**Exercise 1: Inventory Management System**

**Scenario:**

You are developing an inventory management system for a warehouse. Efficient data storage and retrieval are crucial.

**Steps:**

1. **Understand the Problem:**
   * Explain why data structures and algorithms are essential in handling large inventories.
     + Data Structures and Algorithms are essential tools used for efficient storage and retrieval of data, therefore in case of an inventory which might contains thousands or maybe millions of products, a management system is a must for running a business smoothly. Some of the obvious reasons for the requirement of such a system are:-

* Data Structures allow **data storage** in an **efficient way** which utilizes memory effectively and provides easy access and modification facilities.
* Suitable Algorithms regulate **quick and efficient data retrieval** reducing the time complexity.
* An inventory is bound to grow, therefore choosing a **scalable** data structure and algorithms that can take care of data without significant increase in time and resource consumption is essential.
  + Discuss the types of data structures suitable for this problem.
    - For a system that is bound to grow exponentially and requires efficient data storage and retrieval following data structures and algorithms can be used:-
      * **Arraylist**: This provides easy and quick access to objects/elements by index but insertion and deletion operations are slow (O(n)) in the worst case, because elements may need to be shifted.
      * **LinkedList**: Although it is suitable for insertion and deletion, access time is slower than ArrayList (O(n)) for access.
      * **HashMap**: This is the **most suitable** data structure since the average time complexity for access, insertion and deletion is O(1). Accessing the products can be facilitated by using a unique key i.e. product\_ID.
      * **TreeMap**: This can be useful if ordered access to elements is required. Time complexity for insertion, deletion and access is O(log n).

1. **Setup:**
   * Create a new project for the inventory management system.
     + I will be using VS code for Inventory Management System where I have created two java files namely Product.java and InventoryManagementSystem.java for effective usage of the system.
2. **Implementation:**
   * Define a class Product with attributes like **productId**, **productName**, **quantity**, and **price**.
   * Choose an appropriate data structure to store the products (e.g., ArrayList, HashMap).
   * Implement methods to add, update, and delete products from the inventory.
3. **Analysis:**
   * Analyze the time complexity of each operation (add, update, delete) in your chosen data structure.
     + Add Product details : Time complexity for adding products – Average O(1), Worst Case O(n).

Update Product details: O(1) for retrieval and O(1) for update , therefore total O(1)

Delete Product details: O(1)

* + Discuss how you can optimize these operations.
    - Above operations can be optimized by using a goop hash function and rehashing when necessary to maintain efficiency.

**Exercise 2: E-commerce Platform Search Function**

**Scenario:**

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

**Steps:**

1. **Understand Asymptotic Notation:**
   * Explain Big O notation and how it helps in analyzing algorithms.
     + Asymptotic Notation is a mathematical way of representing the time complexity.
   * Describe the best, average, and worst-case scenarios for search operations.
     + Best Case: The minimum time an algorithm takes to complete, given the optimal conditions. For a search operation, the best case occurs when the desired element is the first element in the dataset.
     + Average Case: The expected time an algorithm takes to complete, given random input. This gives a more realistic measure of an algorithm's performance.
     + Worst Case: The maximum time an algorithm takes to complete, given the worst possible conditions. For a search operation, the worst case occurs when the desired element is the last element in the dataset or is not present at all.
2. **Setup:**
   * Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
   * Implement linear search and binary search algorithms.
   * Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
   * Compare the time complexity of linear and binary search algorithms.

Linear Search:

* + - Best Case: O(1) - The desired element is the first element.
    - Average Case: O(n) - The desired element is somewhere in the middle or not present.
    - Worst Case: O(n) - The desired element is the last element or not present.

Binary Search:

* + - Best Case: O(1) - The desired element is the middle element.
    - Average Case: O(log n) - The dataset is halved in each step.
    - Worst Case: O(log n) - The dataset is halved in each step, and the element is found in the last comparison or not present.
  + Discuss which algorithm is more suitable for your platform and why.
    - Linear Search is simple and does not require the dataset to be sorted. It is suitable for small datasets or unsorted data.
    - Binary Search is much faster for large datasets but requires the dataset to be sorted. It is suitable for large datasets where sorting can be maintained.

**Exercise 3: Sorting Customer Orders**

**Scenario:**

You are tasked with sorting customer orders by their total price on an e-commerce platform. This helps in prioritizing high-value orders.

**Steps:**

1. **Understand Sorting Algorithms:**
   * Explain different sorting algorithms (Bubble Sort, Insertion Sort, Quick Sort, Merge Sort).
     + Bubble Sort: Repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.
     + Insertion Sort: Builds the final sorted array one item at a time, placing each new item in its proper place among those already sorted.
     + Quick Sort: Divides the list into two smaller sub-lists based on a pivot element, recursively sorting the sub-lists.
     + Merge Sort: Divides the list into two halves, recursively sorts each half, and then merges the sorted halves.
2. **Setup:**
   * Create a class **Order** with attributes like **orderId**, **customerName**, and **totalPrice**.
3. **Implementation:**
   * Implement **Bubble Sort** to sort orders by **totalPrice**.
   * Implement **Quick Sort** to sort orders by **totalPrice**.
4. **Analysis:**
   * Compare the performance (time complexity) of Bubble Sort and Quick Sort.

**Bubble Sort:**

* + - Simple but inefficient for large datasets.
    - Time complexity: O(n^2) in the average and worst cases.

**Quick Sort:**

* + - Efficient and commonly used for large datasets.
    - Time complexity: O(n log n) in the average and best cases; O(n^2) in the worst case, but with a good pivot selection strategy, this can be minimized.
  + Discuss why Quick Sort is generally preferred over Bubble Sort.
    - Quick Sort is generally faster than Bubble Sort for large datasets due to its average-case time complexity of O(n log n).
    - Bubble Sort is primarily used for educational purposes and small datasets due to its simplicity but is inefficient for larger datasets due to its O(n^2) time complexity.

**Exercise 4: Employee Management System**

**Scenario:**

You are developing an employee management system for a company. Efficiently managing employee records is crucial.

**Steps:**

1. **Understand Array Representation:**
   * Explain how arrays are represented in memory and their advantages.
     + Representation in memory: Arrays are a collection of elements stored in contiguous memory locations. They are stored in a continuous block of memory, which makes accessing elements fast but resizing arrays expensive.
     + Advantages: Fast Access: Direct access to elements using indices. Memory Efficiency: Memory allocation is contiguous, reducing overhead. Cache-Friendly: Due to contiguous memory, it improves cache performance.
2. **Setup:**
   * Create a class Employee with attributes like **employeeId**, **name**, **position**, and **salary**.
3. **Implementation:**
   * Use an array to store employee records.
   * Implement methods to **add**, **search**, **traverse**, and **delete** employees in the array.
4. **Analysis:**
   * Analyze the time complexity of each operation (add, search, traverse, delete).
     + Each operation time complexity is O(n).
   * Discuss the limitations of arrays and when to use them.
     + Fixed Size: Initial size must be defined; resizing is costly.
     + Inefficient Deletion and Insertion: Shifting elements is required, leading to O(n) time complexity.
     + Wasted Space: If the array is not fully utilized, memory is wasted.
     + Use Cases: Suitable for small datasets with infrequent insertions/deletions.

**Exercise 5: Task Management System**

**Scenario:**

You are developing a task management system where tasks need to be added, deleted, and traversed efficiently.

**Steps:**

1. **Understand Linked Lists:**
   * Explain the different types of linked lists (Singly Linked List, Doubly Linked List).
     + Singly Linked List:

A singly linked list is a data structure consisting of nodes. Each node contains a data part and a reference (or link) to the next node in the sequence. The last node's reference is set to null, indicating the end of the list.

* + - Doubly Linked List:

A doubly linked list is similar to a singly linked list but with an additional reference. Each node contains a data part, a reference to the next node, and a reference to the previous node. This allows traversal in both directions (forward and backward).

1. **Setup:**
   * Create a class **Task** with attributes like **taskId**, **taskName**, and **status**.
2. **Implementation:**
   * Implement a singly linked list to manage tasks.
   * Implement methods to **add**, **search**, **traverse**, and **delete** tasks in the linked list.
3. **Analysis:**
   * Analyze the time complexity of each operation.
     + Each operation time complexity is O(n)
   * Discuss the advantages of linked lists over arrays for dynamic data.
     + Dynamic Size: Linked lists can easily grow and shrink in size by adding or removing nodes.
     + Efficient Insertions/Deletions: Adding or removing elements is more efficient in linked lists, as it doesn't require shifting elements (unlike arrays).
     + Memory Utilization: Linked lists use memory only for the elements they contain, which can be more efficient if the list changes frequently.

**Exercise 6: Library Management System**

**Scenario:**

You are developing a library management system where users can search for books by title or author.

**Steps:**

1. **Understand Search Algorithms:**
   * Explain linear search and binary search algorithms.
     + 1**. Linear Search**: Linear search is a simple algorithm where we can just iterate through each element until target element is found.

Time Complexity: O(n) for worst case and O(1) for best case.

* + - 2. **Binary Search**: It works only on sorted list of elements.

Time complexity: O(log n) worst case and O(1) for best case

1. **Setup:**
   * Create a class **Book** with attributes like **bookId**, **title**, and **author**.
2. **Implementation:**
   * Implement linear search to find books by title.
   * Implement binary search to find books by title (assuming the list is sorted).
3. **Analysis:**
   * Compare the time complexity of linear and binary search.
     + Binary search: O(log n)
     + Linear Search: O(n)
   * Discuss when to use each algorithm based on the data set size and order.
     + For small datasets or unsorted lists use linear search.
     + For larger datasets use binary search.

**Exercise 7: Financial Forecasting**

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

**Steps:**

1. **Understand Recursive Algorithms:**
   * Explain the concept of recursion and how it can simplify certain problems.
     + Recursion is a method where solution to a larger problem depends on the solutions of the smaller instances of the same problem. A recursive function requires a base case and a condition to operate. Recursion breaks a problem to smaller manageable parts.
2. **Setup:**
   * Create a method to calculate the future value using a recursive approach.
3. **Implementation:**
   * Implement a recursive algorithm to predict future values based on past growth rates.
4. **Analysis:**
   * Discuss the time complexity of your recursive algorithm.
     + Time complexity : O(n)
   * Explain how to optimize the recursive solution to avoid excessive computation.
     + For more complex problems we can use dynamic programming for optimization.