Exercise1: Inventory Management System

Ans 1. Data structures and algorithms are important in managing large inventories because:

1. They allow to organize the data efficiently by facilitating its access, inclusion and exclusion which is fundamental for someone who has a huge amount of physical products.
2. Better time complexity- If we have good data structures then it will reduce the Time Complexity of operations and thus can improve overall performance.
3. Proper data structures to allow the system also easily scale when we have n number of products.

Ans 2. Appropriate Data Structures

1. Dynamic Array capabilities (Good for fast access to element at an index)
2. Efficient key-value pair storage, great for fast lookups, adds/deletes by product ID - e.g. like a map
3. Keeps the sorted order of elements so if we want to traverse all products in a sorted I will be efficient.

Analysis:-

1. Time Complexity Analysis

A) Add Product: As worst, O(1), because of using HashMap for efficient key...

B) Update Product: O(1) Amortized, as it is a key lookup and insertion into HashMap.

C Delete Product : O(1) in the average case as it is efficient to remove a key/value pair from HashMap

D) Retrieve Product: O(1)(Average time) ==> Key lookups in a HashMap are very fast

2. Optimization Talk:

a) Batch Operations Whenever Multiple products has to be Added, Updated or Deleted in one go batch processing can help reduce the object overhead.
b) Concurrency Handling: For mutli-threaded environment using ConcurrentHashMap over HashMap is better option to have good performance during thread-safe operations.
c) If large number of DIP products are accessed, they can be cached to save time on retrieval for popular items.

Exercise2: E-commerce Platform Search Function

Big O Notation

Ans1. The algorithmic efficiency is being described using Big O notation which provides a mathematical expression for time (and sometimes space) complexity of the algorithm. This is an upper bound on the time in terms of input size which gives us understanding as to how does the algorithm scales.

Ans2. A) Best Case- The minimum number of steps taken by the ALGORITHM.

B) Average Case: The expected situation where the algorithm takes an average number of steps.

C) Worst Case: This is the scenario in which our algorithm will take maximum steps

Analysing:-

Comparison of Time Complexity

Linear Search:

a) Best Case is O(1) (Item found at the beginning).
b) Average Case is O(n) (Item foudn in middle).
c) Worst case it O(n)(item found on last or not present.)

Binary Search:

a) Best Case and Average Case arrival time - O(1) (assuming it found the item at middle location of array)
b) Worst case running time- O(log n)(comparing to linear search, every step dividing list into a half)
c) Platform suitability for algorithm : Linear Searching is straight forward, but works on Unsorted List. It is applicable to small datasets when sorting costs outweigh the benefit.

B) At runtime Binary Search performs too much better for larger datasets, but it requires to be sorted which increases additional O(n log n) time complexity in sorting the array

C) E-commerce portal - Binary search: as binary search works best with large datasets and fast performance. Considering that product searches are common, the cost of first sorting can be validated with fast search.

Exercise 3: Sorting Customer Orders

Understand Sorting Algorithms

Bubble Sort:

A) Simple comparison-based algorithm

B) Repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order.

C) Time Complexity: Best Case O(n), Average and Worst Case O(n^2)

Insertion Sort:

A) Builds the sorted array one item at a time.

B) Moves elements greater than the key to one position ahead of their current

position.

C) Time Complexity: Best Case O(n), Average and Worst Case O(n^2)

Quick Sort:

A) divide-and-conquer algorithm

B) selects an element as a pivot and rearranges elements of the array in such a way that all elements smaller than the pivot go to the left side, while all greater elements go to the right side.

C) Complexities: Best Case and Average Case O(n log n), Worst Case O(n^2)

Merge Sort:

A) It is a divide-and-conquer type of algorithm.

B) It divides the array in a half, recursively sorts the halves, and then merges the

sorted halves.

C) Time Complexity: O(n log n) for all cases

Analysis

Performance Comparison

Bubble Sort:

Best Case: O(n)

Average and Worst Case: O(n^2)

Easy to implement but of low efficiency for big data due to its

quadratic time complexity.

Quick Sort:

Best and Average Case: O(n log n)

Worst Case: O(n^2) (rare, if poor pivot selection occurs)

Widely used because of average case performance and optimal utilization of memory.

Why Quick Sort is Used

- Efficiency: Quick Sort has a better average and best-case time complexity (O(n log n)) compared to Bubble Sort.
- Memory Usage: Quick Sort is an in-place sorting algorithm (i.e., it requires only a small, constant amount of additional storage space).
- Real-World Performance: Quick Sort is often faster in practice due to better cache performance and its nature of divide-and-conquer, which enables parallelism.

Exercise 4: Employee Management System

Array Representation in Memory

A) Contiguous Memory Allocation: Arrays are stored in contiguous memory locations. All elements stand next to each other in memory. This enables direct access through indexing.

B) Fixed Size: The size of an array is fixed at the time of its declaration.This means the array cannot dynamically change its size.

C) Efficient Access: This direct access of elements comes handy, with O(1) time complexity, because of the predictable layout of memory.

Pros of Arrays

A) Time of Access: Due to direct indexing, access of an element by its index is fairly quick.

B) Efficiency in Memory: Arrays are contiguous blocks of memory, thus reducing overhead.

C) Cache Performance: The memory being contiguous, arrays support spatial locality that can improve cache performance.

Analysis

Time Complexity Analysis

Add Employee: O(1) — Time to add an element at the end of the array is constant time, assuming there's space.

Search Employee: O(n) – The worst case to search might have to scan thewhole array.

Traverse Employees: O(n) –Traverse each element of the array.

Delete Employee: O(n) – In the worst case, delete operation may need to shift elements to fill the gap.

Disadvantages of Arrays

Fixed Size: Arrays have a fixed size, and this can result in inefficient usage of Memory or cannot store more elements than those allocated at the beginning.

Insertion and Deletion: Insertion or deletion of elements can be slow due to shifting of elements.

Sparse Data: It is not appropriate for sparse data structures. That is, elements are spread out over a large range of indices.

When to Use Arrays

Fixed Size Collection: When number of elements is known and won't change.

Fast Access: When fast access to elements by index is required.

Memory Contiguity: When the additional memory structures are overhead, and memory contiguity is a benefit.

Exercise 5: Task Management System

Understand Linked Lists

Types of Linked Lists

Singly Linked List:

- Every node is composed of data and a reference next that point to the following node in the series.
- All operations are easily implementable, but it can travel only in the forward direction.

Doubly Linked List:

- Each node contains data, with reference to the next node and the node behind.
- It facilitates traveling in both ways: forward and backward.
- It consumes more memory because of the additional reference.

Analysis

Time Complexity Analysis

Add Task: O(n) - In the worst case, addition of a task requires traversal to the end of the list.

Search Task: O(n) - In the worst case, search may have to traverse through the entire list.

Traverse Tasks: O(n) - Here every element in the list is visited.

Delete Task: O(n) - In the worst case, deletion can be required to scan the entire list to find the task that is to be deleted.

Advantages of Linked Lists over Arrays

Dynamic Size: Unlike arrays, linked lists can grow and shrink dynamically as per need.

Efficient Insertions/Deletions: Insertions and deletions are more efficient compared to arrays, especially for large lists or when operations are performed near the beginning of the list.

Memory Utilization: Linked lists can be more memory-efficient when the number of elements is unknown or varies significantly, as they do not need to allocate a fixed-size block of memory.

Disadvantages of Linked Lists

Access Time: The access time for elements in linked lists is O(n) since they use sequential access. Arrays have O(1) access time.

Memory Overhead: Each node in a linked list requires additional memory to store the reference to the next node. In very large lists, this can be quite significant.

Exercise 6: Library Management System

Linear Search Algorithm:

Description: Linear search is a sequential searching algorithm where every element of the list is searched one by one until the required element gets matched or the end of the list is reached.

Time Complexity:

- Best Case: O(1) (element at the beginning)
- Average Case: O(n)
- Worst Case: O(n) (element at the end or not present)

Binary Search Algorithm:

Description: It is a fast search algorithm in which the search process repetitively divides the search interval in half. It compares the target value to the middle element and decides which half to search next.

Time Complexity:

Best Case: O(1) (middle element is the target)

Average Case: O(log n)

Worst Case: O(log n)

Analysis

Time Complexity Comparison

Linear Search:

Best Case: O(1)

Average Case: O(n)

Worst Case: O(n)

Binary Search:

Best Case: O(1)

Average Case: O(log n)

Worst Case: O(log n)

When to Use Each Algorithm

Linear Search:

- Use when the list is unsorted.
- Suitable for small data sets; when the expense of the sort is unwarranted.
- Simple and no setup is needed.

Binary Search:

- Use when the list is sorted.
- More efficient with large datasets due to logarithmic time complexity.

Exercise 7: Financial Forecasting

Definition: Recursion is a technique of problem-solving in which a function calls itself as a subroutine. This would allow the function to repeat itself a number of times, in that it may call itself in the course of its execution.

Base Case and Recursive Case: Any recursive function needs to have a base case to stop the recursion and a recursive case that reduces the problem to a simpler version of itself. It simplifies complex problems and breaks them down into smaller, more feasible sub-problems.

Analysis

Time Complexity:

This recursive algorithm has a time complexity of O(n), where n is the number of periods. The predict Future Value method reduces the number of periods by 1 in each call, resulting in n recursive calls.

Optimizing the Recursive Solution:

Memoization: One of the ways to optimize the recursive solution is by using Memoization stores the results of expensive function calls and returns the cached result when the same inputs occur again.

Iterative Approach: One of the ways is to turn the recursive solution into an iterative one; in most cases, this will save much memory and prevent stack overflow.