Exercise 1: Inventory Management System

Scenario:

You are developing an inventory management system for a warehouse. Efficient data storage and retrieval are crucial.

Steps:

1. **Understand the Problem:**
    - Explain why data structures and algorithms are essential in handling large inventories.
    - Discuss the types of data structures suitable for this problem.
2. **Setup:**
    - Create a new project for the inventory management system.
3. **Implementation:**
    - Define a class Product with attributes like **productId**, **productName**, **quantity**, and **price**.
    - Choose an appropriate data structure to store the products (e.g., ArrayList, HashMap).
    - Implement methods to add, update, and delete products from the inventory.
4. **Analysis:**
    - Analyze the time complexity of each operation (add, update, delete) in your chosen data structure.
    - Discuss how you can optimize these operations.

Source Code:

```
class Product {
    int productId;
    String productName;
    int quantity;
    double price;

    public Product(int productId, String productName, int quantity, double price)
{
        this.productId = productId;
        this.productName = productName;
        this.quantity = quantity;
        this.price = price;
    }

    public void setProductName(String productName) {
        this.productName = productName;
    }
```

```
    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public String toString() {
        return "ID: " + productId + ", Name: " + productName + ", Quantity: " +
quantity + ", Price: " + price;
    }
}

class Inventory {
    Product[] products;
    int size;

    public Inventory(int capacity) {
        products = new Product[capacity];
        size = 0;
    }

    public void addProduct(Product product) {
        if (size < products.length) {
            products[size] = product;
            size++;
        }
    }

    public void updateProduct(int productId, String name, int quantity, double
price) {
        for (int i = 0; i < size; i++) {
            if (products[i].productId == productId) {
                products[i].setProductName(name);
                products[i].setQuantity(quantity);
                products[i].setPrice(price);
                break;
            }
        }
    }

    public void deleteProduct(int productId) {
```

```
        for (int i = 0; i < size; i++) {
            if (products[i].productId == productId) {
                for (int j = i; j < size - 1; j++) {
                    products[j] = products[j + 1];
                }
                products[size - 1] = null;
                size--;
                break;
            }
        }
    }

    public void displayAllProducts() {
        for (int i = 0; i < size; i++) {
            System.out.println(products[i]);
        }
    }
}

public class InventorySystem {
    public static void main(String[] args) {
        Inventory inventory = new Inventory(100);

        Product p1 = new Product(1, "Laptop", 10, 75000.0);
        Product p2 = new Product(2, "Mouse", 50, 500.0);

        inventory.addProduct(p1);
        inventory.addProduct(p2);

        inventory.updateProduct(1, "Gaming Laptop", 8, 95000.0);
        inventory.displayAllProducts();
        inventory.deleteProduct(2);

        inventory.displayAllProducts();
    }
}
```

Output:

```
PS C:\Users\ramgo\Documents\cognizant_company\deepskilling\solutions\data structures and analysis> javac InventorySystem.java
PS C:\Users\ramgo\Documents\cognizant_company\deepskilling\solutions\data structures and analysis> java InventorySystem
ID: 1, Name: Gaming Laptop, Quantity: 8, Price: 95000.0
ID: 2, Name: Mouse, Quantity: 50, Price: 500.0
ID: 1, Name: Gaming Laptop, Quantity: 8, Price: 95000.0
```

Exercise 2: E-commerce Platform Search Function

Scenario:

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

Steps:

1. **Understand Asymptotic Notation:**
    - Explain Big O notation and how it helps in analyzing algorithms.
    - Describe the best, average, and worst-case scenarios for search operations.
2. **Setup:**
    - Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
    - Implement linear search and binary search algorithms.
    - Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
    - Compare the time complexity of linear and binary search algorithms.
    - Discuss which algorithm is more suitable for your platform and why.

Source Code:

```
class Product {
    int productId;
    String productName;
    String category;

    public Product(int productId, String productName, String category) {
        this.productId = productId;
        this.productName = productName;
        this.category = category;
    }

    public String toString() {
        return "ID: " + productId + ", Name: " + productName + ", Category: " +
category;
    }
}
```

```
public class SearchFunction {

    public static Product linearSearch(Product[] products, String name) {
        for (int i = 0; i < products.length; i++) {
            if (products[i].productName.equalsIgnoreCase(name)) {
                return products[i];
            }
        }
        return null;
    }

    public static Product binarySearch(Product[] products, String name) {
        int low = 0;
        int high = products.length - 1;

        while (low <= high) {
            int mid = (low + high) / 2;
            int result = products[mid].productName.compareToIgnoreCase(name);

            if (result == 0) {
                return products[mid];
            } else if (result < 0) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }

        return null;
    }

    public static void sortProductsByName(Product[] products) {
        for (int i = 0; i < products.length - 1; i++) {
            for (int j = i + 1; j < products.length; j++) {
                if
(products[i].productName.compareToIgnoreCase(products[j].productName) > 0) {
                    Product temp = products[i];
                    products[i] = products[j];
                    products[j] = temp;
                }
            }
        }
```

```
    }

    public static void main(String[] args) {
        Product[] products = new Product[4];
        products[0] = new Product(101, "Laptop", "Electronics");
        products[1] = new Product(102, "Shoes", "Fashion");
        products[2] = new Product(103, "Watch", "Accessories");
        products[3] = new Product(104, "Phone", "Electronics");

        Product result1 = linearSearch(products, "Phone");
        if (result1!=null){
            System.out.println("Linear Search Result: " + result1);
        } else {
            System.out.println("Linear Search Result: Not found");
        }

        sortProductsByName(products);
        Product result2 = binarySearch(products, "Phone");
        if (result2!=null){
            System.out.println("Binary Search Result: " + result1);
        } else {
            System.out.println("Binary Search Result: Not found");
        }
    }
}
```

Output Code:

```
PS C:\Users\ramgo\Documents\cognizant_company\deepskilling\solutions\data structures and analysis> javac SearchFunction.java
PS C:\Users\ramgo\Documents\cognizant_company\deepskilling\solutions\data structures and analysis> java SearchFunction
Linear Search Result: ID: 104, Name: Phone, Category: Electronics
Binary Search Result: ID: 104, Name: Phone, Category: Electronics
```

Exercise 3: Sorting Customer Orders

Scenario:

You are tasked with sorting customer orders by their total price on an e-commerce platform. This helps in prioritizing high-value orders.

Steps:

1. **Understand Sorting Algorithms:**
    - Explain different sorting algorithms (Bubble Sort, Insertion Sort, Quick Sort, Merge Sort).
2. **Setup:**
    - Create a class **Order** with attributes like **orderId**, **customerName**, and **totalPrice**.
3. **Implementation:**
    - Implement **Bubble Sort** to sort orders by **totalPrice**.
    - Implement **Quick Sort** to sort orders by **totalPrice**.
4. **Analysis:**
    - Compare the performance (time complexity) of Bubble Sort and Quick Sort.
    - Discuss why Quick Sort is generally preferred over Bubble Sort.

Source Code:

```
class Order {
    int orderId;
    String customerName;
    double totalPrice;

    public Order(int orderId, String customerName, double totalPrice) {
        this.orderId = orderId;
        this.customerName = customerName;
        this.totalPrice = totalPrice;
    }

    public String toString() {
        return "Order ID: " + orderId + ", Name: " + customerName + ", Total: ₹"
+ totalPrice;
    }
}

public class OrderSorting {

    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].totalPrice < orders[j + 1].totalPrice) {
                    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 pivotIndex = partition(orders, low, high);
            quickSort(orders, low, pivotIndex - 1);
            quickSort(orders, pivotIndex + 1, high);
        }
    }

    public static int partition(Order[] orders, int low, int high) {
        double pivot = orders[high].totalPrice;
        int i = low - 1;
        for (int j = low; j < high; j++) {
            if (orders[j].totalPrice >= 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;
    }
    public static void main(String[] args) {
        Order[] orders1 = {
            new Order(101, "Alice", 2500.0),
            new Order(102, "Bob", 1500.0),
            new Order(103, "Charlie", 4000.0),
            new Order(104, "David", 1000.0),
            new Order(105, "Eve", 3000.0)
        };

        System.out.println("Original Orders:");
        for (Order order : orders1) {
            System.out.println(order);
        }

        bubbleSort(orders1);
```

```
        System.out.println("\nSorted by Bubble Sort (Descending by Total
Price):");
        for (Order order : orders1) {
            System.out.println(order);
        }

        Order[] orders2 = {
            new Order(101, "Alice", 2500.0),
            new Order(102, "Bob", 1500.0),
            new Order(103, "Charlie", 4000.0),
            new Order(104, "David", 1000.0),
            new Order(105, "Eve", 3000.0)
        };

        quickSort(orders2, 0, orders2.length - 1);
        System.out.println("\nSorted by Quick Sort (Descending by Total
Price):");
        for (Order order : orders2) {
            System.out.println(order);
        }
    }
}
```

Output:

```
PS C:\Users\ramgo\Documents\cognizant_company\deepskilling\solutions\data structures and analysis> javac OrderSorting.java
PS C:\Users\ramgo\Documents\cognizant_company\deepskilling\solutions\data structures and analysis> java OrderSorting
Original Orders:
Order ID: 101, Name: Alice, Total: ?2500.0
Order ID: 102, Name: Bob, Total: ?1500.0
Order ID: 103, Name: Charlie, Total: ?4000.0
Order ID: 104, Name: David, Total: ?1000.0
Order ID: 105, Name: Eve, Total: ?3000.0

Sorted by Bubble Sort (Descending by Total Price):
Order ID: 103, Name: Charlie, Total: ?4000.0
Order ID: 105, Name: Eve, Total: ?3000.0
Order ID: 101, Name: Alice, Total: ?2500.0
Order ID: 102, Name: Bob, Total: ?1500.0
Order ID: 104, Name: David, Total: ?1000.0

Sorted by Quick Sort (Descending by Total Price):
Order ID: 103, Name: Charlie, Total: ?4000.0
Order ID: 105, Name: Eve, Total: ?3000.0
Order ID: 101, Name: Alice, Total: ?2500.0
Order ID: 102, Name: Bob, Total: ?1500.0
Order ID: 104, Name: David, Total: ?1000.0
```

Exercise 4: Employee Management System

Scenario:

You are developing an employee management system for a company. Efficiently managing employee records is crucial.

Steps:

1. **Understand Array Representation:**
    - Explain how arrays are represented in memory and their advantages.
2. **Setup:**
    - Create a class Employee with attributes like **employeeId**, **name**, **position**, and **salary**.
3. **Implementation:**
    - Use an array to store employee records.
    - Implement methods to **add**, **search**, **traverse**, and **delete** employees in the array.
4. **Analysis:**
    - Analyze the time complexity of each operation (add, search, traverse, delete).
    - Discuss the limitations of arrays and when to use them

Source Code:

```
class Employee {
    int employeeId;
    String name;
    String position;
    double salary;

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

    public String toString() {
        return "ID: " + employeeId + ", Name: " + name + ", Position: " +
position + ", Salary: ₹" + salary;
    }
}

public class EmployeeManagementSystem {
    static Employee[] employees = new Employee[100];
    static int count = 0;
```

```
    public static void addEmployee(Employee emp) {
        if (count < 100) {
            employees[count] = emp;
            count++;
        } else {
            System.out.println("Employee list is full.");
        }
    }

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

    public static void deleteEmployee(int id) {
        for (int i = 0; i < count; i++) {
            if (employees[i].employeeId == id) {
                for (int j = i; j < count - 1; j++) {
                    employees[j] = employees[j + 1];
                }
                employees[count - 1] = null;
                count--;
                System.out.println("Employee with ID " + id + " deleted
successfully.");
            }
        }
    }

    public static void traverseEmployees() {
        if (count == 0) {
            System.out.println("No employees to display.");
        }
        for (int i = 0; i < count; i++) {
            System.out.println(employees[i]);
        }
    }

    public static void main(String[] args) {
```

```
        addEmployee(new Employee(1, "Alice", "Manager", 75000));
        addEmployee(new Employee(2, "Bob", "Developer", 60000));
        addEmployee(new Employee(3, "Charlie", "Designer", 50000));

        System.out.println("All Employees:");
        traverseEmployees();

        System.out.println("\nSearching for Employee with ID 2:");
        Employee emp = searchEmployee(2);
        if (emp != null) {
            System.out.println(emp);
        } else {
            System.out.println("Employee not found.");
        }

        System.out.println("\nDeleting Employee with ID 1:");
        deleteEmployee(1);

        System.out.println("\nAll Employees after Deletion:");
        traverseEmployees();
    }
}
```

Output:

```
PS C:\Users\ramgo\Documents\cognizant_company\deepskilling\solutions\data structures and analysis> javac EmployeeManagementSystem.java
PS C:\Users\ramgo\Documents\cognizant_company\deepskilling\solutions\data structures and analysis> java EmployeeManagementSystem
All Employees:
ID: 1, Name: Alice, Position: Manager, Salary: ?75000.0
ID: 2, Name: Bob, Position: Developer, Salary: ?60000.0
ID: 3, Name: Charlie, Position: Designer, Salary: ?50000.0

Searching for Employee with ID 2:
ID: 2, Name: Bob, Position: Developer, Salary: ?60000.0

Deleting Employee with ID 1:
Employee with ID 1 deleted successfully.

All Employees after Deletion:
ID: 2, Name: Bob, Position: Developer, Salary: ?60000.0
ID: 3, Name: Charlie, Position: Designer, Salary: ?50000.0
```

Exercise 5: Task Management System

Scenario:

You are developing a task management system where tasks need to be added, deleted, and traversed efficiently.

Steps:

1. **Understand Linked Lists:**
   - Explain the different types of linked lists (Singly Linked List, Doubly Linked List).
2. **Setup:**
   - Create a class **Task** with attributes like **taskId**, **taskName**, and **status**.
3. **Implementation:**
   - Implement a singly linked list to manage tasks.
   - Implement methods to **add**, **search**, **traverse**, and **delete** tasks in the linked list.
4. **Analysis:**
   - Analyze the time complexity of each operation.
   - Discuss the advantages of linked lists over arrays for dynamic data.

Source Code:

```
class Task {
    int taskId;
    String taskName;
    String status;

    public Task(int taskId, String taskName, String status) {
        this.taskId = taskId;
        this.taskName = taskName;
        this.status = status;
    }

    public String toString() {
        return "Task ID: " + taskId + ", Name: " + taskName + ", Status: " +
status;
    }
}

class TaskNode {
    Task task;
    TaskNode next;

    public TaskNode(Task task) {
        this.task = task;
        this.next = null;
    }
```

```
}

class TaskLinkedList {
    TaskNode head;

    public void addTask(Task task) {
        TaskNode newNode = new TaskNode(task);
        if (head == null) {
            head = newNode;
        } else {
            TaskNode temp = head;
            while (temp.next != null) {
                temp = temp.next;
            }
            temp.next = newNode;
        }
    }

    public Task searchTask(int id) {
        TaskNode temp = head;
        while (temp != null) {
            if (temp.task.taskId == id) {
                return temp.task;
            }
            temp = temp.next;
        }
        return null;
    }

    public void deleteTask(int id) {
        if (head == null) return;

        if (head.task.taskId == id) {
            head = head.next;
            return;
        }

        TaskNode prev = head;
        TaskNode curr = head.next;

        while (curr != null) {
            if (curr.task.taskId == id) {
                prev.next = curr.next;
```

```
                return;
            }
            prev = curr;
            curr = curr.next;
        }

        System.out.println("Task not found.");
    }

    public void traverseTasks() {
        if (head == null) {
            System.out.println("No tasks available.");
            return;
        }

        TaskNode temp = head;
        while (temp != null) {
            System.out.println(temp.task);
            temp = temp.next;
        }
    }
}

public class TaskManagementSystem {
    public static void main(String[] args) {
        TaskLinkedList taskList = new TaskLinkedList();

        taskList.addTask(new Task(1, "Design UI", "Pending"));
        taskList.addTask(new Task(2, "Develop Backend", "In Progress"));
        taskList.addTask(new Task(3, "Testing", "Pending"));

        System.out.println("All Tasks:");
        taskList.traverseTasks();

        System.out.println("\nSearching for Task ID 2:");
        Task found = taskList.searchTask(2);
        if (found != null) {
            System.out.println(found);
        } else {
            System.out.println("Task not found.");
        }

        System.out.println("\nDeleting Task ID 1:");
```

```
        taskList.deleteTask(1);

        System.out.println("\nTasks after deletion:");
        taskList.traverseTasks();
    }
}
```

Output:

```
PS C:\Users\ramgo\Documents\cognizant_company\deepskilling\solutions\data structures and analysis> javac TaskManagementSystem.java
PS C:\Users\ramgo\Documents\cognizant_company\deepskilling\solutions\data structures and analysis> java TaskManagementSystem
All Tasks:
Task ID: 1, Name: Design UI, Status: Pending
Task ID: 2, Name: Develop Backend, Status: In Progress
Task ID: 3, Name: Testing, Status: Pending

Searching for Task ID 2:
Task ID: 2, Name: Develop Backend, Status: In Progress

Deleting Task ID 1:
Task with ID 1 deleted successfully.

Tasks after deletion:
Task ID: 2, Name: Develop Backend, Status: In Progress
Task ID: 3, Name: Testing, Status: Pending
```

Exercise 6: Library Management System

Scenario:

You are developing a library management system where users can search for books by title or author.

Steps:

1. **Understand Search Algorithms:**
    - Explain linear search and binary search algorithms.
2. **Setup:**
    - Create a class **Book** with attributes like **bookId**, **title**, and **author**.
3. **Implementation:**
    - Implement linear search to find books by title.
    - Implement binary search to find books by title (assuming the list is sorted).
4. **Analysis:**
    - Compare the time complexity of linear and binary search.
    - Discuss when to use each algorithm based on the data set size and order.

Source Code:

```
class Book {
    int bookId;
    String title;
    String author;

    public Book(int bookId, String title, String author) {
        this.bookId = bookId;
        this.title = title;
        this.author = author;
    }

    public String toString() {
        return "Book ID: " + bookId + ", Title: " + title + ", Author: " +
author;
    }
}

public class LibraryManagementSystem {

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

    public static void sortBooksByTitle(Book[] books) {
        int n = books.length;
        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - 1 - i; j++) {
                if (books[j].title.compareToIgnoreCase(books[j + 1].title) > 0) {
                    Book temp = books[j];
                    books[j] = books[j + 1];
                    books[j + 1] = temp;
                }
            }
        }
    }
```

```java
public static Book binarySearch(Book[] books, String title) {
    int low = 0, high = books.length - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        int cmp = books[mid].title.compareToIgnoreCase(title);
        if (cmp == 0) {
            return books[mid];
        } else if (cmp < 0) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return null;
}

public static void displayBooks(Book[] books) {
    for (Book book : books) {
        System.out.println(book);
    }
}

public static void main(String[] args) {
    Book[] books = {
        new Book(101, "Java Programming", "James Gosling"),
        new Book(102, "Data Structures", "Narasimha Karumanchi"),
        new Book(103, "Operating Systems", "Silberschatz"),
        new Book(104, "Clean Code", "Robert C. Martin"),
        new Book(105, "Design Patterns", "Erich Gamma")
    };

    System.out.println("All Books:");
    displayBooks(books);

    Book result1 = linearSearch(books, "Clean Code");
    System.out.println("Linear Search Result:");
    System.out.println(result1 != null ? result1 : "Book not found");

    sortBooksByTitle(books);

    Book result2 = binarySearch(books, "Clean Code");
    System.out.println("Binary Search Result:");
```

```
        System.out.println(result2 != null ? result2 : "Book not found");
    }
}
```

Output:

```
PS C:\Users\ramgo\Documents\cognizant_company\deepskilling\solutions\data structures and analysis> javac LibraryManagementSystem.java
PS C:\Users\ramgo\Documents\cognizant_company\deepskilling\solutions\data structures and analysis> java LibraryManagementSystem
All Books:
Book ID: 101, Title: Java Programming, Author: James Gosling
Book ID: 102, Title: Data Structures, Author: Narasimha Karumanchi
Book ID: 103, Title: Operating Systems, Author: Silberschatz
Book ID: 104, Title: Clean Code, Author: Robert C. Martin
Book ID: 105, Title: Design Patterns, Author: Erich Gamma
Linear Search Result:
Book ID: 104, Title: Clean Code, Author: Robert C. Martin
Binary Search Result:
Book ID: 104, Title: Clean Code, Author: Robert C. Martin
```

Exercise 7: Financial Forecasting

Scenario:

You are developing a financial forecasting tool that predicts future values based on past data.

Steps:

1. **Understand Recursive Algorithms:**
   - Explain the concept of recursion and how it can simplify certain problems.
2. **Setup:**
   - Create a method to calculate the future value using a recursive approach.
3. **Implementation:**
   - Implement a recursive algorithm to predict future values based on past growth rates.
4. **Analysis:**
   - Discuss the time complexity of your recursive algorithm.
   - Explain how to optimize the recursive solution to avoid excessive computation.

Source Code:

```
public class FinancialForecasting {
```

```java
    public static double futureValue(double presentValue, double rate, int years)
{
        if (years == 0) {
            return presentValue;
        } else {
            return futureValue(presentValue * (1 + rate), rate, years - 1);
        }
    }

    public static void main(String[] args) {
        double presentValue = 10000;
        double rate = 0.08;
        int years = 5;

        double result = futureValue(presentValue, rate, years);
        System.out.printf("Future value after %d years: ₹%.2f%n", years, result);
    }
}
```

Output:

```
PS C:\Users\ramgo\Documents\cognizant_company\deepskilling\solutions\data structures and analysis> javac FinancialForecasting.java
PS C:\Users\ramgo\Documents\cognizant_company\deepskilling\solutions\data structures and analysis> java FinancialForecasting
Future value after 5 years: ?14693.28
```

Note: In above Outputs (?) Represents the Rupee Symbol (₹)