## **Exercise 1: Inventory Management System**

### **Understand the Problem**

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

Data structures and algorithms are crucial for handling large inventories due to the following reasons:

* **Efficiency**: Data structures such as HashMap provide efficient storage and retrieval of data. For example, looking up a product by its ID in a HashMap is very fast (O(1) on average), which is essential when dealing with a large number of products.
* **Performance**: Algorithms dictate how quickly and efficiently operations like searching, adding, and updating can be performed. For instance, using efficient algorithms ensures that inventory operations remain manageable even as the dataset grows.
* **Scalability**: Good data structures and algorithms help in scaling the application. They ensure that performance remains acceptable even as the inventory size increases.

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

* **HashMap**: Suitable for storing product information where quick access and modification are needed. It provides average O(1) time complexity for operations like add, update, and delete. This is because it uses a hash table to store data.
* **ArrayList**: Useful for maintaining a list of products where the order matters and random access is required. It provides O(1) time complexity for access but O(n) for insertions and deletions due to the need to shift elements.
* **LinkedList**: Ideal for scenarios where frequent insertions and deletions are necessary. It allows O(1) time complexity for these operations but has O(n) time complexity for access due to linear traversal.

### **Code**

#### **Product.java**

public class Product {  
 private int productId;  
 private String productName;  
 private int quantity;  
 private double price;  
  
 public Product(int productId, String productName, int quantity, double price) {  
 this.productId = productId;  
 this.productName = productName;  
 this.quantity = quantity;  
 this.price = price;  
 }  
  
 // Getters and Setters  
 public int getProductId() {  
 return productId;  
 }  
  
 public void setProductId(int 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;  
 }  
}

#### **InventoryManagementSystem.java**

import java.util.HashMap;  
  
public class InventoryManagementSystem {  
 private HashMap<Integer, Product> inventory;  
  
 public InventoryManagementSystem() {  
 this.inventory = new HashMap<>();  
 }  
  
 public void addProduct(Product product) {  
 inventory.put(product.getProductId(), product);  
 }  
  
 public void updateProduct(Product product) {  
 inventory.put(product.getProductId(), product);  
 }  
  
 public void deleteProduct(int productId) {  
 inventory.remove(productId);  
 }  
  
 public Product getProduct(int productId) {  
 return inventory.get(productId);  
 }  
}

### **Analysis**

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

* **Add Operation**:
  + **HashMap**: O(1) on average. Adding a product involves computing the hash code and inserting it into the hash table.
* **Update Operation**:
  + **HashMap**: O(1) on average. Updating involves locating the product by its key and replacing the value.
* **Delete Operation**:
  + **HashMap**: O(1) on average. Removing a product involves locating it by key and deleting the entry.

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

* **HashMap Optimization**:
  + **Load Factor and Capacity**: Adjust the initial capacity and load factor to minimize collisions and maintain O(1) operations.
  + **ConcurrentHashMap**: For multi-threaded environments, use ConcurrentHashMap to handle concurrent access more efficiently.
* **ArrayList Optimization**:
  + **Capacity Management**: Preallocate capacity to avoid frequent resizing operations if the number of products is known in advance.
  + **Efficient Operations**: Use LinkedList if frequent insertions/deletions are required.

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

### **Understand Asymptotic Notation**

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

Big O notation is used to describe the upper bound of an algorithm's time or space complexity, providing insight into its efficiency.

* **Time Complexity**: Indicates how the runtime of an algorithm changes with the size of the input. For example, O(n) implies linear growth, while O(log n) implies logarithmic growth, which is more efficient.
* **Space Complexity**: Describes how memory usage changes with input size. O(1) means constant space, while O(n) means linear space usage.

Big O notation helps in evaluating and comparing algorithms to choose the most efficient one based on the problem's requirements and constraints.

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

* **Linear Search**:
  + **Best Case**: O(1). The target element is at the first position.
  + **Average Case**: O(n). The search involves scanning through half of the elements on average.
  + **Worst Case**: O(n). The target element is at the end or not present, requiring a full scan.
* **Binary Search**:
  + **Best Case**: O(1). The target element is at the midpoint.
  + **Average Case**: O(log n). The search interval is halved with each comparison.
  + **Worst Case**: O(log n). The search continues until the interval is reduced to one element.

### **Code**

#### **Product.java**

public class Product {  
 private int productId;  
 private String productName;  
 private String category;  
  
 public Product(int productId, String productName, String category) {  
 this.productId = productId;  
 this.productName = productName;  
 this.category = category;  
 }  
  
 // Getters and Setters  
 public int getProductId() {  
 return productId;  
 }  
  
 public void setProductId(int 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;  
 }  
}

#### **SearchAlgorithms.java**

import java.util.Arrays;  
  
public class SearchAlgorithms {  
 public static boolean linearSearch(Product[] products, int targetId) {  
 for (Product product : products) {  
 if (product.getProductId() == targetId) {  
 return true;  
 }  
 }  
 return false;  
 }  
  
 public static boolean binarySearch(Product[] products, int targetId) {  
 int left = 0;  
 int right = products.length - 1;  
 while (left <= right) {  
 int mid = left + (right - left) / 2;  
 if (products[mid].getProductId() == targetId) {  
 return true;  
 }  
 if (products[mid].getProductId() < targetId) {  
 left = mid + 1;  
 } else {  
 right = mid - 1;  
 }  
 }  
 return false;  
 }  
}

### **Analysis**

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

* **Linear Search**: O(n) in all cases. It scans each element one by one until the target is found or the end is reached.
* **Binary Search**: O(log n) in all cases. It repeatedly divides the search interval in half, making it much more efficient for large sorted datasets.

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

* **Binary Search** is more suitable for large sorted datasets due to its logarithmic time complexity, which provides faster search times compared to linear search. It is ideal when the dataset is static or changes infrequently.
* **Linear Search** is simpler and suitable for small or unsorted datasets where sorting is not practical. It is also useful for dynamic datasets where maintaining order is not feasible.

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

### **Understand Sorting Algorithms**

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

* **Bubble Sort**: A simple comparison-based algorithm that repeatedly swaps adjacent elements if they are in the wrong order. Time complexity: O(n^2).
* **Insertion Sort**: Builds the sorted array one element at a time by inserting each new element into its correct position in the already sorted portion. Time complexity: O(n^2) but performs well on small or partially sorted datasets.
* **Quick Sort**: A divide-and-conquer algorithm that partitions the array into elements less than and greater than a pivot, and recursively sorts the partitions. Time complexity: O(n log n) on average.
* **Merge Sort**: Another divide-and-conquer algorithm that divides the array into halves, recursively sorts each half, and then merges the sorted halves. Time complexity: O(n log n).

### **Code**

#### **Order.java**

public class Order {  
 private int orderId;  
 private String customerName;  
 private double totalPrice;  
  
 public Order(int orderId, String customerName, double totalPrice) {  
 this.orderId = orderId;  
 this.customerName = customerName;  
 this.totalPrice = totalPrice;  
 }  
  
 // Getters and Setters  
 public int getOrderId() {  
 return orderId;  
 }  
  
 public void setOrderId(int 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;  
 }  
}

#### **SortingAlgorithms.java**

import java.util.Arrays;  
  
public class SortingAlgorithms {  
  
 public static 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()) {  
 Order temp = orders[j];  
 orders[j] = orders[j + 1];  
 orders[j + 1] = temp;  
 }  
 }  
 }  
 }  
  
 public static 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 static 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;  
 }  
}

### **Analysis**

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

* **Bubble Sort**: O(n^2) in the worst and average cases. It is less efficient for large datasets because it performs a large number of comparisons and swaps.
* **Quick Sort**: O(n log n) on average. It is much more efficient for large datasets due to its divide-and-conquer approach and is generally faster than bubble sort.

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

* **Efficiency**: Quick Sort is preferred due to its O(n log n) average time complexity, making it significantly faster than Bubble Sort’s O(n^2), especially for large datasets.
* **Performance**: Quick Sort performs fewer swaps and comparisons on average, which leads to better overall performance.

## **Exercise 4: Employee Management System**

### **Understand Array Representation**

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

Arrays are represented in memory as contiguous blocks of memory. Each element is stored at a fixed offset from the start of the array, which allows for:

* **Constant Time Access**: O(1) time complexity for accessing elements by index because the memory address can be calculated directly.
* **Simplicity**: Arrays are simple to implement and use, making them suitable for many scenarios where fixed-size and quick access are required.

**Advantages**:

* **Fast Access**: Direct access to elements via index.
* **Memory Efficiency**: No additional memory overhead for pointers or links.

### **Code**

#### **Employee.java**

public class Employee {  
 private int employeeId;  
 private String name;  
 private String position;  
 private double salary;  
  
 public Employee(int employeeId, String name, String position, double salary) {  
 this.employeeId = employeeId;  
 this.name = name;  
 this.position = position;  
 this.salary = salary;

}  
  
 // Getters and Setters  
 public int getEmployeeId() {  
 return employeeId;  
 }  
  
 public void setEmployeeId(int 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;  
 }  
}

#### **EmployeeManagementSystem.java**

public class EmployeeManagementSystem {  
 private Employee[] employees;  
 private int size;  
  
 public EmployeeManagementSystem(int capacity) {  
 employees = new Employee[capacity];  
 size = 0;  
 }  
  
 public void addEmployee(Employee employee) {  
 if (size < employees.length) {  
 employees[size++] = employee;  
 }  
 }  
  
 public Employee searchEmployee(int employeeId) {  
 for (int i = 0; i < size; i++) {  
 if (employees[i].getEmployeeId() == employeeId) {  
 return employees[i];  
 }  
 }  
 return null;  
 }  
  
 public void traverseEmployees() {  
 for (int i = 0; i < size; i++) {  
 System.out.println(employees[i].getName());  
 }  
 }  
  
 public void 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;  
 }  
 }  
 }  
}

### **Analysis**

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

* **Add Operation**:
  + **Array**: O(1) if there is space in the array. It involves placing the new employee in the next available slot.
* **Search Operation**:
  + **Array**: O(n). Requires scanning through the array to find the employee.
* **Traverse Operation**:
  + **Array**: O(n). Involves iterating over all elements in the array.
* **Delete Operation**:
  + **Array**: O(n). Requires shifting elements to fill the gap left by the deleted employee.

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

* **Limitations**:
  + **Fixed Size**: Arrays have a fixed size once created. If the number of elements exceeds the initial capacity, resizing is required.
  + **Inefficient Deletions**: Deleting elements requires shifting elements, which can be inefficient.
* **When to Use**:
  + **Fixed Size**: When the number of elements is known in advance or doesn't change frequently.
  + **Fast Access**: When quick random access is needed.

## **Exercise 5: Task Management System**

### **Understand Linked Lists**

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

* **Singly Linked List**: Each node contains data and a reference to the next node. It supports efficient insertions and deletions but requires O(n) time for accessing elements by index.
* **Doubly Linked List**: Each node contains data, a reference to the next node, and a reference to the previous node. It allows efficient bidirectional traversal and operations but requires extra memory for the additional reference.

### **Code**

#### **Task.java**

public class Task {  
 private int taskId;  
 private String taskName;  
 private String status;  
  
 public Task(int taskId, String taskName, String status) {  
 this.taskId = taskId;  
 this.taskName = taskName;  
 this.status = status;  
 }  
  
 // Getters and Setters  
 public int getTaskId() {  
 return taskId;  
 }  
  
 public void setTaskId(int 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;  
 }  
}

#### **TaskManagementSystem.java**

public class TaskManagementSystem {  
 private class Node {  
 Task task;  
 Node next;  
  
 Node(Task task) {  
 this.task = task;  
 this.next = null;  
 }  
 }  
  
 private Node head;  
  
 public TaskManagementSystem() {  
 head = null;  
 }  
  
 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;  
 }  
 }  
  
 public Task searchTask(int taskId) {  
 Node current = head;  
 while (current != null) {  
 if (current.task.getTaskId() == taskId) {  
 return current.task;  
 }  
 current = current.next;  
 }  
 return null;  
 }  
  
 public void traverseTasks() {  
 Node current = head;  
 while (current != null) {  
 System.out.println(current.task.getTaskName());  
 current = current.next;  
 }  
 }  
  
 public void deleteTask(int taskId) {  
 if (head == null) return;  
  
 if (head.task.getTaskId() == taskId) {  
 head = head.next;  
 return;  
 }  
  
 Node current = head;  
 while (current.next != null && current.next.task.getTaskId() != taskId) {  
 current = current.next;  
 }  
  
 if (current.next != null) {  
 current.next = current.next.next;  
 }  
 }  
}

### **Analysis**

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

* **Add Operation**:
  + **Singly Linked List**: O(n) in the worst case if we need to traverse to the end. O(1) if adding at the head.
* **Search Operation**:
  + **Singly Linked List**: O(n). Requires scanning through the list to find the task.
* **Traverse Operation**:
  + **Singly Linked List**: O(n). Involves iterating over all nodes.
* **Delete Operation**:
  + **Singly Linked List**: O(n). Requires locating the node and then removing it.

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

* **Dynamic Size**: Linked lists can grow and shrink dynamically without needing resizing or shifting of elements.
* **Efficient Insertions/Deletions**: Inserting or deleting elements is more efficient as it doesn’t require shifting elements, unlike arrays.

## **Exercise 6: Library Management System**

### **Understand Search Algorithms**

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

* **Linear Search**: Iterates through each element sequentially until the target is found or the end is reached. Time complexity: O(n). It is simple but less efficient for large datasets.
* **Binary Search**: Requires a sorted array and works by repeatedly dividing the search interval in half. Time complexity: O(log n). It is efficient for large sorted datasets but requires sorting if the dataset is not already sorted.

### **Code**

#### **Book.java**

public class Book {  
 private int bookId;  
 private String title;  
 private String author;  
  
 public Book(int bookId, String title, String author) {  
 this.bookId = bookId;  
 this.title = title;  
 this.author = author;  
 }  
  
 // Getters and Setters  
 public int getBookId() {  
 return bookId;  
 }  
  
 public void setBookId(int 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;  
 }  
}

#### **LibraryManagementSystem.java**

import java.util.Arrays;  
  
public class LibraryManagementSystem {  
  
 public static boolean linearSearch(Book[] books, String targetTitle) {  
 for (Book book : books) {  
 if (book.getTitle().equals(targetTitle)) {  
 return true;  
 }  
 }  
 return false;  
 }  
  
 public static boolean binarySearch(Book[] books, String targetTitle) {  
 int left = 0;  
 int right = books.length - 1;  
 while (left <= right) {  
 int mid = left + (right - left) / 2;  
 if (books[mid].getTitle().equals(targetTitle)) {  
 return true;  
 }  
 if (books[mid].getTitle().compareTo(targetTitle) < 0) {  
 left = mid + 1;  
 } else {  
 right = mid - 1;  
 }  
 }  
 return false;  
 }  
}

### **Analysis**

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

* **Linear Search**: O(n) in all cases. It scans through each element until it finds the target or reaches the end.
* **Binary Search**: O(log n) in all cases. It is more efficient for sorted datasets due to its divide-and-conquer approach.

**2. Discuss when to use each algorithm based on the dataset size and order.**

* **Binary Search**: Use when the dataset is large and sorted. It offers much faster search times compared to linear search.
* **Linear Search**: Use for smaller or unsorted datasets. It is simpler to implement and doesn’t require sorting.

## **Exercise 7: Financial Forecasting**

### **Understand Recursive Algorithms**

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

Recursion is a technique where a function calls itself to solve a problem. It breaks down a problem into smaller subproblems of the same type.

**Benefits**:

* **Simplicity**: Recursive solutions can be simpler and more intuitive for problems that naturally fit a recursive pattern, such as tree traversals and factorial calculations.
* **Code Reduction**: It can reduce the complexity of code by avoiding iterative loops.

**Example**:

* **Factorial Calculation**: Factorial(n) = n \* Factorial(n-1). This can be directly translated into a recursive function.

### **Code**

#### **FinancialForecasting.java**

public class FinancialForecasting {  
  
 public static double forecastValue(double initialValue, double growthRate, int years) {  
 if (years == 0) {  
 return initialValue;  
 }  
 return forecastValue(initialValue \* (1 + growthRate), growthRate, years - 1);  
 }  
  
 public static void main(String[] args) {  
 double initialValue = 1000;  
 double growthRate = 0.05;  
 int years = 10;  
 double futureValue = forecastValue(initialValue, growthRate, years);  
 System.out.println("Future Value: " + futureValue);  
 }  
}

### **Analysis**

**1. Discuss the time complexity of your recursive algorithm.**

* **Recursive Algorithm**: The time complexity depends on the problem's nature. For example, in the financial forecasting example, each recursive call performs constant work. Thus, the time complexity is O(n), where n is the number of years.

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

* **Memoization**: Store the results of expensive function calls and reuse them when the same inputs occur again. This reduces redundant calculations.
* **Tail Recursion**: Optimize tail-recursive functions by converting them into iterative loops if supported by the language/compiler to avoid excessive stack usage.