Topic 1: Fundamentals of DSA

Task 1: Analyzing Time Complexity of Algorithms

Objective: Understand and compare the time complexity of different algorithms.

Instructions:

1. Implement three sorting algorithms: **Bubble Sort, Merge Sort, and Quick Sort** in Python.
2. Generate a random list of **1000 integers** and sort them using these algorithms.
3. Measure and compare their execution times using the `time` module.
4. Explain the observed time complexities (`O(n^2)`, `O(n log n)`, etc.) and compare their efficiency with a graphical representation (using `matplotlib`).

Deliverables:

- Python code implementation
- A comparison table and execution time graph
- A brief explanation of results in a report

Task 2: Recursive vs Iterative Approach

Objective: Analyze and compare recursion with iteration in solving problems.

Instructions:

1. Implement a function to calculate the **nth Fibonacci number** in both:
   - **Recursive approach**
   - **Iterative approach**
2. Measure and compare their execution time for **n = 10, 20, 30, and 40**.
3. Optimize the recursive version using **memoization (dynamic programming)**.
4. Explain the difference in terms of performance, space complexity, and when recursion should be avoided.

Deliverables:

- Python code for all three implementations
- Execution time analysis for different values of `n`
- Explanation of recursion, iteration, and optimization

Task 3: Visualizing Big-O Notation

Objective: Develop an interactive Python program to visualize the growth of different time complexities.

Instructions:

1. Implement functions for different complexities (`O(1)`, `O(log n)`, `O(n)`, `O(n log n)`, `O(n^2)`, `O(2^n)`).
2. Generate input sizes from **1 to 1000** and compute the corresponding output values.
3. Write a report explaining the meaning of these complexities and real-world examples of each.

Deliverables:

- Python script with complexity functions
- Graphs comparing different complexities
- A report explaining Big-O notation with examples

Topic 2: Arrays & Strings

Task 1: Implementing Custom Array Operations

Objective: Develop an understanding of array manipulation techniques.

Instructions:

1. Implement a **custom dynamic array class** in Python (without using built-in `list`).
2. Include the following operations:
   - Insert an element at the end
   - Insert an element at a specific index
   - Delete an element at a specific index
   - Search for an element (return index if found, else -1)
   - Resize the array dynamically
3. Demonstrate the implementation with test cases.

Deliverables:

- Python code for the dynamic array class
- Test cases demonstrating each operation
- Explanation of time complexity for each operation

Task 2: Finding the Longest Substring Without Repeating Characters

Objective: Solve a real-world string manipulation problem efficiently.

Instructions:

1. Write a Python program to find the **longest substring** in a given string **without repeating characters**.
2. Implement two approaches:
   - **Brute force method** ($O(n^2)$ complexity)
   - **Sliding window method** ($O(n)$ complexity)
3. Compare both implementations in terms of execution time.
4. Provide test cases with various input strings.

Example Input:

```
s = "abcabcbb"
```

Expected Output:

```
Longest substring: "abc", Length: 3
```

Deliverables:

- Python code for both approaches
- Execution time comparison
- Explanation of sliding window technique

Task 3: Two-Dimensional Array – Image Rotation

Objective: Apply matrix transformations to manipulate 2D arrays.

Instructions:

1. Write a Python program to **rotate a given N x N matrix (2D array) by 90 degrees** clockwise.
2. Implement the solution **without using extra space**.
3. Test the program on a sample matrix and verify the output.

Example Input:

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
```

Expected Output (90° Clockwise Rotation):

```
[
    [7, 4, 1],
    [8, 5, 2],
    [9, 6, 3]
]
```

Deliverables:

- Python code for matrix rotation
- Explanation of in-place transformation
- Time complexity analysis

Topic 3: Linked Lists

Task 1: Implementing a Singly Linked List

Objective: Understand and implement the basic operations of a **Singly Linked List (SLL)**.

Instructions:

1. Implement a **Singly Linked List (SLL) class** in Python with the following operations:
    - Insert a node at the **beginning**
    - Insert a node at the **end**
    - Insert a node at a **specific position**
    - Delete a node by **value**
    - Search for a node by **value**
    - Display the linked list
2. Write test cases to verify the correctness of all operations.

Example:

```
LinkedList: 1 → 2 → 3 → 4
Insert(5) at end: 1 → 2 → 3 → 4 → 5
Delete(3): 1 → 2 → 4 → 5
Search(4): Found at position 3
```

Deliverables:

- Python implementation of the **SLL class**
- Test cases demonstrating all operations
- Time complexity analysis of each operation

Task 2: Detecting and Removing a Loop in a Linked List

Objective: Learn how to detect and fix cycles in a linked list using Floyd’s Cycle Detection Algorithm.

Instructions:

1. Implement a function to check if a **Singly Linked List** has a **loop (cycle)**.
2. If a loop is detected, find the **starting node of the loop**.
3. Implement a method to **remove the loop** without breaking the rest of the linked list.
4. Test the program with a linked list containing a loop.

Hint: Use **Floyd’s Cycle Detection Algorithm (Tortoise & Hare Method)**.

Example Input:

```
LinkedList: 1 → 2 → 3 → 4 → 5 → 3 (loop back to node 3)
```

Expected Output:

```
Loop detected at node 3
Loop removed, LinkedList: 1 → 2 → 3 → 4 → 5
```

Deliverables:

- Python code for loop detection and removal
- Explanation of Floyd's Algorithm
- Edge cases handling

Task 3: Implementing a Doubly Linked List and Reverse Traversal

Objective: Learn to work with **Doubly Linked Lists (DLL)** and perform reverse traversal.

Instructions:

1. Implement a **Doubly Linked List (DLL) class** in Python with the following operations:
    - Insert a node at the **beginning**
    - Insert a node at the **end**
    - Delete a node at a **specific position**
    - Traverse the list in **forward and reverse order**
2. Demonstrate the reverse traversal of the linked list.

Example:

```
LinkedList (forward): 1 → 2 → 3 → 4 → 5
LinkedList (reverse): 5 → 4 → 3 → 2 → 1
```

Deliverables:

- Python implementation of the **DLL class**
- Demonstration of forward and reverse traversal
- Time complexity analysis of all operations

Topic 4: Stacks & Queues

Task 1: Implementing a Stack Using Arrays and Linked Lists

Objective: Understand stack operations and implement a stack using two different data structures.

Instructions:

1. Implement a **Stack class** in Python using **arrays (Python lists)**.
2. Implement the same **Stack class** using a **linked list**.
3. Include the following stack operations in both implementations:
   - `push(element)`: Insert an element at the top.
   - `pop()`: Remove the top element.
   - `peek()`: Return the top element without removing it.
   - `is_empty()`: Check if the stack is empty.
   - `size()`: Return the number of elements in the stack.
4. Write test cases to verify each function.
5. Compare the **time complexity** and **memory usage** of both implementations.

Deliverables:

- Python implementations of stack using **arrays and linked lists**
- Comparison of both implementations in terms of performance
- Test cases demonstrating stack operations

Task 2: Evaluating Postfix Expressions Using Stacks

Objective: Apply the stack data structure to solve an arithmetic expression evaluation problem.

Instructions:

1. Write a Python function to evaluate **postfix expressions** using a **stack**.
2. The function should support basic operators: `+`, `-`, `*`, `/`.
3. Implement test cases with different expressions.

Example Input:

```
expression = "5 1 2 + 4 * + 3 -"
```

Step-by-Step Evaluation:

```
1. 1 + 2 = 3
2. 3 * 4 = 12
```

```
3. 5 + 12 = 17
4. 17 - 3 = 14
```

Expected Output:

```
Result: 14
```

Deliverables:

- Python function to evaluate postfix expressions
- Test cases with different expressions
- Explanation of how stack is used in expression evaluation

Task 3: Implementing a Circular Queue

Objective: Implement a **circular queue** and compare it with a linear queue.

Instructions:

1. Implement a **Circular Queue class** in Python with a fixed size.
2. Include the following operations:
   - `enqueue(element)`: Add an element to the rear.
   - `dequeue()`: Remove an element from the front.
   - `front()`: Get the front element without removing it.
   - `rear()`: Get the last element without removing it.
   - `is_empty()`: Check if the queue is empty.
   - `is_full()`: Check if the queue is full.
3. Compare it with a **normal queue** (using a Python list) in terms of efficiency.
4. Provide test cases for different operations.

Example Input & Output:

```
cq = CircularQueue(5)
cq.enqueue(10)
cq.enqueue(20)
cq.enqueue(30)
cq.enqueue(40)
cq.enqueue(50)
cq.dequeue()
cq.enqueue(60)
cq.front()  # Output: 20
cq.rear()   # Output: 60
```

Deliverables:

- Python implementation of a **circular queue**
- Comparison with a **linear queue**

Topic 5: Hashing & Hash Tables

Task 1: Implementing a Custom Hash Table with Collision Handling

Objective: Understand and implement a hash table using different collision resolution techniques.

Instructions:

1. Implement a **Hash Table class** in Python with the following functionalities:
   - `insert(key, value)`: Store a key-value pair.
   - `get(key)`: Retrieve a value using a key.
   - `delete(key)`: Remove a key-value pair.
   - `display()`: Print the current state of the hash table.
2. Implement **two collision resolution techniques**:
   - **Chaining (Separate Chaining using Linked Lists)**
   - **Open Addressing (Linear Probing)**
3. Compare both methods in terms of performance by inserting and searching 1000 random key-value pairs.

Deliverables:

- Python implementation of the **Hash Table class**
- Comparison of chaining vs. open addressing in terms of time complexity
- Test cases verifying different operations

Task 2: Checking if Two Strings Are Anagrams Using Hashing

Objective: Use hashing to efficiently check if two strings are anagrams.

Instructions:

1. Write a Python function to determine whether two strings are **anagrams** using a **hash table (dictionary)**.
2. The function should count the frequency of each character in the first string and compare it with the second string.
3. Test the function with multiple cases, including large strings.

Example Input & Output:

```
are_anagrams("listen", "silent")   # Output: True
are_anagrams("hello", "world")     # Output: False
```

Deliverables:

- Python function for checking anagrams using hashing
- Test cases for different string pairs
- Explanation of why hashing is an efficient approach (O(n) time complexity)

Task 3: Implementing a Simple Caching Mechanism Using Hash Maps

Objective: Use hashing to create an efficient **cache system (LRU Cache)**.

Instructions:

1. Implement an **LRU (Least Recently Used) Cache** using a **hash table and doubly linked list**.
2. The cache should have the following methods:
   - `put(key, value)`: Insert a key-value pair (if the cache is full, remove the least recently used item).
   - `get(key)`: Retrieve the value for a key (if the key exists, update its recent usage).
   - `display()`: Print the current state of the cache.
3. Set the cache capacity to **5** and test it with at least **10 insertions and retrievals**.

Example Input & Output:

```
cache = LRUCache(5)
cache.put(1, "A")
cache.put(2, "B")
cache.put(3, "C")
cache.put(4, "D")
cache.put(5, "E")
cache.get(2)  # Moves '2' to the most recently used position
cache.put(6, "F")  # Removes the least recently used key (1)
cache.display()
```

Expected Output:

```
Cache state: {2: "B", 3: "C", 4: "D", 5: "E", 6: "F"}
```

Deliverables:

- Python implementation of the **LRU Cache**
- Test cases verifying cache behavior
- Explanation of why hashing improves caching performance

Topic 6: Trees & Binary Search Trees (BST)

Task 1: Implementing a Binary Search Tree (BST) with Basic Operations

Objective: Understand the structure and operations of a **Binary Search Tree (BST)**.

Instructions:

1. Implement a **BST class** in Python with the following methods:
   - `insert(value)`: Add a new node with the given value.
   - `search(value)`: Check if a value exists in the tree.
   - `delete(value)`: Remove a node from the tree (handle cases: leaf node, one child, two children).
   - `inorder_traversal()`: Print elements in ascending order.
2. Test the BST by inserting a list of numbers and performing different operations.

Example Input & Output:

```
bst = BinarySearchTree()
bst.insert(50)
bst.insert(30)
bst.insert(70)
bst.insert(20)
bst.insert(40)
bst.insert(60)
bst.insert(80)
bst.inorder_traversal()
```

Expected Output:

```
20 30 40 50 60 70 80
```

Deliverables:

- Python implementation of the **BST class**
- Test cases demonstrating insertion, deletion, and search operations
- Explanation of BST properties and time complexity

Task 2: Finding the Lowest Common Ancestor (LCA) in a BST

Objective: Learn how to traverse a **BST** to find the **Lowest Common Ancestor (LCA)** of two given nodes.

Instructions:

1. Write a Python function that finds the **LCA of two nodes** in a BST.
2. If both nodes are on the **left subtree**, continue searching in the left subtree.

3. If both nodes are on the **right subtree**, continue searching in the right subtree.
4. If one node is on the left and the other is on the right, the current node is the **LCA**.
5. Test the function with multiple test cases.

Example Input & Output:

```
# BST Structure:
#         20
#        /  \
#      10   30
#     /  \  /  \
#    5   15 25 35

LCA(5, 15)  → Output: 10
LCA(5, 25)  → Output: 20
LCA(25, 35) → Output: 30
```

Deliverables:

- Python function to find the **LCA in a BST**
- Test cases with different trees and node pairs
- Explanation of the approach and time complexity

Task 3: Checking if a Binary Tree is Balanced

Objective: Understand the concept of **balanced binary trees** and implement a function to check if a tree is height-balanced.

Instructions:

1. Implement a Python function to check if a given **binary tree** is **height-balanced**.
2. A binary tree is balanced if **the height difference between the left and right subtrees of any node is at most 1**.
3. Use a **recursive approach** to compute the height of each subtree and check the balance condition.
4. Test the function with different tree structures.

Example Input & Output:

```
# Tree 1 (Balanced)
#       10
#      /  \
#     5    15
#    / \  /  \
#   2   7 12  20
is_balanced(root) → Output: True

# Tree 2 (Unbalanced)
#       10
#      /
#     5
#    /
```

```
#    2
is_balanced(root) → Output: False
```

Deliverables:

- Python function to check if a tree is **height-balanced**
- Test cases with both balanced and unbalanced trees
- Explanation of the recursive approach and time complexity

Topic 7: Heaps & Priority Queues

Task 1: Implementing a Min-Heap and Max-Heap

Objective: Understand the **heap data structure** and implement both **Min-Heap and Max-Heap** using Python.

Instructions:

1. Implement a **Heap class** supporting both **Min-Heap** and **Max-Heap**.
2. Include the following operations:
   - `insert(value)`: Insert a new element while maintaining the heap property.
   - `extract_root()`: Remove and return the root element (smallest in Min-Heap, largest in Max-Heap).
   - `peek()`: Return the root element without removing it.
   - `heapify(array)`: Convert an unsorted array into a valid heap.
3. Compare the time complexity of heap operations.

Example Input & Output:

```
min_heap = Heap("min")
min_heap.insert(10)
min_heap.insert(5)
min_heap.insert(20)
min_heap.insert(2)
print(min_heap.extract_root())  # Output: 2

max_heap = Heap("max")
max_heap.insert(10)
max_heap.insert(5)
max_heap.insert(20)
max_heap.insert(2)
print(max_heap.extract_root())  # Output: 20
```

Deliverables:

- Python implementation of **Min-Heap and Max-Heap**
- Test cases demonstrating heap operations
- Explanation of heap properties and time complexity

Task 2: Implementing a Priority Queue Using a Heap

Objective: Learn how **priority queues** work using a **heap**.

Instructions:

1. Implement a **PriorityQueue class** using a **Min-Heap** (lower values = higher priority).

2. Include the following operations:
   - `enqueue(value, priority)`: Insert an element based on priority.
   - `dequeue()`: Remove and return the element with the highest priority.
   - `peek()`: Return the highest priority element without removing it.
3. Test the priority queue with real-world use cases (e.g., task scheduling, emergency room patients).

Example Input & Output:

```
pq = PriorityQueue()
pq.enqueue("Task A", 3)
pq.enqueue("Task B", 1)
pq.enqueue("Task C", 2)
print(pq.dequeue())  # Output: "Task B" (highest priority)
```

Deliverables:

- Python implementation of a **priority queue using a heap**
- Test cases with different priorities
- Explanation of how heaps improve priority queue efficiency

Task 3: Finding the K Smallest and K Largest Elements Using a Heap

Objective: Use a heap to efficiently find the **K smallest and K largest elements** from an unsorted list.

Instructions:

1. Implement two Python functions:
   - `find_k_smallest(arr, k)`: Returns the **K smallest elements** using a **Min-Heap**.
   - `find_k_largest(arr, k)`: Returns the **K largest elements** using a **Max-Heap**.
2. Compare the heap-based approach with sorting (`O(n log n)`).

Example Input & Output:

```
arr = [10, 4, 3, 20, 15, 7]
print(find_k_smallest(arr, 3))  # Output: [3, 4, 7]
print(find_k_largest(arr, 2))   # Output: [20, 15]
```

Deliverables:

- Python functions to find **K smallest and K largest elements**
- Test cases with different inputs
- Comparison of heap vs sorting for efficiency

Topic 8: Graphs & Graph Algorithms

Task 1: Implementing a Graph Using Adjacency List & Adjacency Matrix

Objective: Learn how to represent graphs using **adjacency lists** and **adjacency matrices** in Python.

Instructions:

1. Implement a **Graph class** in Python with support for both **directed** and **undirected** graphs.
2. Implement the following representations:
   - **Adjacency List** (Dictionary of lists)
   - **Adjacency Matrix** (2D List)
3. Implement the following operations:
   - `add_edge(v1, v2)`: Add an edge between two vertices.
   - `remove_edge(v1, v2)`: Remove an edge between two vertices.
   - `display()`: Print the graph representation.

Example Input & Output:

```
g = Graph(5)
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 3)
g.add_edge(2, 4)
g.display()
```

Expected Output (Adjacency List):

```
0: [1, 2]
1: [0, 3]
2: [0, 4]
3: [1]
4: [2]
```

Deliverables:

- Python implementation of a **Graph class** using both **adjacency lists** and **adjacency matrices**
- Test cases demonstrating graph creation and modification
- Explanation of when to use adjacency lists vs adjacency matrices

---

Task 2: Implementing Breadth-First Search (BFS) and Depth-First Search (DFS)

Objective: Understand and implement **graph traversal** using **BFS** and **DFS**.

Instructions:

1. Implement functions for **BFS** and **DFS** traversal in a graph.
2. Use a **queue** for BFS and a **stack (recursion)** for DFS.
3. Test the functions on a sample graph and print the traversal order.

Example Input & Output:

```
g = Graph(6)
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 3)
g.add_edge(1, 4)
g.add_edge(2, 5)

print(g.bfs(0))  # Output: [0, 1, 2, 3, 4, 5]
print(g.dfs(0))  # Output: [0, 2, 5, 1, 4, 3] (or similar order)
```

Deliverables:

- Python implementation of **BFS and DFS** traversal
- Test cases with different graphs
- Comparison of BFS vs DFS with real-world applications

Task 3: Implementing Dijkstra's Algorithm for Shortest Path

Objective: Learn **Dijkstra's Algorithm** for finding the shortest path in a **weighted graph**.

Instructions:

1. Implement **Dijkstra's Algorithm** using a **priority queue (heap)**.
2. The function should take a **graph (dictionary of lists)** and a **starting node** as input.
3. Return the shortest distance to all other nodes.

Example Input & Output:

```
graph = {
    'A': {'B': 4, 'C': 1},
    'B': {'A': 4, 'C': 2, 'D': 5},
    'C': {'A': 1, 'B': 2, 'D': 8},
    'D': {'B': 5, 'C': 8}
}

print(dijkstra(graph, 'A'))
```

Expected Output:

```
{'A': 0, 'B': 3, 'C': 1, 'D': 9}
```

Deliverables:

- Python implementation of **Dijkstra’s Algorithm**
- Test cases for different weighted graphs
- Explanation of **time complexity (O(V log V))** and use cases

---

Topic 9: Sorting Algorithms

Task 1: Implementing and Analyzing Sorting Algorithms

Objective: Understand and implement different sorting algorithms and analyze their time complexity.

Instructions:

1. Implement the following sorting algorithms in Python:
   - **Bubble Sort**
   - **Selection Sort**
   - **Insertion Sort**
2. Write a function that sorts a given list using each of these algorithms.
3. Measure and compare the execution time for different list sizes.
4. Plot a graph to visualize the performance comparison.

Example Input & Output:

```
arr = [64, 25, 12, 22, 11]
print(bubble_sort(arr))     # Output: [11, 12, 22, 25, 64]
print(selection_sort(arr))  # Output: [11, 12, 22, 25, 64]
print(insertion_sort(arr))  # Output: [11, 12, 22, 25, 64]
```

Deliverables:

- Python implementations of **Bubble, Selection, and Insertion Sort**
- Execution time comparison for different input sizes
- Graph comparing time complexity trends

Task 2: Implementing Quick Sort and Merge Sort with Performance Comparison

Objective: Learn **Quick Sort** and **Merge Sort** and compare their efficiency.

Instructions:

1. Implement **Quick Sort** (using a pivot element) and **Merge Sort** (using recursion).
2. Write a function that takes an unsorted list and sorts it using both algorithms.
3. Generate random lists of different sizes (e.g., 1000, 5000, 10000) and compare their execution time.

Example Input & Output:

```
arr = [38, 27, 43, 3, 9, 82, 10]
print(quick_sort(arr))  # Output: [3, 9, 10, 27, 38, 43, 82]
```

```
print(merge_sort(arr))  # Output: [3, 9, 10, 27, 38, 43, 82]
```

Deliverables:

- Python implementations of **Quick Sort and Merge Sort**
- Performance comparison using different input sizes
- Explanation of when to use Quick Sort vs Merge Sort

Task 3: Implementing Heap Sort and Counting Sort for Large Datasets

Objective: Learn and apply **Heap Sort** and **Counting Sort** for sorting large datasets efficiently.

Instructions:

1. Implement **Heap Sort** using a binary heap data structure.
2. Implement **Counting Sort**, which is suitable for sorting numbers within a limited range.
3. Compare the performance of both algorithms with different datasets.

Example Input & Output:

```
arr = [4, 10, 3, 5, 1]
print(heap_sort(arr))  # Output: [1, 3, 4, 5, 10]

arr2 = [1, 4, 1, 2, 7, 5, 2]
print(counting_sort(arr2))  # Output: [1, 1, 2, 2, 4, 5, 7]
```

Deliverables:

- Python implementations of **Heap Sort and Counting Sort**
- Comparison of their performance on large datasets
- Explanation of time complexity and best use cases

Topic 10: Searching Algorithms

Task 1: Implementing Linear Search and Binary Search

Objective: Understand and compare **Linear Search** and **Binary Search** by implementing them in Python.

Instructions:

1. Implement **Linear Search**, which searches for a target element in an unsorted list.
2. Implement **Binary Search**, which works only on sorted lists and uses a divide-and-conquer approach.
3. Compare their performance on lists of different sizes (e.g., 1000, 5000, 10000 elements).
4. Write test cases for both algorithms.

Example Input & Output:

```
arr = [10, 23, 45, 70, 11, 15]
print(linear_search(arr, 45))  # Output: 2 (index of 45)
sorted_arr = [10, 15, 23, 45, 70]
print(binary_search(sorted_arr, 45))  # Output: 3 (index of 45)
```

Deliverables:

- Python implementations of **Linear Search and Binary Search**
- Comparison of execution time for different input sizes
- Explanation of when to use each algorithm

Task 2: Implementing Interpolation Search and Jump Search

Objective: Learn advanced searching techniques like **Interpolation Search** and **Jump Search** for improved efficiency.

Instructions:

1. Implement **Jump Search**, which works on sorted arrays and uses fixed jump sizes to improve efficiency.
2. Implement **Interpolation Search**, an improvement over Binary Search for uniformly distributed data.
3. Compare their efficiency with Binary Search using different datasets.

Example Input & Output:

```
arr = [1, 3, 5, 7, 9, 11, 13, 15]
print(jump_search(arr, 7))  # Output: 3 (index of 7)
```

```
print(interpolation_search(arr, 7))  # Output: 3 (index of 7)
```

Deliverables:

- Python implementations of **Jump Search and Interpolation Search**
- Performance comparison with **Binary Search**
- Explanation of best use cases for each search method

Task 3: Implementing Exponential Search and Fibonacci Search

Objective: Explore efficient searching methods like **Exponential Search** and **Fibonacci Search**.

Instructions:

1. Implement **Exponential Search**, which is useful when the target element is near the beginning of a sorted list.
2. Implement **Fibonacci Search**, which works similarly to Binary Search but uses Fibonacci numbers for partitioning.
3. Compare their performance with other searching algorithms.

Example Input & Output:

```
arr = [2, 4, 8, 16, 32, 64, 128]
print(exponential_search(arr, 32))  # Output: 4 (index of 32)
print(fibonacci_search(arr, 32))  # Output: 4 (index of 32)
```

Deliverables:

- Python implementations of **Exponential Search and Fibonacci Search**
- Performance comparison with other searching methods
- Explanation of when to use these algorithms

Topic 11: Hashing & Hash Tables

Task 1: Implementing a Simple Hash Table with Collision Handling

Objective: Understand **hashing** and implement a **hash table** using Python with collision handling techniques.

Instructions:

1. Implement a **HashTable class** with:
   - `insert(key, value)`: Inserts a key-value pair.
   - `get(key)`: Retrieves the value associated with the key.
   - `delete(key)`: Removes a key-value pair.
2. Implement **collision handling** using **chaining (linked lists)** and **open addressing (linear probing)**.
3. Test the hash table with different key-value pairs and ensure proper collision resolution.

Example Input & Output:

```
ht = HashTable(10)
ht.insert("name", "Alice")
ht.insert("age", 25)
print(ht.get("name"))  # Output: Alice
ht.delete("age")
print(ht.get("age"))  # Output: None
```

Deliverables:

- Python implementation of a **hash table**
- Demonstration of **collision handling techniques**
- Test cases and explanation of **time complexity**

Task 2: Implementing a Custom Hash Function and Analyzing Collisions

Objective: Design a custom **hash function** and analyze its effectiveness in minimizing collisions.

Instructions:

1. Implement a custom hash function that distributes keys uniformly across the hash table.
2. Compare the performance of your function with Python's built-in `hash()` function.
3. Insert a large dataset and measure the number of **collisions**.
4. Plot a **histogram** of hash values to visualize distribution.

Example Input & Output:

```
def custom_hash(key):
    return sum(ord(c) for c in key) % 10

print(custom_hash("hello"))  # Output: Some integer in the range 0-9
```

Deliverables:

- Python implementation of a **custom hash function**
- Collision analysis with different hashing strategies
- Graph showing **distribution of hash values**

Task 3: Implementing a Caching Mechanism using Hashing (LRU Cache)

Objective: Build an **LRU (Least Recently Used) Cache** using a **hash table** and a **doubly linked list**.

Instructions:

1. Implement an **LRUCache class** with:
   - `get(key)`: Returns value if present, otherwise -1.
   - `put(key, value)`: Adds a key-value pair and removes the **least recently used** item when full.
2. Use a **hash table** for quick lookups and a **doubly linked list** to maintain the order of usage.
3. Test with a sequence of cache operations.

Example Input & Output:

```
cache = LRUCache(2)
cache.put(1, "A")
cache.put(2, "B")
print(cache.get(1))  # Output: "A"
cache.put(3, "C")  # Removes least recently used key (2)
print(cache.get(2))  # Output: -1 (not found)
```

Deliverables:

- Python implementation of **LRU Cache**
- Performance comparison with simple dictionary-based caching
- Explanation of **time complexity (O(1) for get & put using OrderedDict/LinkedList)**

Topic 12: Graph Data Structure

Task 1: Implementing a Graph Using Adjacency List & Adjacency Matrix

Objective: Understand and implement **graph representations** in Python using both **adjacency list** and **adjacency matrix**.

Instructions:

1. Implement a `Graph` class that supports both **directed** and **undirected** graphs.
2. Implement **two representations**:
   - **Adjacency List** (Dictionary of lists)
   - **Adjacency Matrix** (2D list)
3. Include methods to:
   - Add a vertex
   - Add an edge
   - Display the graph

Example Input & Output:

```
g = Graph()
g.add_vertex("A")
g.add_vertex("B")
g.add_edge("A", "B")
g.display_adj_list()
g.display_adj_matrix()
```

Expected Output:

```
Adjacency List: {'A': ['B'], 'B': []}
Adjacency Matrix:
    A B
A [ 0 1 ]
B [ 0 0 ]
```

Deliverables:

- Python implementation of **both graph representations**
- Comparison of **space complexity**
- Explanation of **when to use each representation**

Task 2: Implementing Breadth-First Search (BFS) & Depth-First Search (DFS)

Objective: Learn and apply **BFS** and **DFS** traversal techniques on graphs.

Instructions:

1. Implement **BFS (Breadth-First Search)** using a **queue (FIFO)**.
2. Implement **DFS (Depth-First Search)** using both **recursion** and **stack (LIFO)**.
3. Use a sample graph and traverse it using both algorithms.

Example Input & Output:

```
g = Graph()
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 3)
g.add_edge(2, 3)
print(g.bfs(0))  # Output: [0, 1, 2, 3]
print(g.dfs(0))  # Output: [0, 2, 3, 1] or another valid DFS order
```

Deliverables:

- Python implementation of **BFS and DFS**
- Explanation of **time complexity (O(V + E))**
- Comparison of **use cases for BFS vs DFS**

Task 3: Implementing Dijkstra's Algorithm for Shortest Path

Objective: Learn and implement **Dijkstra's Algorithm** for finding the shortest path in a weighted graph.

Instructions:

1. Implement **Dijkstra's Algorithm** using a **priority queue (min-heap)**.
2. The function should take a **graph (adjacency list with weights)** and a **starting node**.
3. Return the shortest path from the source to all other nodes.

Example Input & Output:

```
g = Graph()
g.add_edge("A", "B", 4)
g.add_edge("A", "C", 1)
g.add_edge("C", "B", 2)
g.add_edge("B", "D", 1)
print(g.dijkstra("A"))
```

Expected Output:

```
{'A': 0, 'B': 3, 'C': 1, 'D': 4}
```

Deliverables:

- Python implementation of **Dijkstra's Algorithm**
- Explanation of **time complexity (O((V + E) log V))**
- Comparison with **Bellman-Ford Algorithm**

Topic 13: Greedy Algorithms

Task 1: Implementing Activity Selection Algorithm

Objective: Learn how **Greedy Algorithms** optimize decision-making by implementing the **Activity Selection Problem** to maximize the number of non-overlapping activities.

Instructions:

1. Given `n` activities with start and end times, select the **maximum number of activities** that can be performed without overlap.
2. Implement a function `activity_selection()` that sorts activities based on their **finish times** and selects non-overlapping activities greedily.
3. Test the algorithm on different datasets.

Example Input & Output:

```
activities = [(1, 3), (2, 5), (3, 9), (6, 8), (8, 11)]
print(activity_selection(activities))
```

Expected Output:

```
[(1, 3), (6, 8), (8, 11)]  # Maximum non-overlapping activities
```

Deliverables:

- Python implementation of **Activity Selection** using a **Greedy approach**
- Explanation of **time complexity (O(n log n) due to sorting)**
- Test cases and real-world applications

Task 2: Implementing Huffman Coding for Data Compression

Objective: Understand **Greedy Algorithms** in **data compression** by implementing **Huffman Encoding**, which minimizes the average code length for characters based on frequency.

Instructions:

1. Read an input string and **count character frequencies**.
2. Build a **Huffman Tree** using a **priority queue (min-heap)**.
3. Generate **Huffman codes** and encode the given string.
4. Compare the length of the **original vs. compressed data**.

Example Input & Output:

```
text = "hello greedy"
huffman_codes = huffman_encoding(text)
print(huffman_codes)
```

Expected Output:

```
{'h': '110', 'e': '10', 'l': '111', 'o': '01', 'g': '000', 'r': '001', 'd':
'0110', 'y': '0111'}
```

Deliverables:

- Python implementation of **Huffman Encoding**
- Comparison of **original vs compressed data size**
- Explanation of **why Huffman Coding is Greedy**

Task 3: Implementing Kruskal's Algorithm for Minimum Spanning Tree (MST)

Objective: Learn how **Greedy Algorithms** work in **graph optimization** by implementing **Kruskal's Algorithm** to find a **Minimum Spanning Tree (MST)**.

Instructions:

1. Represent a **weighted graph** using an **edge list**.
2. Sort edges by **weight** and use a **disjoint-set (Union-Find)** to avoid cycles.
3. Construct an **MST** with the **minimum total edge weight**.
4. Compare **Kruskal's Algorithm** with **Prim's Algorithm**.

Example Input & Output:

```
edges = [(1, 2, 4), (2, 3, 1), (1, 3, 3), (3, 4, 2)]
print(kruskal(edges, 4))
```

Expected Output:

```
[(2, 3, 1), (3, 4, 2), (1, 3, 3)]  # MST with minimum weight
```

Deliverables:

- Python implementation of **Kruskal's Algorithm**
- Explanation of **Greedy approach in MST**
- Comparison with **Prim's Algorithm**

Topic 14: Dynamic Programming (DP)

Task 1: Implementing the Fibonacci Sequence Using DP (Memoization & Tabulation)

Objective: Understand **Dynamic Programming** by implementing the **Fibonacci sequence** using both **memoization (top-down)** and **tabulation (bottom-up)** approaches.

Instructions:

1. Implement a recursive Fibonacci function with **memoization** (using a dictionary).
2. Implement an **iterative** Fibonacci function with **tabulation** (using an array).
3. Compare the **time complexity** and efficiency of both approaches.

Example Input & Output:

```
print(fib_memoization(10))  # Output: 55
print(fib_tabulation(10))   # Output: 55
```

Deliverables:

- Python implementation of **Fibonacci using Memoization & Tabulation**
- **Performance comparison** of both techniques
- Explanation of **time complexity (O(n) vs O(2^n) for naive recursion)**

Task 2: Implementing the Longest Common Subsequence (LCS) Algorithm

Objective: Learn **string processing** with **Dynamic Programming** by implementing **LCS**, which finds the longest subsequence common to two given strings.

Instructions:

1. Implement an LCS function using **recursion + memoization**.
2. Optimize it using **tabulation (2D DP table)**.
3. Test on different input strings and analyze the output.

Example Input & Output:

```
print(lcs("AGGTAB", "GXTXAYB"))  # Output: "GTAB"
```

Deliverables:

- Python implementation of **LCS using DP**
- **Comparison of recursive vs tabulation approaches**
- Explanation of **time complexity (O(m*n))**

Task 3: Implementing the 0/1 Knapsack Problem Using Dynamic Programming

Objective: Learn how **DP solves optimization problems** by implementing the **0/1 Knapsack Problem**, where you maximize the total value of items without exceeding a given weight limit.

Instructions:

1. Given `n` items with weights and values, and a total weight capacity `W`, implement a DP solution for the **0/1 Knapsack Problem**.
2. Use **bottom-up tabulation** to fill a **2D DP table**.
3. Return the **maximum value** that can be obtained without exceeding the weight limit.

Example Input & Output:

```
weights = [2, 3, 4, 5]
values = [3, 4, 5, 6]
capacity = 5
print(knapsack(weights, values, capacity))  # Output: 7
```

Deliverables:

- Python implementation of **0/1 Knapsack using DP**
- Explanation of **time complexity (O(n * W))**
- Real-world applications (e.g., **budget optimization, resource allocation**)

Topic 15: Backtracking

Task 1: Solving the N-Queens Problem Using Backtracking

Objective: Understand **backtracking** by solving the **N-Queens problem**, where `N` queens must be placed on an `N × N` chessboard so that no two queens attack each other.

Instructions:

1. Implement a function to solve the **N-Queens problem** using backtracking.
2. The function should return **all possible solutions**, where each solution represents a valid board configuration.
3. Optimize the solution using **constraint propagation** to reduce unnecessary computations.

Example Input & Output:

```
solve_n_queens(4)
```

Expected Output:

```
[
  [".Q..",      # Solution 1
   "...Q",
   "Q...",
   "..Q."],

  ["..Q.",      # Solution 2
   "Q...",
   "...Q",
   ".Q.."]
]
```

Deliverables:

- Python implementation of **N-Queens using Backtracking**
- Explanation of **time complexity (O(N!))**
- Visual representation of board placements

Task 2: Generating All Possible Permutations of a String

Objective: Learn **backtracking with recursion** by generating all possible permutations of a given string.

Instructions:

1. Implement a function that takes a string and returns **all unique permutations**.

2. Use **backtracking** to swap characters and explore all possible combinations.
3. Test with both **small and large strings** and analyze performance.

Example Input & Output:

```
print(permute("ABC"))
```

Expected Output:

```
['ABC', 'ACB', 'BAC', 'BCA', 'CAB', 'CBA']
```

Deliverables:

- Python implementation of **string permutation using Backtracking**
- Explanation of **time complexity (O(N!))**
- Use case examples (e.g., **password generation, anagrams**)

Task 3: Solving the Sudoku Puzzle Using Backtracking

Objective: Apply **backtracking** to solve a **Sudoku puzzle**, where a `9×9` grid must be filled following Sudoku rules.

Instructions:

1. Implement a function that takes a **partially filled** `9×9` Sudoku board and fills in the missing numbers.
2. Use **backtracking** to try all possible values while ensuring **row, column, and 3×3 subgrid constraints** are met.
3. Optimize using **forward checking** (eliminating impossible values early).

Example Input & Output:

```
sudoku_board = [
  [5, 3, 0, 0, 7, 0, 0, 0, 0],
  [6, 0, 0, 1, 9, 5, 0, 0, 0],
  [0, 9, 8, 0, 0, 0, 0, 6, 0],
  ...
]
solve_sudoku(sudoku_board)
```

Expected Output:

```
Solved Sudoku Board:
[[5, 3, 4, 6, 7, 8, 9, 1, 2],
 [6, 7, 2, 1, 9, 5, 3, 4, 8],
 ...
]
```

Deliverables:

- Python implementation of **Sudoku Solver using Backtracking**
- Explanation of **time complexity (O(9^(n)))**
- Comparison with **constraint satisfaction approaches**

Topic 16: Graph Algorithms

Task 1: Implementing Depth-First Search (DFS) and Breadth-First Search (BFS)

Objective: Understand **graph traversal algorithms** by implementing **DFS and BFS** on a given graph.

Instructions:

1. Implement **DFS (recursive and iterative)** to traverse a graph and print the **order of visited nodes**.
2. Implement **BFS (using a queue)** to explore nodes level by level.
3. Compare DFS and BFS in terms of **time complexity** and **use cases**.

Example Input & Output:

```
graph = {
  'A': ['B', 'C'],
  'B': ['D', 'E'],
  'C': ['F'],
  'D': [],
  'E': ['F'],
  'F': []
}
print(dfs(graph, 'A'))  # Output: ['A', 'B', 'D', 'E', 'F', 'C']
print(bfs(graph, 'A'))  # Output: ['A', 'B', 'C', 'D', 'E', 'F']
```

Deliverables:

- Python implementation of **DFS and BFS**
- Explanation of **time complexity (O(V + E))**
- Comparison of **when to use DFS vs BFS**

Task 2: Finding the Shortest Path Using Dijkstra's Algorithm

Objective: Learn **graph-based shortest path algorithms** by implementing **Dijkstra's Algorithm** for weighted graphs.

Instructions:

1. Implement **Dijkstra's Algorithm** using a **priority queue (min-heap)**.
2. Given a graph represented as an **adjacency list**, compute the **shortest path** from a given source node to all other nodes.
3. Test the algorithm on different graph structures.

Example Input & Output:

```
graph = {
  'A': {'B': 1, 'C': 4},
  'B': {'A': 1, 'C': 2, 'D': 5},
  'C': {'A': 4, 'B': 2, 'D': 1},
  'D': {'B': 5, 'C': 1}
}
print(dijkstra(graph, 'A'))
```

Expected Output:

```
{'A': 0, 'B': 1, 'C': 3, 'D': 4}  # Shortest distances from 'A'
```

Deliverables:

- Python implementation of **Dijkstra's Algorithm**
- Explanation of **why a priority queue is used (O((V + E) log V))**
- Real-world applications (e.g., **Google Maps, network routing**)

Task 3: Detecting Cycles in a Graph (Directed & Undirected)

Objective: Understand **cycle detection** in graphs by implementing cycle detection for **directed and undirected graphs** using DFS.

Instructions:

1. Implement **cycle detection** for an **undirected graph** using **Union-Find (Disjoint Set)**.
2. Implement **cycle detection** for a **directed graph** using **DFS + recursion stack**.
3. Test with cyclic and acyclic graphs and analyze the results.

Example Input & Output:

```
graph_undirected = {
  'A': ['B', 'C'],
  'B': ['A', 'D'],
  'C': ['A', 'D'],
  'D': ['B', 'C']
}
print(detect_cycle_undirected(graph_undirected))  # Output: True (cycle
exists)

graph_directed = {
  'A': ['B'],
  'B': ['C'],
  'C': ['A']
}
print(detect_cycle_directed(graph_directed))  # Output: True (cycle exists)
```

Deliverables:

- Python implementation of **cycle detection in graphs**
- Explanation of **DFS-based and Union-Find approaches**

- Discussion on **cycle detection in real-world applications (e.g., deadlock detection)**
