**Algorithms\_Data Structures Hands-on**

**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.

import java.util.*\**;

public class Product {

    int productId;

    String productName;

    String category;

    public Product(int id, String name, String category) {

*this*.productId = id;

*this*.productName = name;

*this*.category = category;

    }

    public String toString() {

        return productId + " - " + productName + " (" + category + ")";

    }

    public static Product linearSearch(Product[] products, String targetName) {

        for (int i = 0; i < products.length; i++) {

            if (products[i].productName.equalsIgnoreCase(targetName)) {

                return products[i];

            }

        }

        return null;

    }

    public static Product binarySearch(Product[] products, String targetName) {

        int low = 0, high = products.length - 1;

        while (low <= high) {

            int mid = (low + high) / 2;

            int result = products[mid].productName.compareToIgnoreCase(targetName);

            if (result == 0)

                return products[mid];

            else if (result < 0)

                low = mid + 1;

            else

                high = mid - 1;

        }

        return null;

    }

    public static void main(String[] args) {

        Product[] products = {

            new Product(101, "Laptop", "Electronics"),

            new Product(102, "Shoes", "Fashion"),

            new Product(103, "Mobile", "Electronics"),

            new Product(104, "Book", "Education")

        };

        Product result1 = linearSearch(products, "Shoes");

        System.out.println("Linear Search: " + (result1 != null ? result1 : "Not Found"));

        Arrays.sort(products, Comparator.comparing(p -> p.productName.toLowerCase()));

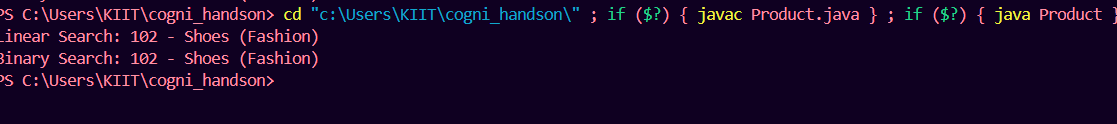
        Product result2 = binarySearch(products, "Shoes");

        System.out.println("Binary Search: " + (result2 != null ? result2 : "Not Found"));

    }

}

Output:



**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.

public class FinancialForecast {

    public static double forecastValue(double currentValue, double rate, int years) {

        if (years == 0) {

            return currentValue;

        }

        return forecastValue(currentValue \* (1 + rate), rate, years - 1);

    }

    public static void main(String[] args) {

        double initialValue = 10000;

        double growthRate = 0.05;

        int years = 5;

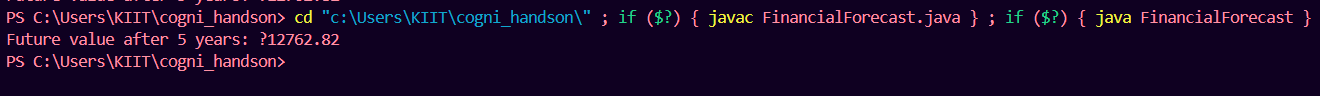
        double futureValue = forecastValue(initialValue, growthRate, years);

        System.out.printf("Future value after %d years: ₹%.2f\n", years, futureValue);

    }

}

OUTPUT:



**Design Principles and patterns hands-on**

**Exercise 1: Implementing the Singleton Pattern**

**Scenario:**

You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **SingletonPatternExample**.
2. **Define a Singleton Class:**
   * Create a class named Logger that has a private static instance of itself.
   * Ensure the constructor of Logger is private.
   * Provide a public static method to get the instance of the Logger class.
3. **Implement the Singleton Pattern:**
   * Write code to ensure that the Logger class follows the Singleton design pattern.
4. **Test the Singleton Implementation:**
   * Create a test class to verify that only one instance of Logger is created and used across the application.

public class SingletonPattern {

    static class Logger {

        private static Logger instance;

        private Logger() {

            System.out.println("Logger instance created");

        }

        public static Logger getInstance() {

            if (instance == null) {

                instance = new Logger();

            }

            return instance;

        }

        public void log(String message) {

            System.out.println("Log: " + message);

        }

    }

    public static void main(String[] args) {

        Logger logger1 = Logger.getInstance();

        logger1.log("First log message");

        Logger logger2 = Logger.getInstance();

        logger2.log("Second log message");

        if (logger1 == logger2) {

            System.out.println("Both logger1 and logger2 are the same instance.");

        } else {

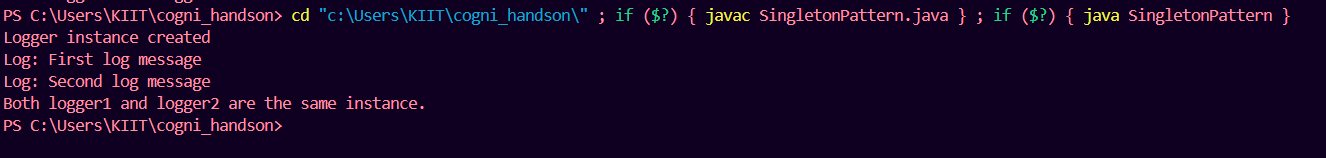
            System.out.println("Different instances (this should not happen in Singleton).");

        }

    }

}

Output:



**Exercise 2: Implementing the Factory Method Pattern**

**Scenario:**

You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **FactoryMethodPatternExample**.
2. **Define Document Classes:**
   * Create interfaces or abstract classes for different document types such as **WordDocument**, **PdfDocument**, and **ExcelDocument**.
3. **Create Concrete Document Classes:**
   * Implement concrete classes for each document type that implements or extends the above interfaces or abstract classes.
4. **Implement the Factory Method:**
   * Create an abstract class **DocumentFactory** with a method **createDocument()**.
   * Create concrete factory classes for each document type that extends DocumentFactory and implements the **createDocument()** method.
5. **Test the Factory Method Implementation:**
   * Create a test class to demonstrate the creation of different document types using the factory method.

public class FactoryMethodPattern {

    interface Document {

        void open();

    }

    static class WordDocument implements Document {

        public void open() {

            System.out.println("Opening Word document...");

        }

    }

    static class PdfDocument implements Document {

        public void open() {

            System.out.println("Opening PDF document...");

        }

    }

    static class ExcelDocument implements Document {

        public void open() {

            System.out.println("Opening Excel document...");

        }

    }

    abstract static class DocumentFactory {

        public abstract Document createDocument();

    }

    static class WordDocumentFactory extends DocumentFactory {

        public Document createDocument() {

            return new WordDocument();

        }

    }

    static class PdfDocumentFactory extends DocumentFactory {

        public Document createDocument() {

            return new PdfDocument();

        }

    }

    static class ExcelDocumentFactory extends DocumentFactory {

        public Document createDocument() {

            return new ExcelDocument();

        }

    }

    public static void main(String[] args) {

        DocumentFactory wordFactory = new WordDocumentFactory();

        Document wordDoc = wordFactory.createDocument();

        wordDoc.open();

        DocumentFactory pdfFactory = new PdfDocumentFactory();

        Document pdfDoc = pdfFactory.createDocument();

        pdfDoc.open();

        DocumentFactory excelFactory = new ExcelDocumentFactory();

        Document excelDoc = excelFactory.createDocument();

        excelDoc.open();

    }

}

Output:

