**DN 3.0 - EXERCISES**

**WEEK 1**

**ALGORITHMS & DATA STRUCTURES**

**Inventory Management System**

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

In inventory management systems, efficient data storage and retrieval is essential, especially in large warehouses with huge product volumes. The responsiveness, scalability, and performance of a system are enhanced by optimal data structures and algorithms. They optimise memory usage when working with large datasets, expedite tasks like locating, adding, updating, and removing products, maintain data integrity to reduce loss or duplication, and facilitate intricate operations like sorting and filtering.

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

ArrayList is less effective for frequent insertions and deletions but works well for dynamic arrays that are simple to access and modify. HashMap is perfect for quick searches and changes because it provides quick access to key-value pairs. LinkedList performs best in situations when there are a lot of additions and deletions, but access may be slower. When it comes to getting items back in a sorted order, TreeMap performs.

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

Add Operation: O(1) time complexity. This means that, as long as the hash function functions properly, adding a new product is rapid and effective since HashMap can handle insertions in constant time. Update Operation: O(1) time complexity also applies to this. Because updating a product just entails replacing the current entry, it may be completed just as quickly as adding one. Delete Operation: Using a key to remove a product also takes O(1) time. HashMap locates and removes objects quickly while preserving a consistent time complexity.

**4. Discuss how you can optimize these operations.**

•Effective Hashing: To minimise collisions and maintain speedy operations, ensure that the hash function distributes products equally throughout the table.

• Concurrent Access: To safely and effectively handle concurrent access in multi-threaded situations, use ConcurrentHashMap.

• Load Factor Management: To maximise memory use and performance, balance the load factor. While a low load factor may result in memory waste, a high load factor may lead to collisions.

• Data Validation: To guarantee consistency and prevent errors, data should always be validated before adding or modifying products.

**E-commerce Platform Search Function**

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

Big O notation is a mathematical technique for describing the upper bound of an algorithm's temporal complexity. It explains how the runtime or space requirements of an algorithm increase as the input size grows. Big O notation depicts an algorithm's efficiency and scalability by focussing on the most important factors while ignoring constants and lower-order terms.

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

* Best Case: The scenario where the algorithm performs the least lines of code to get the optimised solution.
* Average Case: The typical scenario providing an average lines of code under normal conditions.
* Worst Case: The scenario with the most lines of code, which is very lengthy and cannot be optimised properly.

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

Linear Search: Time Complexity is O(n). It goes through each product one by one until it finds the target or completes the list. Best Case: O(1), if the product is at the start. Average and worst cases: O(n) if the product is in the centre, at the end, or not present.

Binary Search: Time complexity: O(logn). It repeatedly divides a sorted list in half to locate the target rapidly. The best case is O(1), if the product is in the middle. Average and worst case: O(log n), due to the search space being halved.

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

Binary Search is more efficient for large datasets since its time complexity is O(log n), which is faster than linear search's O(n). However, binary search requires that the array be sorted. If the product list is often changed, the sorting overhead needs to be considered.

**Sorting Customer Orders**

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

• Bubble Sort: This fundamental sorting method compares and swaps neighbouring components until the list is sorted.

• Insertion Sort: Creates a sorted list one item at a time. It is less efficient for huge lists than more complex algorithms such as quicksort or merge sort.

• Quick Sort is a divide-and-conquer method that selects a pivot, separates the array into smaller sections, and sorts them recursively.

• Merge Sort divides the list into single-element sublists and merges them back together to create a sorted list.

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

• Bubble Sort has a time complexity of O(n2) in the worst and average scenarios. The quadratic time complexity makes it inefficient for large datasets.

• Quick Sort has an average time complexity of O(n log n) and a worst-case of O(n2). However, the worst-case scenario can be avoided by using appropriate pivot selection methods, such as random pivot selection.

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

Quick Sort's average-case performance of O(n log n) outperforms Bubble Sort's O(n²) and works well with real-world data.

**Employee Management System**

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

A contiguous block of elements, each uniquely identified by an index, is stored in memory as an array, a fundamental data structure in programming. They are effective for retrieving and iterating over data because they offer direct access to elements via indexes. However, when working with dynamic data, arrays' constant size may be a drawback.

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

**Time Complexity Analysis:**

• Adding an employee takes O(1) if the array has enough space, and O(n) if resizing is necessary.

• Search Employee: O(n) for a linear search using employeeId.

• Use O(n) to visit all employees.

• Delete Employee takes O(n) due to the necessity to rearrange elements after deletion.

**Limitations of Arrays:**

• Arrays have a fixed capacity, which can be a constraint when the number of employees increases beyond the initial size.

• Inefficient insertion and deletion: Shifting items in the middle can be time-consuming.

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

Arrays are useful when the number of elements is known ahead of time and does not change significantly. When the number of elements varies, ArrayLists or LinkedLists are more efficient.

**Task Management System**

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

• In a singly linked list, each node has data and a reference to the next node. It's simple, but only allows for one-way movement.

• In a doubly linked list, each node contains data and pointers to previous and next nodes. It supports two-way traversal but requires more RAM.

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

* Add Task: O(n) for appending at the end.
* Search Task: O(n) for traversing the list to find a specific task.
* Traverse Tasks: O(n) to go through each task.
* Delete Task: O(n) for finding and removing a specific task.

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

• Dynamic Size: While arrays have a fixed size, linked lists can expand and contract on a dynamic basis.

• Ease of Insertion/Deletion: Unlike arrays, linked lists do not require shifting items in order to add or remove elements. Because of this, linked lists handle insertions and deletions more efficiently.

**Library Management System**

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

• Linear Search: Searches a list iteratively through each element until the required element is located or the list is exhausted. Its temporal complexity is O(n).

• Binary Search: By periodically halving the search interval, this method effectively searches through a sorted list. Its temporal complexity is O(log n).

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

• Linear Search: O(n) - The method searches through the entire list of books one at a time until it either discovers the required book or runs out of space.

• Binary Search: O(log n) - To minimise the number of comparisons, the list must first be sorted. Then, the search splits the list in half each time.

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

• Linear Search: Good when the list is tiny or unsorted because it eliminates the need for sorting. It is simple and independent of the elements' sequence.

• Binary Search: This method drastically cuts down on search time and is more effective for large, sorted lists. But before doing a binary search, the list needs to be sorted, which adds overhead if it hasn't previously been done.