., Two Sum, Product of Array Except Self)**
  - **String problems (e.g., Longest Substring Without Repeating Characters, Anagram)**
  - **Dynamic programming (e.g., Coin Change, Longest Increasing Subsequence)**
  - **Graph problems (e.g., Number of Islands, Course Schedule)**

3. Mock Interviews

- **Simulate the Interview Environment:** Practice solving problems in a mock interview setting. Time yourself, and have a friend or mentor act as the interviewer. This will help you build the stamina and confidence to think and code under pressure.
- **Explain Your Solution:** Practice thinking out loud and explaining your approach clearly. Interviewers appreciate candidates who can articulate their thought process, and it helps build rapport.

4. Review Behavioral Questions

- **STAR Method:** Review behavioral questions and practice answering them using the STAR (Situation, Task, Action, Result) method. Be ready to discuss your past experiences in terms of how you faced challenges and solved problems.
- **Leadership Principles (if applying to Amazon or similar companies):** Prepare examples that demonstrate your ability to lead, take initiative, and collaborate effectively.

5. Review Code Snippets

- **Reusable Code:** Revisit any common code snippets you use often. These may include:
  - Code for reversing a string or linked list.

- Implementations of binary search.
- Methods for deep copying objects or arrays.
- **Edge Case Handling:** Have a mental checklist for edge cases such as empty arrays, large inputs, negative numbers, or null values.

6. Double-Check Your Setup

- **Coding Environment:** Ensure your coding environment is ready, whether it's an online coding platform or your IDE. Make sure you are comfortable with the tools you'll be using during the interview.
- **Internet Connection (for virtual interviews):** If you're doing a virtual interview, check your internet connection and make sure your webcam and microphone are working properly.
- **Comfortable Environment:** Ensure your environment is quiet, comfortable, and free from distractions.

7. Final Review of Key Interview Strategies

- **Time Management:** Be aware of how much time you should spend on each step: understanding the problem, planning your solution, coding, and testing. If you're stuck, ask the interviewer for clarification or a hint.
- **Stay Calm and Positive:** Remember that the interviewer is not only looking for a correct solution but also for how you approach problems. If you get stuck, stay calm, work through the problem, and demonstrate your problem-solving process.
- **Clarify Before Coding:** Make sure you fully understand the problem and the requirements before diving into coding. Don't hesitate to ask clarifying questions.

Recommended Resources

To further enhance your preparation, here's a list of valuable resources that will help you refine your coding and interview skills:

1. LeetCode

- **LeetCode Premium:** If possible, subscribe to LeetCode Premium to access more problems, company-specific questions, and solutions. It's a great platform for practicing coding problems.
- **LeetCode Explore:** LeetCode offers an "Explore" feature, where you can follow structured tracks for specific topics like dynamic programming, arrays, strings, and more.

2. Cracking the Coding Interview by Gayle Laakmann McDowell

- A comprehensive book that provides a deep dive into coding interview preparation, covering algorithms, problem-solving strategies, and behavioral questions.

3. Elements of Programming Interviews by Adnan Aziz

- This book includes a large set of practice problems, along with detailed solutions and explanations. It's a great resource for honing your skills in solving challenging coding problems.

4. GeeksforGeeks

- GeeksforGeeks is a fantastic online platform that offers tutorials, interview experiences, and an enormous library of coding problems with explanations and solutions.

5. Interviewing.io

- This platform provides mock interviews with engineers from top tech companies. It's an excellent way to practice solving problems in a real interview environment with feedback from professionals.

6. System Design Interview by Alex Xu

- If you're preparing for system design interviews, this book is a must-read. It provides a structured approach to solving system design problems and includes several real-world examples.

7. YouTube Channels

- **Tushar Roy – Coding Made Simple:** Offers great explanations for algorithms and data structures with a focus on coding interview preparation.
- **Tech Dummies – Tutorials:** A great channel for learning complex coding interview topics and tips for navigating the interview process.
- **Climbing the Coding Ladder:** Focuses on interview problem-solving, breaking down complex problems into manageable steps.

8. HackerRank and CodeSignal

- Both platforms provide coding challenges that simulate real interview environments. You can practice timed problems, participate in coding competitions, and get immediate feedback.

9. Mock Interview Platforms

- **Pramp:** A platform where you can practice mock interviews with peers. Pramp offers interview simulations for coding as well as behavioral and system design rounds.
- **Exponent:** Provides a series of mock interview questions and solutions, along with an interview coaching service.

10. Soft Skills and Behavioral Interview Prep

- **The Complete Guide to Behavioral Interviews:** A book focused on preparing you for the behavioral rounds of interviews, including questions on leadership, problem-solving, and conflict resolution.
- **Amazon Leadership Principles:** If you're targeting a company like Amazon, review their Leadership Principles thoroughly. Be ready to relate your past experiences to these principles.

Conclusion

Preparing for coding interviews can be overwhelming, but by staying organized and using the right resources, you can ensure you're ready for success. The key to acing coding interviews lies in practice, clear communication, and a calm, methodical approach to problem-solving.

As you move forward in your preparation, keep this final checklist handy, use the recommended resources to deepen your understanding, and make sure you stay focused on continuous improvement. Good luck with your interviews!

title: Coding Interview Prep

[Download PDF](#)