**Exercise-1**

1.why data structures and algorithms are essential in handling projects.

* Data structures and algorithms in projects are important because they allow for effective storing and retrieving of data, a process needed when there are thousands or millions of products.
* They enable quick searching, efficient sorting, and prompt updating of stock items, while also conserving memory and processing time, leading to improved system performance

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

* ArrayList: Ideal for maintaining order and quick index access.
* HashMap: Best for fast product ID lookups.
* TreeMap: Useful for keeping products sorted by attributes.

3. Analysis

* Add O(1) average case, O(n) worst case
* Update : O(1) average case, O(n) worst case
* Delete: O(1) average case, O(n) worst case
* Get:O(1) average case, O(n) worst case

4.Optimizations

* Good Hash Function: Java’s HashMap already uses an effective hash function, but if implementing your own, ensure it distributes hash values well to avoid collisions.
* Initial Capacity: Set an initial capacity for HashMap if the approximate number of products is known, to prevent the need for rehashing.
* Load Factor: Adjust the load factor to balance between memory usage and performance.
* Concurrent Access: Use ConcurrentHashMap to ensure thread safety when multiple threads need to access the inventory simultaneously.
* Caching: Implement caching to quickly return frequently requested products, reducing retrieval time.
* Batch Operations: Implement batchAdd, batchUpdate, or batchDelete methods for efficient handling of bulk product updates.

**Exercise 2**

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

* Big O notation is actually a mathematical notation defining the performance or complexity of an algorithm; it is truly a worst-case description of the maximum time to execute.

Big O notation analyzes the algorithms in two ways:

* Firstly, it provides the standard way to compare algorithm efficiency.
* It means focusing only on the scalability of algorithms as input size grows.

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

For search operations:

* Best-case scenario: O(1) — The item is found immediately.
* Average-case scenario:

Linear search: O(n)

Binary search: O(log n)

* Worst-case scenario:

Linear search: O(n) — The item is last or not in the list.

Binary search: O(log n) — The item is at the end of the search or not present.

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

* Time Complexity Comparison:
  + Linear Search: O(n)
  + Binary Search: O(log n)
* Binary search is generally more efficient, especially for large datasets. However, it requires the data to be sorted.

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

In an e-commerce web application, this decision depends on the following factors.

* In an e-commerce application, the choice between linear and binary search depends on several factors:
* Dataset Size: Linear search suits small datasets; binary search is better for larger ones.
* Update Frequency: Frequent updates make binary search less efficient due to the need to maintain sorted data.
* Search Criteria: Binary search is ideal for exact matches, while complex searches may need advanced algorithms or indexing.
* Memory Constraints: Binary search requires the entire dataset in memory, which can be impractical for large catalogs.
* Implementation Complexity: Linear search is simpler to implement and maintain.

**Exercise 3**

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

Sorting algorithms are methods of arrangement of a list either in increasing or decreasing

order. The following is a small description of each algorithm asked for:

* Bubble Sort: A basic sorting algorithm that repeatedly goes through the list, comparing the adjacent elements and swapping them if they are in the wrong order; its worst-case and average time complexity are O(n^2).
* Insertion sort: This algorithm constructs the final sorted array one item at a time, scanning the input elements and inserting them in the correct position of the already constructed array. Its time complexity is O(n^2).
* Quick Sort: This approach belongs to divide-and-conquer strategies. The method picks up an element of an array as a pivot and partitions the given array around the picked-up pivot, afterward sorting created sub-arrays during the partitioning process. Average case O(n log n) time complexity.
* Merge Sort: The other using divide-and-conquer. This technique divides the array into two halves, recursively sorting them and then merging the two sorted halves. Its time complexity is O(n log n).

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

Time Complexity:

* Bubble Sort: O(n^2) in all cases.
* Quick Sort: Average and best case O(n log n), worst case O(n^2).

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

Quick Sort is generally preferred over Bubble Sort for the following reasons:

* Better average-case performance: O(n log n) vs O(n^2).
* In-place sorting: Quick Sort typically uses less additional memory.
* Cache efficiency: Quick Sort's partitioning approach often works well with modern CPU caches.
* Adaptability: Quick Sort can be easily modified for different data types and criteria.

**Exercise 4**

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

* Arrays are single, contiguous blocks of memory where all elements are located consecutively.
* Each element is directly accessible via any index.

Advantages of arrays:

* Access time is fast, O(1), if elements are accessed by their index, which is known.
* Very simple and easy to use.
* It is memory efficient for the storage of primitive data types.
* Good cache locality, which can potentially improve performance.

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

Time Complexity:

* Add: O(1) - Adding at the end of the array is constant time.
* Search: O(n) - In the worst case, we need to traverse the entire array.
* Traverse: O(n) - We iterate through all elements.
* Delete: O(n) - We need to find the element and then shift the remaining elements.

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

Limitations:

* Fixed size: Once declared, the size of an array cannot be changed.
* Inefficient insertion and deletion: Specifically, these operations can be very pricey for big arrays.
* Wasted memory: Memory is wasted if the array is not full, by the unused slots.
* Lack of built-in features: The arrays do not have built-in functions for performing some complex operations.

When to use:

* This is the case when you know in advance how many elements you will have.
* This comes in handy when you need to quickly access elements via their index.
* If you want to save memory for simple data types when efficiency in memory usage matters.
* This is where you use basic data structures such as stacks or queues.
* When cache performance matters as a result of contiguous memory allocation.

**Exercise 5**

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

Types of Linked Lists:

* Singly Linked List: Every node contains data with a reference to the next node.
* Doubly Linked List: In such a list, every node is imbued with data, having a reference to the next node, and another to the previous node.

2. Analyze the time complexity of each operation.

Time Complexity:

* Add: O(n) - We need to traverse to the end of the list to add a new task.
* Search: O(n) - In the worst case, we need to traverse the entire list.
* Traverse: O(n) - We iterate through all elements.
* Delete: O(n) - We need to find the element, which could be at the end of the list.

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

Advantages linked lists offer over arrays when it comes to dynamic data include:

* Dynamic size: Linked lists have the capability of dynamic changeable size while executing, unlike arrays, which hold fixed sizes at the time of declaration; this is achieved with memory allocation and deallocation.
* Efficient Insertion and Deletion: Insertion or deletion of an element (especially at the start) is done more efficiently, O(1), in the case of insertion or deletion at the beginning or end after maintaining the tail pointer.
* No memory wastage: The amount of memory used by a linked list is the exact amount needed for the data.
* Some data structures, like lists, stacks, and queues, can be implemented more easily from a linked list.

**Exercise 6**

1. Explain linear search and binary search algorithms.

Linear Search:

* This method involves checking all elements of the list one by one until a match is found or the end is reached.
* Works on both sorted and unsorted lists.
* Time complexity: O(n).

Binary Search:

* It involves repeatedly dividing the search interval in half.
* Requires the list to be sorted.
* Time complexity: O(log n).

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

Time Complexity:

* Linear Search: O(n) - In the worst case, we need to traverse the entire list.
* Binary Search: O(log n) - We repeatedly divide the search space in half.

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

When to Use Each Algorithm:

Linear Search:

* The list is unsorted.
* List is small, normally less than 100 elements.
* Searching is done infrequently.
* Overhead of sorting for binary search is not justified.

Binary search:

* List is sorted or can be sorted once and used for multiple searches.
* List is large; normally more than 100 elements.
* Search operations are frequent.
* This occurs when the time saved in searching outweighs that spent sorting.

**Exercise 7**

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

* Recursion is the problem-solving technique whereby a function calls itself to solve smaller instances of the same problem. More specially, it is extremely helpful when a problem can be broken into smaller subproblems that are self-similar.
* Factorials, for example, can be calculated using recursion. Factorial of a number n is defined as the number n multiplied by the factorial of n-1.

2. Discuss the time complexity of your recursive algorithm.

Time Complexity: This recursive algorithm goes in O(n), where n is the number of periods. Each recursive call reduces the number of periods by one until it hits the base case.

3. Explain how to optimize the recursive solution to avoid excessive computation.

Optimization:

* Store calculated values to avoid redundant computations.
* Use a loop to calculate future values incrementally.