**Design principles & Patterns**

**Exercise 1: Implementing the Singleton Pattern**

**Scenario:**

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

**Steps:**

**1. Create a New Java Project:**

**o Create a new Java project named SingletonPatternExample.**

**2. Define a Singleton Class:**

**o Create a class named Logger that has a private static instance of itself.**

**o Ensure the constructor of Logger is private.**

**o Provide a public static method to get the instance of the Logger class.**

**3. Implement the Singleton Pattern:**

**o Write code to ensure that the Logger class follows the Singleton design pattern.**

**4. Test the Singleton Implementation:**

**o Create a test class to verify that only one instance of Logger is created and used across the application.**

**CODE:-**

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($"[{DateTime.UtcNow}] {message}");

}

}

class Program

{

static void Main(string[] args)

{

Logger logger1 = Logger.GetInstance();

logger1.Log("Opening a Word document.");

Logger logger2 = Logger.GetInstance();

logger2.Log("Opening a PDF document.");

Logger logger3 = Logger.GetInstance();

logger3.Log("Opening an Excel document.");

// Verify that all instances are the same

Console.WriteLine($"Same instance? {ReferenceEquals(logger1, logger2)}");

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

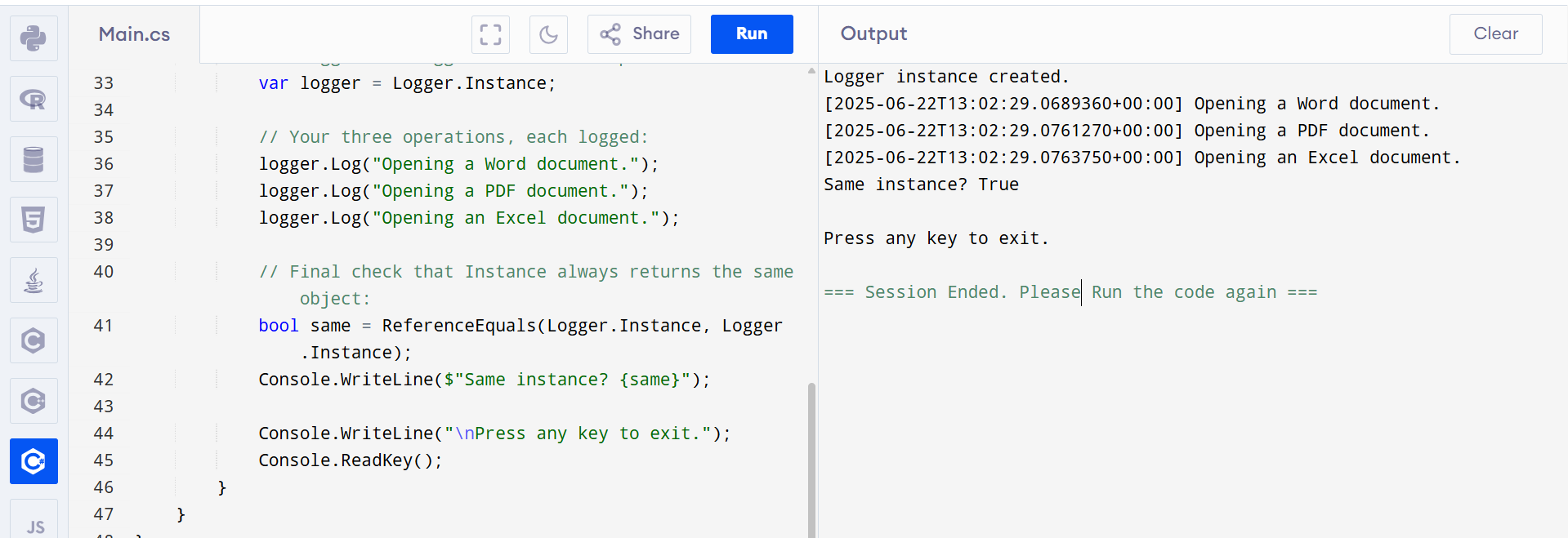
Console.ReadKey();

}

}

}

**OUTPUT:-**

****

**This confirms that the same Logger instance is used across the application.**

**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:**

**o Create a new Java project named FactoryMethodPatternExample.**

**2. Define Document Classes:**

**o Create interfaces or abstract classes for different document types such as WordDocument, PdfDocument, and ExcelDocument.**

**3. Create Concrete Document Classes:**

**o Implement concrete classes for each document type that implements or extends the above interfaces or abstract classes.**

**4. Implement the Factory Method:**

**o Create an abstract class DocumentFactory with a method createDocument().**

**o Create concrete factory classes for each document type that extends DocumentFactory and implements the createDocument() method.**

**5. Test the Factory Method Implementation:**

**o Create a test class to demonstrate the creation of different document types using the factory method.**

**Code:-**

using System;

namespace FactoryMethodPatternExample

{

// Step 1: Document interface

public interface IDocument

{

void Open();

}

// Step 2: Concrete document classes

public class WordDocument : IDocument

{

public void Open()

{

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

}

}

public class PdfDocument : IDocument

{

public void Open()

{

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

}

}

public class ExcelDocument : IDocument

{

public void Open()

{

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

}

}

// Step 3: Abstract factory class

public abstract class DocumentFactory

{

public abstract IDocument CreateDocument();

}

// Step 4: Concrete factories

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();

}

}

// Step 5: Main Program to test

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();

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

Console.ReadKey();

}

}

}

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:**

**o Explain Big O notation and how it helps in analyzing algorithms.**

**o Describe the best, average, and worst-case scenarios for search operations.**

**2. Setup:**

**o Create a class Product with attributes for searching, such as productId, productName, and category.**

**3. Implementation:**

**o Implement linear search and binary search algorithms.**

**o Store products in an array for linear search and a sorted array for binary search.**

**4. Analysis:**

**o Compare the time complexity of linear and binary search algorithms.**

**o Discuss which algorithm is more suitable for your platform and why.**

**1. Understanding Asymptotic Notation:-**

Big O notation provides a way to describe how the running time or space requirements of an algorithm grow relative to the size of its input, nnn, abstracting away constant factors and lower-order terms. For example, O(1) describes constant-time operations (like accessing an element by index), O(n) describes linear-time operations (such as scanning every item in a list), and O(log n) describes logarithmic-time operations (like halving a search space at each step). When applied to search, the best-case scenario occurs if the target is found immediately (O(1) for both linear and binary search), the average-case cost averages across all possible target positions (O(n) for linear, O(log n) for binary), and the worst-case cost is the maximum number of comparisons needed (O(n) for a miss or last-element match in linear search, O(log n) for binary search when the element is absent or at an extreme).

**2. Setup:**

First, we define a simple Product class to represent items in our catalog. Each Product has an integer ProductId, a string ProductName, and a string Category.

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() => $"{ProductId}: {ProductName} ({Category})";

}

### **3. Implementation**

We implement two search methods:

* **Linear Search (O(n))** scans an unsorted array, checking each product’s name for a match until it finds the target or reaches the end.
* **Binary Search (O(log n))** operates on a suitably sorted array (sorted by ProductName), repeatedly comparing the target to the middle element and halving the search interval.

using System;

using System.Linq;

namespace ECommerceSearchExample

{

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 $"{ProductId}: {ProductName} ({Category})";

}

}

class Program

{

static void Main(string[] args)

{

var products = new[]

{

new Product(1, "Apple iPhone", "Electronics"),

new Product(2, "Banana", "Groceries"),

new Product(3, "Asus Laptop", "Electronics"),

new Product(4, "Couch", "Furniture"),

new Product(5, "Apricot Jam", "Groceries"),

new Product(6, "Desk Lamp", "Furniture")

};

Console.WriteLine("=== Linear Search (unsorted array) ===");

RunLinearSearch(products, "Apricot");

RunLinearSearch(products, "Desk");

RunLinearSearch(products, "Zebra");

Console.WriteLine("\n=== Binary Search (sorted by Name) ===");

var sortedProducts = products.OrderBy(p => p.ProductName).ToArray();

RunBinarySearch(sortedProducts, "Asus Laptop");

RunBinarySearch(sortedProducts, "Banana");

RunBinarySearch(sortedProducts, "Zebra");

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

Console.ReadKey();

}

static void RunLinearSearch(Product[] items, string query)

{

int index = LinearSearch(items, query);

if (index >= 0)

Console.WriteLine($"Found \"{query}\" at index {index}: {items[index]}");

else

Console.WriteLine($"\"{query}\" not found (linear).");

}

static int LinearSearch(Product[] items, string query)

{

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

{

if (items[i].ProductName.IndexOf(query, StringComparison.OrdinalIgnoreCase) >= 0)

return i;

}

return -1;

}

static void RunBinarySearch(Product[] items, string query)

{

int index = BinarySearch(items, query);

if (index >= 0)

Console.WriteLine($"Found \"{query}\" at index {index}: {items[index]}");

else

Console.WriteLine($"\"{query}\" not found (binary).");

}

static int BinarySearch(Product[] items, string query)

{

int left = 0;

int right = items.Length - 1;

while (left <= right)

{

int mid = (left + right) / 2;

int compare = string.Compare(items[mid].ProductName, query, StringComparison.OrdinalIgnoreCase);

if (compare == 0)

return mid;

else if (compare < 0)

left = mid + 1;

else

right = mid - 1;

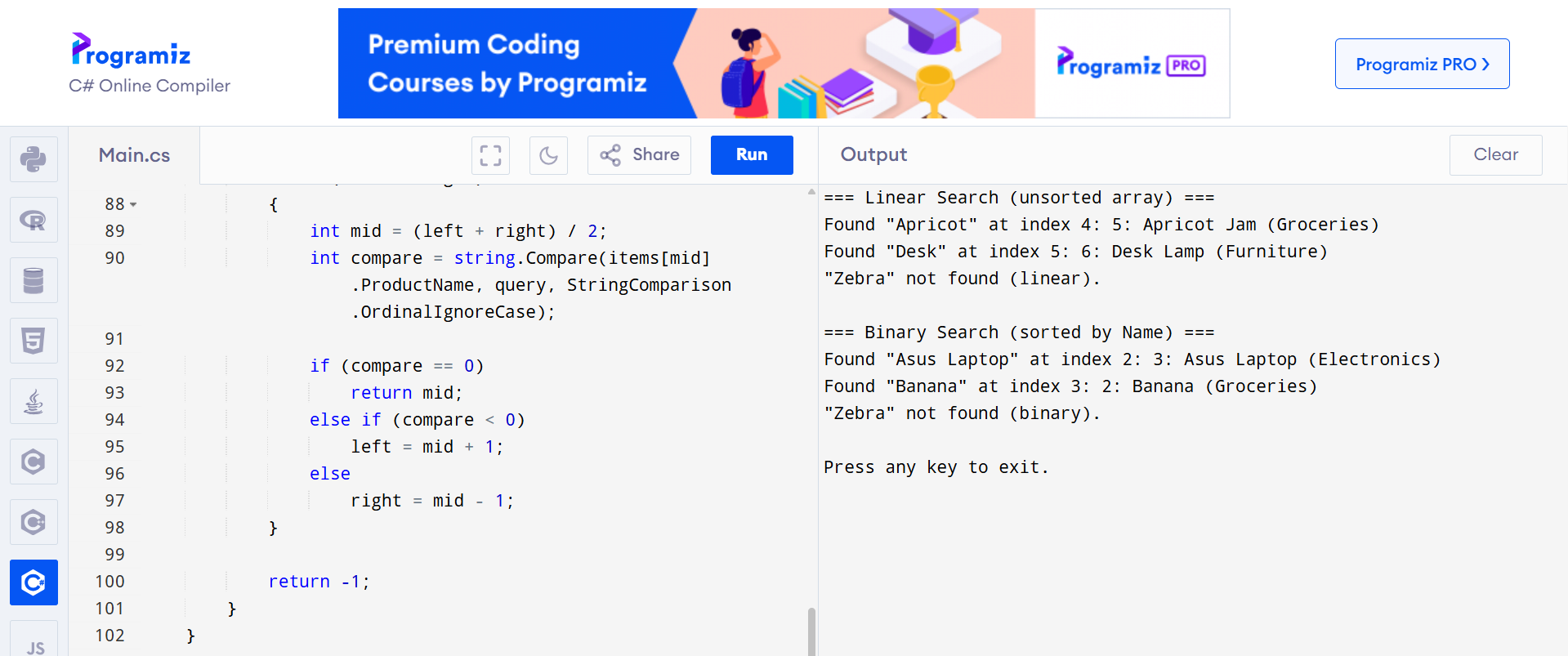
}

return -1;

}

}

}

**Output:-**

**Note:** Both algorithms return the index of the matching product or –1 if not found:

**4. Analysis & Recommendation:-**

Time Complexity:

* Linear Search runs in O(n) time in the average and worst cases, with O(1) best-case performance (target at first position).
* Binary Search requires an initial sort (O(n log n)) but then performs lookups in O(log n) time regardless of case, halving the search range at each step.

Suitability for an E-commerce Platform:

* If your catalog is small or you perform very few searches, the simplicity of linear search (no sorting overhead) may suffice.
* For large catalogs with frequent queries, binary search (or better yet, an indexed data structure such as a balanced tree or hash index) dramatically reduces per-query time, delivering O(log n) lookups instead of O(n).

**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:**

o Explain the concept of recursion and how it can simplify certain problems.

**2. Setup:**

o Create a method to calculate the future value using a recursive approach.

**3. Implementation:**

o Implement a recursive algorithm to predict future values based on past growth rates.

**4. Analysis:**

o Discuss the time complexity of your recursive algorithm.

o Explain how to optimize the recursive solution to avoid excessive computation.

Solution:

**1. Understand Recursive Algorithms:**

Recursion is a programming technique in which a method calls itself to break a complex problem into simpler sub-problems. Each recursive call works on a smaller portion of the input, and the process continues until it reaches a **base case**, a condition under which the method returns a direct answer without further recursion. In our financial forecasting scenario, we can view “compute the value after *n* periods” as “compute the value after *n–1* periods, then apply one more period’s growth.” This mirrors the mathematical definition of compound interest and makes the code both compact and expressive.

**2.Setup:**

We start by defining the inputs our algorithm needs:

* **principal**: the starting amount of money (double).
* **rate**: the growth rate per period, expressed as a decimal (e.g., 5% → 0.05).
* **n**: the number of periods to forecast (int).

**3.. Implementation:**

using System;

namespace FinancialForecasting

{

class Program

{

static void Main(string[] args)

{

double initialValue = 1000.0; // starting principal

double averageRate = 0.05; // 5% per period

int periods = 10; // forecast 10 periods ahead

double futureValue = FutureValueRecursive(initialValue, averageRate, periods);

Console.WriteLine($"Future value after {periods} periods: {futureValue:F2}");

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

Console.ReadKey();

}

/// <summary>

/// Recursively computes the future value by compounding one period at a time.

/// Base case: n == 0 returns the original principal.

/// Recursive case: compute value for n-1 periods, then apply one more period’s growth.

/// </summary>

static double FutureValueRecursive(double principal, double rate, int n)

{

if (n == 0)

return principal;

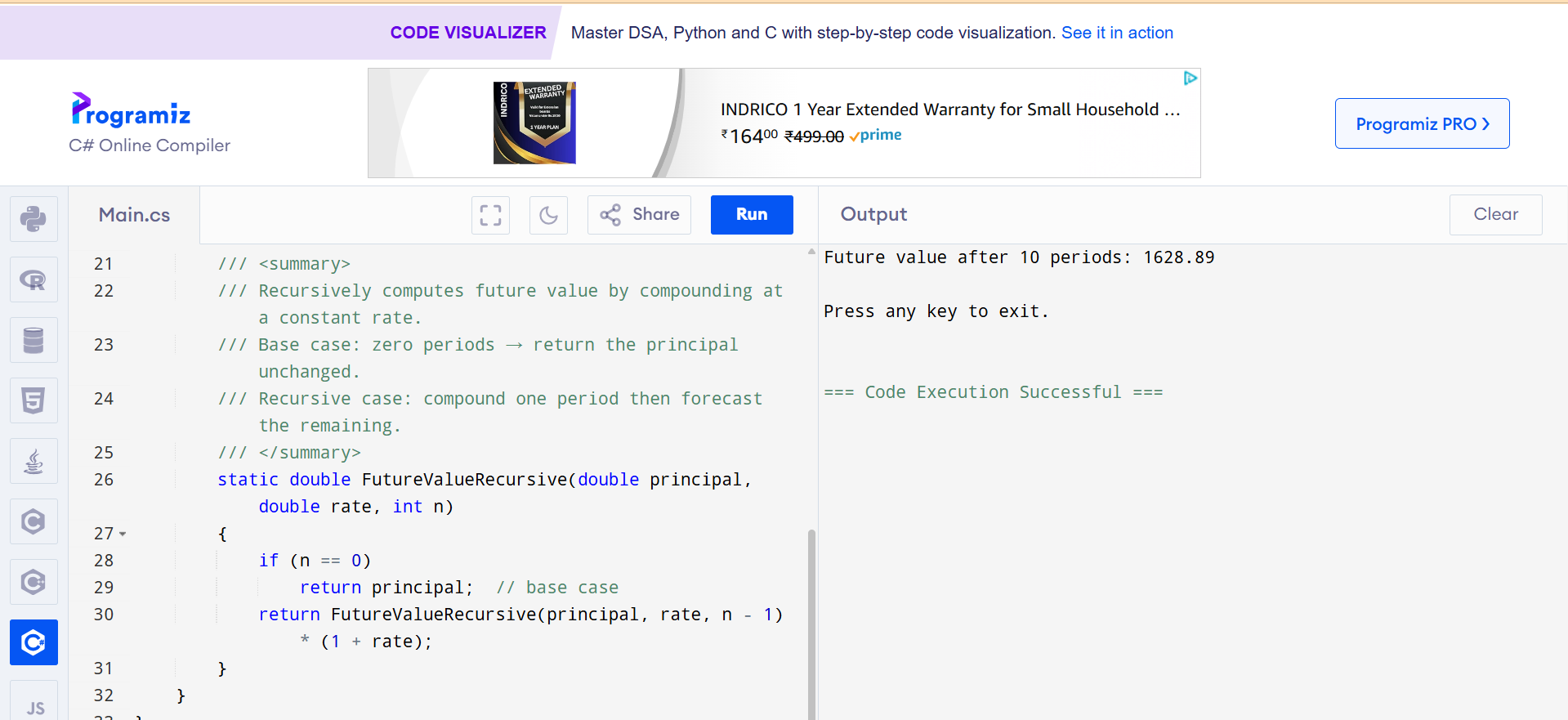
return FutureValueRecursive(principal, rate, n - 1) \* (1 + rate);

}

}

}

**4.OUTPUT:-**

****

**4. Analysis & Optimization:-**

Time Complexity:-

* The naïve recursive method calls itself once per period, so it performs *n* multiplications and *n* method calls.
* This yields O(n) time complexity, and because each call uses a stack frame, it also uses O(n) space on the call stack.