Exercise 2: E-commerce Platform Search Function

Code:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ECommerceSearchDemo

{

// Product class with basic searchable attributes

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

}

}

// Search algorithms

public class SearchDemo

{

// Linear search: O(n)

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

{

foreach (var product in products)

{

if (product.ProductName.Equals(targetName, StringComparison.OrdinalIgnoreCase))

{

return product;

}

}

return null;

}

// Binary search: O(log n), requires sorted array

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

{

int left = 0;

int right = products.Length - 1;

while (left <= right)

{

int mid = (left + right) / 2;

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

if (comparison == 0)

return products[mid];

else if (comparison < 0)

left = mid + 1;

else

right = mid - 1;

}

return null;

}

}

class Program

{

static void Main(string[] args)

{

// Sample product list

Product[] products = new Product[]

{

new Product(1, "Laptop", "Electronics"),

new Product(2, "Shoes", "Footwear"),

new Product(3, "Book", "Education"),

new Product(4, "Mobile", "Electronics"),

new Product(5, "Watch", "Accessories")

};

Console.WriteLine("=== LINEAR SEARCH ===");

var linearResult = SearchDemo.LinearSearch(products, "Book");

Console.WriteLine(linearResult != null ? linearResult.ToString() : "Product not found");

// Sort products by ProductName for binary search

Array.Sort(products, (a, b) => a.ProductName.CompareTo(b.ProductName));

Console.WriteLine("\n=== BINARY SEARCH ===");

var binaryResult = SearchDemo.BinarySearch(products, "Book");

Console.WriteLine(binaryResult != null ? binaryResult.ToString() : "Product not found");

// Time complexity explanation

Console.WriteLine("\n=== TIME COMPLEXITY ===");

Console.WriteLine("Linear Search: O(n) - Good for small or unsorted lists.");

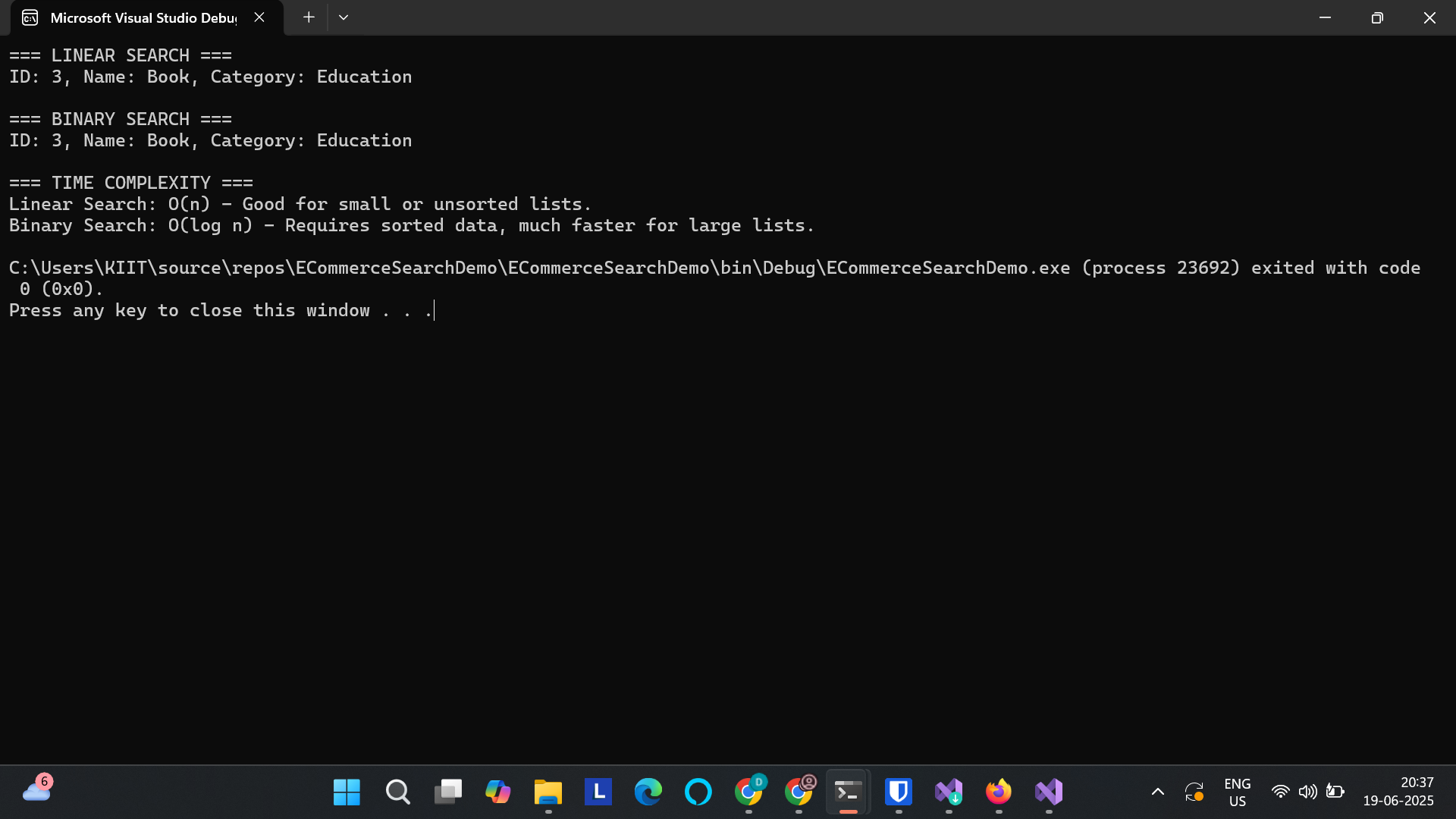
Console.WriteLine("Binary Search: O(log n) - Requires sorted data, much faster for large lists.");

}

}

}

Output:



Time complexity:

Linear Search: O(n)

Binary Search: O(logn)

As an ecommerce site can sort it’s products by product name or product ID, Binary search is better suited for the use case.

Exercise 7: Financial Forecasting

Code:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace FinancialForecastApp

{

class Program

{

// Recursive method to calculate future value

static double CalculateFutureValue(double presentValue, double growthRate, int years)

{

// Base case

if (years == 0)

return presentValue;

// Recursive step

return (1 + growthRate) \* CalculateFutureValue(presentValue, growthRate, years - 1);

}

// Optimized version using memoization

static double CalculateFutureValueMemo(double presentValue, double growthRate, int years, double[] memo)

{

if (years == 0)

return presentValue;

if (memo[years] != 0)

return memo[years];

memo[years] = (1 + growthRate) \* CalculateFutureValueMemo(presentValue, growthRate, years - 1, memo);

return memo[years];

}

static void Main(string[] args)

{

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

Console.Write("Enter Present Value (e.g., 1000): ");

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

Console.Write("Enter Annual Growth Rate (e.g., 0.05 for 5%): ");

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

Console.Write("Enter Number of Years: ");

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

Console.WriteLine("\n--- Using Simple Recursion ---");

double futureValue = CalculateFutureValue(pv, rate, years);

Console.WriteLine($"Future Value after {years} years: {futureValue:F2}");

Console.WriteLine("\n--- Using Memoized Recursion ---");

double[] memo = new double[years + 1];

double futureValueMemo = CalculateFutureValueMemo(pv, rate, years, memo);

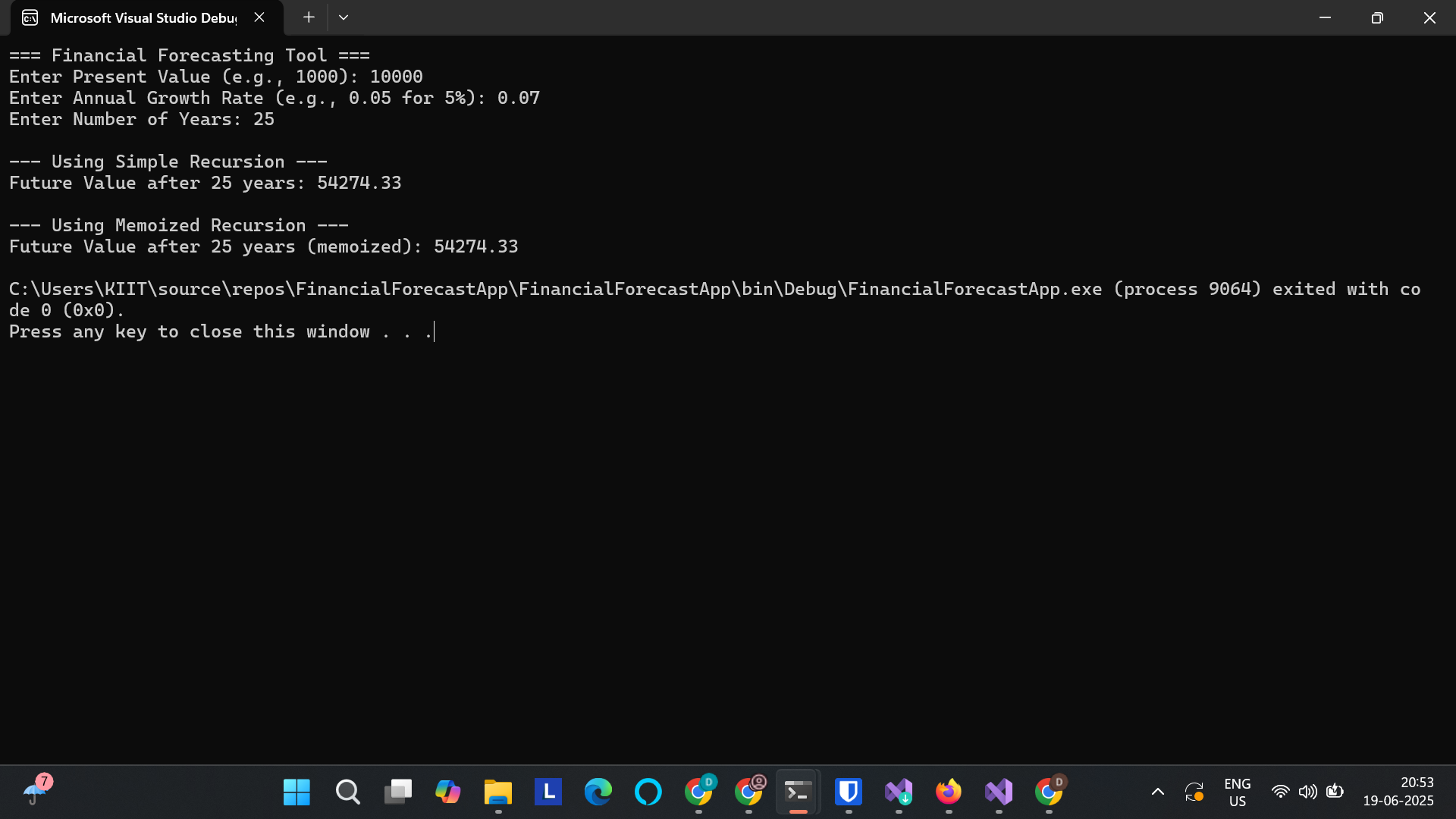
Console.WriteLine($"Future Value after {years} years (memoized): {futureValueMemo:F2}");

}

}

}

Output:



--- Analysis ---

1. Recursion is a technique where a function calls itself to solve smaller instances of a problem.

2. In this case, we recursively compute future value by reducing the year count.

3. Time Complexity (Simple Recursion): O(n) due to n recursive calls.

4. Time Complexity (Memoized Recursion): O(n), but avoids redundant calculations.

5. For larger 'years' values, memoization is preferred to avoid stack overflow or recomputation.