**WEEK ONE ASSIGNMENTS**

**Design principles & Patterns**

#### ****Exercise 1: Singleton Pattern Implementation****

**Scenario**:  
In your application, a logging utility must maintain a single, consistent instance throughout the program's execution to ensure uniform logging behavior.

**Steps**:

**Create a C# Project**:

Name the project SingletonPatternExample.

**Define the Singleton Class**:

Build a class called Logger that includes a private static reference to its own type.

Restrict the constructor from external access using the private keyword.

Provide a public static method to retrieve the unique instance.

**Develop the Singleton Logic**:

Write the logic that ensures the class conforms to the Singleton design principle.

**Test the Singleton**:

Create a testing class that validates only one instance is ever created and shared.

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

using System;

using System.Collections.Generic;

// Logger class implementing Singleton Design Pattern

public class Logger

{

// Private static instance variable

private static Logger \_instance = null;

// Private constructor to prevent external instantiation

private Logger()

{

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

LogEntries = new List<string>();

}

// Public method to access the single instance

public static Logger GetInstance()

{

if (\_instance == null)

{

\_instance = new Logger(); // Lazy initialization

}

return \_instance;

}

// List to store log messages

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

// Method to log general 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 log entries

public void DisplayAllLogs()

{

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

if (LogEntries.Count == 0)

{

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

}

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 uniqueness)

public int GetInstanceId()

{

return this.GetHashCode();

}

}

// Testing the Singleton Behavior

public class SingletonTest

{

public static void TestSingletonPattern()

{

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

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

Logger logger1 = Logger.GetInstance();

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

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

Logger logger2 = Logger.GetInstance();

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

Console.WriteLine("\n3. Verifying both instances are identical...");

bool isSame = ReferenceEquals(logger1, logger2);

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

Console.WriteLine("\n4. Logging via both instances...");

logger1.LogMessage("First message via logger1.");

logger2.LogMessage("Second message via logger2.");

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

logger1.DisplayAllLogs();

Console.WriteLine(" Logs from logger2:");

logger2.DisplayAllLogs();

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

Logger logger3 = Logger.GetInstance();

Logger logger4 = Logger.GetInstance();

Logger logger5 = Logger.GetInstance();

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

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

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

bool allSame = ReferenceEquals(logger1, logger3) && ReferenceEquals(logger1, logger4) && ReferenceEquals(logger1, logger5);

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

}

}

// Logger in Action with Services

public class UserService

{

private Logger logger;

public UserService()

{

logger = Logger.GetInstance();

}

public void CreateUser(string username)

{

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

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

}

public void DeleteUser(string username)

{

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

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: Initiating payment of ${amount}");

if (amount <= 0)

{

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

return;

}

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

}

}

//Main Program

public class Program

{

public static void Main()

{

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

// Introduction

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

Console.WriteLine("Purpose: Ensure a class has only one instance and provide a global point of access to it.");

Console.WriteLine("Ideal for logging, configuration, database connections.\n");

// Pattern Structure

Console.WriteLine("Key Components:");

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

Console.WriteLine("• Private constructor to prevent direct creation");

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

// Test singleton

SingletonTest.TestSingletonPattern();

// Practical Usage

Console.WriteLine("\n=== REAL-WORLD USAGE EXAMPLE ===");

UserService userService = new UserService();

PaymentService paymentService = new PaymentService();

userService.CreateUser("john\_doe");

userService.CreateUser("jane\_smith");

paymentService.ProcessPayment(99.99m);

paymentService.ProcessPayment(-10.00m); // Invalid

paymentService.ProcessPayment(149.50m);

userService.DeleteUser("john\_doe");

Logger finalLogger = Logger.GetInstance();

finalLogger.DisplayAllLogs();

// Benefits summary

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

Console.WriteLine("• Centralized access to one instance");

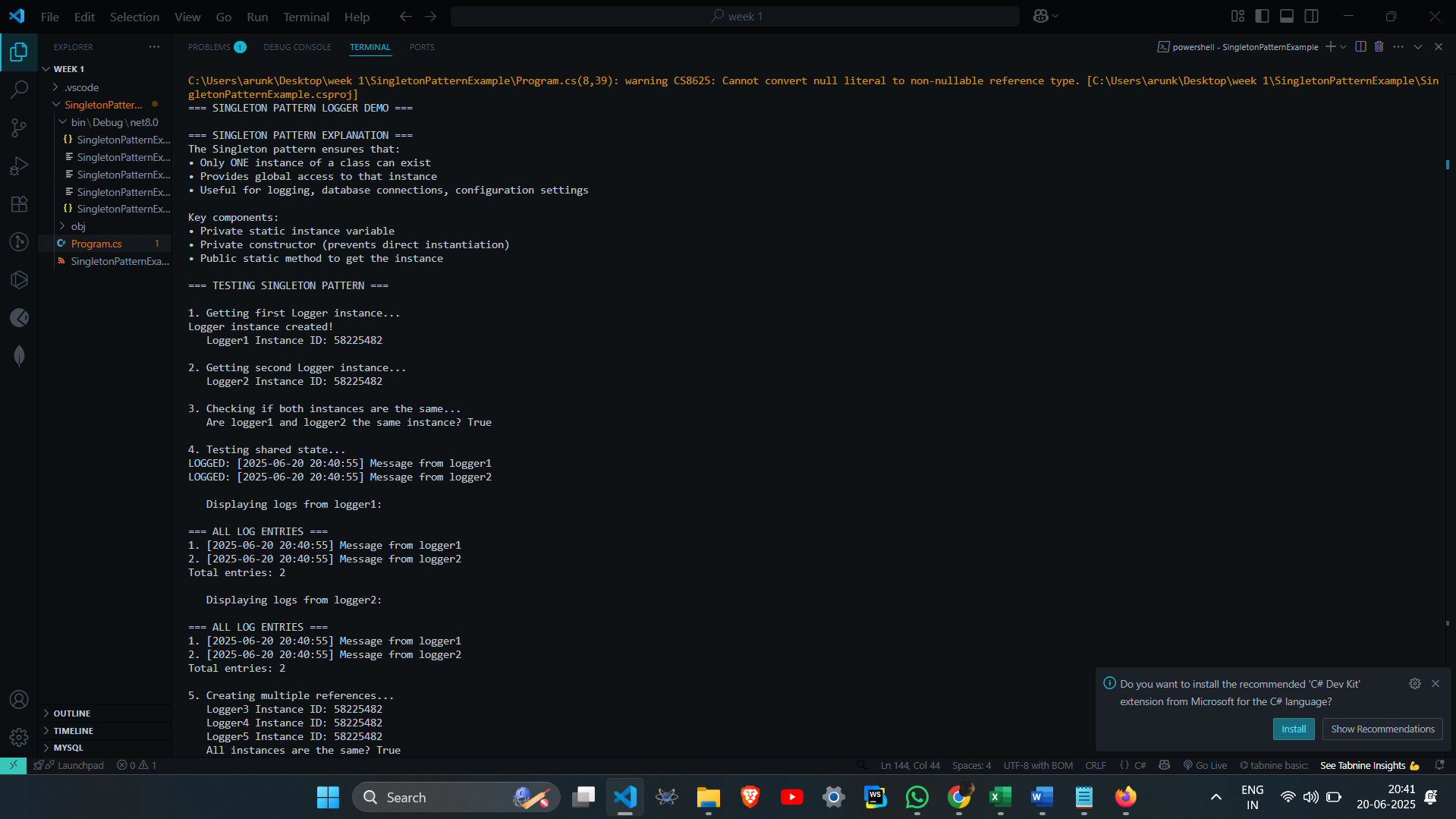
Console.WriteLine("• Saves memory by avoiding redundant instances");

Console.WriteLine("• Consistent state across multiple components");

}

}

**OUTPUT:**



### ****Exercise 2: Factory Method Pattern****

**Scenario:**  
Build a document system that can generate various types like Word, PDF, and Excel using the Factory Method Pattern.

### ****Steps:****

**Set Up Project:**

Create a Java project named FactoryMethodPatternExample.

**Define Interface:**

Create a Document interface or abstract class for common document behavior.

**Create Document Classes:**

Implement WordDocument, PdfDocument, and ExcelDocument classes that extend or implement the base.

**Build Factory Classes:**

Create an abstract DocumentFactory with a method createDocument().

Extend it with specific factories like WordDocumentFactory, PdfDocumentFactory, etc.

**Test It:**

Write a test class to create and use different documents via their respective factories.

**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();

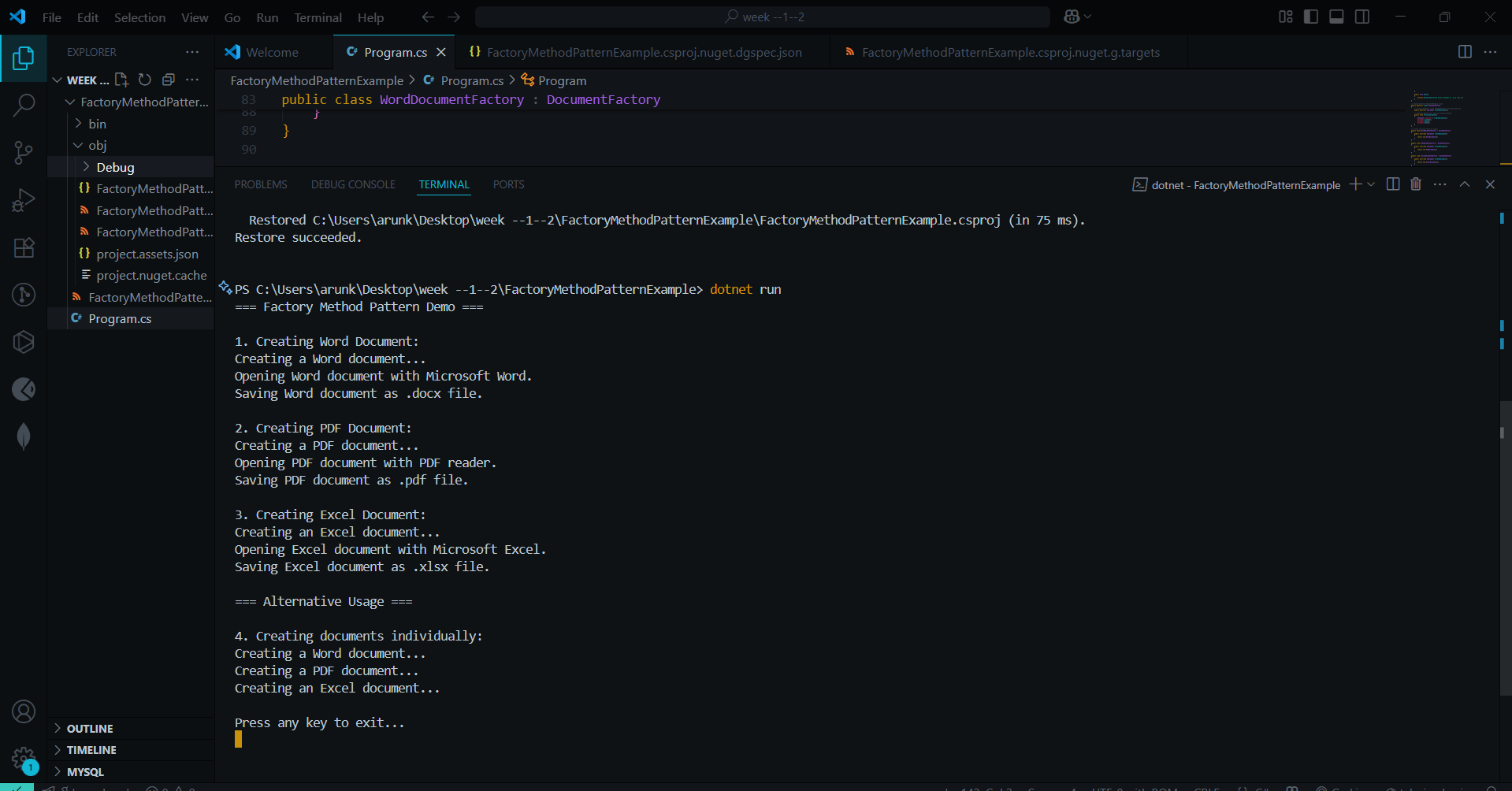
Console.WriteLine("\nPress any key to exit...");

Console.ReadKey();

}

}

**OUTPUT**



**Algorithms\_Data Structures**

### ****Exercise 2: Optimizing Search for E-commerce Platform****

**Scenario:**  
You are developing a product search feature for an e-commerce application and need to ensure quick and efficient search performance.

### ****Steps:****

**Grasp Algorithm Efficiency:**

Explain Big O notation and how it evaluates algorithm performance.

Clarify best, average, and worst-case search scenarios.

**Create Product Structure:**

Define a Product class with fields like productId, productName, and category.

**Implement Search Methods:**

Use linear search on an unsorted array.

Use binary search on a sorted array.

**Evaluate Performance:**

Compare both methods in terms of time complexity.

Determine the more efficient search method based on use case.

**[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");

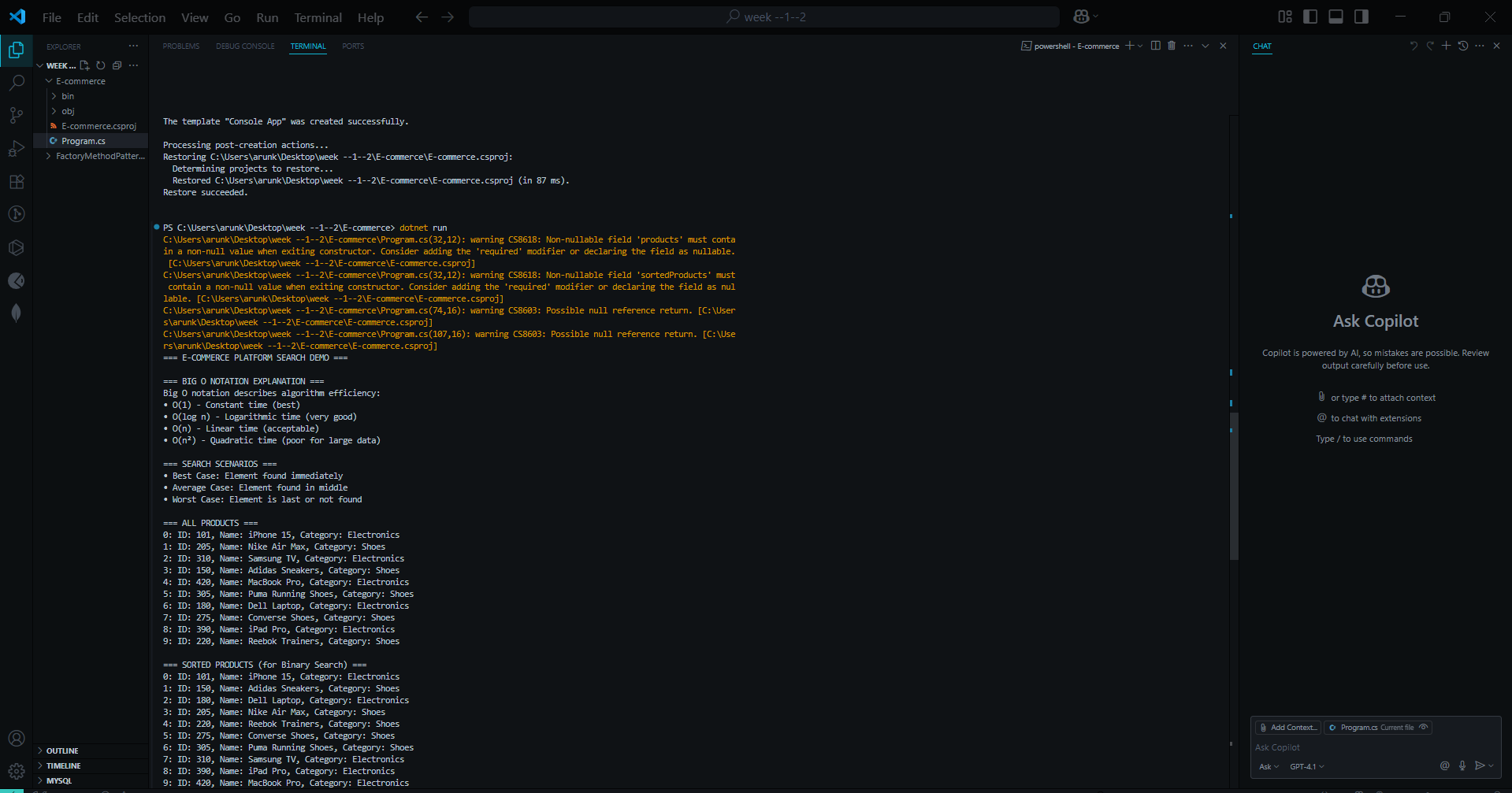
Console.WriteLine("• Implement search engines like Elasticsearch for complex queries");

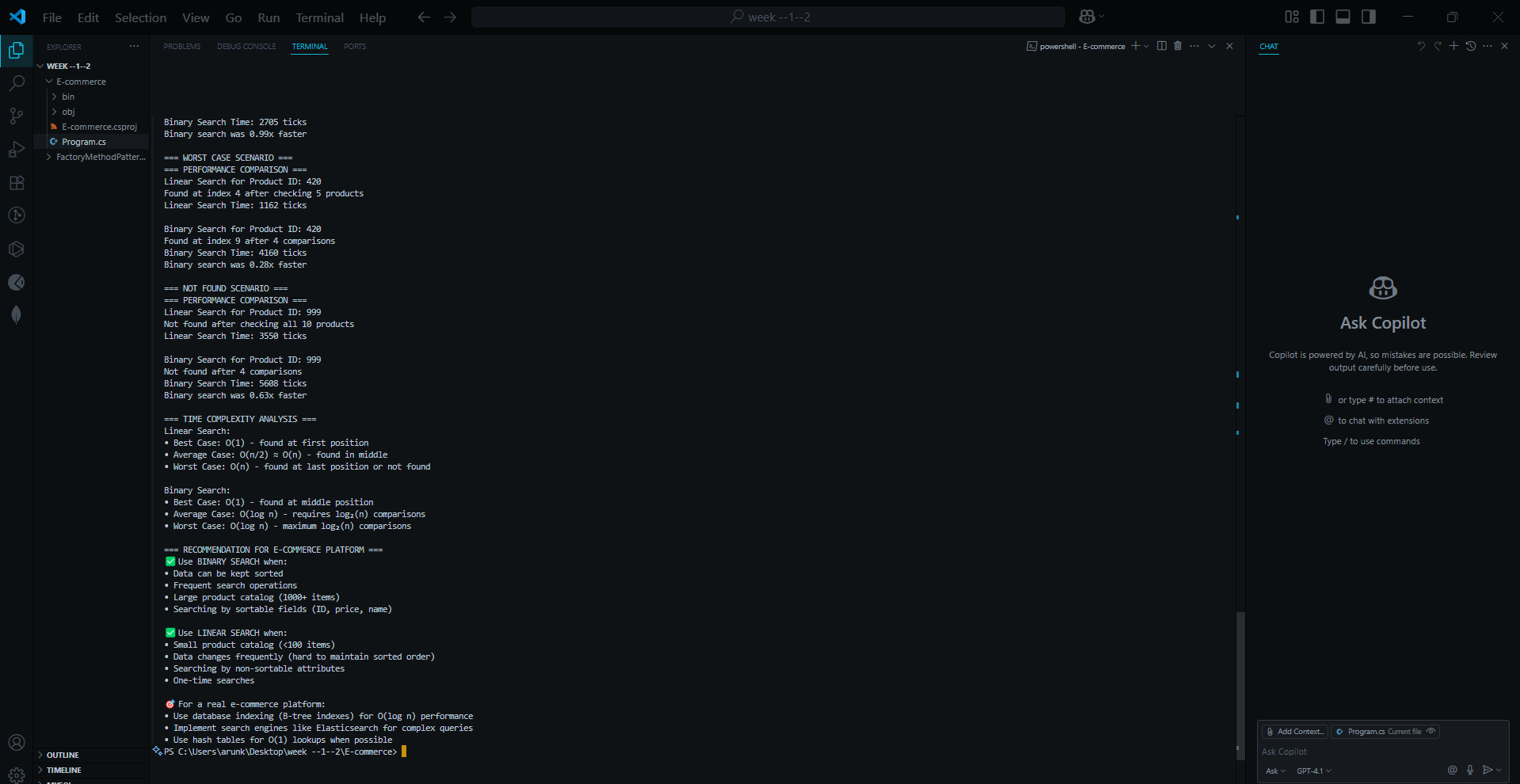
Console.WriteLine("• Use hash tables for O(1) lookups when possible");

}

}

**OUTPUT**





**Exercise 7: Financial Forecasting**

### ****Exercise 7: Future Value Prediction****

**Scenario**:  
Build a tool that forecasts future values based on historical growth, using recursive approaches.

**Steps**:

**Understand Recursive Thinking**:

Recursion solves complex problems by breaking them into smaller, identical subproblems.

Explain base and recursive cases in this context.

**Set Up Forecasting Function**:

Implement a function to recursively compute future value from growth rate and time.

**Implement the Recursive Forecasting**:

Code a recursive model to project values over multiple years based on a starting value and growth rate.

**Discuss Performance**:

Analyze the runtime complexity of your solution.

Explain optimization strategies like memoization to reduce redundant calculations.

[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)");

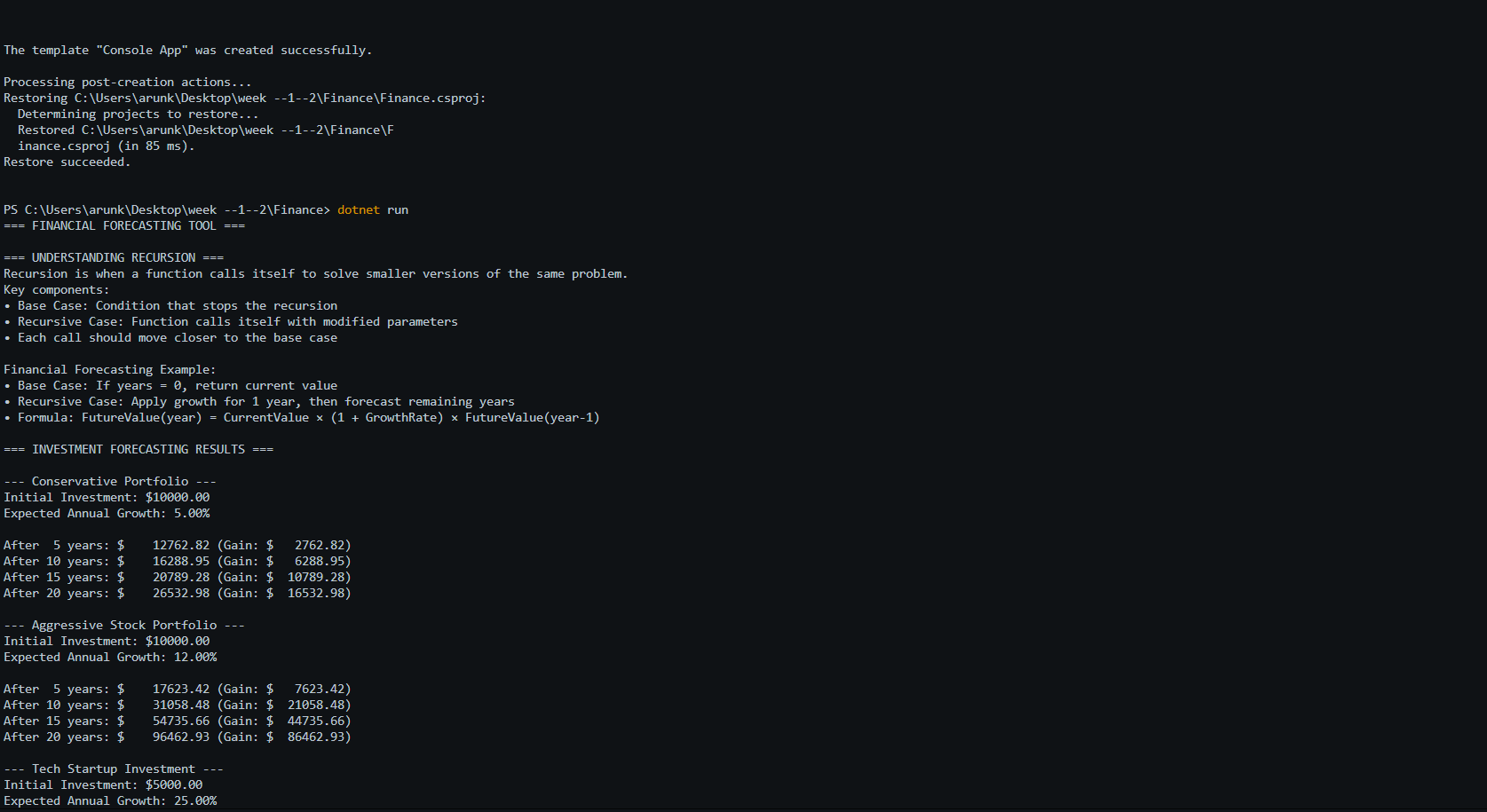
Console.WriteLine("• Space Complexity: O(1) - most memory efficient");

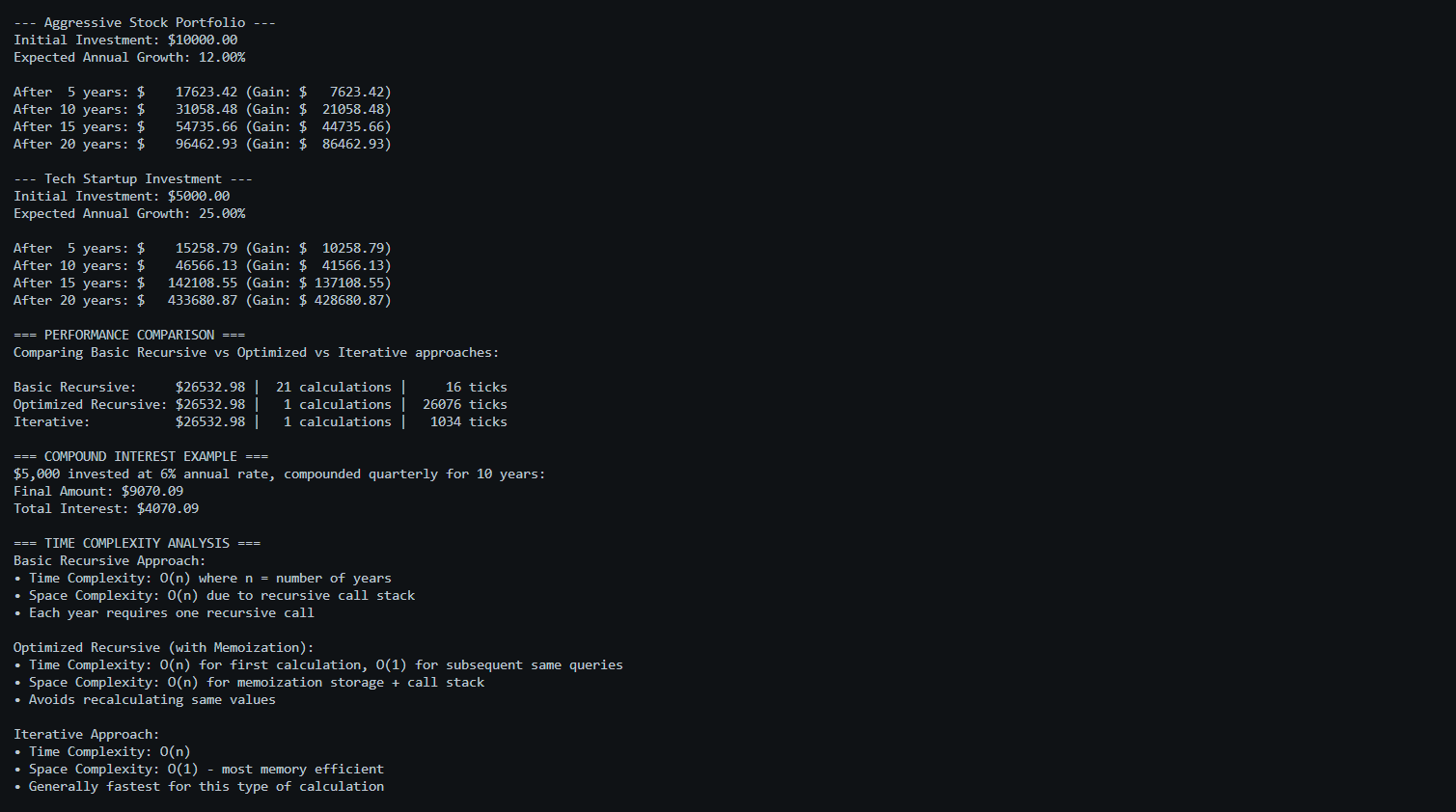
Console.WriteLine("• Generally fastest for this type of calculation");

Console.WriteLine();

}

**OUTPUT**

****

****