**Assignment 2**

**1.1-1**

**Describe your own real-world example that requires sorting. Describe one that requires finding the shortest distance between two points.**

Most times I need to make a decision on where am going to revise from whenever am at the main canteen at UCU after lunch on Mondays, so I have to choose between the computer lab and the library quickly. And so, to find the shortest distance. I consider the building entrances, avoiding stairs, avoiding a lot of rises and also finding a shortcut through open spaces. The shortest path helps me spend minimal time walking and optimizing my study time.

**1.1-2**

**Other than speed, what other measures of efficiency might you need to consider in a real- world setting?**

In a real-world setting, when evaluating the efficiency of an algorithm, several important measures beyond speed are considered and these include the following:

* Space Complexity:

Memory Usage: How much memory or storage an algorithm uses during execution is critical, especially in resource-constrained environments like embedded systems or mobile devices. An algorithm that is fast but uses excessive memory may not be practical.

* Energy Consumption:

Power Efficiency: Algorithms that run longer or require more processing can drain batteries faster, making energy efficiency a key metric.

* Scalability:

Handling Larger Inputs: The algorithm’s ability to scale with increasing input sizes is important. An efficient algorithm should not just handle small datasets but also scale to handle larger datasets without significant degradation in performance.

* Concurrency and Parallelism:

Threading/Parallel Processing: The ability of an algorithm to run in parallel or concurrently can impact its real-world efficiency. Algorithms that can be parallelized to take advantage of multi-core processors or distributed systems can improve overall performance and throughput.

* Accuracy and Precision:

Error Tolerance: Some algorithms, especially in real-world data processing e.g machine learning may trade off speed or simplicity for greater accuracy or precision.

* Robustness:

Handling Edge Cases: An efficient algorithm must handle a wide variety of inputs, including unexpected or extreme cases, without crashing or degrading significantly in performance.

Fault Tolerance: Algorithms that gracefully recover from failures or errors (e.g., network disruptions or memory allocation failures) are essential in real-world applications.

* Complexity of Implementation:

Maintainability: Some highly efficient algorithms may be difficult to implement or maintain. For instance, an algorithm that minimizes time complexity but is overly complex may introduce bugs or increase development and debugging time.

**1.1-3**

**Select a data structure that you have seen, and discuss its strengths and limitations**.

I have seen an array.

An array is a data structure that stores a fixed-size, ordered collection of elements and of the same data type.

**Strengths of an Array**

Some of the strengths of an array include the following;

1. **Direct Access (Constant Time Lookup)**: Arrays allow **O(1) time complexity** for accessing elements when the index is known. This makes it very efficient for accessing any item based on its position.
2. **Memory Efficiency**: Arrays store data contiguously in memory, which leads to efficient memory usage, especially for fixed-size data types.
3. **Simple Data Structure**: Arrays are easy to implement and understand, making them a fundamental data structure for beginners and a building block for more complex structures.
4. **Efficient Iteration**: Iterating through an array is straightforward and efficient, as elements are stored in adjacent memory locations. And many other strengths.

**Limitations of an Array**

The limitations of an array include**;**

**Fixed Size**: Arrays have a **fixed size**, meaning once they are created, their size cannot be changed. This can be inefficient if the size of the data is unknown or varies dynamically, leading to wasted memory or the need for resizing.

**Costly Insertion and Deletion**: Inserting or deleting elements, especially in the middle of an array, is inefficient. It requires shifting elements to maintain the contiguous memory layout, leading to **O(n)** time complexity for both operations. For example, inserting at the beginning of an array requires shifting all elements to the right.

**Memory Allocation Issues**: Arrays require a **contiguous block of memory**, which can be problematic in systems with fragmented memory, especially when dealing with large arrays.

**Homogeneous Data**: Arrays generally store **homogeneous** data types, meaning all elements must be of the same type. This can be limiting in languages that enforce type consistency, making them less flexible for varied data types.

**Inefficient Searching**: Searching for an element in an unsorted array has a time complexity of **O(n)**, as it requires a linear search. Even in sorted arrays, searching requires **O(log n)** time if binary search is used.

**Limited Flexibility**: Arrays cannot easily handle dynamic operations like concatenation, slicing, or advanced manipulations (e.g., adding/removing elements based on conditions) compared to higher-level data structures like lists, sets, or hash maps.

**1.1-4**

**How are the shortest-path and traveling- salesperson problems given above similar? How are they different?**

The **shortest-path problem** and the **traveling salesperson problem (TSP)** are both classic optimization problems in graph theory, but they differ in scope and complexity and some of their similarities and differences are as follows:

**Similarities:**

**Graph-Based Problems**: Both problems are modeled using graphs, where nodes (or vertices) represent locations (e.g., cities or points) and edges (or arcs) represent paths or connections between them with associated costs (e.g., distance or time).

**Goal of Finding Optimal Path**: Both problems involve finding an optimal path based on minimizing some cost (usually distance or time) between points on the graph.

**Weighted Graphs**: The edges in both problems typically have weights (e.g., distances, travel times, or costs), and the goal is to minimize the total weight of the selected path.

**Real-World Applications**: Both problems have practical applications in logistics, route planning, network optimization, and resource management. For example, they are useful for determining efficient routes for transportation or communication networks.

**Differences:**

**Problem Definition**:

**Shortest-Path Problem**: The objective is to find the shortest path between two specific points in a graph (e.g., from point A to point B). The goal is to minimize the distance or cost between these two points without any requirement to visit other nodes. While **Traveling Salesperson Problem (TSP)**: The goal is to find the shortest possible route that visits every node exactly once and returns to the starting node. It’s about visiting all points in the graph while minimizing the total travel cost.

**Scope of the Path**:

**Shortest-Path Problem**: Only involves two points — the start and end. There’s no requirement to visit all other points. While **TSP**: Requires visiting every point (node) exactly once, creating a **complete tour** of the graph.

**Complexity**:

**Shortest-Path Problem**: This is relatively easier to solve and can often be done in **polynomial time** using algorithms like Dijkstra’s or Bellman-Ford. While **TSP**: The TSP is an **NP-hard** problem, meaning there is no known algorithm that can solve it in polynomial time for all cases. Exact solutions for TSP require exponential time in the worst case, and approximation or heuristic algorithms are often used for large instances.

**Algorithms**:

**Shortest-Path Problem**: Algorithms like Dijkstra’s, A\*, or Bellman-Ford are commonly used, which can efficiently find the shortest path in polynomial time. While **TSP**: Exact solutions require more complex algorithms like **brute force (checking all permutations)** or **dynamic programming** (e.g., Held-Karp). Heuristic methods like **genetic algorithms** or **nearest neighbor** are often used to approximate the solution for larger graphs.

**Path Characteristics**:

**Shortest-Path Problem**: The path is **not necessarily a cycle** (does not need to return to the starting point), and it doesn’t visit every node. While **TSP**: The path must form a **cycle** (it must return to the starting point) and must visit all nodes exactly once.

**Applicability**:

**Shortest-Path Problem**: Useful in scenarios where you need to find the most efficient route between two specific locations (e.g., navigating from home to work). While **TSP**: Useful in scenarios where every location must be visited once (e.g., a delivery route where a driver must visit multiple cities and return to the starting point).

**1.1-5**

**Suggest a real-world problem in which only the best solution will do. Then come up with one in which “approximately” the best solution is good enough.**

Real-world problem where only the best solution will do:

**Aircraft Navigation and Safety Systems**: In aviation, flight navigation systems calculate the most efficient and safest route for airplanes, considering factors like weather, fuel efficiency, air traffic, and no-fly zones. The path chosen must ensure maximum safety and compliance with air traffic regulations. Any deviation or compromise on safety can lead to catastrophic consequences, so only the best solution, the safest, most precise route will do. Because Lives are at stake, and any error can lead to fatal outcomes

**“Approximately” the best solution is good enough.**

**Delivery Route Optimization for a Logistics Company**:

In logistics, companies often need to find efficient routes for delivery trucks to visit multiple destinations. While finding the **exact shortest route** (like in the Traveling Salesperson Problem) would be ideal, an **approximate solution** that is “good enough” and saves time or distance compared to manual planning can still lead to significant cost savings and efficiency gains.

**1.1-6**

**Describe a real-world problem in which sometimes the entire input is available before you need to solve the problem, but other times the input is not entirely available in advance and arrives over time.**

A **real-world problem** where sometimes the entire input is available beforehand, but other times the input arrives over time, is **online shopping inventory management and order fulfillment** for an e-commerce platform. And the following are the examples;

**Full Input Available**: A large retailer places an order for 1,000 units of various products for delivery next month. The e-commerce system has time to plan for the best way to fulfill this order, optimize for bulk shipping, and possibly reallocate inventory well in advance.

**Input Arrives Over Time**: On the same day, hundreds of individual customers place orders at different times for various products. As each new order arrives, the system must immediately determine how to fulfill it, without knowing how many more orders will come in later in the day. The system must adjust its strategy as new orders continue to flow in.