Exercise 1: Inventory Management System

Importance of Data Structures and Algorithms:

Data structures and algorithms are crucial in handling large inventories because they ensure efficient data storage, retrieval, and modification. Efficient data structures allow for quick access and updates, which is vital for inventory management tasks like adding new products, updating existing products, and deleting products.

Suitable Data Structures:

- **HashMap**: Provides average O(1) time complexity for insertions, deletions, and lookups.
- **ArrayList**: Provides O(1) time complexity for access by index, but O(n) for search, insertions, and deletions.

Setup

Create a new project for the inventory management system.

Implementation

Define the `Product` Class

```
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;
    }

    // Getters and setters
    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;
    }
}
```

Choose an Appropriate Data Structure

We'll use `HashMap` to store the products.

```
import java.util.HashMap;

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

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

    // Method to add a product
    public void addProduct(Product product) {
        products.put(product.getProductId(), product);
    }

    // Method to update a product
    public void updateProduct(String productId, Product
updatedProduct) {
        if (products.containsKey(productId)) {
            products.put(productId, updatedProduct);
        } else {
            System.out.println("Product not found");
        }
    }

    // Method to delete a product
    public void deleteProduct(String productId) {
        if (products.containsKey(productId)) {
            products.remove(productId);
        } else {
            System.out.println("Product not found");
        }
    }

   }
```

Analysis

Time Complexity

- **Add Operation**: O(1) on average since we're using `HashMap`.
- **Update Operation**: O(1) on average since we're using `HashMap`.
- **Delete Operation**: O(1) on average since we're using `HashMap`.

Optimization

Since `HashMap` already provides average O(1) time complexity for add, update, and delete operations, it's already optimized for this purpose. However, if memory usage becomes a concern, we might need to manage the load factor and rehashing process efficiently.

Exercise 2: E-commerce Platform Search Function

Understand Asymptotic Notation

Big O Notation

Big O notation describes the upper bound of the time complexity of an algorithm, providing an estimate of the worst-case scenario for the algorithm's growth rate as the input size increases.

- **Best-case**: The scenario where the algorithm performs the minimum number of steps.
- **Average-case**: The scenario representing the average number of steps an algorithm performs across all inputs.
- **Worst-case**: The scenario where the algorithm performs the maximum number of steps.

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;
    }

    // Getters and setters
    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;
    }
}
```

Implementation

Linear Search

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

Binary Search

```
import java.util.Arrays;
import java.util.Comparator;

public Product binarySearch(Product[] products, String
targetProductName) {
    Arrays.sort(products,
Comparator.comparing(Product::getProductName));
    int left = 0;
```

```
    int right = products.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        int comparison =
products[mid].getProductName().compareTo(targetProductName);
        if (comparison == 0) {
            return products[mid];
        }
        if (comparison < 0) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return null;
}
```

Analysis

Time Complexity

- **Linear Search**: O(n) in the worst case, where n is the number of products.
- **Binary Search**: O(log n) in the worst case, but it requires the array to be sorted first (O(n log n)).

Suitable Algorithm

For large datasets where the list is sorted, **binary search** is more efficient due to its O(log n) time complexity. For unsorted lists or small datasets, **linear search** might be sufficient due to its simplicity and no need for sorting.

Exercise 3: Sorting Customer Orders

Understand Sorting Algorithms

- **Bubble Sort**: Simple but inefficient, O(n^2) time complexity.
- **Insertion Sort**: Simple, efficient for small or nearly sorted arrays, O(n^2) time complexity.
- **Quick Sort**: Efficient in practice, average O(n log n) time complexity, but O(n^2) in the worst case.
- **Merge Sort**: Stable and guarantees O(n log n) time complexity, but requires additional memory.

Setup

```
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
    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;
    }
}
```

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++;
            // Swap orders[i] and orders[j]
            Order temp = orders[i];
            orders[i] = orders[j];
            orders[j] = temp;
        }
    }
```

```
    // Swap orders[i + 1] and orders[high] (or pivot)
    Order temp = orders[i + 1];
    orders[i + 1] = orders[high];
    orders[high] = temp;
    return i + 1;
}
```

Analysis

Time Complexity

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

Preferred Algorithm

Quick Sort is generally preferred over **Bubble Sort** due to its much better average-case time complexity. Bubble Sort is only suitable for educational purposes or very small datasets.

Exercise 4: Employee Management System

Understand Array Representation

Arrays are a collection of elements stored in contiguous memory locations. Advantages include constant-time access by index and memory efficiency.

Setup

Define the `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;
```

```
    }

    // Getters and setters
    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;
    }
}
```

Implementation

```
public class EmployeeManagement {
```

```
    private Employee[] employees;
    private int count;

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

    // Method to add an employee
    public void addEmployee(Employee employee) {
        if (count < employees.length) {
            employees[count++] = employee;
        } else {
            System.out.println("Array is full");
        }
    }

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

    // Method to traverse employees
    public void traverseEmployees() {
        for (int i = 0; i < count; i++) {
            System.out.println(employees[i].getEmployeeId() + " " +
employees[i].getName());
        }
    }

    // Method to delete an employee
    public void deleteEmployee(String employeeId) {
        for (int i = 0; i < count; i++) {
```

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

Analysis

Time Complexity

- **Add Operation**: O(1) if there's space, otherwise O(n) for resizing.
- **Search Operation**: O(n) in the worst case.
- **Traverse Operation**: O(n).
- **Delete Operation**: O(n) in the worst case due to shifting elements.

Limitations

Arrays have fixed size, so if the number of employees exceeds the initial capacity, the array needs to be resized, which is costly. Linked lists or dynamic arrays (like ArrayList) can be used to overcome these limitations.

Exercise 5: Task Management System

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.

```
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
    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;
    }
}
```

Implementation

```
public class SinglyLinkedList {
    private Node head;

    private class Node {
        Task task;
        Node next;
```

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

    // Method to add a 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;
        }
    }

    // Method to search a task
    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;
    }

    // Method to traverse tasks
    public void traverseTasks() {
        Node current = head;
        while (current != null) {
            System.out.println(current.task.getTaskId() + " " +
current.task.getTaskName());
```

```
            current = current.next;
        }
    }

    // Method to delete a task
    public void deleteTask(String taskId) {
        if (head == null) return;

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

        Node current = head;
        while (current.next != null &&
!current.next.task.getTaskId().equals(taskId)) {
            current = current.next;
        }

        if (current.next != null) {
            current.next = current.next.next;
        }
    }
}
```

Analysis

Time Complexity

- **Add Operation**: O(n) if appending to the end.
- **Search Operation**: O(n) in the worst case.
- **Traverse Operation**: O(n).
- **Delete Operation**: O(n) in the worst case.

Advantages

Linked lists are better than arrays for dynamic data because they allow efficient insertions and deletions without resizing or shifting elements.

Exercise 6: Library Management System

Linear Search

Simple, no need for sorted data, O(n) time complexity.

Binary Search

Efficient for sorted data, O(log n) time complexity, but requires sorting the data first.

Setup

```
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;
    }
}
```

Implementation

Linear Search

```
public Book linearSearch(Book[] books, String targetTitle) {
    for (Book book : books) {
        if (book.getTitle().equals(targetTitle)) {
            return book;
        }
    }
    return null;
}
```

Binary Search

```
import java.util.Arrays;
import java.util.Comparator;

public Book binarySearch(Book[] books, String targetTitle) {
    Arrays.sort(books, Comparator.comparing(Book::getTitle));
    int left = 0;
    int right = books.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        int comparison = books[mid].getTitle().compareTo(targetTitle);
        if (comparison == 0) {
            return books[mid];
        }
        if (comparison < 0) {
            left = mid + 1;
        } else {
            right = mid - 1;
```

```
        }
    }
    return null;
}
```

Analysis

Time Complexity

- **Linear Search**: O(n) in the worst case.
- **Binary Search**: O(log n) in the worst case, but requires sorting first (O(n log n)).

Suitable Algorithm

Use **binary search** for large, sorted datasets due to its efficiency. For small or unsorted datasets, **linear search** may be sufficient.

Exercise 7: Financial Forecasting

Understand Recursive Algorithms

Concept of Recursion

Recursion is a method of solving problems where a function calls itself as a subroutine. It simplifies complex problems by breaking them down into simpler sub-problems.

Setup

```
public class FinancialForecast {
    public double futureValue(double principal, double rate, int
years) {
        if (years == 0) {
            return principal;
        }
        return futureValue(principal * (1 + rate), rate, years - 1);
    }
}
```

Analysis

Time Complexity

The time complexity of the recursive algorithm is O(n), where n is the number of years, due to the recursive calls.

Optimization

To optimize the recursive solution and avoid excessive computation, use **memoization** to store previously computed values and reuse them.

```
import java.util.HashMap;

public class FinancialForecast {
    private HashMap<Integer, Double> memo;

    public FinancialForecast() {
        memo = new HashMap<>();
    }

    public double futureValue(double principal, double rate, int
years) {
        if (years == 0) {
            return principal;
        }
        if (memo.containsKey(years)) {
            return memo.get(years);
        }
        double result = futureValue(principal * (1 + rate), rate,
years - 1);
        memo.put(years, result);
        return result;
    }
}
```