**DATA STRUCTURES AND ALGORITHMS**

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

**1.Understanding Asymptotic Notation**

**What is Big O Notation?**

Big O notation is a mathematical concept used in computer science to **describe the performance or complexity** of an algorithm in terms of input size (n). It helps predict **how long an algorithm will take** to run or how much memory it will use, regardless of hardware or programming language.

**Why is it Useful?**

In large-scale applications like e-commerce platforms with millions of products, knowing how algorithms scale with increasing data helps us choose the most efficient one, which in turn improves:

* **Response time**
* **User experience**
* **Server performance**

**Types of Cases in Big O:**

| **Case** | **Description** | **Example (Search)** |
| --- | --- | --- |
| **Best Case** | The desired product is found in the first comparison | Searching for the first product |
| **Average Case** | The product is found somewhere in the middle | Searching for a random product |
| **Worst Case** | Product is last in the list or not present at all | Searching for a non-existent product |

**2.Setup**

In Java, a product in the e-commerce platform is represented using a Product class with the following attributes:

* **productId**: Unique identifier for each product
* **productName**: Name used for search
* **category**: Classification (e.g., Electronics)

**Product.java**

public class Product {

    int productId;

    String productName;

    String category;

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

        this.productId = productId;

        this.productName = productName;

        this.category = category;

    }

}

**3.Implementation**

**A. Linear Search – O(n)**

This method scans each product one by one and compares the name.

**B. Binary Search – O(log n)**

Binary search only works after sorting the product list (by name). It divides the list into halves repeatedly to locate the product.

**EcommerceSearch.java**

import java.util.\*;

public class EcommerceSearch {

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

        for (int i = 0; i < products.length; i++) {

            if (products[i].productName.equalsIgnoreCase(target)) {

                return i;

            }

        }

        return -1;

    }

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

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

        while (low <= high) {

            int mid = (low + high) / 2;

            int comparison = products[mid].productName.compareToIgnoreCase(target);

            if (comparison == 0)

                return mid;

            else if (comparison < 0)

                low = mid + 1;

            else

                high = mid - 1;

        }

        return -1;

    }

    public static void main(String[] args) {

        Product[] products = {

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

            new Product(102, "T-shirt", "Apparel"),

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

            new Product(104, "Book", "Education")

        };

        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter product name to search: ");

        String input = scanner.nextLine();

        // Linear Search

        int linearResult = linearSearch(products, input);

        if (linearResult != -1)

            System.out.println("Linear Search: Found at index " + linearResult);

        else

            System.out.println("Linear Search: Product not found");

        // Sort products by name for binary search

        Arrays.sort(products, Comparator.comparing(p -> p.productName.toLowerCase()));

        // Binary Search

        int binaryResult = binarySearch(products, input);

        if (binaryResult != -1)

            System.out.println("Binary Search: Found at index " + binaryResult + " (sorted)");

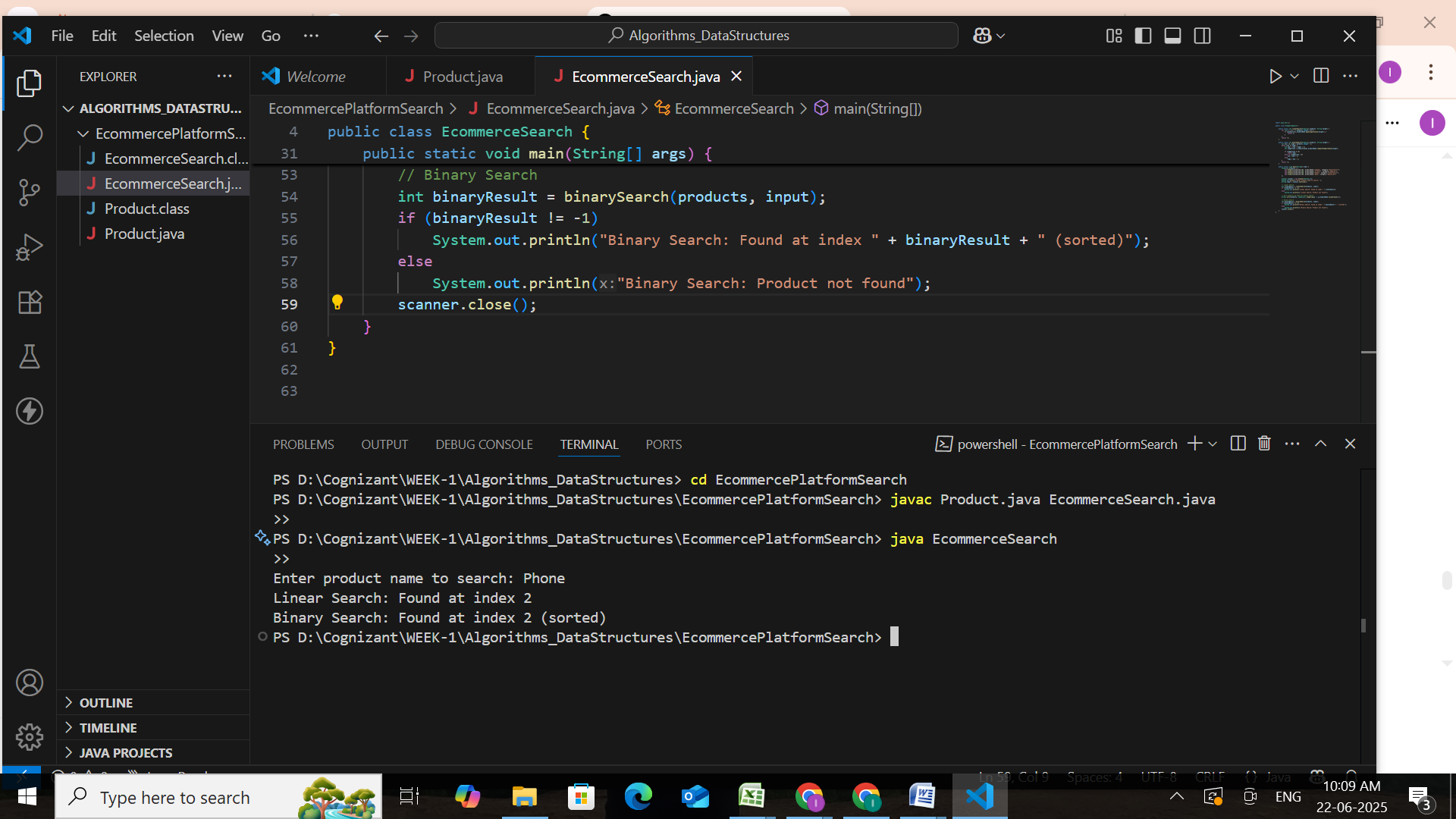
        else

            System.out.println("Binary Search: Product not found");

        scanner.close();

    }

}

**Output:**  


**4.Analysis**

**Time Complexity Comparison:**

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** | **Time Complexity** |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| **Linear Search** | O(1) | O(n) | O(n) | Slower for large data |
| **Binary Search** | O(1) | O(log n) | O(log n) | Much faster with sorted data |

**Suitable Algorithm:**

**Binary Search** is more suitable because:

* It’s **much faster** for large product lists.
* It requires sorted data, but this can be done once.
* Provides **better user experience** with quick search results.

**Linear Search** is only ideal for **small or unsorted** datasets.

**Exercise 7: Financial Forecasting**

#### **1.Understanding Recursive Algorithms**

#### **What is Recursion?**

**Recursion** is a method where a function **calls itself** to solve smaller subproblems. The idea is to **break down** a large problem into smaller, manageable parts, and solve the smallest one first (the **base case**). The result then builds up from there. A recursive function has two main parts:

 **Base Case** – The stopping condition where the function stops calling itself.

 **Recursive Case** – The part where the function calls itself with a smaller input.

**Why is Recursion Useful?**

 It simplifies code for problems that involve **repeated computation**.

 It's ideal for tasks where each step **depends on the result of a previous step** (e.g., predicting future values year by year).

 Recursion can make the code **more readable** and **closer to the mathematical formula**.

**2. Setup:**

We are required to calculate the amount of money (future value) after a given number of years, considering a fixed annual growth or interest rate. This is a classic compound interest problem that can be broken down recursively.

We use the formula:

**Future Value = Present Value × (1 + rate)^years**

To apply recursion, we redefine this as:

* **Base Case:**  
  If the number of years is 0, the future value is equal to the present value.  
  FV(p, r, 0) = p
* **Recursive Case:**  
  If years > 0, then we calculate:  
  FV(p, r, y) = FV(p, r, y-1) × (1 + r)

Where:

* p = present value
* r = annual growth rate (as decimal)
* y = number of year

**3. Implementation:**

package FinancialForecastApp;

public class FinancialForecast {

    public static double calculateFutureValue(double presentValue, double rate, int years) {

        if (years == 0) {

            return presentValue; // base case

        }

        return calculateFutureValue(presentValue, rate, years - 1) \* (1 + rate);

    }

    public static double calculateFutureValueIterative(double presentValue, double rate, int years) {

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

            presentValue \*= (1 + rate);

        }

        return presentValue;

    }

    public static void main(String[] args) {

        double presentValue = 10000.0; // ₹10,000

        double rate = 0.08; // 8% growth rate

        int years = 5;

        // Using recursive method

        double futureRecursive = calculateFutureValue(presentValue, rate, years);

        System.out.printf("Recursive: Value after %d years = Rs.%.2f%n", years, futureRecursive);

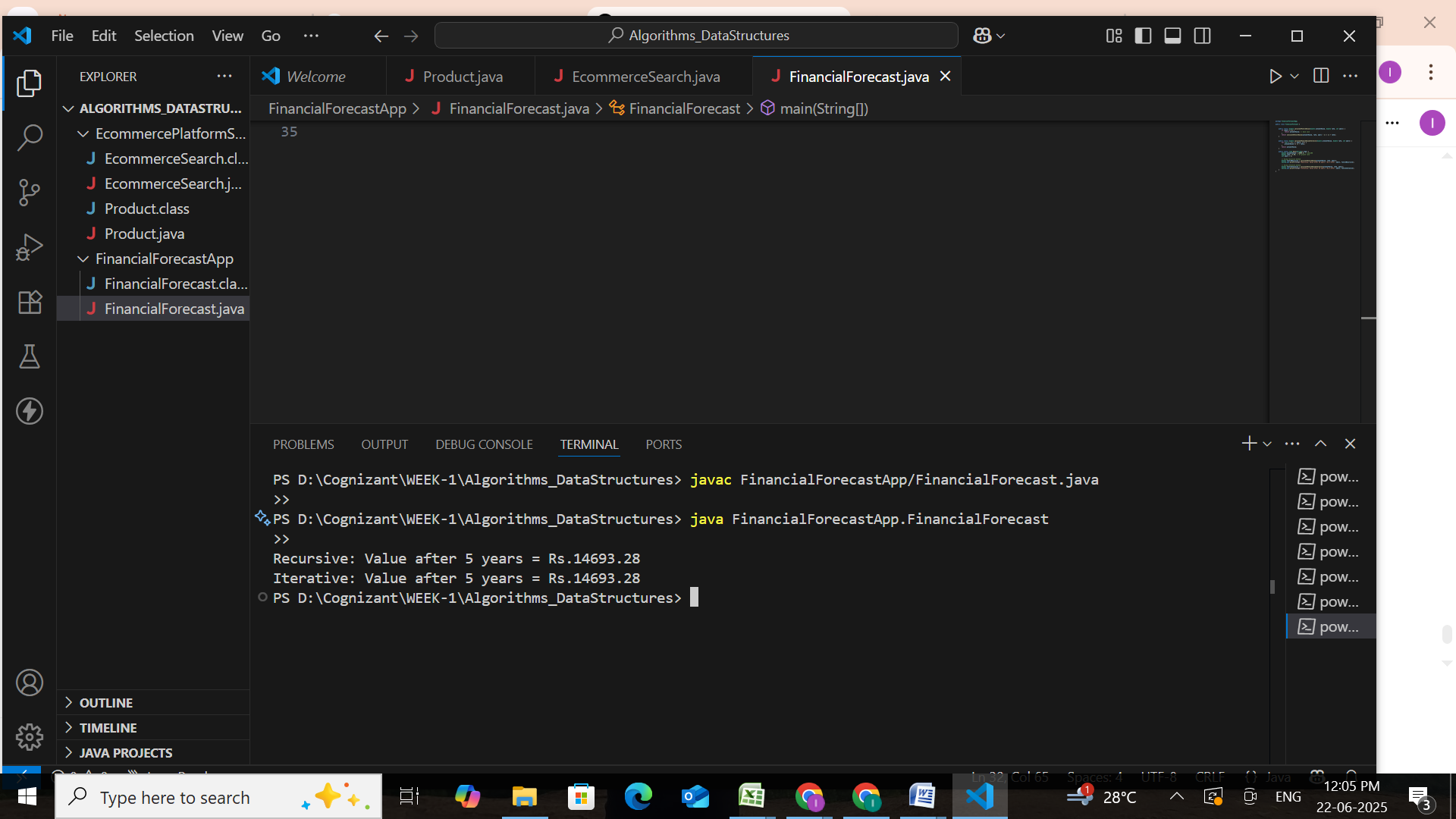
        // Using iterative method

        double futureIterative = calculateFutureValueIterative(presentValue, rate, years);

        System.out.printf("Iterative: Value after %d years = Rs.%.2f%n", years, futureIterative);

    }

}

**Output:** **4. Analysis:**

#### **Time Complexity:**

In the recursive approach to predicting future financial value, the method calls itself once for each year until it reaches the base case (when years = 0).

**T(n) = T(n - 1) + O(1)**

**T(n) = O(n)**

Therefore, the overall time complexity is linear with respect to the number of years.

### Optimization:

**Problem with Recursion:**

Although recursion simplifies the logic, it comes with **two main issues**:

1. **Stack Overhead**: Each recursive call consumes memory on the call stack.
2. **Performance Bottleneck**: For large inputs (e.g., 10,000 years), deep recursion can lead to **stack overflow** or **slow execution**.

## Optimization Strategy: Convert Recursion to Iteration

### Why Iteration?

* Uses **loops instead of recursive calls**
* **No call stack memory** required
* **Faster and safer** for large input values

public static double calculateFutureValueIterative(double presentValue, double rate, int years) {

    for (int i = 0; i < years; i++) {

        presentValue \*= (1 + rate);

    }

    return presentValue;

}

This version avoids recursion and is better when working with very large n values to prevent **stack overflow** or unnecessary function call overhead.

| **Feature** | **Recursive** | **Iterative (Optimized)** |
| --- | --- | --- |
| **Time Complexity** | **O(n)** | **O(n)** |
| **Space Complexity** | **O(n) (due to stack)** | **O(1)** |
| **Risk of Stack Overflow** | **Yes** | **No** |
| **Performance (Large n)** | **Slower** | **Faster** |