**Week – 1 (DSA)**

**EX-1**

**Explain why data structures and algorithms are essential in handling large inventories.**

Efficient data storage and retrieval are crucial in handling large inventories because it ensures quick access to product information, smooth operation of the warehouse, and timely updates to the inventory.

Proper data structures can handle large amounts of data effectively, enabling faster search, insert, update, and delete operations, which are critical in maintaining an accurate inventory.

**Discuss the types of data structures suitable for this problem.**

**ArrayList:** Provides dynamic arrays that can grow as needed. Suitable for a small to moderate number of products where random access is needed.

**HashMap:** Provides an efficient way to store key-value pairs, allowing for fast retrieval based on unique keys (productId). This is suitable for larger inventories due to its average constant-time complexity for most operations.

**Analyze the time complexity of each operation (add, update, delete) in your chosen data structure.**

**Add Product:** O(1) on average since HashMap insertion is constant time.

**Update Product:** O(1) on average because accessing an element by key in a HashMap is constant time.

**Delete Product:** O(1) on average since removing an element by key is constant time.

**Get Product:** O(1) on average for retrieving an element by key.

**EX-2**

**Explain Big O notation and how it helps in analyzing algorithms.**

Big O notation is a mathematical representation used to describe the upper bound of an algorithm's running time or space complexity in terms of input size. It helps in understanding the efficiency and scalability of an algorithm.

**Describe the best, average, and worst-case scenarios for search operations.**

**Best Case:** The scenario where the algorithm performs the minimum number of operations (e.g., the target element is the first element in the list for linear search).

**Average Case:** The scenario where the algorithm performs a moderate number of operations, averaged over all possible inputs.

**Worst Case:** The scenario where the algorithm performs the maximum number of operations (e.g., the target element is not in the list for linear search).

**Compare the time complexity of linear and binary search algorithms.**

**Linear Search:** O(n) for best, average, and worst-case scenarios.

**Binary Search:** O(log n) for best, average, and worst-case scenarios, assuming the array is sorted.

**Discuss which algorithm is more suitable for your platform and why.**

More efficient for larger datasets due to its logarithmic time complexity. Requires the array to be sorted, which adds an initial O(n log n) sorting overhead but provides faster search performance afterward.

**EX-3**

**Explain different sorting algorithms (Bubble Sort, Insertion Sort, Quick Sort, Merge Sort).**

**Bubble Sort:**

* Description: A simple comparison-based sorting algorithm where each pair of adjacent elements is compared, and elements are swapped if they are in the wrong order. This process is repeated until the list is sorted.
* Time Complexity: O(n^2) in the average and worst-case scenarios.
* Use Case: Suitable for small datasets or nearly sorted arrays due to its simplicity.

**Insertion Sort:**

* Description: Builds the final sorted array one item at a time by repeatedly picking the next item and inserting it into the correct position.
* Time Complexity: O(n^2) in the average and worst-case scenarios.
* Use Case: Efficient for small or nearly sorted datasets.

**Quick Sort:**

* Description: A divide-and-conquer algorithm that picks a pivot element and partitions the array into two sub-arrays (elements less than the pivot and elements greater than the pivot). It then recursively sorts the sub-arrays.
* Time Complexity: O(n log n) on average and O(n^2) in the worst case.
* Use Case: Generally preferred for large datasets due to its efficient average-case performance.

**Merge Sort:**

* Description: A divide-and-conquer algorithm that divides the array into two halves, recursively sorts them, and then merges the sorted halves.
* Time Complexity: O(n log n) in all cases.
* Use Case: Suitable for large datasets and for external sorting (sorting data that doesn't fit into memory).

**Compare the performance (time complexity) of Bubble Sort and Quick Sort.**

**Bubble Sort:** O(n^2) for the average and worst-case scenarios. It performs poorly on large datasets due to its quadratic time complexity.

**Quick Sort:** O(n log n) on average, but O(n^2) in the worst case (when the pivot selection is poor). However, with good pivot selection (e.g., using the median or random pivot), it performs efficiently.

**Discuss why Quick Sort is generally preferred over Bubble Sort.**

Quick Sort is generally preferred over Bubble Sort for large datasets because of its average-case time complexity of O(n log n), making it significantly faster than Bubble Sort's O(n^2).

Quick Sort is more efficient in practice, even though it has a worst-case complexity of O(n^2), because it can be optimized with good pivot selection strategies.

**EX-4**

**Explain how arrays are represented in memory and their advantages.**

**Array Representation in Memory:**

* Arrays are a collection of elements stored in contiguous memory locations.
* The base address (address of the first element) allows direct access to any element using an index.
* **Advantages:**
  + Constant-time complexity (O(1)) for accessing elements by index.
  + Predictable memory layout, which can be efficiently managed by the system.
  + Suitable for fixed-size collections where the number of elements is known in advance.

**Analyze the time complexity of each operation (add, search, traverse, delete).**

**Add:** O(1) - Adding an employee to the array is constant time until the array is full.

**Search:** O(n) - In the worst case, you may need to search through the entire array.

**Traverse:** O(n) - You need to iterate through the array to traverse all employees.

**Delete:** O(n) - In the worst case, you need to search through the entire array to find the employee to delete.

**Discuss the limitations of arrays and when to use them.**

Fixed size: Once an array is created, its size cannot be changed. This can be limiting if the number of employees is not known in advance.

Inefficient for insertions and deletions: Inserting or deleting elements can be expensive, especially if it involves shifting elements.

Not suitable for dynamic datasets.

**EX-5**

**Explain the different types of linked lists (Singly Linked List, Doubly Linked List).**

**Singly Linked List:**

* A singly linked list consists of nodes where each node contains a data part and a reference (or link) to the next node in the sequence.
* **Advantages:**
  + Dynamic size: Can grow and shrink during runtime.
  + Efficient insertions and deletions: Adding or removing elements does not require shifting elements as in arrays.

**Doubly Linked List:**

* A doubly linked list contains an additional reference to the previous node, allowing traversal in both directions.
* **Advantages:**
  + Easier backward traversal.
  + More flexibility in certain operations, such as deletion of a given node.

**Analyze the time complexity of each operation.**

**Add:** O(n)

**Search:** O(n)

**Traverse:** O(n)

**Delete:** O(n)

**Discuss the advantages of linked lists over arrays for dynamic data.**

Dynamic Size

Efficient Insertions/Deletions

Memory Utilization

**EX-6**

**Explain linear search and binary search algorithms.**

**Linear Search:**

* A simple search algorithm that checks each element in the list until the desired element is found or the list ends.

**Binary Search:**

* A more efficient search algorithm that works on sorted lists. It repeatedly divides the search interval in half.

**Compare the time complexity of linear and binary search.**

**Linear Search:**

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

**Binary Search:**

* **Best Case:** O(1) - The element is at the middle position.
* **Average Case:** O(log n) - The element is somewhere in the list.
* **Worst Case:** O(log n) - The element is not present.

**Discuss when to use each algorithm based on the data set size and order.**

**Linear Search:** Suitable for small or unsorted data sets. It does not require the data to be sorted.

**Binary Search:** Suitable for large, sorted data sets. It is much faster than linear search for large lists due to its logarithmic time complexity.

**EX-7**

**Explain the concept of recursion and how it can simplify certain problems.**

Recursion is a technique where a function calls itself directly or indirectly to solve a problem.

It can simplify problems by breaking them down into smaller, more manageable sub-problems.

**Discuss the time complexity of your recursive algorithm.**

The time complexity of a naive recursive approach can be exponential **(O(2^n))** if it involves overlapping sub-problems.