Exercise 1: Inventory Management System

Step 1: Understand the Problem

Why Data Structures and Algorithms are Essential:

Efficient data structures and algorithms are crucial for handling large inventories because they directly impact the performance of the system in terms of speed and memory usage. With a large number of products, operations such as adding, updating, and deleting products must be optimized to ensure the system remains responsive.

Suitable Data Structures:

- **ArrayList:** Useful for dynamic arrays where the size can change.

- **HashMap:** Provides fast retrieval based on keys, making it ideal for inventory management where products can be accessed by product ID.

Step 2: Setup

Create a new project named `InventoryManagementSystem`.

Step 3: Implementation

Define the `Product` class:

```java
// Product.java
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;
```

```java
        this.productName = productName;
        this.quantity = quantity;
        this.price = price;
    }

    // Getters and setters...
}
```

Choose a data structure and implement methods to manage products:

```java
// Inventory.java
import java.util.HashMap;
import java.util.Map;

public class Inventory {
    private Map<String, Product> products;

    public Inventory() {
        products = new HashMap<>();
    }

    public void addProduct(Product product) {
        products.put(product.getProductId(), product);
    }

    public void updateProduct(Product product) {
        products.put(product.getProductId(), product);
    }

    public void deleteProduct(String productId) {
```

```
        products.remove(productId);
    }

    public Product getProduct(String productId) {
        return products.get(productId);
    }
}
```

Step 4: Analysis

Time Complexity:

- **Add Product:** O(1) (average case, due to HashMap insertion)
- **Update Product:** O(1) (average case, due to HashMap update)
- **Delete Product:** O(1) (average case, due to HashMap removal)
- **Get Product:** O(1) (average case, due to HashMap retrieval)

Optimization:

- Use a balanced binary search tree (like `TreeMap`) if ordered access to products is required.
- Use concurrent data structures for multi-threaded environments.

Exercise 2: E-commerce Platform Search Function

Step 1: Understand Asymptotic Notation

Big O Notation:

Big O notation describes the upper bound of the time complexity of an algorithm. It helps in understanding the worst-case scenario of an algorithm's performance.

Scenarios for Search Operations:

- **Best Case:** O(1) (first element is the target)

- **Average Case:** O(n/2) for linear search, O(log n) for binary search
- **Worst Case:** O(n) for linear search, O(log n) for binary search

Step 2: Setup

Create a class named `Product`:
```java
// Product.java
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;
    }

    // Getters and setters...
}
```

Step 3: Implementation

Implement linear and binary search:
```java
// Search.java
import java.util.Arrays;

public class Search {
```

```
    public static Product linearSearch(Product[] products, String productId) {
        for (Product product : products) {
            if (product.getProductId().equals(productId)) {
                return product;
            }
        }
        return null;
    }

    public static Product binarySearch(Product[] products, String productId) {
        Arrays.sort(products, (p1, p2) -> p1.getProductId().compareTo(p2.getProductId()));
        int left = 0;
        int right = products.length - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            int cmp = products[mid].getProductId().compareTo(productId);
            if (cmp == 0) {
                return products[mid];
            } else if (cmp < 0) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return null;
    }
}
```

Step 4: Analysis

Time Complexity:

- **Linear Search:** O(n)

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

Suitability:

Binary search is more suitable for large, sorted datasets due to its logarithmic time complexity, making it significantly faster than linear search for large datasets.

Exercise 3: Sorting Customer Orders

Step 1: Understand Sorting Algorithms

Sorting Algorithms:

- **Bubble Sort:** Simple but inefficient with O(n^2) complexity.

- **Quick Sort:** Efficient with average O(n log n) complexity.

- **Merge Sort:** Stable and efficient with O(n log n) complexity.

- **Insertion Sort:** Efficient for small or nearly sorted datasets with O(n^2) complexity.

Step 2: Setup

Create a class `Order`:

```java
// Order.java

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;
    }

    // Getters and setters...
}
```

Step 3: Implementation

Implement Bubble Sort and Quick Sort:

```java
// Sorting.java
public class Sorting {
    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;
    }
}
```

Step 4: Analysis

Time Complexity:

- **Bubble Sort:** O(n^2)
- **Quick Sort:** Average O(n log n), Worst O(n^2)

Preference:

Quick Sort is generally preferred over Bubble Sort due to its better average-case performance.

Exercise 4: Employee Management System

Step 1: Understand Array Representation

Arrays in Memory:

Arrays are contiguous blocks of memory with a fixed size, allowing fast access via index. They are efficient for read and write operations but less flexible for dynamic data.

Step 2: Setup

Create a class `Employee`:

```java
// Employee.java
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;
    }

    // Getters and setters...
}
```

Step 3: Implementation

Use an array to manage employees:

```java
// EmployeeManagement.java
public class EmployeeManagement {
    private Employee[] employees;
    private int count;

    public EmployeeManagement(int capacity) {
        employees = new Employee[capacity];
        count = 0;
    }

    public void addEmployee(Employee employee) {
        if (count < employees.length) {
            employees[count++] = employee;
        } else {
            System.out.println("Array is full. Cannot add more employees.");
        }
    }

    public Employee searchEmployee(String employeeId) {
        for (int i = 0; i < count; i++) {
            if (employees[i].getEmployeeId().equals(employeeId)) {
                return employees[i];
            }
        }
        return null;
    }

    public void deleteEmployee(String employeeId) {
```

```
        for (int i = 0; i < count; i++) {
            if (employees[i].getEmployeeId().equals(employeeId)) {
                employees[i] = employees[count - 1];
                employees[count - 1] = null;
                count--;
                return;
            }
        }
        System.out.println("Employee not found.");
    }

    public void traverseEmployees() {
        for (int i = 0; i < count

; i++) {
            System.out.println(employees[i].getName());
        }
    }
}
```

Step 4: Analysis

Time Complexity:

- **Add Employee:** O(1) if there is space, O(n) if resizing is needed
- **Search Employee:** O(n)
- **Delete Employee:** O(n)
- **Traverse Employees:** O(n)

Limitations of Arrays:

- Fixed size, not suitable for dynamic data.

- Inefficient for insertion and deletion in the middle of the array.

Exercise 5: Task Management System

Step 1: Understand Linked Lists

Types of Linked Lists:

- **Singly Linked List:** Each node points to the next node.

- **Doubly Linked List:** Each node points to both the next and previous nodes.

Step 2: Setup

Create a class `Task`:

```java
// Task.java
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;
    }

    // Getters and setters...
}
```

Step 3: Implementation

Implement a singly linked list to manage tasks:

```java
// SinglyLinkedList.java
public class SinglyLinkedList {
    private Node head;

    private class Node {
        Task task;
        Node next;

        Node(Task task) {
            this.task = task;
        }
    }

    public void addTask(Task task) {
        Node newNode = new Node(task);
        newNode.next = head;
        head = newNode;
    }

    public Task searchTask(String taskId) {
        Node current = head;
        while (current != null) {
            if (current.task.getTaskId().equals(taskId)) {
                return current.task;
            }
            current = current.next;
        }
        return null;
```

```
    }

    public void deleteTask(String taskId) {
        Node current = head;
        Node prev = null;
        while (current != null && !current.task.getTaskId().equals(taskId)) {
            prev = current;
            current = current.next;
        }
        if (current == null) {
            System.out.println("Task not found.");
            return;
        }
        if (prev == null) {
            head = head.next;
        } else {
            prev.next = current.next;
        }
    }

    public void traverseTasks() {
        Node current = head;
        while (current != null) {
            System.out.println(current.task.getTaskName());
            current = current.next;
        }
    }
}
```

Step 4: Analysis

Time Complexity:

- **Add Task:** O(1)

- **Search Task:** O(n)

- **Delete Task:** O(n)

- **Traverse Tasks:** O(n)

Advantages of Linked Lists:

- Dynamic size.

- Efficient for insertions and deletions at the beginning.

Exercise 6: Library Management System

Step 1: Understand Search Algorithms

Search Algorithms:

- **Linear Search:** O(n), scans each element.

- **Binary Search:** O(log n), requires a sorted array, divides the search interval in half.

Step 2: Setup

Create a class `Book`:

```java
// Book.java
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...
}
```

Step 3: Implementation

Implement linear and binary search:

```java
// Library.java
import java.util.Arrays;

public class Library {
    public static Book linearSearch(Book[] books, String title) {
        for (Book book : books) {
            if (book.getTitle().equalsIgnoreCase(title)) {
                return book;
            }
        }
        return null;
    }

    public static Book binarySearch(Book[] books, String title) {
        Arrays.sort(books, (b1, b2) -> b1.getTitle().compareToIgnoreCase(b2.getTitle()));
        int left = 0;
        int right = books.length - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
```

```
            int cmp = books[mid].getTitle().compareToIgnoreCase(title);
            if (cmp == 0) {
                return books[mid];
            } else if (cmp < 0) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return null;
    }
}
```

Step 4: Analysis

Time Complexity:

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

When to Use:

- Linear search is suitable for small or unsorted datasets.
- Binary search is preferable for large, sorted datasets due to its logarithmic complexity.

Exercise 7: Financial Forecasting

Step 1: Understand Recursive Algorithms

Recursion:

Recursion simplifies certain problems by breaking them down into smaller subproblems. However, it can lead to excessive computation if not optimized.

Step 2: Setup

Create a method to calculate future value using recursion:

```java
// FinancialForecasting.java
public class FinancialForecasting {
    public static double predictFutureValue(double currentValue, double growthRate, int periods) {
        if (periods == 0) {
            return currentValue;
        }
        return predictFutureValue(currentValue * (1 + growthRate), growthRate, periods - 1);
    }
}
```

Step 3: Implementation

Test the recursive algorithm:

```java
// FinancialForecastingTest.java
public class FinancialForecastingTest {
    public static void main(String[] args) {
        double currentValue = 1000.0;
        double growthRate = 0.05;
        int periods = 10;
        double futureValue = FinancialForecasting.predictFutureValue(currentValue, growthRate, periods);
        System.out.println("Future Value: " + futureValue);
    }
}
```

```

#### Step 4: Analysis

**Time Complexity:**

- The time complexity of the recursive algorithm is O(n) for n periods.

**Optimization:**

- Use memoization to store already computed results.

- Convert to an iterative approach to avoid excessive stack usage.

These exercises cover the implementation and analysis of different data structures and algorithms in Java, addressing various real-world scenarios.
```