**Week-1**

**Algorithms and Data Structures**

**Exercise-2: Ecommerce Platform Search Function  
  
Implementation:**

import java.util.Scanner;

public class Main {

    public static void main(String[] args) {

        Product[] products = {

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

            new Product(102, "Shirt", "Clothing"),

            new Product(103, "Book", "Stationery"),

            new Product(104, "Mobile", "Electronics"),

            new Product(105, "Pen", "Stationery")

        };

Scanner sc = new Scanner(System.in);

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

        String searchName = sc.nextLine();

int result = SearchFunctions.linearSearch(products, searchName);

        if (result != -1)

            System.out.println("Linear Search: Found - " + products[result]);

        else

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

SearchFunctions.sortProducts(products);

        int result2 = SearchFunctions.binarySearch(products, searchName);

if (result2 != -1)

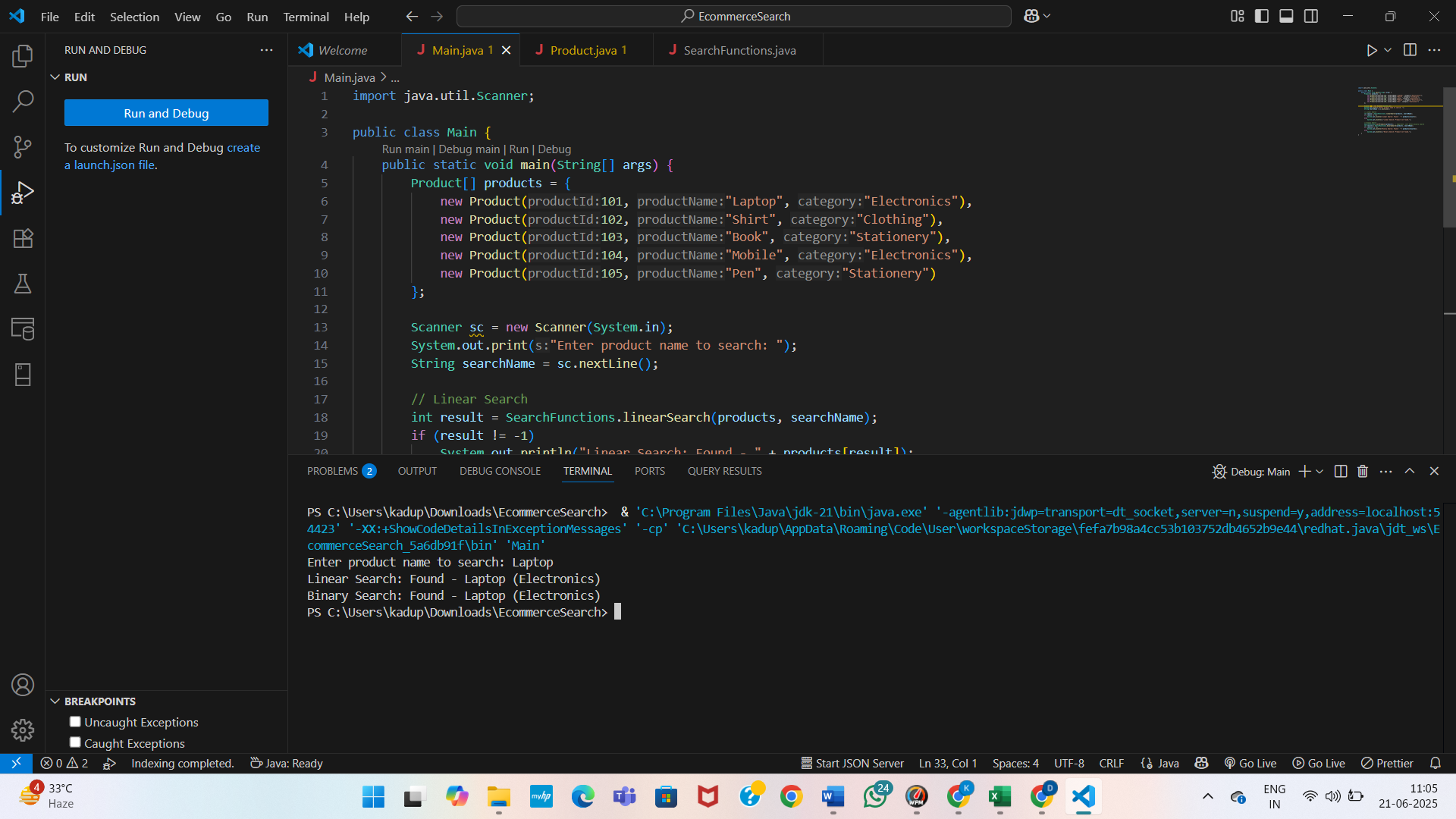
            System.out.println("Binary Search: Found - " + products[result2]);

        else

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

    }

}  
  
**Output:**

****

**Analysis Of Both Algorithms:**

| **Search Algorithm** | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| **Linear Search** | O(1) | O(n) | O(n) |
| **Binary Search** | O(1) | O(log n) | O(log n) |

**Linear Search** performs a sequential check of each element in the list until the desired item is found or the end is reached.

* Best case: item is at the first position → **O(1)**
* Worst case: item is not present or at the last position → **O(n)**

**Binary Search** divides the sorted array in half repeatedly and checks the middle element each time.

* Best case: item is exactly in the middle → **O(1)**
* Worst case: item is deep within one side of the array → **O(log n)**

**Suitable Algorithm for E-Commerce Platform:**

**Binary search** is **more suitable** for an e-commerce platform when:

* The product list is **pre-sorted by name or ID**.
* The product data is **mostly static** (i.e., not updated frequently).
* The platform handles **large datasets**, such as thousands or millions of products.

**Exercise 7: Financial Forecasting**

**What is Recursion?**

Recursion is a programming technique where a method calls itself to solve a smaller instance of the same problem. It helps in breaking down complex problems into simpler sub-problems. Each recursive call should eventually lead to a base case that stops further recursion.

In financial forecasting, recursion is useful when future values depend on previous values — such as predicting next year’s revenue based on the current year's revenue and growth rate.

**Why Use Recursion?**

Recursion simplifies logic where the problem can be defined in terms of itself. For example, to predict the revenue for year *n*, if we know the revenue for year *(n - 1)* and the annual growth rate, we can compute the current value as:

FutureValue(n) = FutureValue(n - 1) \* (1 + growthRate)

We define a recursive method to calculate the future value given:

* Initial value (e.g., revenue in year 0)
* Growth rate per year
* Number of years to project

The method will multiply the previous year’s value by the growth rate recursively until it reaches year 0.

**Implementation:**

public class FinancialForecast {

public static double futureValueRecursive(double initialValue, double growthRate, int years) {

if (years == 0) {

return initialValue;

}

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

}

public static void main(String[] args) {

double initial = 10000.0

double rate = 0.10;

int years = 5;

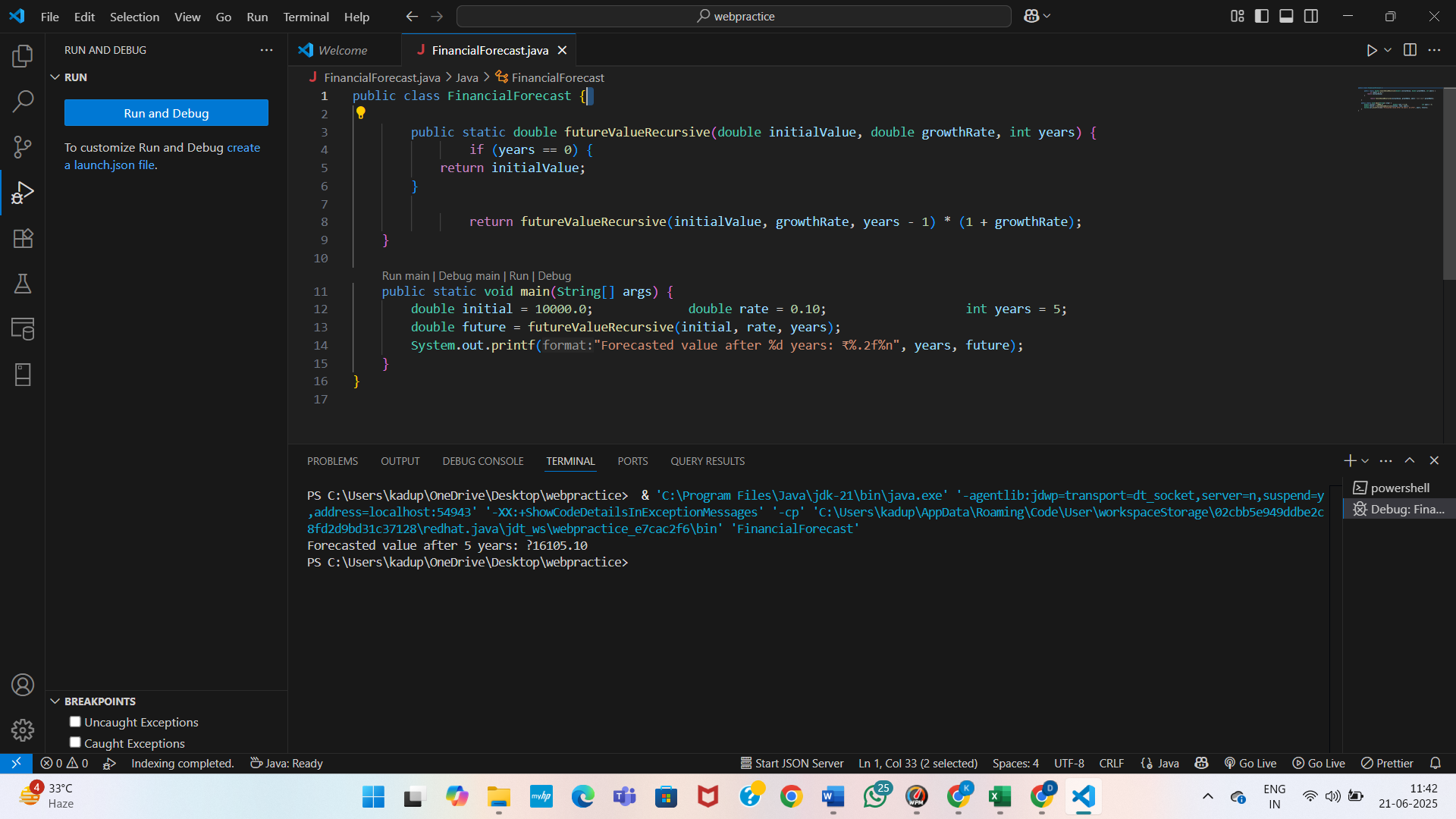
double future = futureValueRecursive(initial, rate, years);

System.out.printf("Forecasted value after %d years: ₹%.2f%n", years, future);

}

}

**Output:**

****

**Time Complexity:**

The recursive algorithm calls itself once for each year, reducing the problem size by one in every call. Hence, the time complexity is:

* **O(n)** — where *n* is the number of years

Each call performs a constant-time operation (multiplication), and there are *n* calls in total until the base case is reached.

**Optimization Suggestions**

Though the algorithm is simple and efficient for small n, it can become inefficient if the same values are recomputed multiple times or if other variables are added. Optimization techniques include:

* **Memorization**: Store already computed results in an array or map.
* **Iteration**: Replace recursion with a loop (especially for large n to avoid stack overflow).
* **Tail Recursion**: If the language or compiler supports it, use tail-recursive techniques to save stack memory.
* Recursive algorithms make financial forecasting logic cleaner and easier to understand, especially when projecting values that depend on prior results. However, for larger datasets or real-world applications, iterative approaches or memorization should be used to optimize performance and memory usage.