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

Understanding Asymptotic Notations (Short)

Asymptotic notation, especially Big O notation, is used to analyze the efficiency of algorithms by describing how their performance scales with input size. It helps estimate the best, average, and worst-case scenarios. The best case is the fastest possible outcome, the average case reflects typical performance, and the worst case shows the maximum time an algorithm might take. These help in selecting the most efficient algorithm for different situations.

**Product.java**

public class Product {

    private int productId;

    private String productName;

    private String category;

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

        this.productId = id;

        this.productName = name;

        this.category = cat;

    }

    public int getProductId() {

        return productId;

    }

    public String getProductName() {

        return productName;

    }

    public String getCategory() {

        return category;

    }

    @Override

    public String toString() {

        return "Product [ID=" + productId + ", Name=" + productName + ", Category=" + category + "]";

    }

}

**ProductSearch.java**

import java.util.Arrays;

import java.util.Comparator;

public class ProductSearch {

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

        for (Product p : products) {

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

                return p;

            }

        }

        return null;

    }

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

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

        int low = 0, high = products.length - 1;

        while (low <= high) {

            int mid = (low + high) / 2;

            int compare = products[mid].getProductName().compareToIgnoreCase(targetName);

            if (compare == 0) {

                return products[mid];

            } else if (compare < 0) {

                low = mid + 1;

            } else {

                high = mid - 1;

            }

        }

        return null;

    }

}

**SearchDemo.java**

public class SearchDemo {

    public static void main(String[] args) {

        Product[] catalog = {

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

            new Product(102, "Shoes", "Fashion"),

            new Product(103, "Mobile", "Electronics"),

            new Product(104, "Watch", "Accessories"),

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

        };

        String targetProduct = "Mobile";

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

        Product foundLinear = ProductSearch.linearSearch(catalog, targetProduct);

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

        System.out.println("\nUsing Binary Search:");

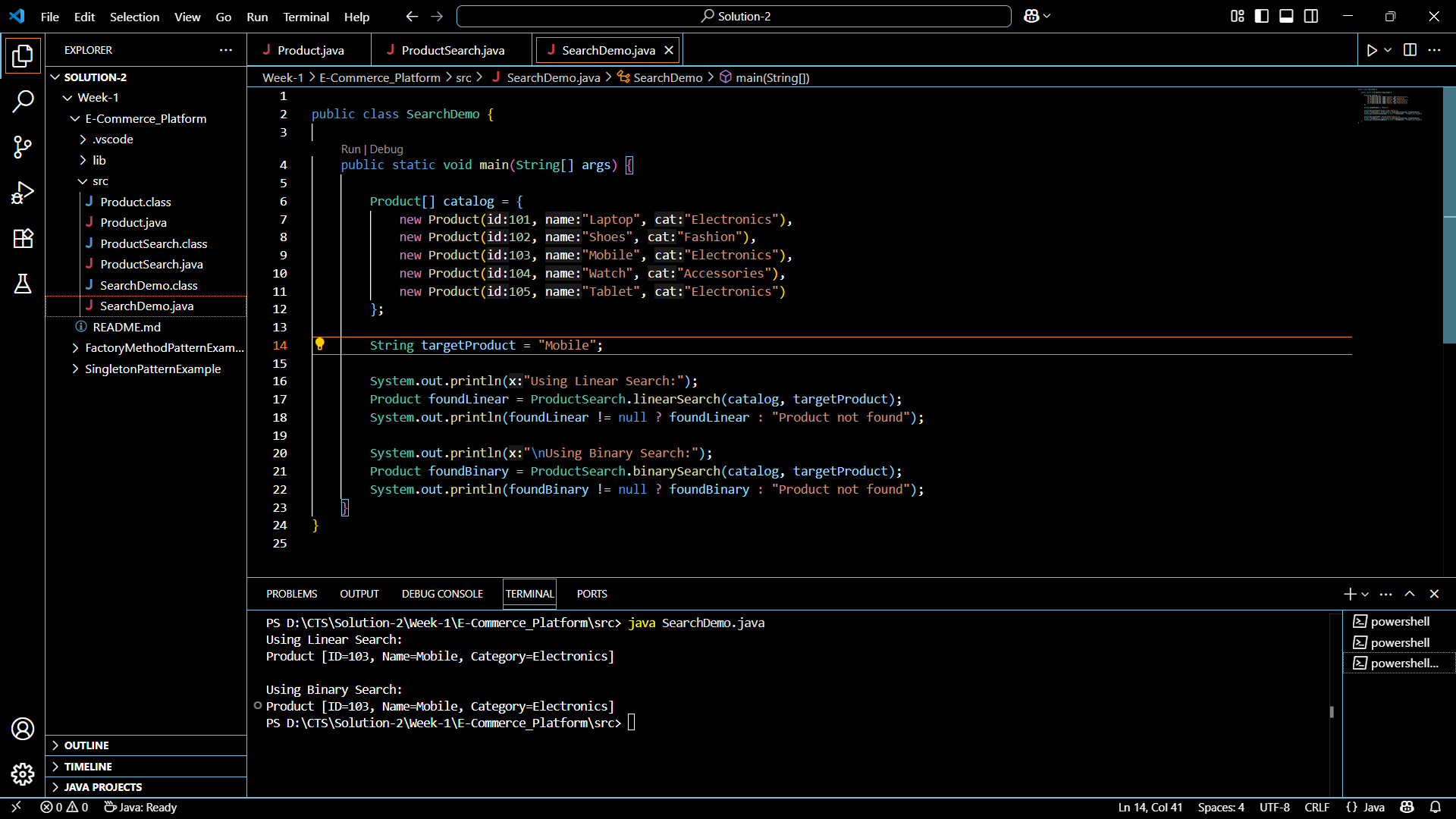
        Product foundBinary = ProductSearch.binarySearch(catalog, targetProduct);

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

    }

}

**Output**



**Analysis of Search Algorithms (Short)**

**Linear search** checks each element one by one and works on unsorted data , but has a time complexity of **O(n)**, making it slow for large datasets. **Binary search**, with **O(log n)** time complexity, is faster but requires the data to be **sorted**. For an e-commerce platform with large product data, **binary search** is more suitable due to its speed and scalability, ensuring faster and more efficient search performance.

**Exercise 7: Financial Forecasting**

Understand Recursive Algorithms

Recursion is a programming technique where a method calls itself to solve smaller instances of a problem. It's particularly useful for problems that can be divided into similar sub-problems. Recursion simplifies the code and mirrors the mathematical definition of many problems, such as calculating factorial, Fibonacci numbers, or in this case, predicting future financial values based on consistent growth.

Code:

FinancialForecast.java

package forecasting;

public class FinancialForecast {

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

if (years == 0) {

return currentValue;

} else {

return predictValue(currentValue \* (1 + growthRate / 100), growthRate, years - 1);

}

}

public static void main(String[] args) {

double initialValue = 10000;

double growthRate = 7.5;

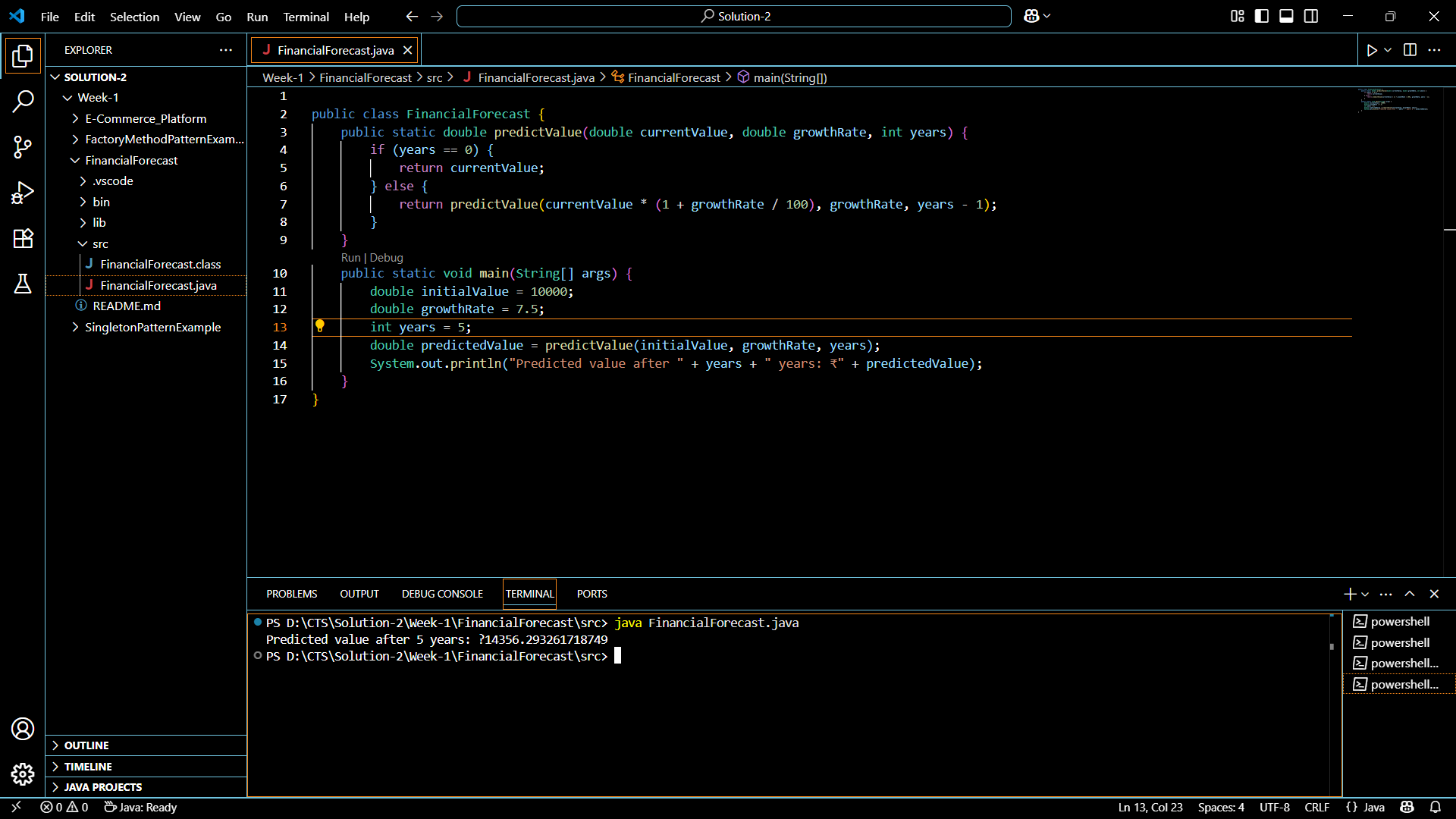
int years = 5;

double predictedValue = predictValue(initialValue, growthRate, years);

System.out.println("Predicted value after " + years + " years: ₹" + predictedValue);

}

}  
  
**Output:**

****

Time Complexity:

* The time complexity of the recursive function is O(n), where *n* is the number of years.
* Each recursive call reduces years by one until it reaches the base case (years == 0), resulting in *n* calls.

Optimization:

* While this recursion is not deeply nested, it can still be optimized for performance.
* Tail recursion (if supported by the compiler) or iteration can help.
* Alternatively, caching intermediate results helps when the computation overlaps (e.g., in Fibonacci).