SOLUTION

Exercise 1: Inventory Management System

1. Understand the Problem

Why Data Structures and Algorithms are Essential: In handling large inventories, efficient data storage and retrieval are crucial due to:

- **Scalability:** As inventory size grows, operations need to scale efficiently.
- **Performance:** Operations such as searching, updating, and deleting must be optimized to handle large datasets quickly.
- **Resource Management:** Efficient use of memory and processing power is important to ensure smooth system performance.

Types of Data Structures Suitable:

1. **ArrayList (or List in Java):**
    - **Advantages:** Provides dynamic resizing and simple indexing.
    - **Disadvantages:** Slower insertions and deletions, as these operations might require shifting elements.
2. **HashMap (or Dictionary in Python):**
    - **Advantages:** Offers average O(1) time complexity for insertions, deletions, and lookups due to hashing.
    - **Disadvantages:** Not ordered; requires handling of hash collisions.
3. **Binary Search Tree (BST) or Balanced Trees (e.g., AVL Tree):**
    - **Advantages:** Provides O(log n) time complexity for search, insertion, and deletion operations, maintaining a sorted order.
    - **Disadvantages:** More complex to implement and manage compared to HashMap.
4. **Priority Queue (if inventory needs sorting by priority):**
    - **Advantages:** Efficiently manages elements with priorities.
    - **Disadvantages:** Complexity depends on the specific implementation.
5. **LinkedList** : It allows efficient insertion and deletion but has linear time complexity for indexing.

2. Setup

Create a New Project:

1. **Initialize Project:**
    - Use an IDE like IntelliJ IDEA, Eclipse, or Visual Studio Code.
    - Create a new project directory and set up a version control system (e.g., Git).
2. **Project Structure:**
    - Create directories for source code, test cases, and documentation.

3. Implementation

Define a Class Product:

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

    // Constructor
    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; }

}
```

Using HashMap for efficient storage and retrieval:

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

public class InventorySystem {
    private Map<String, Product> inventory;

    // Constructor
    public InventorySystem() {
        this.inventory = new HashMap<>();
    }

    // Add Product
    public void addProduct(Product product) {
        inventory.put(product.getProductId(), product);
    }

    // Update Product
    public void updateProduct(String productId, Product newProduct) {
        if (inventory.containsKey(productId)) {
            inventory.put(productId, newProduct);
        }
    }

    // Delete Product
    public void deleteProduct(String productId) {
```

```
        inventory.remove(productId);
    }

    // Get Product
    public Product getProduct(String productId) {
        return inventory.get(productId);
    }
}
```

4. Analysis

Time Complexity:

- **Add Operation:**
  - **HashMap:** O(1) average time complexity. Inserting a new product involves hashing the key and storing it in the map.
  - **ArrayList:** O(1) if adding at the end; O(n) if inserting at a specific index.
- **Update Operation:**
  - **HashMap:** O(1) average time complexity. Updating involves replacing the value associated with a key.
  - **ArrayList:** O(n) as it requires searching for the product, then updating.
- **Delete Operation:**
  - **HashMap:** O(1) average time complexity. Removing a product involves hashing and deleting the key-value pair.
  - **ArrayList:** O(n) as it requires searching for the product, then removing it and shifting elements.

Optimizations:

- **HashMap:** Ensure a good hash function and handle collisions to maintain O(1) performance.
- **Memory Management:** Consider using more space-efficient data structures or compressing data if memory usage becomes a concern.
- **Concurrency:** Use concurrent data structures or synchronize access if multiple threads are involved in accessing or modifying the inventory.

Exercise 2: E-commerce Platform Search Function

1. Understand Asymptotic Notation

Big O Notation:

- **Definition: Big O notation is a mathematical representation that describes the upper bound of an algorithm's runtime or space requirements in terms of the size of the input. It provides a way to express the efficiency of an algorithm in the worst-case scenario.**
- **Purpose: It helps in comparing algorithms by focusing on their growth rates relative to input size, allowing you to choose the most efficient algorithm for large datasets.**

Best, Average, and Worst-Case Scenarios:

- **Best Case:**
  - **Linear Search: O(1) if the target element is the first element in the array.**
  - **Binary Search: O(1) if the target element is the middle element of the array.**
- **Average Case:**
  - **Linear Search: O(n) where n is the number of elements. On average, you might have to look at half the elements.**
  - **Binary Search: O(log n) where n is the number of elements. The search space is halved with each comparison.**
- **Worst Case:**
  - **Linear Search: O(n) where n is the number of elements. You might have to look through all elements if the target is not found.**
  - **Binary Search: O(log n) where n is the number of elements. The search space is reduced exponentially with each comparison, but requires the array to be sorted.**

2. Setup

Create a Class Product:

```
public class Product {
    private String productId;
    private String productName;
    private String category;
```

```
    // Constructor
    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; }
}
```

3. Implementation

Linear Search Algorithm:

```
public class SearchUtil {

    // Linear Search
    public static Product linearSearch(Product[] products, String searchTerm) {
        for (Product product : products) {
            if (product.getProductName().equalsIgnoreCase(searchTerm)) {
                return product;
            }
        }
        return null; // Product not found
    }
```

```
}
```

Binary Search Algorithm (requires sorted array by productName):

```
import java.util.Arrays;

public class SearchUtil {

    // Binary Search
    public static Product binarySearch(Product[] products, String searchTerm) {
        int left = 0;
        int right = products.length - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;
            int cmp = products[mid].getProductName().compareToIgnoreCase(searchTerm);

            if (cmp == 0) {
                return products[mid]; // Product found
            } else if (cmp < 0) {
                left = mid + 1; // Search in the right half
            } else {
                right = mid - 1; // Search in the left half
            }
        }
        return null; // Product not found
    }

    // Utility method to sort the array before binary search
    public static void sortProductsByName(Product[] products) {
        Arrays.sort(products, (a, b) ->
a.getProductName().compareToIgnoreCase(b.getProductName()));
```

}

}

4. Analysis

Time Complexity Comparison:

- **Linear Search:**
  - **Time Complexity: O(n)**
  - **Space Complexity: O(1)**
  - **Advantages: Simple implementation; does not require sorted data.**
  - **Disadvantages: Inefficient for large datasets.**
- **Binary Search:**
  - **Time Complexity: O(log n)**
  - **Space Complexity: O(1)**
  - **Advantages: Efficient for large datasets; requires sorted data.**
  - **Disadvantages: Requires preprocessing to sort the data, which is O(n log n).**

Suitability for E-commerce Platform:

- **Binary Search is generally more suitable for scenarios where the dataset is large and the data is sorted, as it offers faster search times with O(log n) complexity compared to O(n) for linear search. The trade-off is the need for sorted data and possibly additional preprocessing.**
- **Linear Search might be used in smaller datasets or when data is unsorted, but its performance degrades with larger datasets.**

Exercise 3: Sorting Customer Orders

1. Understand Sorting Algorithms

Bubble Sort:

- **Description: Bubble Sort repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the list is sorted.**
- **Time Complexity:**
  - **Best Case: O(n) (when the array is already sorted, and only one pass is needed)**
  - **Average Case: O(n^2)**
  - **Worst Case: O(n^2) (when the array is sorted in reverse order)**
- **Space Complexity: O(1)**
- **Characteristics: Simple to implement but inefficient for large datasets due to its O(n^2) time complexity.**

Insertion Sort:

- **Description: Insertion Sort builds the final sorted array one item at a time. It takes each element and inserts it into its correct position in the already sorted part of the array.**
- **Time Complexity:**
  - **Best Case: O(n) (when the array is already sorted)**
  - **Average Case: O(n^2)**
  - **Worst Case: O(n^2) (when the array is sorted in reverse order)**
- **Space Complexity: O(1)**
- **Characteristics: More efficient than Bubble Sort for small datasets or nearly sorted data.**

Quick Sort:

- **Description: Quick Sort is a divide-and-conquer algorithm that selects a 'pivot' element and partitions the array into two sub-arrays, according to whether elements are less than or greater than the pivot. It recursively applies the same process to the sub-arrays.**
- **Time Complexity:**
  - **Best Case: O(n log n) (when the pivot is well-chosen)**
  - **Average Case: O(n log n)**

- Worst Case: O(n^2) (when the pivot is the smallest or largest element every time, though this can be mitigated with good pivot selection)

- Space Complexity: O(log n) (due to recursion stack)
- Characteristics: Very efficient for large datasets; commonly used in practice due to its average-case efficiency.

Merge Sort:

- Description: Merge Sort is another divide-and-conquer algorithm that divides the array into two halves, recursively sorts each half, and then merges the sorted halves.
- Time Complexity:
  - Best Case: O(n log n)
  - Average Case: O(n log n)
  - Worst Case: O(n log n)
- Space Complexity: O(n) (due to the need for temporary arrays during the merge process)
- Characteristics: Stable sort with consistent performance, but uses more memory compared to Quick Sort.

2. Setup

Create a Class Order:

```
public class Order {
    private String orderId;
    private String customerName;
    private double totalPrice;

    // Constructor
    public Order(String orderId, String customerName, double totalPrice) {
        this.orderId = orderId;
        this.customerName = customerName;
        this.totalPrice = totalPrice;
```

```
    }

    // Getters
    public String getOrderId() { return orderId; }
    public String getCustomerName() { return customerName; }
    public double getTotalPrice() { return totalPrice; }

    // Setters
    public void setOrderId(String orderId) { this.orderId = orderId; }
    public void setCustomerName(String customerName) { this.customerName =
customerName; }
    public void setTotalPrice(double totalPrice) { this.totalPrice = totalPrice; }
}
```

3. Implementation

Bubble Sort Implementation:

```
public class SortUtil {

    public static void bubbleSort(Order[] orders) {
        int n = orders.length;
        boolean swapped;
        for (int i = 0; i < n - 1; i++) {
            swapped = false;
            for (int j = 0; j < n - 1 - i; 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;
```

```
                swapped = true;
            }
        }
        // If no two elements were swapped by inner loop, break
        if (!swapped) break;
    }
  }
}
```

Quick Sort Implementation:

```
public class SortUtil {

  // 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);
    }
  }

  // Partition method
  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++;
```

```
            // 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;
    }
}
```

4. Analysis

Time Complexity Comparison:

- **Bubble Sort:**
    - **Best Case:** O(n) (when the array is already sorted)
    - **Average Case:** O(n^2)
    - **Worst Case:** O(n^2) (when the array is sorted in reverse order)
    - **Space Complexity:** O(1)
- **Quick Sort:**
    - **Best Case:** O(n log n) (with good pivot selection)
    - **Average Case:** O(n log n)
    - **Worst Case:** O(n^2) (with poor pivot selection)
    - **Space Complexity:** O(log n) (due to recursion stack)

Why Quick Sort is Generally Preferred:

1. **Efficiency:** Quick Sort is faster on average than Bubble Sort due to its O(n log n) time complexity compared to Bubble Sort's O(n^2).
2. **Scalability:** Quick Sort handles large datasets more efficiently and performs better in practice, even though it has a worse-case time complexity.
3. **Memory Usage:** Quick Sort typically uses less additional memory than Merge Sort (which has O(n) space complexity), making it more suitable for large arrays.

Exercise 4: Employee Management System

1. Understand Array Representation

Array Representation in Memory:

- **Memory Layout:** Arrays are stored in contiguous memory locations. Each element is placed next to the previous one, which makes accessing elements by index very efficient.
- **Advantages:**
    - **Constant Time Access:** Direct access to elements by index in O(1) time due to their contiguous placement.
    - **Efficient Use of Space:** No overhead for pointers or additional data structures.
    - **Simplicity:** Arrays are simple to implement and understand, with predictable performance.

Drawbacks:

- **Fixed Size:** Arrays have a fixed size, which means their size must be known at the time of creation and cannot be changed dynamically.
- **Inefficient Insertions/Deletions:** Adding or removing elements involves shifting elements to maintain order, which can be inefficient (O(n) time complexity).

2. Setup

Create a Class Employee:

```
public class Employee {
    private String employeeId;
    private String name;
    private String position;
    private double salary;

    // Constructor
    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; }
}
```

3. Implementation

Use an Array to Store Employee Records:

```
public class EmployeeManagementSystem {
    private Employee[] employees;
    private int size;
    private static final int INITIAL_CAPACITY = 10;

    // Constructor
    public EmployeeManagementSystem() {
        this.employees = new Employee[INITIAL_CAPACITY];
        this.size = 0;
    }

    // Method to add an employee
    public void addEmployee(Employee employee) {
        if (size == employees.length) {
            // Resize the array if needed
```

```
        resize();
    }
    employees[size++] = employee;
}

// Method to search for an employee by ID
public Employee searchEmployee(String employeeId) {
    for (int i = 0; i < size; i++) {
        if (employees[i].getEmployeeId().equals(employeeId)) {
            return employees[i];
        }
    }
    return null; // Employee not found
}

// Method to traverse and display all employees
public void traverseEmployees() {
    for (int i = 0; i < size; i++) {
        Employee emp = employees[i];
        System.out.println("ID: " + emp.getEmployeeId() + ", Name: " + emp.getName() +
            ", Position: " + emp.getPosition() + ", Salary: " + emp.getSalary());
    }
}

// Method to delete an employee by ID
public void deleteEmployee(String employeeId) {
    int index = -1;
    for (int i = 0; i < size; i++) {
        if (employees[i].getEmployeeId().equals(employeeId)) {
```

```
                index = i;
                break;
            }
        }
        if (index != -1) {
            // Shift elements to remove the employee
            for (int i = index; i < size - 1; i++) {
                employees[i] = employees[i + 1];
            }
            employees[size - 1] = null; // Clear the last element
            size--;
        }
    }

    // Resize the array to accommodate more elements
    private void resize() {
        int newCapacity = employees.length * 2;
        Employee[] newArray = new Employee[newCapacity];
        System.arraycopy(employees, 0, newArray, 0, size);
        employees = newArray;
    }
}
```

4. Analysis

Time Complexity Analysis:

- **Add Operation:**
  - **Average Case:** O(1) (when there is space available)
  - **Worst Case:** O(n) (when resizing is needed, which involves copying elements to a new array)
- **Search Operation:**

    - **Time Complexity:** O(n) (in the worst case, you may need to search through the entire array)
- **Traverse Operation:**
    - **Time Complexity:** O(n) (you need to visit each element once)
- **Delete Operation:**
    - **Average Case:** O(n) (finding the element takes O(n), and shifting elements takes O(n) in the worst case)

Limitations of Arrays:

- **Fixed Size:** Arrays have a fixed size which needs to be known in advance or require resizing when the array is full.
- **Inefficient Insertions/Deletions:** Inserting or deleting elements requires shifting elements, making it less efficient compared to data structures like LinkedLists or dynamic arrays with amortized resizing strategies.

When to Use Arrays:

- **Predictable Size:** When the number of elements is known ahead of time or doesn't change frequently.
- **Fast Access:** When you need fast, constant-time access to elements by index.
- **Simple Data:** When the dataset is small or simple, and you don't need dynamic resizing or frequent insertions/deletions.

Exercise 5: Task Management System

1. Understand Linked Lists

Types of Linked Lists:

1. **Singly Linked List:**
    - **Description:** A singly linked list consists of nodes where each node contains data and a reference (or pointer) to the next node in the sequence. It is unidirectional.
    - **Advantages:**
        - **Dynamic Size:** Can easily grow and shrink in size by adding or removing nodes without needing to resize.
        - **Efficient Insertions/Deletions:** Inserting or deleting nodes is efficient if you have a reference to the node before the insertion or deletion point.
2. **Doubly Linked List:**
    - **Description:** A doubly linked list consists of nodes where each node contains data and two references (or pointers): one to the next node and one to the previous node. It is bidirectional.
    - **Advantages:**
        - **Bidirectional Traversal:** You can traverse the list in both forward and backward directions.
        - **Efficient Deletions:** Easier to delete a node if you have a reference to the node itself, as it has pointers to both previous and next nodes.

2. Setup

Create a Class Task:

```
public class Task {
    private String taskId;
    private String taskName;
    private String status;

    // Constructor
    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; }

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

3. Implementation

Singly Linked List Implementation:

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

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

```
    }
}

private Node head;

// Constructor
public TaskManager() {
    this.head = 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 for a task by ID
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; // Task not found
}

// Method to traverse and display all tasks
public void traverseTasks() {
    Node current = head;
    while (current != null) {
        System.out.println(current.task);
        current = current.next;
    }
}

// Method to delete a task by ID
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;
        }
    }
}
```

4. Analysis

Time Complexity Analysis:

- **Add Operation:**
  - **Time Complexity:** O(n) in the worst case (when adding at the end of the list and needing to traverse to find the last node). O(1) if adding at the head.
  - **Space Complexity:** O(1) (excluding space for the list).
- **Search Operation:**
  - **Time Complexity:** O(n) (you may need to traverse the entire list to find the task).
  - **Space Complexity:** O(1).
- **Traverse Operation:**
  - **Time Complexity:** O(n) (you need to visit each node once).
  - **Space Complexity:** O(1).
- **Delete Operation:**
  - **Time Complexity:** O(n) (you need to find the node to delete, then update the references).
  - **Space Complexity:** O(1).

Advantages of Linked Lists over Arrays for Dynamic Data:

- **Dynamic Size:** Linked lists can grow and shrink dynamically without needing to resize like arrays.
- **Efficient Insertions/Deletions:** Inserting or deleting nodes is more efficient in a linked list (O(1) time) compared to an array where shifting elements is needed (O(n) time).
- **Memory Utilization:** Linked lists use memory only for the elements currently in the list, whereas arrays might have unused allocated space if the size is overestimated.

When to Use Linked Lists:

- **Frequent Insertions/Deletions:** When you need to frequently insert or delete elements from the list.
- **Unknown Size:** When the size of the dataset is not known in advance or changes frequently.
- **Less Concern About Index Access:** When direct index access is not crucial, as linked lists do not support O(1) access to arbitrary elements.

Exercise 6: Library Management System

1. Understand Search Algorithms

Linear Search:

- **Description:** Linear search scans each element in the list sequentially until it finds the target value or reaches the end of the list.
- **Time Complexity:**
    - **Best Case:** O(1) (when the target is the first element)
    - **Average Case:** O(n) (where n is the number of elements)
    - **Worst Case:** O(n) (when the target is the last element or not present)
- **Space Complexity:** O(1)
- **Characteristics:** Simple to implement and works on unsorted data but can be inefficient for large datasets.

Binary Search:

- **Description:** Binary search works on sorted data by repeatedly dividing the search interval in half. It compares the target value with the middle element and eliminates half of the remaining elements from consideration.
- **Time Complexity:**
    - **Best Case:** O(1) (when the target is the middle element)
    - **Average Case:** O(log n) (where n is the number of elements)
    - **Worst Case:** O(log n) (if the target is not present, or is at the end)
- **Space Complexity:** O(1) for iterative implementation or O(log n) for recursive implementation (due to recursion stack)
- **Characteristics:** Much faster than linear search for large datasets but requires the data to be sorted.

2. Setup

Create a Class Book:

```
public class Book {
    private String bookId;
    private String title;
```

```
    private String author;

    // Constructor
    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; }

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

3. Implementation

Linear Search Implementation:

```
import java.util.List;

public class LibraryManager {
```

```
    // Method to perform linear search
    public Book linearSearchByTitle(List<Book> books, String title) {
        for (Book book : books) {
            if (book.getTitle().equalsIgnoreCase(title)) {
                return book;
            }
        }
        return null; // Book not found
    }
}
```

Binary Search Implementation:

To use binary search, the list must be sorted by title. Here's an implementation assuming the list is sorted:

```
import java.util.List;

public class LibraryManager {

    // Method to perform binary search
    public Book binarySearchByTitle(List<Book> books, String title) {
        int low = 0;
        int high = books.size() - 1;

        while (low <= high) {
            int mid = low + (high - low) / 2;
            Book midBook = books.get(mid);

            int comparison = midBook.getTitle().compareToIgnoreCase(title);
            if (comparison == 0) {
                return midBook;
```

```
        } else if (comparison < 0) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return null; // Book not found
  }
}
```

4. Analysis

Time Complexity Comparison:

- **Linear Search:**
  - **Best Case:** O(1)
  - **Average Case:** O(n)
  - **Worst Case:** O(n)
  - **Space Complexity:** O(1)
- **Binary Search:**
  - **Best Case:** O(1)
  - **Average Case:** O(log n)
  - **Worst Case:** O(log n)
  - **Space Complexity:** O(1) for iterative, O(log n) for recursive implementation

When to Use Each Algorithm:

- **Linear Search:**
  - **Use When:**
    - The dataset is small.
    - The dataset is unsorted or changes frequently.
    - Simplicity is more important than performance.
  - **Advantages:** Can be used on unsorted data; easy to implement.
- **Binary Search:**

- **Use When:**
  - The dataset is large.
  - The data is sorted (or can be sorted).
  - Faster search performance is required.
- **Advantages:** Much faster than linear search for large datasets; efficient for sorted data.

Exercise 7: Financial Forecasting

1. Understand Recursive Algorithms

Concept of Recursion:

- **Definition: Recursion is a technique where a function calls itself in order to solve smaller instances of the same problem. It is particularly useful for problems that can be broken down into smaller, similar problems.**
- **Base Case and Recursive Case: A recursive function generally has two parts:**
  - **Base Case: The condition under which the recursion stops. It provides a simple, non-recursive result.**
  - **Recursive Case: The part of the function where it calls itself with a modified argument, working towards the base case.**
- **Advantages:**
  - **Simplicity: Recursive solutions can be more intuitive and easier to understand compared to iterative solutions for problems like tree traversals or factorial calculations.**
  - **Reduction of Code: Often results in more concise code.**

Drawbacks:

- **Stack Overflow: Deep recursion can lead to stack overflow errors if the recursion depth is too large.**
- **Performance Issues: Recursive solutions can be less efficient due to repeated calculations if not optimized.**

2. Setup

Create a Method to Calculate Future Value Using Recursion:

Let's say the future value of an investment is predicted based on past growth rates. We can use a recursive approach to compute the future value.

Assumptions:

- **initialValue: The initial amount of money.**
- **growthRate: The rate of growth per period (e.g., 5% would be 0.05).**
- **periods: The number of future periods to predict.**

3. Implementation

Recursive Algorithm to Predict Future Values:

```
public class FinancialForecasting {
```

```
    // Method to calculate future value recursively
    public double calculateFutureValue(double initialValue, double growthRate, int periods)
{
        // Base case: no more periods to forecast
        if (periods == 0) {
            return initialValue;
        }
        // Recursive case: calculate future value
        return calculateFutureValue(initialValue * (1 + growthRate), growthRate, periods - 1);
    }

    public static void main(String[] args) {
        FinancialForecasting forecasting = new FinancialForecasting();
        double initialValue = 1000; // initial investment
        double growthRate = 0.05; // 5% growth rate
        int periods = 10; // forecast for 10 periods

        double futureValue = forecasting.calculateFutureValue(initialValue, growthRate,
periods);
        System.out.println("Future Value: " + futureValue);
    }
}
```

4. Analysis

Time Complexity:

- **Time Complexity of Recursive Algorithm: O(n), where n is the number of periods. Each recursive call reduces the number of periods by 1, leading to a linear number of recursive calls.**

Optimization:

- **Memoization: To avoid recalculating results for the same inputs, you can use memoization to store intermediate results. This technique is especially useful for problems with overlapping subproblems.**
- **Iterative Approach: For simple problems like this one, an iterative approach might be more efficient. Here's how you could implement it:**

```
public class FinancialForecasting {


  // Method to calculate future value iteratively

  public double calculateFutureValueIterative(double initialValue, double growthRate, int
periods) {

    double futureValue = initialValue;

    for (int i = 0; i < periods; i++) {

      futureValue *= (1 + growthRate);

    }

    return futureValue;

  }


  public static void main(String[] args) {

    FinancialForecasting forecasting = new FinancialForecasting();

    double initialValue = 1000; // initial investment

    double growthRate = 0.05; // 5% growth rate

    int periods = 10; // forecast for 10 periods


    double futureValueRecursive = forecasting.calculateFutureValue(initialValue,
growthRate, periods);

    double futureValueIterative = forecasting.calculateFutureValueIterative(initialValue,
growthRate, periods);


    System.out.println("Future Value (Recursive): " + futureValueRecursive);

    System.out.println("Future Value (Iterative): " + futureValueIterative);
```

```
    }
}
```

Explanation:

- **Recursive Approach: While elegant, it may not be the most efficient for this particular problem due to its linear time complexity and potential stack overflow issues with deep recursion.**
- **Iterative Approach: More practical for problems with a straightforward recurrence relation, as it avoids the overhead associated with recursive function calls and is generally more efficient in terms of both time and space complexity.**