**Cognizant - Digital-Nurture-4.0-JavaFSE (Week 1)**

**SUPERSET ID:** **6377161**

**Design principles & Patterns:**

Exercise 1: Implementing the Singleton Pattern

Code:

public class Singleton {

static class Logger {

private static Logger instance = null;

private Logger() {

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

}

public static Logger getLogger() {

if (instance == null) {

synchronized (Logger.class) {

if (instance == null) {

instance = new Logger();

}

}

}

return instance;

}

public void logMessage(String message) {

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

}

}

public static void main(String[] args) {

Logger loggerOne = Logger.getLogger();

Logger loggerTwo = Logger.getLogger();

loggerOne.logMessage("First log entry.");

loggerTwo.logMessage("Second log entry.");

if (loggerOne == loggerTwo) {

System.out.println("Same logger instance used.");

} else {

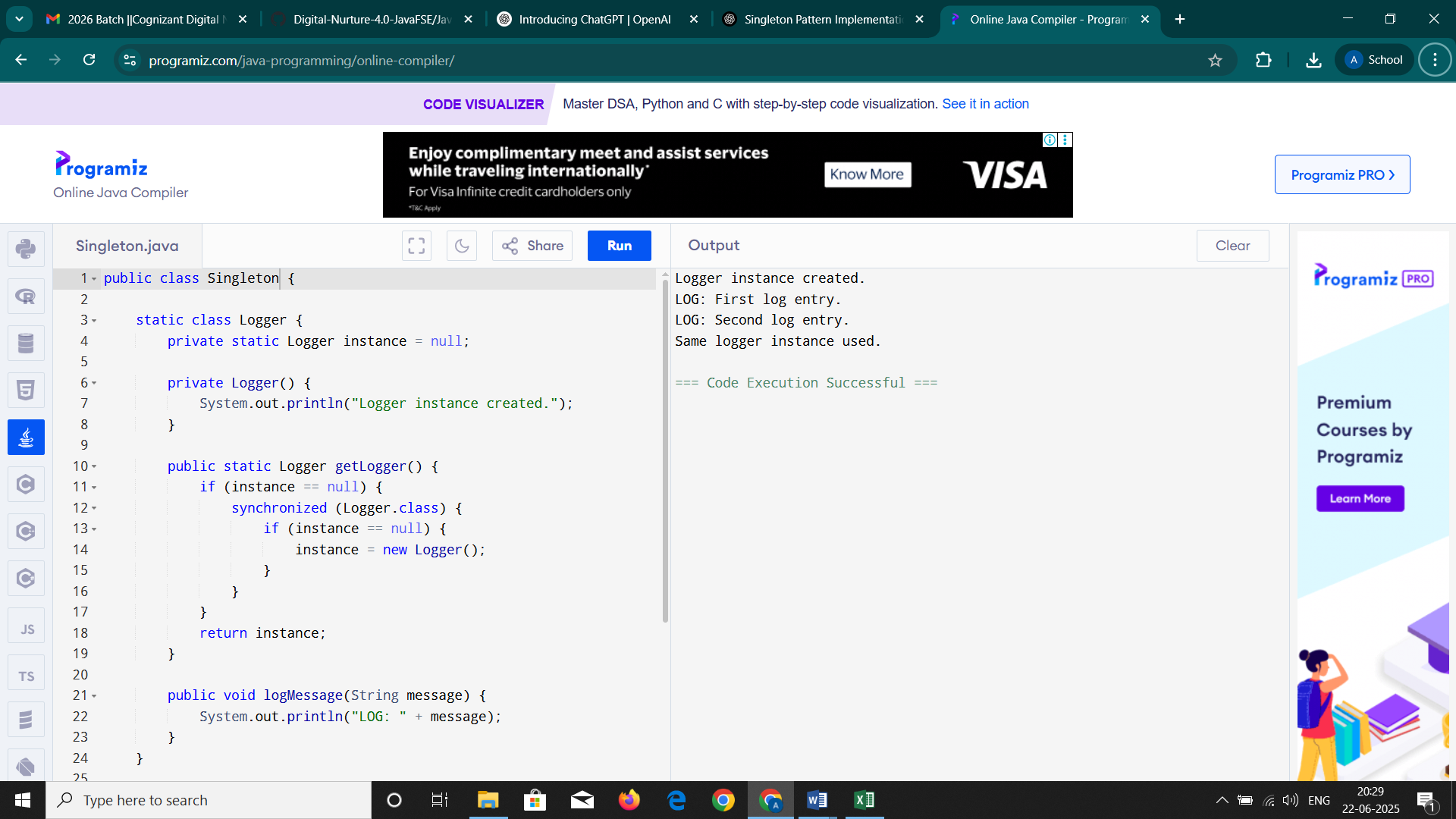
System.out.println("Different instances detected.");

}

}

}

Output:



Exercise 2: Implementing the Factory Method Pattern

Code:

public class FactoryMethodPattern {

interface Document {

void open();

}

static class WordFile implements Document {

public void open() {

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

}

}

static class PdfFile implements Document {

public void open() {

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

}

}

static class ExcelFile implements Document {

public void open() {

System.out.println("Opening an Excel spreadsheet.");

}

}

abstract static class DocumentCreator {

public abstract Document createDocument();

}

static class WordCreator extends DocumentCreator {

public Document createDocument() {

return new WordFile();

}

}

static class PdfCreator extends DocumentCreator {

public Document createDocument() {

return new PdfFile();

}

}

static class ExcelCreator extends DocumentCreator {

public Document createDocument() {

return new ExcelFile();

}

}

public static void main(String[] args) {

DocumentCreator wordFactory = new WordCreator();

DocumentCreator pdfFactory = new PdfCreator();

DocumentCreator excelFactory = new ExcelCreator();

Document doc1 = wordFactory.createDocument();

Document doc2 = pdfFactory.createDocument();

Document doc3 = excelFactory.createDocument();

doc1.open();

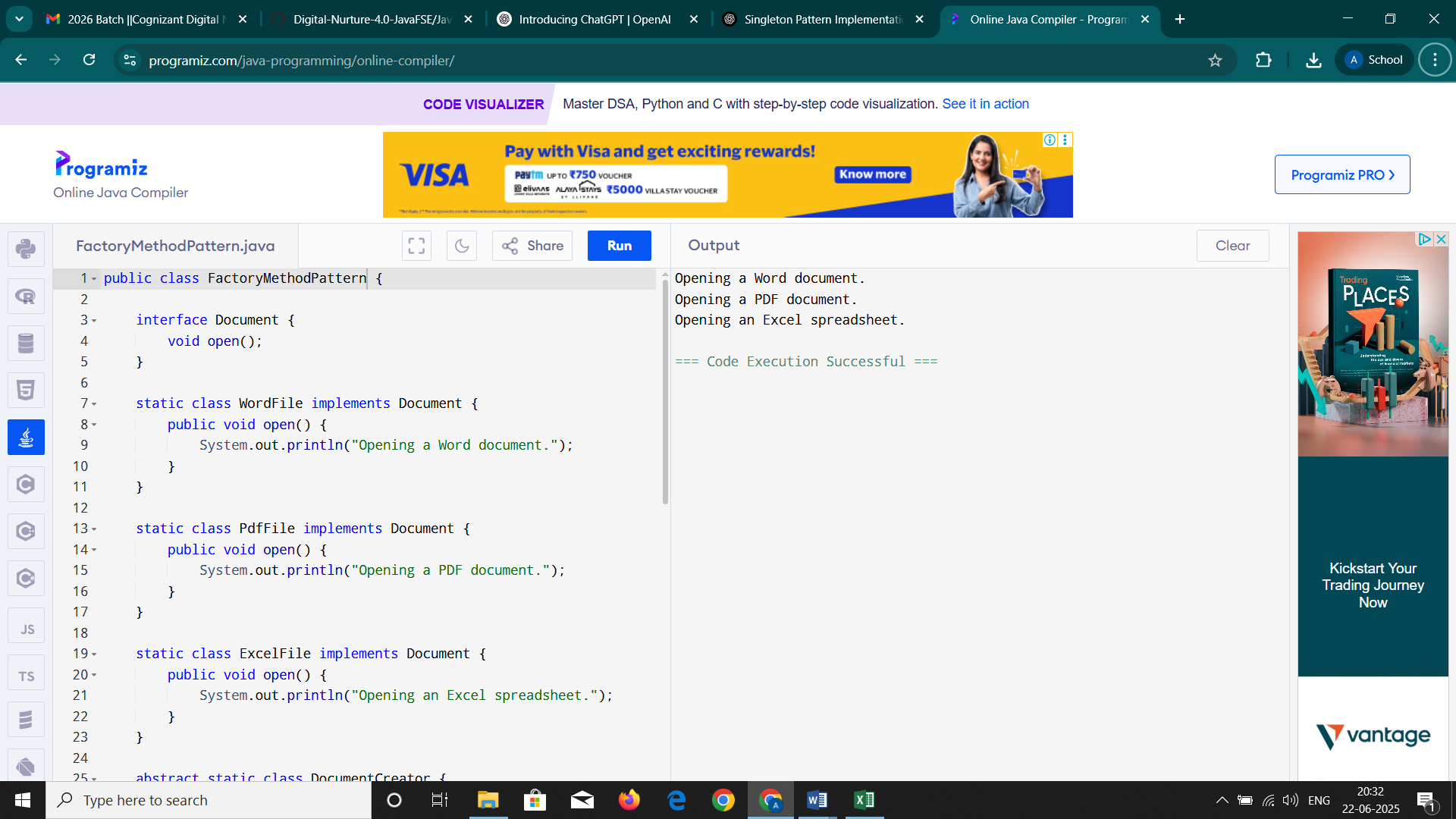
doc2.open();

doc3.open();

}

}

Output:



**Data structures and Algorithms:**

Exercise 2: E-commerce Platform Search Function

Code:

import java.util.Arrays;

public class EcommerceSearch {

static class Product {

int productId;

String productName;

String category;

Product(int id, String name, String cat) {

this.productId = id;

this.productName = name;

this.category = cat;

}

public String toString() {

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

}

}

public static Product linearSearch(Product[] items, String target) {

for (Product item : items) {

if (item.productName.equalsIgnoreCase(target)) {

return item;

}

}

return null;

}

public static Product binarySearch(Product[] items, String target) {

int start = 0, end = items.length - 1;

while (start <= end) {

int mid = (start + end) / 2;

int cmp = items[mid].productName.compareToIgnoreCase(target);

if (cmp == 0) {

return items[mid];

} else if (cmp < 0) {

start = mid + 1;

} else {

end = mid - 1;

}

}

return null;

}

public static void main(String[] args) {

System.out.println("=== Understanding Asymptotic Notation ===");

System.out.println("Big O describes the upper bound performance of an algorithm.");

System.out.println("Linear Search: Best = O(1), Avg/Worst = O(n)");

System.out.println("Binary Search: Best = O(1), Avg/Worst = O(log n)");

System.out.println();

Product[] inventory = {

new Product(201, "Shoes", "Fashion"),

new Product(105, "Laptop", "Electronics"),

new Product(312, "Headphones", "Electronics"),

new Product(144, "Notebook", "Stationery"),

new Product(222, "Watch", "Accessories")

};

System.out.println("=== Linear Search ===");

Product result1 = linearSearch(inventory, "Laptop");

if (result1 != null)

System.out.println("Product Found: " + result1);

else

System.out.println("Product not found.");

System.out.println();

Arrays.sort(inventory, (a, b) -> a.productName.compareToIgnoreCase(b.productName));

System.out.println("=== Binary Search ===");

Product result2 = binarySearch(inventory, "Laptop");

if (result2 != null)

System.out.println("Product Found: " + result2);

else

System.out.println("Product not found.");

System.out.println();

System.out.println("=== Analysis ===");

System.out.println("Linear search does not require sorted data but is slower for large lists.");

System.out.println("Binary search is much faster but requires sorted input.");

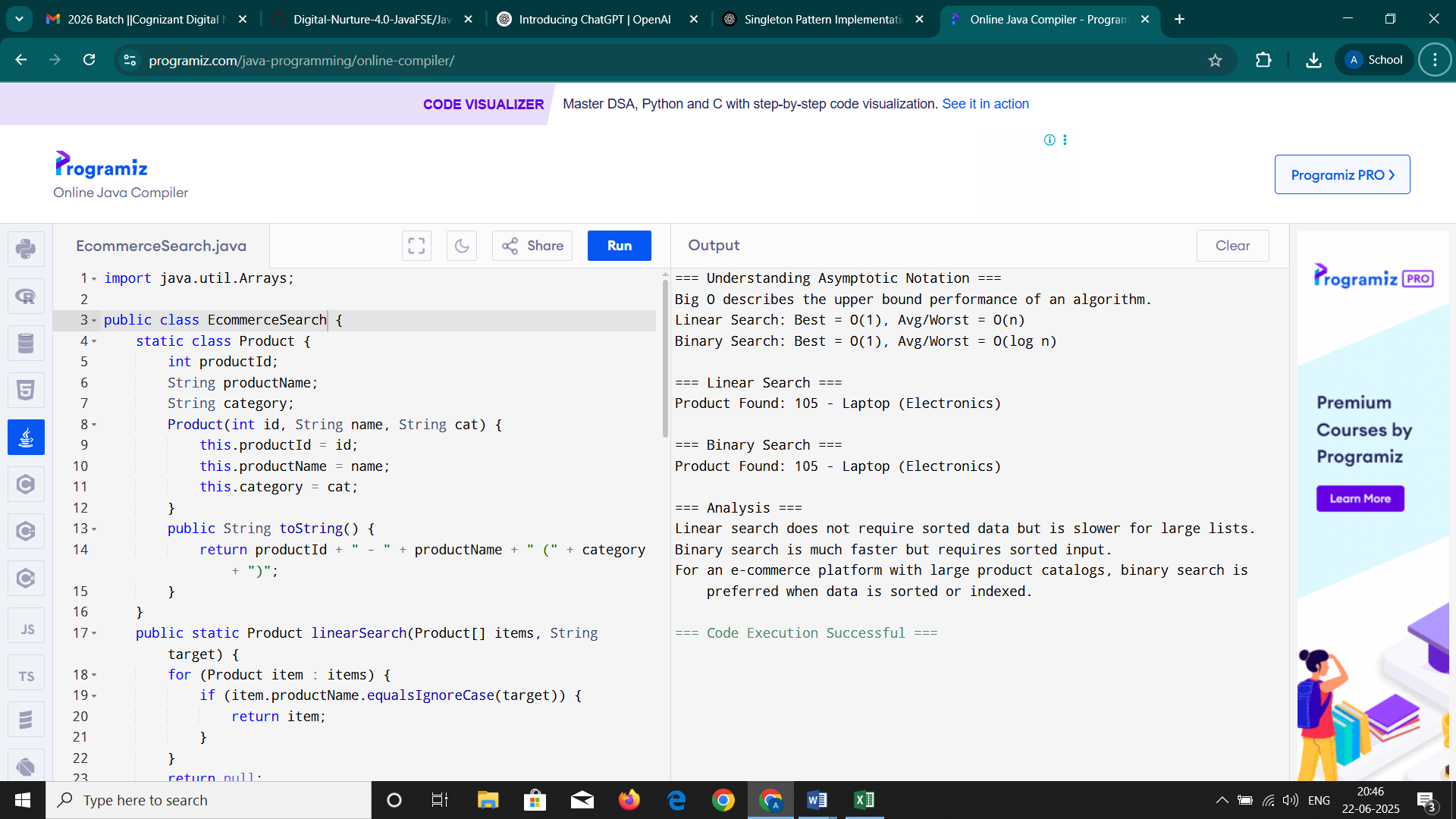
System.out.println("For an e-commerce platform with large product catalogs, binary search is

preferred when data is sorted or indexed.");

}

}

Output:



Exercise 7: Financial Forecasting

Code:

public class FinancialForecast {

public static double predictFutureValue(double currentValue, double growthRate, int years) {

if (years == 0) {

return currentValue;

}

return predictFutureValue(currentValue \* (1 + growthRate), growthRate, years - 1);

}

public static void main(String[] args) {

System.out.println("=== Understanding Recursion ===");

System.out.println("Recursion is when a method calls itself to solve smaller versions of a problem.");

System.out.println("It simplifies code for problems with repetitive structure.\n");

double startingValue = 10000.0;

double annualGrowthRate = 0.07;

int forecastYears = 5;

System.out.println("Starting Value: $" + startingValue);

System.out.println("Annual Growth Rate: " + (annualGrowthRate \* 100) + "%");

System.out.println("Forecast Period: " + forecastYears + " years\n");

double futureAmount = predictFutureValue(startingValue, annualGrowthRate, forecastYears);

System.out.printf("Predicted Future Value after %d years: $%.2f%n\n", forecastYears, futureAmount);

System.out.println("=== Time Complexity Analysis ===");

System.out.println("This recursive method has time complexity O(n), where n is the number of years.");

System.out.println("Each recursive call reduces the years by 1, and work done per call is constant.\n");

System.out.println("=== Optimization ===");

System.out.println("Since the recursion depth is equal to the number of years, it's safe for small n.");

System.out.println("For large n, consider using an iterative version or memoization to avoid stack overflow.");

}

}

Output:

