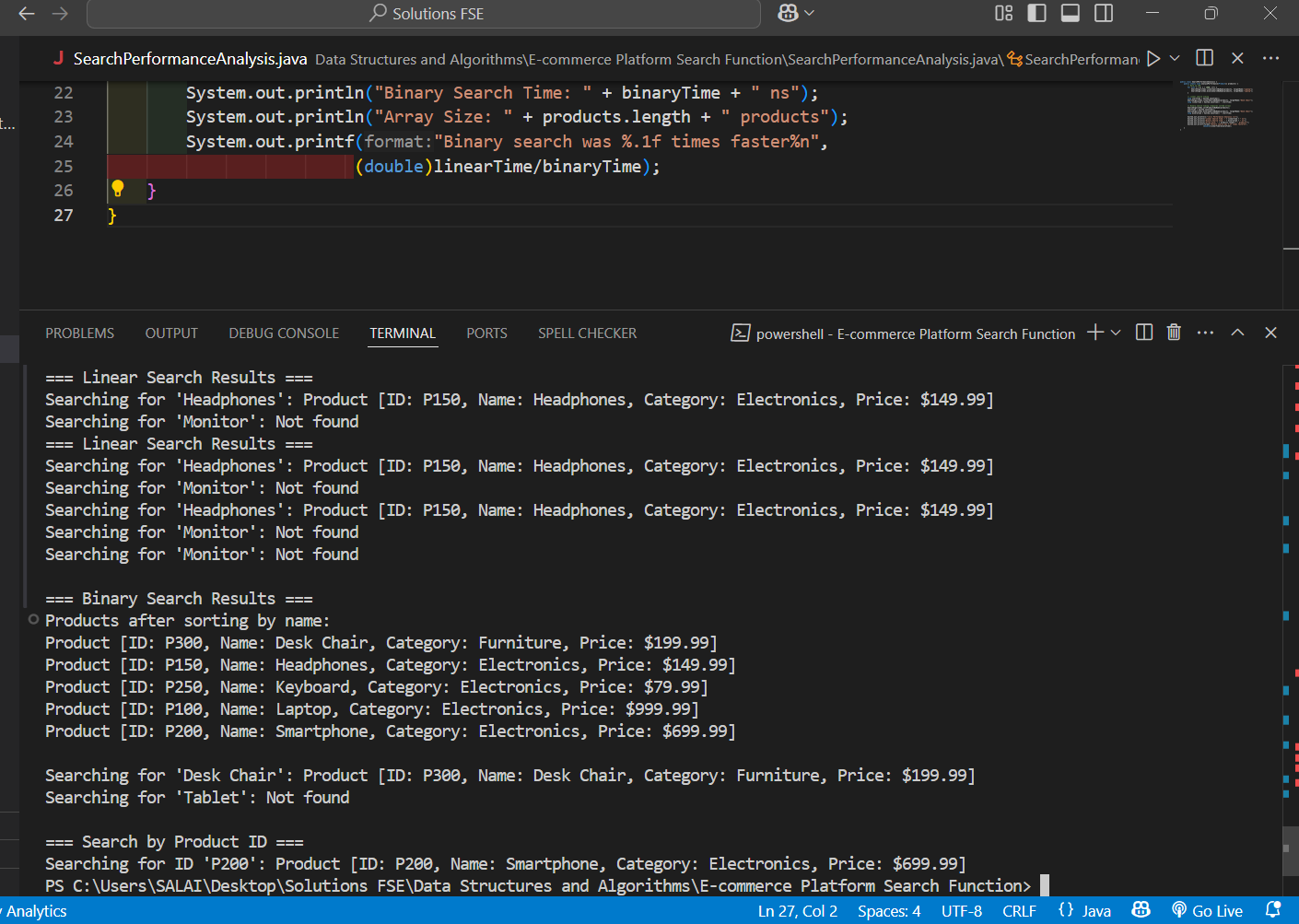
Data Structures and Algorithms

Exercise 2

Output



Code

public class Product {

private String productId;

private String productName;

private String category;

private double price;

public Product(String productId, String productName, String category, double price) {

this.productId = productId;

this.productName = productName;

this.category = category;

this.price = price;

}

// Getters

public String getProductId() { return productId; }

public String getProductName() { return productName; }

public String getCategory() { return category; }

public double getPrice() { return price; }

@Override

public String toString() {

return String.format("Product [ID: %s, Name: %s, Category: %s, Price: $%.2f]",

productId, productName, category, price);

}

}

import java.util.Arrays;

import java.util.Comparator;

public class SearchAlgorithms {

// Linear Search - O(n)

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

for (Product product : products) {

if (product.getProductName().equalsIgnoreCase(targetName)) {

return product;

}

}

return null;

}

// Binary Search - O(log n) - requires sorted array

public static Product binarySearchByName(Product[] sortedProducts, String targetName) {

int left = 0;

int right = sortedProducts.length - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

int comparison = sortedProducts[mid].getProductName()

.compareToIgnoreCase(targetName);

if (comparison == 0) {

return sortedProducts[mid];

} else if (comparison < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

// Helper method to sort products by name

public static void sortProductsByName(Product[] products) {

Arrays.sort(products, Comparator.comparing(Product::getProductName));

}

// Additional search by ID using linear search

public static Product linearSearchById(Product[] products, String targetId) {

for (Product product : products) {

if (product.getProductId().equalsIgnoreCase(targetId)) {

return product;

}

}

return null;

}

}

public class ECommercePlatform {

public static void main(String[] args) {

// Initialize product catalog

Product[] products = {

new Product("P100", "Laptop", "Electronics", 999.99),

new Product("P200", "Smartphone", "Electronics", 699.99),

new Product("P150", "Headphones", "Electronics", 149.99),

new Product("P300", "Desk Chair", "Furniture", 199.99),

new Product("P250", "Keyboard", "Electronics", 79.99)

};

System.out.println("=== E-Commerce Platform Search Demo ===");

System.out.println("Available Products:");

for (Product p : products) {

System.out.println(p);

}

System.out.println();

// Linear Search Demo

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

String searchTerm = "Headphones";

Product result = SearchAlgorithms.linearSearchByName(products, searchTerm);

System.out.println("Searching for '" + searchTerm + "': " +

(result != null ? result : "Not found"));

searchTerm = "Monitor";

result = SearchAlgorithms.linearSearchByName(products, searchTerm);

System.out.println("Searching for '" + searchTerm + "': " +

(result != null ? result : "Not found"));

// Binary Search Demo (requires sorting first)

System.out.println("\n=== Binary Search Results ===");

SearchAlgorithms.sortProductsByName(products);

System.out.println("Products after sorting by name:");

for (Product p : products) {

System.out.println(p);

}

System.out.println();

searchTerm = "Desk Chair";

result = SearchAlgorithms.binarySearchByName(products, searchTerm);

System.out.println("Searching for '" + searchTerm + "': " +

(result != null ? result : "Not found"));

searchTerm = "Tablet";

result = SearchAlgorithms.binarySearchByName(products, searchTerm);

System.out.println("Searching for '" + searchTerm + "': " +

(result != null ? result : "Not found"));

// Search by ID

System.out.println("\n=== Search by Product ID ===");

String searchId = "P200";

result = SearchAlgorithms.linearSearchById(products, searchId);

System.out.println("Searching for ID '" + searchId + "': " +

(result != null ? result : "Not found"));

}

}

public class SearchPerformanceAnalysis {

public static void analyzePerformance(Product[] products) {

// Warm up JVM

for (int i = 0; i < 1000; i++) {

SearchAlgorithms.linearSearchByName(products, "Laptop");

SearchAlgorithms.binarySearchByName(products, "Laptop");

}

// Linear search timing

long startTime = System.nanoTime();

SearchAlgorithms.linearSearchByName(products, "Desk Chair");

long linearTime = System.nanoTime() - startTime;

// Binary search timing (requires sorted array)

SearchAlgorithms.sortProductsByName(products);

startTime = System.nanoTime();

SearchAlgorithms.binarySearchByName(products, "Desk Chair");

long binaryTime = System.nanoTime() - startTime;

System.out.println("\n=== Performance Analysis ===");

System.out.println("Linear Search Time: " + linearTime + " ns");

System.out.println("Binary Search Time: " + binaryTime + " ns");

System.out.println("Array Size: " + products.length + " products");

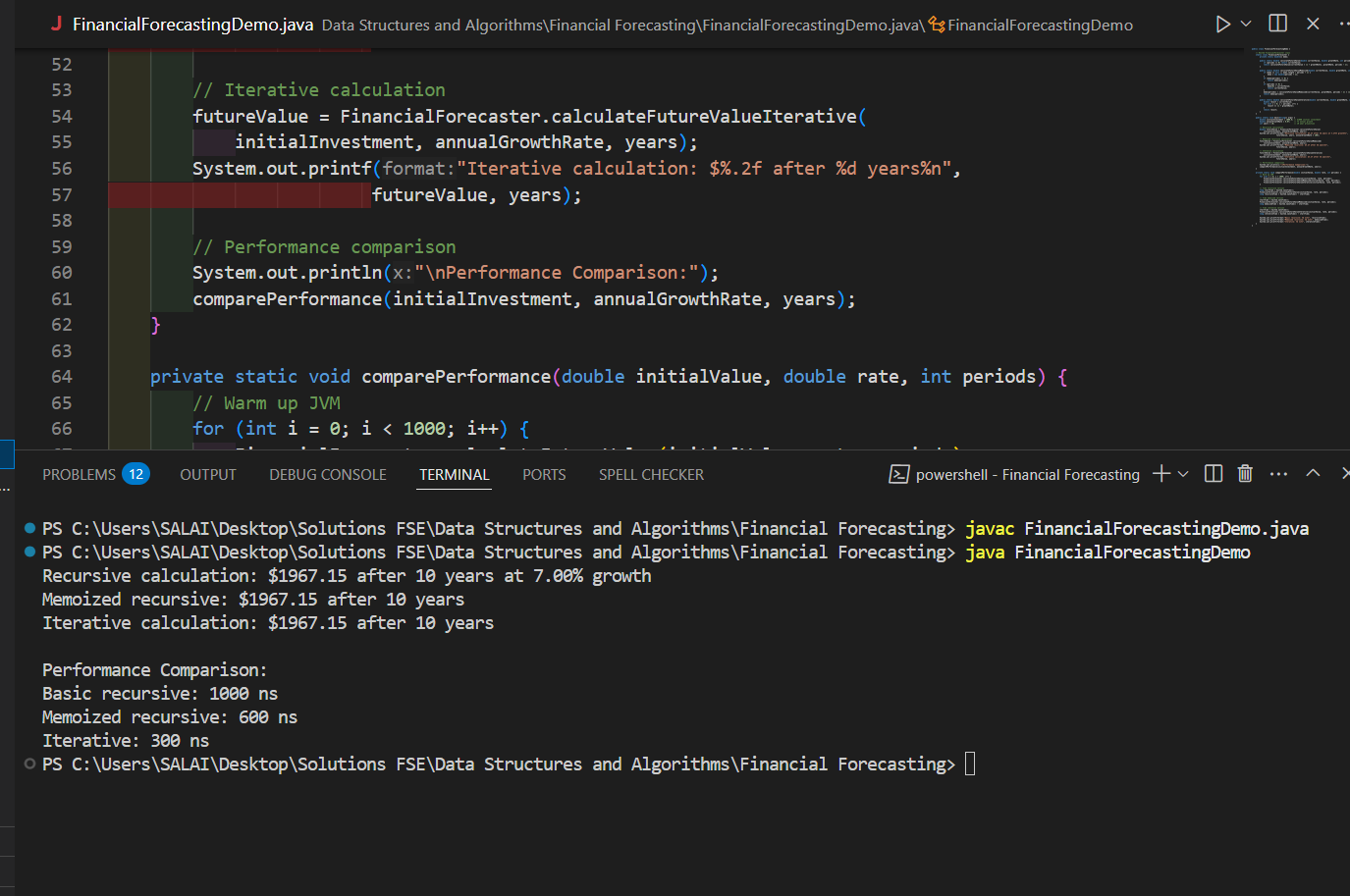
System.out.printf("Binary search was %.1f times faster%n",

(double)linearTime/binaryTime);

}

}

Exercise 7  
Output



public class FinancialForecastingDemo {

    // Nested FinancialForecaster class

    static class FinancialForecaster {

        private static double[] memo;

        public static double calculateFutureValue(double currentValue, double growthRate, int periods) {

            if (periods <= 0) return currentValue;

            return calculateFutureValue(currentValue \* (1 + growthRate), growthRate, periods - 1);

        }

        public static double calculateFutureValueMemoized(double currentValue, double growthRate, int periods) {

            if (memo == null || memo.length < periods + 1) {

                memo = new double[periods + 1];

            }

            if (memo[periods] != 0) {

                return memo[periods];

            }

            if (periods == 0) {

                memo[0] = currentValue;

                return currentValue;

            }

            memo[periods] = calculateFutureValueMemoized(currentValue, growthRate, periods - 1) \* (1 + growthRate);

            return memo[periods];

        }

        public static double calculateFutureValueIterative(double currentValue, double growthRate, int periods) {

            double result = currentValue;

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

                result \*= (1 + growthRate);

            }

            return result;

        }

    }

    public static void main(String[] args) {

        double initialInvestment = 1000.0; // $1000 initial investment

        double annualGrowthRate = 0.07;    // 7% annual growth

        int years = 10;                    // 10 year projection

        // Recursive calculation

        double futureValue = FinancialForecaster.calculateFutureValue(

            initialInvestment, annualGrowthRate, years);

        System.out.printf("Recursive calculation: $%.2f after %d years at %.2f%% growth%n",

                         futureValue, years, annualGrowthRate \* 100);

        // Memoized recursive calculation

        futureValue = FinancialForecaster.calculateFutureValueMemoized(

            initialInvestment, annualGrowthRate, years);

        System.out.printf("Memoized recursive: $%.2f after %d years%n",

                         futureValue, years);

        // Iterative calculation

        futureValue = FinancialForecaster.calculateFutureValueIterative(

            initialInvestment, annualGrowthRate, years);

        System.out.printf("Iterative calculation: $%.2f after %d years%n",

                         futureValue, years);

        // Performance comparison

        System.out.println("\nPerformance Comparison:");

        comparePerformance(initialInvestment, annualGrowthRate, years);

    }

    private static void comparePerformance(double initialValue, double rate, int periods) {

        // Warm up JVM

        for (int i = 0; i < 1000; i++) {

            FinancialForecaster.calculateFutureValue(initialValue, rate, periods);

            FinancialForecaster.calculateFutureValueMemoized(initialValue, rate, periods);

            FinancialForecaster.calculateFutureValueIterative(initialValue, rate, periods);

        }

        // Time recursive version

        long startTime = System.nanoTime();

        FinancialForecaster.calculateFutureValue(initialValue, rate, periods);

        long recursiveTime = System.nanoTime() - startTime;

        // Time memoized version

        startTime = System.nanoTime();

        FinancialForecaster.calculateFutureValueMemoized(initialValue, rate, periods);

        long memoizedTime = System.nanoTime() - startTime;

        // Time iterative version

        startTime = System.nanoTime();

        FinancialForecaster.calculateFutureValueIterative(initialValue, rate, periods);

        long iterativeTime = System.nanoTime() - startTime;

        System.out.printf("Basic recursive: %d ns%n", recursiveTime);

        System.out.printf("Memoized recursive: %d ns%n", memoizedTime);

        System.out.printf("Iterative: %d ns%n", iterativeTime);

    }

}