**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 **efficiency** of an algorithm in relation to the input size, usually represented as n. It helps determine **how quickly** an algorithm runs or how much **memory** it consumes, especially as the input grows.

**Why is Big O Important?**

In large systems like **e-commerce platforms** (where millions of products may be searched or filtered), understanding Big O helps developers:

* Choose faster, more efficient algorithms
* Improve **response time**
* Enhance **user experience**
* Reduce **server load and processing time**

**Types of Cases in Big O:**

| **Case** | **Description** | **Example (Search)** |
| --- | --- | --- |
| **Best Case** | The target product is found in the very first check | Searching for the first item in a list |
| **Average Case** | The product is located somewhere in the middle | Searching for a random item in an unsorted list |
| **Worst Case** | The item is at the end or doesn’t exist in the list | Searching for a product that is not in the catalog |

**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 technique goes through the list of products **sequentially**, checking each product's name with the target name. It continues until a match is found or the list ends.

* **Use case:** Works on both sorted and unsorted product arrays.
* **Efficiency:** Time complexity is **O(n)** because, in the worst case, every element might be checked.

**B. Binary Search – O(log n)**

Binary search is a more efficient technique but requires the product list to be **sorted alphabetically by name** beforehand. It works by repeatedly dividing the array in half and comparing the middle element with the target.

* **Use case:** Only works on **sorted lists**.
* **Efficiency:** Time complexity is **O(log n)** because the search area is halved with each step.

**EcommerceSearch.java**

import java.util.\*;

public class EcommerceSearch{

    public static int findProductLinear(Product[] items, String keyword) {

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

            if (items[i].productName.equalsIgnoreCase(keyword)) {

                return i;

            }

        }

        return -1;

    }

    public static int findProductBinary(Product[] items, String keyword) {

        int start = 0;

        int end = items.length - 1;

        while (start <= end) {

            int middle = (start + end) / 2;

            int result = items[middle].productName.compareToIgnoreCase(keyword);

            if (result == 0)

                return middle;

            else if (result < 0)

                start = middle + 1;

            else

                end = middle - 1;

        }

        return -1;

    }

    public static void main(String[] args) {

        Product[] itemList = {

            new Product(201, "Camera", "Electronics"),

            new Product(202, "Notebook", "Stationery"),

            new Product(203, "Sneakers", "Footwear"),

            new Product(204, "Watch", "Accessories")

        };

        Scanner inputScanner = new Scanner(System.in);

        System.out.print("Search for a product by name: ");

        String searchQuery = inputScanner.nextLine();

        // linear search

        int foundIndexLinear = findProductLinear(itemList, searchQuery);

        if (foundIndexLinear != -1)

            System.out.println("Linear Search: Product found at position " + foundIndexLinear);

        else

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

        // Sort products before binary search

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

        // binary search

        int foundIndexBinary = findProductBinary(itemList, searchQuery);

        if (foundIndexBinary != -1)

            System.out.println("Binary Search: Product found at position " + foundIndexBinary + " (sorted list)");

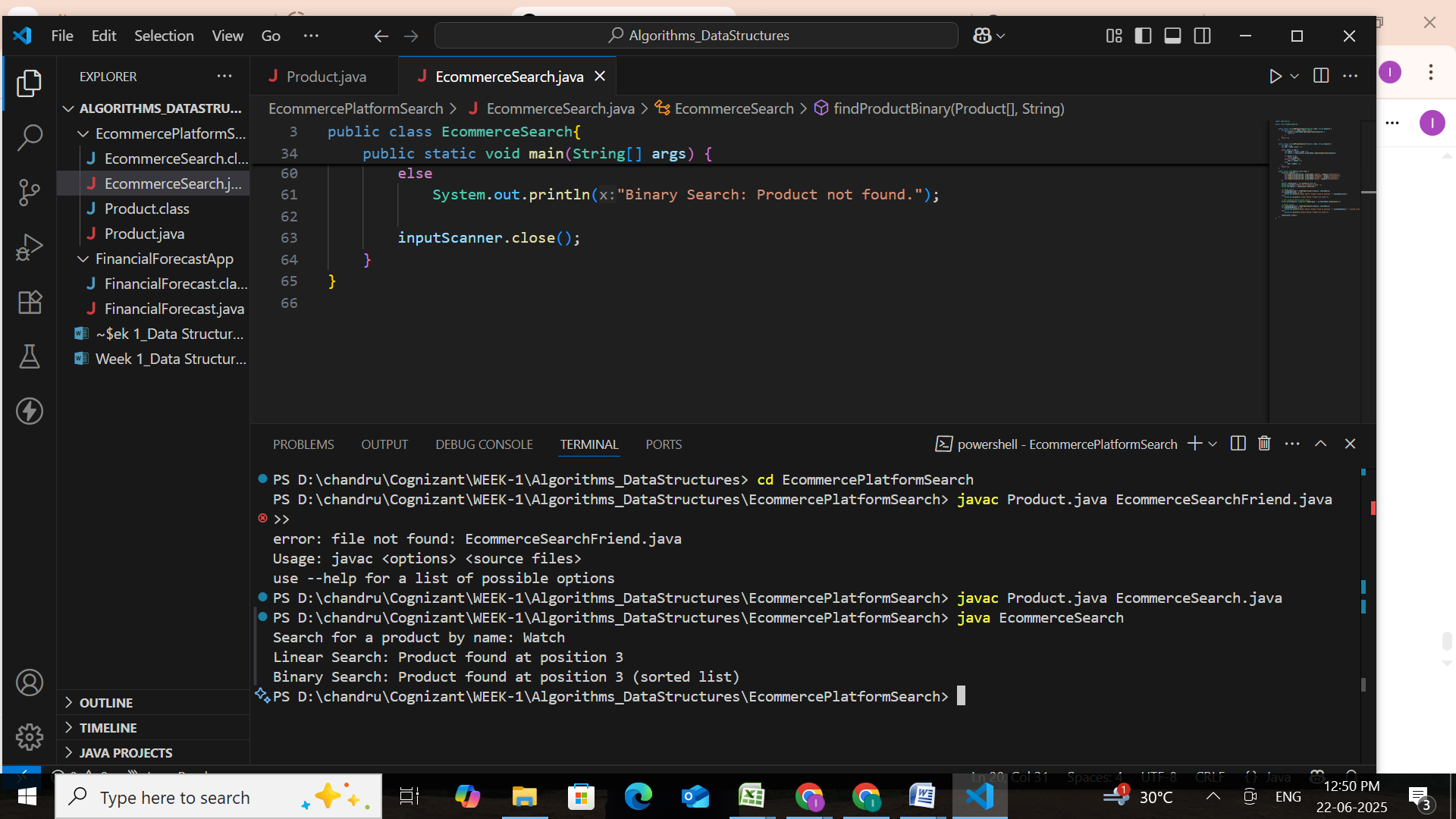
        else

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

        inputScanner.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 as the number of products increases | |
| **Binary Search** | O(1) | O(log n) | O(log n) | |  |  | | --- | --- | |  | Much faster on sorted product lists | |

**Suitable Algorithm:**

**Binary Search** is generally the better option for an e-commerce platform because:

* It handles **large datasets efficiently**
* Once the product list is sorted, search operations become very quick
* It delivers a **faster and smoother experience** for users

**Linear Search**, on the other hand:

* Is simple and does **not require sorting**
* But becomes inefficient as data grows
* Best used only when the dataset is **small or frequently unsorted**

.

**Exercise 7: Financial Forecasting**

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

**Recursion** is a programming technique where a method calls itself to solve smaller instances of a problem. It is particularly useful when a problem can be broken down into similar sub-problems.

In financial forecasting, recursion helps model a situation where each year’s future value is based on the previous year’s value. This makes the logic clean and closely aligned with the actual formula used in finance.

#### **Advantages of Using Recursion:**

* Leads to **cleaner, shorter code**
* Matches the **mathematical representation** of many problems
* Easier to understand and debug for problems with repeated structure (e.g., trees, sequences, compound interest)

**2. Setup:**

In this task, we aim to build a method that calculates how much an investment will be worth after a specific number of years, given a steady annual growth rate.

This situation models **compound growth**, and rather than solving it using a standard formula, we'll take a **recursive approach**, which breaks the calculation into repeated yearly growth steps.

In financial forecasting, the **future value** of an investment can be calculated using the compound interest formula:

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

To implement this recursively, we reframe the formula by computing the value **one year at a time**.

We define a function **FV(p, r, y)** that returns the future value for:

* p = present amount,
* r = annual growth rate (as a decimal),
* y = number of years.

**3. Implementation:**

package FinancialForecasting;

public class FinancialForecast {

    public static double computeProjection(double principal, double rate, int time) {

        if (time == 0) {

            return principal;

        }

        double previousYear = computeProjection(principal, rate, time - 1);

        return previousYear + (previousYear \* rate);

    }

    public static double computeProjectionIterative(double principal, double rate, int time) {

        double result = principal;

        for (int t = 1; t <= time; t++) {

            result += result \* rate;

        }

        return result;

    }

    public static void main(String[] args) {

        double principal = 7500.0;  // Starting with ₹7,500

        double annualRate = 0.10;   // 10% annual increase

        int yearsAhead = 6;

        double projectedRecursive = computeProjection(principal, annualRate, yearsAhead);

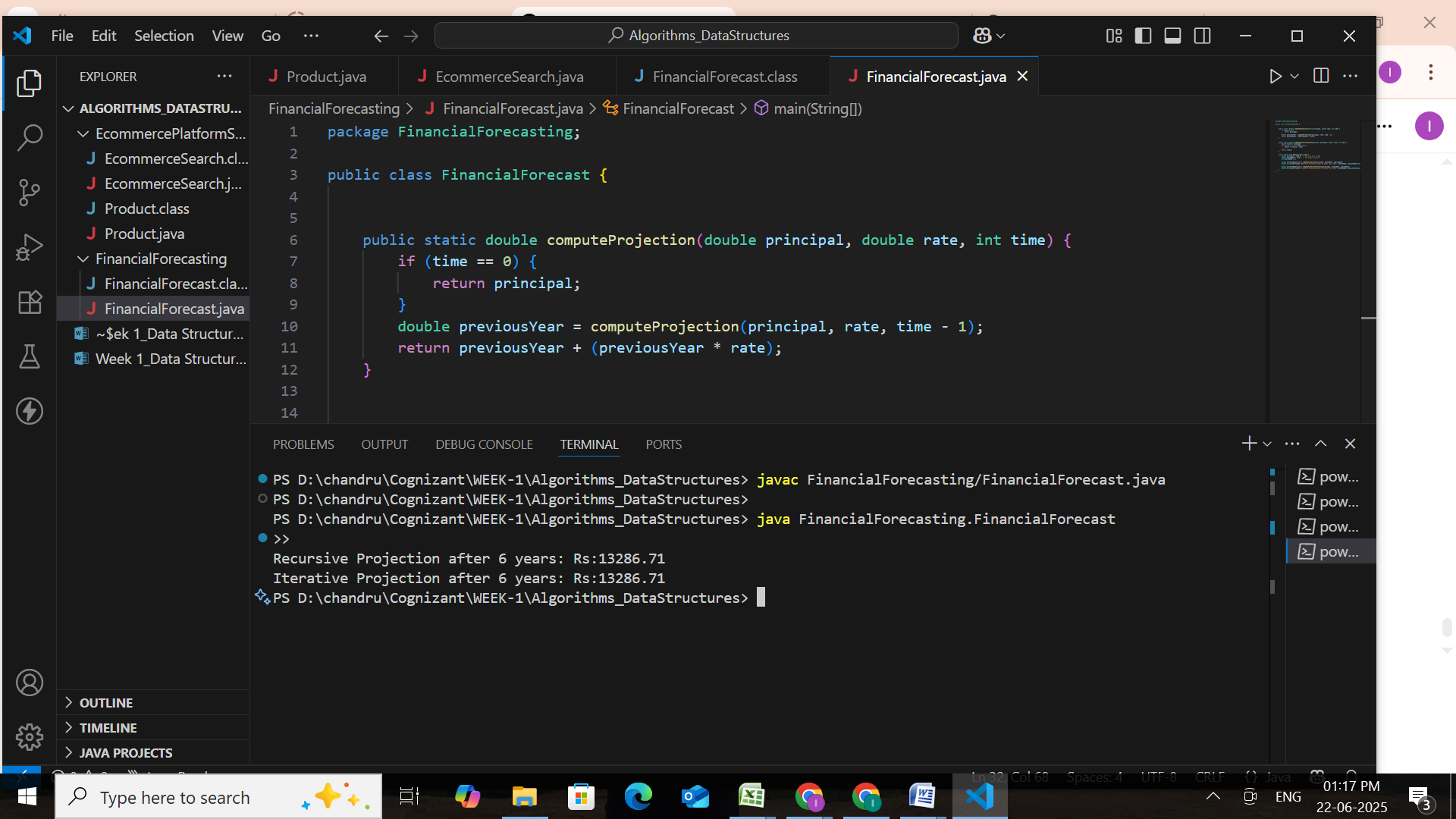
        System.out.printf("Recursive Projection after %d years: Rs:%.2f%n", yearsAhead, projectedRecursive);

        double projectedIterative = computeProjectionIterative(principal, annualRate, yearsAhead);

        System.out.printf("Iterative Projection after %d years: Rs:%.2f%n", yearsAhead, projectedIterative);

    }

}

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

#### **Time Complexity:**

In the recursive method used to calculate the projected future value, the function calls itself **once per year**, reducing the year count by 1 on each call until it hits zero.

This leads to the following recurrence relation:

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

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

So, the time complexity is **linear**, meaning the function scales directly with the number of years provided as input.

### Optimization: Switch to Iteration

**Limitations with Recursion:**

 **Call Stack Consumption**:  
Every recursive call adds a new frame to the call stack, which can consume significant memory for large n.

 **Stack Overflow Risk**:  
For very high input values (e.g., 10,000 years), the recursion depth might exceed the JVM's stack limit, resulting in a crash.

## Optimization Strategy:

### Why Iteration works better?

 Efficient use of memory (no function stack frames)

 Safe to run for very large input sizes

 Typically executes faster than recursion

public static double computeProjectedValueIterative(double amount, double rate, int years) {

    for (int i = 0; i < years; i++) {

        amount \*= (1 + rate);

    }

    return amount;

}

This iterative version is more reliable for **large projections** and is preferred in production-level financial tools where performance and stability matter.

| **Feature** | **Recursive Method** | **Iterative (Optimized) Method** |
| --- | --- | --- |
| **Time Complexity** | **O(n) – One call per year** | **O(n) – One loop per year** |
| **Space Complexity** | **O(n) – Due to function call stack** | **O(1) – Constant memory usage** |
| **Stack Overflow Risk** | **High – Deep recursion for large n** | **None – Safe even for very large inputs** |
| **Performance on Large n** | **Slower – Stack management adds overhead** | **Faster – Efficient and more stable** |