Data structures and Algorithms

**1. E-commerce Platform Search Function  
  
Code :   
  
SearchAlgorithms.java**

import models.Product;

import java.util.Arrays;

import java.util.Comparator;

public class SearchAlgorithms {

// Linear Search

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

for (Product p : products) {

if (p.productId == targetId) {

return p;

}

}

return null;

}

// Binary Search

public static Product binarySearch(Product[] products, int targetId) {

int left = 0, right = products.length - 1;

while (left <= right) {

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

if (products[mid].productId == targetId) {

return products[mid];

} else if (products[mid].productId < targetId) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

// sort

public static void sortById(Product[] products) {

Arrays.sort(products, Comparator.comparingInt(p -> p.productId));

}

}

**Product.java**

public class Product {

public int productId;

public String productName;

public String category;

public Product(int productId, String productName, String category) {

this.productId = productId;

this.productName = productName;

this.category = category;

}

@Override

public String toString() {

return "[" + productId + "] " + productName + " (" + category + ")";

}

}

**Main.java**

import algorithms.SearchAlgorithms;

import models.Product;

public class Main {

public static void main(String[] args) {

Product[] products = {

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

new Product(202, "Shoes","Footwear"),

new Product(303, "Phone","Electronics"),

new Product(404, "Watch","Accessories")

};

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

Product found1 = SearchAlgorithms.linearSearch(products, 303);

System.out.println(found1 != null ? found1 : "Product not found");

SearchAlgorithms.sortById(products);

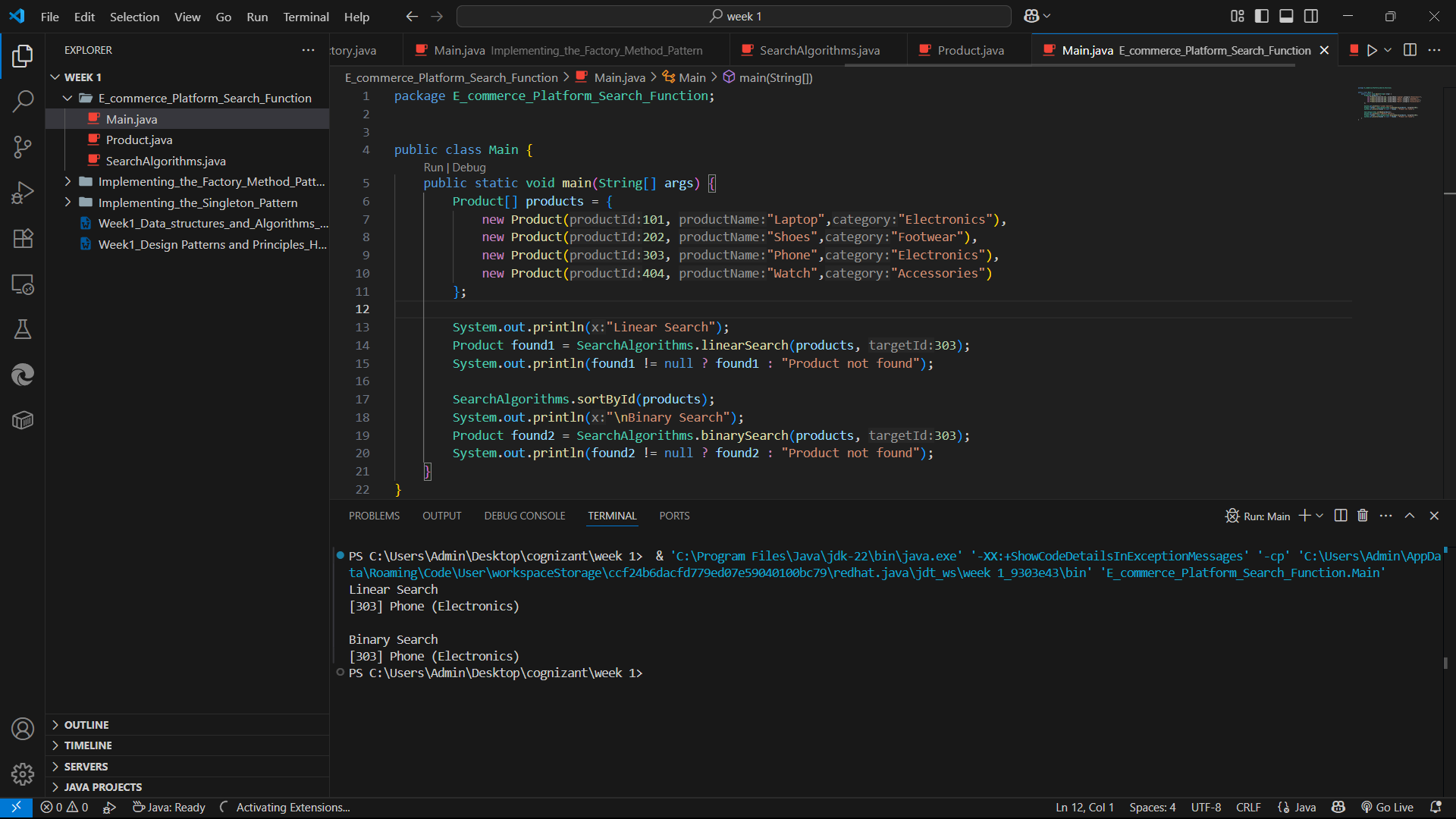
System.out.println("\nBinary Search");

Product found2 = SearchAlgorithms.binarySearch(products, 303);

System.out.println(found2 != null ? found2 : "Product not found");

}

}

**Output :  
  
  
**

**Analysis :**

| **Algorithm** | **Time Complexity** | **Space Complexity** | **Suitable for?** |
| --- | --- | --- | --- |
| **Linear Search** | O(n) | O(1) | Small datasets, unsorted data |
| **Binary Search** | O(log n) | O(1) | Large datasets, sorted data |

Binary Search’s O(log n) efficiency makes it faster, scalable search on large e‑commerce platform

**2. Financial Forecasting  
  
Code :**

**FinancialForecasting.java**

public class FinancialForecast {

public static double forecastRecursive(double initialValue, double[] rates, int yearIndex) {

if (yearIndex == 0) {

return initialValue;

}

double prev = forecastRecursive(initialValue, rates, yearIndex - 1);

return prev \* (1 + rates[yearIndex - 1]);

}

public static double forecastMemo(double initialValue, double[] rates, int yearIndex, Double[] cache) {

if (yearIndex == 0) {

return initialValue;

}

if (cache[yearIndex] != null) {

return cache[yearIndex];

}

double prev = forecastMemo(initialValue, rates, yearIndex - 1, cache);

double curr = prev \* (1 + rates[yearIndex - 1]);

cache[yearIndex] = curr;

return curr;

}

public static double forecastIterative(double initialValue, double[] rates, int yearIndex) {

double value = initialValue;

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

value \*= (1 + rates[i]);

}

return value;

}

public static Double[] newCache(int years) {

return new Double[years + 1];

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

double initialValue = 1000.0;

double[] growthRates = { 0.05, 0.04, 0.06, 0.03 };

int targetYear = 4;

double rec = FinancialForecast.forecastRecursive(initialValue, growthRates, targetYear);

System.out.printf("Recursive → Year %d: %.2f%n", targetYear, rec);

Double[] cache = FinancialForecast.newCache(targetYear);

double memo = FinancialForecast.forecastMemo(initialValue, growthRates, targetYear, cache);

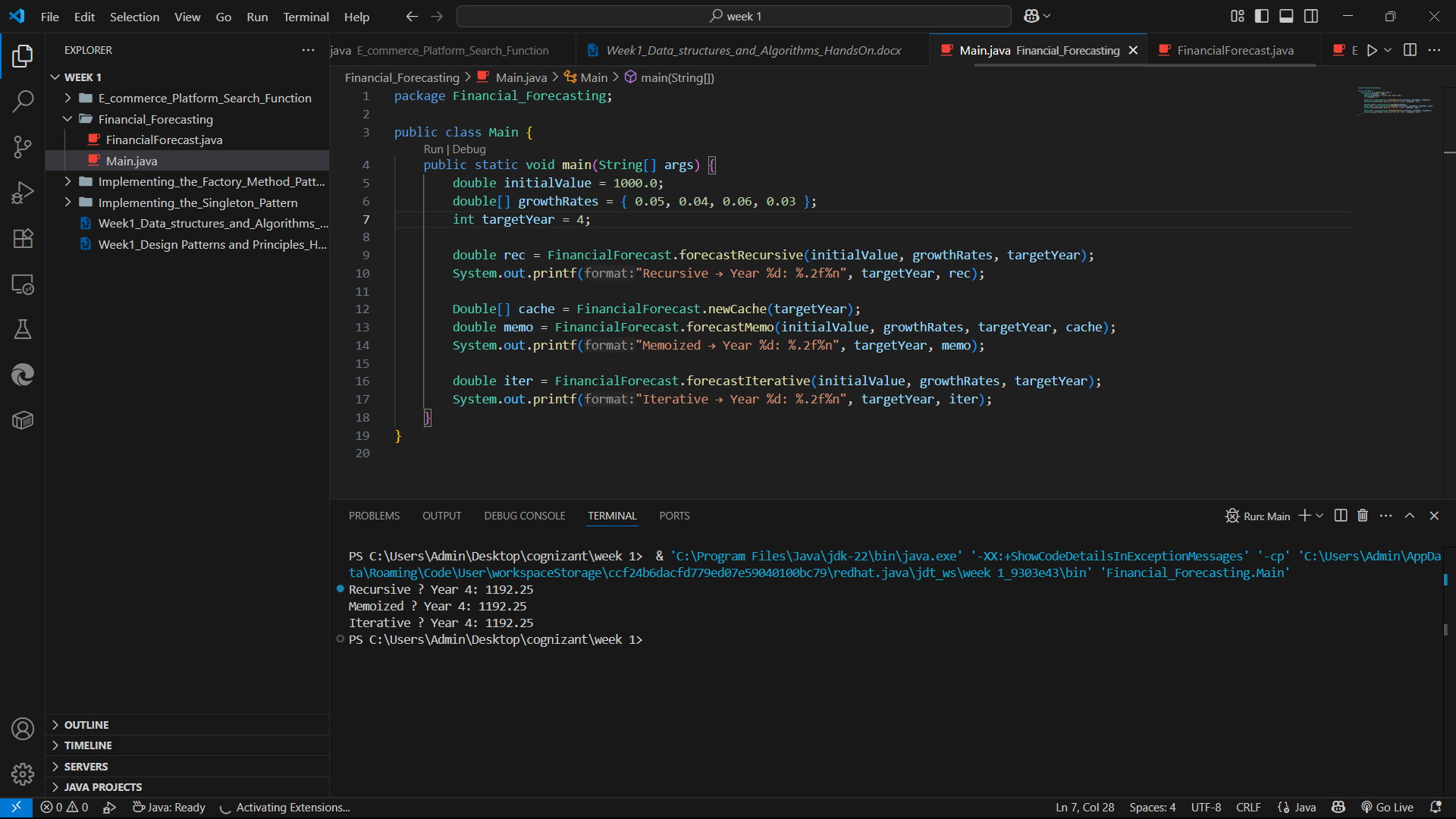
System.out.printf("Memoized → Year %d: %.2f%n", targetYear, memo);

double iter = FinancialForecast.forecastIterative(initialValue, growthRates, targetYear);

System.out.printf("Iterative → Year %d: %.2f%n", targetYear, iter);

}

}

**Output :**

**Analysis :**

**Time Complexity**: O(n) for all three;

**Optimization**: forecastMemo avoids repeated subcalls, forecastIterative avoids recursion entirely.