CS Fundamentals

## Week 1-2: Programming Fundamentals (PF)

## 1.1 Variables, Data Types

### Discussion:`

* + Introduction to different data types (integer, float, string, boolean) and their usage.
  + Explanation of variables and how they store data.

### Tasks:

* + Write a program to convert a string to an integer and vice versa.

## 1.2 Control Structures (If-else, Loops)

### Discussion:

* + Overview of conditional statements (if, else-if, else) and their importance in decision making.
  + Introduction to loops (for, while) for repeating tasks.

### Tasks:

Write a program that prints the numbers from 1 to 100.

However, for multiples of three, print "Fizz" instead of the number, and for the multiples of five, print "Buzz".

For numbers which are multiples of both three and five, print "FizzBuzz".

* + Write a loop that sums all numbers from 1 to 100.

## 1.3 Introduction to Functions and Recursion

### Discussion:

* + Basics of functions, including parameters, return types, and scope.
  + Introduction to the concept of recursion and its use cases.

### Tasks:

* + Implement a function to calculate factorial using recursion.
  + Create a function to check if a word is a palindrome.

## 1.4 Advanced Functions, Pointers, Memory Management

### Discussion:

* + Deep dive into advanced function topics like callbacks and closures.
  + Introduction to pointers and memory management in programming.

### Tasks:

* + Write a program using pointers to swap the values of two variables.
  + Develop a memory-efficient program for managing a dynamic list of integers.

## Week 3-4: Object-Oriented Programming (OOP)

## 2.1 Classes and Objects, Encapsulation

### Discussion:

### Understanding classes and objects, the building blocks of OOP.

### Detailed look at encapsulation and access modifiers.

### Tasks:

### Create a class with private attributes and public methods to access them.

### Develop a program that uses getter and setter methods for data encapsulation.

## 2.2 Inheritance and its Types

### Discussion:

### Explaining different types of inheritance (single, multiple, multilevel, hierarchical, hybrid).

### Discussing the "Diamond Problem" in multiple inheritance.

### Tasks:

### Create a multilevel inheritance structure and demonstrate the use of super() function.

### Analyze a scenario where multiple inheritance can lead to the Diamond Problem and propose a solution.

## 2.3 Polymorphism (Overloading and Overriding), Abstraction

### Discussion:

### Differentiating between overloading and overriding.

### Exploring abstract classes

### Tasks:

### Implement method overloading and overriding in a class.

## 2.4 Design Patterns

### Discussion:

### Introduction to common design patterns (Singleton, Factory, Observer).

### Tasks:

### Implement a Singleton pattern in a small project.

## Week 5-6-7: Data Structures and Algorithms (DSA)

## Module 1 – Arrays

## 1.1 Discussion: Arrays

### Array as a Collection of Items:

### Arrays store multiple items under a single variable name, organized in contiguous memory locations. Each item in an array is called an element, and they are accessed using their index.

### Contiguous Memory Allocation:

### This means that array elements are stored in consecutive memory addresses, which allows for quick access times.

### Advantages of Using Arrays:

### Indexing for Random Access: Elements can be accessed directly using their index in constant time.

### Ease of Iteration: Due to their contiguous nature, iterating over array elements is straightforward and efficient.

### Efficient Memory Access: Being stored in contiguous memory spaces allows for efficient data retrieval.

### Disadvantages of Using Arrays:

### Fixed Size: Traditional arrays have a fixed size, meaning the number of elements they can store is determined at the time of their creation.

### Potential Memory Wastage: If the allocated array size is larger than the actual number of elements needed, it results in wasted memory space.

### Cost of Insertion and Deletion: Inserting or deleting an element in an array involves shifting elements, which can be costly in terms of processing.

## 1.2 Tasks:

### Task: Find Two Numbers That Add Up to a Target

* + Description: Given an array nums and a target value, return the indices of the two numbers that add up to the target.
  + Input: nums = [0,-1,2,-3,1], target = -2
  + Output: [3,4]
  + Test Case: assert findTwoSum([0,-1,2,-3,1], -2) == [3,4]

### Task: Indices of Two Numbers for Maximum Sum

* + Description: Modify the above solution to return the indices of two numbers that add up to the maximum possible number.
  + Input: nums = [0,-1,2,-3,1]
  + Output: [2,4]
  + Test Case: assert findMaxSumIndices([0,-1,2,-3,1]) == [2,4]

### Task: Largest Subarray Sum

* + Description: Find a subarray with the largest sum and return its sum as well as the subarray.
  + Input: nums = [5,4,-1,7,8]
  + Output: 23, subarray = [5,4,-1,7,8]
  + Test Case: assert largestSubarraySum([5,4,-1,7,8]) == (23, [5,4,-1,7,8])

### Task: Union and Intersection of Two Sorted Arrays

* + Description: Find the union and intersection of two sorted arrays.
  + Input: arr1 = [1,3,4,5,7], arr2 = [2,3,5,6]
  + Output: Union: [1,2,3,4,5,6,7], Intersection: [3,5]
  + Test Case: assert unionIntersection([1,3,4,5,7], [2,3,5,6]) == ([1,2,3,4,5,6,7], [3,5])

### Task: Rotate Array

* + Description: Rotate the array to the right by k steps.
  + Input: nums = [1,2,3,4,5,6,7], k = 3
  + Output: [5,6,7,1,2,3,4]
  + Test Case: assert rotateArray([1,2,3,4,5,6,7], 3) == [5,6,7,1,2,3,4]

### Task: Rotate 2D Matrix

* + Description: Rotate an n x n 2D matrix by 90 degrees (clockwise) in-place.
  + Input: matrix = [[1,2,3], [4,5,6], [7,8,9]]
  + Output: [[7,4,1], [8,5,2], [9,6,3]]
  + Test Case: assert rotateMatrix([[1,2,3], [4,5,6], [7,8,9]]) == [[7,4,1], [8,5,2], [9,6,3]]

### Task: Array Element Insertion and Deletion

* + Description: Write functions for insertion and deletion of an element in an array.
  + Input: Insert 10 at index 2, Delete element at index 3.
  + Output: Updated array after each operation.
  + Test Case: assert insertDeleteElement([1,2,3,4,5], 10, 2, 3) == [1,2,10,4]

## Module 2 – Linked List

### 2.1 Discussion: Linked List

### Overview of Linked List:

* A linked list is a linear data structure where elements are not stored in contiguous memory locations. Each element (node) contains the data and a reference (link) to the next node.

### Implementation:

* Consists of nodes with two parts: data and a pointer to the next node. The first node is called the head, and the last node points to null.

### Functions and Complexity:

* + Insertion and Deletion: These operations can be more efficient than arrays as they don't require shifting elements. Complexity is generally O(1) if the position is known.
  + Traversal: To access an element, one needs to traverse from the head node. The complexity is O(n) in the worst case.

### Memory Allocation and De-allocation:

* Unlike arrays, memory is allocated dynamically for each node. This can lead to fragmented memory but allows for efficient memory utilization.

### Types of Linked List:

* + Singly Linked List: Each node points to the next node and the last node points to null.
  + Doubly Linked List: Nodes contain two links, one to the next node and another to the previous node.
  + Circular Linked List: Last node points back to the first node, forming a circle.

## 2.2 Tasks:

### Task: Merge Two Sorted Linked Lists

* + Description: Merge two sorted linked lists into a single sorted linked list.
  + Input: L1 = 1 -> 3 -> 9, L2 = 2 -> 5 -> 6 -> 10
  + Output: Merged List: 1 -> 2 -> 3 -> 5 -> 6 -> 9 -> 10
  + Test Case: assert mergeLists([1,3,9], [2,5,6,10]) == [1,2,3,5,6,9,10]

### Task: Delete N Nodes After M Nodes

* + Description: From a linked list, delete 'N' nodes after skipping 'M' nodes of a linked list until the end.
  + Input: M = 2, N = 3, L1 = 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7
  + Output: 1 -> 2 -> 6 -> 7
  + Test Case: assert deleteNodes([1,2,3,4,5,6,7], 2, 3) == [1,2,6,7]

### Task: Detect and Remove Cycle in Linked List

* + Description: Identify if a cycle exists in a linked list and remove it.
  + Input: A linked list where a cycle exists.
  + Output: The same list but without any cycle.
  + Test Case: assert detectRemoveCycle(linkedListWithCycle) == modifiedLinkedList

### Task: Reverse a Linked List

* + Description: Reverse the elements of a linked list.
  + Input: L1 = 1 -> 2 -> 3 -> 4 -> NULL
  + Output: L1 = 4 -> 3 -> 2 -> 1 -> NULL
  + Test Case: assert reverseLinkedList([1,2,3,4]) == [4,3,2,1]

### Task: Delete Nth Last Node of a Singly Linked List

* + Description: Remove the Nth node from the end of the list.
  + Input: L1 = 1 -> 2 -> 3 -> 4 -> 5, N = 2
  + Output: L1 = 1 -> 2 -> 3 -> 5
  + Test Case: assert deleteNthLastNode([1,2,3,4,5], 2) == [1,2,3,5]

### Task: Find Middle of Linked List

* + Description: Find the middle node of a linked list.
  + Input: L1 = 1 -> 2 -> 3 -> 4 -> 5
  + Output: Middle Node: 3
  + Test Case: assert findMiddleNode([1,2,3,4,5]) == 3

### Task: Implement a Doubly Linked List

* + Description: Create a doubly linked list with basic operations (insert, delete).
  + Input: Operations like insert 4 at the beginning, delete last node.
  + Output: Updated doubly linked list.
  + Test Case: assert doublyLinkedListOperations([1,2,3], 4, 'deleteLast') == [4,1,2]

### Task: Circular Linked List Traversal

* + Description: Traverse a circular linked list and print elements.
  + Input: Circular Linked List 1 -> 2 -> 3 -> 1 (circular)
  + Output: 1, 2, 3
  + Test Case: assert traverseCircularLinkedList([1,2,3]) == [1,2,3]

### Task: Sorted Insert for Circular Linked List

* + Description: Insert a new node into a sorted circular linked list.
  + Input: Circular Linked List 1 -> 2 -> 4 -> 1, Insert 3
  + Output: Updated List: 1 -> 2 -> 3 -> 4 -> 1
  + Test Case: assert sortedInsertCircularLinkedList([1,2,4], 3) == [1,2,3,4]

### Task: Check if a Linked List is Palindrome

* + Description: Determine if the elements of a linked list form a palindrome.
  + Input: L1 = 1 -> 2 -> 2 -> 1
  + Output: True
  + Test Case: assert isLinkedListPalindrome([1,2,2,1]) == Tru

## Module 3 – Stacks and Queues

### 3.1 Discussion: Stacks and Queues

### Stack as a LIFO Data Structure:

* + Concept of Last-In-First-Out (LIFO).
  + Common operations: PUSH (add), POP (remove), and TOP (peek).

### Use Cases for Stacks:

* + Implementing function calls (call stack).
  + Undo operations in software.
  + Solving certain algorithmic problems like parsing expressions.

### Queue as a FIFO Data Structure:

* + Concept of First-In-First-Out (FIFO).
  + Operations: ENQUEUE (add), DEQUEUE (remove), Front, and Rear pointers.

### Use Cases for Queues:

* + Handling tasks in sequential order, such as printing documents.
  + CPU task scheduling.
  + Buffer for data transfer in networking.

### Types of Queue:

* + Linear Queue: Simple FIFO implementation.
  + Circular Queue: Utilizes the queue space efficiently.
  + Priority Queue: Elements are processed based on priority.
  + Double-Ended Queue (Deque): Allows insertion and deletion at both ends.

## 3.2 Tasks:

### Task: Implement a Stack

* + Description: Create a stack with basic operations: PUSH, POP, and TOP.
  + Test Case: PUSH 3, PUSH 5, POP, TOP should return 3.

### Task: Reverse a String without Recursion

* + Description: Use a stack to reverse a given string.
  + Input: "hello"
  + Output: "olleh"
  + Test Case: assert reverseString("hello") == "olleh"

### Task: Valid Parentheses

* + Description: Check if a string of parentheses is valid (balanced).
  + Input: "()[]{}"
  + Output: True
  + Test Case: assert isValidParentheses("()[]{}") == True

### Task: Implement a Circular Queue using Linked List

* + Description: Create a circular queue with ENQUEUE and DEQUEUE operations.
  + Test Case: ENQUEUE 4, ENQUEUE 7, DEQUEUE should return 4.

### Task: Implement a Priority Queue

* + Description: Create a priority queue and demonstrate element processing based on priority.
  + Test Case: Insert items with different priorities and ensure correct processing order.

### Task: Queue using Two Stacks

* + Description: Implement a queue using two stacks.
  + Test Case: ENQUEUE 1, 2, 3 and DEQUEUE should return 1.

### Task: Balancing Symbols using Stack

* + Description: Check if symbols in a string are balanced using a stack.
  + Input: "{[()]}"
  + Output: True
  + Test Case: assert isBalanced("{[()]}") == True

### Task: Implement a Deque (Double-Ended Queue)

* + Description: Create a deque with insertion and deletion from both ends.
  + Test Case: Insert 5 at front, 10 at rear, then delete from front should return 5.

### Task: Sort a Queue without Extra Space

* + Description: Sort the elements of a queue using no additional space.
  + Input: Queue [5, 1, 3, 2, 8]
  + Output: Sorted Queue [1, 2, 3, 5, 8]
  + Test Case: assert sortQueue([5,1,3,2,8]) == [1,2,3,5,8]

## Module 4 – Dictionary and Hashmaps

### 4.1 Discussion: Dictionary and Hashmaps

#### Difference Between Dictionary and Hashmaps:

* + Dictionaries are abstract data types, while hashmaps are a concrete implementation.
  + Dictionaries allow for a broader concept of key-value pair storage, while hashmaps specifically use a hashing function for storage and retrieval.

#### Hashing Functions:

* + A hashing function maps keys to specific locations in the hashmap.
  + Example function: h(K) = K % M, where K is the key and M is the size of the array.

#### Concept of Collisions and Collision Handling:

* + Collision: When two keys hash to the same index.
  + Handling methods:
    - Chaining: Store all elements that hash to the same index in a list at that index.
    - Open Addressing: Find the next available slot.
    - Quadratic Probing: Use a quadratic function to find the next slot.

#### Advantages of Hashmaps/Hashtables:

* + Fast data retrieval with average time complexity of O(1) for insertion, deletion, and search operations.

## 4.2 Tasks:

### Task: Implement a Simple Hashmap

* + Description: Create a basic hashmap with insert, delete, and get functions.
  + Test Case: Insert ("key1", 10), Get key1 should return 10.

### Task: Check Subset Array

* + Description: Determine if one array is a subset of another using hashmap.
  + Input: arr1 = [11, 1, 13, 21, 3, 7], arr2 = [11, 3, 7, 1]
  + Output: True
  + Test Case: assert isSubset([11, 1, 13, 21, 3, 7], [11, 3, 7, 1]) == True

### Task: Frequency Sort

* + Description: Sort elements in an array by their frequency.
  + Input: arr = [2, 5, 2, 8, 5, 6, 8, 8]
  + Output: [8, 8, 8, 2, 2, 5, 5, 6]
  + Test Case: assert frequencySort([2, 5, 2, 8, 5, 6, 8, 8]) == [8, 8, 8, 2, 2, 5, 5, 6]

### Task: Unique Character Sets

* + Description: Group words that have the same unique character set.
  + Input: words = ["may", "student", "students", "dog"]
  + Output: [["may"], ["student", "students"], ["dog"]]
  + Test Case: assert groupWordsWithSameCharSet(["may", "student", "students", "dog"]) == [["may"], ["student", "students"], ["dog"]]

### Task: Implement a Dictionary Class

* + Description: Create a dictionary class with methods to add, remove, and find key-value pairs.
  + Test Case: Add ("apple", "fruit"), Remove apple, Find apple should return null.

### Task: Count Distinct Elements

* + Description: Use a hashmap to count distinct elements in an array.
  + Input: [1, 2, 2, 3, 3, 3, 4]
  + Output: 4
  + Test Case: assert countDistinctElements([1, 2, 2, 3, 3, 3, 4]) == 4

### Task: First Non-Repeating Character

* + Description: Find the first non-repeating character in a string using hashmap.
  + Input: "swiss"
  + Output: "w"
  + Test Case: assert firstNonRepeatingChar("swiss") == "w"

### Task: Intersection of Two Arrays

* + Description: Find the intersection of two arrays using hashmaps.
  + Input: arr1 = [1, 2, 3, 4], arr2 = [2, 3, 5]
  + Output: [2, 3]
  + Test Case: assert intersection([1, 2, 3, 4], [2, 3, 5]) == [2, 3]

### Task: Custom Hash Function for String Keys

* + Description: Implement a custom hash function for handling string keys in a hashmap.
  + Test Case: Ensure the custom hash function distributes string keys evenly across the hashmap.

## Module –Time Complexity

## Discussion:

### Introduction to Time Complexity:

* + Time complexity is a measure of the amount of time an algorithm takes to complete as a function of the length of the input.
  + It's crucial for understanding the efficiency of algorithms, particularly in large-scale data processing.

### Big O Notation:

* + The most common way to express time complexity, providing an upper bound on the time required.
  + Common complexities include O(1), O(log n), O(n), O(n log n), and O(n²).

### Best, Worst, and Average Case Analysis:

* + Understanding these different cases and how they impact the performance of an algorithm.

### Common Algorithmic Time Complexities:

* + Discussing the time complexities of standard algorithms like sorting, searching, and traversal algorithms.

### Space Complexity:

* + Alongside time complexity, understanding how much memory an algorithm uses.

### Tasks:

* Analyze Basic Operations:
  + Given Scenario: Simple operations like addition, lookup in an array.
  + Task: Determine the time complexity of these operations (e.g., O(1) for array lookup).
* Comparing Sorting Algorithms:
  + Given Scenario: Implementations of Bubble Sort, Merge Sort, and Quick Sort.
  + Task: Analyze and compare their time complexities.
* Recursive Function Analysis:
  + Given Scenario: A recursive function, such as calculating Fibonacci numbers.
  + Task: Determine the time complexity of the recursive solution.
* Nested Loops Complexity:
  + Given Scenario: Algorithms with nested loops.
  + Task: Analyze the time complexity of algorithms with different nesting levels.
* Search Algorithm Analysis:
  + Given Scenario: Linear search vs. binary search.
  + Task: Analyze and compare their time complexities.
* Divide and Conquer Complexity:
  + Given Scenario: Algorithms like Merge Sort or Binary Search.
  + Task: Explain how divide and conquer affects time complexity, typically O(n log n) or O(log n).

## Module 5 – Sorting Algorithms (Bubble, Insertion, Selection)

#### 

## 5.1 Discussion:

* + Basic sorting algorithms: Bubble Sort, Insertion Sort, and Selection Sort.
  + Discussing the time complexity of each algorithm

## 5.2 Tasks:

* + Task 1: Implement Bubble Sort  
     Description: Write a function to sort an array of integers using Bubble Sort.  
     Example: Input: [5, 3, 8, 4, 2], Output: [2, 3, 4, 5, 8]  
     Test Case: assert bubbleSort([5, 3, 8, 4, 2]) == [2, 3, 4, 5, 8]
  + Task 2: Implement Insertion Sort  
     Description: Create a function that sorts an array using Insertion Sort.  
     Example: Input: [7, 8, 5, 2, 4], Output: [2, 4, 5, 7, 8]  
     Test Case: assert insertionSort([7, 8, 5, 2, 4]) == [2, 4, 5, 7, 8]
  + Task 3: Implement Selection Sort  
     Description: Develop a function to perform Selection Sort on an array of numbers.  
     Example: Input: [9, 7, 5, 11, 12, 2], Output: [2, 5, 7, 9, 11, 12]  
     Test Case: assert selectionSort([9, 7, 5, 11, 12, 2]) == [2, 5, 7, 9, 11, 12]

## Module 6 – Tree Data Structures

### 6.1 Discussion: Tree Data Structures

#### Tree Structure:

* + Trees are hierarchical data structures with a root node from which all other nodes branch out. Each node in a tree can have zero or more child nodes.

#### Binary Trees:

* + A special type of tree where each node has at most two children, often referred to as the left and right children. Ideal for representing sorted data and facilitating quick search, insert, and delete operations.

#### Tree Traversal:

* + Involves visiting all the nodes of the tree. Primary methods are:
  + In-order Traversal: Visits left subtree, root, then right subtree.
  + Pre-order Traversal: Visits root, left subtree, then right subtree.
  + Post-order Traversal: Visits left subtree, right subtree, then the root.
  + Balanced Trees: Trees like AVL or Red-Black trees maintain balance to ensure operations are performed efficiently.

#### Applications:

* + Widely used in various applications like databases, file systems, and in implementing abstract data types like sets and maps.

### 6.2 Tasks and Test Cases

#### Implement a Binary Tree:

* + Description: Create a binary tree and implement functions for insertion and search.
  + Test Case: After inserting elements [10, 5, 15, 3, 7, 12, 18], check if 12 exists in the tree.

### Tree Traversal Algorithms:

* + Description: Implement in-order, pre-order, and post-order traversal for a binary tree.
  + Test Case: For a tree with elements [1, 2, 3, 4, 5], verify the in-order traversal is [1, 2, 3, 4, 5].

#### Construct a Binary Search Tree (BST):

* + Description: Build a BST and ensure that it maintains the property of a BST upon insertions.
  + Test Case: Insert [8, 3, 10, 1, 6, 14, 4, 7, 13] and verify the structure of the BST.

#### Height of a Binary Tree:

* + Description: Write a function to find the height of a binary tree.
  + Test Case: For a tree with elements [3, 1, 7, 5, 9], the height should be 3.

#### Find Minimum and Maximum in a BST:

* + Description: Implement functions to find the minimum and maximum value in a BST.
  + Test Case: In a BST with [5, 2, 12, -4, 3, 9, 21], minimum is -4 and maximum is 21.

#### Check if a Binary Tree is Balanced:

* + Description: Determine if a given binary tree is height-balanced.
  + Test Case: Check if a tree with nodes [1, 2, 3, 4, 5, 6, null, null, 7] is balanced.

#### Level Order Traversal in a Binary Tree:

* + Description: Perform level order traversal on a binary tree.
  + Test Case: For tree [3, 9, 20, null, null, 15, 7], the level order should be [3, 9, 20, 15, 7].

#### Convert Sorted Array to Binary Search Tree:

* + Description: Create a function to build a height-balanced BST from a sorted array.
  + Test Case: Convert [-10, -3, 0, 5, 9] to a BST and check balance.

#### Find LCA (Lowest Common Ancestor) in BST:

* + Description: Find the LCA of two given nodes in a BST.
  + Test Case: For nodes 2 and 8 in BST [6, 2, 8, 0, 4, 7, 9], LCA is 6.

#### Delete a Node in a BST:

* + Description: Implement the deletion of a node in a BST.
  + Test Case: Delete 15 from BST [10, 5, 20, 3, 7, 15] and verify the new structure.

#### 

## Week 8-9: Database Concepts

## Module 1: Relational Databases, SQL Basics, Keys, ER Diagrams

### 1.1 Discussion:

#### Relational Databases:

* + Introduction to relational databases and their role in storing and managing data.
  + Discussion of tables, rows, and columns and how they represent data.

#### SQL Basics:

* + Overview of SQL (Structured Query Language) for database interaction.
  + Basic commands: SELECT, INSERT, UPDATE, DELETE.

#### Keys in Databases:

* + Primary keys: Uniquely identifying each row in a table.
  + Foreign keys: Linking rows between tables and maintaining referential integrity.

#### Entity-Relationship (ER) Diagrams:

* + Tools for visualizing the relationships between different entities in a database.
  + Understanding entities, relationships, and cardinalities.

### 1.2 Tasks:

#### Create a Basic Database:

* + Set up a simple relational database with a few tables.
  + Practice creating tables with appropriate data types.

#### Basic SQL Queries:

* + Write SQL queries to SELECT data from a table.
  + Practice INSERT, UPDATE, and DELETE operations.

#### Primary and Foreign Key Implementation:

* + Create a table with a primary key and another table with a foreign key that references the first table.

#### Design an ER Diagram:

* + Design an ER diagram for a simple application like a library or a bookstore

### Module 2: Normalization

### 2.2 Discussion:

#### Normalization in Databases:

* + The process of organizing data to minimize redundancy and dependency.
  + Normalization involves dividing large tables into smaller, more manageable tables and defining relationships between them.

#### Normal Forms:

* + Explanation of different normal forms (1NF, 2NF, 3NF, BCNF) and their significance.
  + The goal is to reduce data duplication and improve data integrity.

#### Anomalies in Databases:

* + Understanding update, insert, and delete anomalies and how normalization addresses these issues.

#### Practical Implementation:

* + Discussion on the balance between normalization and performance, considering query complexity and database size.

### 2.3 Tasks:

#### Convert to 1NF:

* + Take a denormalized table and convert it to First Normal Form (1NF).
  + Ensure that each column contains atomic values, and each record is unique.
* Implement 2NF:
  + Advance a 1NF database table to Second Normal Form (2NF).
  + Eliminate redundant data by removing subsets of data that apply to multiple rows and placing them in separate tables.

#### Achieve 3NF:

* + Transform a 2NF table structure to Third Normal Form (3NF).
  + Ensure that all non-primary key fields are dependent on the primary key.

#### BCNF (Boyce-Codd Normal Form):

* + Apply BCNF to a 3NF table structure.
  + Resolve any remaining anomalies not addressed by 3NF.

#### Normalization Case Study:

* + Analyze a real-world dataset and perform normalization steps to optimize its structure.

## Module 3: Sub-Queries

### 3.1Discussion:

#### Sub-Queries in SQL:

* + Sub-queries, also known as nested queries, are queries within a query, used to perform operations that require multiple query steps in a single SQL statement.
  + They can be used in various clauses like SELECT, FROM, WHERE, and HAVING.

#### Types of Sub-Queries:

* + Single-row sub-queries: Return only one row.
  + Multi-row sub-queries: Return multiple rows and often use operators like IN, ANY, ALL.
  + Correlated sub-queries: Refer to a column in the outer query.

#### Performance Considerations:

* + Discuss the impact of sub-queries on query performance and when to use them effectively.

### 3.2 Tasks:

#### Basic Sub-Query:

* + Given Dataset: Employee table with EmployeeID, Name, Salary, DepartmentID.
  + Task: Find the names of employees who earn more than the average salary.

#### Sub-Query in FROM Clause:

* + Given Dataset: Products table with ProductID, ProductName, Price, and Orders table with OrderID, ProductID, Quantity.
  + Task: Use a sub-query in the FROM clause to find the total quantity ordered for each product.

#### Correlated Sub-Query:

* + Given Dataset: Customer table with CustomerID, Name, City, and Orders table with OrderID, CustomerID, OrderDate.
  + Task: List all customers who placed orders after their first order.

#### Sub-Query with EXISTS:

* + Given Dataset: Books table with BookID, Title, AuthorID, and Authors table with AuthorID, Name.
  + Task: Find all authors who have written more than 5 books.

#### Sub-Query with IN Operator:

* + Given Dataset: Employee table and a Department table with DepartmentID, DepartmentName.
  + Task: Find employees working in departments with more than 10 employees.

#### Nested Sub-Queries:

* + Given Dataset: Sales table with SaleID, Date, Amount, CustomerID.
  + Task: Identify customers who made purchases above the average amount on more than 3 different days.

#### Sub-Query in SELECT Clause:

* + Given Dataset: A table of Students and a table of Exams with scores.
  + Task: Display each student's name and their highest exam score.

#### Sub-Query for Data Cleaning:

* + Given Dataset: A database with inconsistent date formats across tables.
  + Task: Use sub-queries to normalize date formats.

#### Comparing Data with Sub-Queries:

* + Given Dataset: A table of Products and a table of Suppliers.
  + Task: Find products that are cheaper than the average price of products from the same supplier.

#### Analytical Sub-Query:

* + Given Dataset: A Sales table with daily sales data.
  + Task: Calculate monthly sales growth using sub-queries.

## Module 4: Joins

### 4.1 Discussion:

#### Joins in SQL:

* + Joins are used to combine rows from two or more tables, based on a related column between them.
  + Essential for querying data that is spread across different tables in a relational database.

#### Types of Joins:

* + Inner Join: Returns rows when there is a match in both tables.
  + Left (Outer) Join: Returns all rows from the left table, and the matched rows from the right table.
  + Right (Outer) Join: Returns all rows from the right table, and the matched rows from the left table.
  + Full (Outer) Join: Combines the results of both left and right outer joins.
  + Cross Join: Returns the Cartesian product of the sets of rows from the joined tables.
  + Self Join: A regular join, but the table is joined with itself.

#### Join Conditions and Filtering:

* + Discussion on using ON, USING, and WHERE clauses to specify join conditions.
  + Techniques for filtering and sorting results obtained from joins.

### 4.2 Tasks:

#### Implement an Inner Join:

* + Given Dataset: Employees table and Departments table.
  + Task: List all employees with their respective department names.

#### Perform a Left Join:

* + Given Dataset: Customers table and Orders table.
  + Task: Find all customers and their order details, including customers who have not placed any orders.

#### Execute a Right Join:

* + Given Dataset: Authors table and Books table.
  + Task: Display all books, including those without a listed author.

#### Utilize Full Outer Join:

* + Given Dataset: Products table and Suppliers table.
  + Task: Combine all products with their suppliers, including products without suppliers and suppliers without products.

#### Cross Join Usage:

* + Given Dataset: Colors table and Shapes table.
  + Task: Create a list of all possible combinations of colors and shapes.

#### Self Join Scenario:

* + Given Dataset: Employees table with a manager reference.
  + Task: List all employees with their manager's name.

#### Multiple Joins in a Single Query:

* + Given Dataset: Orders, Customers, and Products tables.
  + Task: Display all orders with customer names and product details.

#### Advanced Join with Aggregation:

* + Given Dataset: Sales table and Employees table.
  + Task: Calculate the total sales for each employee.

#### Join with Complex Conditions:

* + Given Dataset: Students table and Grades table.
  + Task: Display students' names with their highest grade, where grade is above a certain threshold.

#### Nested Joins:

* + Given Dataset: Orders, OrderDetails, Customers, and Products tables.
  + Task: Retrieve detailed order information, including customer and product details, for all orders above a certain amount.

## Module 5: Indexes

### 5.1 Discussion:

#### Indexes in Databases:

* + Indexes are database objects that improve the speed of data retrieval operations on a database table, at the cost of additional writes and storage space.
  + They are similar to indexes in books, providing quick access to rows within tables.

#### Types of Indexes:

* + Single-Column Indexes: Built on only one column of a table.
  + Composite Indexes: Built on more than one column of a table for more complex queries.
  + Unique Indexes: Ensure that the index key contains only unique values.
  + Clustered and Non-Clustered Indexes: Clustered indexes sort and store the data rows in the table based on their key values, while non-clustered indexes store only the index and a pointer to the actual data.

#### Index Creation and Management:

* + Discuss the SQL commands for creating and managing indexes.
  + Understanding the impact of indexes on database performance, especially in insert, update, and delete operations.

### 5.2 Tasks:

#### Create a Single-Column Index:

* + Given Dataset: A Customers table with numerous rows.
  + Task: Create an index on the LastName column to optimize search queries.

#### Implement a Composite Index:

* + Given Dataset: Orders table with CustomerID and OrderDate.
  + Task: Create a composite index on both columns to improve query performance for reports.

#### Unique Index Application:

* + Given Dataset: Employees table with an EmployeeNumber.
  + Task: Ensure EmployeeNumber is unique across the table by creating a unique index.

#### Explore Clustered Indexes:

* + Given Dataset: A large Products table.
  + Task: Create a clustered index on the ProductID column and observe the changes in data storage.

#### Non-Clustered Index Demonstration:

* + Given Dataset: Transactions table with detailed records.
  + Task: Build a non-clustered index on the TransactionDate column for faster querying.

#### Index on Foreign Key Columns:

* + Given Dataset: Orders table with a foreign key to Customers.
  + Task: Create an index on the foreign key column to enhance join operations.
* Analyze Index Impact on CRUD Operations:
  + Given Dataset: Any table with existing indexes.
  + Task: Analyze the performance impact of indexes on insert, update, and delete operations.

#### Use Indexes in Complex Queries:

* + Given Dataset: A database with multiple related tables.
  + Task: Write complex queries and use indexes to optimize their performance.

#### Index Management:

* + Given Dataset: A database with outdated and unused indexes.
  + Task: Identify and remove unnecessary indexes to improve overall performance.

#### Balancing Indexes and Performance:

* + Scenario: A database experiencing slow write operations.
  + Task: Optimize the number and type of indexes to balance read and write performance.

## Week 10-11: Mock Interview Sessions and Final Review

* + Mock Technical Interviews
  + Technical Interview Practice (Coding, DSA, OOP,Database)
  + Final Preparation and Mock Interviews

6.Past Interview Questions Data

* + - <https://drive.google.com/file/d/1RFpNXTy2HWeiuNFNh9ljdK6Ql45q_wO2/view>
    - <https://docs.google.com/document/d/14JhM004ubU9nP53bOPEJdcEPLe2lXTeMouTWb-IoxGg/edit?usp=sharing>
    - <https://docs.google.com/document/d/1Gh9yaRmDeTwTUr3reliDGwrceWO5f8bVmJJfI8ZweJM/edit?usp=sharing>
    - <https://docs.google.com/document/d/1hwkVZ24ZSpJn7Dt4CRNtiWaSLvh_hmbLrkfT5ddu7A8/edit?usp=sharing>