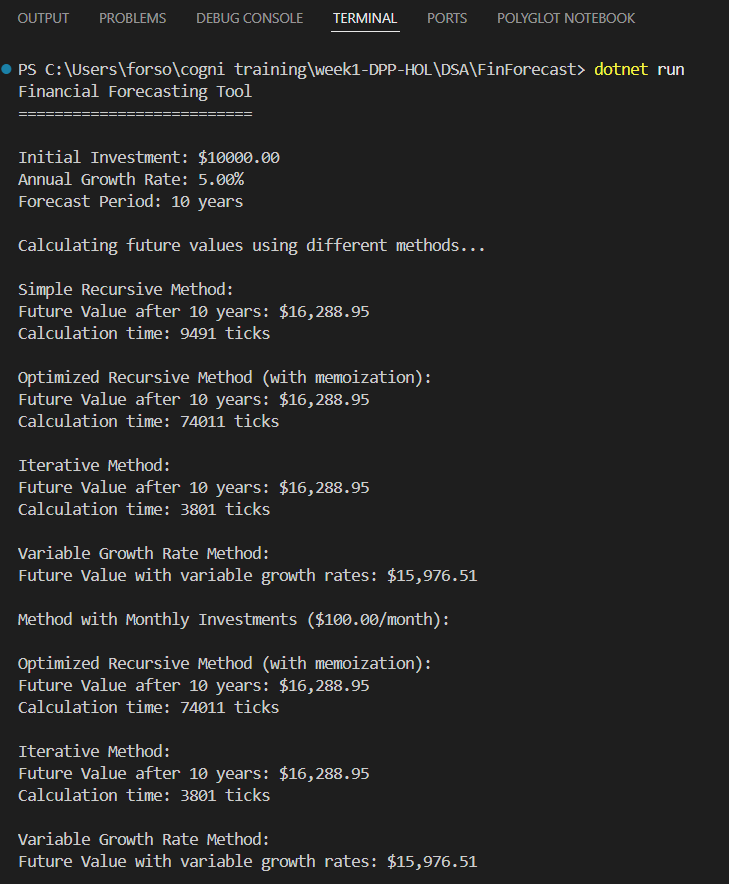
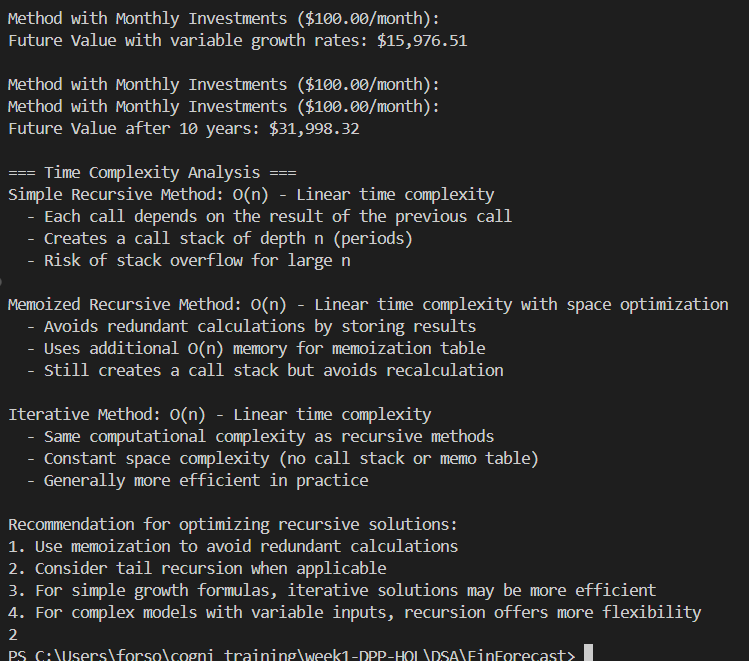
**Exercise 7: Financial Forecasting**

**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();

        }

    }

}