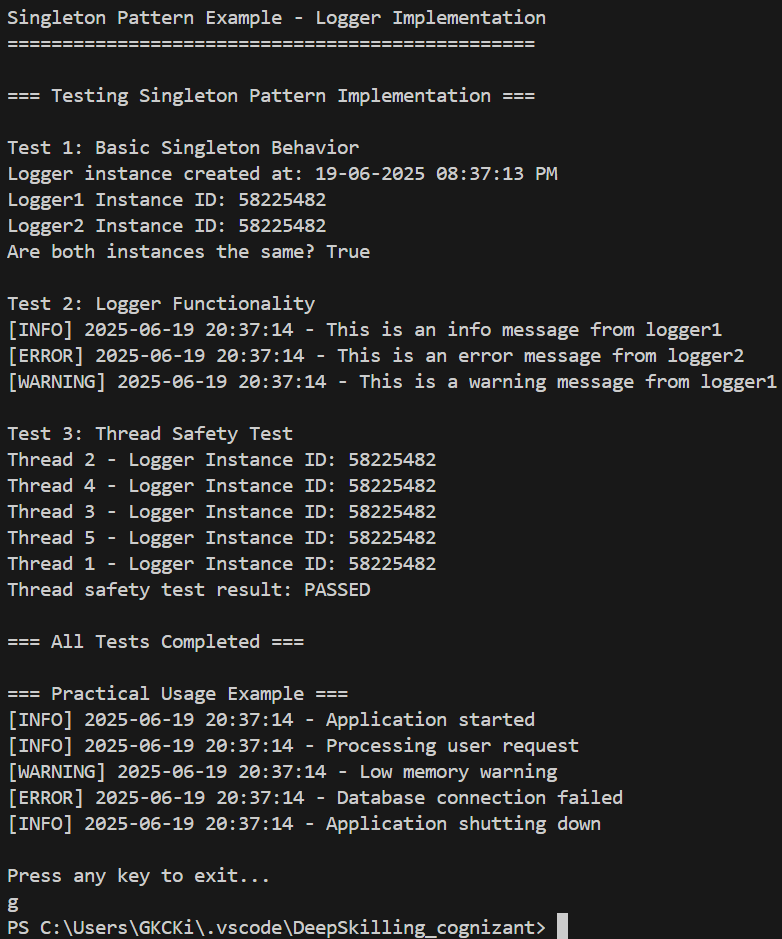
Cognizant Week 1 Assignment:

Handson – 1:

using System;  
using System.Threading;  
using System.Threading.Tasks;  
  
namespace SingletonPatternExample  
{  
    public sealed class Logger  
    {  
        private static readonly Lazy<Logger> \_instance = new Lazy<Logger>(() => new Logger());  
          
        private Logger()  
        {  
            Console.WriteLine("Logger instance created at: " + DateTime.Now);  
        }  
          
        public static Logger Instance => \_instance.Value;  
          
        public void LogInfo(string message)  
        {  
            Console.WriteLine($"[INFO] {DateTime.Now:yyyy-MM-dd HH:mm:ss} - {message}");  
        }  
          
        public void LogError(string message)  
        {  
            Console.WriteLine($"[ERROR] {DateTime.Now:yyyy-MM-dd HH:mm:ss} - {message}");  
        }  
          
        public void LogWarning(string message)  
        {  
            Console.WriteLine($"[WARNING] {DateTime.Now:yyyy-MM-dd HH:mm:ss} - {message}");  
        }  
          
        public string GetInstanceId()  
        {  
            return this.GetHashCode().ToString();  
        }  
    }  
      
    public class SingletonTest  
    {  
        public static void TestSingletonImplementation()  
        {  
            Console.WriteLine("=== Testing Singleton Pattern Implementation ===\n");  
              
            Console.WriteLine("Test 1: Basic Singleton Behavior");  
            Logger logger1 = Logger.Instance;  
            Logger logger2 = Logger.Instance;  
              
            Console.WriteLine($"Logger1 Instance ID: {logger1.GetInstanceId()}");  
            Console.WriteLine($"Logger2 Instance ID: {logger2.GetInstanceId()}");  
            Console.WriteLine($"Are both instances the same? {ReferenceEquals(logger1, logger2)}");  
              
            Console.WriteLine("\nTest 2: Logger Functionality");  
            logger1.LogInfo("This is an info message from logger1");  
            logger2.LogError("This is an error message from logger2");  
            logger1.LogWarning("This is a warning message from logger1");  
              
            Console.WriteLine("\nTest 3: Thread Safety Test");  
            TestThreadSafety();  
              
            Console.WriteLine("\n=== All Tests Completed ===");  
        }  
          
        private static void TestThreadSafety()  
        {  
            const int numberOfThreads = 5;  
            Logger[] loggerInstances = new Logger[numberOfThreads];  
            Task[] tasks = new Task[numberOfThreads];  
              
            for (int i = 0; i < numberOfThreads; i++)  
            {  
                int threadIndex = i;  
                tasks[i] = Task.Run(() =>  
                {  
                    Thread.Sleep(new Random().Next(1, 100));  
                    loggerInstances[threadIndex] = Logger.Instance;  
                    Console.WriteLine($"Thread {threadIndex + 1} - Logger Instance ID: {loggerInstances[threadIndex].GetInstanceId()}");  
                });  
            }  
              
            Task.WaitAll(tasks);  
              
            bool allInstancesAreSame = true;  
            for (int i = 1; i < numberOfThreads; i++)  
            {  
                if (!ReferenceEquals(loggerInstances[0], loggerInstances[i]))  
                {  
                    allInstancesAreSame = false;  
                    break;  
                }  
            }  
              
            Console.WriteLine($"Thread safety test result: {(allInstancesAreSame ? "PASSED" : "FAILED")}");  
        }  
    }  
      
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Console.WriteLine("Singleton Pattern Example - Logger Implementation");  
            Console.WriteLine("================================================\n");  
              
            SingletonTest.TestSingletonImplementation();  
              
            Console.WriteLine("\n=== Practical Usage Example ===");  
              
            Logger appLogger = Logger.Instance;  
            appLogger.LogInfo("Application started");  
            appLogger.LogInfo("Processing user request");  
            appLogger.LogWarning("Low memory warning");  
            appLogger.LogError("Database connection failed");  
            appLogger.LogInfo("Application shutting down");  
              
            Console.WriteLine("\nPress any key to exit...");  
            Console.ReadKey();  
        }  
    }  
}

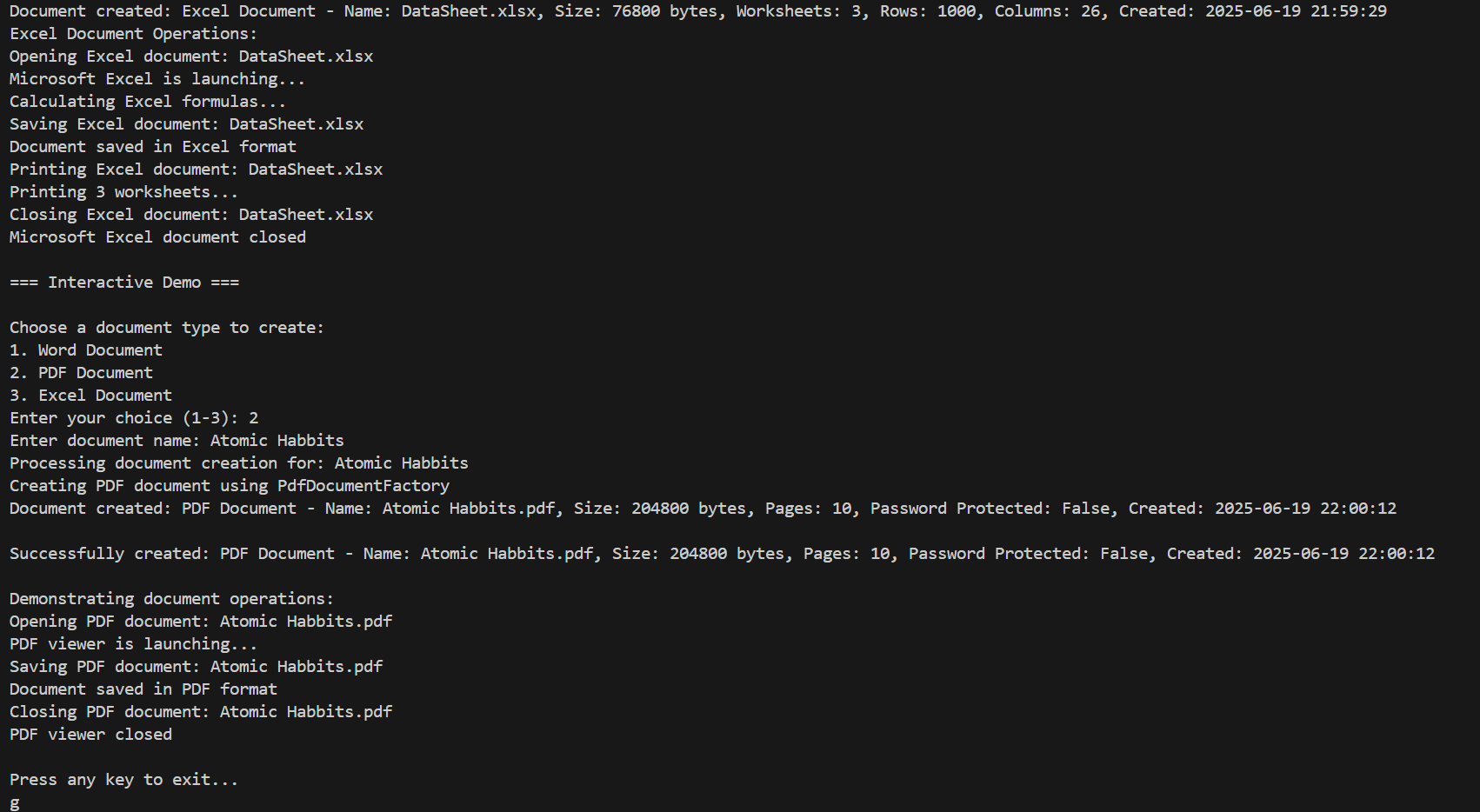
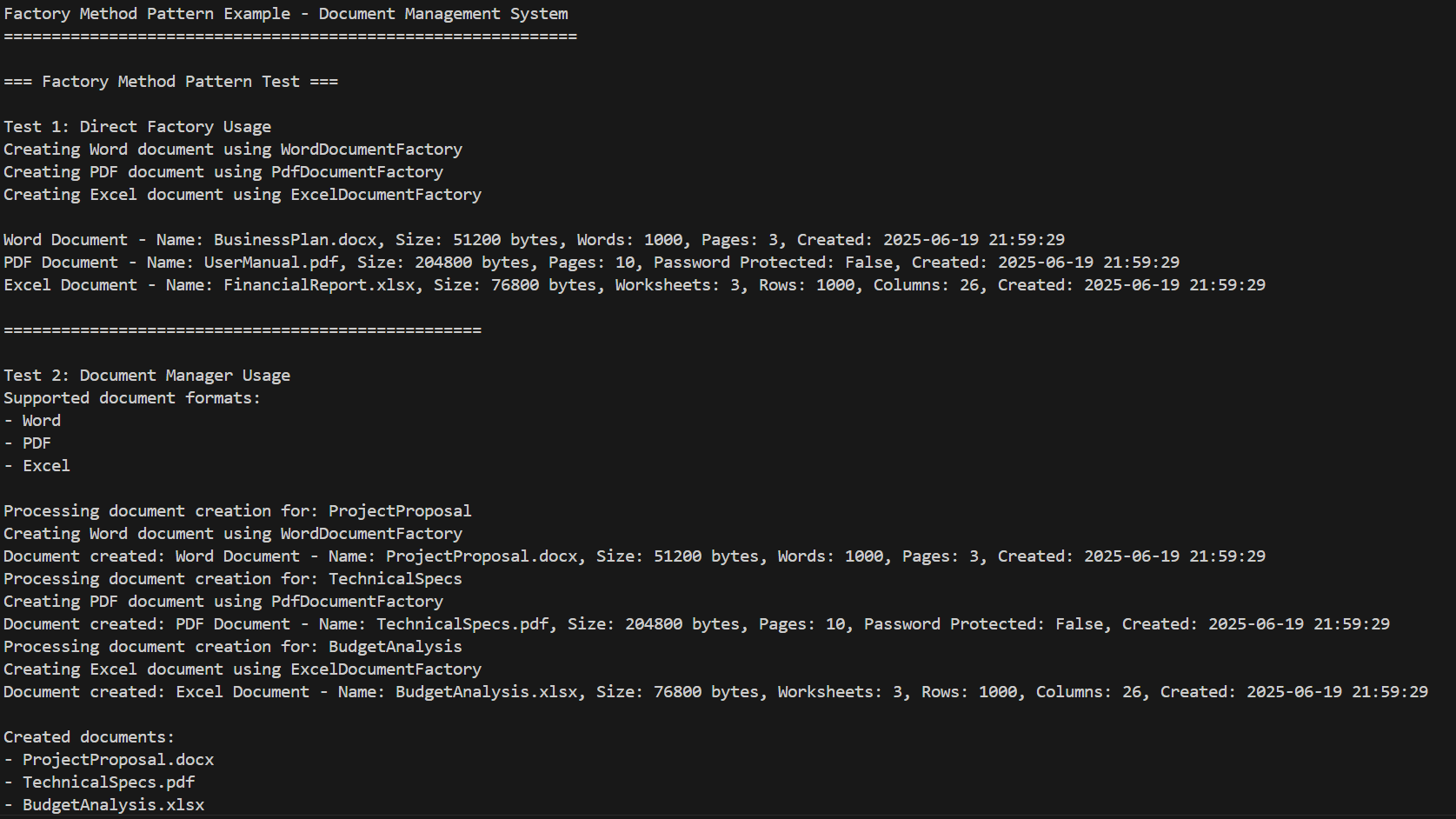
Output:



Handson – 2:

using System;  
using System.Collections.Generic;  
  
namespace FactoryMethodPatternExample  
{  
    public enum DocumentType  
    {  
        Word,  
        PDF,  
        Excel  
    }  
  
    public abstract class Document  
    {  
        public string Name { get; protected set; }  
        public string Extension { get; protected set; }  
        public DateTime CreatedDate { get; protected set; }  
        public long Size { get; protected set; }  
  
        protected Document(string name)  
        {  
            Name = name;  
            Extension = string.Empty;  
            CreatedDate = DateTime.Now;  
        }  
        public abstract void Open();  
        public abstract void Save();  
        public abstract void Close();  
        public abstract void Print();  
        public abstract string GetDocumentInfo();  
    }  
  
    public class WordDocument : Document  
    {  
        public int WordCount { get; private set; }  
        public int PageCount { get; private set; }  
  
        public WordDocument(string name) : base(name)  
        {  
            Extension = ".docx";  
            Size = 1024 \* 50;  
            WordCount = 1000;  
            PageCount = 3;  
        }  
  
        public override void Open()  
        {  
            Console.WriteLine($"Opening Word document: {Name}{Extension}");  
            Console.WriteLine("Microsoft Word is launching...");  
        }  
  
        public override void Save()  
        {  
            Console.WriteLine($"Saving Word document: {Name}{Extension}");  
            Console.WriteLine("Document saved in Word format");  
        }  
  
        public override void Close()  
        {  
            Console.WriteLine($"Closing Word document: {Name}{Extension}");  
            Console.WriteLine("Microsoft Word document closed");  
        }  
  
        public override void Print()  
        {  
            Console.WriteLine($"Printing Word document: {Name}{Extension}");  
            Console.WriteLine($"Printing {PageCount} pages...");  
        }  
  
        public override string GetDocumentInfo()  
        {  
            return $"Word Document - Name: {Name}{Extension}, Size: {Size} bytes, " +  
                   $"Words: {WordCount}, Pages: {PageCount}, Created: {CreatedDate:yyyy-MM-dd HH:mm:ss}";  
        }  
  
        public void CheckSpelling()  
        {  
            Console.WriteLine("Running spell check on Word document...");  
        }  
    }  
  
    public class PdfDocument : Document  
    {  
        public int PageCount { get; private set; }  
        public bool IsPasswordProtected { get; private set; }  
  
        public PdfDocument(string name) : base(name)  
        {  
            Extension = ".pdf";  
            Size = 1024 \* 200;  
            PageCount = 10;  
            IsPasswordProtected = false;  
        }  
  
        public override void Open()  
        {  
            Console.WriteLine($"Opening PDF document: {Name}{Extension}");  
            Console.WriteLine("PDF viewer is launching...");  
        }  
  
        public override void Save()  
        {  
            Console.WriteLine($"Saving PDF document: {Name}{Extension}");  
            Console.WriteLine("Document saved in PDF format");  
        }  
  
        public override void Close()  
        {  
            Console.WriteLine($"Closing PDF document: {Name}{Extension}");  
            Console.WriteLine("PDF viewer closed");  
        }  
  
        public override void Print()  
        {  
            Console.WriteLine($"Printing PDF document: {Name}{Extension}");  
            Console.WriteLine($"Printing {PageCount} pages in high quality...");  
        }  
  
        public override string GetDocumentInfo()  
        {  
            return $"PDF Document - Name: {Name}{Extension}, Size: {Size} bytes, " +  
                   $"Pages: {PageCount}, Password Protected: {IsPasswordProtected}, " +  
                   $"Created: {CreatedDate:yyyy-MM-dd HH:mm:ss}";  
        }  
  
        public void SetPassword(string password)  
        {  
            IsPasswordProtected = !string.IsNullOrEmpty(password);  
            Console.WriteLine($"PDF password protection: {(IsPasswordProtected ? "Enabled" : "Disabled")}");  
        }  
    }  
  
    public class ExcelDocument : Document  
    {  
        public int WorksheetCount { get; private set; }  
        public int RowCount { get; private set; }  
        public int ColumnCount { get; private set; }  
  
        public ExcelDocument(string name) : base(name)  
        {  
            Extension = ".xlsx";  
            Size = 1024 \* 75;  
            WorksheetCount = 3;  
            RowCount = 1000;  
            ColumnCount = 26;  
        }  
  
        public override void Open()  
        {  
            Console.WriteLine($"Opening Excel document: {Name}{Extension}");  
            Console.WriteLine("Microsoft Excel is launching...");  
        }  
  
        public override void Save()  
        {  
            Console.WriteLine($"Saving Excel document: {Name}{Extension}");  
            Console.WriteLine("Document saved in Excel format");  
        }  
  
        public override void Close()  
        {  
            Console.WriteLine($"Closing Excel document: {Name}{Extension}");  
            Console.WriteLine("Microsoft Excel document closed");  
        }  
  
        public override void Print()  
        {  
            Console.WriteLine($"Printing Excel document: {Name}{Extension}");  
            Console.WriteLine($"Printing {WorksheetCount} worksheets...");  
        }  
  
        public override string GetDocumentInfo()  
        {  
            return $"Excel Document - Name: {Name}{Extension}, Size: {Size} bytes, " +  
                   $"Worksheets: {WorksheetCount}, Rows: {RowCount}, Columns: {ColumnCount}, " +  
                   $"Created: {CreatedDate:yyyy-MM-dd HH:mm:ss}";  
        }  
  
        public void CalculateFormulas()  
        {  
            Console.WriteLine("Calculating Excel formulas...");  
        }  
    }  
  
    public abstract class DocumentFactory  
    {  
        public abstract Document CreateDocument(string name);  
  
        public Document ProcessDocument(string name)  
        {  
            Console.WriteLine($"Processing document creation for: {name}");  
              
            Document document = CreateDocument(name);  
              
            Console.WriteLine($"Document created: {document.GetDocumentInfo()}");  
              
            return document;  
        }  
    }  
  
    public class WordDocumentFactory : DocumentFactory  
    {  
        public override Document CreateDocument(string name)  
        {  
            Console.WriteLine("Creating Word document using WordDocumentFactory");  
            return new WordDocument(name);  
        }  
    }  
  
    public class PdfDocumentFactory : DocumentFactory  
    {  
        public override Document CreateDocument(string name)  
        {  
            Console.WriteLine("Creating PDF document using PdfDocumentFactory");  
            return new PdfDocument(name);  
        }  
    }  
  
    public class ExcelDocumentFactory : DocumentFactory  
    {  
        public override Document CreateDocument(string name)  
        {  
            Console.WriteLine("Creating Excel document using ExcelDocumentFactory");  
            return new ExcelDocument(name);  
        }  
    }  
  
    public class DocumentManager  
    {  
        private readonly Dictionary<DocumentType, DocumentFactory> \_factories;  
  
        public DocumentManager()  
        {  
            \_factories = new Dictionary<DocumentType, DocumentFactory>  
            {  
                { DocumentType.Word, new WordDocumentFactory() },  
                { DocumentType.PDF, new PdfDocumentFactory() },  
                { DocumentType.Excel, new ExcelDocumentFactory() }  
            };  
        }  
  
        public Document CreateDocument(DocumentType type, string name)  
        {  
            if (\_factories.TryGetValue(type, out var factory))  
            {  
                return factory.ProcessDocument(name);  
            }  
              
            throw new ArgumentException($"Unsupported document type: {type}");  
        }  
  
        public void DisplaySupportedFormats()  
        {  
            Console.WriteLine("Supported document formats:");  
            foreach (var format in \_factories.Keys)  
            {  
                Console.WriteLine($"- {format}");  
            }  
        }  
    }  
  
    public class FactoryMethodTest  
    {  
        public static void RunTests()  
        {  
            Console.WriteLine("=== Factory Method Pattern Test ===\n");  
  
            Console.WriteLine("Test 1: Direct Factory Usage");  
            TestDirectFactoryUsage();  
  
            Console.WriteLine("\n" + new string('=', 50) + "\n");  
  
            Console.WriteLine("Test 2: Document Manager Usage");  
            TestDocumentManager();  
  
            Console.WriteLine("\n" + new string('=', 50) + "\n");  
  
            Console.WriteLine("Test 3: Document Operations");  
            TestDocumentOperations();  
        }  
  
        private static void TestDirectFactoryUsage()  
        {  
            DocumentFactory wordFactory = new WordDocumentFactory();  
            DocumentFactory pdfFactory = new PdfDocumentFactory();  
            DocumentFactory excelFactory = new ExcelDocumentFactory();  
  
            Document wordDoc = wordFactory.CreateDocument("BusinessPlan");  
            Document pdfDoc = pdfFactory.CreateDocument("UserManual");  
            Document excelDoc = excelFactory.CreateDocument("FinancialReport");  
  
            Console.WriteLine($"\n{wordDoc.GetDocumentInfo()}");  
            Console.WriteLine($"{pdfDoc.GetDocumentInfo()}");  
            Console.WriteLine($"{excelDoc.GetDocumentInfo()}");  
        }  
  
        private static void TestDocumentManager()  
        {  
            DocumentManager manager = new DocumentManager();  
              
            manager.DisplaySupportedFormats();  
            Console.WriteLine();  
  
            Document[] documents = {  
                manager.CreateDocument(DocumentType.Word, "ProjectProposal"),  
                manager.CreateDocument(DocumentType.PDF, "TechnicalSpecs"),  
                manager.CreateDocument(DocumentType.Excel, "BudgetAnalysis")  
            };  
  
            Console.WriteLine("\nCreated documents:");  
            foreach (var doc in documents)  
            {  
                Console.WriteLine($"- {doc.Name}{doc.Extension}");  
            }  
        }  
  
        private static void TestDocumentOperations()  
        {  
            DocumentManager manager = new DocumentManager();  
  
            Document wordDoc = manager.CreateDocument(DocumentType.Word, "TestDocument");  
            Console.WriteLine("\nWord Document Operations:");  
            wordDoc.Open();  
            if (wordDoc is WordDocument wd)  
            {  
                wd.CheckSpelling();  
            }  
            wordDoc.Save();  
            wordDoc.Print();  
            wordDoc.Close();  
  
            Console.WriteLine("\n" + new string('-', 30) + "\n");  
  
            Document pdfDoc = manager.CreateDocument(DocumentType.PDF, "SecureDoc");  
            Console.WriteLine("PDF Document Operations:");  
            pdfDoc.Open();  
            if (pdfDoc is PdfDocument pd)  
            {  
                pd.SetPassword("secret123");  
            }  
            pdfDoc.Save();  
            pdfDoc.Print();  
            pdfDoc.Close();  
  
            Console.WriteLine("\n" + new string('-', 30) + "\n");  
  
            Document excelDoc = manager.CreateDocument(DocumentType.Excel, "DataSheet");  
            Console.WriteLine("Excel Document Operations:");  
            excelDoc.Open();  
            if (excelDoc is ExcelDocument ed)  
            {  
                ed.CalculateFormulas();  
            }  
            excelDoc.Save();  
            excelDoc.Print();  
            excelDoc.Close();  
        }  
    }  
  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Console.WriteLine("Factory Method Pattern Example - Document Management System");  
            Console.WriteLine("============================================================\n");  
  
            try  
            {  
                FactoryMethodTest.RunTests();  
  
                Console.WriteLine("\n=== Interactive Demo ===");  
                InteractiveDemo();  
            }  
            catch (Exception ex)  
            {  
                Console.WriteLine($"Error: {ex.Message}");  
            }  
  
            Console.WriteLine("\nPress any key to exit...");  
            Console.ReadKey();  
        }  
  
        private static void InteractiveDemo()  
        {  
            DocumentManager manager = new DocumentManager();  
              
            Console.WriteLine("\nChoose a document type to create:");  
            Console.WriteLine("1. Word Document");  
            Console.WriteLine("2. PDF Document");  
            Console.WriteLine("3. Excel Document");  
            Console.Write("Enter your choice (1-3): ");  
  
            string input = Console.ReadLine() ?? string.Empty;  
              
            if (int.TryParse(input, out int choice) && choice >= 1 && choice <= 3)  
            {  
                DocumentType type = (DocumentType)(choice - 1);  
                  
                Console.Write("Enter document name: ");  
                string name = Console.ReadLine() ?? string.Empty;  
                  
                if (!string.IsNullOrWhiteSpace(name))  
                {  
                    Document doc = manager.CreateDocument(type, name);  
                    Console.WriteLine($"\nSuccessfully created: {doc.GetDocumentInfo()}");  
                      
                    Console.WriteLine("\nDemonstrating document operations:");  
                    doc.Open();  
                    doc.Save();  
                    doc.Close();  
                }  
                else  
                {  
                    Console.WriteLine("Invalid document name provided.");  
                }  
            }  
            else  
            {  
                Console.WriteLine("Invalid choice. Please select 1, 2, or 3.");  
            }  
        }  
    }  
}

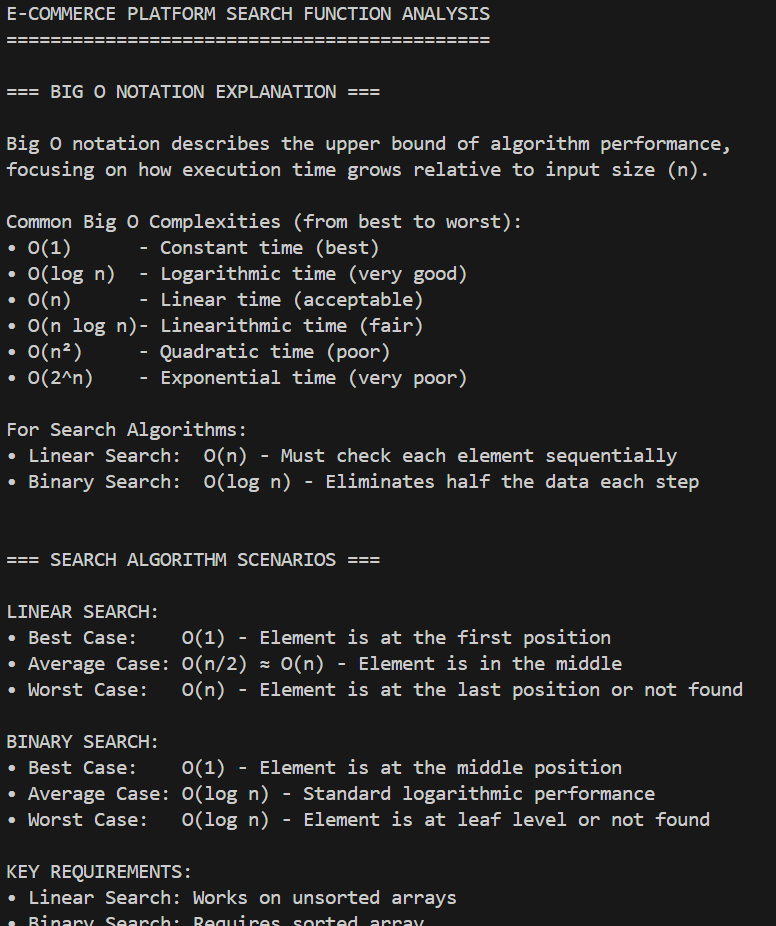
Output:

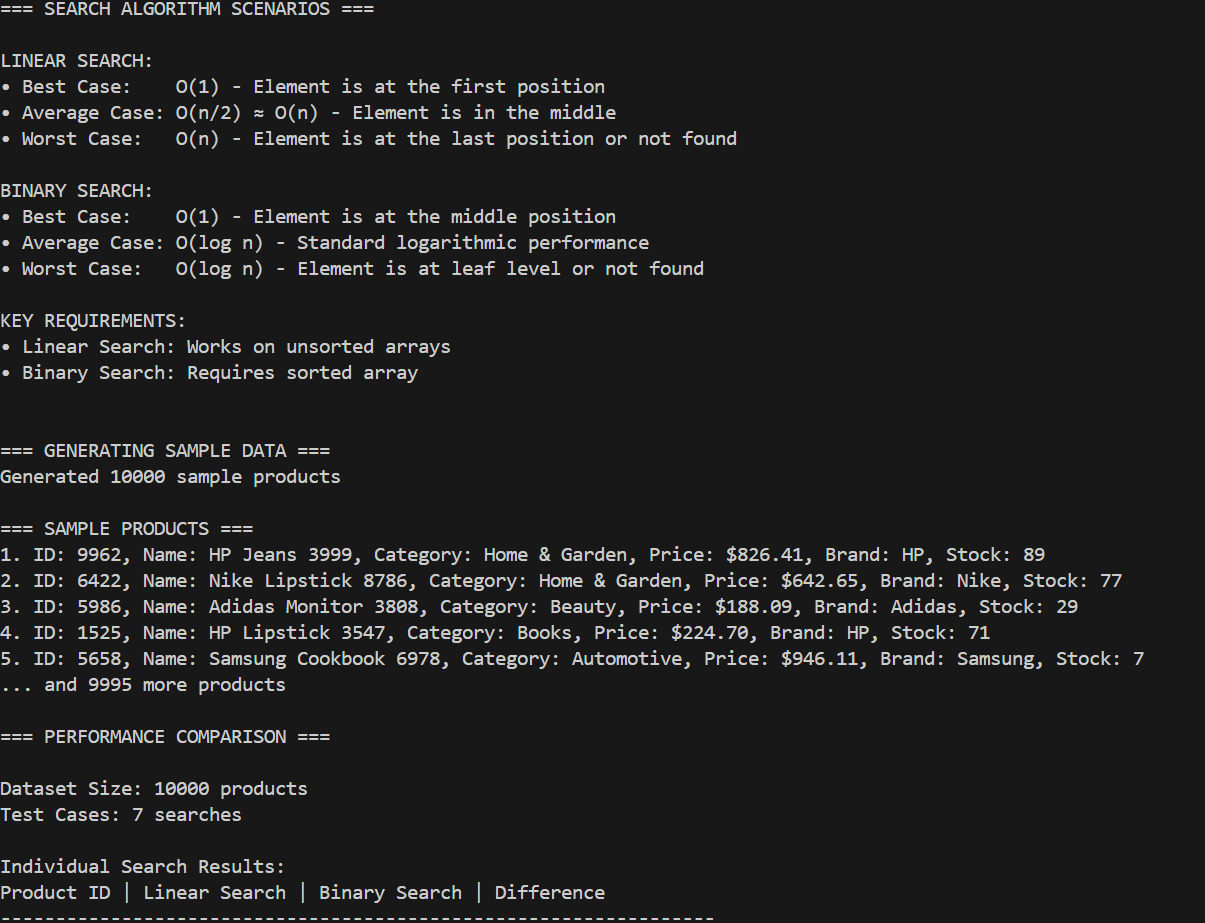


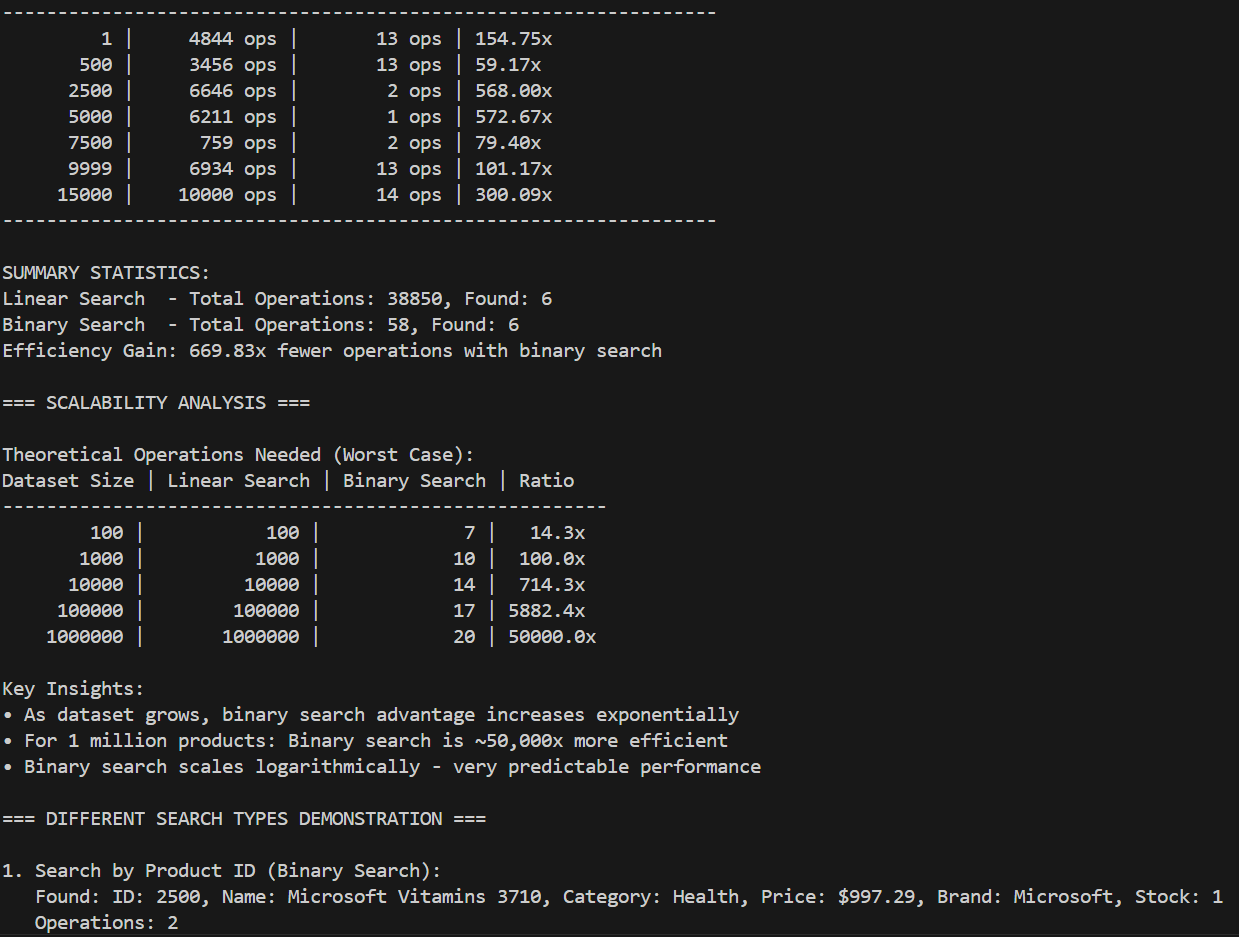
Handson – 3:

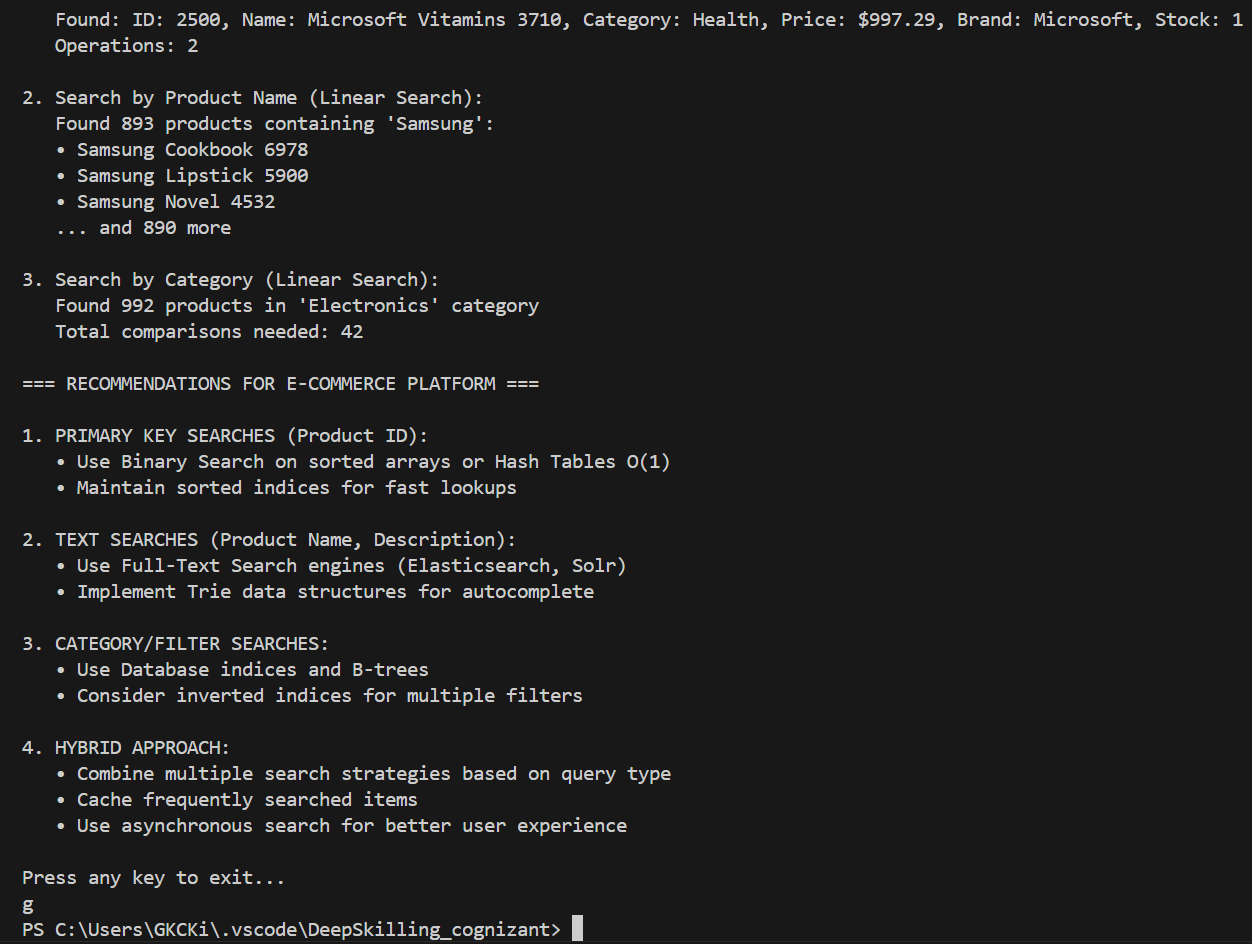
using System;  
using System.Collections.Generic;  
using System.Diagnostics;  
using System.Linq;  
  
namespace EcommerceSearchAlgorithms  
{  
      
    public class Product : IComparable<Product>  
    {  
        public int ProductId { get; set; }  
        public string ProductName { get; set; }  
        public string Category { get; set; }  
        public decimal Price { get; set; }  
        public string Brand { get; set; }  
        public int StockQuantity { get; set; }  
  
        public Product(int productId, string productName, string category, decimal price, string brand, int stockQuantity)  
        {  
            ProductId = productId;  
            ProductName = productName;  
            Category = category;  
            Price = price;  
            Brand = brand;  
            StockQuantity = stockQuantity;  
        }  
  
        // Implement IComparable for sorting (by ProductId)  
        public int CompareTo(Product? other)  
        {  
            if (other == null) return 1;  
            return ProductId.CompareTo(other.ProductId);  
        }  
  
        public override string ToString()  
        {  
            return $"ID: {ProductId}, Name: {ProductName}, Category: {Category}, Price: ${Price:F2}, Brand: {Brand}, Stock: {StockQuantity}";  
        }  
  
        public override bool Equals(object? obj)  
        {  
            if (obj is Product product)  
                return ProductId == product.ProductId;  
            return false;  
        }  
  
        public override int GetHashCode()  
        {  
            return ProductId.GetHashCode();  
        }  
    }  
  
      
    public class SearchResult  
    {  
        public Product? Product { get; set; }  
        public int Index { get; set; }  
        public long ElapsedTicks { get; set; }  
        public int ComparisonsCount { get; set; }  
        public bool Found { get; set; }  
  
        public SearchResult(Product? product, int index, long elapsedTicks, int comparisons, bool found)  
        {  
            Product = product;  
            Index = index;  
            ElapsedTicks = elapsedTicks;  
            ComparisonsCount = comparisons;  
            Found = found;  
        }  
    }  
  
      
    public static class AlgorithmComplexityExplainer  
    {  
        public static void ExplainBigONotation()  
        {  
            Console.WriteLine("=== BIG O NOTATION EXPLANATION ===");  
            Console.WriteLine();  
            Console.WriteLine("Big O notation describes the upper bound of algorithm performance,");  
            Console.WriteLine("focusing on how execution time grows relative to input size (n).");  
            Console.WriteLine();  
            Console.WriteLine("Common Big O Complexities (from best to worst):");  
            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(n log n)- Linearithmic time (fair)");  
            Console.WriteLine("• O(n²)     - Quadratic time (poor)");  
            Console.WriteLine("• O(2^n)    - Exponential time (very poor)");  
            Console.WriteLine();  
            Console.WriteLine("For Search Algorithms:");  
            Console.WriteLine("• Linear Search:  O(n) - Must check each element sequentially");  
            Console.WriteLine("• Binary Search:  O(log n) - Eliminates half the data each step");  
            Console.WriteLine();  
        }  
  
        public static void ExplainSearchScenarios()  
        {  
            Console.WriteLine("=== SEARCH ALGORITHM SCENARIOS ===");  
            Console.WriteLine();  
            Console.WriteLine("LINEAR SEARCH:");  
            Console.WriteLine("• Best Case:    O(1) - Element is at the first position");  
            Console.WriteLine("• Average Case: O(n/2) ≈ O(n) - Element is in the middle");  
            Console.WriteLine("• Worst Case:   O(n) - Element is at the last position or not found");  
            Console.WriteLine();  
            Console.WriteLine("BINARY SEARCH:");  
            Console.WriteLine("• Best Case:    O(1) - Element is at the middle position");  
            Console.WriteLine("• Average Case: O(log n) - Standard logarithmic performance");  
            Console.WriteLine("• Worst Case:   O(log n) - Element is at leaf level or not found");  
            Console.WriteLine();  
            Console.WriteLine("KEY REQUIREMENTS:");  
            Console.WriteLine("• Linear Search: Works on unsorted arrays");  
            Console.WriteLine("• Binary Search: Requires sorted array");  
            Console.WriteLine();  
        }  
    }  
  
      
    public class EcommerceSearchEngine  
    {  
        private Product[] products;  
        private Product[] sortedProducts;  
  
        public EcommerceSearchEngine(Product[] productArray)  
        {  
            products = productArray;  
            sortedProducts = new Product[productArray.Length];  
            Array.Copy(productArray, sortedProducts, productArray.Length);  
            Array.Sort(sortedProducts);   
        }  
  
         
        public SearchResult LinearSearchById(int productId)  
        {  
            Stopwatch stopwatch = Stopwatch.StartNew();  
            int comparisons = 0;  
  
            for (int i = 0; i < products.Length; i++)  
            {  
                comparisons++;  
                if (products[i].ProductId == productId)  
                {  
                    stopwatch.Stop();  
                    return new SearchResult(products[i], i, stopwatch.ElapsedTicks, comparisons, true);  
                }  
            }  
  
            stopwatch.Stop();  
            return new SearchResult(null, -1, stopwatch.ElapsedTicks, comparisons, false);  
        }  
  
          
        public SearchResult BinarySearchById(int productId)  
        {  
            Stopwatch stopwatch = Stopwatch.StartNew();  
            int comparisons = 0;  
            int left = 0;  
            int right = sortedProducts.Length - 1;  
  
            while (left <= right)  
            {  
                comparisons++;  
                int mid = left + (right - left) / 2;  
  
                if (sortedProducts[mid].ProductId == productId)  
                {  
                    stopwatch.Stop();  
                    return new SearchResult(sortedProducts[mid], mid, stopwatch.ElapsedTicks, comparisons, true);  
                }  
  
                if (sortedProducts[mid].ProductId < productId)  
                {  
                    left = mid + 1;  
                }  
                else  
                {  
                    right = mid - 1;  
                }  
            }  
  
            stopwatch.Stop();  
            return new SearchResult(null, -1, stopwatch.ElapsedTicks, comparisons, false);  
        }  
  
          
        public List<SearchResult> LinearSearchByName(string productName)  
        {  
            Stopwatch stopwatch = Stopwatch.StartNew();  
            List<SearchResult> results = new List<SearchResult>();  
            int comparisons = 0;  
  
            for (int i = 0; i < products.Length; i++)  
            {  
                comparisons++;  
                if (products[i].ProductName.IndexOf(productName, StringComparison.OrdinalIgnoreCase) >= 0)  
                {  
                    results.Add(new SearchResult(products[i], i, stopwatch.ElapsedTicks, comparisons, true));  
                }  
            }  
  
            stopwatch.Stop();  
              
              
            foreach (var result in results)  
            {  
                result.ElapsedTicks = stopwatch.ElapsedTicks;  
            }  
  
            if (results.Count == 0)  
            {  
                results.Add(new SearchResult(null, -1, stopwatch.ElapsedTicks, comparisons, false));  
            }  
  
            return results;  
        }  
  
         
        public List<SearchResult> LinearSearchByCategory(string category)  
        {  
            Stopwatch stopwatch = Stopwatch.StartNew();  
            List<SearchResult> results = new List<SearchResult>();  
            int comparisons = 0;  
  
            for (int i = 0; i < products.Length; i++)  
            {  
                comparisons++;  
                if (string.Equals(products[i].Category, category, StringComparison.OrdinalIgnoreCase))  
                {  
                    results.Add(new SearchResult(products[i], i, stopwatch.ElapsedTicks, comparisons, true));  
                }  
            }  
  
            stopwatch.Stop();  
              
            foreach (var result in results)  
            {  
                result.ElapsedTicks = stopwatch.ElapsedTicks;  
            }  
  
            if (results.Count == 0)  
            {  
                results.Add(new SearchResult(null, -1, stopwatch.ElapsedTicks, comparisons, false));  
            }  
  
            return results;  
        }  
  
        public int GetProductCount() => products.Length;  
        public Product[] GetAllProducts() => products;  
        public Product[] GetSortedProducts() => sortedProducts;  
    }  
  
      
    public class PerformanceAnalyzer  
    {  
        public static void CompareSearchAlgorithms(EcommerceSearchEngine searchEngine, int[] testProductIds)  
        {  
            Console.WriteLine("=== PERFORMANCE COMPARISON ===");  
            Console.WriteLine();  
            Console.WriteLine($"Dataset Size: {searchEngine.GetProductCount()} products");  
            Console.WriteLine($"Test Cases: {testProductIds.Length} searches");  
            Console.WriteLine();  
  
            long totalLinearTicks = 0;  
            long totalBinaryTicks = 0;  
            int totalLinearComparisons = 0;  
            int totalBinaryComparisons = 0;  
            int linearFoundCount = 0;  
            int binaryFoundCount = 0;  
  
            Console.WriteLine("Individual Search Results:");  
            Console.WriteLine("Product ID | Linear Search | Binary Search | Difference");  
            Console.WriteLine(new string('-', 65));  
  
            foreach (int productId in testProductIds)  
            {  
                SearchResult linearResult = searchEngine.LinearSearchById(productId);  
                SearchResult binaryResult = searchEngine.BinarySearchById(productId);  
  
                totalLinearTicks += linearResult.ElapsedTicks;  
                totalBinaryTicks += binaryResult.ElapsedTicks;  
                totalLinearComparisons += linearResult.ComparisonsCount;  
                totalBinaryComparisons += binaryResult.ComparisonsCount;  
  
                if (linearResult.Found) linearFoundCount++;  
                if (binaryResult.Found) binaryFoundCount++;  
  
                double timeDifference = linearResult.ElapsedTicks > 0 ?   
                    (double)linearResult.ElapsedTicks / binaryResult.ElapsedTicks : 1.0;  
  
                Console.WriteLine($"{productId,10} | {linearResult.ComparisonsCount,8} ops | {binaryResult.ComparisonsCount,8} ops | {timeDifference:F2}x");  
            }  
  
            Console.WriteLine(new string('-', 65));  
            Console.WriteLine();  
            Console.WriteLine("SUMMARY STATISTICS:");  
            Console.WriteLine($"Linear Search  - Total Operations: {totalLinearComparisons}, Found: {linearFoundCount}");  
            Console.WriteLine($"Binary Search  - Total Operations: {totalBinaryComparisons}, Found: {binaryFoundCount}");  
            Console.WriteLine($"Efficiency Gain: {(double)totalLinearComparisons / totalBinaryComparisons:F2}x fewer operations with binary search");  
            Console.WriteLine();  
        }  
  
        public static void AnalyzeScalability()  
        {  
            Console.WriteLine("=== SCALABILITY ANALYSIS ===");  
            Console.WriteLine();  
            Console.WriteLine("Theoretical Operations Needed (Worst Case):");  
            Console.WriteLine("Dataset Size | Linear Search | Binary Search | Ratio");  
            Console.WriteLine(new string('-', 55));  
  
            int[] dataSizes = { 100, 1000, 10000, 100000, 1000000 };  
  
            foreach (int size in dataSizes)  
            {  
                int linearOps = size;  
                int binaryOps = (int)Math.Ceiling(Math.Log2(size));  
                double ratio = (double)linearOps / binaryOps;  
  
                Console.WriteLine($"{size,11} | {linearOps,13} | {binaryOps,13} | {ratio,6:F1}x");  
            }  
  
            Console.WriteLine();  
            Console.WriteLine("Key Insights:");  
            Console.WriteLine("• As dataset grows, binary search advantage increases exponentially");  
            Console.WriteLine("• For 1 million products: Binary search is ~50,000x more efficient");  
            Console.WriteLine("• Binary search scales logarithmically - very predictable performance");  
            Console.WriteLine();  
        }  
    }  
  
      
    public class SampleDataGenerator  
    {  
        private static readonly string[] Categories =   
        {  
            "Electronics", "Clothing", "Books", "Home & Garden", "Sports",   
            "Beauty", "Automotive", "Toys", "Health", "Food"  
        };  
  
        private static readonly string[] Brands =   
        {  
            "Samsung", "Apple", "Nike", "Adidas", "Sony", "LG", "Dell",   
            "HP", "Canon", "Microsoft", "Google", "Amazon"  
        };  
  
        private static readonly string[] ProductNames =   
        {  
            "Smartphone", "Laptop", "Headphones", "T-Shirt", "Jeans", "Novel",   
            "Cookbook", "Garden Tool", "Soccer Ball", "Lipstick", "Car Battery",   
            "Toy Car", "Vitamins", "Coffee", "Monitor", "Keyboard", "Mouse"  
        };  
  
        public static Product[] GenerateProducts(int count)  
        {  
            Random random = new Random(42);   
            Product[] products = new Product[count];  
  
            for (int i = 0; i < count; i++)  
            {  
                int productId = i + 1;  
                string category = Categories[random.Next(Categories.Length)];  
                string brand = Brands[random.Next(Brands.Length)];  
                string productName = $"{brand} {ProductNames[random.Next(ProductNames.Length)]} {random.Next(1000, 9999)}";  
                decimal price = (decimal)(random.NextDouble() \* 1000 + 10);  
                int stock = random.Next(0, 100);  
  
                products[i] = new Product(productId, productName, category, price, brand, stock);  
            }  
  
              
            for (int i = products.Length - 1; i > 0; i--)  
            {  
                int j = random.Next(i + 1);  
                Product temp = products[i];  
                products[i] = products[j];  
                products[j] = temp;  
            }  
  
            return products;  
        }  
    }  
  
      
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Console.WriteLine("E-COMMERCE PLATFORM SEARCH FUNCTION ANALYSIS");  
            Console.WriteLine("============================================");  
            Console.WriteLine();  
  
            try  
            {  
                // Step 1: Explain algorithmic concepts  
                AlgorithmComplexityExplainer.ExplainBigONotation();  
                Console.WriteLine();  
                AlgorithmComplexityExplainer.ExplainSearchScenarios();  
                Console.WriteLine();  
  
                // Step 2: Generate sample data  
                Console.WriteLine("=== GENERATING SAMPLE DATA ===");  
                Product[] products = SampleDataGenerator.GenerateProducts(10000);  
                Console.WriteLine($"Generated {products.Length} sample products");  
                Console.WriteLine();  
  
                // Step 3: Initialize search engine  
                EcommerceSearchEngine searchEngine = new EcommerceSearchEngine(products);  
  
                // Step 4: Display sample products  
                Console.WriteLine("=== SAMPLE PRODUCTS ===");  
                for (int i = 0; i < Math.Min(5, products.Length); i++)  
                {  
                    Console.WriteLine($"{i + 1}. {products[i]}");  
                }  
                Console.WriteLine($"... and {products.Length - 5} more products");  
                Console.WriteLine();  
  
                // Step 5: Performance testing  
                int[] testIds = { 1, 500, 2500, 5000, 7500, 9999, 15000 }; // Include non-existent ID  
                PerformanceAnalyzer.CompareSearchAlgorithms(searchEngine, testIds);  
  
                // Step 6: Scalability analysis  
                PerformanceAnalyzer.AnalyzeScalability();  
  
                // Step 7: Demonstrate different search types  
                DemonstrateSearchTypes(searchEngine);  
  
                // Step 8: Recommendations  
                Console.WriteLine("=== RECOMMENDATIONS FOR E-COMMERCE PLATFORM ===");  
                Console.WriteLine();  
                Console.WriteLine("1. PRIMARY KEY SEARCHES (Product ID):");  
                Console.WriteLine("   • Use Binary Search on sorted arrays or Hash Tables O(1)");  
                Console.WriteLine("   • Maintain sorted indices for fast lookups");  
                Console.WriteLine();  
                Console.WriteLine("2. TEXT SEARCHES (Product Name, Description):");  
                Console.WriteLine("   • Use Full-Text Search engines (Elasticsearch, Solr)");  
                Console.WriteLine("   • Implement Trie data structures for autocomplete");  
                Console.WriteLine();  
                Console.WriteLine("3. CATEGORY/FILTER SEARCHES:");  
                Console.WriteLine("   • Use Database indices and B-trees");  
                Console.WriteLine("   • Consider inverted indices for multiple filters");  
                Console.WriteLine();  
                Console.WriteLine("4. HYBRID APPROACH:");  
                Console.WriteLine("   • Combine multiple search strategies based on query type");  
                Console.WriteLine("   • Cache frequently searched items");  
                Console.WriteLine("   • Use asynchronous search for better user experience");  
                Console.WriteLine();  
  
            }  
            catch (Exception ex)  
            {  
                Console.WriteLine($"Error: {ex.Message}");  
            }  
  
            Console.WriteLine("Press any key to exit...");  
            Console.ReadKey();  
        }  
  
        private static void DemonstrateSearchTypes(EcommerceSearchEngine searchEngine)  
        {  
            Console.WriteLine("=== DIFFERENT SEARCH TYPES DEMONSTRATION ===");  
            Console.WriteLine();  
  
            // Search by ID  
            Console.WriteLine("1. Search by Product ID (Binary Search):");  
            SearchResult idResult = searchEngine.BinarySearchById(2500);  
            if (idResult.Found)  
            {  
                Console.WriteLine($"   Found: {idResult.Product}");  
                Console.WriteLine($"   Operations: {idResult.ComparisonsCount}");  
            }  
            else  
            {  
                Console.WriteLine("   Product not found");  
            }  
            Console.WriteLine();  
  
            // Search by name  
            Console.WriteLine("2. Search by Product Name (Linear Search):");  
            List<SearchResult> nameResults = searchEngine.LinearSearchByName("Samsung");  
            Console.WriteLine($"   Found {nameResults.Count(r => r.Found)} products containing 'Samsung':");  
            foreach (var result in nameResults.Where(r => r.Found).Take(3))  
            {  
                if (result.Product != null)  
                {  
                    Console.WriteLine($"   • {result.Product.ProductName}");  
                }  
            }  
            if (nameResults.Count(r => r.Found) > 3)  
            {  
                Console.WriteLine($"   ... and {nameResults.Count(r => r.Found) - 3} more");  
            }  
            Console.WriteLine();  
  
            // Search by category  
            Console.WriteLine("3. Search by Category (Linear Search):");  
            List<SearchResult> categoryResults = searchEngine.LinearSearchByCategory("Electronics");  
            Console.WriteLine($"   Found {categoryResults.Count(r => r.Found)} products in 'Electronics' category");  
            if (categoryResults.Any(r => r.Found))  
            {  
                Console.WriteLine($"   Total comparisons needed: {categoryResults.First().ComparisonsCount}");  
            }  
            Console.WriteLine();  
        }  
    }  
}

Output:









Handson – 4:

using System;  
using System.Collections.Generic;  
using System.Diagnostics;  
  
namespace FinancialForecastingRecursive  
{  
    public class FinancialData  
    {  
        public DateTime Date { get; set; }  
        public decimal Value { get; set; }  
        public decimal GrowthRate { get; set; }  
  
        public FinancialData(DateTime date, decimal value, decimal growthRate = 0)  
        {  
            Date = date;  
            Value = value;  
            GrowthRate = growthRate;  
        }  
  
        public override string ToString()  
        {  
            return $"{Date:yyyy-MM-dd}: ${Value:F2} (Growth: {GrowthRate:P2})";  
        }  
    }  
  
    public class RecursionExplainer  
    {  
        public static void ExplainRecursion()  
        {  
            Console.WriteLine("=== RECURSION CONCEPTS ===");  
            Console.WriteLine("Recursion: A function that calls itself to solve smaller subproblems");  
            Console.WriteLine("Base Case: Condition that stops the recursion");  
            Console.WriteLine("Recursive Case: Function calls itself with modified parameters");  
            Console.WriteLine("Stack Frame: Each recursive call creates a new stack frame");  
            Console.WriteLine();  
            Console.WriteLine("Benefits: Simplifies complex problems, elegant solutions");  
            Console.WriteLine("Drawbacks: Stack overflow risk, potential performance issues");  
            Console.WriteLine();  
        }  
    }  
  
    public class FinancialForecaster  
    {  
        private Dictionary<string, decimal> memoCache = new Dictionary<string, decimal>();  
        private long recursiveCallCount = 0;  
        private long memoizedCallCount = 0;  
  
        public decimal CalculateFutureValueRecursive(decimal initialValue, decimal growthRate, int periods)  
        {  
            recursiveCallCount++;  
              
            if (periods == 0)  
                return initialValue;  
              
            return CalculateFutureValueRecursive(initialValue \* (1 + growthRate), growthRate, periods - 1);  
        }  
  
        public decimal CalculateFutureValueMemoized(decimal initialValue, decimal growthRate, int periods)  
        {  
            string key = $"{initialValue}\_{growthRate}\_{periods}";  
              
            if (memoCache.ContainsKey(key))  
                return memoCache[key];  
  
            memoizedCallCount++;  
              
            decimal result;  
            if (periods == 0)  
            {  
                result = initialValue;  
            }  
            else  
            {  
                result = CalculateFutureValueMemoized(initialValue \* (1 + growthRate), growthRate, periods - 1);  
            }  
              
            memoCache[key] = result;  
            return result;  
        }  
  
        public decimal CalculateCompoundInterestRecursive(decimal principal, decimal rate, int years)  
        {  
            if (years == 0)  
                return principal;  
              
            return CalculateCompoundInterestRecursive(principal \* (1 + rate), rate, years - 1);  
        }  
  
        public decimal CalculateNPVRecursive(decimal[] cashFlows, decimal discountRate, int index = 0)  
        {  
            if (index >= cashFlows.Length)  
                return 0;  
              
            decimal presentValue = cashFlows[index] / (decimal)Math.Pow((double)(1 + discountRate), index);  
            return presentValue + CalculateNPVRecursive(cashFlows, discountRate, index + 1);  
        }  
  
        public List<FinancialData> ForecastSeriesRecursive(decimal initialValue, decimal baseGrowthRate,   
            decimal volatility, int periods, Random? random = null)  
        {  
            random = random ?? new Random();  
            var forecasts = new List<FinancialData>();  
              
            ForecastSeriesRecursiveHelper(initialValue, baseGrowthRate, volatility, periods,   
                DateTime.Today, forecasts, random);  
              
            return forecasts;  
        }  
  
        private void ForecastSeriesRecursiveHelper(decimal currentValue, decimal baseGrowthRate,   
            decimal volatility, int periodsLeft, DateTime currentDate,   
            List<FinancialData> results, Random random)  
        {  
            if (periodsLeft == 0)  
                return;  
              
            decimal adjustedGrowthRate = baseGrowthRate + (decimal)(random.NextDouble() - 0.5) \* volatility;  
            decimal nextValue = currentValue \* (1 + adjustedGrowthRate);  
              
            results.Add(new FinancialData(currentDate.AddMonths(results.Count + 1), nextValue, adjustedGrowthRate));  
              
            ForecastSeriesRecursiveHelper(nextValue, baseGrowthRate, volatility, periodsLeft - 1,   
                currentDate, results, random);  
        }  
  
        public decimal CalculateFibonacciGrowthRecursive(decimal baseValue, int period)  
        {  
            if (period <= 1)  
                return baseValue;  
              
            if (period == 2)  
                return baseValue \* 1.1m;  
              
            decimal fib1 = CalculateFibonacciGrowthRecursive(baseValue, period - 1);  
            decimal fib2 = CalculateFibonacciGrowthRecursive(baseValue, period - 2);  
              
            return fib1 + (fib2 \* 0.1m);  
        }  
  
        public decimal CalculateFibonacciGrowthOptimized(decimal baseValue, int period)  
        {  
            Dictionary<int, decimal> memo = new Dictionary<int, decimal>();  
            return CalculateFibonacciGrowthMemo(baseValue, period, memo);  
        }  
  
        private decimal CalculateFibonacciGrowthMemo(decimal baseValue, int period, Dictionary<int, decimal> memo)  
        {  
            if (memo.ContainsKey(period))  
                return memo[period];  
              
            decimal result;  
            if (period <= 1)  
                result = baseValue;  
            else if (period == 2)  
                result = baseValue \* 1.1m;  
            else  
            {  
                decimal fib1 = CalculateFibonacciGrowthMemo(baseValue, period - 1, memo);  
                decimal fib2 = CalculateFibonacciGrowthMemo(baseValue, period - 2, memo);  
                result = fib1 + (fib2 \* 0.1m);  
            }  
              
            memo[period] = result;  
            return result;  
        }  
  
        public void ResetCounters()  
        {  
            recursiveCallCount = 0;  
            memoizedCallCount = 0;  
            memoCache.Clear();  
        }  
  
        public long GetRecursiveCallCount() => recursiveCallCount;  
        public long GetMemoizedCallCount() => memoizedCallCount;  
    }  
  
    public class PerformanceAnalyzer  
    {  
        public static void AnalyzeTimeComplexity()  
        {  
            Console.WriteLine("=== TIME COMPLEXITY ANALYSIS ===");  
            Console.WriteLine("Basic Recursive Future Value: O(n) - linear time");  
            Console.WriteLine("Memoized Version: O(n) - but with reduced constant factor");  
            Console.WriteLine("Fibonacci-based Growth: O(2^n) - exponential without memoization");  
            Console.WriteLine("Fibonacci with Memoization: O(n) - linear time");  
            Console.WriteLine("NPV Calculation: O(n) - linear in number of cash flows");  
            Console.WriteLine();  
        }  
  
        public static void CompareRecursiveVsIterative(FinancialForecaster forecaster)  
        {  
            Console.WriteLine("=== RECURSIVE VS ITERATIVE COMPARISON ===");  
              
            decimal initialValue = 1000m;  
            decimal growthRate = 0.05m;  
            int[] testPeriods = { 5, 10, 15, 20 };  
              
            Console.WriteLine("Periods | Recursive | Iterative | Difference");  
            Console.WriteLine("--------|-----------|-----------|----------");  
              
            foreach (int periods in testPeriods)  
            {  
                Stopwatch sw1 = Stopwatch.StartNew();  
                decimal recursiveResult = forecaster.CalculateFutureValueRecursive(initialValue, growthRate, periods);  
                sw1.Stop();  
                  
                Stopwatch sw2 = Stopwatch.StartNew();  
                decimal iterativeResult = CalculateFutureValueIterative(initialValue, growthRate, periods);  
                sw2.Stop();  
                  
                bool same = Math.Abs(recursiveResult - iterativeResult) < 0.01m;  
                string status = same ? "Same" : "Different";  
                  
                Console.WriteLine($"{periods,7} | {sw1.ElapsedTicks,9} | {sw2.ElapsedTicks,9} | {status}");  
            }  
            Console.WriteLine();  
        }  
  
        private static decimal CalculateFutureValueIterative(decimal initialValue, decimal growthRate, int periods)  
        {  
            decimal result = initialValue;  
            for (int i = 0; i < periods; i++)  
            {  
                result \*= (1 + growthRate);  
            }  
            return result;  
        }  
  
        public static void TestFibonacciPerformance(FinancialForecaster forecaster)  
        {  
            Console.WriteLine("=== FIBONACCI PERFORMANCE TEST ===");  
            Console.WriteLine("Period | Recursive (ms) | Optimized (ms) | Speedup");  
            Console.WriteLine("-------|----------------|----------------|--------");  
              
            int[] testPeriods = { 10, 15, 20, 25 };  
              
            foreach (int period in testPeriods)  
            {  
                Stopwatch sw1 = Stopwatch.StartNew();  
                decimal recursiveResult = forecaster.CalculateFibonacciGrowthRecursive(1000m, period);  
                sw1.Stop();  
                  
                Stopwatch sw2 = Stopwatch.StartNew();  
                decimal optimizedResult = forecaster.CalculateFibonacciGrowthOptimized(1000m, period);  
                sw2.Stop();  
                  
                double speedup = sw1.ElapsedMilliseconds > 0 ?   
                    (double)sw1.ElapsedMilliseconds / sw2.ElapsedMilliseconds : 1.0;  
                  
                Console.WriteLine($"{period,6} | {sw1.ElapsedMilliseconds,14} | {sw2.ElapsedMilliseconds,14} | {speedup:F1}x");  
            }  
            Console.WriteLine();  
        }  
    }  
  
    public class OptimizationTechniques  
    {  
        public static void ExplainOptimizations()  
        {  
            Console.WriteLine("=== OPTIMIZATION TECHNIQUES ===");  
            Console.WriteLine("1. MEMOIZATION:");  
            Console.WriteLine("   - Cache results of expensive recursive calls");  
            Console.WriteLine("   - Trades space for time complexity");  
            Console.WriteLine("   - Most effective for overlapping subproblems");  
            Console.WriteLine();  
            Console.WriteLine("2. TAIL RECURSION:");  
            Console.WriteLine("   - Recursive call is the last operation");  
            Console.WriteLine("   - Can be optimized to iterative by compiler");  
            Console.WriteLine("   - Reduces stack frame usage");  
            Console.WriteLine();  
            Console.WriteLine("3. ITERATIVE CONVERSION:");  
            Console.WriteLine("   - Convert recursive solution to loops");  
            Console.WriteLine("   - Eliminates stack overflow risk");  
            Console.WriteLine("   - Often more memory efficient");  
            Console.WriteLine();  
            Console.WriteLine("4. DYNAMIC PROGRAMMING:");  
            Console.WriteLine("   - Bottom-up approach using tables");  
            Console.WriteLine("   - Eliminates redundant calculations");  
            Console.WriteLine("   - Optimal for problems with optimal substructure");  
            Console.WriteLine();  
        }  
    }  
  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Console.WriteLine("FINANCIAL FORECASTING WITH RECURSIVE ALGORITHMS");  
            Console.WriteLine("===============================================");  
            Console.WriteLine();  
  
            try  
            {  
                RecursionExplainer.ExplainRecursion();  
                  
                FinancialForecaster forecaster = new FinancialForecaster();  
                  
                Console.WriteLine("=== BASIC RECURSIVE FORECASTING ===");  
                decimal initialValue = 10000m;  
                decimal growthRate = 0.08m;  
                int periods = 10;  
                  
                decimal futureValue = forecaster.CalculateFutureValueRecursive(initialValue, growthRate, periods);  
                Console.WriteLine($"Initial Value: ${initialValue:F2}");  
                Console.WriteLine($"Growth Rate: {growthRate:P2} per period");  
                Console.WriteLine($"Periods: {periods}");  
                Console.WriteLine($"Future Value: ${futureValue:F2}");  
                Console.WriteLine();  
                  
                Console.WriteLine("=== COMPOUND INTEREST CALCULATION ===");  
                decimal compound = forecaster.CalculateCompoundInterestRecursive(5000m, 0.06m, 5);  
                Console.WriteLine($"$5,000 at 6% for 5 years: ${compound:F2}");  
                Console.WriteLine();  
                  
                Console.WriteLine("=== NET PRESENT VALUE CALCULATION ===");  
                decimal[] cashFlows = { -1000m, 300m, 400m, 500m, 600m };  
                decimal npv = forecaster.CalculateNPVRecursive(cashFlows, 0.10m);  
                Console.WriteLine($"NPV of cash flows at 10% discount rate: ${npv:F2}");  
                Console.WriteLine();  
                  
                Console.WriteLine("=== FORECASTING WITH VOLATILITY ===");  
                var forecasts = forecaster.ForecastSeriesRecursive(1000m, 0.05m, 0.02m, 12, new Random(42));  
                Console.WriteLine("12-month forecast with 5% base growth and 2% volatility:");  
                foreach (var forecast in forecasts.Take(6))  
                {  
                    Console.WriteLine($"  {forecast}");  
                }  
                Console.WriteLine($"  ... and {forecasts.Count - 6} more periods");  
                Console.WriteLine();  
                  
                PerformanceAnalyzer.AnalyzeTimeComplexity();  
                PerformanceAnalyzer.CompareRecursiveVsIterative(forecaster);  
                PerformanceAnalyzer.TestFibonacciPerformance(forecaster);  
                OptimizationTechniques.ExplainOptimizations();  
                  
                Console.WriteLine("=== MEMOIZATION DEMONSTRATION ===");  
                forecaster.ResetCounters();  
                  
                decimal result1 = forecaster.CalculateFutureValueRecursive(1000m, 0.05m, 15);  
                long recursiveCalls = forecaster.GetRecursiveCallCount();  
                  
                forecaster.ResetCounters();  
                decimal result2 = forecaster.CalculateFutureValueMemoized(1000m, 0.05m, 15);  
                long memoizedCalls = forecaster.GetMemoizedCallCount();  
                  
                Console.WriteLine($"Recursive calls: {recursiveCalls}");  
                Console.WriteLine($"Memoized calls: {memoizedCalls}");  
                Console.WriteLine($"Results match: {Math.Abs(result1 - result2) < 0.01m}");  
                Console.WriteLine();  
                  
                Console.WriteLine("=== PRACTICAL RECOMMENDATIONS ===");  
                Console.WriteLine("1. Use iterative solutions for simple growth calculations");  
                Console.WriteLine("2. Apply memoization for recursive algorithms with overlapping subproblems");  
                Console.WriteLine("3. Consider tail recursion optimization where possible");  
                Console.WriteLine("4. Implement stack depth limits to prevent overflow");  
                Console.WriteLine("5. Use dynamic programming for complex financial models");  
                Console.WriteLine("6. Cache frequently computed values in production systems");  
                Console.WriteLine();  
                  
            }  
            catch (Exception ex)  
            {  
                Console.WriteLine($"Error: {ex.Message}");  
            }  
  
            Console.WriteLine("Press any key to exit...");  
            Console.ReadKey();  
        }  
    }  
}

Output: 