**ALGORITHMS DATA STRUCTURES**

**Exercise : 2 - E-commerce Platform Search Function**

**Code1: product.java**

public class product implements Comparable<product> {

private int productId;

private String productName;

private String category;

private double price;

public product(int productId, String productName, String category, double price) {

this.productId = productId;

this.productName = productName;

this.category = category;

this.price = price;

}

public int getProductId() {

return productId;

}

public String getProductName() {

return productName;

}

public String getCategory() {

return category;

}

public double getPrice() {

return price;

}

@Override

public int compareTo(product other) {

return Integer.compare(this.productId, other.productId);

}

@Override

public String toString() {

return String.format("ID: %d | %s | %s | $%.2f",

productId, productName, category, price);

}

}

**Code2: SearchAlgo.java**

public class SearchAlgo {

// Linear Search - O(n) time complexity

public static product linearSearch(product[] products, int targetId) {

for (product product : products) {

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

return product;

}

}

return null;

}

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

public static product binarySearch(product[] sortedProducts, int targetId) {

int left = 0;

int right = sortedProducts.length - 1;

while (left <= right) {

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

int midId = sortedProducts[mid].getProductId();

if (midId == targetId) {

return sortedProducts[mid];

} else if (midId < targetId) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

// Helper method to generate test products

public static product[] generateProducts(int count) {

product[] products = new product[count];

String[] categories = { "Electronics", "Clothing", "Home", "Books" };

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

int id = 1000 + i;

String name = "Product-" + (char) (65 + (i % 26)) + i;

String category = categories[i % categories.length];

double price = 10.99 + (i \* 5);

products[i] = new product(id, name, category, price);

}

return products;

}

}

**Code3: SearchBenchmark.java**

import java.util.Arrays;

import java.util.Random;

public class SearchBenchmark {

public static void main(String[] args) {

// Generate test data

final int NUM\_PRODUCTS = 10000;

product[] products = SearchAlgo.generateProducts(NUM\_PRODUCTS);

product[] sortedProducts = Arrays.copyOf(products, products.length);

Arrays.sort(sortedProducts);

// Select random product to search for

Random rand = new Random();

int targetId = products[rand.nextInt(NUM\_PRODUCTS)].getProductId();

System.out.println("Searching for product ID: " + targetId);

// Benchmark Linear Search

long startTime = System.nanoTime();

product linearResult = SearchAlgo.linearSearch(products, targetId);

long linearDuration = System.nanoTime() - startTime;

// Benchmark Binary Search

startTime = System.nanoTime();

product binaryResult = SearchAlgo.binarySearch(sortedProducts, targetId);

long binaryDuration = System.nanoTime() - startTime;

// Display results

System.out.println("\nLinear\_Search Result:");

System.out.println(linearResult != null ? linearResult : "Not\_found");

System.out.printf("Time: %,d ns\n", linearDuration);

System.out.println("\nBinary\_Search Result:");

System.out.println(binaryResult != null ? binaryResult : "Not\_found");

System.out.printf("Time: %,d ns\n", binaryDuration);

System.out.println("\nAlgorithm Analysis:");

System.out.println("Linear\_Search: O(n) time complexity");

System.out.println("Binary\_Search: O(log n) time complexity");

System.out.println("\nRecommendation: " + getRecommendation(NUM\_PRODUCTS));

}

private static String getRecommendation(int numProducts) {

if (numProducts < 100) {

return "For small catalogs (<100 items), linear search is sufficient";

} else if (numProducts < 10000) {

return "For medium catalogs (100-10,000 items), binary search is better";

} else {

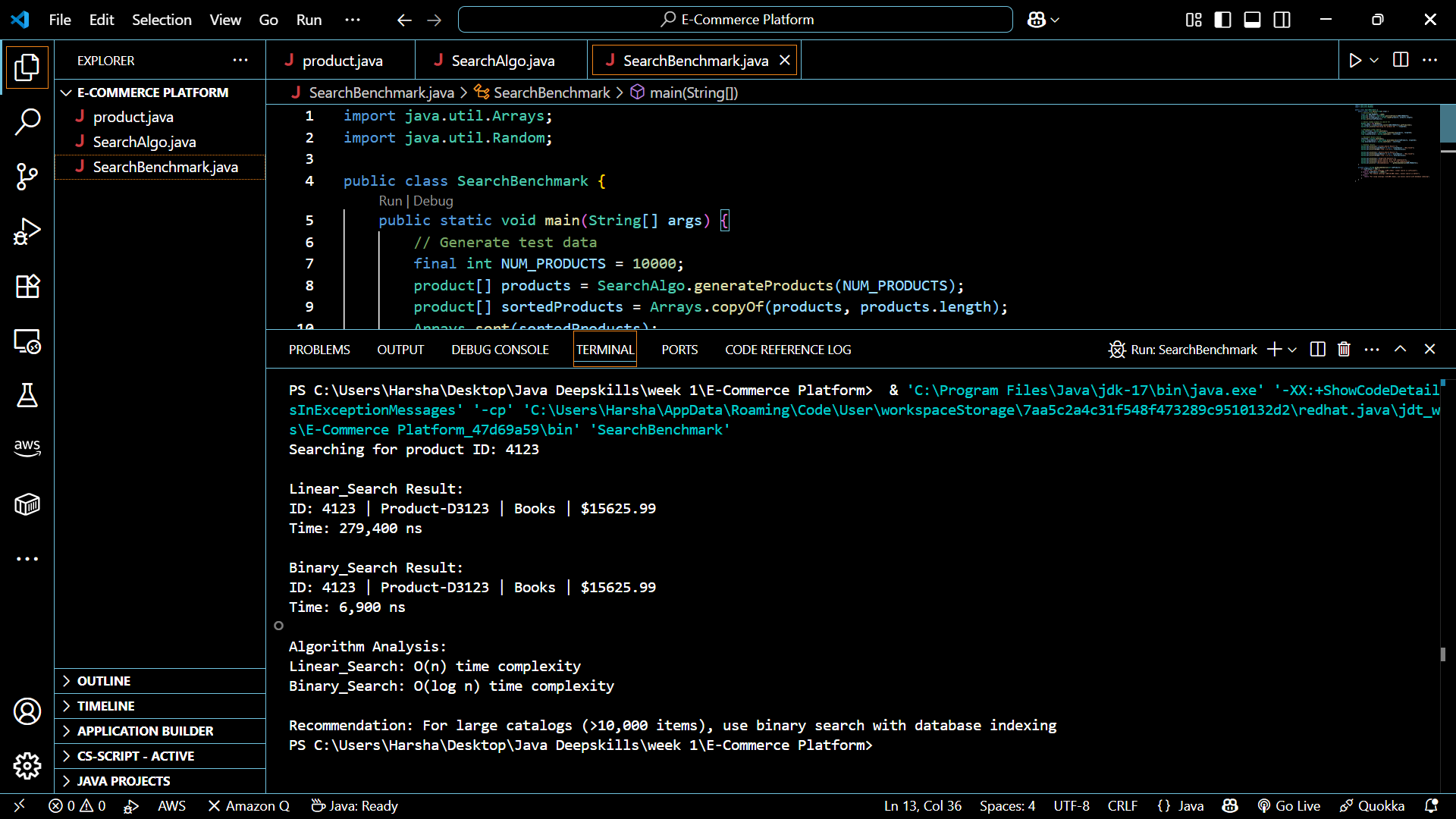
return "For large catalogs (>10,000 items), use binary search with database indexing";

}

}

}

**Output:**



**Exercise : 7 - Financial Forecasting**

**Code1: FinancialForecaster.java**

import java.util.Arrays;

import java.util.HashMap;

import java.util.Map;

public class FinancialForecaster {

private static Map<String, Double> memoizationCache = new HashMap<>();

public static double predictFutureValue(double[] historicalValues, int periodsForward) {

if (periodsForward <= 0) {

return historicalValues[historicalValues.length - 1];

}

double lastValue = historicalValues[historicalValues.length - 1];

double growthRate = calculateAverageGrowthRate(historicalValues);

double[] newHistory = Arrays.copyOf(historicalValues, historicalValues.length + 1);

newHistory[newHistory.length - 1] = lastValue \* (1 + growthRate);

return predictFutureValue(newHistory, periodsForward - 1);

}

public static double predictFutureValueOptimized(double[] historicalValues, int periodsForward) {

String cacheKey = Arrays.toString(historicalValues) + periodsForward;

if (memoizationCache.containsKey(cacheKey)) {

return memoizationCache.get(cacheKey);

}

if (periodsForward <= 0) {

return historicalValues[historicalValues.length - 1];

}

double lastValue = historicalValues[historicalValues.length - 1];

double growthRate = calculateAverageGrowthRate(historicalValues);

double[] newHistory = Arrays.copyOf(historicalValues, historicalValues.length + 1);

newHistory[newHistory.length - 1] = lastValue \* (1 + growthRate);

double result = predictFutureValueOptimized(newHistory, periodsForward - 1);

memoizationCache.put(cacheKey, result);

return result;

}

private static double calculateAverageGrowthRate(double[] values) {

if (values.length < 2) return 0.05;

double totalGrowth = 0.0;

for (int i = 1; i < values.length; i++) {

totalGrowth += (values[i] - values[i-1]) / values[i-1];

}

return totalGrowth / (values.length - 1);

}

public static String formatCurrency(double value) {

return String.format("$%,.2f", value);

}

}

**Code2: ForecastTester.java**

public class ForecastTester {

public static void main(String[] args) {

// Sample historical data (quarterly revenue in millions)

double[] revenueHistory = { 12.5, 13.2, 14.1, 15.0, 16.3, 17.8 };

System.out.println("Financial Forecasting Tool");

System.out.println("Historical Data: ");

for (double value : revenueHistory) {

System.out.print(FinancialForecaster.formatCurrency(value) + " ");

}

int forecastPeriods = 4; // Predict 4 quarters ahead

System.out.println("\n\nForecasting " + forecastPeriods + " periods forward...");

// Basic recursive forecast

long startTime = System.nanoTime();

double forecast = FinancialForecaster.predictFutureValue(revenueHistory, forecastPeriods);

long basicTime = System.nanoTime() - startTime;

// Optimized recursive forecast

startTime = System.nanoTime();

double optimizedForecast = FinancialForecaster.predictFutureValueOptimized(revenueHistory, forecastPeriods);

long optimizedTime = System.nanoTime() - startTime;

System.out.println("\nBasic Recursive Forecast:");

System.out.println("Predicted value: " + FinancialForecaster.formatCurrency(forecast));

System.out.println("Execution time: " + basicTime + " ns");

System.out.println("\nOptimized Recursive Forecast (with memoization):");

System.out.println("Predicted value: " + FinancialForecaster.formatCurrency(optimizedForecast));

System.out.println("Execution time: " + optimizedTime + " ns");

System.out.println("\nAlgorithm Analysis:");

System.out.println("Basic recursive: O(2^n) time complexity (without optimization)");

System.out.println("Optimized recursive: O(n) time complexity with memoization");

System.out.println("\nKey Insight: Memoization stores intermediate results to avoid redundant calculations");

}

}

**Output:**

