**Exercise 1: Inventory Management System**

**Understanding the Problem:**

**Why are data structures and algorithms essential in handling large inventories?**

* **Efficiency**: They help in fast data storage and retrieval, which is crucial for managing large inventories.
* **Performance**: Good algorithms make operations like adding, updating, and deleting products faster.
* **Scalability**: They ensure the system works well as the inventory grows.

**Types of data structures suitable for this problem**

* **ArrayList**: Good for dynamic resizing and quick access to elements.
* **HashMap**: Ideal for fast lookups using product IDs.

**Analysis:**

**Time complexity of operations in HashMap**

* **Add**: O(1) on average.
* **Update**: O(1) on average.
* **Delete**: O(1) on average.

**Optimization Strategies:**

* **Rehashing**: Keep the load factor low to avoid collisions.
* **Good Hash Functions**: Distribute entries evenly.
* **Data Partitioning**: Use multiple HashMaps for very large inventories.

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

**Understanding Asymptotic Notation**

**Big O notation**

* **Big O notation**: Describes how an algorithm’s time complexity grows with input size.
* **Importance**: Helps predict how an algorithm will perform with large data.

**Best, average, and worst-case scenarios for search operations**

* **Linear Search**:
  + Best: O(1).
  + Average: O(n).
  + Worst: O(n).
* **Binary Search**:
  + Best: O(1).
  + Average: O(log n).
  + Worst: O(log n).

**Analysis**

**Time complexity comparison**

* **Linear Search**: O(n).
* **Binary Search**: O(log n).

**Suitable Algorithm**

* **Binary Search** is better for large, sorted data because it’s faster.

**Exercise 3: Sorting Customer Orders**

**Understanding Sorting Algorithms**

**Sorting algorithms**

* **Bubble Sort**: Simple but slow, O(n²).
* **Insertion Sort**: Good for small lists, O(n²) in average and worst cases.
* **Quick Sort**: Fast, O(n log n) on average.
* **Merge Sort**: Stable and fast, O(n log n).

**Analysis**

**Performance comparison**

* **Bubble Sort**: O(n²).
* **Quick Sort**: O(n log n) on average.

**Why Quick Sort?**

* **Quick Sort** is faster and more efficient for large datasets compared to Bubble Sort.

**Exercise 4: Employee Management System**

**Understanding Array Representation**

**Array representation in memory**

* **Contiguous memory**: Arrays are stored next to each other, making access quick.
* **Constant time access**: Accessing elements by index is fast, O(1).

**Analysis**

**Time complexity of operations**

* **Add**: O(1) if adding at the end.
* **Search**: O(n).
* **Traverse**: O(n).
* **Delete**: O(n) because elements need to be shifted.

**Limitations of arrays**

* **Fixed size**: You can’t change the size once created.
* **Inefficient insertions/deletions**: These operations are slow because elements need to be moved.

**Exercise 5: Task Management System**

**Understanding Linked Lists**

**Types of linked lists**

* **Singly Linked List**: Each node points to the next one.
* **Doubly Linked List**: Nodes point to both the next and previous nodes.

**Analysis**

**Time complexity of operations in a singly linked list**

* **Add**: O(1) at the beginning, O(n) at the end.
* **Search**: O(n).
* **Traverse**: O(n).
* **Delete**: O(n).

**Advantages over arrays**

* **Dynamic size**: Linked lists can grow and shrink.
* **Efficient insertions/deletions**: No need to shift elements.

**Exercise 6: Library Management System**

**Understanding Search Algorithms**

**Search algorithms**

* **Linear Search**: Checks each element one by one, O(n).
* **Binary Search**: Faster on sorted lists, O(log n).

**Analysis**

**Time complexity comparison**

* **Linear Search**: O(n).
* **Binary Search**: O(log n).

**When to use each algorithm**

* **Linear Search**: Good for small or unsorted lists.
* **Binary Search**: Best for large, sorted lists.

**Exercise 7: Financial Forecasting**

**Understanding Recursive Algorithms**

**Concept of recursion**

* **Recursion**: A function calls itself to solve smaller parts of the problem.
* **Advantages**: Simplifies problems and makes code easier to read.

**Analysis**

**Time complexity of the recursive algorithm**

* **Basic Recursion**: Can be O(2^n) if not optimized.
* **Optimized Recursion**: Use memoization or dynamic programming to make it O(n).

**Optimization strategies**

* **Memoization**: Store results to avoid recalculating them.
* **Dynamic Programming**: Solve subproblems iteratively to reuse results.