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.

Step 1: Understand the Problem

Why Are Data Structures and Algorithms Essential?

Efficient data handling is key in inventory systems due to:

- **Speed**: Warehouses deal with thousands (or millions) of products; slow operations affect performance.
- **Search & Update Efficiency**: Retrieving or updating inventory must be fast and reliable.
- **Scalability**: Good data structures help the system handle growth without performance degradation

Step 2: Setup

- Create a Java project named: InventoryManagementSystem

Step 3: Implementation

Define 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;
    }
    public String getProductId() {
        return productId;
    }
    public String getProductName() {
        return productName;
    }
  public int getQuantity() {
        return quantity;
    }
public double getPrice() {
        return price;
    }

    public void setQuantity(int quantity) {
```

```java
        this.quantity = quantity;
    }

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

    public void display() {
        System.out.println("Product ID: " + productId +
                           ", Name: " + productName +
                           ", Quantity: " + quantity +
                           ", Price: $" + price);
    }
}
```

Inventory Management Using HashMap

```java
import java.util.HashMap;
public class InventoryManager {
    private HashMap<String, Product> inventory = new HashMap<>();
public void addProduct(Product product) {
        inventory.put(product.getProductId(), product);
        System.out.println("Product added: " + product.getProductId());
    }
 public void updateProduct(String productId, int quantity, double price) {
        Product product = inventory.get(productId);
        if (product != null) {
            product.setQuantity(quantity);
            product.setPrice(price);
            System.out.println("Product updated: " + productId);
        } else {
```

```java
            System.out.println("Product not found: " + productId);
        }
    }
public void deleteProduct(String productId) {
        if (inventory.remove(productId) != null) {
            System.out.println("Product deleted: " + productId);
        } else {
            System.out.println("Product not found: " + productId);
        }
    }
    public void displayAllProducts() {
        if (inventory.isEmpty()) {
            System.out.println("Inventory is empty.");
        } else {
            for (Product p : inventory.values()) {
                p.display();
            }
        }
    }
}
```

Test Class

// File: InventoryTest.java

```java
public class InventoryTest {
    public static void main(String[] args) {
        InventoryManager manager = new InventoryManager();

        Product p1 = new Product("P001", "Laptop", 10, 999.99);
        Product p2 = new Product("P002", "Mouse", 100, 19.99);
```

```java
        manager.addProduct(p1);

        manager.addProduct(p2);

        System.out.println();

        manager.displayAllProducts();

        System.out.println();

        manager.updateProduct("P002", 150, 17.99);

        manager.displayAllProducts();

        System.out.println();

        manager.deleteProduct("P001");

        manager.displayAllProducts();

    }

}
```

Step 4: Analysis

Time Complexity (Using HashMap)

Operation	Time Complexity
addProduct	O(1) average
updateProduct	O(1) average
deleteProduct	O(1) average
displayAll	O(n)

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.

Step 1: Understand Asymptotic Notation

What is Big O Notation?

Big O notation describes the **upper bound** of an algorithm's time or space complexity in terms of input size n. It allows developers to:

- Predict performance
- Compare scalability
- Identify bottlenecks

Best, Average, and Worst Case for Search:

Search Type	Best Case	Average Case	Worst Case
Linear Search	`O(1)` (first match)	`O(n/2)` → `O(n)`	`O(n)`
Binary Search	`O(1)` (middle match)	`O(log n)`	`O(log n)`

Note: Binary search only works on **sorted data**.

Step 2: Setup – Define the Product Class

```
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 String getProductName() {
        return productName;
    }

    public String getCategory() {
        return category;
    }

    public void display() {
        System.out.println("ID: " + productId + ", Name: " + productName + ", Category: " + category);
    }
}
```

Step 3: Implement Search Algorithms

Linear Search

```
public class SearchUtil {
    public static Product linearSearch(Product[] products, String targetName) {
        for (Product product : products) {
            if (product.getProductName().equalsIgnoreCase(targetName)) {
                return product;
            }
        }
        return null;
    }
```

Binary Search (Array must be sorted by productName)

```
    public static Product binarySearch(Product[] products, String targetName) {
        int left = 0;
        int right = products.length - 1;

        while (left <= right) {
            int mid = (left + right) / 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;
    }
}
```

Step 4: Analysis and Test Code

Test Class

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

public class SearchTest {
    public static void main(String[] args) {
        Product[] products = {
            new Product("P001", "Laptop", "Electronics"),
            new Product("P002", "Phone", "Electronics"),
            new Product("P003", "Shoes", "Fashion"),
            new Product("P004", "Book", "Education")
        };

        System.out.println(" Linear Search:");
        Product found1 = SearchUtil.linearSearch(products, "Shoes");
        if (found1 != null) found1.display();

        System.out.println("\n Binary Search:");
        Arrays.sort(products, Comparator.comparing(Product::getProductName)); // Important: sort first
        Product found2 = SearchUtil.binarySearch(products, "Shoes");
        if (found2 != null) found2.display();
    }
}
```

Time Complexity Comparison:

Operation	Time Complexity
Linear Search	`O(n)`
Binary Search	`O(log n)`

Which Is Better?

Factor	Linear Search	Binary Search
Works on unsorted?	Yes	No (must sort first)
Setup overhead	None	Must sort (`O(n log n)`)

Factor	Linear Search	Binary Search
Speed for large n	Slower	Much faster
Best for?	Small or unsorted data	Large sorted datasets

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.

Step 1: Understand Sorting Algorithms

Bubble Sort

- **How it works**: Repeatedly swaps adjacent elements if they are in the wrong order.
- **Time Complexity**:
  - Best: O(n) (already sorted)
  - Average & Worst: $O(n^2)$
- **Space**: O(1)
- **Use case**: Small datasets or educational purposes.

Quick Sort

- **How it works**: Picks a pivot, partitions the array, and recursively sorts partitions.

- **Time Complexity**:
    - Best & Average: $O(n \log n)$
    - Worst: $O(n^2)$ (rare if pivot is poorly chosen)
- **Space**: $O(\log n)$ (due to recursion)
- **Use case**: Large datasets, fast in practice.

Step 2: Setup – Define the 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 String getCustomerName() {
        return customerName;
    }

    public double getTotalPrice() {
        return totalPrice;
    }
```

```
    public void display() {
        System.out.println("Order ID: " + orderId +
                           ", Customer: " + customerName +
                           ", Total: $" + totalPrice);
    }
}
```

Step 3: Implement Sorting Algorithms

Bubble Sort

```
public class SortUtil {
    public static void bubbleSort(Order[] orders) {
        int n = orders.length;
        for (int i = 0; i < n - 1; i++) {
            boolean swapped = false;
            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;
                    swapped = true;
                }
            }
            if (!swapped) break; // Optimization
        }
    }
```

Quick Sort

```
    public static void quickSort(Order[] orders, int low, int high) {
        if (low < high) {
```

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

Step 2: Setup – Define the Order Class

```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;
    }

    public String getOrderId() {
        return orderId;
    }

    public String getCustomerName() {
        return customerName;
    }

    public double getTotalPrice() {
        return totalPrice;
    }

    public void display() {
        System.out.println("Order ID: " + orderId +
                           ", Customer: " + customerName +
                           ", Total: $" + totalPrice);
```

```
    }
}
```

Step 3: Implement Sorting Algorithms

Bubble Sort

```
public class SortUtil {
    public static void bubbleSort(Order[] orders) {
        int n = orders.length;
        for (int i = 0; i < n - 1; i++) {
            boolean swapped = false;
            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;
                    swapped = true;
                }
            }
            if (!swapped) break; // Optimization
        }
    }
```

Quick Sort

```
    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 & Test Code

Test Class

```
public class SortTest {
    public static void main(String[] args) {
        Order[] orders = {
            new Order("O001", "Alice", 250.75),
            new Order("O002", "Bob", 120.50),
            new Order("O003", "Charlie", 520.00),
            new Order("O004", "Diana", 300.00)
```

```
        };

        System.out.println("Original Orders:");
        for (Order order : orders) {
            order.display();
        }
        System.out.println("\nSorted by Bubble Sort:");
        SortUtil.bubbleSort(orders);
        for (Order order : orders) {
            order.display();
        }
        orders = new Order[] {
            new Order("O001", "Alice", 250.75),
            new Order("O002", "Bob", 120.50),
            new Order("O003", "Charlie", 520.00),
            new Order("O004", "Diana", 300.00)
        };
        System.out.println("\nSorted by Quick Sort:");
        SortUtil.quickSort(orders, 0, orders.length - 1);
        for (Order order : orders) {
            order.display();
        }
    }
}
```

Time Complexity Comparison

Algorithm	Best Case	Average Case	Worst Case	Space
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$	O(log n) (recursive stack)

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.

Step 1: Understand Array Representation

How Arrays Work in Memory:

- Arrays are **contiguous blocks of memory**.
- Each element can be accessed directly using its index (constant time).
- Indexing starts from 0 in Java.

Advantages of Arrays:

- **Fast access** using index: O(1)

- **Predictable performance**
- Low memory overhead (especially for primitive types)

Limitations of Arrays:

- Fixed size (must know capacity in advance)
- Costly insertions/deletions (especially in the middle)
- No dynamic resizing without copying to a new array

Step 2: 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;
    }

    public String getEmployeeId() {
        return employeeId;
    }

    public void display() {
        System.out.println("ID: " + employeeId + ", Name: " + name +
                           ", Position: " + position + ", Salary: $" + salary);
    }
}
```

Step 3: Implementation Using an Array

```
public class EmployeeManager {
    private Employee[] employees;
    private int count;

    public EmployeeManager(int size) {
        employees = new Employee[size];
        count = 0;
    }

    public void addEmployee(Employee emp) {
```

```
        if (count < employees.length) {
            employees[count++] = emp;
            System.out.println("Employee added: " + emp.getEmployeeId());
        } else {
            System.out.println("Employee array is full.");
        }
    }

    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 traverseEmployees() {
        if (count == 0) {
            System.out.println("No employees in the system.");
        } else {
            for (int i = 0; i < count; i++) {
                employees[i].display();
            }
        }
    }

    public void deleteEmployee(String employeeId) {
        for (int i = 0; i < count; i++) {
            if (employees[i].getEmployeeId().equals(employeeId)) {
                // Shift remaining elements left
                for (int j = i; j < count - 1; j++) {
                    employees[j] = employees[j + 1];
                }
                employees[--count] = null;
                System.out.println("Employee deleted: " + employeeId);
                return;
            }
        }
        System.out.println("Employee not found: " + employeeId);
    }
}
```

Test the System

```
public class EmployeeSystemTest {
    public static void main(String[] args) {
        EmployeeManager manager = new EmployeeManager(5);
```

```
        manager.addEmployee(new Employee("E101", "Alice", "Manager", 80000));
        manager.addEmployee(new Employee("E102", "Bob", "Developer", 60000));
        manager.addEmployee(new Employee("E103", "Charlie", "HR", 50000));

        System.out.println("\nAll Employees:");
        manager.traverseEmployees();

        System.out.println("\nSearch Employee (E102):");
        Employee found = manager.searchEmployee("E102");
        if (found != null) found.display();

        System.out.println("\nDelete Employee (E102):");
        manager.deleteEmployee("E102");

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

Step 4: Time Complexity Analysis

Operation	Time Complexity
Add	`O(1)` (at end)
Search	`O(n)`
Traverse	`O(n)`
Delete	`O(n)` (shifting elements)

When to Use Arrays

Use Arrays When:

- You know the fixed size of data
- Random access is important
- Memory efficiency matters

Prefer Alternatives (e.g., ArrayList) When:

- Size may change dynamically
- Frequent insertions/deletions are required

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.

1. Understanding Linked Lists

Singly Linked List

- Each node contains data and a reference to the next node.
- Navigation is unidirectional (forward only).
- Memory efficient, but harder to traverse backward.

Doubly Linked List

- Each node has references to both next and previous nodes.
- Easier to traverse in both directions.
- Requires more memory per node.

2. Setup: Task Class

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

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

    @Override
    public String toString() {
        return "TaskID: " + taskId + ", Name: " + taskName + ", Status: " + status;
    }
}
```

3. Implementation: Singly Linked List

```
class TaskNode {
    Task task;
    TaskNode next;

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

class TaskLinkedList {
    private TaskNode head;

    // Add task at the end
    public void addTask(Task task) {
        TaskNode newNode = new TaskNode(task);
```

```
        if (head == null) {
            head = newNode;
        } else {
            TaskNode current = head;
            while (current.next != null) {
                current = current.next;
            }
            current.next = newNode;
        }
    }
    public Task searchTask(int taskId) {
        TaskNode current = head;
        while (current != null) {
            if (current.task.taskId == taskId) {
                return current.task;
            }
            current = current.next;
        }
        return null;
    }
    public boolean deleteTask(int taskId) {
        if (head == null) return false;

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

        TaskNode current = head;
```

```
        while (current.next != null) {
            if (current.next.task.taskId == taskId) {
                current.next = current.next.next;
                return true;
            }
            current = current.next;
        }
        return false;
    }

    // Traverse and display all tasks
    public void displayTasks() {
        TaskNode current = head;
        while (current != null) {
            System.out.println(current.task);
            current = current.next;
        }
    }
}
```

4. Analysis

Time Complexity

Operation	Time Complexity	Explanation
Add Task	O(n)	Traverse to end to append
Search Task	O(n)	Linear search
Delete Task	O(n)	May need to traverse entire list
Traverse Tasks	O(n)	Visit every node

Linked List vs Array

Feature	Linked List	Array
Dynamic Size	Yes (can grow/shrink as needed)	No (fixed size or costly resizing)
Insert/Delete	Efficient at front/middle (O(1)/O(n))	Costly (O(n) for shift operations)
Random Access	No (O(n) access)	Yes (O(1) access by index)
Memory Usage	More (pointers per node)	Less (just elements)

Example Usage (Main Method)

```
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, "Implement Backend", "In Progress"));
        taskList.addTask(new Task(3, "Testing", "Not Started"));

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

        System.out.println("\nSearching Task with ID 2:");
        Task task = taskList.searchTask(2);
        System.out.println(task != null ? task : "Task not found.");

        System.out.println("\nDeleting Task with ID 1:");
        boolean deleted = taskList.deleteTask(1);
        System.out.println("Deleted: " + deleted);

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

```
    }
}
```

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.

1. Understand Search Algorithms

Linear Search

- Iterates through each element until the target is found.
- Works on unsorted data.
- **Time Complexity:**
    - Best: O(1)
    - Worst: O(n)

Binary Search

- Efficient search on **sorted data** by repeatedly dividing the search interval in half.
- **Time Complexity:**
    - Best: O(1)
    - Worst: O(log n)

2. Setup: Book Class

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

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

    @Override
    public String toString() {
        return "BookID: " + bookId + ", Title: " + title + ", Author: " + author;
    }
}
```

3. Implementation

Linear Search (by Title)

```
public static Book linearSearchByTitle(List<Book> books, String title) {
    for (Book book : books) {
        if (book.title.equalsIgnoreCase(title)) {
            return book;
        }
    }
    return null;
}
```

Binary Search (by Title) – Requires Sorted List

```
public static Book binarySearchByTitle(List<Book> books, String title) {
```

```
    int left = 0, right = books.size() - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;
        Book midBook = books.get(mid);
        int comparison = midBook.title.compareToIgnoreCase(title);

        if (comparison == 0) {
            return midBook;
        } else if (comparison < 0) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return null;
}
```

Sort Method (for Binary Search)

```
public static void sortBooksByTitle(List<Book> books) {
    books.sort((b1, b2) -> b1.title.compareToIgnoreCase(b2.title));
}
```

4. Analysis

Search Algorithm	Time Complexity (Avg/Worst)	Requirements	Use Case
Linear Search	O(n) / O(n)	No sorting needed	Small datasets or unsorted data
Binary Search	O(log n) / O(log n)	Requires sorted list	Large, sorted datasets for faster lookups

Main Method Example

```
import java.util.*;
public class LibraryManagementSystem {

    public static void main(String[] args) {
        List<Book> books = new ArrayList<>();
        books.add(new Book(101, "Java Programming", "John Smith"));
        books.add(new Book(102, "Data Structures", "Alice Johnson"));
        books.add(new Book(103, "Algorithms", "Robert Martin"));
        books.add(new Book(104, "Database Systems", "Chris Evans"));
        System.out.println("Linear Search for 'Algorithms':");
        Book found = linearSearchByTitle(books, "Algorithms");
        System.out.println(found != null ? found : "Book not found");
        sortBooksByTitle(books);
        System.out.println("\nBinary Search for 'Algorithms':");
        Book foundBinary = binarySearchByTitle(books, "Algorithms");
        System.out.println(foundBinary != null ? foundBinary : "Book not found");
    }
}
```

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.

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.

3. Implementation: Recursive Algorithm in Java

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

    public static void main(String[] args) {
        double principal = 10000; // initial investment
        double rate = 0.05;       // 5% annual growth
        int years = 5;

        double result = futureValue(principal, rate, years);
        System.out.printf("Future value after %d years: %.2f\n", years, result);
    }
}
public class FinancialForecast {
    public static double futureValue(double principal, double rate, int years) {
        if (years == 0) {
            return principal;
        }
        return (1 + rate) * futureValue(principal, rate, years - 1);
    }

    public static void main(String[] args) {
        double principal = 10000;
        double rate = 0.05
        int years = 5;
```

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

4. Analysis

Time Complexity

- **T(n) = T(n-1) + O(1) → O(n)**
- The function is called once for each year, leading to **linear time**.

Optimization

To avoid **excessive computation or stack overflow**, especially with large n, we can:

1. **Use Iteration:** Convert to a loop.
2. **Use Memoization:** Store results of subproblems.
3. **Use Exponentiation by Squaring:** For faster computation:
   Reduces time to **O(log n)**.

Bonus: Optimized Recursive Version (Exponentiation by Squaring)

```
public static double futureValueOptimized(double principal, double rate, int years) {
    return principal * power(1 + rate, years);
}
public static double power(double base, int exp) {
    if (exp == 0) return 1;
    if (exp % 2 == 0) {
        double half = power(base, exp / 2);
        return half * half;
    } else {
        return base * power(base, exp - 1);
    }
}
```

Conclusion

Version	**Time Complexity**	**Space Complexity**	**Remarks**
Simple Recursion	O(n)	O(n)	Easy to implement, not optimal
Optimized Recursion	O(log n)	O(log n)	Much faster for large n
Iterative	O(n)	O(1)	Best for memory-limited devices