**Exercise 1:InventoryManagementSystem:**

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

-> Data structures help store large amounts of inventory data in a organized and structured way, making it easier to manage and retrieve.

->Algorithms enable quick and efficient retrieval of specific inventory data, reducing the time and effort required to locate and manage inventory.

-> Data structures and algorithms can handle large volumes of data, making them suitable for large warehouses with extensive inventory.

->Data structures and algorithms help ensure data accuracy and consistency, reducing errors and discrepancies in inventory management.

-> Data structures and algorithms can be optimized to improve performance, reducing the time and resources required to manage inventory.

Discuss the types of data structures suitable for this problem.

Arrays are Suitable for storing small to medium-sized inventories, arrays provide fast access and modification of data.

Linked Lists are Useful for storing large inventories, linked lists allow for efficient insertion and deletion of data.

Stacks and Queues are Suitable for managing inventory with a Last-In-First-Out (LIFO) or First-In-First-Out (FIFO) policy.

OUTPUT:

java -cp /tmp/zFPQfwsjbb/InventoryManagementSystem

Initial Inventory:

Product ID: 1

Product Name: Apple iPhone

Quantity: 10

Price: 999.99

Product ID: 2

Product Name: Samsung TV

Quantity: 5

Price: 1299.99

Product ID: 3

Product Name: Nike Shoes

Quantity: 20

Price: 79.99

Updated Inventory:

Product ID: 1

Product Name: Apple iPhone

Quantity: 10

Price: 999.99

Product ID: 2

Product Name: Samsung TV

Quantity: 10

Price: 1499.99

Product ID: 3

Product Name: Nike Shoes

Quantity: 20

Price: 79.99

Final Inventory:

Product ID: 2

Product Name: Samsung TV

Quantity: 10

Price: 1499.99

Product ID: 3

Product Name: Nike Shoes

Quantity: 20

Price: 79.99

Analysis:

Add Operation Time complexity: O(1) on average, O(n) in the worst case

Delete Operation Time complexity: O(1) on average, O(n) in the worst case

Update Operation Time complexity: O(1) on average, O(n) in the worst case

Search Operation Time complexity: O(1) on average, O(n) in the worst case

Optimization:

->Optimizing the operations in the InventoryManagementSystem class can be achieved through several strategies.

->One approach is to use a more efficient data structure, such as a TreeMap, which provides a guaranteed O(log n) time complexity for search, insertion, and deletion operations.

->This can be particularly beneficial in scenarios where the inventory size is large and frequent updates are expected.

-> Another optimization technique is to implement a caching mechanism, where frequently accessed products are stored in a separate, faster data structure, such as a HashSet.

--------------------------------------------------------------------------------------------------------------------------------------

**Exercise 2: E-commerce Platform Search Function**

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

->Big O notation is a powerful tool for analyzing algorithms, allowing us to compare, optimize, and predict the performance of algorithms, which is crucial in many areas of computer science.

->Big O notation is defined as an upper bound on the number of steps an algorithm takes to complete, relative to the size of the input.

->It's usually expressed as a function of the input size, typically represented as 'n'.

->Formally, we say that an algorithm has a time complexity of O(f(n)) if the number of steps it takes to complete grows at most as fast as f(n) grows, as n approaches infinity.

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

->Best-Case Scenario the search operation finds the target element in the minimum number of steps possible.

->This typically occurs when the target element is located at the beginning of the search space or is easily accessible.

->Eg:Binary search in a sorted array: If the target element is at the middle index, the search operation finds it in a single comparison.

->Average-case scenario, the search operation takes a reasonable amount of time to find the target element.

->This is the most common scenario and represents the typical performance of the search algorithm.

->Eg:Linear search in an unsorted array:On average, the search operation finds the target element in n/2 comparisons, where n is the size of the array.Linear search in an unsorted array: On average, the search operation finds the target element in n/2 comparisons, where n is the size of the array.

->Worst-case scenario, the search operation takes the maximum number of steps possible to find the target element.

->This typically occurs when the target element is located at the end of the search space or is difficult to access.

->Linear search in an unsorted array: In the worst case, the search operation finds the target element in n comparisons, where n is the size of the array

Output:

java -cp /tmp/t8jjcdkbbs/ECommercePlatform

Linear search found: Product ID: 3, Product Name: Nike Shoes, Category: Fashion

Binary search found: Product ID: 2, Product Name: Samsung TV, Category: Electronics

Compare the time complexity of linear and binary search algorithms.

->Linear Search: O(n) - searches each element one by one, resulting in a linear increase in time complexity.

->Binary Search: O(log n) - divides the search space in half with each iteration, resulting in a logarithmic decrease in time complexity.

->Binary Search is generally faster than Linear Search, especially for large datasets.

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

->Binary Search is more suitable for our e-commerce platform due to its efficiency, scalability, and ability to handle large datasets.

->Frequent Searches: Binary Search can handle frequent searches more efficiently, providing a better user experience for customers searching for products.

->Scalability: As the product catalog grows, Binary Search's logarithmic time complexity ensures that search performance remains relatively consistent, making it a more scalable solution.

--------------------------------------------------------------------------------------------------------------------------------------

**Exercise 3:** **Sorting Customer Orders**

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

Bubble Sort

Definition: Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.

How it works:

Start at the beginning of the list

Compare the first two elements

If they are in the wrong order, swap them

Move to the next pair of elements and repeat the process

Continue until the end of the list is reached

Repeat the process until no more swaps are needed

Time Complexity: O(n^2)

2. Insertion Sort

Definition: Insertion Sort is a sorting algorithm that builds the final sorted array (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms like quicksort, heapsort, or merge sort.

How it works:

Start with the first element as the sorted list

Take the next element from the unsorted list and insert it into the sorted list in the correct position

Repeat the process until the entire list is sorted

Time Complexity: O(n^2)

3. Quick Sort

Definition: Quick Sort is a divide-and-conquer algorithm that selects a 'pivot' element from the array and partitions the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then recursively sorted.

How it works:

Choose a pivot element from the array

Partition the array into two sub-arrays: elements less than the pivot and elements greater than the pivot

Recursively apply the quicksort algorithm to the sub-arrays

Combine the results to produce the final sorted array

Time Complexity: O(n log n) on average, O(n^2) in the worst case

4. Merge Sort

Definition: Merge Sort is a divide-and-conquer algorithm that divides the input into several parts and then combines the results. It uses a temporary array to store the merged result.

How it works:

Divide the array into two halves

Recursively apply the merge sort algorithm to each half

Merge the two sorted halves into a single sorted array

Time Complexity: O(n log n)

Output:

java -cp /tmp/xmFAiuhzIu/SortOrders

Before sorting:

Order ID: 1, Customer Name: John Doe, Total Price: 100.0

Order ID: 2, Customer Name: Jane Doe, Total Price: 50.0

Order ID: 3, Customer Name: Bob Smith, Total Price: 200.0

Order ID: 4, Customer Name: Alice Johnson, Total Price: 150.0

Order ID: 5, Customer Name: Mike Brown, Total Price: 75.0

Sorting using Bubble Sort:

Order ID: 2, Customer Name: Jane Doe, Total Price: 50.0

Order ID: 5, Customer Name: Mike Brown, Total Price: 75.0

Order ID: 1, Customer Name: John Doe, Total Price: 100.0

Order ID: 4, Customer Name: Alice Johnson, Total Price: 150.0

Order ID: 3, Customer Name: Bob Smith, Total Price: 200.0

Sorting using Quick Sort:

Order ID: 2, Customer Name: Jane Doe, Total Price: 50.0

Order ID: 5, Customer Name: Mike Brown, Total Price: 75.0

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

Bubble Sort Time complexity: O(n^2) (average and worst case), O(n) (best case)

Simple to implement, but inefficient for large datasets

Quick Sort Time complexity: O(n log n) (average case), O(n^2) (worst case)

More efficient than Bubble Sort for large datasets, but more complex to implement

Discuss why Quick Sort is generally preferred over Bubble Sort.

Quick Sort's average time complexity is O(n log n), while Bubble Sort's is O(n^2), making Quick Sort much faster for large datasets.

Quick Sort can handle large datasets efficiently, while Bubble Sort becomes impractically slow.

Quick Sort requires less memory and CPU resources than Bubble Sort

-------------------------------------------------------------------------------------------------------------------------------------.

**Exercise 4: Employee Management System**

Explain how arrays are represented in memory and their advantages.

->Arrays are stored in contiguous memory locations.

->Each element of the array is stored in a separate memory location, and each location has a unique address.

->The base address of the array is the address of the first element, and the address of each subsequent element is calculated by adding the size of each element to the base address.

->For example, if an array arr has 5 elements, each of size 4 bytes,

Output:

All Employees:

Employee{employeeId=1, name='John Doe', position='Software Engineer', salary=50000.0}

Employee{employeeId=2, name='Jane Doe', position='Data Scientist', salary=60000.0}

Employee{employeeId=3, name='Bob Smith', position='Product Manager', salary=70000.0}

Search Employee by ID 2:

Employee{employeeId=2, name='Jane Doe', position='Data Scientist', salary=60000.0}

Delete Employee by ID 2:

All Employees after deletion:

Employee{employeeId=1, name='John Doe', position='Software Engineer', salary=50000.0}

Employee{employeeId=3, name='Bob Smith', position='Product Manager', salary=70000.0}

Analysis:

->Add Operation Time Complexity: O(n)

->Traverse Operation Time Complexity: O(n)

->Delete Operation Time Complexity: O(n)

->Search Operation Time Complexity: O(n)

Discuss the limitations of arrays and when to use them.

->Arrays have a fixed size that is determined at the time of declaration.

->This means that once an array is created, its size cannot be changed.

->Arrays can only store elements of the same data type.

->This means that an array can only store integers, or only store strings, but not a mix of both.

->Inserting or deleting elements in an array can be inefficient, especially if the array is large.

->This is because all elements after the insertion or deletion point need to be shifted.

**Exercise 5: Task Management System**

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

Singly Linked List (SLL):

->Each node has a reference to the next node in the sequence.

->Each node only points to the next node, not to the previous node.

->Insertion and deletion operations are relatively simple.

->Searching for a node is O(n) because we have to traverse the list from the beginning.

->Example: A list of nodes where each node points to the next node in the list.

2. Doubly Linked List (DLL):

->Each node has references to both the next node and the previous node in the sequence.

->Each node points to both the next node and the previous node.

->Insertion and deletion operations are more complex than SLL.

->Searching for a node is O(n) because we have to traverse the list from the beginning.

->Example: A list of nodes where each node points to both the next node and the previous node in the list.

Key difference: SLL has only one pointer (next), while DLL has two pointers (next and previous).

SLL is simpler to implement, but DLL provides more flexibility and efficiency in certain operations.

Output:

java -cp /tmp/A2izvPQ18V/TaskManagement

Task ID: 1

Task Name: Task 1

Status: Open

Task ID: 2

Task Name: Task 2

Status: In Progress

Task ID: 3

Task Name: Task 3

Status: Done

Task found: Task 2

Task ID: 1

Task Name: Task 1

Status: Open

Task ID: 3

Task Name: Task 3

Status: Done

Analysis:

Inserting a new task at the beginning of the list Time complexity: O(1)

Inserting a new task at the end of the list Time complexity: O(n)

Deleting a task from the beginning of the list Time complexity: O(1)

Deleting a task from the end of the list Time complexity: O(n)

Searching for a task by name and Displaying Time complexity: O(n)

Discuss the advantages of linked lists over arrays for dynamic data:

->Linked lists can allocate memory dynamically, which means they can grow or shrink as elements are added or removed.

->Linked lists can insert or delete elements at any position in O(1) time, whereas arrays require shifting all elements, which takes O(n) time.

->linked lists are a better choice than arrays when dealing with dynamic data that requires frequent insertions, deletions, or modifications, as they offer more flexibility and efficiency in terms of memory allocation and data manipulation.

**Exercise 6: Library Management System**

Explain linear search and binary search algorithms.

Linear Search:

->How it works: Iterate through a list of elements one by one, checking if each element matches the target value.

->Time complexity: O(n), where n is the number of elements in the list.

->Example: Finding a specific book in a library by checking each book on the shelf one by one.

->Pros: Simple to implement, works for unsorted lists.

->Cons: Slow for large lists, inefficient.

Binary Search:

->How it works: Divide a sorted list of elements into two halves, compare the target value to the middle element, and recursively search the appropriate half until the target value is found.

->Time complexity: O(log n), where n is the number of elements in the list.

->Example: Finding a specific word in a dictionary by dividing the pages into two halves and searching for the word in the appropriate half.

->Pros: Fast, efficient, and scalable for large lists.

->Cons: Requires a sorted list, more complex to implement.

Output:

Linear Search: Found book 1984 by George Orwell

Binary Search: Found book The Great Gatsby by F. Scott Fitzgerald

Compare the time complexity of linear and binary search.

Linear Search:

->Time complexity: O(n)

->Meaning: The time taken to search increases linearly with the size of the list.

->Example: If it takes 1 second to search 100 elements, it will take 10 seconds to search 1000 elements.

Binary Search:

->Time complexity: O(log n)

->Meaning: The time taken to search increases logarithmically with the size of the list.

->Example: If it takes 1 second to search 100 elements, it will take 3 seconds to search 1000 elements, and 6 seconds to search 1,000,000 elements.

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

Use Linear Search:

->Small data sets (e.g., < 100 elements)

->Unsorted data sets

->Data sets with frequent insertions/deletions

Use Binary Search:

Large data sets (e.g., > 1000 elements)

->Sorted data sets

->Data sets with infrequent insertions/deletions

**Exercise 7: Financial Forecasting**

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

Recursion is a programming technique where a function calls itself repeatedly until it reaches a base case that stops the recursion.

This process allows the function to break down a complex problem into smaller, more manageable sub-problems.

->Divide and Conquer: Recursion helps divide a problem into smaller sub-problems, making it easier to solve each one.

->Reduced Code Complexity: Recursion can simplify code by avoiding lengthy loops and conditional statements.

->Elegant Solutions: Recursion can lead to elegant, concise solutions that are easy to understand and maintain.

->Solves Problems with Self-Similar Structure: Recursion is particularly useful for problems with a self-similar structure, such as tree traversals, factorial calculations, and Fibonacci sequences.

->Example: Factorial Calculation

Output:

java -cp /tmp/59pwzGrugn/FutureValueCalculator

Future value after 10 years: 1628.8946267774422

Discuss the time complexity of your recursive algorithm.

The time complexity of the recursive algorithm for the Future Value Calculator is O(n), where n is the number of years.

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

Store intermediate results: Create a cache (e.g., a dictionary) to store the results of expensive function calls.

Check cache before recalculating: Before making a recursive call, check if the result is already in the cache. If it is, return the cached result instead of recalculating.

This approach reduces the time complexity by avoiding redundant calculations.