Assignment

1. Instructions

You will be given a set of real-world problems related to data structures, sorting algorithms, searching systems, and graph algorithms.

- **Problem 1**
  You will implement a recursive function to calculate the sum of the digits of a positive integer. The function should continue summing the digits until the number becomes a single-digit value.
- **Problem 2**
  You will implement a queue using different data structures (array, singly linked list, doubly linked list) and analyze their performance in terms of time and space complexity.
- **Problem 3**
  You will implement and benchmark several sorting algorithms (QuickSort, MergeSort, HeapSort, InsertionSort, SelectionSort) on different datasets, then visualize and compare their performance under various conditions.
- **Problem 4**
  You will implement a searching system that efficiently handles item searches in a warehouse, investigating performance issues and proposing improvements over time.
- **Problem 5**
  You will create a system for calculating optimal routes with both distance and toll considerations, using Dijkstra's Algorithm to find the shortest paths while modifying it to account for toll costs.

Error Handling: You do not need to implement error handling in your code. Assume that the input will always be in a normal format. It is sufficient to ensure the function runs correctly with one or two test cases.

References: You do not need to include references for this assignment.

Word Limit: There is no word limit for the assignment. Focus on the quality of your analysis and explanations.

2. Submission Guidelines

- **Programming Language**: You can use any programming language, such as C++, Java, Python, etc.
- **Report File**: You will be given a sample document. Use this as a template to structure your submission.

3. Rubric

Score for each problem:

- Problem 1 (5 points)
- Problem 2 (10 points)
- Problem 3 (10 points)
- Problem 4 (5 points)
- Problem 5 (5 points)

Each problem will be evaluated based on the following criteria:

1. **Functionality (40%)**
   Does the solution correctly implement all the required tasks and functionalities?
2. **Code Quality (30%)**
   Is the code well-structured and readable?
   - Is the code modular and organized into logical blocks?
   - Are comments included where necessary to explain the logic behind the code?
3. **Analysis and Explanation (20%)**
   Are the results of each task thoroughly analyzed and explained?
   - Are the time and space complexities of different algorithms discussed?
   - Are the performance results interpreted and explained clearly, with reasoning for optimizations where applicable?
4. **Creativity and Optimization (10%)**
   Does the submission show innovation or creative approaches to solving the problems?
   - Are optimizations made where the basic solutions perform poorly?
   - Are alternative solutions or improvements proposed, such as optimized sorting or search algorithms?

Problem 1

You are given a positive integer, and your task is to find the sum of its digits using recursion. For example, the sum of the digits of `123` is `1 + 2 + 3 = 6`. The function should keep summing the digits until the number becomes `0`.

Tasks:

1. Write a recursive function `sum_of_digits(n)` that takes a positive integer `n` as input and returns the sum of its digits.
2. If `n` is a single-digit number, return the number itself.

Example Input/Output:

- **Input:** `123`
- **Output:** `6` (1 + 2 + 3 = 6)
- **Input:** `98765`
- **Output:** `35` (9 + 8 + 7 + 6 + 5 = 35)
- **Input:** `5`

- **Output:** 5 (Single-digit number)

Problem 2

A **restaurant** manages incoming online food delivery orders. The system needs to handle orders in a **first-come, first-served** manner while allowing customers to cancel orders before they are processed. Suppose that there are **no priority levels**, no need for **reordering**, and no fixed number of order slots.

Tasks:

1. **Implement the queue using an array.**
   The queue should support:
   - **Add order**: Enqueue a new order.
   - **Process order**: Dequeue and remove the order from the queue.
   - **View next order**: Peek at the next order without removing it.
   - **Cancel order**: Remove a specific order from the queue.
   - **Check if empty**: Determine if there are pending orders.
2. Discuss the **limitations** of using an array-based queue for this implementation. **What alternative data structures** can improve efficiency when handling cancellations?

Problem 3

This section focuses on implementing and analyzing various sorting algorithms to determine their efficiency across different scenarios. You will benchmark their performance on different input sizes and types of lists, visualize the results, and explore hybrid sorting techniques.

Tasks:

1. **Benchmark Sorting Algorithms:**
   - Implement the following sorting algorithms:
     - **QuickSort**
     - **MergeSort**
     - **InsertionSort**
     - **SelectionSort**
   - Test their performance on lists of various sizes:
     - **1000, 5000, 10000, 20000, 30000, 40000, 50000, and 100000 elements**.
   - Run tests on **different types of input data**:
     - **Random lists**
     - **Reverse sorted lists**
     - **Lists with extreme value distributions (e.g., mostly small values with a few large outliers)**

        - **Lists with many duplicate values**
    - Visualize execution time results using line **graphs**.
2. **Scenario-Based Analysis:**
    - Based on the benchmarks, identify **which sorting algorithm performs best** under different conditions.
    - Provide an explanation of why certain algorithms perform better in some cases.
    - **In what scenario does QuickSort perform poorly?** Analyze cases where QuickSort has high time complexity and explain why it happens.

Problem 4

Due to an **accident** in a **warehouse**, a large collection of items lost their labels and now must be identified solely by their **weight**. The warehouse system needs to quickly find an object with a specific weight to fulfill customer orders. However, searching for an item manually through the entire collection is inefficient, so an organized system is needed to allow fast searches while also handling items being relocated or removed from the inventory over time.

Tasks:

Implement a Searching System:

- Choose a suitable data structure to organize the items efficiently.
- Implement an efficient method to retrieve an item based on its weight.
- If an exact match is not found, return the closest available item.

Problem 5

A **navigation system** is designed to help users find the best route between two locations while considering both **distance and toll costs**. The system represents roads as **edges** (with distances) and intersections as **nodes** (which may have toll fees). Some intersections have no toll fees (toll = 0), while others require payment.

Your task is to implement a system that calculates multiple route options for a user, allowing them to choose between:

- The shortest route (minimizing distance only).
- The most cost-effective route (minimizing toll fees).
- A balanced route that considers both distance and toll fees.

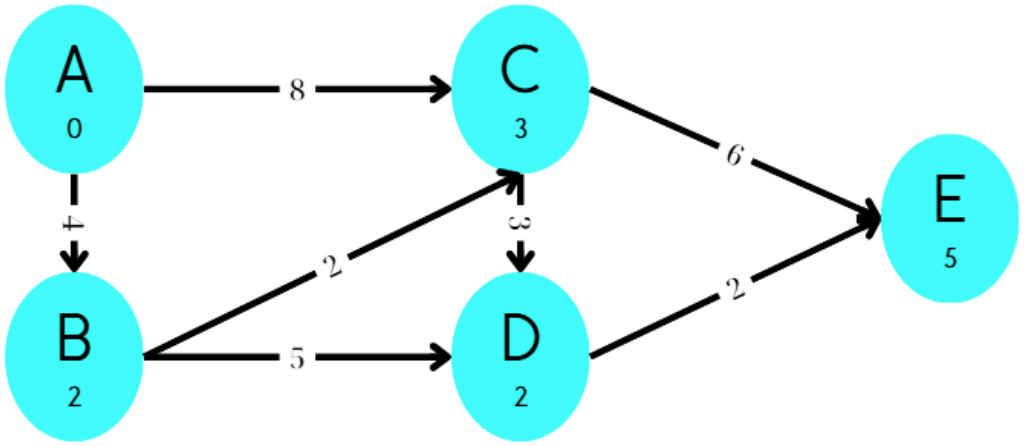
Additionally, routes that are **both long and expensive** should be automatically discarded.

Tasks:

Implement a Route Selection System:

1. Model the road network as a graph where:
    - **Edges** represent roads (with distances as weights).
    - **Nodes** represent intersections (with associated toll costs).
2. Implement **Dijkstra's Algorithm** to find the shortest path.
3. Modify the algorithm to account for toll fees and generate multiple route options.

***Note:** The example of the Map for input method:



Map = {

"A": {"toll": 0, "roads": {"B": 4, "C": 8}},

"B": {"toll": 2, "roads": {"A": 4, "C": 2, "D": 5}},

"C": {"toll": 3, "roads": {"A": 8, "B": 2, "D": 3, "E": 6}},

"D": {"toll": 2, "roads": {"B": 5, "C": 3, "E": 2}},

"E": {"toll": 5, "roads": {"C": 6, "D": 2}},

}