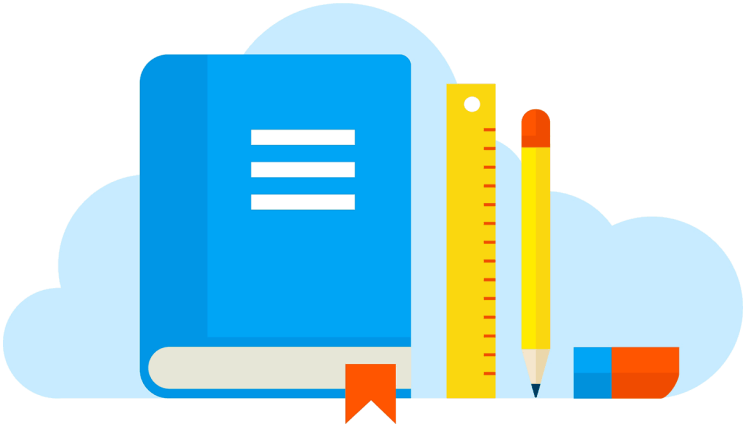

Competitive Coding Prep

Comprehensive Competitive Coding Lesson Plan

Duration: 12-16 Weeks

Target: Preparing for top tech interviews (Amazon, Google, JP Morgan Chase, etc.)

Week 1: Foundations of Problem-Solving & Time Complexity

Week 1: Foundations of Problem-Solving & Time Complexity

Day 1: Introduction to Competitive Programming & Big O Notation

Concepts:

- What is Competitive Programming?
- Understanding problem constraints and optimizing solutions.
- Time & Space Complexity.
- Big O Notation - Best, Average, Worst cases.

- Common Complexity Classes: O(1), O(log N), O(N), O(N log N), O(N^2), O(2^N), O(N!).
- Importance of Dry Running and Debugging.

Algorithm:

- Analyze an example problem with different approaches (Brute Force vs. Optimized Solutions).

Reference Links:

- Big-O Notation Explained
- Time Complexity Cheatsheet

Problems:

1. Leetcode - Time Complexity Analysis
2. Find the time complexity of common functions.

Code Discussion:

- Dry run different solutions and analyze their time complexity.

Day 2: Arrays & Looping Constructs

Concepts:

- Introduction to Arrays.
- Iterating over arrays efficiently.
- Nested loops and their time complexity.
- Common array operations: Insert, Delete, Update, Search.
- Importance of Edge Cases.

Problems:

1. Leetcode - Maximum Subarray
2. Leetcode - Move Zeroes
3. Leetcode - Find Numbers with Even Number of Digits

Code Discussion:

- Debugging common mistakes.
- Optimized array traversal.

Java Code for Move Zeroes:

```java
class Solution {
    public void moveZeroes(int[] nums) {
        int index = 0;
        for (int num : nums) {
            if (num != 0) {
                nums[index++] = num;
            }
        }
        while (index < nums.length) {
            nums[index++] = 0;
        }
    }
}
```

Python Code for Move Zeroes:

```python
def moveZeroes(nums):
    index = 0
    for num in nums:
        if num != 0:
            nums[index] = num
            index += 1
    for i in range(index, len(nums)):
        nums[i] = 0
```

Day 3-4: Two-Pointer Approach & Basic Sorting

Concepts:

- Introduction to Two-Pointer Technique.
- Sorting + Two-Pointer Technique.
- Understanding Bubble Sort, Selection Sort, and Insertion Sort.

Problems:

1. Leetcode - Two Sum
2. Leetcode - Container With Most Water
3. Leetcode - 3Sum

Code Discussion:

- Dry run of each approach.
- Understanding edge cases.

Day 5-6: Sliding Window & Prefix Sum

Concepts:

- Optimizing array traversals using Sliding Window.
- Difference between Fixed & Variable Sliding Window.
- Prefix Sum for range queries.

Problems:

1. Leetcode - Longest Substring Without Repeating Characters
2. Leetcode - Minimum Size Subarray Sum

Code Discussion:

- Debugging common mistakes in window size calculation.

Day 7: Recursion & Binary Search Introduction

Concepts:

- Understanding Recursion.
- Base case and recursive case.
- Binary Search Basics.

Problems:

1. Leetcode - Search in Rotated Sorted Array
2. Leetcode - Find Peak Element

Code Discussion:

- Binary search edge cases.

Week 2: Sorting & Searching Techniques

Day 1-2: Brute Force Sorting (Bubble, Selection, Insertion Sort)

Sorting is a fundamental operation in computer science that helps organize data efficiently. We start with simple sorting algorithms:

1. **Bubble Sort** – Repeatedly swaps adjacent elements if they are in the wrong order.
2. **Selection Sort** – Selects the smallest element and places it in the correct position.
3. **Insertion Sort** – Builds the sorted array one element at a time.

Reference Links

- Bubble Sort Explanation
- Selection Sort
- Insertion Sort

LeetCode Problems

1. Sort an Array
2. **Maximum Gap**

Code Implementations

Java - Bubble Sort

```
void bubbleSort(int arr[]) {
    int n = arr.length;
    for (int i = 0; i < n-1; i++)
        for (int j = 0; j < n-i-1; j++)
            if (arr[j] > arr[j+1]) {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            } }
```

Python - Bubble Sort

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

Day 3-4: Divide & Conquer Sorting (Merge Sort & Quick Sort)

1. **Merge Sort** – Recursively divides the array and merges sorted halves.
2. **Quick Sort** – Selects a pivot and partitions the array around it.

Reference Links

- Merge Sort
- Quick Sort

LeetCode Problems

1. **Sort Colors**
2. **Kth Largest Element in an Array**

Code Implementations

Java - Merge Sort

```
void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = (l + r) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
```

```
        merge(arr, l, m, r);
    }
}
```

Python - Quick Sort

```python
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)
```

Day 5: Advanced Sorting (Counting, Radix, Bucket Sort)

1. **Counting Sort** – Counts occurrences and places elements accordingly.
2. **Radix Sort** – Sorts numbers digit by digit.
3. **Bucket Sort** – Distributes elements into multiple buckets.

Reference Links

- Counting Sort
- Radix Sort
- Bucket Sort

LeetCode Problems

1. Top K Frequent Elements
2. **Sort Integers by The Number of 1 Bits**

Day 6-7: Binary Search & Variations

Binary search is an efficient searching algorithm used in sorted arrays. **Variations** include:

1. **Finding first/last occurrence of an element**
2. **Search in rotated sorted array**
3. **Finding peak element**

Reference Links

- Binary Search Basics
- Binary Search in Rotated Array

LeetCode Problems

1. Binary Search
2. **Find Peak Element**
3. **Search in Rotated Sorted Array**

Code Implementations

Java - Binary Search

```
int binarySearch(int arr[], int x) {
    int l = 0, r = arr.length - 1;
    while (l <= r) {
        int m = l + (r - l) / 2;
        if (arr[m] == x)
            return m;
        if (arr[m] < x)
            l = m + 1;
        else
            r = m - 1;
    }
    return -1;
}
```

Python - Binary Search

```
def binary_search(arr, x):
    l, r = 0, len(arr) - 1
    while l <= r:
        mid = l + (r - l) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] < x:
            l = mid + 1
        else:
            r = mid - 1
    return -1
```

Week 3: Two Pointers & Sliding

Day 1-2: Two Pointers Technique

The **Two Pointers** technique is used for problems involving **sorted arrays, finding pairs, or removing elements efficiently**. It reduces the need for nested loops, improving efficiency to **O(n)**.

Common Use Cases:

1. **Finding pairs with a given sum**
2. **Removing duplicates in sorted arrays**
3. **Merging sorted arrays**

Reference Links

- Two Pointers Technique

LeetCode Problems

1. **Two Sum II – Input Array Is Sorted**
2. **Remove Duplicates from Sorted Array**
3. **Merge Sorted Array**

Code Implementations

Java - Two Sum (Two Pointers Approach)

```
public int[] twoSum(int[] numbers, int target) {
    int left = 0, right = numbers.length - 1;
    while (left < right) {
        int sum = numbers[left] + numbers[right];
        if (sum == target) return new int[]{left + 1, right + 1};
        if (sum < target) left++;
        else right--;
    }
    return new int[]{-1, -1};
}
```

Python - Two Sum (Two Pointers Approach)

```python
def two_sum(numbers, target):
    left, right = 0, len(numbers) - 1
    while left < right:
        curr_sum = numbers[left] + numbers[right]
        if curr_sum == target:
            return [left + 1, right + 1]
        elif curr_sum < target:
            left += 1
        else:
            right -= 1
    return [-1, -1]
```

Day 3-4: Sliding Window Technique

The **Sliding Window** technique is useful for problems involving **subarrays, substrings, and sequences**. It allows maintaining a **fixed or variable-sized window** while efficiently updating computations.

Common Use Cases:

1. **Finding maximum/minimum sum subarray**
2. **Finding longest substring with unique characters**
3. **Finding smallest subarray with a given sum**

Reference Links

- Sliding Window Technique

LeetCode Problems

1. **Maximum Sum Subarray of Size K**
2. **Longest Substring Without Repeating Characters**
3. **Minimum Size Subarray Sum**

Code Implementations

Java - Maximum Sum Subarray (Sliding Window)

```java
public int maxSumSubarray(int[] arr, int k) {
    int maxSum = 0, windowSum = 0;
    for (int i = 0; i < k; i++)
        windowSum += arr[i];
    for (int i = k; i < arr.length; i++) {
        windowSum += arr[i] - arr[i - k];
        maxSum = Math.max(maxSum, windowSum);
    }
    return maxSum;
}
```

Python - Maximum Sum Subarray (Sliding Window)

```python
def max_sum_subarray(arr, k):
    max_sum = 0
    window_sum = sum(arr[:k])

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

    return max_sum
```

Day 5-6: Variable Sliding Window

Unlike fixed-window problems, **variable sliding window** problems dynamically adjust the window size to meet conditions.

Common Use Cases:

1. **Finding longest/shortest substring satisfying a condition**
2. **Finding subarrays meeting a constraint**

3. **Optimizing memory-efficient searching**

Reference Links

- Variable Sliding Window

LeetCode Problems

1. **Longest Substring with At Most K Distinct Characters**
2. **Smallest Subarray with a Given Sum**
3. **Sliding Window Maximum**

Code Implementations

Java - Longest Substring with K Distinct Characters

```
public int longestSubstringKDistinct(String s, int k) {
    int left = 0, maxLength = 0;
    Map<Character, Integer> map = new HashMap<>();

    for (int right = 0; right < s.length(); right++) {
        map.put(s.charAt(right), map.getOrDefault(s.charAt(right), 0)
+ 1);

        while (map.size() > k) {
            map.put(s.charAt(left), map.get(s.charAt(left)) - 1);
            if (map.get(s.charAt(left)) == 0)
map.remove(s.charAt(left));
            left++;
        }

        maxLength = Math.max(maxLength, right - left + 1);
    }
    return maxLength;
}
```

Python - Longest Substring with K Distinct Characters

```python
def longest_substring_k_distinct(s, k):
    left, max_length = 0, 0
    char_map = {}

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

        while len(char_map) > k:
            char_map[s[left]] -= 1
            if char_map[s[left]] == 0:
                del char_map[s[left]]
            left += 1

        max_length = max(max_length, right - left + 1)

    return max_length
```

Week 4: Hashing & Prefix Sum

Day 1-2: Hashing Technique

Hashing is a technique used to store and retrieve data efficiently. It helps solve problems involving **fast lookups, counting elements, and detecting duplicates**.

Common Use Cases:

1. **Checking for duplicates in an array**
2. **Counting frequency of elements**
3. **Finding pairs with a given sum**
4. **Checking for anagrams in strings**

Reference Links

- Hashing Technique

LeetCode Problems

1. **Contains Duplicate**
2. **Two Sum**
3. **Valid Anagram**

Code Implementations

Java - Two Sum Using HashMap

```
public int[] twoSum(int[] nums, int target) {
    Map<Integer, Integer> map = new HashMap<>();
    for (int i = 0; i < nums.length; i++) {
        int complement = target - nums[i];
        if (map.containsKey(complement)) {
            return new int[]{map.get(complement), i};
        }
        map.put(nums[i], i);
    }
    return new int[]{};  }
```

Python - Two Sum Using HashMap

```python
def two_sum(nums, target):
    num_map = {}
    for i, num in enumerate(nums):
        complement = target - num
        if complement in num_map:
            return [num_map[complement], i]
        num_map[num] = i
    return []
```

Day 3-4: Prefix Sum Technique

The **Prefix Sum** technique is useful for problems involving **range queries, subarray sums, and difference arrays**. It allows precomputing cumulative sums to answer queries in constant time.

Common Use Cases:

1. **Finding subarray sum in constant time**
2. **Finding equilibrium index in an array**
3. **Finding the number of subarrays with a given sum**

Reference Links

- Prefix Sum Technique

LeetCode Problems

1. **Range Sum Query - Immutable**
2. **Subarray Sum Equals K**
3. **Equilibrium Index**

Code Implementations

Java - Subarray Sum Equals K (Prefix Sum)

```java
public int subarraySum(int[] nums, int k) {
    Map<Integer, Integer> prefixSum = new HashMap<>();
    prefixSum.put(0, 1);
    int sum = 0, count = 0;
    for (int num : nums) {
        sum += num;
        if (prefixSum.containsKey(sum - k)) {
            count += prefixSum.get(sum - k);
        }
        prefixSum.put(sum, prefixSum.getOrDefault(sum, 0) + 1);
    }
    return count;
}
```

Python - Subarray Sum Equals K (Prefix Sum)

```python
def subarray_sum(nums, k):
    prefix_sum = {0: 1}
    sum_, count = 0, 0
    for num in nums:
        sum_ += num
        count += prefix_sum.get(sum_ - k, 0)
        prefix_sum[sum_] = prefix_sum.get(sum_, 0) + 1
    return count
```

Day 5-6: Advanced Hashing & Prefix Sum

Advanced applications of **Hashing and Prefix Sum** include **rolling hash, rabin-karp algorithm, and difference arrays**. These techniques are useful in pattern searching and range updates.

Common Use Cases:

1. **Checking for repeated substrings using rolling hash**
2. **Pattern matching using Rabin-Karp**
3. **Efficient range updates using Difference Arrays**

Reference Links

- Rolling Hash
- Rabin-Karp Algorithm

LeetCode Problems

1. **Repeated Substring Pattern**
2. **Find All Anagrams in a String**
3. **Difference Array**

Code Implementations

Java - Rabin-Karp Algorithm

```
public List<Integer> rabinKarp(String text, String pattern) {
    int prime = 101, hash = 0, patHash = 0, h = 1;
    int m = pattern.length(), n = text.length();
    List<Integer> result = new ArrayList<>();
    for (int i = 0; i < m - 1; i++) h = (h * 256) % prime;
    for (int i = 0; i < m; i++) {
        patHash = (256 * patHash + pattern.charAt(i)) % prime;
        hash = (256 * hash + text.charAt(i)) % prime;
    }
    for (int i = 0; i <= n - m; i++) {
        if (patHash == hash && text.substring(i, i +
m).equals(pattern))
            result.add(i);
        if (i < n - m) {
            hash = (256 * (hash - text.charAt(i) * h) + text.charAt(i
+ m)) % prime;
            if (hash < 0) hash += prime;
        }      }
    return result;  }
```

Python - Rabin-Karp Algorithm

```
def rabin_karp(text, pattern):
    prime = 101
    hash_, pat_hash, h = 0, 0, 1
    m, n = len(pattern), len(text)
    result = []
    for i in range(m - 1): h = (h * 256) % prime
    for i in range(m):
        pat_hash = (256 * pat_hash + ord(pattern[i])) % prime
        hash_ = (256 * hash_ + ord(text[i])) % prime
    for i in range(n - m + 1):
        if pat_hash == hash_ and text[i:i + m] == pattern:
result.append(i)
        if i < n - m:
            hash_ = (256 * (hash_ - ord(text[i]) * h) + ord(text[i +
m])) % prime
            if hash_ < 0: hash_ += prime
    return result
```

Week 5: Stacks & Queues

Day 1-2: Introduction to Stacks

A **stack** is a linear data structure that follows the **Last In, First Out (LIFO)** principle. Operations include **push, pop, peek, and isEmpty**.

Common Use Cases:

1. **Balanced Parentheses Checking**
2. **Undo/Redo Operations**
3. **Backtracking Algorithms**
4. **Evaluating Expressions (Postfix, Prefix)**

Reference Links

- Stack Data Structure

LeetCode Problems

1. **Valid Parentheses**
2. **Min Stack**
3. **Evaluate Reverse Polish Notation**

Code Implementations

Java - Valid Parentheses Using Stack

```
public boolean isValid(String s) {
    Stack<Character> stack = new Stack<>();
    for (char c : s.toCharArray()) {
        if (c == '(' || c == '{' || c == '[') {
            stack.push(c);
        } else {
            if (stack.isEmpty()) return false;
            char top = stack.pop();
            if ((c == ')' && top != '(') || (c == '}' && top != '{')
|| (c == ']' && top != '[')) {
```

```
                return false;
            }
        }
    }
    return stack.isEmpty();
}
```

Python - Valid Parentheses Using Stack

```
def is_valid(s):
    stack = []
    mapping = {')': '(', '}': '{', ']': '['}
    for char in s:
        if char in mapping:
            top_element = stack.pop() if stack else '#'
            if mapping[char] != top_element:
                return False
        else:
            stack.append(char)
    return not stack
```

Day 3-4: Queues & Deques

A **queue** follows the **First In, First Out (FIFO)** principle. A **deque (double-ended queue)** allows insertion and deletion from both ends.

Common Use Cases:

1. **Scheduling Tasks (CPU Scheduling, Print Queue)**
2. **Handling Requests (Web Servers, Call Centers)**
3. **Sliding Window Problems**

Reference Links

- Queue Data Structure

LeetCode Problems

1. Implement Queue using Stacks
2. Sliding Window Maximum
3. Design Circular Queue

Code Implementations

Java - Implement Queue Using Stacks

```
class MyQueue {
    private Stack<Integer> input = new Stack<>();
    private Stack<Integer> output = new Stack<>();
    public void push(int x) {
        input.push(x);
    }
    public int pop() {
        if (output.isEmpty()) {
            while (!input.isEmpty()) {
                output.push(input.pop());
            }
        }
        return output.pop();
    }
    public int peek() {
        if (output.isEmpty()) {
            while (!input.isEmpty()) {
                output.push(input.pop());
            }
        }
        return output.peek();
    }
    public boolean empty() {
        return input.isEmpty() && output.isEmpty();
    }
}
```

Python - Implement Queue Using Stacks

```
class MyQueue:
    def __init__(self):
        self.input = []
        self.output = []
    def push(self, x):
        self.input.append(x)
    def pop(self):
        if not self.output:
            while self.input:
                self.output.append(self.input.pop())
        return self.output.pop()
    def peek(self):
        if not self.output:
            while self.input:
                self.output.append(self.input.pop())
        return self.output[-1]
    def empty(self):
        return not self.input and not self.output
```

Day 5-6: Advanced Stack & Queue Application

Advanced applications involve **monotonic stacks, min/max stacks, and deque-based optimizations**.

Common Use Cases:

1. **Finding Next Greater Element using Monotonic Stack**
2. **Finding Maximum Element in a Sliding Window using Deque**
3. **Implementing LRU Cache using Deque & HashMap**

Reference Links

- Monotonic Stack
- LRU Cache

LeetCode Problems

1. Next Greater Element
2. Sliding Window Maximum
3. LRU Cache

Code Implementations

Java - Next Greater Element Using Stack

```java
public int[] nextGreaterElement(int[] nums) {
    Stack<Integer> stack = new Stack<>();
    Map<Integer, Integer> map = new HashMap<>();
    int[] res = new int[nums.length];
    for (int num : nums) {
        while (!stack.isEmpty() && stack.peek() < num) {
            map.put(stack.pop(), num);
        }
        stack.push(num);
    }
    for (int i = 0; i < nums.length; i++) {
        res[i] = map.getOrDefault(nums[i], -1);
    }
    return res;
}
```

Python - Next Greater Element Using Stack

```python
def next_greater_element(nums):
    stack, res = [], {}
    for num in nums:
        while stack and stack[-1] < num:
            res[stack.pop()] = num
        stack.append(num)
    return [res.get(num, -1) for num in nums]
```

Week 6: Linked Lists

Day 1-2: Introduction to Linked Lists

A **linked list** is a linear data structure where elements are stored in **nodes**, and each node contains a reference (or pointer) to the next node.

Types of Linked Lists:

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

Common Operations:

1. **Insertion (at beginning, end, middle)**
2. **Deletion (by value, position)**
3. **Traversal**
4. **Searching for an element**

Reference Links

- Linked List Data Structure

LeetCode Problems

1. **Reverse Linked List**
2. **Merge Two Sorted Lists**
3. **Linked List Cycle**

Code Implementations

Java - Singly Linked List Implementation

```
class ListNode {
    int val;
    ListNode next;
    ListNode(int x) { val = x; }
```

```
}

class LinkedList {
    ListNode head;
    public void insertAtEnd(int val) {
        if (head == null) {
            head = new ListNode(val);
            return;
        }
        ListNode temp = head;
        while (temp.next != null) temp = temp.next;
        temp.next = new ListNode(val);
    }
}
```

Day 3-4: Advanced Linked List Operations

- **Reversing a Linked List**
- **Detecting and Removing a Cycle (Floyd's Cycle Detection Algorithm)**
- **Finding Middle Element (Tortoise and Hare Algorithm)**

LeetCode Problems

1. Reverse Linked List
2. Remove Nth Node From End
3. Detect Cycle in Linked List

Code Implementations

Java - Reverse a Linked List

```
public ListNode reverseList(ListNode head) {
    ListNode prev = null, curr = head, next = null;
    while (curr != null) {
        next = curr.next;
        curr.next = prev;
        prev = curr;
        curr = next;  }
    return prev;  }
```

Day 5-6: Doubly & Circular Linked Lists

- **Doubly Linked List (DLL) Implementation**
- **Circular Linked List (CLL) Implementation**

LeetCode Problems

1. [Flatten a Multilevel Doubly Linked List]
2. [Insertion in Sorted Circular Linked List]

Code Implementations

Java - Doubly Linked List Implementation

```
class DoublyListNode {
    int val;
    DoublyListNode next, prev;
    DoublyListNode(int x) { val = x; }
}

class DoublyLinkedList {
    DoublyListNode head;
    public void insertAtEnd(int val) {
        if (head == null) {
            head = new DoublyListNode(val);
            return;
        }
        DoublyListNode temp = head;
        while (temp.next != null) temp = temp.next;
        DoublyListNode newNode = new DoublyListNode(val);
        temp.next = newNode;
        newNode.prev = temp;
    }
}
```

Week 7: Recursion & Backtracking

Day 1-2: Introduction to Recursion

Recursion is a technique in which a function calls itself to solve smaller subproblems. It is used in problems that can be broken down into similar subproblems.

Types of Recursion:

1. **Direct Recursion** - A function calls itself directly.
2. **Indirect Recursion** - A function calls another function, which in turn calls the original function.

Common Recursive Problems:

1. **Factorial of a number**
2. **Fibonacci sequence**
3. **Sum of N natural numbers**

LeetCode Problems

1. Factorial using Recursion
2. Fibonacci Number
3. Power of Two

Code Implementations

Java - Factorial using Recursion

```
public int factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);
}
```

python - Factorial using Recursion

```
def factorial(n):
    if n == 0:
```

```
        return 1
    return n * factorial(n - 1)
```

Day 3-4: Recursion on Arrays & Strings

- Recursively reversing an array
- Checking if a string is a palindrome using recursion
- Recursively finding the maximum element in an array

LeetCode Problems

1. Reverse String
2. Palindrome Check
3. Max Element in an Array

Code Implementations

Java - Reverse a String using Recursion

```java
public String reverseString(String s) {
    if (s.length() <= 1) return s;
    return reverseString(s.substring(1)) + s.charAt(0);
}
```

Python - Reverse a String using Recursion

```python
def reverse_string(s):
    if len(s) <= 1:
        return s
    return reverse_string(s[1:]) + s[0]
```

Day 5-6: Introduction to Backtracking

Backtracking is an algorithmic technique that explores all possible solutions to a problem and backtracks when it encounters a dead-end. It is useful in problems like permutations, combinations, and solving puzzles.

Key Problems:

1. **Generating All Subsets of an Array**
2. **Solving N-Queens Problem**
3. **Finding All Permutations of a String**

LeetCode Problems

1. Subsets
2. N-Queens
3. Permutations

Code Implementations

Java - Generating Subsets Using Backtracking

```
public void generateSubsets(int[] nums, int index, List<Integer>
curr, List<List<Integer>> result) {
    if (index == nums.length) {
        result.add(new ArrayList<>(curr));
        return;
    }
    curr.add(nums[index]);
    generateSubsets(nums, index + 1, curr, result);
    curr.remove(curr.size() - 1);
    generateSubsets(nums, index + 1, curr, result);
}
```

Python - Generating Subsets Using Backtracking

```
def generate_subsets(nums, index, curr, result):
    if index == len(nums):
        result.append(curr[:])
        return
    curr.append(nums[index])
    generate_subsets(nums, index + 1, curr, result)
    curr.pop()
    generate_subsets(nums, index + 1, curr, result)
```

Week 8: Dynamic Programming

Day 1-2: Introduction to Dynamic Programming

Dynamic Programming (DP) is an optimization technique used to solve complex problems by breaking them down into overlapping subproblems and storing results to avoid redundant computations.

Types of Dynamic Programming:

1. **Top-Down Approach (Memoization)** - Solve recursively and store results.
2. **Bottom-Up Approach (Tabulation)** - Solve iteratively and use previously computed results.

Key Problems:

1. **Fibonacci Sequence using DP**
2. **Climbing Stairs Problem**
3. **Coin Change Problem**

LeetCode Problems

1. Fibonacci Number
2. Climbing Stairs
3. Coin Change

Code Implementations

Java - Fibonacci Using Memoization

```
public class FibonacciDP {
    private HashMap<Integer, Integer> memo = new HashMap<>();
    public int fib(int n) {
        if (n <= 1) return n;
        if (memo.containsKey(n)) return memo.get(n);
        int result = fib(n - 1) + fib(n - 2);
        memo.put(n, result);
        return result;    }  }
```

Python - Fibonacci Using Memoization

```
def fib(n, memo={}):
    if n <= 1:
        return n
    if n in memo:
        return memo[n]
    memo[n] = fib(n - 1, memo) + fib(n - 2, memo)
    return memo[n]
```

Day 3-4: DP on 1D Arrays

- Solving problems where solutions build upon previous results in an array.
- Common examples include house robber problems and maximum sum subarrays.

Key Problems:

1. **Maximum Subarray Sum (Kadane's Algorithm)**
2. **House Robber Problem**

LeetCode Problems

1. Maximum Subarray
2. House Robber

Code Implementations

Java - Maximum Subarray (Kadane's Algorithm)

```
public int maxSubArray(int[] nums) {
    int maxSum = nums[0], currentSum = nums[0];
    for (int i = 1; i < nums.length; i++) {
        currentSum = Math.max(nums[i], currentSum + nums[i]);
        maxSum = Math.max(maxSum, currentSum);
    }
    return maxSum;  }
```

Python - Maximum Subarray (Kadane's Algorithm)

```
def max_subarray(nums):
    max_sum = nums[0]
    current_sum = nums[0]
    for num in nums[1:]:
        current_sum = max(num, current_sum + num)
        max_sum = max(max_sum, current_sum)
    return max_sum
```

Day 5-6: DP on 2D Arrays (Grid Problems)

- Used in problems that involve moving in a 2D grid, like pathfinding or game theory problems.
- Key approach involves storing intermediate results in a 2D table.

Key Problems:

1. **Unique Paths in a Grid**
2. **Minimum Path Sum**

LeetCode Problems

1. Unique Paths
2. Minimum Path Sum

Code Implementations

Java - Unique Paths in a Grid

```
public int uniquePaths(int m, int n) {
    int[][] dp = new int[m][n];
    for (int i = 0; i < m; i++) dp[i][0] = 1;
    for (int j = 0; j < n; j++) dp[0][j] = 1;
    for (int i = 1; i < m; i++) {
```

```
        for (int j = 1; j < n; j++) {
            dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
        }
    }
    return dp[m - 1][n - 1];
}
```

Python - Unique Paths in a Grid

```
def unique_paths(m, n):
    dp = [[1] * n for _ in range(m)]
    for i in range(1, m):
        for j in range(1, n):
            dp[i][j] = dp[i - 1][j] + dp[i][j - 1]
    return dp[m - 1][n - 1]
```

Week 9: Graph Algorithms

Day 1-2: Introduction to Graphs

Graphs are a data structure consisting of nodes (vertices) and edges connecting them. They are used to represent networks, relationships, and dependencies.

Types of Graphs:

1. **Directed vs. Undirected Graphs**
2. **Weighted vs. Unweighted Graphs**
3. **Adjacency Matrix vs. Adjacency List Representation**

Key Problems:

1. **Graph Representation & Traversal**
2. **DFS and BFS Traversal**

LeetCode Problems

1. Graph Representation
2. BFS and DFS Traversal

Code Implementations

Java - Graph Representation using Adjacency List

```
import java.util.*;
class Graph {
    private Map<Integer, List<Integer>> adjList = new HashMap<>();
    public void addEdge(int u, int v) {
        adjList.putIfAbsent(u, new ArrayList<>());
        adjList.putIfAbsent(v, new ArrayList<>());
        adjList.get(u).add(v);
        adjList.get(v).add(u);
    }
}
```

Python - Graph Representation using Adjacency List

```
from collections import defaultdict
class Graph:
    def __init__(self):
        self.adj_list = defaultdict(list)
    def add_edge(self, u, v):
        self.adj_list[u].append(v)
        self.adj_list[v].append(u)
```

Day 3-4: Graph Traversal - BFS & DFS

- **Breadth-First Search (BFS):** Explores neighbors first, uses a queue.
- **Depth-First Search (DFS):** Explores deeper first, uses recursion or a stack.

Key Problems:

1. **Connected Components**
2. **Detecting Cycles in Graphs**

LeetCode Problems

1. Number of Connected Components
2. Detecting Cycles

Code Implementations

Java - BFS Implementation

```
public void bfs(int start, Map<Integer, List<Integer>> graph) {
    Queue<Integer> queue = new LinkedList<>();
    Set<Integer> visited = new HashSet<>();
    queue.add(start);
    visited.add(start);
    while (!queue.isEmpty()) {
        int node = queue.poll();
        System.out.print(node + " ");
        for (int neighbor : graph.get(node)) {
```

```
            if (!visited.contains(neighbor)) {
                queue.add(neighbor);
                visited.add(neighbor);
            }
        }
    }
}
```

Python - BFS Implementation

```
from collections import deque
def bfs(start, graph):
    queue = deque([start])
    visited = set([start])
    while queue:
        node = queue.popleft()
        print(node, end=" ")
        for neighbor in graph[node]:
            if neighbor not in visited:
                queue.append(neighbor)
                visited.add(neighbor)
```

Day 5-6: Shortest Path Algorithms

- **Dijkstra's Algorithm:** Finds shortest paths in weighted graphs.
- **Floyd-Warshall Algorithm:** Solves all-pairs shortest path problem.

Key Problems:

1. **Dijkstra's Algorithm Implementation**
2. **Minimum Cost Path in a Weighted Graph**

LeetCode Problems

1. [Network Delay Time]

2. [Cheapest Flights Within K Stops]

Code Implementations

Java - Dijkstra's Algorithm

```
import java.util.*;
class Graph {
    public int dijkstra(int n, int[][] edges, int start) {
        Map<Integer, List<int[]>> graph = new HashMap<>();
        for (int[] edge : edges) {
            graph.putIfAbsent(edge[0], new ArrayList<>());
            graph.get(edge[0]).add(new int[]{edge[1], edge[2]});
        }
        PriorityQueue<int[]> pq = new
PriorityQueue<>(Comparator.comparingInt(a -> a[1]));
        pq.add(new int[]{start, 0});
        Map<Integer, Integer> distances = new HashMap<>();
        while (!pq.isEmpty()) {
            int[] current = pq.poll();
            int node = current[0], cost = current[1];
            if (distances.containsKey(node)) continue;
            distances.put(node, cost);
            for (int[] neighbor : graph.getOrDefault(node, new
ArrayList<>())) {
                if (!distances.containsKey(neighbor[0])) {
                    pq.add(new int[]{neighbor[0], cost +
neighbor[1]});
                }
            }
        }
        return distances.size() == n ?
Collections.max(distances.values()) : -1;
    }
}
```

Week 10: Trees and Binary Search Trees (BSTs)

Day 1: Introduction to Trees

- **Tree Basics**: Nodes, Edges, Root, Parent, Child, Leaf
- **Types of Trees**: General Tree, Binary Tree, Binary Search Tree (BST), AVL Tree
- **Tree Traversal Methods**:
    - **Depth-First Search (DFS):** Preorder, Inorder, Postorder
    - **Breadth-First Search (BFS):** Level Order Traversal

Key Problems:

1. **Preorder, Inorder, and Postorder Traversals**
2. **Level Order Traversal**

Java - Preorder Traversal

```
class TreeNode {
    int val;
    TreeNode left, right;
    TreeNode(int val) { this.val = val; }
}
class Solution {
    public void preorder(TreeNode root) {
        if (root == null) return;
        System.out.print(root.val + " ");
        preorder(root.left);
        preorder(root.right);
    }
}
```

Day 2-3: Binary Search Trees (BSTs)

- **BST Properties**: Left subtree < Root < Right Subtree

- **Operations on BST**:
  - Insert
  - Delete
  - Search
  - Finding Min/Max

Key Problems:

1. **Search in a BST**
2. **Insert into a BST**
3. **Delete a Node in BST**

Java - Search in BST

```
class Solution {
    public TreeNode searchBST(TreeNode root, int val) {
        if (root == null || root.val == val) return root;
        return val < root.val ? searchBST(root.left, val) :
searchBST(root.right, val);
    }
}
```

Day 4: Advanced Trees - AVL Trees & Segment Trees

- **AVL Tree**: Self-balancing BST (Rotations: Left, Right, Left-Right, Right-Left)
- **Segment Tree**: Used for Range Queries (Sum, Min, Max)

Key Problems:

1. **Check if a BST is Balanced**
2. **Range Sum Query using Segment Tree**

Java - Check if a BST is Balanced (AVL Property)

```
class Solution {
    public boolean isBalanced(TreeNode root) {
        return height(root) != -1;
    }
    private int height(TreeNode node) {
        if (node == null) return 0;
```

```
        int left = height(node.left);
        int right = height(node.right);
        if (left == -1 || right == -1 || Math.abs(left - right) > 1)
return -1;
        return Math.max(left, right) + 1;
    }
}
```

Day 5: Practice & Problem Solving

LeetCode Problems

1. Validate Binary Search Tree
2. Lowest Common Ancestor of a BST
3. Kth Smallest Element in a BST
4. Balanced Binary Tree

Week 11: Advanced Data Structures & Graph Algorithms

- **Day 1-2: Tries (Prefix Trees) & Applications**
  - **Concepts:** Dictionary, Autocomplete, Longest Common Prefix
  - **LeetCode Problems:**
    1. Implement Trie (Prefix Tree)
    2. Longest Word in Dictionary
- **Day 3-4: Segment Trees, Fenwick Trees & Disjoint Set Union (Union-Find)**
  - **Concepts:** Range Queries (Sum, Min, Max), Lazy Propagation
  - **LeetCode Problems:**
    1. Range Sum Query - Mutable
    2. Accounts Merge
- **Day 5-6: Advanced Graph Algorithms**
  - **Concepts:**
    1. **Shortest Path:** Dijkstra's & Bellman-Ford
    2. **Connectivity:** Bridges & Articulation Points (Tarjan's Algorithm)
  - **LeetCode Problems:**
    1. Network Delay Time
    2. Critical Connections in a Network
- **Day 7: Minimum Spanning Tree (Kruskal's & Prim's)**
  - **LeetCode Problem:** Connecting Cities With Minimum Cost

---

Week 12: Math, Bit Manipulation & Miscellaneous Topics

- **Day 1-2: Bit Manipulation & Combinatorics**
  - **Concepts:** XOR tricks, Subsets, Bitmask DP
  - **LeetCode Problems:**
    1. Single Number
    2. Letter Tile Possibilities
- **Day 3-4: Modular Arithmetic, Fermat's Theorem & Probability**
  - **Concepts:** NCr Mod P, Modular Inverse, Fast Exponentiation
  - **LeetCode Problems:**
    1. Unique Paths
    2. Sum of Two Integers
- **Day 5-6: Game Theory (Nim Game, Grundy Numbers, Sprague-Grundy Theorem)**
  - **LeetCode Problems:**
    1. Nim Game
    2. Stone Game
- **Day 7: Geometry & Computational Geometry**
  - **Concepts:** Convex Hull (Graham's Scan), Line Intersections
  - **LeetCode Problem:** Erect the Fence