1. **Suitable Data Structures for Inventory Management**

**Hash Tables**

* **Advantages**: Fast access, insertion, and deletion (O(1) on average).
* **Disadvantages**: Poor worst-case performance, requires good hash functions.
* **Use Case**: Large inventories needing fast operations.

1. **Big O Notation and Its Importance**

Big O Notation describes an algorithm's efficiency in terms of time complexity and space complexity, providing an upper bound on time or space required as input size grows, which helps compare and evaluate algorithm performance.

* **Time Complexity**: Amount of time an algorithm takes as a function of input size (n).
* **Space Complexity**: Amount of memory an algorithm uses as a function of input size (n).

**Best, Average, and Worst-Case Scenarios for Search Operations**

* **Linear Search**:
  + **Best Case (O(1))**: Target element is at the first position.
  + **Average Case (O(n))**: Target element is somewhere in the middle.
  + **Worst Case (O(n))**: Target element is at the last position or not present.
* **Binary Search (on a sorted array)**:
  + **Best Case (O(1))**: Target element is at the middle of the array.
  + **Average Case (O(log n))**: Target element is somewhere in the array, requiring halving the array log n times.
  + **Worst Case (O(log n))**: Target element is not present, requiring elimination of half of the remaining elements at each step until the search space is empty

1. **Different Sorting Algorithms**

**Bubble Sort**

* **Description**: A simple comparison-based sorting algorithm that repeatedly compares and swaps adjacent elements until the list is sorted.
* **Time Complexity**:
  + **Best Case**: O(n) (array is already sorted)
  + **Average Case**: O(n^2)
  + **Worst Case**: O(n^2)
* **Space Complexity**: O(1) (in-place sorting)
* **Advantages**: Simple to understand and implement.
* **Disadvantages**: Inefficient for large datasets due to quadratic time complexity.

**Quick Sort**

* **Description**: An efficient sorting algorithm that uses a divide-and-conquer approach, selecting a pivot element and partitioning the array into two sub-arrays.
* **Time Complexity**:
  + **Best Case**: O(n log n)
  + **Average Case**: O(n log n)
  + **Worst Case**: O(n^2) (poor pivot selection)
* **Space Complexity**: O(log n) (recursive stack space)
* **Advantages**: Very efficient for large datasets.
* **Disadvantages**: Poor worst-case performance, mitigated by good pivot selection strategies.

1. **Array Representation in Memory**

* **Memory Representation**: Arrays are stored in contiguous memory locations, allocated as a single block of memory.
* **Indexing**: Each element can be accessed directly using its index (0-based).
* **Address Calculation**: The address of an element can be calculated using the base address and the element size. For an array arr with base address B, the address of the i-th element is calculated as B + i \* size\_of\_one\_element.

**Advantages of Arrays**

* **Direct Access**: O(1) time complexity for element access using index.
* **Cache-Friendly**: Contiguous storage provides better CPU cache utilization.
* **Ease of Use**: Simple initialization and iteration.
* **Memory Efficiency**: Minimal overhead, predictable memory usage.
* **Sorting and Searching**: Compatible with various algorithms, effective for known and fixed array sizes.

**Limitations of Arrays**

* **Fixed Size**: Cannot be dynamically resized (in static arrays).
* **Insertion and Deletion**: Inefficient due to element shifting (O(n) time complexity).
* **Memory Allocation**: Large arrays require contiguous memory, which may be unavailable.

1. **Linked Lists**

**Singly Linked List**

* 1. **Structure**: Each node contains data and a next pointer.
  2. **Traversal**: Forward only, from head to end.
  3. **Advantages**: Memory efficient, simpler operations.
  4. **Disadvantages**: Unidirectional traversal, no backward traversal.
  5. **Example**: Head -> [Data | Next] -> [Data | Next] -> [Data | Next] -> null

**Doubly Linked List**

* 1. **Structure**: Each node contains data, next pointer, and prev pointer.
  2. **Traversal**: Bidirectional, forward and backward.
  3. **Advantages**: Easier node deletion, bidirectional traversal.
  4. **Disadvantages**: Higher memory usage, more complex implementation.
  5. **Example**: null <- [Prev | Data | Next] <-> [Prev | Data | Next] <-> [Prev | Data | Next] -> null

1. **SEARCHING**

**Linear Search**

* **Concept**: Checks each element sequentially until the target element is found or the end is reached.
* **Algorithm**: Compare elements one by one with the target; return the index if found, otherwise -1.
* **Time Complexity**:
  + **Best Case**: O(1) (first element match)
  + **Average Case**: O(n)
  + **Worst Case**: O(n) (last element or not present)
* **Advantages**: Simple, works on any list.
* **Disadvantages**: Inefficient for large lists.

**Binary Search**

* **Concept**: Efficiently finds an element in a sorted list by repeatedly halving the search interval.
* **Algorithm**: Compare the middle element with the target; adjust search interval accordingly; return the index if found, otherwise -1.
* **Time Complexity**:
  + **Best Case**: O(1) (middle element match)
  + **Average Case**: O(log n)
  + **Worst Case**: O(log n)
* **Advantages**: Efficient for large sorted lists.
* **Disadvantages**: Requires sorted lists.

**7.RECURSION**

**The Concept of Recursion**

* **Definition**: A technique where a function calls itself to solve a problem.
* **Key Components**:
  + **Base Case**: Stops recursion, provides a solution to the simplest problem instance.
  + **Recursive Case**: Calls the function with a modified argument towards the base case.

**How Recursion Simplifies Problems**

* **Divide and Conquer**: Breaks problems into smaller sub-problems, effective for recursive problems.

**Advantages of Recursion**

* **Simplified Code**: Easier to understand and maintain.
* **Natural Fit**: Ideal for inherently recursive problems.
* **Modularity**: Promotes code reuse through smaller problem instances.

**Disadvantages of Recursion**

* **Performance**: Less