**Algorithms\_Data Structures Hands-on**

**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.

using System;

class Program

{

class Product

{

public int ProductId;

public string ProductName;

public string Category;

public Product(int id, string name, string category)

{

ProductId = id;

ProductName = name;

Category = category;

}

public override string ToString()

{

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

}

}

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

{

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

{

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

{

return products[i];

}

}

return null;

}

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

{

int low = 0;

int high = products.Length - 1;

while (low <= high)

{

int mid = (low + high) / 2;

int result = string.Compare(products[mid].ProductName, targetName, ignoreCase: true);

if (result == 0)

return products[mid];

else if (result < 0)

low = mid + 1;

else

high = mid - 1;

}

return null;

}

static void Main()

{

Product[] products = new Product[]

{

new Product(101, "Laptop", "Electronics"),

new Product(102, "Shoes", "Fashion"),

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

new Product(104, "Book", "Education"),

};

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

Product result1 = LinearSearch(products, "Shoes");

Console.WriteLine(result1 != null ? result1.ToString() : "Not Found");

Array.Sort(products, (p1, p2) => string.Compare(p1.ProductName, p2.ProductName, ignoreCase: true));

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

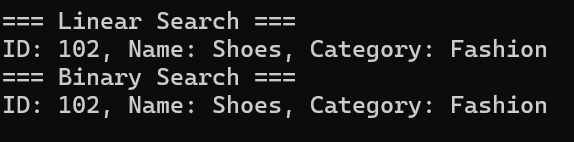
Product result2 = BinarySearch(products, "Shoes");

Console.WriteLine(result2 != null ? result2.ToString() : "Not Found");

}

}

Output:



**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.

using System;

using System.Collections.Generic;

class Program

{

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

{

if (years == 0)

return currentValue;

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

}

static Dictionary<int, double> memo = new Dictionary<int, double>();

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

{

if (years == 0)

return currentValue;

if (memo.ContainsKey(years))

return memo[years];

double result = PredictFutureValueMemo(currentValue, growthRate, years - 1) \* (1 + growthRate);

memo[years] = result;

return result;

}

static void Main()

{

double initialValue = 1000;

double growthRate = 0.08;

int years = 5;

Console.WriteLine("=== Financial Forecasting ===");

Console.WriteLine($"Initial Value: ${initialValue}");

Console.WriteLine($"Growth Rate: {growthRate \* 100}% per year");

Console.WriteLine($"Years: {years}");

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

Console.WriteLine($"\nPredicted Future Value (Recursive): ${futureValue:F2}");

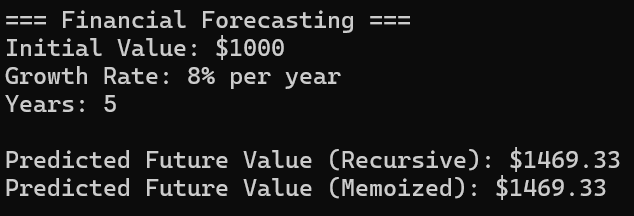
double futureValueMemo = PredictFutureValueMemo(initialValue, growthRate, years);

Console.WriteLine($"Predicted Future Value (Memoized): ${futureValueMemo:F2}");

}

}

OUTPUT:



**Design Principles and patterns hands-on**

**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:**
   * Create a new Java 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.

using System;

class Program

{

public class Logger

{

private static Logger instance;

private Logger()

{

Console.WriteLine("Logger instance created");

}

public static Logger GetInstance()

{

if (instance == null)

{

instance = new Logger();

}

return instance;

}

public void Log(string message)

{

Console.WriteLine("Log: " + message);

}

}

static void Main()

{

Logger logger1 = Logger.GetInstance();

logger1.Log("First message");

Logger logger2 = Logger.GetInstance();

logger2.Log("Second message");

if (logger1 == logger2)

{

Console.WriteLine("Both logger1 and logger2 are the same instance.");

}

else

{

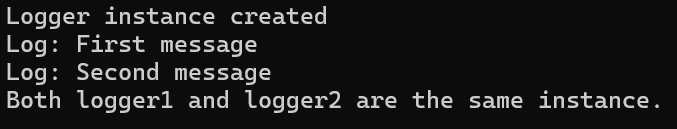
Console.WriteLine("Different instances (this should not happen).");

}

}

}

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.

using System;

class Program

{

public interface IDocument

{

void Open();

}

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...");

}

}

public abstract class DocumentFactory

{

public abstract IDocument CreateDocument();

}

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();

}

}

static void Main()

{

DocumentFactory wordFactory = new WordDocumentFactory();

IDocument word = wordFactory.CreateDocument();

word.Open();

DocumentFactory pdfFactory = new PdfDocumentFactory();

IDocument pdf = pdfFactory.CreateDocument();

pdf.Open();

DocumentFactory excelFactory = new ExcelDocumentFactory();

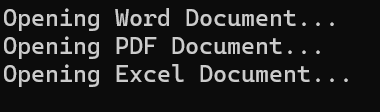
IDocument excel = excelFactory.CreateDocument();

excel.Open();

}

}

Output:

-