**WEEK-1**

**ALGORITHMS\_DATA STRUCTURES**

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

**File name- Product.java**

package com.ecommerce;

public class Product{

private int id;

private String name;

private String category;

private double price;

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

this.id = id;

this.name = name;

this.category = category;

this.price = price;

}

public int getId() { return id; }

public String getName() { return name; }

public String getCategory() { return category; }

public double getPrice() { return price; }

@Override

public String toString() {

return "Product{id=" + id + ", name='" + name + "', category='" + category + "', price=" + price + "}";

}

}

**File name- EcommerceSearch.java**

package com.ecommerce;

import java.util.\*;

public class EcommerceSearch {

private ArrayList<Product> products;

private ArrayList<Product> sortedProducts;

public EcommerceSearch() {

products = new ArrayList<>();

sortedProducts = new ArrayList<>();

setupProducts();

}

private void setupProducts() {

products.add(new Product(5, "iPhone 15", "Electronics", 999.99));

products.add(new Product(2, "Nike Air Max", "Shoes", 120.00));

products.add(new Product(8, "Coffee Machine", "Appliances", 299.99));

products.add(new Product(1, "MacBook Pro", "Electronics", 1299.99));

products.add(new Product(9, "Yoga Mat", "Sports", 25.99));

products.add(new Product(3, "Headphones", "Electronics", 199.99));

products.add(new Product(7, "Blender", "Appliances", 89.99));

products.add(new Product(4, "Jeans", "Clothing", 79.99));

products.add(new Product(6, "Book", "Books", 12.99));

products.add(new Product(10, "Water Bottle", "Sports", 19.99));

sortedProducts = new ArrayList<>(products);

sortedProducts.sort((p1, p2) -> Integer.compare(p1.getId(), p2.getId()));

System.out.println("Loaded " + products.size() + " products");

}

public Product linearSearch(int productId) {

int count = 0;

for(Product p : products) {

count++;

if(p.getId() == productId) {

System.out.println("Linear search found product in " + count + " steps");

return p;

}

}

System.out.println("Linear search: product not found after " + count + " steps");

return null;

}

public Product binarySearch(int productId) {

int left = 0;

int right = sortedProducts.size() - 1;

int steps = 0;

while(left <= right) {

steps++;

int middle = (left + right) / 2;

Product currentProduct = sortedProducts.get(middle);

if(currentProduct.getId() == productId) {

System.out.println("Binary search found product in " + steps + " steps");

return currentProduct;

}

if(currentProduct.getId() < productId) {

left = middle + 1;

} else {

right = middle - 1;

}

}

System.out.println("Binary search: product not found after " + steps + " steps");

return null;

}

public ArrayList<Product> searchByName(String searchTerm) {

ArrayList<Product> results = new ArrayList<>();

for(Product p : products) {

if(p.getName().toLowerCase().contains(searchTerm.toLowerCase())) {

results.add(p);

}

}

return results;

}

public void showAllProducts() {

System.out.println("\nAll products (unsorted):");

for(Product p : products) {

System.out.println(p);

}

System.out.println("\nAll products (sorted by ID):");

for(Product p : sortedProducts) {

System.out.println(p);

}

}

public static void main(String[] args) {

System.out.println("Linear Search: O(n) - checks each item");

System.out.println("Binary Search: O(log n) - eliminates half each time");

System.out.println();

EcommerceSearch searchEngine = new EcommerceSearch();

System.out.println("Use Binary Search for ID lookups (much faster)");

System.out.println("Use Linear Search for name searches (can't sort names easily)");

System.out.println("\nDemo - searching for 'book':");

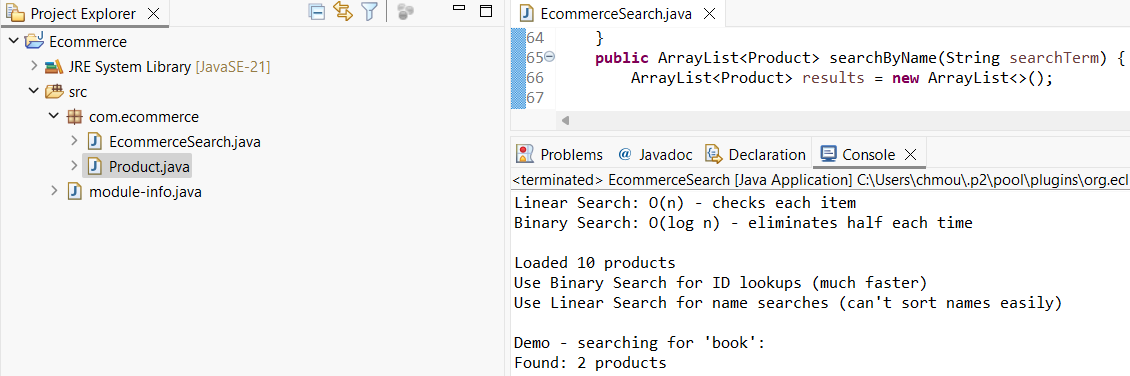
ArrayList<Product> results = searchEngine.searchByName("book");

System.out.println("Found: " + results.size() + " products");

}

}

**Output:**

****

**Exercise 7: Financial Forecasting**

**File name- FinancialForecasting.java**

package com.dependancy;

import java.util.\*;

public class FinancialForecasting {

private List<Double> historicalValues;

private Map<String, Double> memoCache;

private int recursiveCallCount = 0;

public FinancialForecasting() {

this.historicalValues = new ArrayList<>();

this.memoCache = new HashMap<>();

setupSampleData();

}

private void setupSampleData() {

historicalValues.add(100.0);

historicalValues.add(108.0);

historicalValues.add(116.6);

historicalValues.add(125.9);

historicalValues.add(136.0);

System.out.println("Portfolio data loaded: " + historicalValues.size() + " years");

}

private double calculateAverageGrowthRate() {

if (historicalValues.size() < 2) {

return 0.0;

}

double totalGrowthRate = 0.0;

int growthPeriods = 0;

for (int i = 1; i < historicalValues.size(); i++) {

double previousValue = historicalValues.get(i - 1);

double currentValue = historicalValues.get(i);

if (previousValue > 0) {

double growthRate = (currentValue - previousValue) / previousValue;

totalGrowthRate += growthRate;

growthPeriods++;

}

}

return growthPeriods > 0 ? totalGrowthRate / growthPeriods : 0.0;

}

public double calculateFutureValueRecursive(double initialValue, double growthRate, int years) {

recursiveCallCount++;

if (years == 0) {

return initialValue;

}

return calculateFutureValueRecursive(initialValue \* (1 + growthRate), growthRate, years - 1);

}

public double calculateFutureValueMemoized(double initialValue, double growthRate, int years) {

String key = initialValue + "," + growthRate + "," + years;

if (memoCache.containsKey(key)) {

return memoCache.get(key);

recursiveCallCount++;

if (years == 0) {

memoCache.put(key, initialValue);

return initialValue;

}

double result = calculateFutureValueMemoized(initialValue \* (1 + growthRate), growthRate, years - 1);

memoCache.put(key, result);

return result;

}

public double calculateFutureValueOptimized(double initialValue, double growthRate, int years) {

return calculateFutureValueTailRecursive(initialValue, growthRate, years, 1.0);

}

private double calculateFutureValueTailRecursive(double initialValue, double growthRate, int years, double accumulator) {

recursiveCallCount++;

if (years == 0) {

return initialValue \* accumulator;

}

return calculateFutureValueTailRecursive(initialValue, growthRate, years - 1, accumulator \* (1 + growthRate));

}

public double calculateFutureValueIterative(double initialValue, double growthRate, int years) {

double result = initialValue;

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

result \*= (1 + growthRate);

}

return result;

}

public double calculateCompoundInterest(double principal, double rate, int compoundingPeriods, int years) {

return calculateCompoundRecursive(principal, rate / compoundingPeriods, compoundingPeriods \* years);

}

private double calculateCompoundRecursive(double amount, double periodRate, int totalPeriods) {

recursiveCallCount++;

if (totalPeriods == 0) {

return amount;

}

return calculateCompoundRecursive(amount \* (1 + periodRate), periodRate, totalPeriods - 1);

}

public void comparePerformance() {

double initialValue = 100000.0;

double growthRate = calculateAverageGrowthRate();

int years = 15;

System.out.println("\nTesting calculation methods:");

System.out.println("Starting: $" + String.format("%.0f", initialValue));

System.out.println("Growth: " + String.format("%.1f", growthRate \* 100) + "% yearly");

System.out.println();

recursiveCallCount = 0;

long startTime = System.nanoTime();

double result1 = calculateFutureValueRecursive(initialValue, growthRate, years);

long endTime = System.nanoTime();

System.out.println("Basic recursive:");

System.out.println(" Result: $" + String.format("%.0f", result1));

System.out.println(" Time: " + (endTime - startTime) / 1\_000\_000.0 + " ms");

System.out.println(" Calls: " + recursiveCallCount);

System.out.println();

memoCache.clear();

recursiveCallCount = 0;

startTime = System.nanoTime();

double result2 = calculateFutureValueMemoized(initialValue, growthRate, years);

endTime = System.nanoTime();

System.out.println("With caching:");

System.out.println(" Result: $" + String.format("%.0f", result2));

System.out.println(" Time: " + (endTime - startTime) / 1\_000\_000.0 + " ms");

System.out.println(" Calls: " + recursiveCallCount);

System.out.println();

startTime = System.nanoTime();

double result3 = calculateFutureValueIterative(initialValue, growthRate, years);

endTime = System.nanoTime();

System.out.println("Simple loop:");

System.out.println(" Result: $" + String.format("%.0f", result3));

System.out.println(" Time: " + (endTime - startTime) / 1\_000\_000.0 + " ms");

}

public void generateForecastReport() {

double currentValue = historicalValues.get(historicalValues.size() - 1);

double growthRate = calculateAverageGrowthRate();

System.out.println("\nYour Investment Forecast:");

System.out.println("Current value: $" + String.format("%.0f", currentValue \* 1000));

System.out.println("Average growth: " + String.format("%.1f", growthRate \* 100) + "% yearly");

System.out.println();

System.out.println("Future projections:");

for (int year = 5; year <= 15; year += 5) {

double futureValue = calculateFutureValueOptimized(currentValue, growthRate, year);

System.out.println(" " + year + " years: $" + String.format("%.0f", futureValue \* 1000));

}

}

public static void main(String[] args) {

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

System.out.println("Smart algorithms for financial predictions");

System.out.println();

FinancialForecasting forecaster = new FinancialForecasting();

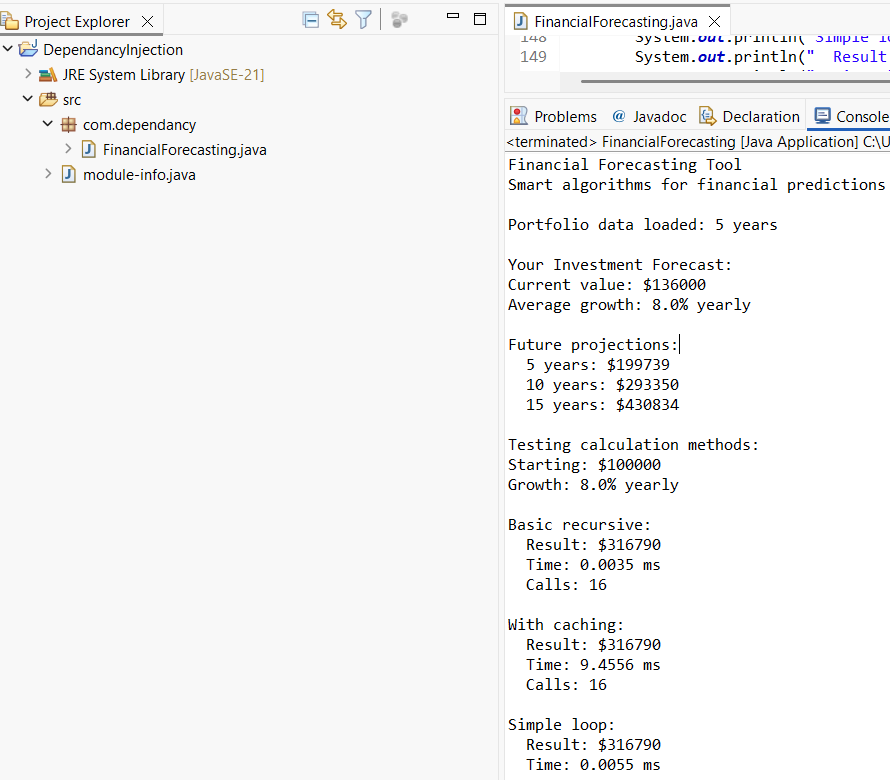
forecaster.generateForecastReport();

forecaster.comparePerformance();

}

}

**Output:**

****