**Name: Suchana Singh**

**Email:** singhsuchana170@gmail.com

**Superset ID: 5008056**

**Solution : -** Exercise 1: Inventory Management System

1. Understanding the Problem

Importance of Data Structures and Algorithms in Inventory Management

Data structures and algorithms are critical in managing large inventories because they affect how efficiently data can be stored, accessed, and manipulated. Here’s why they matter:

* Quick Access: In a warehouse with thousands of products, finding a specific item quickly is crucial. Data structures like arrays or hash maps can help achieve fast lookup times, ensuring that the inventory management system can retrieve information without significant delays.
* Efficient Updates: An inventory system often requires updates, such as adding new products, updating quantities, or removing items. The choice of data structure impacts how efficiently these updates can be performed. For example, some structures allow for quick additions and deletions, which is essential for maintaining an up-to-date inventory.
* Memory Efficiency: Proper data structures ensure that the system uses memory efficiently, storing only the necessary information and reducing waste. This is important in large-scale systems where memory resources may be limited.
* Scalability: As the inventory grows, the system must handle increased data without performance degradation. Efficient algorithms ensure that operations like searching, sorting, and updating remain fast, even with a growing dataset.

Suitable Data Structures

1. ArrayList: This is a dynamic array that can grow as needed. It provides fast access to elements by index, making it suitable for storing products. However, inserting or deleting elements in the middle of the list can be slow because it may require shifting other elements.
2. HashMap: A HashMap (or Dictionary) uses key-value pairs to store data, allowing for quick retrieval using a unique key, such as a productId. This structure is highly efficient for search, insert, and delete operations, as it generally provides constant time complexity for these operations.

2. Setup

To start, we'll create a new project called "Inventory Management System." This project will include a class named Product and a data structure to store instances of this class.

3. Implementation:

public class Product {

private String productId;

private String productName;

private int quantity;

private double price;

public Product(String productId, String productName, int quantity, double price) {

this.productId = productId;

this.productName = productName;

this.quantity = quantity;

this.price = price;

}

public String getProductId() {

return productId;

}

public void setProductId(String productId) {

this.productId = productId;

}

public String getProductName() {

return productName;

}

public void setProductName(String productName) {

this.productName = productName;

}

public int getQuantity() {

return quantity;

}

public void setQuantity(int quantity) {

this.quantity = quantity;

}

public double getPrice() {

return price;

}

public void setPrice(double price) {

this.price = price;

}

}

**Choosing a Data Structure: HashMap**

We will use a HashMap to store the products, with the productId as the key and the Product object as the value. This choice allows for efficient access, insertion, and deletion of products.

import java.util.HashMap;

public class Inventory {

private HashMap<String, Product> products;

public Inventory() {

products = new HashMap<>();

}

public void addProduct(Product product) {

products.put(product.getProductId(), product);

}

public void updateProduct(String productId, Product newProduct) {

products.put(productId, newProduct);

}

public void deleteProduct(String productId) {

products.remove(productId);

}

public Product getProduct(String productId) {

return products.get(productId);

}

}

**4. Analysis**

**Time Complexity of Operations**

* **Add Product:** The time complexity is O(1) on average because inserting into a HashMap is generally constant time.
* **Update Product:** Also O(1) on average, as updating an existing key in a HashMap is similar to insertion.
* **Delete Product:** Again, O(1) on average, since removing an element from a HashMap is constant time.

**Optimizing Operations**

To further optimize, we can consider the following:

* **Indexing:** Implement additional indexing strategies if we need to frequently search by other attributes like productName or price.
* **Load Factor Management:** Ensure the HashMap maintains an optimal load factor (usually around 0.75) to balance memory usage and time complexity.
* **Concurrency:** For large-scale systems, consider thread-safe alternatives like ConcurrentHashMap if multiple threads will access the inventory simultaneously.

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

1. Understanding Asymptotic Notation

Big O Notation: Big O notation describes the upper bound of an algorithm's running time, helping us understand the worst-case scenario as the input size grows. It simplifies the analysis by focusing on the dominant term and ignoring constants and lower-order terms.

Scenarios for Search Operations:

* Best Case: The element is found at the first position (O(1) for linear search, O(1) if the middle element for binary search).
* Average Case: The time taken on average, depending on the distribution of elements.
* Worst Case: The element is not present, or is at the last position (O(n) for linear search, O(log n) for binary search).

2. Setup

public class Product {

private String productId;

private String productName;

private String category;

public Product(String productId, String productName, String category) {

this.productId = productId;

this.productName = productName;

this.category = category;

}

public String getProductId() {

return productId;

}

public void setProductId(String productId) {

this.productId = productId;

}

public String getProductName() {

return productName;

}

public void setProductName(String productName) {

this.productName = productName;

}

public String getCategory() {

return category;

}

public void setCategory(String category) {

this.category = category;

}

}

3. Implementation

Linear Search:

public Product linearSearch(Product[] products, String targetName) {

for (Product product : products) {

if (product.getProductName().equalsIgnoreCase(targetName)) {

return product;

}

}

return null;

}

**Binary Search:** Assuming products are sorted by productName.

public Product binarySearch(Product[] products, String targetName) {

int left = 0;

int right = products.length - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

int comparison = products[mid].getProductName().compareToIgnoreCase(targetName);

if (comparison == 0) {

return products[mid];

} else if (comparison < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

**4. Analysis**

**Time Complexity:**

* **Linear Search:** O(n)
* **Binary Search:** O(log n)

For an e-commerce platform, where the data set can be large and frequent searches are required, binary search is more suitable due to its logarithmic time complexity. However, it requires the data to be sorted, so ensuring up-to-date sorting is necessary.

**Solution : -** Exercise 3: Sorting Customer Orders

**1. Understanding Sorting Algorithms**

Bubble Sort:

* Algorithm: 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.
* Time Complexity:
  + Best: O(n)
  + Average/Worst: O(n²)
* Space Complexity: O(1)
* Characteristics: Simple but inefficient for large lists. It's a stable sort (preserves the relative order of equal elements).

Insertion Sort:

* Algorithm: Builds the final sorted array one item at a time. It takes an element from the unsorted portion and inserts it into the correct position in the sorted portion.
* Time Complexity:
  + Best: O(n)
  + Average/Worst: O(n²)
* Space Complexity: O(1)
* Characteristics: Efficient for small or nearly sorted lists. Stable and in-place.

Quick Sort:

* Algorithm: Selects a 'pivot' element and partitions the array into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.
* Time Complexity:
  + Best/Average: O(n log n)
  + Worst: O(n²) (rare, depends on pivot selection)
* Space Complexity: O(log n) (for the recursive stack)
* Characteristics: Generally very fast. Not stable but can be made so.

Merge Sort:

* Algorithm: Divides the list into halves, recursively sorts them, and then merges the sorted halves.
* Time Complexity: O(n log n) for all cases.
* Space Complexity: O(n)
* Characteristics: Stable and efficient. Not in-place due to additional memory requirement for the merging process.

**2. Setup**

**Order Class:**

public class Order {

private String orderId;

private String customerName;

private double totalPrice;

public Order(String orderId, String customerName, double totalPrice) {

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

public String getOrderId() {

return orderId;

}

public void setOrderId(String orderId) {

this.orderId = orderId;

}

public String getCustomerName() {

return customerName;

}

public void setCustomerName(String customerName) {

this.customerName = customerName;

}

public double getTotalPrice() {

return totalPrice;

}

public void setTotalPrice(double totalPrice) {

this.totalPrice = totalPrice;

}

}

**3. Implementation**

Bubble Sort:

public void bubbleSort(Order[] orders) {

int n = orders.length;

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (orders[j].getTotalPrice() > orders[j + 1].getTotalPrice()) {

// Swap orders[j] and orders[j + 1]

Order temp = orders[j];

orders[j] = orders[j + 1];

orders[j + 1] = temp;

}

}

}

}

Quick Sort:

public void quickSort(Order[] orders, int low, int high) {

if (low < high) {

int pi = partition(orders, low, high);

quickSort(orders, low, pi - 1);

quickSort(orders, pi + 1, high);

}

}

private int partition(Order[] orders, int low, int high) {

double pivot = orders[high].getTotalPrice();

int i = (low - 1);

for (int j = low; j < high; j++) {

if (orders[j].getTotalPrice() < pivot) {

i++;

Order temp = orders[i];

orders[i] = orders[j];

orders[j] = temp;

}

}

Order temp = orders[i + 1];

orders[i + 1] = orders[high];

orders[high] = temp;

return i + 1;

}

**4. Analysis**

**Performance Comparison:**

* **Bubble Sort:** The time complexity is O(n²), making it inefficient for large datasets. It requires multiple passes over the array and swaps adjacent elements repeatedly, which leads to high execution time.
* **Quick Sort:** The average and best-case time complexity is O(n log n), making it more efficient than Bubble Sort. However, the worst-case complexity is O(n²), which can occur if the pivot selection is poor. Nonetheless, with good pivot selection strategies (like random pivot or median-of-three), Quick Sort generally performs very well.

**Why Quick Sort is Preferred:**

* Quick Sort is typically faster for large datasets due to its average-case time complexity of O(n log n).
* It is more efficient in terms of space as it can be implemented in-place, whereas Bubble Sort also operates in-place but with a significantly higher number of swaps and comparisons.
* Quick Sort's divide-and-conquer approach allows it to efficiently sort large arrays, making it a preferred choice in many practical applications.

In conclusion, while both algorithms have their uses, Quick Sort's efficiency and scalability make it the better choice for sorting large lists.

**Solution : -** Exercise 4: Employee Management System

**1. Understanding Array Representation**

Array Representation in Memory:

* Contiguous Memory Allocation: Arrays are data structures that store elements of the same type in contiguous memory locations. This means each element in the array is stored sequentially in memory, with no gaps between them.
* Indexing: The elements of the array can be accessed using an index, which represents the position of the element within the array. The index typically starts from 0.

Advantages of Arrays:

* Constant-Time Access: Due to contiguous memory allocation, arrays allow O(1) time complexity for accessing any element, given its index.
* Cache-Friendly: The sequential storage of array elements improves cache performance because accessing one element often brings the next few elements into the cache.
* Ease of Implementation: Arrays are straightforward to implement and use, making them a popular choice for storing and managing data collections.

**2. Setup**

**Employee Class:**

public class Employee {

private String employeeId;

private String name;

private String position;

private double salary;

public Employee(String employeeId, String name, String position, double salary) {

this.employeeId = employeeId;

this.name = name;

this.position = position;

this.salary = salary;

}

public String getEmployeeId() {

return employeeId;

}

public void setEmployeeId(String employeeId) {

this.employeeId = employeeId;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getPosition() {

return position;

}

public void setPosition(String position) {

this.position = position;

}

public double getSalary() {

return salary;

}

public void setSalary(double salary) {

this.salary = salary;

}

}

**3. Implementation**

**Employee Management System with Array:**

public class EmployeeManagementSystem {

private Employee[] employees;

private int size;

private int capacity;

public EmployeeManagementSystem(int capacity) {

this.capacity = capacity;

this.employees = new Employee[capacity];

this.size = 0;

}

// Add a new employee

public boolean addEmployee(Employee employee) {

if (size >= capacity) {

System.out.println("Array is full. Cannot add more employees.");

return false;

}

employees[size++] = employee;

return true;

}

// Search for an employee by employeeId

public Employee searchEmployee(int employeeId) {

for (int i = 0; i < size; i++) {

if (employees[i].getEmployeeId() == employeeId) {

return employees[i];

}

}

return null;

}

// Traverse and print all employees

public void traverseEmployees() {

for (int i = 0; i < size; i++) {

System.out.println(employees[i]);

}

}

// Delete an employee by employeeId

public boolean deleteEmployee(int employeeId) {

for (int i = 0; i < size; i++) {

if (employees[i].getEmployeeId() == employeeId) {

employees[i] = employees[size - 1];

employees[size - 1] = null;

size--;

return true;

}

}

System.out.println("Employee not found.");

return false;

}

}

**4. Analysis**

**Time Complexity of Operations:**

* **Add Employee:** O(1) (amortized) since we are appending the employee at the end of the array. However, if the array is full and needs resizing, it may take O(n) time.
* **Search Employee:** O(n) as we may need to scan through the entire array to find the employee with the specified employeeId.
* **Traverse Employees:** O(n) since we iterate through all the elements in the array.
* **Delete Employee:** O(n) as we need to find the employee to delete (O(n)), and possibly shift elements to maintain the array's structure.

**Limitations of Arrays:**

* **Fixed Size:** Once declared, the size of an array cannot be changed. If the array is full, a new, larger array must be created, and the elements must be copied over.
* **Inefficient Insertions/Deletions:** Inserting or deleting elements from an array (except at the end) requires shifting elements, which is an O(n) operation.
* **Memory Waste:** If the declared capacity is not fully used, it can lead to wasted memory space.

**When to Use Arrays:**

* Use arrays when the number of elements is known beforehand and will not change significantly.
* Arrays are suitable when frequent access to elements by index is required and when memory usage is not a critical concern.
* For scenarios requiring dynamic resizing or frequent insertions/deletions, consider using data structures like ArrayLists, LinkedLists, or other dynamic data structures.

**Solution : -** Exercise 5: Task Management System

**1. Understanding Linked Lists**

**Singly Linked List:**

* **Structure:** A singly linked list consists of nodes where each node contains two parts: the data and a reference (or pointer) to the next node in the sequence. The first node is called the head, and the last node points to null.
* **Traversal:** You can only traverse in one direction, from the head to the last node.
* **Advantages:** Simple implementation, efficient insertions and deletions from the beginning or middle of the list.
* **Disadvantages:** Cannot traverse backwards, additional memory required for storing the pointer.

**Doubly Linked List:**

* **Structure:** A doubly linked list consists of nodes where each node contains three parts: the data, a reference to the next node, and a reference to the previous node. It has both a head and a tail pointer.
* **Traversal:** You can traverse in both directions, forward from the head and backward from the tail.
* **Advantages:** Easier to delete nodes since you have a reference to the previous node, can traverse in both directions.
* **Disadvantages:** Requires more memory to store the additional pointer, more complex implementation than a singly linked list.

**2. Setup**

**Task Class:**

public class Task {

private String taskId;

private String taskName;

private String status;

public Task(String taskId, String taskName, String status) {

this.taskId = taskId;

this.taskName = taskName;

this.status = status;

}

public String getTaskId() {

return taskId;

}

public void setTaskId(String taskId) {

this.taskId = taskId;

}

public String getTaskName() {

return taskName;

}

public void setTaskName(String taskName) {

this.taskName = taskName;

}

public String getStatus() {

return status;

}

public void setStatus(String status) {

this.status = status;

}

}

**3. Implementation**

**Singly Linked List for Task Management:**

public class TaskLinkedList {

private Node head;

private class Node {

Task task;

Node next;

Node(Task task) {

this.task = task;

this.next = null;

}

}

// Add a new task

public void addTask(Task task) {

Node newNode = new Node(task);

if (head == null) {

head = newNode;

} else {

Node current = head;

while (current.next != null) {

current = current.next;

}

current.next = newNode;

}

}

// Search for a task by taskId

public Task searchTask(int taskId) {

Node current = head;

while (current != null) {

if (current.task.getTaskId() == taskId) {

return current.task;

}

current = current.next;

}

return null;

}

// Traverse and print all tasks

public void traverseTasks() {

Node current = head;

while (current != null) {

System.out.println(current.task);

current = current.next;

}

}

// Delete a task by taskId

public boolean deleteTask(int taskId) {

if (head == null) {

return false;

}

if (head.task.getTaskId() == taskId) {

head = head.next;

return true;

}

Node current = head;

while (current.next != null && current.next.task.getTaskId() != taskId) {

current = current.next;

}

if (current.next == null) {

return false;

}

current.next = current.next.next;

return true;

}

}

**4. Analysis**

**Time Complexity of Operations:**

* **Add Task:** O(n) (in the worst case, when inserting at the end) for singly linked lists since we need to traverse the list to find the last node. O(1) if inserting at the beginning.
* **Search Task:** O(n) as we may need to traverse the entire list to find a task with a specific taskId.
* **Traverse Tasks:** O(n) since we need to visit each node in the list.
* **Delete Task:** O(n) as we may need to find the node to delete, and potentially adjust the pointers.

**Advantages of Linked Lists over Arrays for Dynamic Data:**

* **Dynamic Size:** Linked lists can easily grow or shrink in size, as nodes are dynamically allocated and deallocated. Arrays have a fixed size, and resizing them requires creating a new array and copying elements.
* **Efficient Insertions/Deletions:** Linked lists allow for efficient insertions and deletions from the middle or beginning of the list (O(1) for operations if the pointer is known). In contrast, arrays require shifting elements, which is O(n).
* **Memory Utilization:** Linked lists allocate memory as needed, whereas arrays may waste memory if the allocated capacity is not fully used.

**Limitations:**

* **Access Time:** Accessing elements in a linked list is O(n) since we must traverse the list, while arrays provide O(1) access time with indexing.
* **Extra Memory:** Linked lists require extra memory for storing pointers, which can be significant, especially if the data elements are small.

In conclusion, linked lists are well-suited for scenarios where dynamic resizing is needed, or where frequent insertions and deletions occur, especially in the middle of the collection. Arrays are better when fast access to elements is required and the size of the data is known beforehand.

**Solution : -** Exercise 6: Library Management System

**Linear Search:**

* **Algorithm:** Linear search involves scanning each element in a list sequentially until the desired element is found or the end of the list is reached.
* **Time Complexity:**
  + Best: O(1) (if the desired element is at the beginning)
  + Average/Worst: O(n) (if the element is at the end or not present)
* **Space Complexity:** O(1) (constant space)
* **Characteristics:** Simple and straightforward, works on unsorted or sorted lists.

**Binary Search:**

* **Algorithm:** Binary search works by repeatedly dividing a sorted list in half and comparing the target value to the middle element. If the target is less than the middle element, it continues searching in the left half; if greater, in the right half.
* **Time Complexity:**
  + Best/Average/Worst: O(log n)
* **Space Complexity:** O(1) for iterative implementation or O(log n) for recursive implementation (due to stack space in recursion)
* **Characteristics:** Efficient for large, sorted lists, but requires the list to be sorted.

### 2. Setup

**Book Class:**

public class Book {

private String bookId;

private String title;

private String author;

public Book(String bookId, String title, String author) {

this.bookId = bookId;

this.title = title;

this.author = author;

}

// Getters and setters

public String getBookId() {

return bookId;

}

public void setBookId(String bookId) {

this.bookId = bookId;

}

public String getTitle() {

return title;

}

public void setTitle(String title) {

this.title = title;

}

public String getAuthor() {

return author;

}

public void setAuthor(String author) {

this.author = author;

}

}

**3. Implementation**

**Linear Search:**

public class LibraryManagementSystem {

// Linear search for books by title

public static Book linearSearchByTitle(Book[] books, String title) {

for (Book book : books) {

if (book.getTitle().equalsIgnoreCase(title)) {

return book;

}

}

return null; // Book not found

}

}

**Binary Search:**

public class LibraryManagementSystem {

// Binary search for books by title

public static Book binarySearchByTitle(Book[] books, String title) {

int low = 0;

int high = books.length - 1;

while (low <= high) {

int mid = low + (high - low) / 2;

int comparison = books[mid].getTitle().compareToIgnoreCase(title);

if (comparison == 0) {

return books[mid]; // Book found

} else if (comparison < 0) {

low = mid + 1; // Search in the right half

} else {

high = mid - 1; // Search in the left half

}

}

return null; // Book not found

}

}

**4. Analysis**

**Time Complexity Comparison:**

* **Linear Search:** O(n) in the worst case. It checks each element one by one, so it scales linearly with the number of elements.
* **Binary Search:** O(log n) in all cases, assuming the list is sorted. It divides the search space in half each time, leading to logarithmic time complexity.

**When to Use Each Algorithm:**

* **Linear Search:** Use linear search when the list is unsorted or when the data set is relatively small. It's simple and doesn't require any preconditions.
* **Binary Search:** Use binary search when dealing with a large, sorted list. It is much more efficient for large datasets due to its logarithmic time complexity. However, the list must be sorted before binary search can be applied.

**Considerations:**

* For dynamic data where the list is frequently updated, maintaining a sorted list for binary search might be costly, so linear search might be preferred in such cases.
* If you only need occasional searches and the list remains mostly static, sorting the list once and using binary search for repeated queries can be highly efficient.

In summary, binary search is more efficient for large, sorted datasets, while linear search is more versatile and easier to implement, especially for unsorted or small lists.

**Solution : -** Exercise 7: Financial Forecasting

**1. Understanding Recursive Algorithms**

**Recursion Concept:**

* **Definition:** Recursion is a programming technique where a function calls itself in order to solve a problem. Each recursive call should work on a smaller instance of the same problem, eventually reaching a base case which stops the recursion.
* **Base Case:** The condition under which the recursive calls stop. It is crucial for avoiding infinite recursion.
* **Recursive Case:** The part of the function that includes the recursive call, which breaks down the problem into smaller instances.

**Advantages of Recursion:**

* **Simplifies Code:** Recursive solutions can often be more straightforward and elegant for problems that can be broken down into similar subproblems (e.g., tree traversal, factorial calculation).
* **Divide and Conquer:** Recursion naturally fits problems that involve dividing a problem into smaller, manageable subproblems.

**Disadvantages of Recursion:**

* **Performance Overhead:** Recursive function calls can lead to high memory usage and overhead due to the call stack.
* **Potential for Stack Overflow:** Deep recursion levels can exhaust the call stack, leading to stack overflow errors.

**2. Setup**

**Future Value Calculation Using Recursion:**

For financial forecasting, we can calculate the future value of an investment based on past growth rates using a recursive approach.

**Formula:** FutureValue(P,r,n)=P×(1+r)n where:

* P is the principal amount,
* r is the growth rate,
* n is the number of periods.

**Recursive Approach:** We will use a recursive approach to calculate future values, where each call calculates the value for one period and then calls itself to handle the remaining periods.

**3. Implementation**

**Recursive Method to Calculate Future Value:**

public class FinancialForecasting {

// Recursive method to calculate future value

public static double calculateFutureValue(double principal, double growthRate, int periods) {

// Base case: no periods left

if (periods == 0) {

return principal;

}

// Recursive case: calculate value for one period and call recursively

return calculateFutureValue(principal \* (1 + growthRate), growthRate, periods - 1);

}

public static void main(String[] args) {

double principal = 1000.0; // Initial investment

double growthRate = 0.05; // 5% growth rate

int periods = 10; // Number of periods

double futureValue = calculateFutureValue(principal, growthRate, periods);

System.out.println("Future Value: " + futureValue);

}

}

**4. Analysis**

**Time Complexity:**

* **Time Complexity of Recursive Algorithm:** O(n), where n is the number of periods. Each recursive call processes one period, and there are n calls in total.

**Optimization Techniques:**

* **Memoization:** To avoid redundant calculations, store results of recursive calls in a cache (e.g., a HashMap) and reuse them when needed. This technique is particularly useful when the same subproblems are solved multiple times.
* **Tail Recursion:** If the recursive function is in tail position (the recursive call is the last operation in the function), some compilers or interpreters can optimize it to avoid additional stack usage. However, Java does not perform tail-call optimization.

Example of Memoization for Future Value:

import java.util.HashMap;

public class FinancialForecasting {

private static HashMap<Integer, Double> memo = new HashMap<>();

// Recursive method with memoization

public static double calculateFutureValue(double principal, double growthRate, int periods) {

if (periods == 0) {

return principal;

}

if (memo.containsKey(periods)) {

return memo.get(periods);

}

double futureValue = calculateFutureValue(principal \* (1 + growthRate), growthRate, periods - 1);

memo.put(periods, futureValue);

return futureValue;

}

public static void main(String[] args) {

double principal = 1000.0;

double growthRate = 0.05;

int periods = 10;

double futureValue = calculateFutureValue(principal, growthRate, periods);

System.out.println("Future Value: " + futureValue);

}

}