**WEEK ONE ASSIGNMENTS**

**Design principles & Patterns**

**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.**

**Steps:**

1. **Create a New C# Project:**
   * **Create a new C# project named SingletonPatternExample.**
2. **Define a Singleton Class:**
   * **Create a class named Logger that has a private static instance of itself.**
   * **Ensure the constructor of Logger is private.**
   * **Provide a public static method to get the instance of the Logger class.**
3. **Implement the Singleton Pattern:**
   * **Write code to ensure that the Logger class follows the Singleton design pattern.**
4. **Test the Singleton Implementation:**
   * **Create a test class to verify that only one instance of Logger is created and used across the application.**

**[Program.cs](http://program.cs)**

using System;

using System.Collections.Generic;

// Logger class implementing Singleton Pattern

public class Logger

{

// Step 1: Private static instance of the class

private static Logger \_instance = null;

// Step 2: Private constructor to prevent direct instantiation

private Logger()

{

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

LogEntries = new List<string>();

}

// Step 3: Public static method to get the single instance

public static Logger GetInstance()

{

// Create instance only if it doesn't exist (lazy initialization)

if (\_instance == null)

{

\_instance = new Logger();

}

return \_instance;

}

// Logger functionality

private List<string> LogEntries { get; set; }

// Method to log messages

public void LogMessage(string message)

{

string timestamp = DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss");

string logEntry = $"[{timestamp}] {message}";

LogEntries.Add(logEntry);

Console.WriteLine($"LOGGED: {logEntry}");

}

// Method to log errors

public void LogError(string error)

{

LogMessage($"ERROR: {error}");

}

// Method to log warnings

public void LogWarning(string warning)

{

LogMessage($"WARNING: {warning}");

}

// Method to display all logs

public void DisplayAllLogs()

{

Console.WriteLine("\n=== ALL LOG ENTRIES ===");

if (LogEntries.Count == 0)

{

Console.WriteLine("No log entries found.");

}

else

{

for (int i = 0; i < LogEntries.Count; i++)

{

Console.WriteLine($"{i + 1}. {LogEntries[i]}");

}

}

Console.WriteLine($"Total entries: {LogEntries.Count}\n");

}

// Method to get instance ID (for testing purposes)

public int GetInstanceId()

{

return this.GetHashCode();

}

}

// Test class to verify Singleton implementation

public class SingletonTest

{

public static void TestSingletonPattern()

{

Console.WriteLine("=== TESTING SINGLETON PATTERN ===\n");

// Test 1: Get first instance

Console.WriteLine("1. Getting first Logger instance...");

Logger logger1 = Logger.GetInstance();

Console.WriteLine($" Logger1 Instance ID: {logger1.GetInstanceId()}");

// Test 2: Get second instance

Console.WriteLine("\n2. Getting second Logger instance...");

Logger logger2 = Logger.GetInstance();

Console.WriteLine($" Logger2 Instance ID: {logger2.GetInstanceId()}");

// Test 3: Check if both references point to same instance

Console.WriteLine("\n3. Checking if both instances are the same...");

bool areSameInstance = ReferenceEquals(logger1, logger2);

Console.WriteLine($" Are logger1 and logger2 the same instance? {areSameInstance}");

// Test 4: Test shared state

Console.WriteLine("\n4. Testing shared state...");

logger1.LogMessage("Message from logger1");

logger2.LogMessage("Message from logger2");

Console.WriteLine("\n Displaying logs from logger1:");

logger1.DisplayAllLogs();

Console.WriteLine(" Displaying logs from logger2:");

logger2.DisplayAllLogs();

// Test 5: Multiple references test

Console.WriteLine("5. Creating multiple references...");

Logger logger3 = Logger.GetInstance();

Logger logger4 = Logger.GetInstance();

Logger logger5 = Logger.GetInstance();

Console.WriteLine($" Logger3 Instance ID: {logger3.GetInstanceId()}");

Console.WriteLine($" Logger4 Instance ID: {logger4.GetInstanceId()}");

Console.WriteLine($" Logger5 Instance ID: {logger5.GetInstanceId()}");

bool allSame = ReferenceEquals(logger1, logger3) &&

ReferenceEquals(logger1, logger4) &&

ReferenceEquals(logger1, logger5);

Console.WriteLine($" All instances are the same? {allSame}");

}

}

// Simulate different parts of application using the logger

public class UserService

{

private Logger logger;

public UserService()

{

logger = Logger.GetInstance();

}

public void CreateUser(string username)

{

logger.LogMessage($"UserService: Creating user '{username}'");

// Simulate user creation logic

logger.LogMessage($"UserService: User '{username}' created successfully");

}

public void DeleteUser(string username)

{

logger.LogWarning($"UserService: Attempting to delete user '{username}'");

// Simulate user deletion logic

logger.LogMessage($"UserService: User '{username}' deleted");

}

}

public class PaymentService

{

private Logger logger;

public PaymentService()

{

logger = Logger.GetInstance();

}

public void ProcessPayment(decimal amount)

{

logger.LogMessage($"PaymentService: Processing payment of ${amount}");

// Simulate payment processing

if (amount <= 0)

{

logger.LogError($"PaymentService: Invalid payment amount ${amount}");

return;

}

logger.LogMessage($"PaymentService: Payment of ${amount} processed successfully");

}

}

// Main program

public class Program

{

public static void Main()

{

Console.WriteLine("=== SINGLETON PATTERN LOGGER DEMO ===\n");

// Explain Singleton Pattern

Console.WriteLine("=== SINGLETON PATTERN EXPLANATION ===");

Console.WriteLine("The Singleton pattern ensures that:");

Console.WriteLine("• Only ONE instance of a class can exist");

Console.WriteLine("• Provides global access to that instance");

Console.WriteLine("• Useful for logging, database connections, configuration settings");

Console.WriteLine("\nKey components:");

Console.WriteLine("• Private static instance variable");

Console.WriteLine("• Private constructor (prevents direct instantiation)");

Console.WriteLine("• Public static method to get the instance\n");

// Test the Singleton implementation

SingletonTest.TestSingletonPattern();

Console.WriteLine("\n=== PRACTICAL USAGE EXAMPLE ===");

Console.WriteLine("Different services using the same Logger instance:\n");

// Create different services that use the logger

UserService userService = new UserService();

PaymentService paymentService = new PaymentService();

// Use the services (they all use the same logger instance)

userService.CreateUser("john\_doe");

userService.CreateUser("jane\_smith");

paymentService.ProcessPayment(99.99m);

paymentService.ProcessPayment(-10.00m); // This will cause an error

paymentService.ProcessPayment(149.50m);

userService.DeleteUser("john\_doe");

// Display all logs from any logger instance

Logger finalLogger = Logger.GetInstance();

finalLogger.DisplayAllLogs();

Console.WriteLine("=== SINGLETON PATTERN BENEFITS ===");

Console.WriteLine(" Controlled access to single instance");

Console.WriteLine(" Reduced memory usage (only one instance)");

Console.WriteLine(" Global access point");

Console.WriteLine(" Consistent state across application");

Console.WriteLine(" Perfect for logging, caching, thread pools");

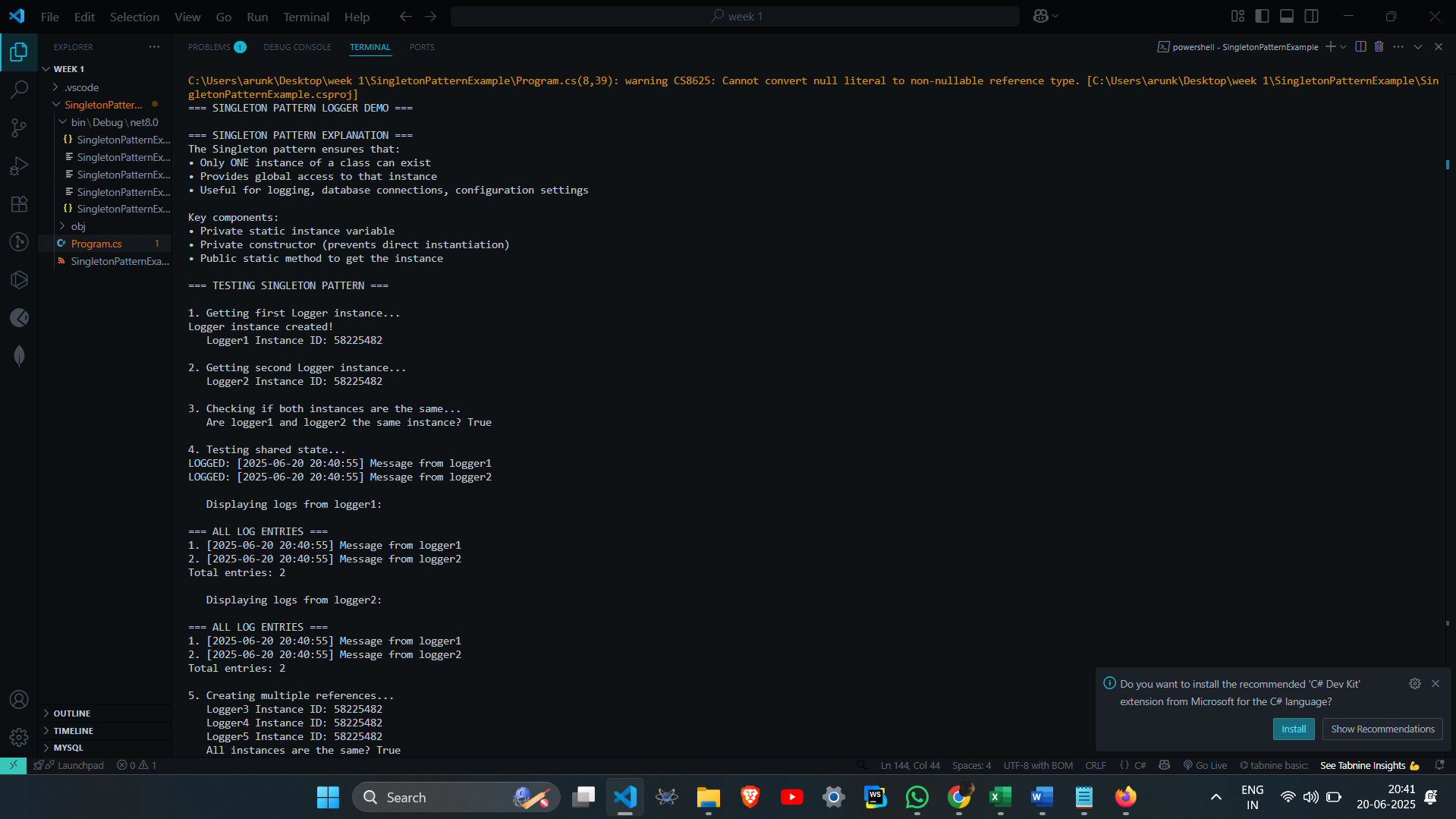
}

}

}

OUTPUTS

:



**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.**

**Steps:**

1. **Create a New Java Project:**
   * **Create a new Java project named FactoryMethodPatternExample.**
2. **Define Document Classes:**
   * **Create interfaces or abstract classes for different document types such as WordDocument, PdfDocument, and ExcelDocument.**
3. **Create Concrete Document Classes:**
   * **Implement concrete classes for each document type that implements or extends the above interfaces or abstract classes.**
4. **Implement the Factory Method:**
   * **Create an abstract class DocumentFactory with a method createDocument().**
   * **Create concrete factory classes for each document type that extends DocumentFactory and implements the createDocument() method.**
5. **Test the Factory Method Implementation:**
   * **Create a test class to demonstrate the creation of different document types using the factory method.**

**[Program.cs](http://program.cs)**

using System;

public interface IDocument

{

void Create();

void Open();

void Save();

}

public class WordDocument : IDocument

{

public void Create()

{

Console.WriteLine("Creating a Word document...");

}

public void Open()

{

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

}

public void Save()

{

Console.WriteLine("Saving Word document as .docx file.");

}

}

public class PdfDocument : IDocument

{

public void Create()

{

Console.WriteLine("Creating a PDF document...");

}

public void Open()

{

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

}

public void Save()

{

Console.WriteLine("Saving PDF document as .pdf file.");

}

}

public class ExcelDocument : IDocument

{

public void Create()

{

Console.WriteLine("Creating an Excel document...");

}

public void Open()

{

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

}

public void Save()

{

Console.WriteLine("Saving Excel document as .xlsx file.");

}

}

public abstract class DocumentFactory

{

public abstract IDocument CreateDocument();

public void ProcessDocument()

{

IDocument document = CreateDocument();

document.Create();

document.Open();

document.Save();

}

}

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();

}

}

public class Program

{

public static void Main(string[] args)

{

Console.WriteLine("=== Factory Method Pattern Demo ===\n");

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

DocumentFactory wordFactory = new WordDocumentFactory();

wordFactory.ProcessDocument();

Console.WriteLine("\n2. Creating PDF Document:");

DocumentFactory pdfFactory = new PdfDocumentFactory();

pdfFactory.ProcessDocument();

Console.WriteLine("\n3. Creating Excel Document:");

DocumentFactory excelFactory = new ExcelDocumentFactory();

excelFactory.ProcessDocument();

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

Console.WriteLine("\n4. Creating documents individually:");

IDocument word = wordFactory.CreateDocument();

IDocument pdf = pdfFactory.CreateDocument();

IDocument excel = excelFactory.CreateDocument();

word.Create();

pdf.Create();

excel.Create();

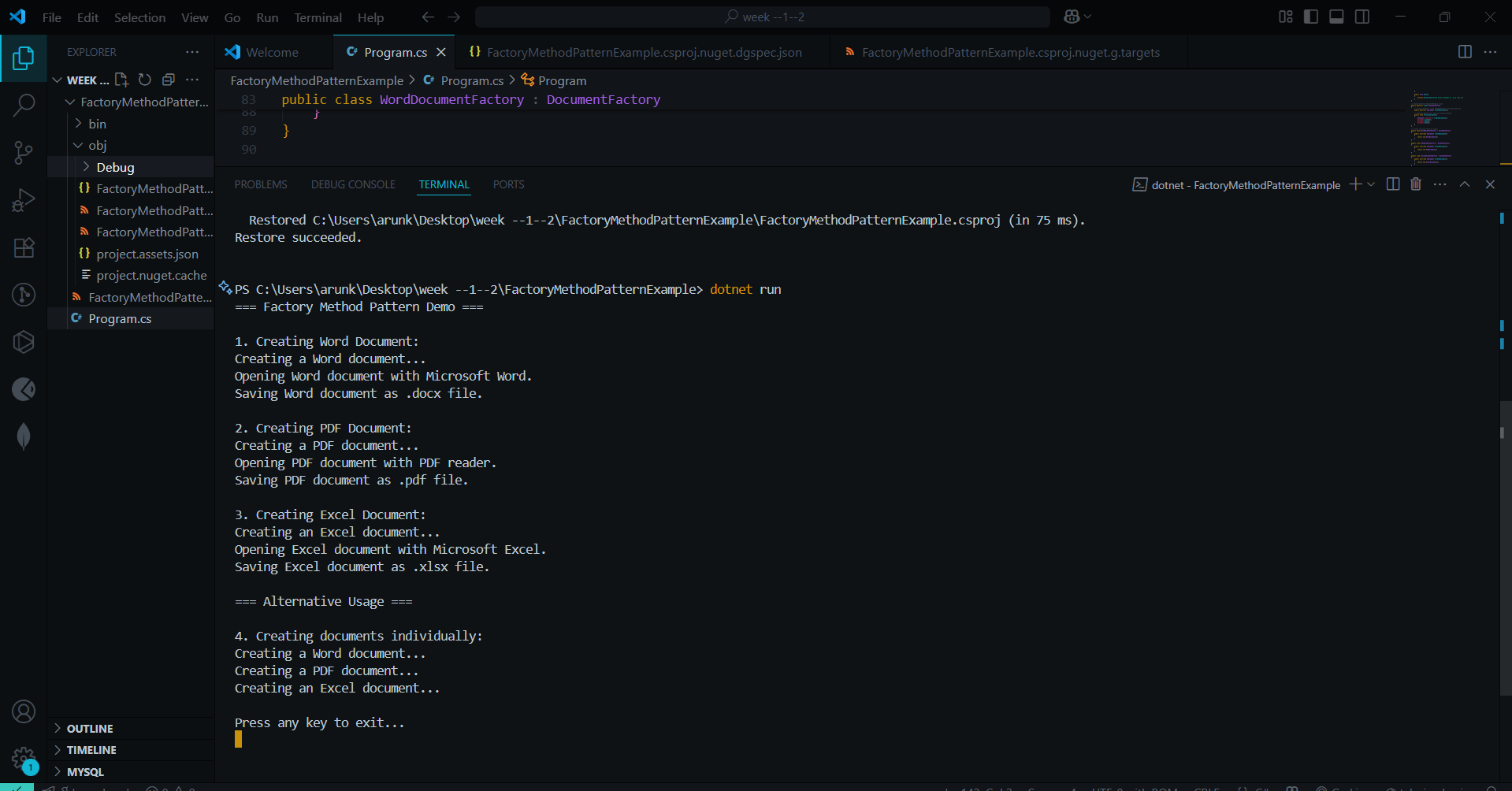
Console.WriteLine("\nPress any key to exit...");

Console.ReadKey();

}

}

**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.**

**Steps:**

1. **Understand Asymptotic Notation:**
   * **Explain Big O notation and how it helps in analyzing algorithms.**
   * **Describe the best, average, and worst-case scenarios for search operations.**
2. **Setup:**
   * **Create a class Product with attributes for searching, such as productId, productName, and category.**
3. **Implementation:**
   * **Implement linear search and binary search algorithms.**
   * **Store products in an array for linear search and a sorted array for binary search.**
4. **Analysis:**
   * **Compare the time complexity of linear and binary search algorithms.**
   * **Discuss which algorithm is more suitable for your platform and why.**

**[Program.cs](http://program.cs)**

using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.Linq;

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

}

}

public class ECommerceSearchPlatform

{

private Product[] products;

private Product[] sortedProducts;

public ECommerceSearchPlatform()

{

InitializeProducts();

}

private void InitializeProducts()

{

products = new Product[]

{

new Product(101, "iPhone 15", "Electronics"),

new Product(205, "Nike Air Max", "Shoes"),

new Product(310, "Samsung TV", "Electronics"),

new Product(150, "Adidas Sneakers", "Shoes"),

new Product(420, "MacBook Pro", "Electronics"),

new Product(305, "Puma Running Shoes", "Shoes"),

new Product(180, "Dell Laptop", "Electronics"),

new Product(275, "Converse Shoes", "Shoes"),

new Product(390, "iPad Pro", "Electronics"),

new Product(220, "Reebok Trainers", "Shoes")

};

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

}

public Product LinearSearchById(int productId)

{

Console.WriteLine($"Linear Search for Product ID: {productId}");

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

{

if (products[i].ProductId == productId)

{

Console.WriteLine($"Found at index {i} after checking {i + 1} products");

return products[i];

}

}

Console.WriteLine($"Not found after checking all {products.Length} products");

return null;

}

public Product BinarySearchById(int productId)

{

Console.WriteLine($"Binary Search for Product ID: {productId}");

int left = 0;

int right = sortedProducts.Length - 1;

int comparisons = 0;

while (left <= right)

{

comparisons++;

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

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

{

Console.WriteLine($"Found at index {mid} after {comparisons} comparisons");

return sortedProducts[mid];

}

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

{

left = mid + 1;

}

else

{

right = mid - 1;

}

}

Console.WriteLine($"Not found after {comparisons} comparisons");

return null;

}

public void CompareSearchPerformance(int searchId)

{

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

Stopwatch sw = new Stopwatch();

sw.Start();

var linearResult = LinearSearchById(searchId);

sw.Stop();

long linearTime = sw.ElapsedTicks;

Console.WriteLine($"Linear Search Time: {linearTime} ticks\n");

sw.Restart();

var binaryResult = BinarySearchById(searchId);

sw.Stop();

long binaryTime = sw.ElapsedTicks;

Console.WriteLine($"Binary Search Time: {binaryTime} ticks");

Console.WriteLine($"Binary search was {(double)linearTime / binaryTime:F2}x faster\n");

}

public void DisplayProducts()

{

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

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

{

Console.WriteLine($"{i}: {products[i]}");

}

Console.WriteLine();

Console.WriteLine("=== SORTED PRODUCTS (for Binary Search) ===");

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

{

Console.WriteLine($"{i}: {sortedProducts[i]}");

}

Console.WriteLine();

}

}

public class Program

{

public static void Main()

{

Console.WriteLine("=== E-COMMERCE PLATFORM SEARCH DEMO ===\n");

// Big O Notation Explanation

Console.WriteLine("=== BIG O NOTATION EXPLANATION ===");

Console.WriteLine("Big O notation describes algorithm efficiency:");

Console.WriteLine("• O(1) - Constant time (best)");

Console.WriteLine("• O(log n) - Logarithmic time (very good)");

Console.WriteLine("• O(n) - Linear time (acceptable)");

Console.WriteLine("• O(n²) - Quadratic time (poor for large data)\n");

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

Console.WriteLine("• Best Case: Element found immediately");

Console.WriteLine("• Average Case: Element found in middle");

Console.WriteLine("• Worst Case: Element is last or not found\n");

var platform = new ECommerceSearchPlatform();

// Display products

platform.DisplayProducts();

// Test different scenarios

Console.WriteLine("=== BEST CASE SCENARIO ===");

platform.CompareSearchPerformance(101); // First element in sorted array

Console.WriteLine("=== AVERAGE CASE SCENARIO ===");

platform.CompareSearchPerformance(275); // Middle element

Console.WriteLine("=== WORST CASE SCENARIO ===");

platform.CompareSearchPerformance(420); // Last element

Console.WriteLine("=== NOT FOUND SCENARIO ===");

platform.CompareSearchPerformance(999); // Non-existent product

Console.WriteLine("=== TIME COMPLEXITY ANALYSIS ===");

Console.WriteLine("Linear Search:");

Console.WriteLine("• Best Case: O(1) - found at first position");

Console.WriteLine("• Average Case: O(n/2) ≈ O(n) - found in middle");

Console.WriteLine("• Worst Case: O(n) - found at last position or not found");

Console.WriteLine();

Console.WriteLine("Binary Search:");

Console.WriteLine("• Best Case: O(1) - found at middle position");

Console.WriteLine("• Average Case: O(log n) - requires log₂(n) comparisons");

Console.WriteLine("• Worst Case: O(log n) - maximum log₂(n) comparisons");

Console.WriteLine();

Console.WriteLine("=== RECOMMENDATION FOR E-COMMERCE PLATFORM ===");

Console.WriteLine("Use BINARY SEARCH when:");

Console.WriteLine("• Data can be kept sorted");

Console.WriteLine("• Frequent search operations");

Console.WriteLine("• Large product catalog (1000+ items)");

Console.WriteLine("• Searching by sortable fields (ID, price, name)");

Console.WriteLine();

Console.WriteLine("Use LINEAR SEARCH when:");

Console.WriteLine("• Small product catalog (<100 items)");

Console.WriteLine("• Data changes frequently (hard to maintain sorted order)");

Console.WriteLine("• Searching by non-sortable attributes");

Console.WriteLine("• One-time searches");

Console.WriteLine();

Console.WriteLine( For a real e-commerce platform:");

Console.WriteLine("• Use database indexing (B-tree indexes) for O(log n) performance");

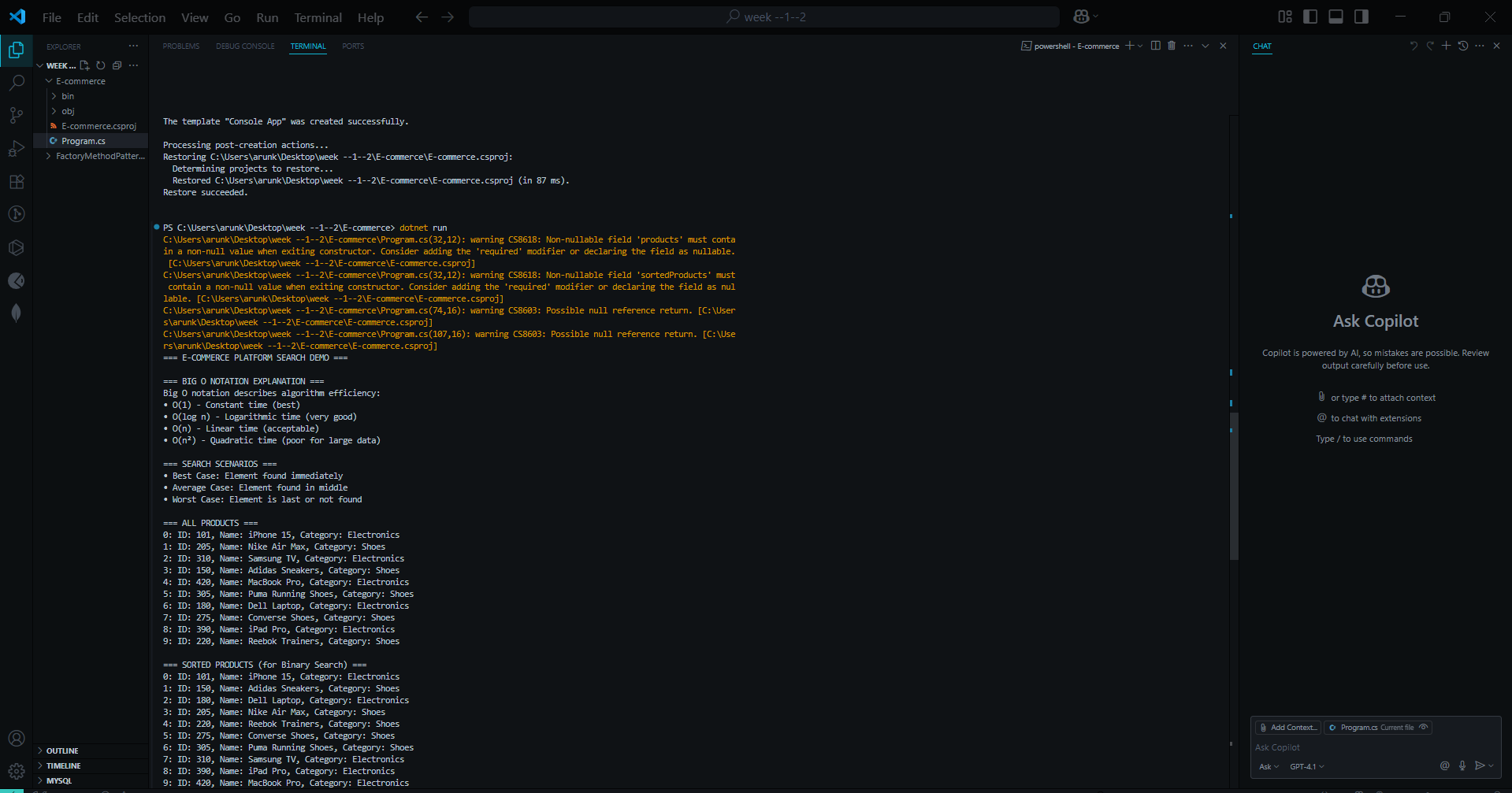
Console.WriteLine("• Implement search engines like Elasticsearch for complex queries");

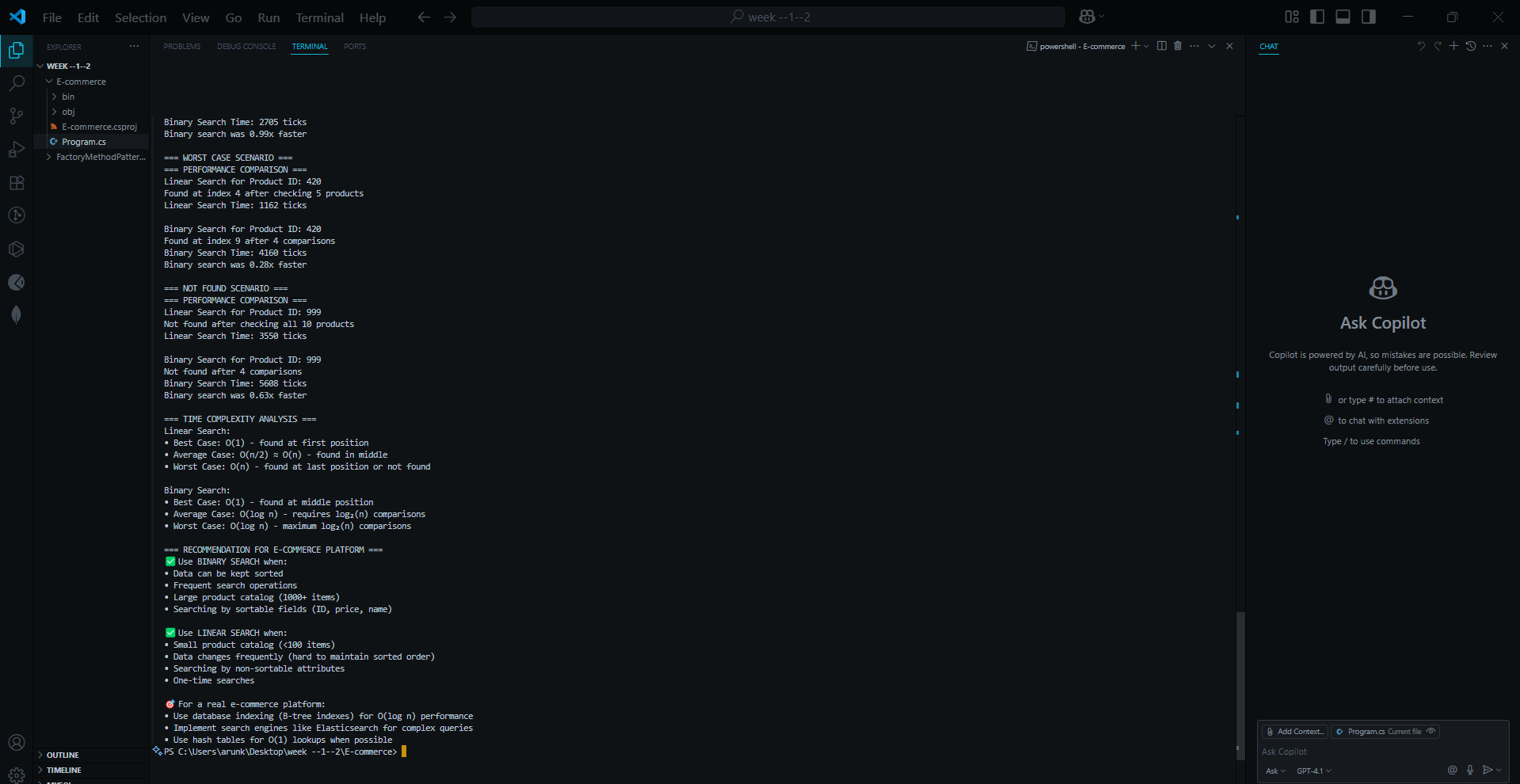
Console.WriteLine("• Use hash tables for O(1) lookups when possible");

}

}

**OUTPUT**





**Exercise 7: Financial Forecasting**

**Scenario:**

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

**Steps:**

1. **Understand Recursive Algorithms:**
   * **Explain the concept of recursion and how it can simplify certain problems.**
2. **Setup:**
   * **Create a method to calculate the future value using a recursive approach.**
3. **Implementation:**
   * **Implement a recursive algorithm to predict future values based on past growth rates.**
4. **Analysis:**
   * **Discuss the time complexity of your recursive algorithm.**
   * **Explain how to optimize the recursive solution to avoid excessive computation.**

[Program.cs](http://program.cs)

using System;

using System.Collections.Generic;

using System.Diagnostics;

public class FinancialForecaster

{

(optimization)

private Dictionary<int, double> memo = new Dictionary<int, double>();

public int CalculationCount { get; private set;

}

public double PredictFutureValue\_Basic(double initialValue, double growthRate, int years)

{

CalculationCount++;

if (years == 0)

{

return initialValue;

}

return PredictFutureValue\_Basic(initialValue \* (1 + growthRate), growthRate, years - 1);

}

public double PredictFutureValue\_Optimized(double initialValue, double growthRate, int years)

{

if (memo.ContainsKey(years))

{

return initialValue \* memo[years];

}

CalculationCount++;

if (years == 0)

{

memo[years] = 1.0; // No growth multiplier

return initialValue;

}

double growthMultiplier = (1 + growthRate) \* GetGrowthMultiplier(growthRate, years - 1);

memo[years] = growthMultiplier;

return initialValue \* growthMultiplier;

}

private double GetGrowthMultiplier(double growthRate, int years)

{

if (memo.ContainsKey(years))

{

return memo[years];

}

if (years == 0)

{

return 1.0;

}

double multiplier = (1 + growthRate) \* GetGrowthMultiplier(growthRate, years - 1);

memo[years] = multiplier;

return multiplier;

}

public double PredictFutureValue\_Iterative(double initialValue, double growthRate, int years)

{

CalculationCount++;

double result = initialValue;

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

{

result \*= (1 + growthRate);

}

return result;

}

public double CalculateCompoundInterest(double principal, double annualRate, int timesCompounded, int years)

{

CalculationCount++;

if (years == 0)

{

return principal;

}

// Compound interest formula applied recursively

// A = P(1 + r/n)^n for one year, then recurse for remaining years

double ratePerPeriod = annualRate / timesCompounded;

double amountAfterOneYear = principal;

// Apply compounding for one year

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

{

amountAfterOneYear \*= (1 + ratePerPeriod);

}

return CalculateCompoundInterest(amountAfterOneYear, annualRate, timesCompounded, years - 1);

}

public void Reset()

{

CalculationCount = 0;

memo.Clear();

}

}

public class Investment

{

public string Name { get; set; }

public double InitialAmount { get; set; }

public double GrowthRate { get; set; }

public Investment(string name, double initial, double rate)

{

Name = name;

InitialAmount = initial;

GrowthRate = rate;

}

}

// Main program

public class Program

{

public static void Main()

{

Console.WriteLine("=== FINANCIAL FORECASTING TOOL ===\n");

// Explain Recursion Concept

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

Console.WriteLine("Recursion is when a function calls itself to solve smaller versions of the same problem.");

Console.WriteLine("Key components:");

Console.WriteLine("• Base Case: Condition that stops the recursion");

Console.WriteLine("• Recursive Case: Function calls itself with modified parameters");

Console.WriteLine("• Each call should move closer to the base case\n");

Console.WriteLine("Financial Forecasting Example:");

Console.WriteLine("• Base Case: If years = 0, return current value");

Console.WriteLine("• Recursive Case: Apply growth for 1 year, then forecast remaining years");

Console.WriteLine("• Formula: FutureValue(year) = CurrentValue × (1 + GrowthRate) × FutureValue(year-1)\n");

var forecaster = new FinancialForecaster();

// Sample investments

var investments = new List<Investment>

{

new Investment("Conservative Portfolio", 10000, 0.05), // 5% annual growth

new Investment("Aggressive Stock Portfolio", 10000, 0.12), // 12% annual growth

new Investment("Tech Startup Investment", 5000, 0.25) // 25% annual growth

};

// Demonstrate forecasting for different time periods

int[] forecastYears = { 5, 10, 15, 20 };

Console.WriteLine("=== INVESTMENT FORECASTING RESULTS ===");

foreach (var investment in investments)

{

Console.WriteLine($"\n--- {investment.Name} ---");

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

Console.WriteLine($"Expected Annual Growth: {investment.GrowthRate:P2}");

Console.WriteLine();

foreach (int years in forecastYears)

{

forecaster.Reset();

var futureValue = forecaster.PredictFutureValue\_Basic(

investment.InitialAmount,

investment.GrowthRate,

years);

Console.WriteLine($"After {years,2} years: ${futureValue,12:F2} " +

$"(Gain: ${futureValue - investment.InitialAmount,10:F2})");

}

}

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

Console.WriteLine("Comparing Basic Recursive vs Optimized vs Iterative approaches:\n");

var testInvestment = investments[0];

int testYears = 20;

Stopwatch sw = new Stopwatch();

forecaster.Reset();

sw.Start();

var basicResult = forecaster.PredictFutureValue\_Basic(

testInvestment.InitialAmount, testInvestment.GrowthRate, testYears);

sw.Stop();

long basicTime = sw.ElapsedTicks;

int basicCalculations = forecaster.CalculationCount;

forecaster.Reset();

sw.Restart();

var optimizedResult = forecaster.PredictFutureValue\_Optimized(

testInvestment.InitialAmount, testInvestment.GrowthRate, testYears);

sw.Stop();

long optimizedTime = sw.ElapsedTicks;

int optimizedCalculations = forecaster.CalculationCount;

forecaster.Reset();

sw.Restart();

var iterativeResult = forecaster.PredictFutureValue\_Iterative(

testInvestment.InitialAmount, testInvestment.GrowthRate, testYears);

sw.Stop();

long iterativeTime = sw.ElapsedTicks;

int iterativeCalculations = forecaster.CalculationCount;

Console.WriteLine($"Basic Recursive: ${basicResult:F2} | {basicCalculations,3} calculations | {basicTime,6} ticks");

Console.WriteLine($"Optimized Recursive: ${optimizedResult:F2} | {optimizedCalculations,3} calculations | {optimizedTime,6} ticks");

Console.WriteLine($"Iterative: ${iterativeResult:F2} | {iterativeCalculations,3} calculations | {iterativeTime,6} ticks");

Console.WriteLine("\n=== COMPOUND INTEREST EXAMPLE ===");

Console.WriteLine("$5,000 invested at 6% annual rate, compounded quarterly for 10 years:");

forecaster.Reset();

var compoundResult = forecaster.CalculateCompoundInterest(5000, 0.06, 4, 10);

Console.WriteLine($"Final Amount: ${compoundResult:F2}");

Console.WriteLine($"Total Interest: ${compoundResult - 5000:F2}");

Console.WriteLine("\n=== TIME COMPLEXITY ANALYSIS ===");

Console.WriteLine("Basic Recursive Approach:");

Console.WriteLine("• Time Complexity: O(n) where n = number of years");

Console.WriteLine("• Space Complexity: O(n) due to recursive call stack");

Console.WriteLine("• Each year requires one recursive call");

Console.WriteLine();

Console.WriteLine("Optimized Recursive (with Memoization):");

Console.WriteLine("• Time Complexity: O(n) for first calculation, O(1) for subsequent same queries");

Console.WriteLine("• Space Complexity: O(n) for memoization storage + call stack");

Console.WriteLine("• Avoids recalculating same values");

Console.WriteLine();

Console.WriteLine("Iterative Approach:");

Console.WriteLine("• Time Complexity: O(n)");

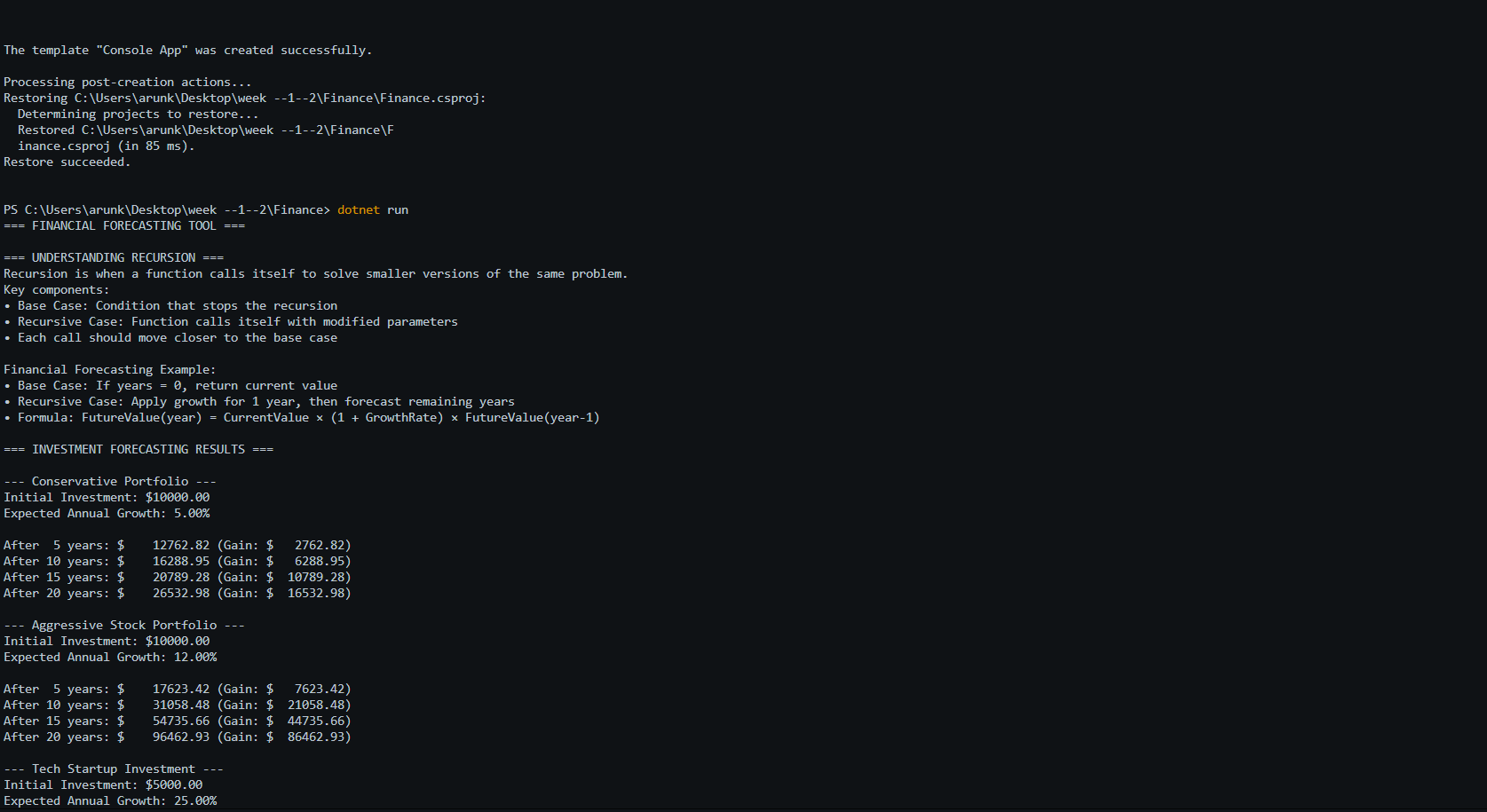
Console.WriteLine("• Space Complexity: O(1) - most memory efficient");

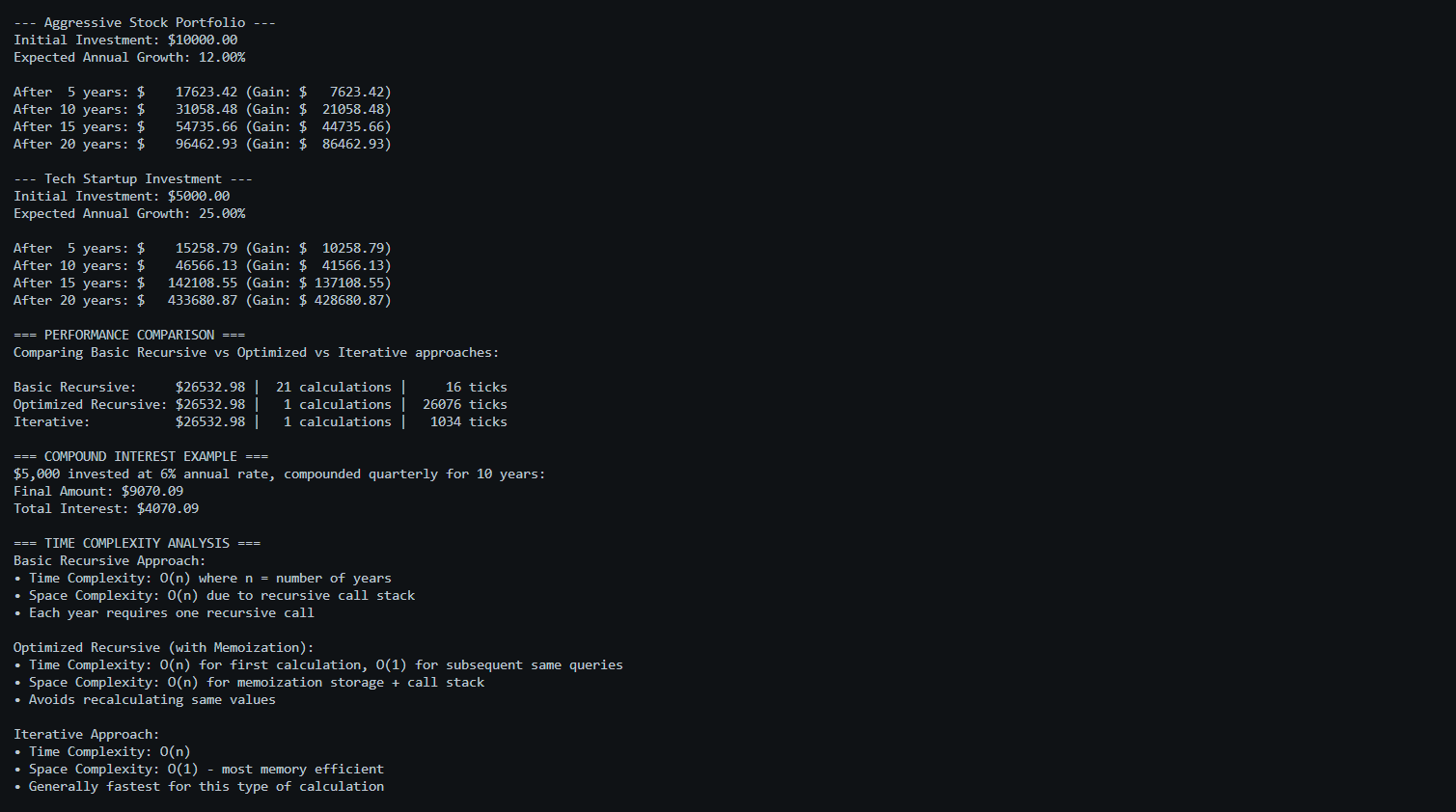
Console.WriteLine("• Generally fastest for this type of calculation");

Console.WriteLine();

}

**OUTPUT**

****

****