**Design patterns and principle**

**Exercise 1: Implementing the Singleton Pattern:**

**CODE:**

using System;

class Program

{

    static void Main(string[] args)

    {

        Logger logger1 = Logger.GetInstance();

        logger1.Log("This is the first log message.");

        Logger logger2 = Logger.GetInstance();

        logger2.Log("This is the second log message.");

        if (ReferenceEquals(logger1, logger2))

        {

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

        }

        else

        {

            Console.WriteLine("Different instances exist. Singleton failed.");

        }

    }

}

using System;

public class Logger

{

    private static Logger? instance;

    private Logger()

    {

        Console.WriteLine("Logger instance created");

    }

  public static Logger GetInstance()

    {

        if (instance == null)

        {

            instance = new Logger();

        }

        return instance;

    }

    public void Log(string message)

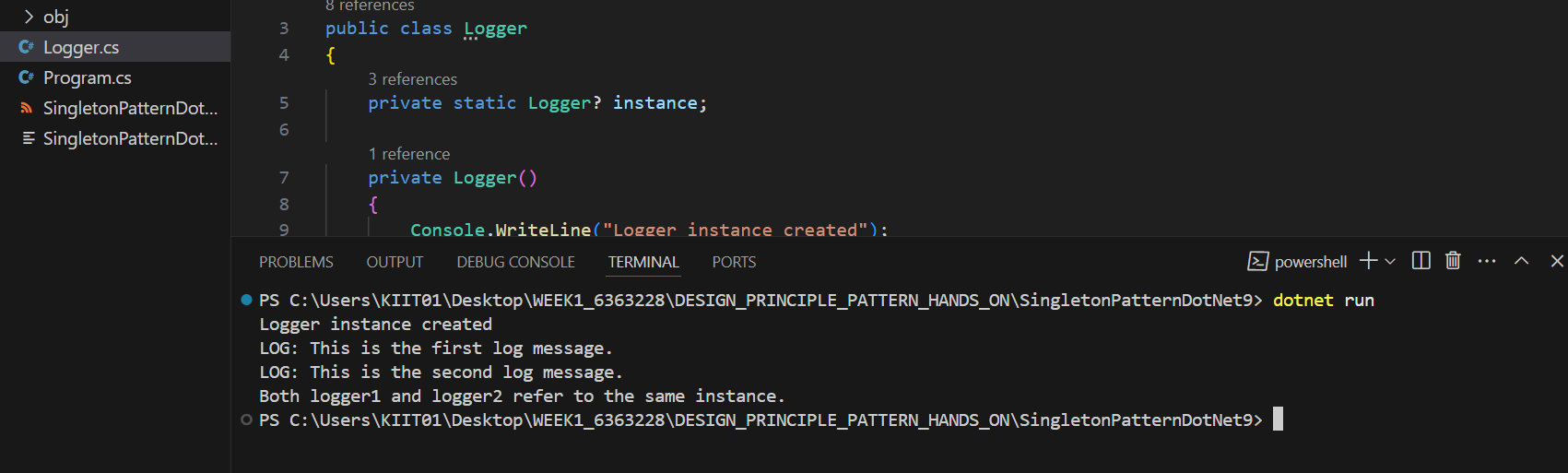
    {

        Console.WriteLine("LOG: " + message);

    }

}

**Output:**

****

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

**Code:**

**Prgm.cs**

using FactoryMethodPatternExample.Factories;

using FactoryMethodPatternExample.Documents;

class Program

{

    static void Main()

    {

        List<DocumentFactory> factories = new List<DocumentFactory>

        {

            new WordDocumentFactory(),

            new PdfDocumentFactory(),

            new ExcelDocumentFactory()

        };

        foreach (var factory in factories)

        {

            IDocument doc = factory.CreateDocument();

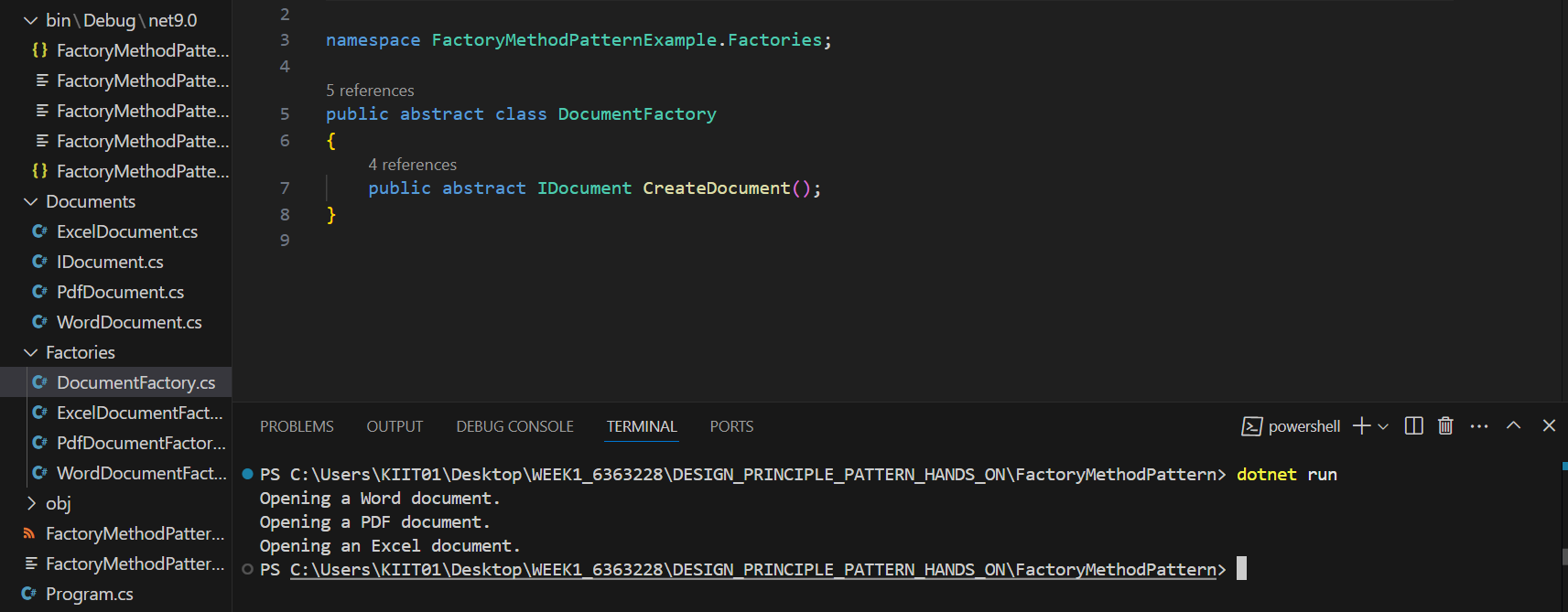
            doc.Open();

        }

    }

}

**Output:**

****

**Data structures and algorithms:**

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

**Code:**

using System;

using System.Diagnostics;

using System.Linq;

class Program

{

    static void Main()

    {

        Console.WriteLine("E-commerce Platform Search Function");

        Console.WriteLine();

        Console.WriteLine("Understanding Asymptotic Notation and Search Algorithms");

        Console.WriteLine();

        var products = new Product[]

        {

            new Product(1, "Laptop", "Electronics", 999.99),

            new Product(2, "Smartphone", "Electronics", 699.99),

            new Product(3, "Headphones", "Electronics", 199.99),

            new Product(4, "T-Shirt", "Clothing", 29.99),

            new Product(5, "Jeans", "Clothing", 49.99),

            new Product(6, "Sneakers", "Footwear", 89.99),

            new Product(7, "Watch", "Accessories", 199.99),

            new Product(8, "Backpack", "Accessories", 59.99),

            new Product(9, "Book", "Books", 15.99),

            new Product(10, "Tablet", "Electronics", 349.99)

        };

        Console.WriteLine("Product List:");

        foreach (var product in products)

        {

            Console.WriteLine(product);

        }

        Console.WriteLine();

        Console.WriteLine("=== Linear Search Demo ===");

        Console.Write("Enter a product ID to search: ");

        if (!int.TryParse(Console.ReadLine(), out int targetId))

        {

            Console.WriteLine("Invalid input.");

            return;

        }

        var stopwatch = Stopwatch.StartNew();

        var linearResult = LinearSearch(products, targetId);

        stopwatch.Stop();

        if (linearResult != null)

            Console.WriteLine("Linear Search Result: " + linearResult);

        else

            Console.WriteLine("Product not found.");

        Console.WriteLine($"Linear search took {stopwatch.ElapsedTicks} ticks");

        Console.WriteLine();

        Console.WriteLine("=== Binary Search Demo ===");

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

        stopwatch.Restart();

        var binaryResult = BinarySearch(sortedProducts, targetId);

        stopwatch.Stop();

      if (binaryResult != null)

            Console.WriteLine("Binary Search Result: " + binaryResult);

        else

            Console.WriteLine("Product not found.");

        Console.WriteLine($"Binary search took {stopwatch.ElapsedTicks} ticks");

        Console.WriteLine();

        Console.WriteLine("=== Time Complexity Analysis ===");

        Console.WriteLine("Linear Search: O(n) - Must check each element in the worst case");

        Console.WriteLine("Binary Search: O(log n) - Divides search space in half each time");

        Console.WriteLine();

        Console.WriteLine("Binary search is more efficient for large datasets, but requires sorted data.");

        Console.WriteLine("Linear search works on unsorted data and is simple to implement.");

        Console.WriteLine("For small datasets, the difference in performance is negligible.");

        Console.WriteLine("For our e-commerce platform with potentially millions of products,");

        Console.WriteLine("binary search would be more suitable for ID-based searches on indexed fields.");

    }

    static Product? LinearSearch(Product[] products, int targetId)

    {

        foreach (var product in products)

        {

            if (product.ProductId == targetId)

                return product;

        }

        return null;

    }

    static Product? BinarySearch(Product[] products, int targetId)

    {

        int left = 0;

        int right = products.Length - 1;

        while (left <= right)

        {

            int mid = (left + right) / 2;

            if (products[mid].ProductId == targetId)

                return products[mid];

            else if (products[mid].ProductId < targetId)

                left = mid + 1;

            else

                right = mid - 1;

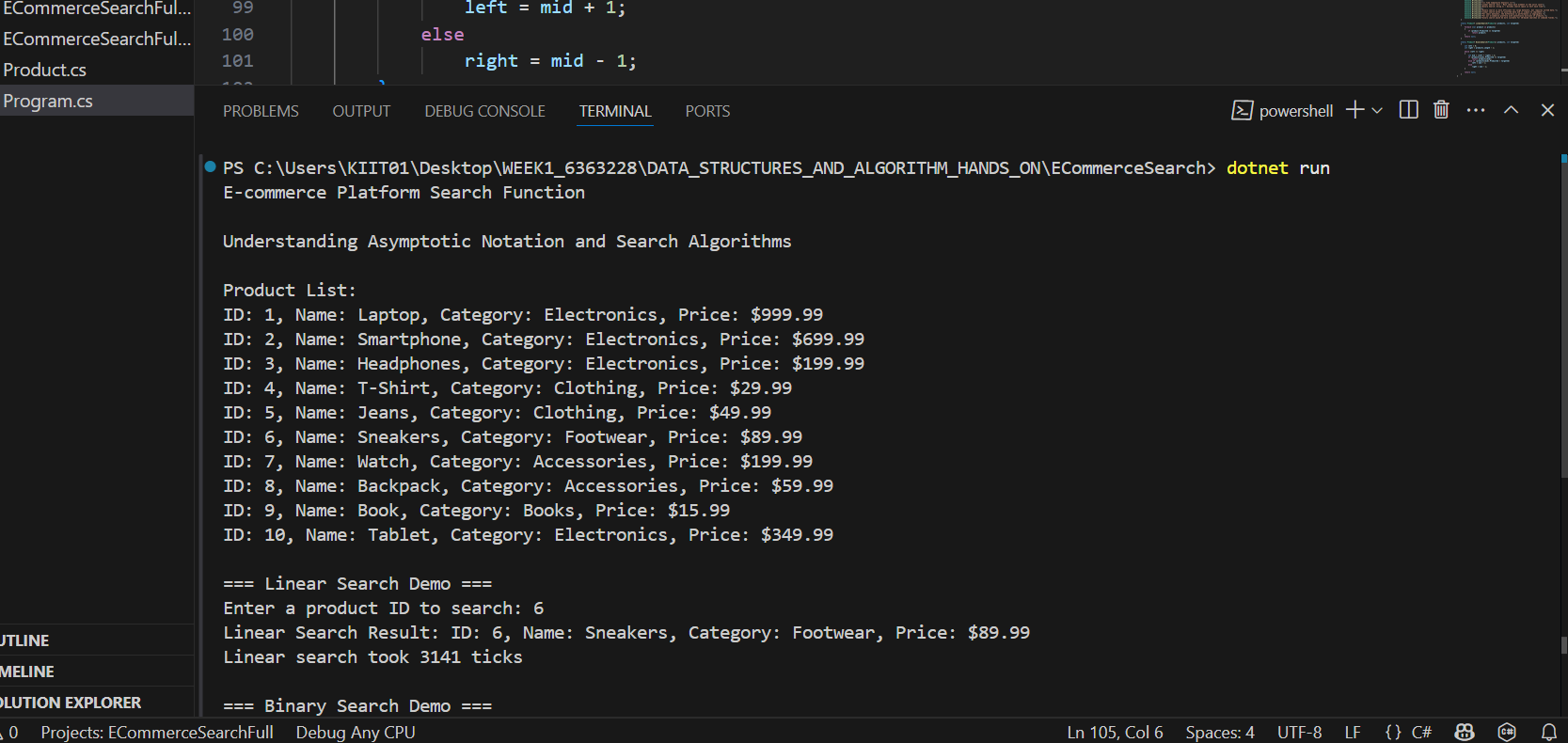
        }

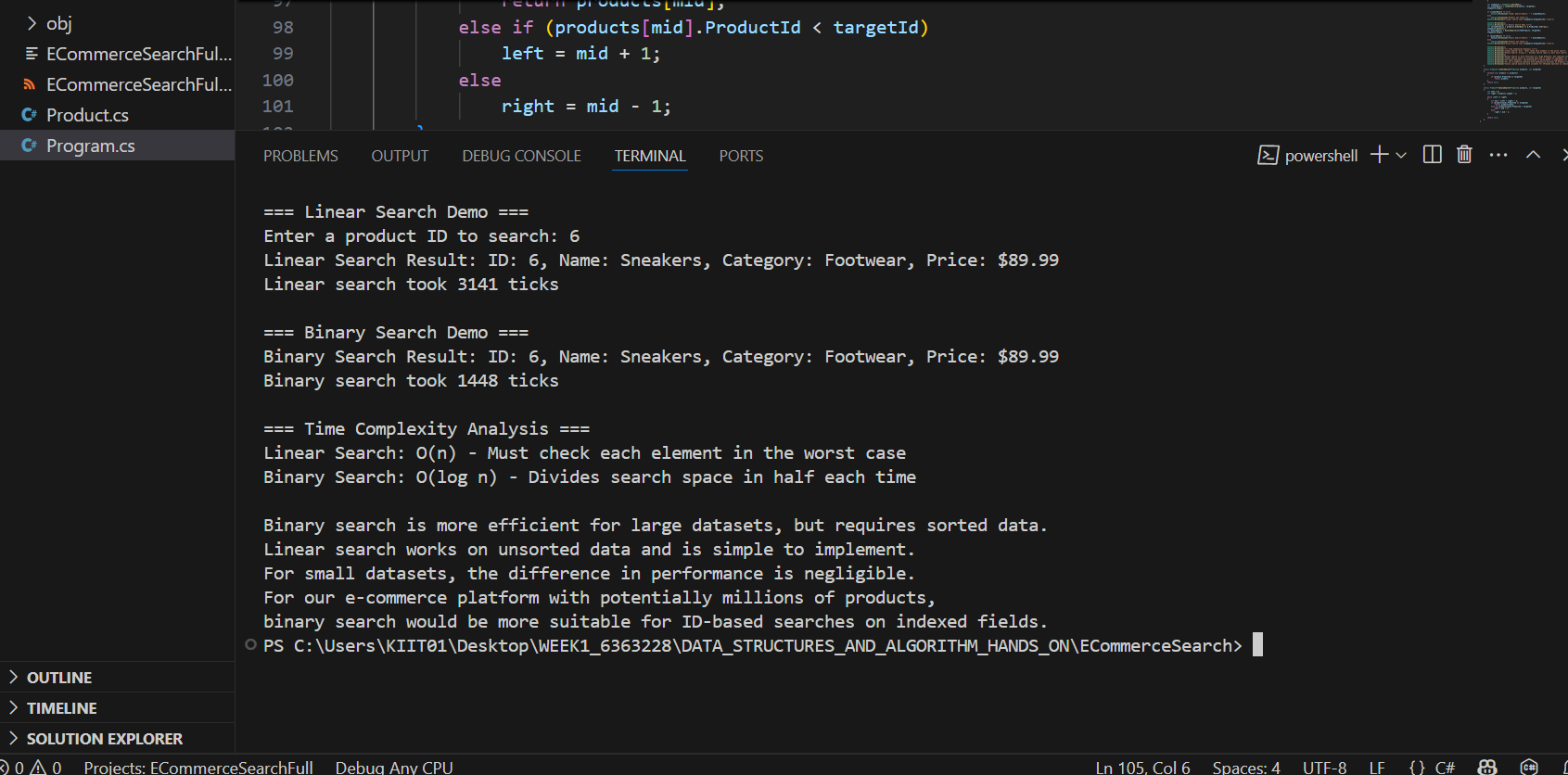
        return null;

    }

}

**Output:**

****

****

**Exercise 7: Financial Forecasting**

**Code:**

using System;

using System.Diagnostics;

class Program

{

    static void Main()

    {

        Console.WriteLine("Financial Forecasting Tool");

        Console.WriteLine("===========================");

        Console.WriteLine();

        double initialValue = 10000.0;

        double growthRate = 0.05;

        int years = 10;

        double monthlyInvestment = 100.0;

        Console.WriteLine($"Initial Investment: ${initialValue:F2}");

        Console.WriteLine($"Annual Growth Rate: {growthRate \* 100:F2}%");

        Console.WriteLine($"Forecast Period: {years} years");

        Console.WriteLine();

        Console.WriteLine("Calculating future values using different methods...");

        Console.WriteLine();

        // Simple recursive

        Stopwatch sw = Stopwatch.StartNew();

        double value1 = ForecastRecursive(initialValue, growthRate, years);

        sw.Stop();

        Console.WriteLine("Simple Recursive Method:");

        Console.WriteLine($"Future Value after {years} years: ${value1:F2}");

        Console.WriteLine($"Calculation time: {sw.ElapsedTicks} ticks");

        Console.WriteLine();

        // Memoized recursive

        sw.Restart();

        double[] memo = new double[years + 1];

        double value2 = ForecastMemoized(initialValue, growthRate, years, memo);

        sw.Stop();

        Console.WriteLine("Optimized Recursive Method (with memoization):");

        Console.WriteLine($"Future Value after {years} years: ${value2:F2}");

        Console.WriteLine($"Calculation time: {sw.ElapsedTicks} ticks");

        Console.WriteLine();

        // Iterative

        sw.Restart();

        double value3 = ForecastIterative(initialValue, growthRate, years);

        sw.Stop();

        Console.WriteLine("Iterative Method:");

        Console.WriteLine($"Future Value after {years} years: ${value3:F2}");

        Console.WriteLine($"Calculation time: {sw.ElapsedTicks} ticks");

        Console.WriteLine();

        // Variable growth rate

        double[] rates = {0.05,0.045,0.048,0.052,0.049,0.051,0.047,0.053,0.05,0.05};

        double varGrowth = VariableGrowth(initialValue, rates);

        Console.WriteLine("Variable Growth Rate Method:");

        Console.WriteLine($"Future Value with variable growth rates: ${varGrowth:F2}");

        Console.WriteLine();

        // With Monthly Investments

        Console.WriteLine($"Method with Monthly Investments (${monthlyInvestment:F2}/month):");

        double compounded = initialValue;

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

        {

            compounded \*= (1 + growthRate);

            compounded += monthlyInvestment \* 12;

        }

        Console.WriteLine($"Future Value after {years} years: ${compounded:F2}");

        Console.WriteLine();

        Console.WriteLine("=== Time Complexity Analysis ===");

        Console.WriteLine("Simple Recursive Method: O(n) - Linear time complexity");

        Console.WriteLine(" - Each call depends on the result of the previous call");

        Console.WriteLine(" - Creates a call stack of depth n (periods)");

        Console.WriteLine(" - Risk of stack overflow for large n");

        Console.WriteLine();

        Console.WriteLine("Memoized Recursive Method: O(n) - Linear time complexity with space optimization");

        Console.WriteLine(" - Avoids redundant calculations by storing results");

        Console.WriteLine(" - Uses additional O(n) memory for memoization table");

        Console.WriteLine(" - Still creates a call stack but avoids recalculation");

        Console.WriteLine();

        Console.WriteLine("Iterative Method: O(n) - Linear time complexity");

        Console.WriteLine(" - Same computational complexity as recursive methods");

        Console.WriteLine(" - Constant space complexity (no call stack or memo table)");

        Console.WriteLine(" - Generally more efficient in practice");

        Console.WriteLine();

        Console.WriteLine("Recommendation for optimizing recursive solutions:");

        Console.WriteLine("1. Use memoization to avoid redundant calculations");

        Console.WriteLine("2. Consider tail recursion when applicable");

        Console.WriteLine("3. For simple growth formulas, iterative solutions may be more efficient");

        Console.WriteLine("4. For complex models with variable inputs, recursion offers more flexibility");

    }

    static double ForecastRecursive(double value, double rate, int years)

    {

        if (years == 0) return value;

        return ForecastRecursive(value \* (1 + rate), rate, years - 1);

    }

    static double ForecastMemoized(double value, double rate, int years, double[] memo)

    {

        if (years == 0) return value;

        if (memo[years] != 0) return memo[years];

        memo[years] = ForecastMemoized(value \* (1 + rate), rate, years - 1, memo);

        return memo[years];

    }

    static double ForecastIterative(double value, double rate, int years)

{

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

        {

            value \*= (1 + rate);

        }

        return value;

    }

    static double VariableGrowth(double value, double[] rates)

    {

        foreach (var r in rates)

        {

            value \*= (1 + r);

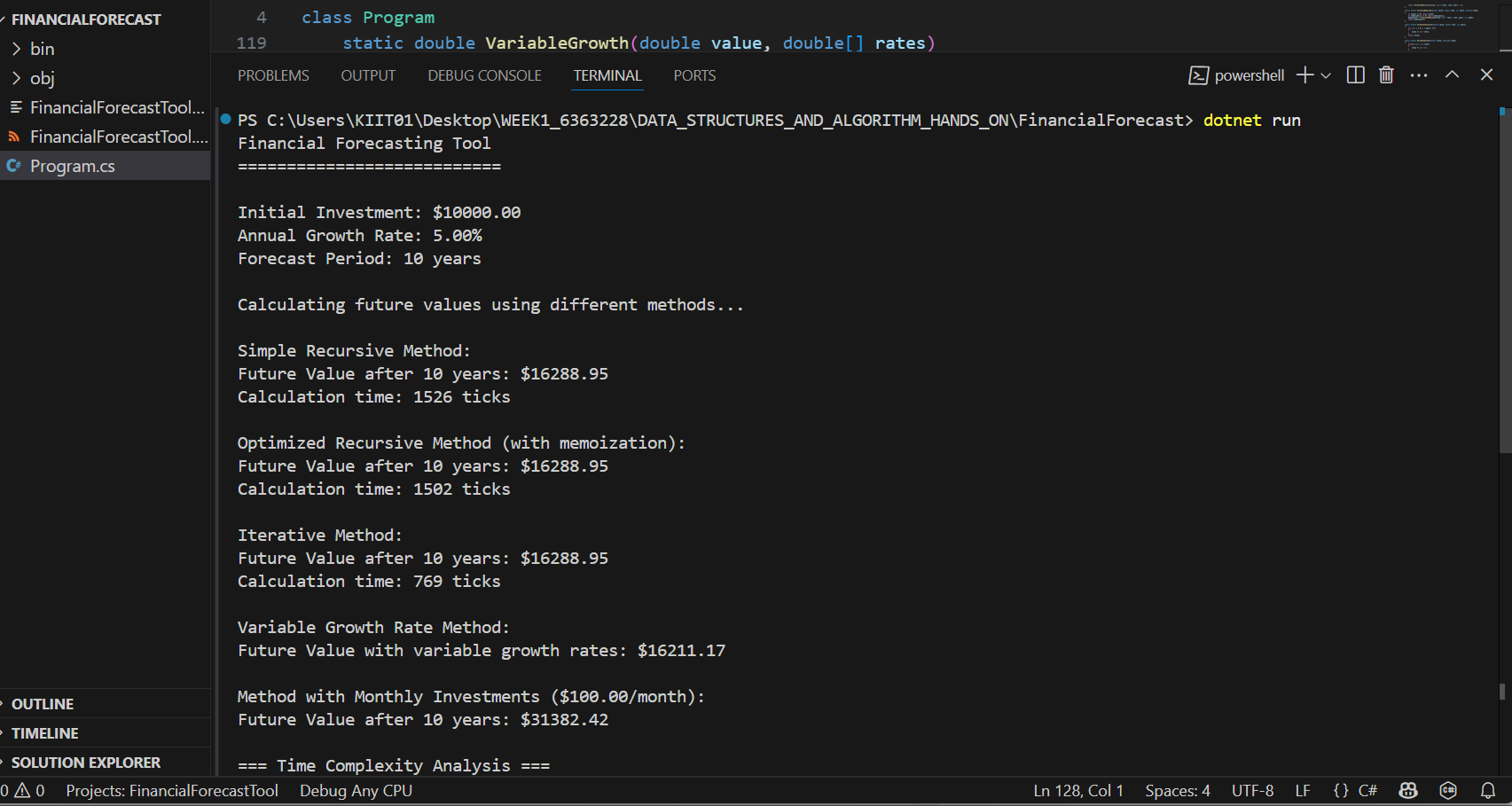
        }

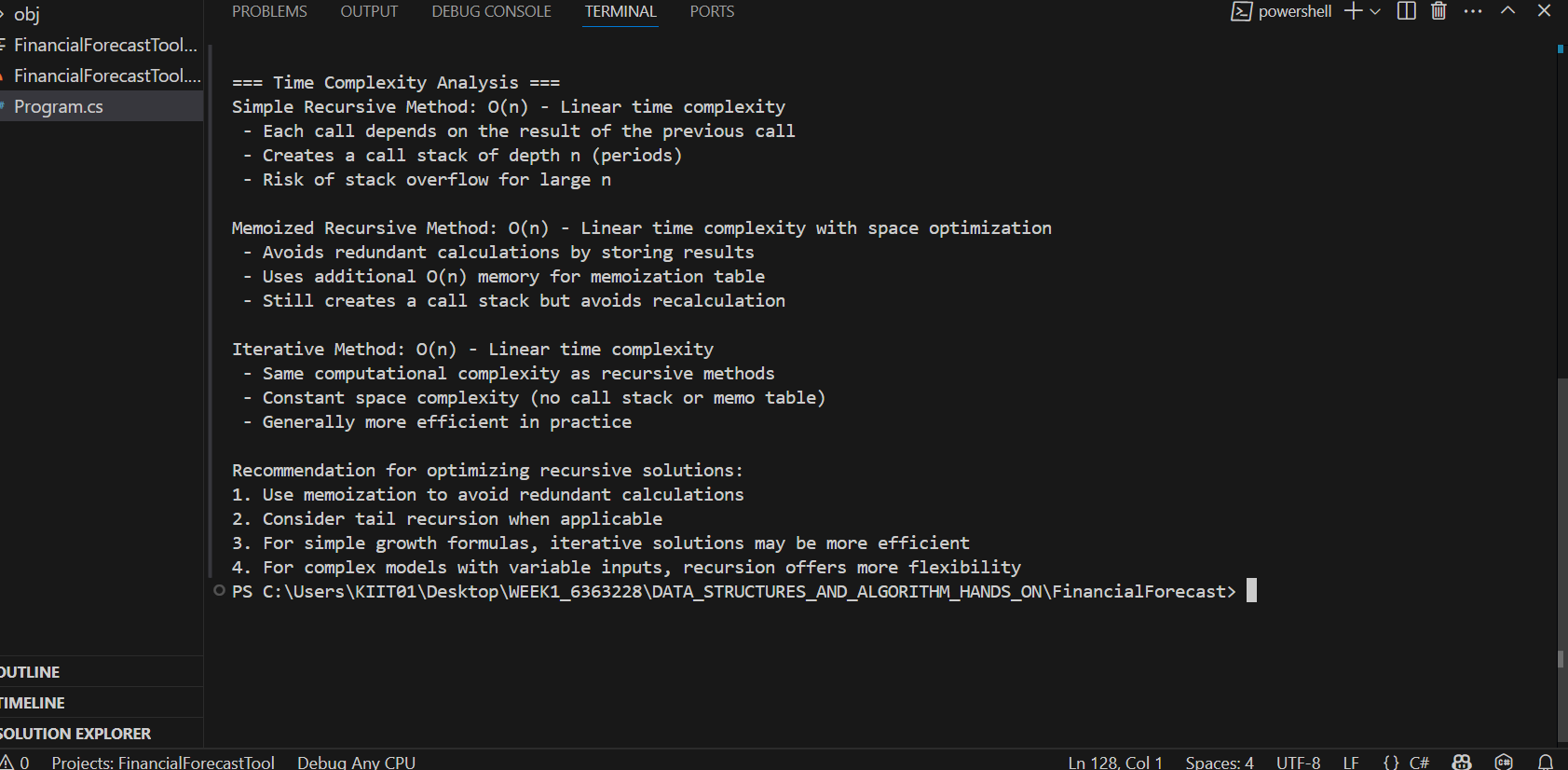
        return value;

    }

}

**Output:**

****

****