**Exercise 1: Inventory Management System**

1. **Understand the Problem:**

* Data structures and algorithms ensure efficient inventory data storage, retrieval, and manipulation in handling large inventories. Using efficient data structures can reduce time and space complexity, improving the overall performance of the system.
* Suitable data structures for this problem are:

ArrayList: Dynamic arrays which allow fast access by index.

HashMap: Fast access, insertion and deletion using key-value pairs.

1. **Analysis**:

* HashMap data structure is used in the code.
* Add product: O(1) on average
* Update product: O(1) on average
* Delete product: O(1) on average
* Using a HashMap provides constant time complexity for these operations, making it highly efficient for large inventories.

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

1. **Understand the Problem:**

* Big O notation describes the upper limit of an algorithm's running time, providing a measure of the worst-case scenario.
* - Best Case: The scenario where the operation takes the least time (e.g., finding the first element in a search).

- Average Case: The expected time complexity, considering all possible inputs.

- Worst Case: The scenario where the operation takes the most time (e.g., the element is not found).

1. **Analysis**:

* Time Complexity:
* Linear Search: O(n) on average
* Binary Search: O(logn) on average
* Binary Search is more suitable for this platform as it has a significantly lower time complexity (O(log n)) compared to linear search (O(n)), making it much faster for large datasets.

**Exercise 3: Sorting Customer Orders**

1. **Understand Sorting Algorithms:**

* **Bubble Sort** is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.
* **Insertion sort**is a simple sorting algorithm that works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list. It is a **stable sorting**algorithm, meaning that elements with equal values maintain their relative order in the sorted output.
* **QuickSort**is a sorting algorithm based on the Divide and Conquer algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.
* **Merge sort**is a sorting algorithm that follows the divide and conquer approach. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array.

1. Analysis:

* Time Complexity:
* Bubble Sort: O(n^2)
* Quick Sort: O(nlogn)
* Quick Sort generally outperforms Bubble Sort, especially for large datasets. The average-case time complexity of Quick Sort is O(n log n), significantly better than Bubble Sort's O(n^2).

**Exercise 4: Employee Management System**

1. **Understand Array Representation:**

Arrays are a collection of elements stored in contiguous memory locations. Each element in an array is of the same data type and can be accessed using its index. The memory address of an element can be calculated using the base address of the array and the size of each element.

Advantages:

* **Constant Time Access:** Accessing an element by its index is an O(1) operation.
* **Memory Efficiency:** Arrays have a low memory overhead since they don't store extra information about the elements.
* **Cache Friendly:** Due to their contiguous memory layout, arrays have better cache performance.

1. Analysis:

* Time Complexity:
* Add: O(1) (if space is available)
* Search: O(n)
* Traverse: O(n)
* Delete: O(n)
* Limitations of Arrays:
* Fixed Size
* Inefficient Insertions/Deletions
* When to Use Arrays:
* Static Data: When the size is known and doesn’t change frequently.
* Fast Access Required: When constant time access by index is needed.

**Exercise 5: Task Management System**

1. **Understand Linked Lists:**

* Types of Linked Lists:
* Singly Linked List: Each node points to the next node. Efficient for insertions/deletions at the head.
* Doubly Linked List: Each node points to both the next and previous nodes. Allows bidirectional traversal but uses more memory.

1. Analysis:

* Time Complexity:
* Add: O(1)
* Search: O(n)
* Traverse: O(n)
* Delete: O(n)
* Advantages of Linked Lists Over Arrays:
* Dynamic Size: Can grow/shrink as needed.
* Efficient Insertions/Deletions: O(1)O(1)O(1) for operations at the head.
* Memory utilization: Memory is allocated as needed.

**Exercise 6: Library Management System**

1. **Understand Search Algorithms:**

* Linear Search:
* Algorithm: Traverse each element in the list and compare it with the target element.
* Time Complexity:

Best Case: O(1) (target is the first element).

Average Case: O(n) (target is in the middle).

Worst Case: O(n) (target is the last element or not present).

* Space Complexity: O(1).
* Binary Search:
* Algorithm: Repeatedly divide the sorted list into halves and compare the middle element with the target.
* Time Complexity:

Best Case: O(1) (target is the middle element).

Average Case: O(log n).

Worst Case: O(log n).

* Space Complexity: O(1).

1. Analysis:

* Time Complexity:
* Linear Search: O(n)
* Binary Search)O(logn)
* Algorithm Suitability:
* Linear Search: Suitable for small or unsorted datasets.
* Binary Search: Suitable for large, sorted datasets.

**Exercise 7: Financial Forecasting**

1. **Understand Recursive Algorithms:**

A method of solving a problem where the solution involves solving smaller instances of the same problem.

How Recursion Simplifies Problems:

* + Divide and Conquer: Breaks down complex problems into simpler sub-problems.
  + Elegant Solutions: Often leads to more readable and shorter code for problems like tree traversal, factorial calculation, and Fibonacci series.

1. Analysis:

* Time Complexity: O(n), where n is the number of periods. The function calls itself once for each period, leading to linear time complexity.
* Optimizing the Recursive Solution:
* Memoization: Store results of previously computed sub-problems to avoid redundant calculations.
* Iterative Approach: Convert the recursive solution into an iterative one to eliminate the overhead of recursive function calls.