Week 1 Assignment Solutions (Design Pattern and Principles & Data Structures and Algorithm)

Design Pattern and Principles

Exercise 1: Implementing the Singleton Pattern

Logger.cs

```csharp
using System;

namespace SingletonPattern
{
    public sealed class Logger
    {
        private static readonly Lazy<Logger> instance = new Lazy<Logger>(() => new
Logger());

        // Private constructor
        private Logger()
        {
            Console.WriteLine("Logger initialized.");
        }

        public static Logger Instance => instance.Value;

        public void Log(string message)
        {
            Console.WriteLine($"[LOG]: {message}");
        }
    }
}
```

Program.cs

```csharp
using System;

namespace SingletonPattern
{
    class Program
    {
        static void Main(string[] args)
        {
            Logger logger1 = Logger.Instance;
            Logger logger2 = Logger.Instance;

            logger1.Log("First message.");
            logger2.Log("Second message.");

            Console.WriteLine(Object.ReferenceEquals(logger1, logger2)
                ? "Both instances are the same."
                : "Different instances exist!");
        }
    }
}
```

Output

```
C:\Users\KIIT\6363514 learning program solutions\Week 1\SingletonPattern>dotnet build
  Determining projects to restore...
  All projects are up-to-date for restore.
  SingletonPattern -> C:\Users\KIIT\6363514 learning program solutions\Week 1\SingletonPattern\bin\Debug\net8.0\SingletonPattern.dll

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:02.60

C:\Users\KIIT\6363514 learning program solutions\Week 1\SingletonPattern>dotnet run
Logger initialized.
[LOG]: First message.
[LOG]: Second message.
Both instances are the same.
```

Exercise 2: Implementing the Factory Method Pattern

ConcreteDocument.cs

```csharp
using System;

namespace FactoryMethodPatternExample
{
    public class WordDocument : IDocument
    {
        public void Open()
        {
            Console.WriteLine("Opening Word Document...");
        }
    }

    public class PdfDocument : IDocument
    {
        public void Open()
        {
            Console.WriteLine("Opening PDF Document...");
        }
    }

    public class ExcelDocument : IDocument
    {
        public void Open()
        {
            Console.WriteLine("Opening Excel Document...");
        }
    }
}
```

FactoryBase.cs

```csharp
namespace FactoryMethodPatternExample
{
    public abstract class DocumentFactory
    {
        public abstract IDocument CreateDocument();
    }

    public class WordDocumentFactory : DocumentFactory
    {
        public override IDocument CreateDocument() => new WordDocument();
    }

    public class PdfDocumentFactory : DocumentFactory
    {
        public override IDocument CreateDocument() => new PdfDocument();
    }

    public class ExcelDocumentFactory : DocumentFactory
    {
        public override IDocument CreateDocument() => new ExcelDocument();
    }
}
```

IDocument.cs

```csharp
namespace FactoryMethodPatternExample
{
    public interface IDocument
    {
        void Open();
    }
}
```

Program.cs

```csharp
using System;

namespace FactoryMethodPatternExample
{
    class Program
    {
        static void Main(string[] args)
        {
            DocumentFactory wordFactory = new WordDocumentFactory();
            DocumentFactory pdfFactory = new PdfDocumentFactory();
            DocumentFactory excelFactory = new ExcelDocumentFactory();

            IDocument wordDoc = wordFactory.CreateDocument();
            IDocument pdfDoc = pdfFactory.CreateDocument();
            IDocument excelDoc = excelFactory.CreateDocument();

            wordDoc.Open();
            pdfDoc.Open();
            excelDoc.Open();
        }
    }
}
```

Output

```
C:\Users\KIIT\6363514 learning program solutions\Week 1\FactoryMethodPattern>dotnet build
  Determining projects to restore...
  All projects are up-to-date for restore.
  FactoryMethodPattern -> C:\Users\KIIT\6363514 learning program solutions\Week 1\FactoryMethodPattern\bin\Debug\net8.0\FactoryMethodPattern.dll

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:01.00

C:\Users\KIIT\6363514 learning program solutions\Week 1\FactoryMethodPattern>dotnet run
Opening Word Document...
Opening PDF Document...
Opening Excel Document...
```

Data Structures and Algorithm
Exercise 2: E-commerce Platform Search Function

Asymptotic Notation Explanation

```None
Big O Notation Explanation : It is used to describe the time complexity of algorithms in
terms of input size (n).
It helps us understand the scalability and performance of algorithms.

    - Best Case: The fastest scenario (e.g., first item match)
    - Average Case: Expected performance over random input
    - Worst Case: The slowest scenario (e.g., item not found)

(i) Linear Search: O(n)
    - Best: O(1) (if the element is at the beginning)
    - Worst: O(n) (if the element is at the end or not present)

(ii) Binary Search: O(log n)
    - Requires sorted input.
    - Best: O(1) (if the middle item is the match)
    - Worst: O(log n)
```

SearchEngine.cs

```csharp
namespace ECommerceSearchOptimization
{
    public static class SearchEngine
    {
        public static Product? LinearSearch(Product[] products, int id)
        {
            foreach (var product in products)
            {
                if (product.ProductId == id)
                    return product;
            }
            return null;
        }

        public static Product? BinarySearch(Product[] products, int id)
        {
            int left = 0, right = products.Length - 1;

            while (left <= right)
            {
                int mid = (left + right) / 2;

                if (products[mid].ProductId == id)
                    return products[mid];
                else if (products[mid].ProductId < id)
                    left = mid + 1;
                else
                    right = mid - 1;
            }

            return null;
        }
    }
}
```

Product.cs

CSharp

```
namespace ECommerceSearchOptimization
{
    public class Product : IComparable<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 int CompareTo(Product? other)
        {
            if (other == null)
                return 1;
            return ProductId.CompareTo(other.ProductId);
        }

        public override string ToString()
        {
            return $"{ProductId}: {ProductName} [{Category}]";
        }
    }
}
```

Program.cs

```csharp
using System;

namespace ECommerceSearchOptimization
{
    class Program
    {
        static void Main(string[] args)
        {
            Product[] products = new Product[]
            {
                new Product(105, "Bluetooth Speaker", "Electronics"),
                new Product(101, "Running Shoes", "Footwear"),
                new Product(108, "Laptop", "Electronics"),
                new Product(103, "Notebook", "Stationery"),
                new Product(102, "Backpack", "Accessories"),
                new Product(107, "Smartphone", "Electronics"),
            };

            Console.WriteLine("Original Product List:");
            foreach (var product in products)
                Console.WriteLine(product);

            // Linear Search Technique
            Console.WriteLine("\nLinear Search for ProductId 107:");
            Product? linearResult = SearchEngine.LinearSearch(products, 107);
            Console.WriteLine(linearResult != null ? $"Found: {linearResult}" : "Not
Found");

            // Sorting for Binary Search
            Array.Sort(products);

            Console.WriteLine("\nSorted Product List (for Binary Search):");
            foreach (var product in products)
                Console.WriteLine(product);

            // Binary Search Tecghnique
            Console.WriteLine("\nBinary Search for ProductId 107:");
            Product? binaryResult = SearchEngine.BinarySearch(products, 107);
            Console.WriteLine(binaryResult != null ? $"Found: {binaryResult}" : "Not
Found");

            // Time Complexity Comparison
            Console.ForegroundColor = ConsoleColor.Yellow;
            Console.WriteLine("\nTime Complexity Comparison:");
            Console.ResetColor();
```

```
            Console.WriteLine("- Linear Search: O(n) → Scans each element.");
            Console.WriteLine("- Binary Search: O(log n) → Splits the array into
halves.");
            Console.WriteLine("\nBinary Search is more efficient for large datasets
(sorted), while Linear Search works for small or unsorted arrays.");
        }
    }
}
```

Output

```
C:\Users\KIIT\6363514 learning program solutions\Week 1\ECommerceSearchOptimization>dotnet run
Original Product List:
105: Bluetooth Speaker [Electronics]
101: Running Shoes [Footwear]
108: Laptop [Electronics]
103: Notebook [Stationery]
102: Backpack [Accessories]
107: Smartphone [Electronics]

Linear Search for ProductId 107:
Found: 107: Smartphone [Electronics]

Sorted Product List (for Binary Search):
101: Running Shoes [Footwear]
102: Backpack [Accessories]
103: Notebook [Stationery]
105: Bluetooth Speaker [Electronics]
107: Smartphone [Electronics]
108: Laptop [Electronics]

Binary Search for ProductId 107:
Found: 107: Smartphone [Electronics]

Time Complexity Comparison:
- Linear Search: O(n) → Scans each element.
- Binary Search: O(log n) → Splits the array into halves.

Binary Search is more efficient for large datasets (sorted), while Linear Search works for small or unsorted arrays.
```

Exercise 7: Financial Forecasting

Understand Recursive Algorithms

```
None
Recursion is a technique where a method calls itself to solve a problem by breaking it
down into smaller subproblems.
It is useful when:
    (i) The problem is defined in terms of smaller subproblems (e.g., Fibonacci,
factorial)
    (ii) You want clean, readable code (but must control stack depth and redundancy)

In financial forecasting, recursion can simulate year-on-year growth based on a given
rate.
```

Program.cs

```csharp
CSharp
using System;
using System.Collections.Generic;

namespace FinancialForecasting
{
    class Program
    {
        static void Main(string[] args)
        {
            double initialAmount = 10000;  // ₹10,000, for a while kuch bhi le liya
            double annualGrowthRate = 0.07;  // 7% growth, suppose that
            int years = 10;

            Console.WriteLine($"Predicting future value for ₹{initialAmount} at
{annualGrowthRate * 100}% annual growth for {years} years...");

            double recursiveValue = ForecastRecursive(initialAmount, annualGrowthRate,
years);
            Console.WriteLine($"\nRecursive Prediction: ₹{recursiveValue:F2}");

            var memo = new Dictionary<int, double>();
            double optimizedValue = ForecastRecursiveMemo(initialAmount,
annualGrowthRate, years, memo);
            Console.WriteLine($"Optimized Recursive (Memoized) Prediction:
₹{optimizedValue:F2}");
```

```
            double iterativeValue = ForecastIterative(initialAmount, annualGrowthRate,
years);
            Console.WriteLine($"Iterative Prediction: ₹{iterativeValue:F2}");
        }

        // Recursive Method
        static double ForecastRecursive(double amount, double rate, int years)
        {
            if (years == 0)
                return amount;

            return (1 + rate) * ForecastRecursive(amount, rate, years - 1);
        }

        // Recursive with Memoization (to avoid repeated work)
        static double ForecastRecursiveMemo(double amount, double rate, int years,
Dictionary<int, double> memo)
        {
            if (years == 0)
                return amount;

            if (memo.ContainsKey(years))
                return memo[years];

            memo[years] = (1 + rate) * ForecastRecursiveMemo(amount, rate, years - 1,
memo);
            return memo[years];
        }

        // Iterative Alternative for comparison
        static double ForecastIterative(double amount, double rate, int years)
        {
            double result = amount;
            for (int i = 0; i < years; i++)
                result *= (1 + rate);

            return result;
        }
    }
}
```

Time Complexity Analysis

None

```
Time Complexity :
        What's my Approach     Time Complexity Space Complexity
        ForecastRecursive         O(n)           O(n) (stack calls)
        ForecastRecursiveMemo      O(n)           O(n) (dict + stack)
        ForecastIterative          O(n)           O(1)

(i) Plain recursion repeats computations unnecessarily — avoid for large n.
(ii) Memoization stores already computed results to avoid re-calculation.
(iii) Iterative is the most space-efficient and best for production code.
```

Output

```
C:\Users\KIIT\6363514 learning program solutions\Week 1\FinancialForecasting>dotnet build
  Determining projects to restore...
  All projects are up-to-date for restore.
  FinancialForecasting -> C:\Users\KIIT\6363514 learning program solutions\Week 1\FinancialForecasting\bin\Debug\net8.0\FinancialForecasting.dll

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:02.39

C:\Users\KIIT\6363514 learning program solutions\Week 1\FinancialForecasting>dotnet run
Predicting future value for ₹10000 at 7.000000000000001% annual growth for 10 years...

Recursive Prediction: ₹19671.51
Optimized Recursive (Memoized) Prediction: ₹19671.51
Iterative Prediction: ₹19671.51
```