Devjit Mondal  
Superset ID **:-** 6363230

**Design Patterns and Principles**

**Exercise 1: Implementing the Singleton Pattern**

**Scenario:**

You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

1. **Creating Logger.cs**:

using System;

public class Logger

{

    private static Logger \_instance;

    private static readonly object \_lock = new object();

    private Logger()

    {

        Console.WriteLine("Logger instance created.");

    }

    public static Logger Instance

    {

        get

        {

            if (\_instance == null)

            {

                lock (\_lock)

                {

                    if (\_instance == null)

                    {

                        \_instance = new Logger();

                    }

                }

            }

            return \_instance;

        }

    }

    public void Log(string message)

    {

        Console.WriteLine($"[LOG] {message}");

    }

}

1. **Program.cs**

using System;

class Program

{

    static void Main(string[] args)

    {

        Logger logger1 = Logger.Instance;

        Logger logger2 = Logger.Instance;

        logger1.Log("First log message.");

        logger2.Log("Second log message.");

        if (object.ReferenceEquals(logger1, logger2))

        {

            Console.WriteLine("Both logger1 and logger2 refer to the same instance.");

        }

        else

        {

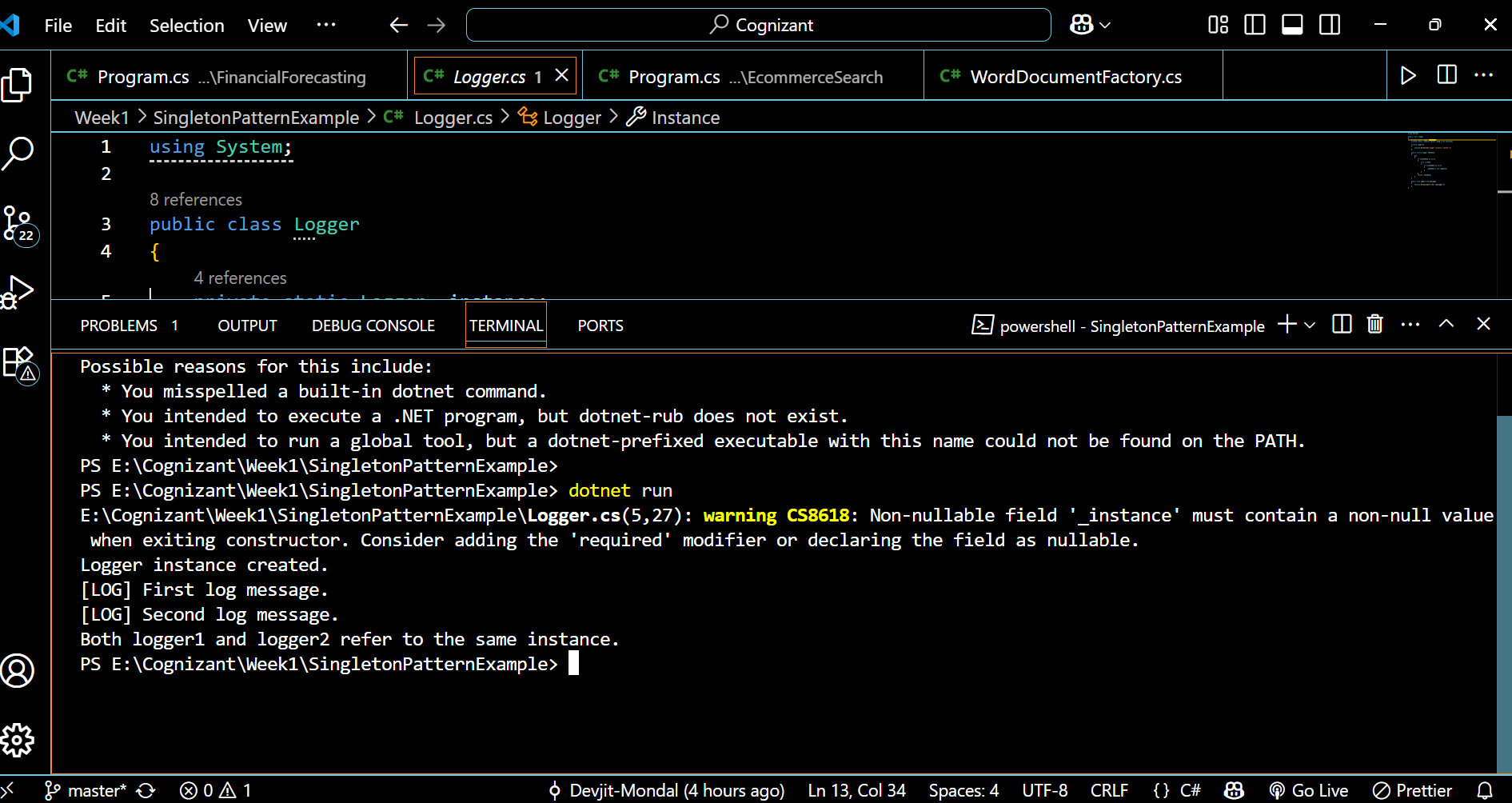
            Console.WriteLine("logger1 and logger2 are different instances.");

        }

    }

}

3.**Output**



**Exercise 2: Implementing the Factory Method Pattern**

**Scenario:**

You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.

1. **Creating IDocument.cs**

namespace FactoryMethodPatternExample

{

    public interface IDocument

    {

        void Open();

    }

}

## Creating Concrete Document Classes

**WordDocument.cs**

using System;

namespace FactoryMethodPatternExample

{

    public class WordDocument : IDocument

    {

        public void Open()

        {

            Console.WriteLine("Opening Word document...");

        }

    }

}

**PdfDocument.cs**

using System;

namespace FactoryMethodPatternExample

{

    public class PdfDocument : IDocument

    {

        public void Open()

        {

            Console.WriteLine("Opening PDF document...");

        }

    }

}

**ExcelDocument.cs**

using System;

namespace FactoryMethodPatternExample

{

    public class ExcelDocument : IDocument

    {

        public void Open()

        {

            Console.WriteLine("Opening Excel document...");

        }

    }

}

## Implement the Factory Method(Abstract Factory)

**DocumentFactory.cs**

namespace FactoryMethodPatternExample

{

    public abstract class DocumentFactory

    {

        public abstract IDocument CreateDocument();

    }

}

## Concrete Factories

**WordDocumentFactory.cs**

namespace FactoryMethodPatternExample

{

    public class WordDocumentFactory : DocumentFactory

    {

        public override IDocument CreateDocument()

        {

            return new WordDocument();

        }

    }

}

**PdfDocumentFactory.cs**

namespace FactoryMethodPatternExample

{

    public class PdfDocumentFactory : DocumentFactory

    {

        public override IDocument CreateDocument()

        {

            return new PdfDocument();

        }

    }

}

**ExcelDocumentFactory.cs**

namespace FactoryMethodPatternExample

{

    public class ExcelDocumentFactory : DocumentFactory

    {

        public override IDocument CreateDocument()

        {

            return new ExcelDocument();

        }

    }

}

1. **Program.cs**

using System;

using FactoryMethodPatternExample;

class Program

{

    static void Main(string[] args)

    {

        DocumentFactory wordFactory = new WordDocumentFactory();

        IDocument wordDoc = wordFactory.CreateDocument();

        wordDoc.Open();

        DocumentFactory pdfFactory = new PdfDocumentFactory();

        IDocument pdfDoc = pdfFactory.CreateDocument();

        pdfDoc.Open();

        DocumentFactory excelFactory = new ExcelDocumentFactory();

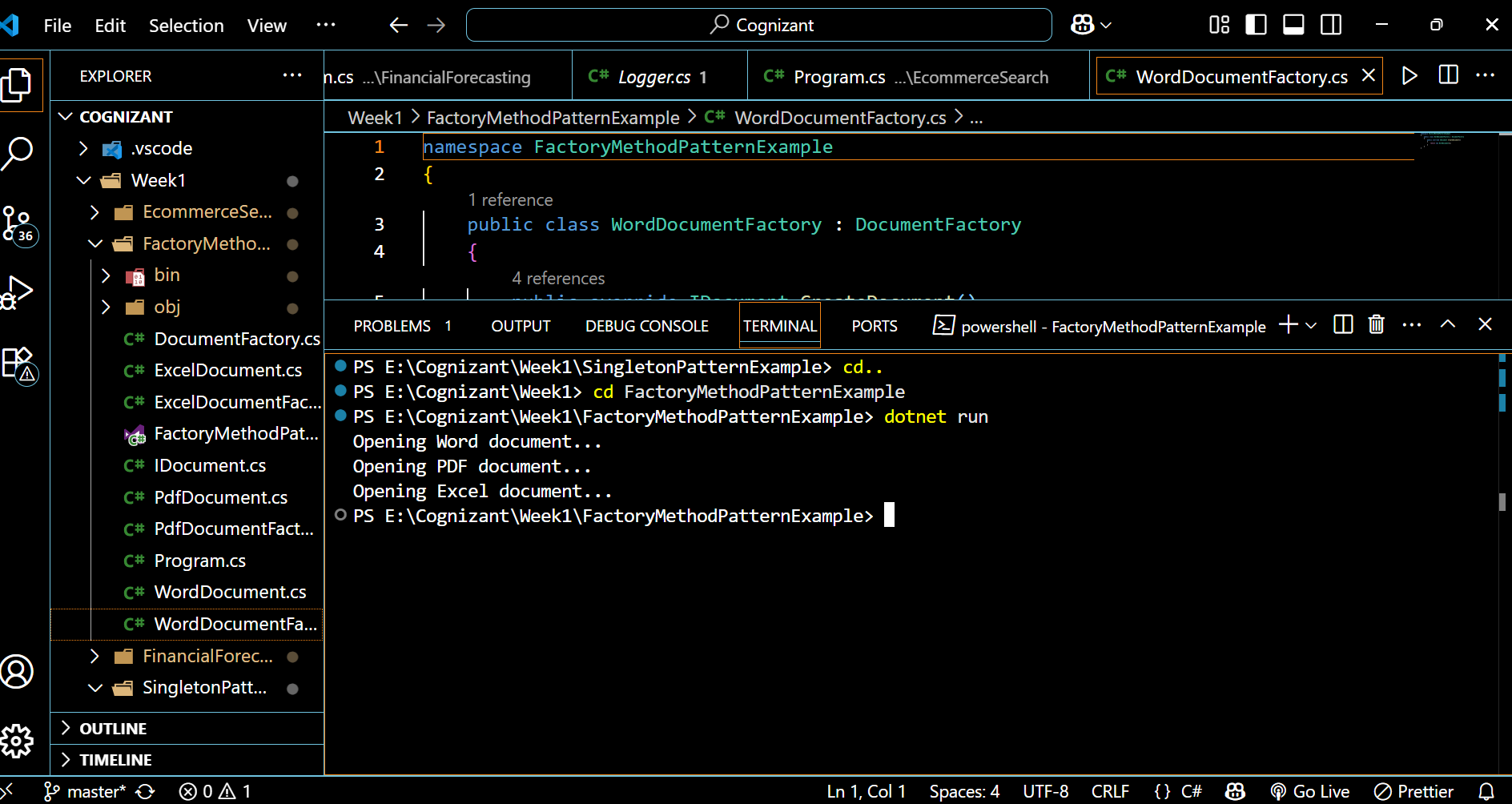
        IDocument excelDoc = excelFactory.CreateDocument();

        excelDoc.Open();

    }

}

1. **Output**

****

**Algorithms Data Structures**

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

**Scenario:**

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

## **1**. Understanding Asymptotic Notation (Big O)****

**Big O notation** describes how an algorithm’s runtime or space requirements grow as the input size increases. It allows you to compare algorithms independently of hardware or language, focusing on how they scale

* **O(1):** Constant time—operation takes the same time regardless of input size.
* **O(n):** Linear time—operation time grows proportionally with input size (e.g., linear search)
* **O(log n):** Logarithmic time—operation time grows slowly as input increases (e.g., binary search)
* **O(n^2):** Quadratic time—operation time grows with the square of input size.

**Best, Average, and Worst Case:**

**Best case:** Minimum steps needed (e.g., first element is the target in linear search)

**Average case:** Expected steps for a random input.

**Worst case:** Maximum steps needed (e.g., target is last or not present)

In real-world systems, **worst-case and average-case** are most important for performance guarantees

2**.Creating SearchServices.cs**

using System;

namespace EcommerceSearch

{

    public static class SearchService

    {

        public static Product LinearSearch(Product[] products, int id)

        {

            foreach (var product in products)

            {

                if (product.ProductId == id)

                    return product;

            }

            return null;

        }

        public static Product BinarySearch(Product[] sortedProducts, int id)

        {

            int left = 0;

            int right = sortedProducts.Length - 1;

            while (left <= right)

            {

                int mid = (left + right) / 2;

                if (sortedProducts[mid].ProductId == id)

                    return sortedProducts[mid];

                else if (sortedProducts[mid].ProductId < id)

                    left = mid + 1;

                else

                    right = mid - 1;

            }

            return null;

        }

    }

}

1. **Creating Product.cs**

namespace EcommerceSearch

{

    public class Product

    {

        public int ProductId { get; set; }

        public string ProductName { get; set; }

        public string Category { get; set; }

        public Product(int id, string name, string category)

        {

            ProductId = id;

            ProductName = name;

            Category = category;

        }

        public override string ToString()

        {

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

        }

    }

}

1. **Program.cs**

using System;

using System.Linq;

namespace EcommerceSearch

{

    class Program

    {

        static void Main()

        {

            Product[] products = new Product[]

            {

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

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

                new Product(104, "Shoes", "Fashion"),

                new Product(102, "Book", "Education")

            };

            var sortedProducts = products.OrderBy(p => p.ProductId).ToArray();

            Console.WriteLine("Searching for Product ID 104 (Linear Search):");

            var resultLinear = SearchService.LinearSearch(products, 104);

            Console.WriteLine(resultLinear != null ? resultLinear.ToString() : "Product not found.");

            Console.WriteLine("\nSearching for Product ID 104 (Binary Search):");

            var resultBinary = SearchService.BinarySearch(sortedProducts, 104);

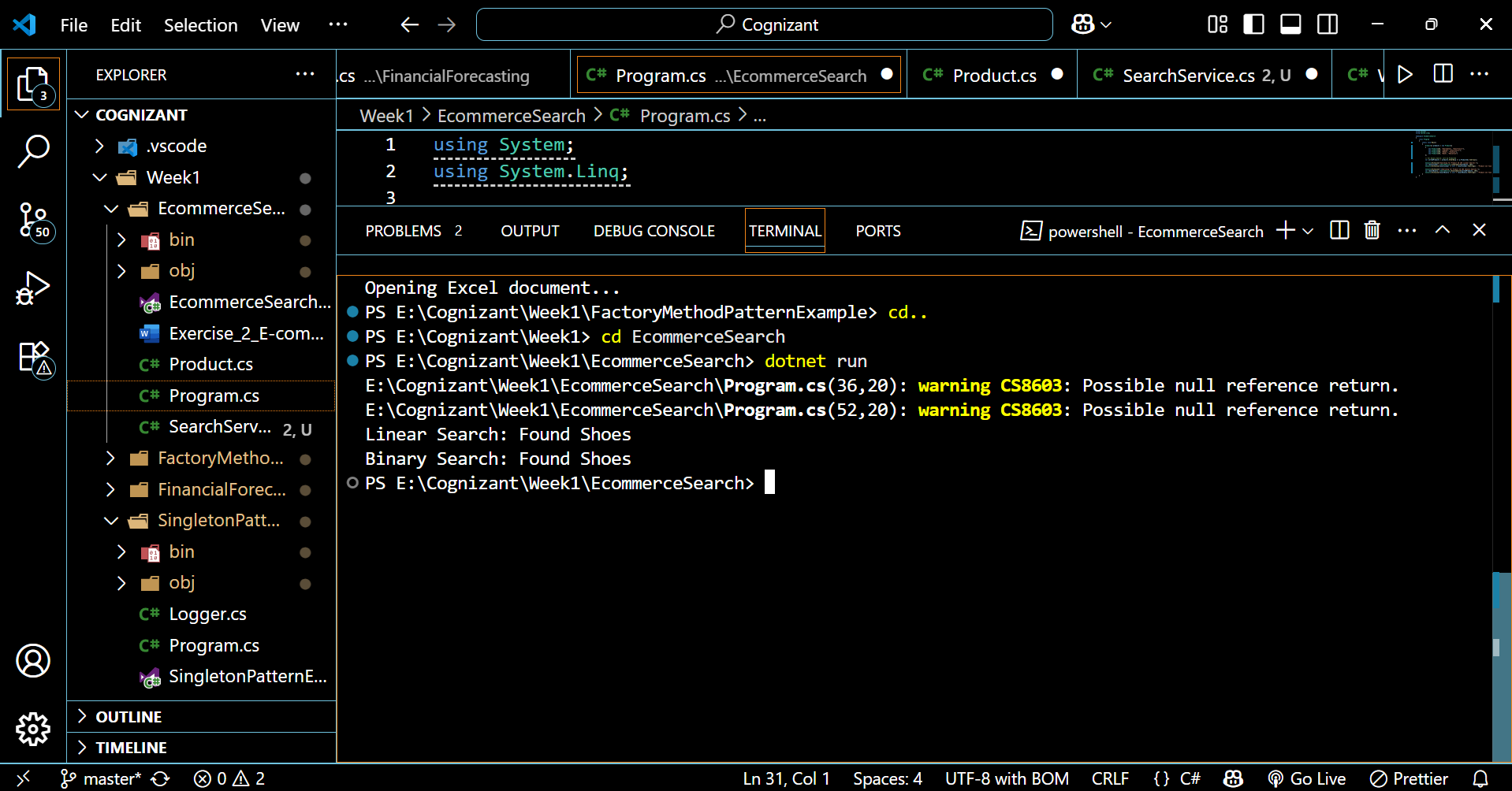
            Console.WriteLine(resultBinary != null ? resultBinary.ToString() : "Product not found.");

        }

    }

}

1. **Output**

****

## **6**. Analysis: Time Complexity and Suitability****

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** | **Time Complexity** | **Suitable For** |
| Linear Search | O(1) | O(n) | O(n) | Linear | Small/unsorted arrays |
| Binary Search | O(1) | O(log n) | O(log n) | Logarithmic | Large/sorted arrays |

**Linear Search:**

Simple, works on unsorted data.

Becomes slow as data grows—performance degrades linearly

**Binary Search:**

Much faster for large datasets, but requires sorted data.

Scales well, as each step halves the search space.

**Which to use?**

For an e-commerce platform with large product catalogs, **binary search** is far more efficient, provided you keep your product array sorted by the search key (e.g., ProductId).

For small datasets or when sorting is not feasible, linear search is acceptable.

**Exercise 7: Financial Forecasting**

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

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

**Recursion** is a programming technique where a function calls itself to solve smaller instances of the same problem. This approach can simplify problems that have a natural "smaller subproblem" structure, such as calculating factorials, Fibonacci numbers, or forecasting sequences. Every recursive function must have a base case to prevent infinite calls. Recursion is elegant and concise, but can be less efficient than iteration due to repeated calculations and call stack overhead.

## 2. ****Setup: Recursive Future Value Calculation in C#****

To forecast a future value based on an initial value and a constant growth rate (like compound interest):

FV(n)=FV(n−1)×(1+r)*FV*(*n*)=*FV*(*n*−1)×(1+*r*)

FV(n)*FV*(*n*): future value after n periods

r*r*: growth rate per period

Base case: FV(0)=initial value*FV*(0)=initial value

1. **Program.cs**

using System;

class Program

{

    static double ForecastFutureValue(double initialValue, double growthRate, int periods)

    {

        if (periods == 0)

            return initialValue;

        return ForecastFutureValue(initialValue, growthRate, periods - 1) \* (1 + growthRate);

    }

    static void Main()

    {

        double initial = 1000;

        double rate = 0.05;

        int years = 10;

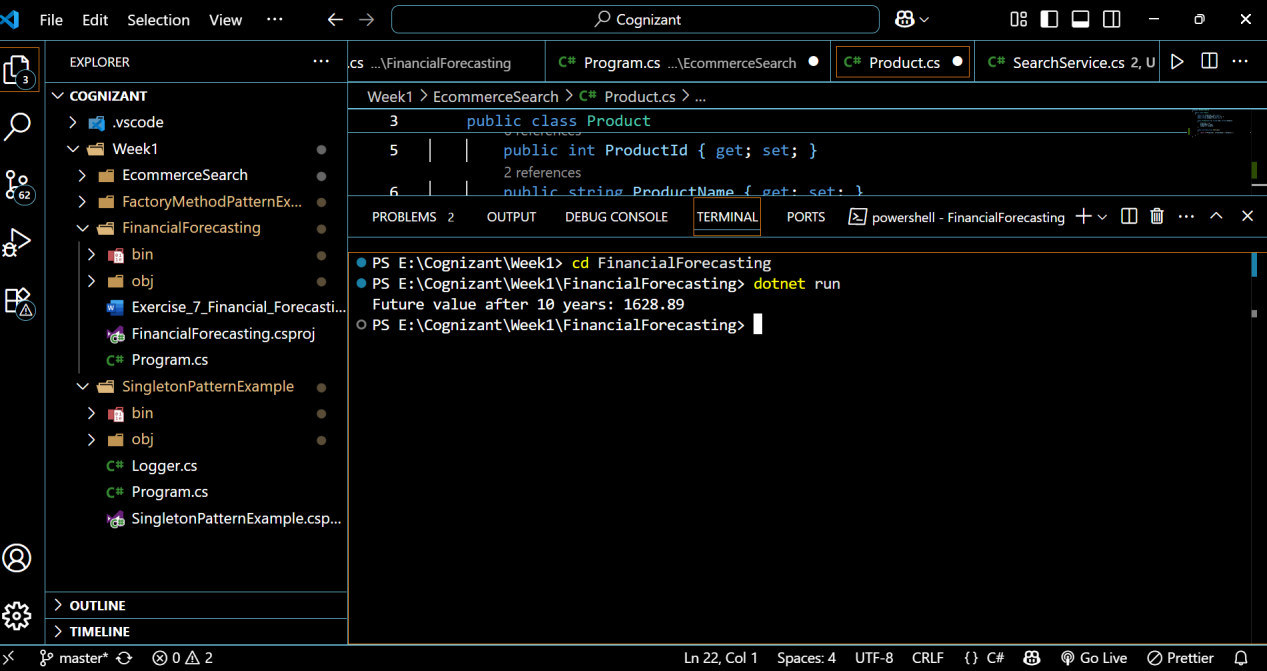
        double futureValue = ForecastFutureValue(initial, rate, years);

        Console.WriteLine($"Future value after {years} years: {futureValue:F2}");

    }

}

1. **Output**



## 5. ****Analysis: Time Complexity and Optimization Time Complexity****

This simple recursion has **O(n)** time complexity, as it makes one recursive call per period.

Each call reduces periods by 1, so for n periods, there are n calls.

## ****Optimizing Recursive Solutions****

* **Memoization:** For problems with overlapping subproblems (like Fibonacci), store results of recursive calls to avoid redundant calculations. For this forecasting example, memoization isn’t needed because each call depends only on the previous period, not multiple branches.
* **Tail Recursion:** If supported by the compiler, tail recursion can reduce stack usage by reusing stack frames.
* **Iterative Approach:** For simple recurrences like this, an iterative loop is often clearer and more efficient, avoiding call stack overhead.
* **Dynamic Programming:** For complex forecasting with overlapping subproblems (e.g., variable rates or constraints), dynamic programming with memoization is beneficial

## ****Summary Table****

|  |  |  |  |
| --- | --- | --- | --- |
| **Approach** | **Time Complexity** | **Space Complexity** | **Notes/Optimization** |
| Simple Recursion | O(n) | O(n) | One call per period, stack grows |
| With Memoization | O(n) | O(n) | Useful for overlapping subproblems |
| Iterative | O(n) | O(1) | Most efficient for this scenario |