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

**Understanding Asymptotic Notation**

**Big O Notation:**

Big O notation describes the upper bound of an algorithm's running time or space requirement in terms of input size n. It helps analyze scalability and performance.

| **Notation** | **Meaning** |
| --- | --- |
| O(1) | Constant time |
| O(n) | Linear time |
| O(log n) | Logarithmic time |
| O(n²) | Quadratic time |

**Best, Average, and Worst Cases for Searching:**

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| Linear Search | O(1) (first item) | O(n) | O(n) (last/missing) |
| Binary Search | O(1) (middle) | O(log n) | O(log n) |

1) Use **Linear Search** for small, dynamic, or unsorted product arrays.

2) Use **Binary Search** for large, static, sorted product lists (ideal in e-commerce catalogs).

Code:

using System;

*public* class Product

{

*public* int ProductId { get; set; }

*public* string ProductName { get; set; }

*public* string Category { get; set; }

*public* Product(int productId, string productName, string category)

    {

        ProductId = productId;

        ProductName = productName;

        Category = category;

    }

*public* *override* string ToString()

    {

        return $"{ProductId} - {ProductName} ({Category})";

    }

}

*public* class Program

{

*public* *static* Product LinearSearch(Product[] products, string targetName)

    {

        foreach (var product in products)

        {

            if (product.ProductName.Equals(targetName, StringComparison.OrdinalIgnoreCase))

            {

                return product;

            }

        }

        return null;

    }

*public* *static* void SortProductsByName(Product[] products)

    {

        Array.Sort(products, (p1, p2) => string.Compare(p1.ProductName, p2.ProductName, true));

    }

*public* *static* Product BinarySearch(Product[] products, string targetName)

    {

        int left = 0;

        int right = products.Length - 1;

        while (left <= right)

        {

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

            int comparison = string.Compare(targetName, products[mid].ProductName, true);

            if (comparison == 0)

            {

                return products[mid];

            }

            else if (comparison < 0)

            {

                right = mid - 1;

            }

            else

            {

                left = mid + 1;

            }

        }

        return null;

    }

*public* *static* void Main()

    {

        Product[] products = new Product[]

        {

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

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

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

            new Product(104, "T-Shirt", "Fashion"),

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

        };

        var resultLinear = LinearSearch(products, "Camera");

        Console.WriteLine("Linear Search Result: " + (resultLinear != null ? resultLinear.ToString() : "Not Found"));

        SortProductsByName(products);

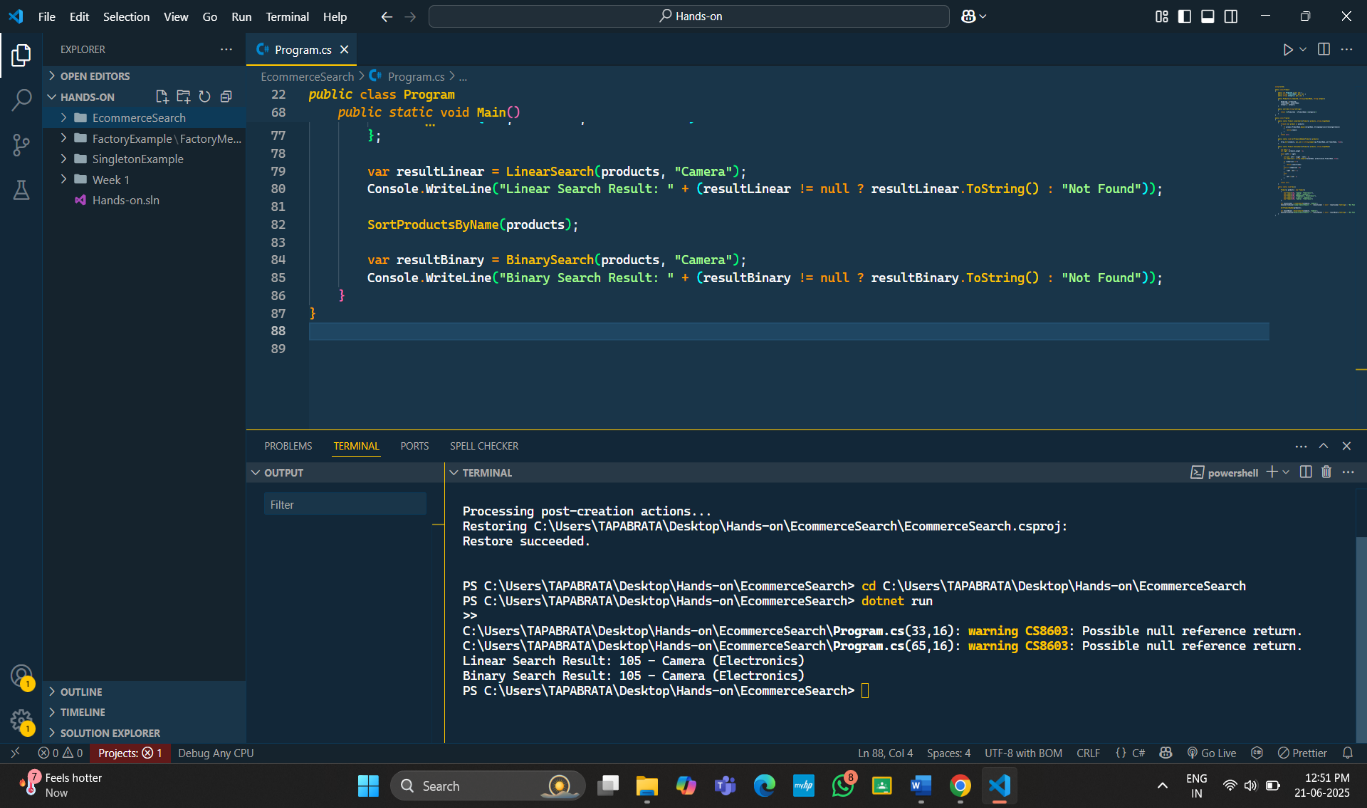
        var resultBinary = BinarySearch(products, "Camera");

        Console.WriteLine("Binary Search Result: " + (resultBinary != null ? resultBinary.ToString() : "Not Found"));

    }

}

Output:



**Exercise 7: Financial Forecasting**

Recursion:

**Recursion** is a programming technique where a function calls itself to solve a smaller subproblem.

It simplifies problems like **factorials, Fibonacci sequences, tree traversals, and forecasting by breaking them down into smaller,** repeated operations.

Code:

using System;

public class FinancialForecast

{

public static double ForecastRecursive(double currentValue, double growthRate, int years)

{

if (years == 0)

return currentValue;

double nextYearValue = ForecastRecursive(currentValue, growthRate, years - 1);

return nextYearValue \* (1 + growthRate);

}

public static void Main()

{

double initialValue = 1000;

double annualGrowthRate = 0.08;

int forecastYears = 5;

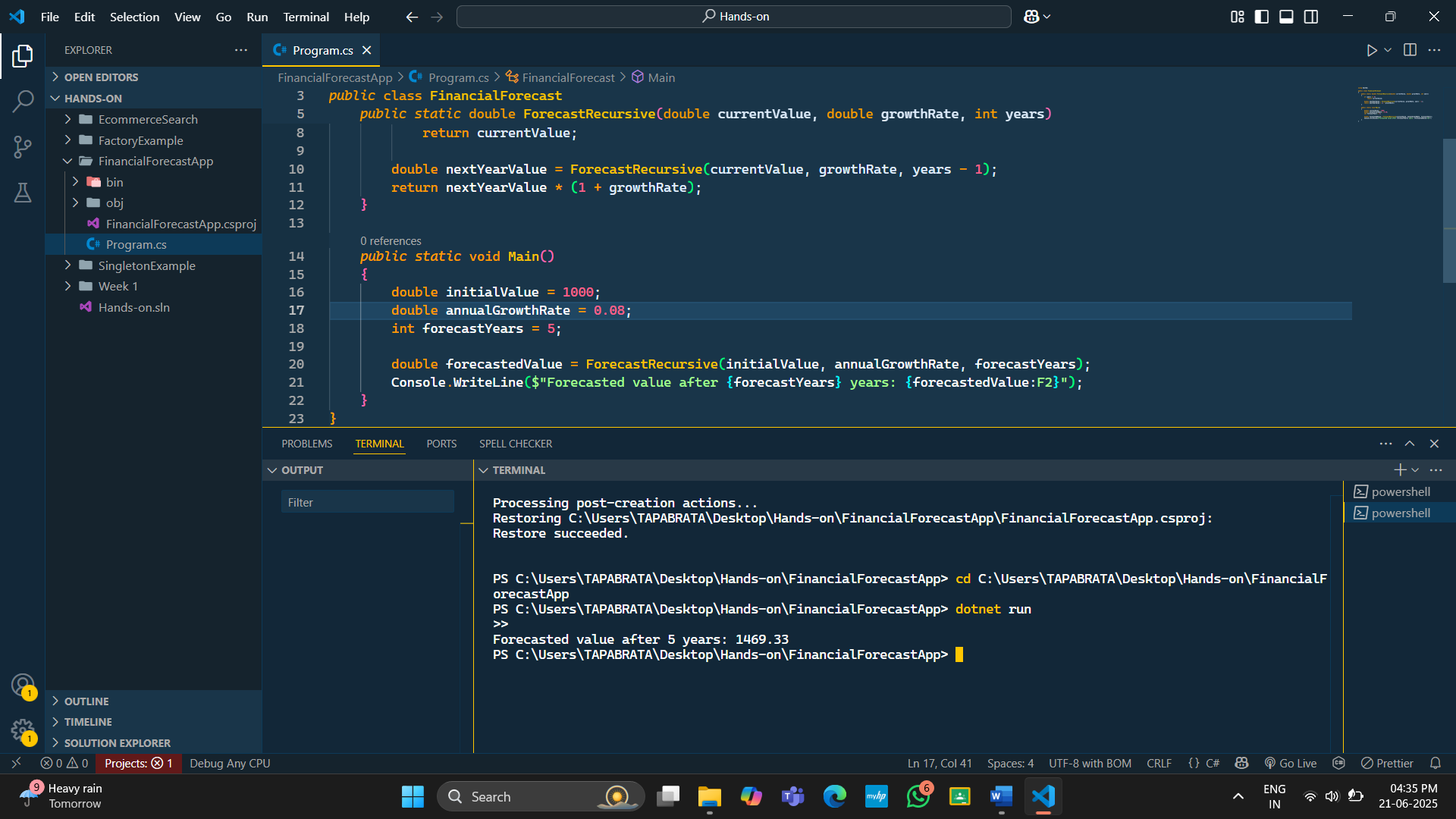
double forecastedValue = ForecastRecursive(initialValue, annualGrowthRate, forecastYears);

Console.WriteLine($"Forecasted value after {forecastYears} years: {forecastedValue:F2}");

}

}

Output:



Time Complexity

The recursive method has **O(n)** time complexity, where n is the number of years.

How to Optimize :

1. Use Iteration Instead of Recursion
2. Use Memoization