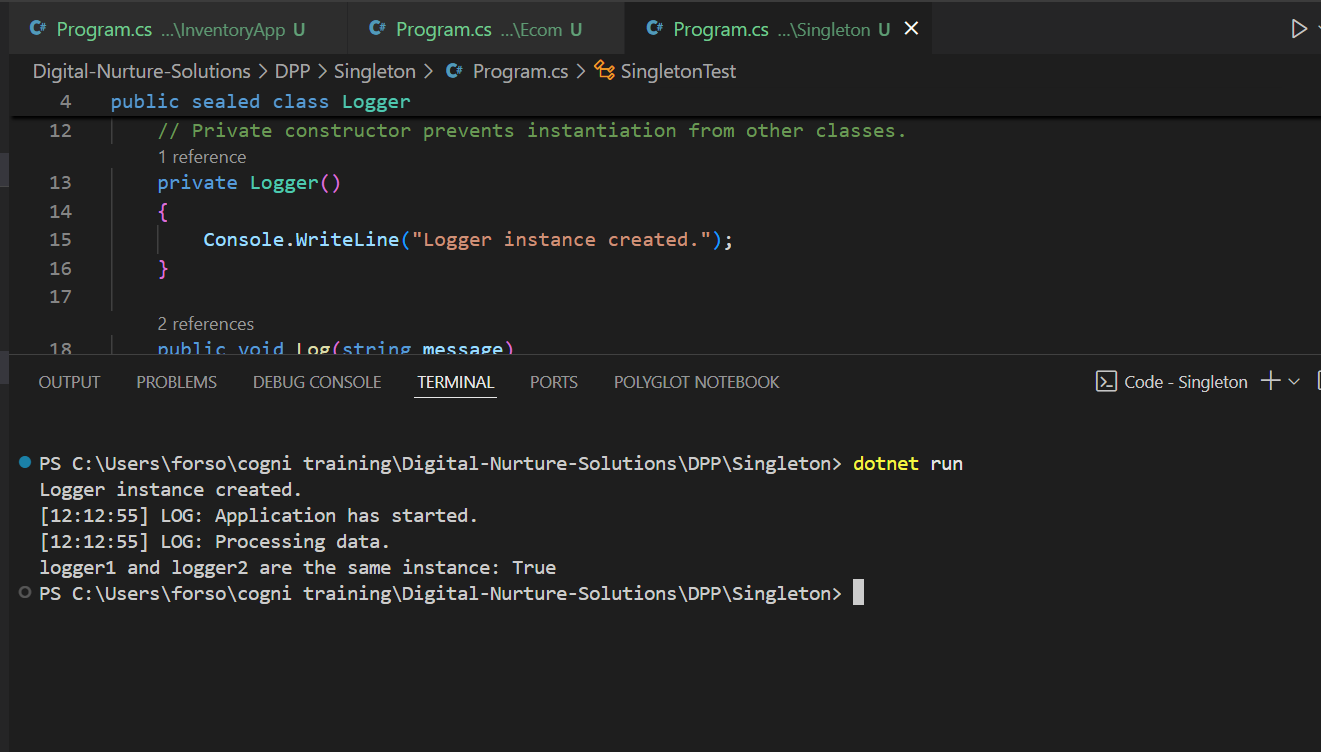
**Exercise 1: Implementing the Singleton Pattern**

**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.

**OUTPUT:**

****

**CODE:**

using System;

public sealed class Logger

{

    private static readonly Lazy<Logger> lazyInstance = new Lazy<Logger>(() => new Logger());

    public static Logger Instance => lazyInstance.Value;

    private Logger()

    {

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

    }

    public void Log(string message)

    {

        Console.WriteLine($"[{DateTime.Now:HH:mm:ss}] LOG: {message}");

    }

}

public class SingletonTest

{

    public static void Main(string[] args)

    {

        Logger logger1 = Logger.Instance;

        logger1.Log("Application has started.");

        Logger logger2 = Logger.Instance;

        logger2.Log("Processing data.");

        Console.WriteLine($"logger1 and logger2 are the same instance: {object.ReferenceEquals(logger1, logger2)}");

    }

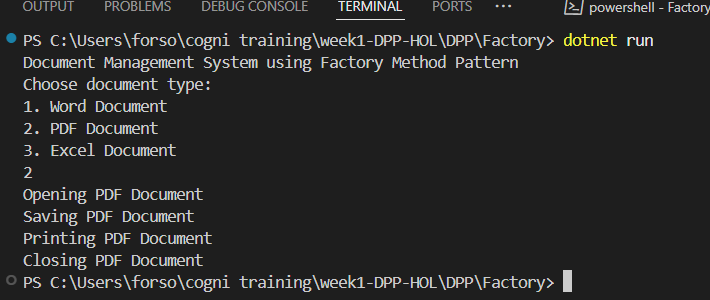
}

**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.

**OUTPUT:**



**CODE:**

// Exercise 2: Implementing the Factory Method Pattern

// Document Management System

using System;

// Step 2: Define Document Interfaces

public interface IDocument

{

    void Open();

    void Close();

    void Save();

    void Print();

}

// Step 3: Create Concrete Document Classes

public class WordDocument : IDocument

{

    public void Open()

    {

        Console.WriteLine("Opening Word Document");

    }

    public void Close()

    {

        Console.WriteLine("Closing Word Document");

    }

    public void Save()

    {

        Console.WriteLine("Saving Word Document");

    }

    public void Print()

    {

        Console.WriteLine("Printing Word Document");

    }

}

public class PdfDocument : IDocument

{

    public void Open()

    {

        Console.WriteLine("Opening PDF Document");

    }

    public void Close()

    {

        Console.WriteLine("Closing PDF Document");

    }

    public void Save()

    {

        Console.WriteLine("Saving PDF Document");

    }

    public void Print()

    {

        Console.WriteLine("Printing PDF Document");

    }

}

public class ExcelDocument : IDocument

{

    public void Open()

    {

        Console.WriteLine("Opening Excel Document");

    }

    public void Close()

    {

        Console.WriteLine("Closing Excel Document");

    }

    public void Save()

    {

        Console.WriteLine("Saving Excel Document");

    }

    public void Print()

    {

        Console.WriteLine("Printing Excel Document");

    }

}

// Step 4: Implement the Factory Method

public abstract class DocumentFactory

{

    public abstract IDocument CreateDocument();

}

public class WordDocumentFactory : DocumentFactory

{

    public override IDocument CreateDocument()

    {

        return new WordDocument();

    }

}

public class PdfDocumentFactory : DocumentFactory

{

    public override IDocument CreateDocument()

    {

        return new PdfDocument();

    }

}

public class ExcelDocumentFactory : DocumentFactory

{

    public override IDocument CreateDocument()

    {

        return new ExcelDocument();

    }

}

// Step 5: Test the Factory Method Implementation

public class FactoryMethodPatternExample

{

    static void Main(string[] args)

    {

        Console.WriteLine("Document Management System using Factory Method Pattern");

        Console.WriteLine("Choose document type:");

        Console.WriteLine("1. Word Document");

        Console.WriteLine("2. PDF Document");

        Console.WriteLine("3. Excel Document");

        int choice = int.Parse(Console.ReadLine());

        DocumentFactory factory = null;

        switch (choice)

        {

            case 1:

                factory = new WordDocumentFactory();

                break;

            case 2:

                factory = new PdfDocumentFactory();

                break;

            case 3:

                factory = new ExcelDocumentFactory();

                break;

            default:

                Console.WriteLine("Invalid choice!");

                return;

        }

        IDocument document = factory.CreateDocument();

        document.Open();

        document.Save();

        document.Print();

        document.Close();

        Console.ReadKey();

    }

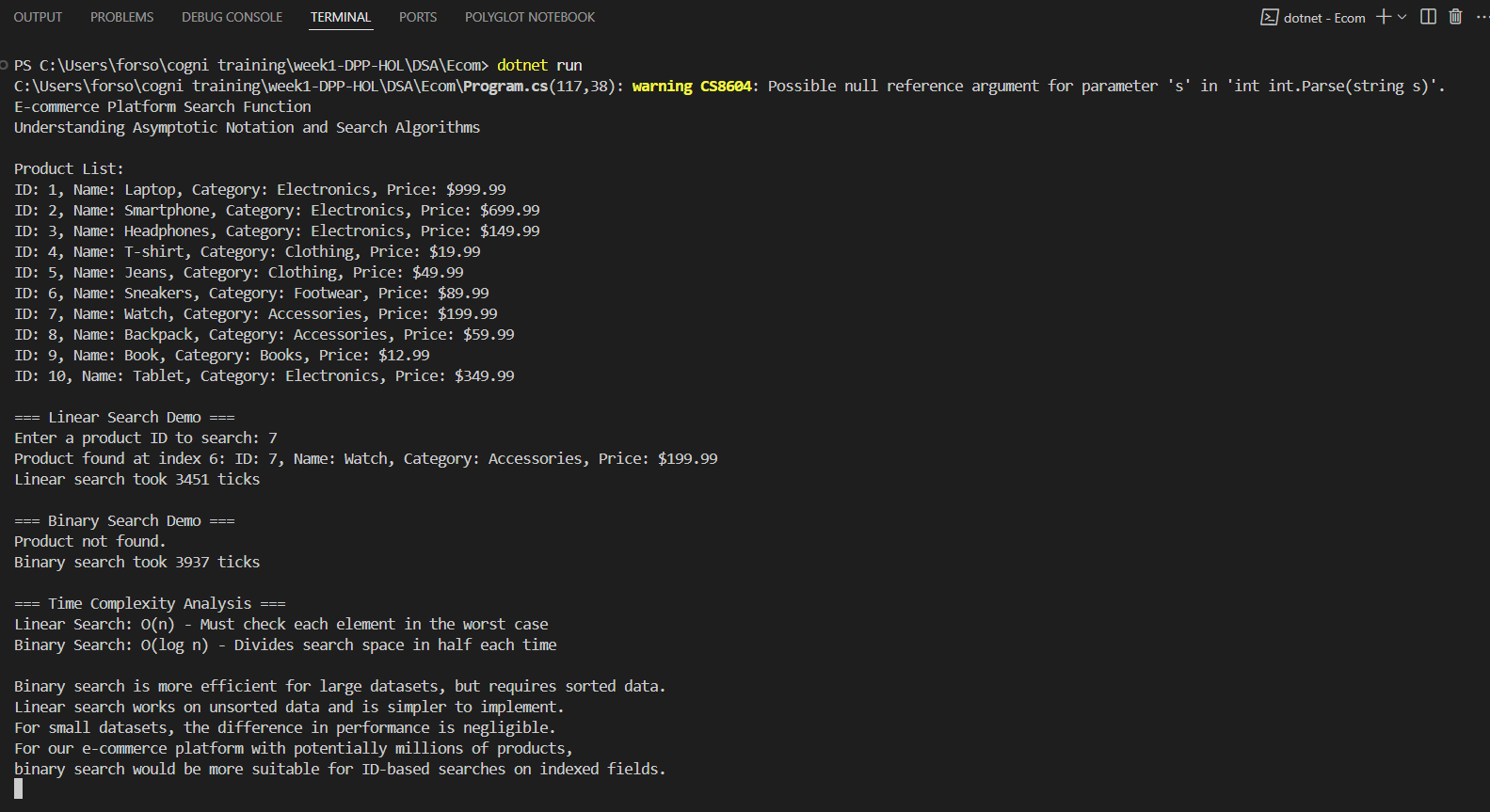
}

**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.

**OUTPUT:**



**CODE:**

// Exercise 3: E-commerce Platform Search Function

// Search Algorithms Implementation

using System;

using System.Collections.Generic;

using System.Diagnostics;

namespace EcommerceSearchFunction

{

    // Product class with attributes for searching

    public class Product

    {

        public int ProductId { get; set; }

        public string ProductName { get; set; }

        public string Category { get; set; }

        public decimal Price { get; set; }

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

        {

            ProductId = id;

            ProductName = name;

            Category = category;

            Price = price;

        }

        public override string ToString()

        {

            return $"ID: {ProductId}, Name: {ProductName}, Category: {Category}, Price: ${Price}";

        }

    }

    public class SearchAlgorithms

    {

        // Linear search implementation

        public static int LinearSearch<T>(T[] array, Predicate<T> match)

        {

            for (int i = 0; i < array.Length; i++)

            {

                if (match(array[i]))

                {

                    return i;

                }

            }

            return -1; // Not found

        }

        // Binary search implementation (requires sorted array)

        public static int BinarySearch<T>(T[] sortedArray, Predicate<T> match, Func<T, T, int> compare)

        {

            int left = 0;

            int right = sortedArray.Length - 1;

            while (left <= right)

            {

                int mid = (left + right) / 2;

                if (match(sortedArray[mid]))

                {

                    return mid;

                }

                // Assuming sortedArray[0] is the smallest value

                T midValue = sortedArray[mid];

                if (compare(midValue, sortedArray[0]) < 0)

                {

                    right = mid - 1;

                }

                else

                {

                    left = mid + 1;

                }

            }

            return -1; // Not found

        }

    }

    public class Program

    {

        static void Main(string[] args)

        {

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

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

            // Create a list of products

            List<Product> products = new List<Product>

            {

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

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

                new Product(3, "Headphones", "Electronics", 149.99m),

                new Product(4, "T-shirt", "Clothing", 19.99m),

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

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

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

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

                new Product(9, "Book", "Books", 12.99m),

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

            };

            // Convert to array for search algorithms

            Product[] productArray = products.ToArray();

            // Sorted array by ProductId for binary search

            Product[] sortedByIdArray = new Product[productArray.Length];

            Array.Copy(productArray, sortedByIdArray, productArray.Length);

            Array.Sort(sortedByIdArray, (p1, p2) => p1.ProductId.CompareTo(p2.ProductId));

            Console.WriteLine("\nProduct List:");

            foreach (var product in productArray)

            {

                Console.WriteLine(product);

            }

            // Demo: Linear Search

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

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

            int searchId = int.Parse(Console.ReadLine());

            Stopwatch stopwatch = new Stopwatch();

            stopwatch.Start();

            int linearResult = SearchAlgorithms.LinearSearch(productArray, p => ((Product)p).ProductId == searchId);

            stopwatch.Stop();

            if (linearResult != -1)

            {

                Console.WriteLine($"Product found at index {linearResult}: {productArray[linearResult]}");

            }

            else

            {

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

            }

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

            // Demo: Binary Search

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

            stopwatch.Restart();

            int binaryResult = SearchAlgorithms.BinarySearch(

                sortedByIdArray,

                p => ((Product)p).ProductId == searchId,

                (p1, p2) => ((Product)p1).ProductId.CompareTo(((Product)p2).ProductId)

            );

            stopwatch.Stop();

            if (binaryResult != -1)

            {

                Console.WriteLine($"Product found at index {binaryResult}: {sortedByIdArray[binaryResult]}");

            }

            else

            {

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

            }

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

            // Analysis

            Console.WriteLine("\n=== 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("\nBinary search is more efficient for large datasets, but requires sorted data.");

            Console.WriteLine("Linear search works on unsorted data and is simpler 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.");

            Console.ReadKey();

        }

    }

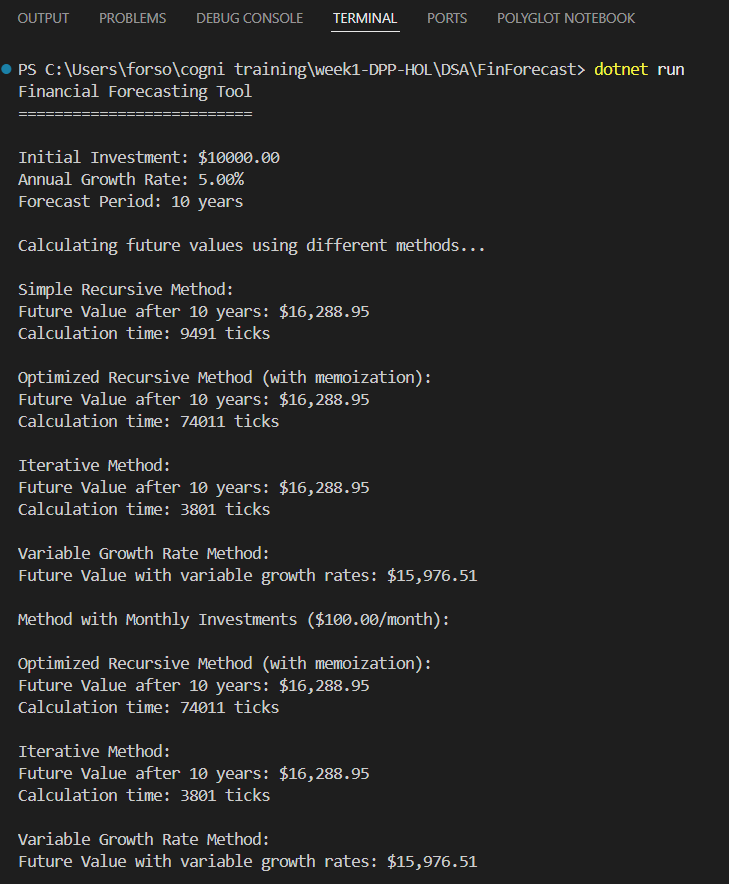
}

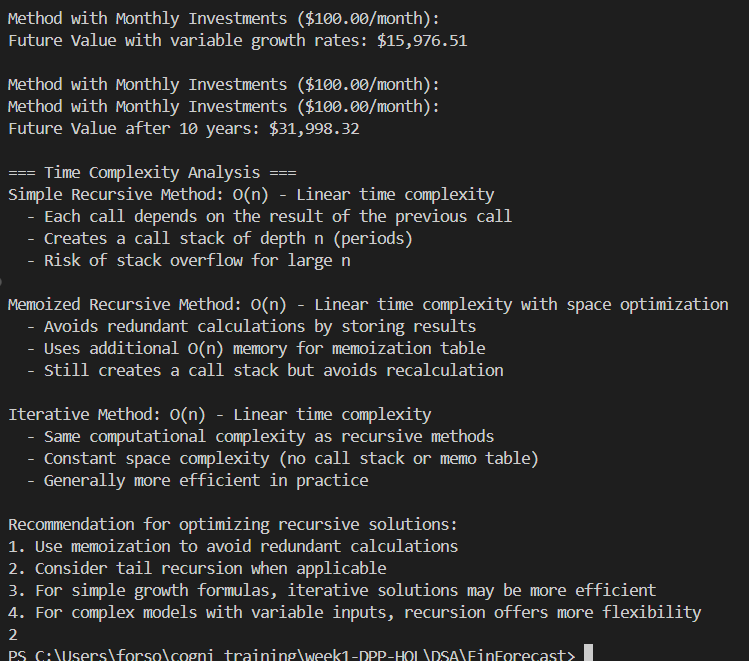
**Exercise 7: Financial Forecasting**

**Scenario:**

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

**OUTPUT:**



****

**CODE:**

// Exercise 7: Financial Forecasting

// Recursive algorithm to predict future values

using System;

using System.Collections.Generic;

using System.Diagnostics;

namespace FinancialForecasting

{

    public class FinancialForecast

    {

        // Simple recursive method to calculate future value with constant growth rate

        public static decimal CalculateFutureValueRecursive(decimal presentValue, decimal growthRate, int periods)

        {

            // Base case: when no periods left, return the present value

            if (periods == 0)

            {

                return presentValue;

            }

            // Recursive case: calculate future value based on previous period

            return CalculateFutureValueRecursive(presentValue, growthRate, periods - 1) \* (1 + growthRate);

        }

        // Optimized recursive method using memoization to avoid redundant calculations

        public static decimal CalculateFutureValueMemoized(decimal presentValue, decimal growthRate, int periods, Dictionary<int, decimal> memo = null)

        {

            // Initialize memoization dictionary if not provided

            if (memo == null)

            {

                memo = new Dictionary<int, decimal>();

            }

            // If result is already calculated, return from memo

            if (memo.ContainsKey(periods))

            {

                return memo[periods];

            }

            // Base case: when no periods left, return the present value

            if (periods == 0)

            {

                return presentValue;

            }

            // Recursive case: calculate future value based on previous period

            decimal result = CalculateFutureValueMemoized(presentValue, growthRate, periods - 1, memo) \* (1 + growthRate);

            // Store result in memo for future use

            memo[periods] = result;

            return result;

        }

        // Iterative method for comparison

        public static decimal CalculateFutureValueIterative(decimal presentValue, decimal growthRate, int periods)

        {

            decimal result = presentValue;

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

            {

                result \*= (1 + growthRate);

            }

            return result;

        }

        // Advanced recursive forecasting with variable growth rates

        public static decimal CalculateFutureValueWithVariableGrowth(decimal presentValue, decimal[] growthRates, int currentPeriod = 0)

        {

            // Base case: when we've applied all growth rates, return the current value

            if (currentPeriod >= growthRates.Length)

            {

                return presentValue;

            }

            // Calculate value for current period

            decimal newValue = presentValue \* (1 + growthRates[currentPeriod]);

            // Recurse for next period

            return CalculateFutureValueWithVariableGrowth(newValue, growthRates, currentPeriod + 1);

        }

        // Recursive forecasting with additional investments

        public static decimal CalculateFutureValueWithInvestments(decimal currentValue, decimal periodicInvestment, decimal growthRate, int periodsRemaining)

        {

            // Base case: no more periods

            if (periodsRemaining == 0)

            {

                return currentValue;

            }

            // Calculate new value after growth and add investment

            decimal newValue = currentValue \* (1 + growthRate) + periodicInvestment;

            // Recurse for next period

            return CalculateFutureValueWithInvestments(newValue, periodicInvestment, growthRate, periodsRemaining - 1);

        }

    }

    public class Program

    {

        static void Main(string[] args)

        {

            Console.WriteLine("Financial Forecasting Tool");

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

            // Example initial values

            decimal initialValue = 10000.00m;

            decimal annualGrowthRate = 0.05m; // 5%

            int forecastYears = 10;

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

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

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

            // Measure performance of different methods

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

            Stopwatch stopwatch = new Stopwatch();

            // Test simple recursive method

            stopwatch.Start();

            decimal futureValueRecursive = FinancialForecast.CalculateFutureValueRecursive(initialValue, annualGrowthRate, forecastYears);

            stopwatch.Stop();

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

            Console.WriteLine($"Future Value after {forecastYears} years: ${futureValueRecursive:N2}");

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

            // Test memoized recursive method

            stopwatch.Restart();

            decimal futureValueMemoized = FinancialForecast.CalculateFutureValueMemoized(initialValue, annualGrowthRate, forecastYears);

            stopwatch.Stop();

            Console.WriteLine($"\nOptimized Recursive Method (with memoization):");

            Console.WriteLine($"Future Value after {forecastYears} years: ${futureValueMemoized:N2}");

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

            // Test iterative method

            stopwatch.Restart();

            decimal futureValueIterative = FinancialForecast.CalculateFutureValueIterative(initialValue, annualGrowthRate, forecastYears);

            stopwatch.Stop();

            Console.WriteLine($"\nIterative Method:");

            Console.WriteLine($"Future Value after {forecastYears} years: ${futureValueIterative:N2}");

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

            // Example with variable growth rates

            decimal[] variableGrowthRates = { 0.03m, 0.04m, 0.05m, 0.045m, 0.05m, 0.055m, 0.06m, 0.055m, 0.05m, 0.045m };

            decimal futureValueVariableGrowth = FinancialForecast.CalculateFutureValueWithVariableGrowth(initialValue, variableGrowthRates);

            Console.WriteLine($"\nVariable Growth Rate Method:");

            Console.WriteLine($"Future Value with variable growth rates: ${futureValueVariableGrowth:N2}");

            // Example with additional investments

            decimal monthlyInvestment = 100.00m;

            decimal monthlyGrowthRate = annualGrowthRate / 12;

            int forecastMonths = forecastYears \* 12;

            decimal futureValueWithInvestments = FinancialForecast.CalculateFutureValueWithInvestments(initialValue, monthlyInvestment, monthlyGrowthRate, forecastMonths);

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

            Console.WriteLine($"Future Value after {forecastYears} years: ${futureValueWithInvestments:N2}");

            // Analysis

            Console.WriteLine("\n=== 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("\nMemoized 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("\nIterative 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("\nRecommendation 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");

            Console.ReadKey();

        }

    }

}