1.)

Exercise 1: Inventory Management System:

Efficient handling of large inventories requires optimal data storage and retrieval mechanisms. Proper data structures and algorithms allow us to:

Reduce the time complexity: Efficiently manage the operations of adding, updating, and deleting products.

Enhance performance: Quick access to products, minimizing the time taken for search operations.

Manage memory usage: Optimize the storage requirements and avoid memory overheads.

Suitable Data Structures

For an inventory management system, the following data structures are suitable:

ArrayList: Useful for maintaining an ordered list of products. Suitable for scenarios where the size of the inventory doesn't change frequently.

HashMap: Ideal for scenarios requiring quick lookups, updates, and deletions. It allows O(1) average-time complexity for these operations.

LinkedList: Can be used if the inventory operations involve frequent insertions and deletions.

Given the requirements, HashMap would be a good choice due to its efficient average-time complexity for key-based operations.

2. Setup

Create a new project in your preferred programming environment. For example, in Java, you can set up a Maven or Gradle project.

3. Implementation:

```
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;
    }
}
```

Use a HashMap to store the products.

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

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

    public InventoryManagementSystem() {
        this.inventory = new HashMap<>();
    }

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

    // Method to update a product
    public void updateProduct(String productId, int quantity, double price) {
        Product product = inventory.get(productId);
        if (product != null) {
            product.setQuantity(quantity);
            product.setPrice(price);
        }
    }

    // Method to delete a product
    public void deleteProduct(String productId) {
        inventory.remove(productId);
    }

    // Method to retrieve a product
    public Product getProduct(String productId) {
        return inventory.get(productId);
    }
}
```

4. Analysis

Time Complexity

Add Operation:

Using HashMap: O(1) average time complexity.

Update Operation:

Using HashMap: O(1) average time complexity.

Delete Operation:

Using HashMap: O(1) average time complexity.

Optimization

Indexing: Ensure product IDs are unique and use them as keys in the HashMap for O(1) average-time complexity.

Concurrency: For a multi-threaded environment, consider using concurrent data structures like ConcurrentHashMap.

Bulk Operations: Implement bulk add, update, and delete operations to reduce overhead when handling large inventories.

By following these steps, we ensure efficient data storage and retrieval in our inventory management system.

2.)

<u>Exercise 2: E-commerce Platform Search Function</u>

Understand Asymptotic Notation

Big O Notation

Big O notation is a mathematical notation used to describe the upper bound of an algorithm's runtime or space requirements in terms of input size (n). It provides an asymptotic analysis of the algorithm, indicating how the runtime or space grows as the input size increases.

Best Case: The scenario where the algorithm performs the minimum number of operations.

Average Case: The expected scenario considering all possible inputs.

Worst Case: The scenario where the algorithm performs the maximum number of operations.

For search operations:

Linear Search: O(n) in the worst case, where n is the number of elements.

Binary Search: O(log n) in the worst case, assuming the array is sorted.

2. Setup:

```
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
    public String getProductId() {
        return productId;
    }

    public String getProductName() {
        return productName;
    }

    public String getCategory() {
        return category;
    }
}
```

3. Implementation:

```
 public class Search {

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

Binary search:

```
import java.util.Arrays;

public class Search {

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

        // Sort products array by productName
        Arrays.sort(products, (p1, p2) ->
p1.getProductName().compareToIgnoreCase(p2.getProductName()));

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

            if (result == 0) {
                return products[mid];
            } else if (result < 0) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return null; // If product is not found
    }
}
```

4. 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, assuming the array is sorted.

Comparison and Suitability

Linear Search: Suitable for small datasets or unsorted data. It is simple to implement but inefficient for large datasets due to O(n) time complexity.

Binary Search: More suitable for large datasets where the data is sorted. It is efficient with O(log n) time complexity, making it faster than linear search for large datasets.

Conclusion:

For an e-commerce platform, where quick search functionality is crucial, binary search is more suitable due to its lower time complexity compared to linear search. Ensuring the products array is sorted will optimize search operations, providing a better user experience.

<u>3.) Sorting Customer Orders:</u>

1. Understand Sorting Algorithms

Bubble Sort

Description: A simple comparison-based algorithm where each pair of adjacent elements is compared and swapped if they are in the wrong order. This process is repeated until the array is sorted.

Time Complexity:

Best Case: O(n)

Average Case: O(n^2)

Worst Case: O(n^2)

Insertion Sort

Description: Builds the final sorted array one item at a time, with each new element being compared and placed in its correct position within the already sorted part of the array.

Time Complexity:

Best Case: O(n)

Average Case: O(n^2)

Worst Case: O(n^2)

Quick Sort

Description: 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 then recursively sorts the sub-arrays.

Time Complexity:

Best Case: O(n log n)

Average Case: O(n log n)

Worst Case: O(n^2) (rare, depends on pivot selection)

Merge Sort

Description: Another divide-and-conquer algorithm that divides the array into two halves, sorts them, and then merges the sorted halves back together.

Time Complexity:

Best Case: O(n log n)

Average Case: O(n log n)

Worst Case: O(n log n)

2. Setup

Create a class Order with attributes like orderId, customerName, and totalPrice.

```
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;
    }
}
```

3. Implementation

Bubble Sort

```
public class SortOrders {

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

Quick Sort:

```
public class SortOrders {

    // Quick Sort Algorithm
    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++;
                // 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

Bubble Sort:

Best Case: O(n)

Average Case: O(n^2)

Worst Case: O(n^2)

Quick Sort:

Best Case: O(n log n)

Average Case: O(n log n)

Worst Case: O(n^2) (rare, depends on pivot selection)

Performance Comparison

Bubble Sort: Simple but inefficient for large datasets due to O(n^2) time complexity. It is generally not used for large lists.

Quick Sort: Efficient for large datasets with O(n log n) average time complexity. It is generally preferred over Bubble Sort due to its significantly better performance in most cases.

Conclusion

Quick Sort is generally preferred over Bubble Sort for sorting large datasets due to its better average and worst-case time complexity. While Bubble Sort might be easier to implement and understand, its inefficiency for large data makes it unsuitable for high-performance requirements like sorting customer orders on an e-commerce platform.

Test Case

Here's how you might set up a test case:

```
public class Main {

    public static void main(String[] args) {

        Order[] orders = {

            new Order("1", "Alice", 250.0),

            new Order("2", "Bob", 150.0),

            new Order("3", "Charlie", 300.0),

            new Order("4", "David", 200.0)

        };


        // Bubble Sort Test

        SortOrders.bubbleSort(orders);

        System.out.println("Orders sorted by Bubble Sort:");

        for (Order order : orders) {

            System.out.println(order.getCustomerName() + ": $" + order.getTotalPrice());

        }
        // Quick Sort Test
        orders = new Order[]{
            new Order("1", "Alice", 250.0),
            new Order("2", "Bob", 150.0),
            new Order("3", "Charlie", 300.0),
            new Order("4", "David", 200.0)
        };
        SortOrders.quickSort(orders, 0, orders.length - 1);
        System.out.println("Orders sorted by Quick Sort:");
        for (Order order : orders) {
            System.out.println(order.getCustomerName() + ": $" + order.getTotalPrice());
        }
    }
}
```

This code sets up an array of orders, sorts them using both Bubble Sort and Quick Sort, and prints the sorted orders, demonstrating the functionality and performance of both algorithms.

4.)

Exercise 4: Employee Management System

Array Representation in Memory

Arrays are contiguous blocks of memory where each element is stored next to the others. This allows constant-time access to any element by its index. The advantages of arrays include:

Direct Access: Elements can be accessed directly using their index, making lookups O(1).

Predictable Memory Usage: The size of the array is fixed, which makes memory allocation straightforward.

Disadvantages:

Fixed Size: Once an array is created, its size cannot be changed.

Memory Allocation: If the array is too large, it can cause memory issues.

2. Setup

Create a class Employee with attributes like employeeId, name, position, and salary.

```
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 count;

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

    // Method to add an employee
    public boolean addEmployee(Employee employee) {
        if (count < employees.length) {
            employees[count++] = employee;
            return true;
        } else {
            return false; // Array is full
        }
    }

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

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

    // Method to delete an employee by ID
    public boolean deleteEmployee(String employeeId) {
        for (int i = 0; i < count; i++) {
            if (employees[i].getEmployeeId().equals(employeeId)) {
                // Shift all elements to the left
                for (int j = i; j < count - 1; j++) {
                    employees[j] = employees[j + 1];
                }
                employees[--count] = null; // Decrease count and clear the last element
                return true;
            }
        }
        return false; // Employee not found
    }
}
```

4. Analysis

Time Complexity

Add Operation: O(1) - Adding an employee is constant time if there is space in the array.

Search Operation: O(n) - Searching for an employee requires a linear scan in the worst case.

Traverse Operation: O(n) - Traversing through all employees takes linear time.

Delete Operation: O(n) - Deleting an employee requires shifting elements, which takes linear time in the worst case.

Limitations of Arrays

Fixed Size: Arrays have a fixed size, which can be limiting if the number of employees grows beyond the array size.

Inefficient Deletion: Deleting an element from the middle of the array requires shifting elements, which is not efficient.

Static Allocation: Memory is allocated statically, leading to potential waste if the array is not fully used.

When to Use Arrays

Small, Fixed Number of Elements: Arrays are suitable when the number of elements is known and small.

Frequent Access by Index: If elements need to be accessed frequently by their index, arrays provide efficient O(1) access.

Test Case

Here's how you might set up a test case:

```
public class Main {
    public static void main(String[] args) {
        EmployeeManagementSystem ems = new EmployeeManagementSystem(10);

        // Add employees
        ems.addEmployee(new Employee("1", "Alice", "Manager", 70000));
        ems.addEmployee(new Employee("2", "Bob", "Developer", 60000));
        ems.addEmployee(new Employee("3", "Charlie", "Analyst", 50000));

        // Traverse employees
        System.out.println("All Employees:");
        ems.traverseEmployees();

        // Search for an employee
        System.out.println("\nSearch Employee with ID 2:");
        Employee emp = ems.searchEmployee("2");
        if (emp != null) {
            System.out.println("ID: " + emp.getEmployeeId() + ", Name: " + emp.getName() +
                    ", Position: " + emp.getPosition() + ", Salary: $" + emp.getSalary());
        } else {
            System.out.println("Employee not found");
        }

        // Delete an employee
        System.out.println("\nDelete Employee with ID 2:");
        if (ems.deleteEmployee("2")) {
            System.out.println("Employee deleted successfully");
        } else {
            System.out.println("Employee not found");
        }
        // Traverse employees after deletion
        System.out.println("\nAll Employees after deletion:");
        ems.traverseEmployees();
    }
}
```

This code sets up an array of employees, adds, searches, traverses, and deletes employees, demonstrating the functionality of the Employee Management System.

5.)

<u>Exercise 5: Task Management System:</u>

Types of Linked Lists

Singly Linked List: Each node contains data and a reference (or link) to the next node in the sequence. The last node points to null.

Doubly Linked List: Each node contains data, a reference to the next node, and a reference to the previous node. This allows traversal in both directions (forward and backward).

2. Setup

Create a class Task with attributes like taskId, taskName, and status.

```
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;
    }
}
```

3. Implementation

Singly Linked List

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

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

public class TaskManagementSystem {
    private Node head;

    // Add a task to the linked list
    public void addTask(Task task) {
        Node newNode = new Node(task);
        if (head == null) {
            head = newNode;
        } else {
            Node temp = head;
            while (temp.next != null) {
                temp = temp.next;
            }
            temp.next = newNode;
        }
    }

    // Search for a task by taskId
    public Task searchTask(String taskId) {
        Node temp = head;
        while (temp != null) {
            if (temp.task.getTaskId().equals(taskId)) {
                return temp.task;
            }
            temp = temp.next;
        }
        return null; // Task not found
```

```
}

// Traverse the linked list
public void traverseTasks() {
    Node temp = head;
    while (temp != null) {
        System.out.println("ID: " + temp.task.getTaskId() + ", Name: " + temp.task.getTaskName() +
                ", Status: " + temp.task.getStatus());
        temp = temp.next;
    }
}

// Delete a task by taskId
public boolean deleteTask(String taskId) {
    if (head == null) {
        return false; // List is empty
    }

    if (head.task.getTaskId().equals(taskId)) {
        head = head.next; // Delete the head node
        return true;
    }

    Node temp = head;
    while (temp.next != null) {
        if (temp.next.task.getTaskId().equals(taskId)) {
            temp.next = temp.next.next; // Delete the node
            return true;
        }
        temp = temp.next;
    }
```

```
        return false; // Task not found
    }
}
```

4. Analysis

Time Complexity

Add Operation: O(n) - Adding a task requires traversing to the end of the list in the worst case.

Search Operation: O(n) - Searching for a task requires traversing the list in the worst case.

Traverse Operation: O(n) - Traversing all tasks requires visiting each node.

Delete Operation: O(n) - Deleting a task requires traversing the list to find the node in the worst case.

Advantages of Linked Lists over Arrays

Dynamic Size: Linked lists can grow and shrink dynamically, unlike arrays which have a fixed size.

Efficient Insertions/Deletions: Inserting or deleting elements in a linked list is more efficient, as it only involves updating the links, whereas arrays require shifting elements.

Test Case

Here's how you might set up a test case:

```
public class Main {
    public static void main(String[] args) {
        TaskManagementSystem tms = new TaskManagementSystem();

        // Add tasks
        tms.addTask(new Task("1", "Task One", "Pending"));
        tms.addTask(new Task("2", "Task Two", "Completed"));
        tms.addTask(new Task("3", "Task Three", "In Progress"));

        // Traverse tasks
        System.out.println("All Tasks:");
        tms.traverseTasks();

        // Search for a task
        System.out.println("\nSearch Task with ID 2:");
        Task task = tms.searchTask("2");
        if (task != null) {
            System.out.println("ID: " + task.getTaskId() + ", Name: " + task.getTaskName() + ", Status: " +
task.getStatus());
        } else {
            System.out.println("Task not found");
        }

        // Delete a task
        System.out.println("\nDelete Task with ID 2:");
        if (tms.deleteTask("2")) {
            System.out.println("Task deleted successfully");
        } else {
            System.out.println("Task not found");
        }

        // Traverse tasks after deletion
        System.out.println("\nAll Tasks after deletion:");
        tms.traverseTasks();
    }
}
```

This code sets up a linked list of tasks, adds, searches, traverses, and deletes tasks, demonstrating the functionality of the Task Management System.

6). Library Management System

Linear Search

Description: A simple search algorithm that checks every element in the list sequentially until the desired element is found or the list ends.

Time Complexity:

Best Case: O(1) (element is the first in the list)

Average Case: O(n)

Worst Case: O(n)

Binary Search

Description: An efficient search algorithm that works on sorted lists. It repeatedly divides the search interval in half and compares the target value to the middle element.

Time Complexity:

Best Case: O(1) (element is the middle of the list)

Average Case: O(log n)

Worst Case: O(log n)

2. Setup

Create a class Book with attributes like bookId, title, and author.

```
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;
    }
}
```

3. Implementation

Linear Search

```
public class LibraryManagementSystem {
    private List<Book> books;

    // Constructor
    public LibraryManagementSystem() {
        books = new ArrayList<>();
    }

    // Method to add a book
    public void addBook(Book book) {
        books.add(book);
    }

    // Linear search to find books by title
    public List<Book> findBooksByTitleLinear(String title) {
        List<Book> result = new ArrayList<>();
        for (Book book : books) {
            if (book.getTitle().equalsIgnoreCase(title)) {
                result.add(book);
            }
        }
        return result;
    }

    // Linear search to find books by author
    public List<Book> findBooksByAuthorLinear(String author) {
        List<Book> result = new ArrayList<>();
        for (Book book : books) {
            if (book.getAuthor().equalsIgnoreCase(author)) {
                result.add(book);
            }
        }
        return result;
    }
}
```

Binary Search

```
import java.util.Collections;
import java.util.Comparator;

public class LibraryManagementSystem {

    // Binary search to find books by title (assuming list is sorted)
    public Book findBookByTitleBinary(String title) {
        int left = 0;
        int right = books.size() - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            Book midBook = books.get(mid);
            int cmp = midBook.getTitle().compareToIgnoreCase(title);

            if (cmp == 0) {
                return midBook;
            } else if (cmp < 0) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return null; // Book not found
    }

    // Method to sort books by title
    public void sortBooksByTitle() {
        Collections.sort(books, Comparator.comparing(Book::getTitle));
    }
}
```

4. Analysis

Time Complexity

Linear Search:

Best Case: O(1) (if the element is the first one)

Average Case: O(n)

Worst Case: O(n)

Binary Search:

Best Case: O(1) (if the element is the middle one)

Average Case: O(log n)

Worst Case: O(log n)

When to Use Each Algorithm

Linear Search:

Use when the list is unsorted or very small.

Suitable for unsorted lists where sorting is not practical.

Binary Search:

Use when the list is sorted.

Suitable for larger lists due to its O(log n) complexity.

Requires an initial sort of the list if it is not already sorted, which adds an overhead of O(n log n).

Test Case

Here's how you might set up a test case:

```
public class Main {
  public static void main(String[] args) {
    LibraryManagementSystem lms = new LibraryManagementSystem();

    // Add books
    lms.addBook(new Book("1", "The Great Gatsby", "F. Scott Fitzgerald"));
    lms.addBook(new Book("2", "To Kill a Mockingbird", "Harper Lee"));
    lms.addBook(new Book("3", "1984", "George Orwell"));
    lms.addBook(new Book("4", "Moby Dick", "Herman Melville"));

    // Linear search for books by title
    System.out.println("Linear Search by Title '1984':");
    List<Book> booksByTitle = lms.findBooksByTitleLinear("1984");
    for (Book book : booksByTitle) {
        System.out.println("ID: " + book.getBookId() + ", Title: " + book.getTitle() + ", Author: " +
book.getAuthor());
    }

    // Linear search for books by author
    System.out.println("\nLinear Search by Author 'Harper Lee':");
    List<Book> booksByAuthor = lms.findBooksByAuthorLinear("Harper Lee");
    for (Book book : booksByAuthor) {
        System.out.println("ID: " + book.getBookId() + ", Title: " + book.getTitle() + ", Author: " +
book.getAuthor());
    }

    // Binary search for books by title
    System.out.println("\nBinary Search by Title 'Moby Dick':");
    lms.sortBooksByTitle(); // Sort books before binary search
    Book book = lms.findBookByTitleBinary("Moby Dick");
    if (book != null) {
        System.out.println("ID: " + book.getBookId() + ", Title: " + book.getTitle() + ", Author: " +
book.getAuthor());
    } else {
        System.out.println("Book not found");
    }
  }
}
```

This code sets up a library management system, adds books, performs linear and binary searches, and demonstrates the functionality of the system.

7). **Financial Forecasting:**

Concept of Recursion

Recursion: A technique in which a function calls itself directly or indirectly to solve a problem. It simplifies problems by breaking them down into smaller, more manageable sub-problems of the same type.

Base Case: The condition under which the recursion stops. This prevents infinite recursion and provides a solution to the simplest sub-problem.

Recursive Case: The part of the function where the function calls itself with modified arguments, gradually approaching the base case.

2. Setup

Create a method to calculate the future value using a recursive approach. We'll assume that the future value is based on a fixed growth rate applied over a number of periods.

3. Implementation

Recursive Algorithm for Predicting Future Values

Assume we have:

currentValue: The current value.

growthRate: The rate of growth per period.

periods: The number of periods into the future we want to predict.

```
public class FinancialForecasting {

    // Recursive method to calculate future value
    public double calculateFutureValue(double currentValue, double growthRate, int periods) {
        // Base case: if periods is 0, the future value is the current value
        if (periods == 0) {
            return currentValue;
        }
        // Recursive case: calculate the future value for the next period
        return calculateFutureValue(currentValue * (1 + growthRate), growthRate, periods - 1);
    }

    public static void main(String[] args) {
        FinancialForecasting forecasting = new FinancialForecasting();
        double currentValue = 1000.0;
        double growthRate = 0.05; // 5% growth rate
        int periods = 10;
        double futureValue = forecasting.calculateFutureValue(currentValue, growthRate, periods);
        System.out.println("Future Value: " + futureValue);
    }
}
```

4. Analysis

Time Complexity

The time complexity of the recursive algorithm is O(n), where n is the number of periods. This is because the function makes a single recursive call for each period until the base case is reached.

Optimizing the Recursive Solution

Recursion can lead to excessive computation and stack overflow if the depth of recursion is too high. To optimize the solution, we can use an iterative approach or memoization.

Iterative Approach

```
public class FinancialForecasting {

    // Iterative method to calculate future value
    public double calculateFutureValueIterative(double currentValue, double growthRate, int periods)
{
        double futureValue = currentValue;
        for (int i = 0; i < periods; i++) {
            futureValue *= (1 + growthRate);
        }
        return futureValue;
    }

    public static void main(String[] args) {
        FinancialForecasting forecasting = new FinancialForecasting();
        double currentValue = 1000.0;
        double growthRate = 0.05; // 5% growth rate
        int periods = 10;
        double futureValue = forecasting.calculateFutureValueIterative(currentValue, growthRate, 
periods);
        System.out.println("Future Value: " + futureValue);
    }
}
public class FinancialForecasting {

    // Iterative method to calculate future value
    public double calculateFutureValueIterative(double currentValue, double growthRate, int periods) 
{
        double futureValue = currentValue;
        for (int i = 0; i < periods; i++) {
            futureValue *= (1 + growthRate);
        }
        return futureValue;
    }

    public static void main(String[] args) {
        FinancialForecasting forecasting = new FinancialForecasting();
        double currentValue = 1000.0;
        double growthRate = 0.05; // 5% growth rate
        int periods = 10;

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

Memoization Approach

Memoization is useful if the function is called multiple times with the same parameters. However, for a simple linear growth prediction, memoization is less relevant.

Test Case

Here's how you might set up a test case:

```
public class Main {
    public static void main(String[] args) {
        FinancialForecasting forecasting = new FinancialForecasting();

        // Test recursive method
        double currentValue = 1000.0;
        double growthRate = 0.05; // 5% growth rate
        int periods = 10;

        double futureValueRecursive = forecasting.calculateFutureValue(currentValue, growthRate,
periods);
        System.out.println("Future Value (Recursive): " + futureValueRecursive);

        // Test iterative method
        double futureValueIterative = forecasting.calculateFutureValueIterative(currentValue,
growthRate, periods);
        System.out.println("Future Value (Iterative): " + futureValueIterative);
    }
}
```

This code sets up the financial forecasting tool, calculates the future value using both recursive and iterative methods, and demonstrates the functionality of the system.