**Design principles & Patterns**

**Name: Ayush Srivastava**

**SUPERSET ID: 6363064**

**Exercise 1: Implementing the Singleton Pattern**

**Scenario:**

You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

**Steps:**

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

**CODE:**

**Program.cs:**

using System;

namespace SingletonPatternExample

{

internal class Program

{

static void Main(string[] args)

{

Logger logger1 = Logger.GetInstance();

logger1.Log("Logging first message...");

Logger logger2 = Logger.GetInstance();

logger2.Log("Logging second message...");

if (object.ReferenceEquals(logger1, logger2))

{

Console.WriteLine("✅ Both logger instances are the same (Singleton works!)");

}

else

{

Console.WriteLine("❌ Logger instances are different (Singleton failed!)");

}

}

}

}

**Logger.cs:**

using System;

namespace SingletonPatternExample

{

public class Logger

{

private static Logger \_instance;

private static readonly object \_lock = new object();

private Logger()

{

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

}

public static Logger GetInstance()

{

if (\_instance == null)

{

lock (\_lock)

{

if (\_instance == null)

{

\_instance = new Logger();

}

}

}

return \_instance;

}

public void Log(string message)

{

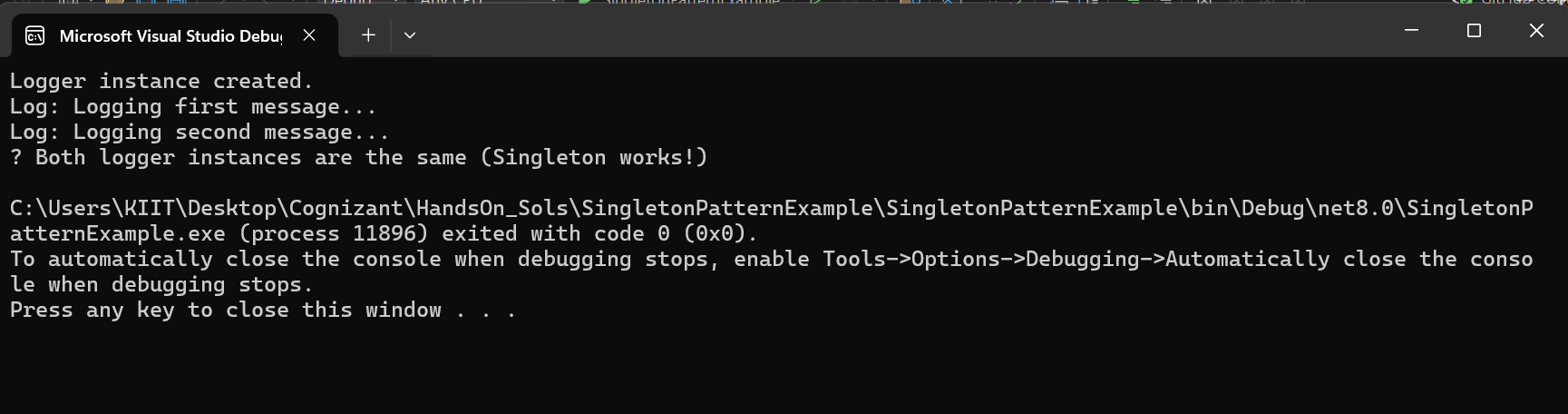
Console.WriteLine($"Log: {message}");

}

}

}

**OUTPUT:**



**Exercise 2: Implementing the Factory Method Pattern**

**Scenario:**

You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.

**Steps:**

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

**CODE:**

**Program.cs:**

using System;

namespace FactoryMethodPatternExample

{

internal class Program

{

static void Main(string[] args)

{

DocumentFactory wordFactory = new WordFactory();

IDocument wordDoc = wordFactory.CreateDocument();

wordDoc.Open();

DocumentFactory pdfFactory = new PdfFactory();

IDocument pdfDoc = pdfFactory.CreateDocument();

pdfDoc.Open();

DocumentFactory excelFactory = new ExcelFactory();

IDocument excelDoc = excelFactory.CreateDocument();

excelDoc.Open();

}

}

}

**IDocument.cs:**

namespace FactoryMethodPatternExample

{

public interface IDocument

{

void Open();

}

}

**WordDocument.cs:**

namespace FactoryMethodPatternExample

{

public class WordDocument : IDocument

{

public void Open()

{

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

}

}

}

**ExcelDocument.cs:**

namespace FactoryMethodPatternExample

{

public class ExcelDocument : IDocument

{

public void Open()

{

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

}

}

}

**PdfDocument.cs:**

namespace FactoryMethodPatternExample

{

public class PdfDocument : IDocument

{

public void Open()

{

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

}

}

}

**DocumentFactory.cs:**

namespace FactoryMethodPatternExample

{

public abstract class DocumentFactory

{

public abstract IDocument CreateDocument();

}

}

**WordFactory.cs:**

namespace FactoryMethodPatternExample

{

public class WordFactory : DocumentFactory

{

public override IDocument CreateDocument()

{

return new WordDocument();

}

}

}

**ExcelFactory.cs:**

namespace FactoryMethodPatternExample

{

public class ExcelFactory : DocumentFactory

{

public override IDocument CreateDocument()

{

return new ExcelDocument();

}

}

}

**PdfFactory.cs:**

namespace FactoryMethodPatternExample

{

public class PdfFactory : DocumentFactory

{

public override IDocument CreateDocument()

{

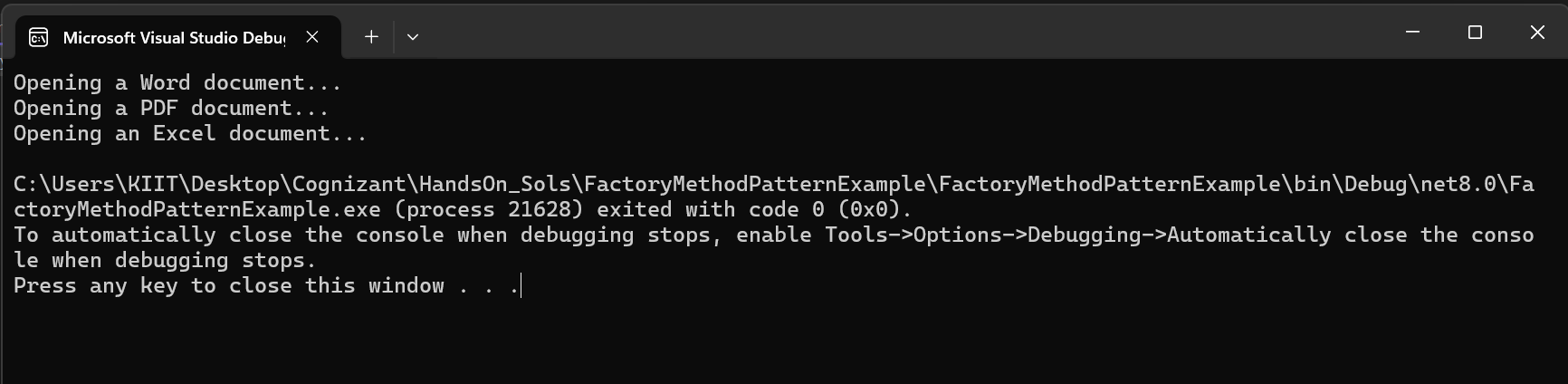
return new PdfDocument();

}

}

}

**OUTPUT:**



**Data structures and Algorithms**

**Exercise 2: E-commerce Platform Search Function**

**Scenario:**

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

**Steps:**

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

**ANSWER:**

1.Big O notation describes the upper bound of an algorithm's runtime or space requirements in terms of input size n. It helps developers understand how algorithms scale.

| **Case** | **Linear Search** | **Binary Search** |
| --- | --- | --- |
| **Best** | O(1) (first element) | O(1) (middle match) |
| **Average** | O(n) | O(log n) |
| **Worst** | O(n) | O(log n) |

**CODE:**

**Program.cs:**

using System;

namespace ECommerceSearch

{

class Program

{

static void Main(string[] args)

{

// Sample products

Product[] products = new Product[]

{

new Product(101, "Shoes", "Footwear"),

new Product(102, "T-shirt", "Clothing"),

new Product(103, "Laptop", "Electronics"),

new Product(104, "Phone", "Electronics"),

new Product(105, "Book", "Stationery")

};

// Sort products by ProductName for binary search

Array.Sort(products, (p1, p2) => string.Compare(p1.ProductName, p2.ProductName, StringComparison.OrdinalIgnoreCase));

Console.WriteLine("🛍️ Welcome to the E-Commerce Product Search System!");

bool running = true;

while (running)

{

Console.WriteLine("\nPlease choose an option:");

Console.WriteLine("1. Linear Search");

Console.WriteLine("2. Binary Search");

Console.WriteLine("3. Exit");

Console.Write("Enter your choice: ");

string input = Console.ReadLine();

switch (input)

{

case "1":

Console.Write("Enter product name to search (Linear): ");

string linearSearchTerm = Console.ReadLine();

int linearIndex = LinearSearch(products, linearSearchTerm);

if (linearIndex != -1)

Console.WriteLine($"✅ Product found: {products[linearIndex]}");

else

Console.WriteLine("❌ Product not found using linear search.");

break;

case "2":

Console.Write("Enter product name to search (Binary): ");

string binarySearchTerm = Console.ReadLine();

int binaryIndex = BinarySearch(products, binarySearchTerm);

if (binaryIndex != -1)

Console.WriteLine($"✅ Product found: {products[binaryIndex]}");

else

Console.WriteLine("❌ Product not found using binary search.");

break;

case "3":

Console.WriteLine("👋 Exiting the search system. Have a nice day!");

running = false;

break;

default:

Console.WriteLine("⚠️ Invalid choice! Please select 1, 2 or 3.");

break;

}

}

}

public static int LinearSearch(Product[] products, string targetName)

{

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

{

if (products[i].ProductName.Equals(targetName, StringComparison.OrdinalIgnoreCase))

return i;

}

return -1;

}

public static int BinarySearch(Product[] products, string targetName)

{

int left = 0;

int right = products.Length - 1;

while (left <= right)

{

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

int comparison = string.Compare(products[mid].ProductName, targetName, StringComparison.OrdinalIgnoreCase);

if (comparison == 0)

return mid;

else if (comparison < 0)

left = mid + 1;

else

right = mid - 1;

}

return -1;

}

}

public class Product

{

public int ProductId { get; }

public string ProductName { get; }

public string Category { get; }

public Product(int productId, string productName, string category)

{

ProductId = productId;

ProductName = productName;

Category = category;

}

public override string ToString()

{

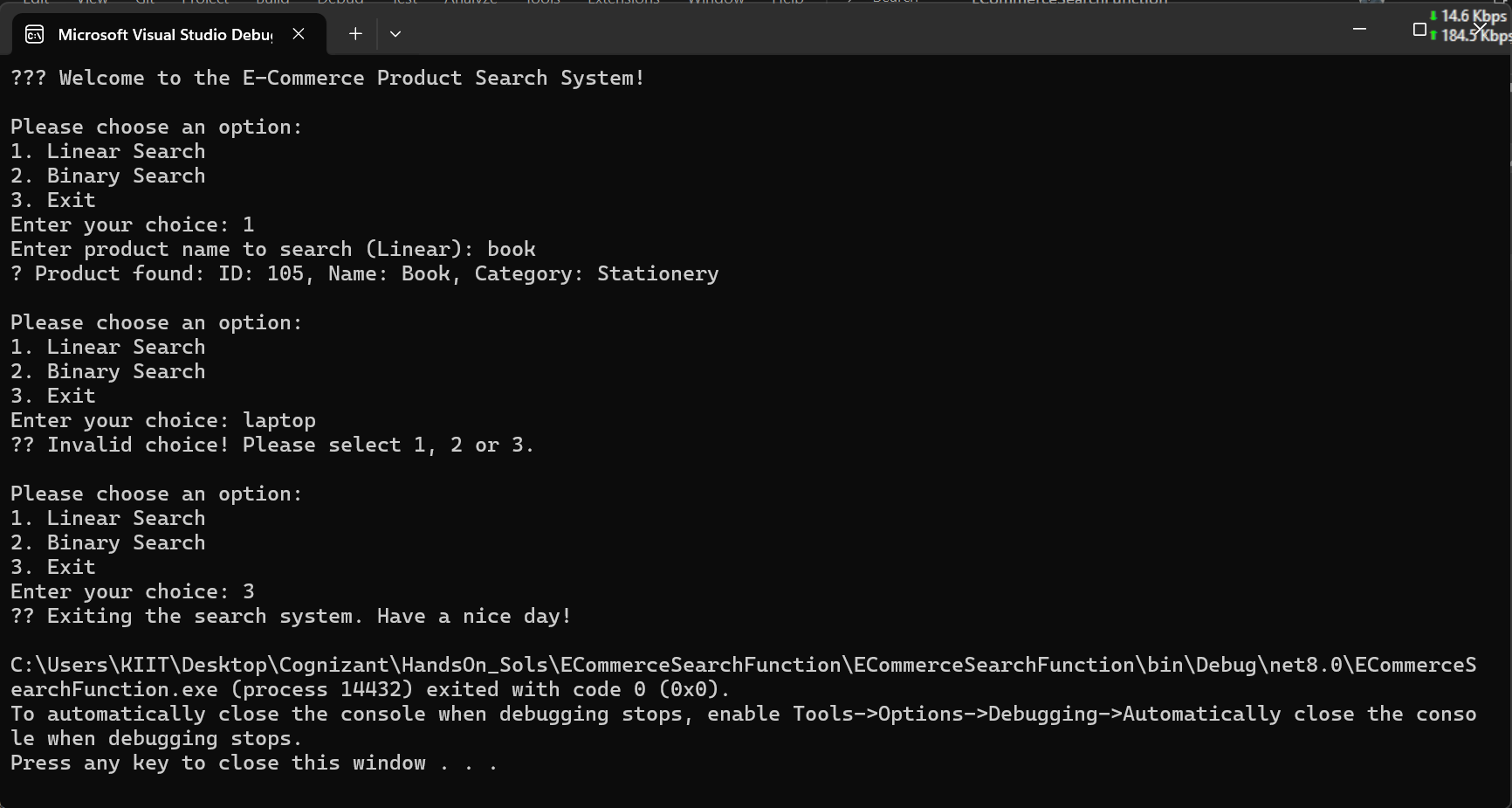
return $"ID: {ProductId}, Name: {ProductName}, Category: {Category}";

}

}

}

**OUTPUT:**



4.

| **Algorithm** | **Time Complexity** | **When to Use** |
| --- | --- | --- |
| Linear Search | **O(n)** | Small datasets or unsorted data |
| Binary Search | **O(log n)** | Large datasets that are sorted |
|  |  |  |

For an **e-commerce platform**, where performance and scalability are critical:

* Prefer Binary Search with sorted data or better, use indexes or hashing or even databases with indexing.
* For small datasets or rare use-cases where data isn't sorted, linear search is acceptable.

**Exercise 7: Financial Forecasting**

**Scenario:**

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

**Steps:**

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

**ANSWERS:**

1. Recursion is a method where a function calls itself to solve smaller subproblems.

Ideal for problems that can be broken down into similar subproblems (e.g., factorial, Fibonacci, forecasting based on past values).

**CODE:**

**Program.cs:**

using System;

namespace FinancialForecasting

{

internal class Program

{

static void Main(string[] args)

{

Console.WriteLine("Enter current value (e.g., 1000):");

double currentValue = Convert.ToDouble(Console.ReadLine());

Console.WriteLine("Enter annual growth rate (as percentage, e.g., 10):");

double growthRate = Convert.ToDouble(Console.ReadLine()) / 100;

Console.WriteLine("Enter number of years to forecast:");

int years = Convert.ToInt32(Console.ReadLine());

double futureValue = PredictFutureValue(currentValue, growthRate, years);

Console.WriteLine($"\nPredicted value after {years} years: {futureValue:F2}");

}

static double PredictFutureValue(double currentValue, double growthRate, int years)

{

if (years == 0)

return currentValue;

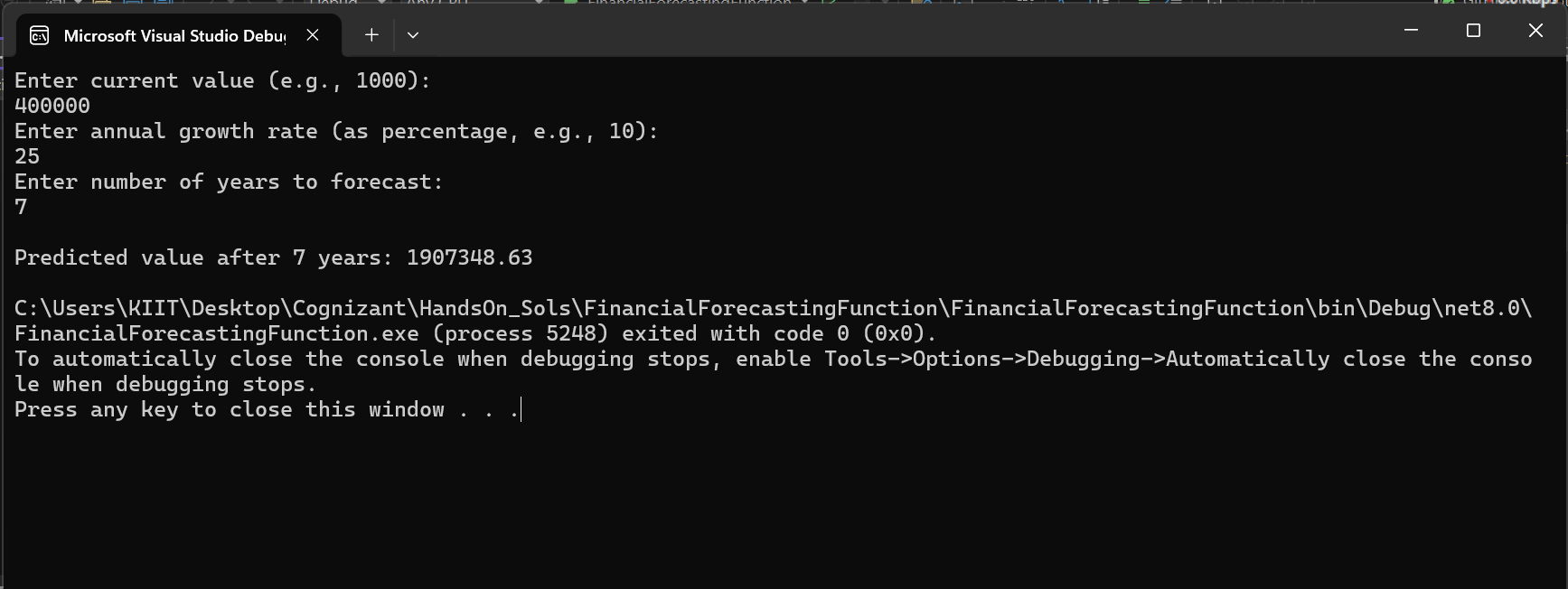
return PredictFutureValue(currentValue \* (1 + growthRate), growthRate, years - 1);

}

}

}

**OUTPUT:**



4.

| **Version** | **Time Complexity** | **Space Complexity** |
| --- | --- | --- |
| Basic Recursion | O(n) | O(n) (call stack) |
| Memoized Recursion | O(n) | O(n) (memo array) |

* Recursive calls add up fast. Without optimization, values may be recomputed multiple times.
* Memoization stores previously computed values, reducing redundant calculations.