

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 Book?

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 book is designed to help you navigate coding interviews with a **Python-focused approach**, emphasizing **efficient solutions, trade-offs, and best practices**.

Python-Specific Optimizations

Python is a powerful language for coding interviews due to its **expressiveness, built-in data structures, and concise syntax**. However, its **dynamic typing, memory management, and built-in functions** introduce unique challenges. This book helps you:

- **Leverage Python's built-in functions and libraries** to write cleaner and more efficient code.
- **Understand Pythonic solutions** and when they outperform traditional approaches.
- **Optimize time and space complexity** while using Python's features effectively.

Problem-Solving Patterns for Structured Learning

Instead of solving problems randomly, this book 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 book 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 Book

Mastering coding interviews is not about memorizing solutions but about **understanding patterns and trade-offs**. This book 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 book 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 book 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 book 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 used to optimize 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**.
- ✓ The problem requires finding an **optimal subarray** (e.g., maximum/minimum sum, longest/shortest length).
- ✓ Brute-force solutions involve **recomputing overlapping parts** of an array.

Types of Sliding Window Approaches

- ◆ **Fixed-size window:** When the window size is predetermined (e.g., "find the max sum of a subarray of size k ").
 - ◆ **Dynamic-size window:** When the window size changes based on conditions (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. **Longest Substring Without Repeating Characters** (LeetCode #3)
2. **Permutation in String** (LeetCode #567)
3. **Minimum Window Substring** (LeetCode #76)

Each of these problems can be **solved efficiently** using a **dynamic sliding window approach**. Below, we analyze each problem and discuss brute-force vs. optimized solutions.

Solutions & Trade-offs

1. Longest Substring Without Repeating Characters

💡 **Problem:** Given a string S , find the length of the **longest substring** without repeating characters.

Brute-Force Approach ($O(n^2)$)

- Try **all substrings** and check if they have unique characters.
- **Time Complexity:** $O(n^2)$ – for each start index, try all possible end indices.
- **Space Complexity:** $O(n)$ – for storing unique characters in a set.

Optimized Sliding Window Approach ($O(n)$)

- Use a **hash set** to store unique characters.
- Maintain a **left pointer** (l) and expand the **right pointer** (r), adjusting l when duplicates appear.
- **Time Complexity:** $O(n)$ – each character is processed at most twice.
- **Space Complexity:** $O(\min(n, 26)) \approx O(1)$, since we store at most 26 letters in the hash set.

Python Implementation

```
def lengthOfLongestSubstring(s: str) -> int:
    char_set = set()
    l, max_length = 0, 0


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

```
return max_length
```

Trade-offs:

- Uses extra space for `char_set`, but ensures **$O(n)$ performance**.
- Works efficiently for all **ASCII character sets** (but may need adjustments for Unicode).

2. Permutation in String

 **Problem:** Given two strings `s1` and `s2`, return `True` if `s2` contains a **permutation** of `s1`.

Brute-Force Approach ($O(n * m!)$)

- Generate all permutations of `s1` and check if they appear in `s2`.
- **Time Complexity:** $O(n * m!)$ – infeasible for large inputs.

Optimized Sliding Window Approach ($O(n)$)

- Instead of generating permutations, we compare **character frequency counts** within a moving window of size `len(s1)`.
- If two frequency maps match, a permutation exists.
- **Time Complexity:** $O(n)$ – single pass through `s2`.
- **Space Complexity:** $O(1)$ – since we store at most **26** characters in frequency maps.

Python Implementation

```
from collections import Counter

def checkInclusion(s1: str, s2: str) -> bool:
    if len(s1) > len(s2):
        return False

    s1_count = Counter(s1)
    window_count = Counter(s2[:len(s1)])
```

```

for i in range(len(s1), len(s2)):
    if window_count == s1_count:
        return True
    window_count[s2[i]] += 1
    window_count[s2[i - len(s1)]] -= 1
    if window_count[s2[i - len(s1)]] == 0:
        del window_count[s2[i - len(s1)]]


return window_count == s1_count

```

Trade-offs:

- Instead of generating **permutations**, we efficiently track **character frequencies**.
- Requires **extra space** for frequency maps, but ensures **O(n) performance**.

3. Minimum Window Substring

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

Brute-Force Approach ($O(n^2 * m)$)

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

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

```

from collections import Counter

```



```

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

    char_count = Counter(t)
    l, r, min_length = 0, 0, float('inf')
    required_chars, current_chars = len(char_count), 0
    window_counts = {}

    result = ""
    while r < len(s):
        char = s[r]
        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 l <= r and current_chars == required_chars:
            if r - l + 1 < min_length:
                min_length = r - l + 1
                result = s[l:r+1]

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

        r += 1

    return result

```

Trade-offs:

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

Key Takeaways

- ✓ **Sliding Window reduces time complexity** in problems involving subarrays or substrings.
- ✓ **Fixed vs. dynamic windows** depend on problem constraints.
- ✓ **Character frequency maps** are useful for substring containment problems.

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

By mastering this pattern, you'll solve many problems efficiently and recognize **when to apply it in 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 ($O(n^2)$)

- Use **two nested loops** to find the pair that sums to `target`.
- **Time Complexity:** $O(n^2)$ – 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

```
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 ($O(n^3)$)

- Try **all triplets** and check if they sum to 0.
- **Time Complexity:** $O(n^3)$ – extremely slow for large inputs.
- **Space Complexity:** $O(n)$ – storing triplets in a result list.

Optimized Sorting + Two Pointers Approach ($O(n^2)$)

- **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:** $O(n^2)$ – 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

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

    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 ($O(n^2)$)

- Try **all pairs** and calculate water capacity.
- **Time Complexity:** $O(n^2)$ – 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

```
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)

```
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)

```
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)

```
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) – $O(n^2)$ Time

- Compare each interval with every other interval to check for overlaps.
- **Time Complexity:** $O(n^2)$ – due to nested comparisons.
- **Space Complexity:** $O(n)$ – for storing merged intervals.

Python Implementation (Brute Force, Inefficient)

```
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)

```
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) – $O(n^2)$ Time

- Compare each meeting with every other meeting to count overlaps.
- **Time Complexity:** $O(n^2)$ – due to nested loops.
- **Space Complexity:** $O(n)$ – storing overlaps.

Python Implementation (Brute Force, Inefficient)

```
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 \leq 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)

```
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 5: 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 ($\text{nums}[i]$ should be at $\text{nums}[\text{nums}[i] - 1]$).
- ◆ **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 $1 \leq \text{nums}[i] \leq n$ (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)

```
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)

```
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)

```
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 ($\text{nums}[i] = i + 1$).
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)

```
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 6: Two Heaps (Min Heap & Max Heap)

Concept & When to Use

The **Two Heaps** pattern is useful for solving problems that require **finding medians, scheduling tasks efficiently, or handling dynamic data** where elements are continuously added or removed.

This technique uses:

- A **Max Heap** to store the smaller half of elements.
 - A **Min Heap** to store the larger half of elements.
 - By maintaining a balance between these heaps, we can efficiently **retrieve median values, process priorities, and optimize scheduling**.
-

When to Use Two Heaps

- ✓ When **finding the median of a dynamic data stream** (e.g., "Find Median in a Stream").
- ✓ When **handling scheduling or priority-based tasks** (e.g., "Task Scheduler").
- ✓ When **continuously inserting and removing elements** while maintaining order.
- ✓ When **retrieving smallest or largest elements efficiently** (better than sorting).

Key Idea

- ◆ Use a **Min Heap** (min-heap) to store the larger half of numbers.
- ◆ Use a **Max Heap** (max-heap) to store the smaller half.
- ◆ Ensure both heaps stay **balanced** (difference in sizes ≤ 1).
- ◆ **Median can be found in $O(1)$** by looking at the top of heaps.

- ◆ **Insertion/removal takes $O(\log n)$** due to heap operations.
-

Grind 75 Problems

The **Two Heaps** pattern is essential for solving these **Grind 75** problems:

1. **Find Median in a Stream** (LeetCode #295)
2. **Task Scheduler** (LeetCode #621)

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

Solutions & Trade-offs

1. 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**.
- **Finding the median:** Take the middle element(s).

Python Implementation (Sorting)

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

    def addNum(self, num: int) -> None:
        self.data.append(num)
        self.data.sort() # Sorting every time (O(n log n))

    def findMedian(self) -> float:
        n = len(self.data)
        if n % 2 == 1:
            return self.data[n // 2] # Odd length -> Middle element
```

```
        else:
            return (self.data[n // 2 - 1] + self.data[n // 2]) / 2 # Average of
two middle elements
```

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

Python Implementation (Two Heaps)

```
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) # Push to Max Heap
        heapq.heappush(self.large, -heapq.heappop(self.small)) # Balance heaps

        if len(self.small) < len(self.large): # Ensure max heap has more elements
            heapq.heappush(self.small, -heapq.heappop(self.large))

    def findMedian(self) -> float:
        if len(self.small) > len(self.large):
            return -self.small[0] # Max Heap root is median
        return (-self.small[0] + self.large[0]) / 2 # Average of two roots
```

🚀 Trade-offs:

- $O(\log n)$ insertion (heap operations) vs. $O(n \log n)$ sorting.
 - $O(1)$ median retrieval vs. $O(n)$ median retrieval in sorting.
 - Uses extra space for heaps ($O(n)$), but avoids frequent sorting.
-

2. 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)

```
from collections import Counter

def leastInterval(tasks: list[str], n: int) -> int:
    freq = list(Counter(tasks).values())
    freq.sort(reverse=True) # Sort by frequency
    max_freq = freq[0] # Most frequent task count
    idle_time = (max_freq - 1) * n # Idle slots needed

    for f in freq[1:]: # Reduce idle time by filling with tasks
        idle_time -= min(max_freq - 1, f)

    idle_time = max(0, idle_time) # Cannot be negative
    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)

```
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()] # Max heap (negate values)
    heapq.heapify(max_heap)
```

```

queue = deque() # Store (frequency, available_time)
time = 0

while max_heap or queue:
    time += 1

    if max_heap:
        count = 1 + heapq.heappop(max_heap) # Process most frequent task
        if count:
            queue.append((count, time + n)) # Add to cooldown queue

    if queue and queue[0][1] == time: # Time to reinsert cooled-down task
        heapq.heappush(max_heap, queue.popleft()[0])

return time


```

Trade-offs:

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

Key Takeaways

- ✅ Two Heaps provide efficient ways to handle median and scheduling problems.
- ✅ Heap operations ($O(\log n)$) are often better than sorting ($O(n \log n)$).
- ✅ Use Max Heap for processing largest elements first (priority scheduling).
- ✅ Use Min Heap for efficiently finding the smallest element (median retrieval).

Mastering **Two Heaps** will help you solve **median-finding, scheduling, 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)

```
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)

```
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 \rightarrow root \rightarrow right).
- Ensure values are strictly increasing.

Python Implementation (DFS)

```
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 8: 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)

```
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 = {}

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

```

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)

```
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)

```

from collections 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 – $O(M \cdot N)$ Time, $O(M \cdot N)$ 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)

```
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

```
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
    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 9: 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:** $O(n * \text{amount})$ where n is the number of coin denominations.
- **Space Complexity:** $O(\text{amount})$ due to the DP array.

Python Implementation (Bottom-up Tabulation)

```
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)

```
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)

```
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
    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 10: 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:

- $a \oplus a = 0$: XORing a number with itself results in 0.
- $a \oplus 0 = a$: 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 ($a \oplus a = 0$), 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


```
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

```
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 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

```
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

```
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

```
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.

