Suman Sourabh

6363866

**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:**

public class Product

{

public int ProductId { get; set; }

public string ProductName { get; set; }

public string Category { get; set; }

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

{

ProductId = productId;

ProductName = productName;

Category = category;

}

public override string ToString()

{

return $"{ProductId} - {ProductName} ({Category})";

}

}

using System;

class SearchDemo

{

// Linear Search by Product Name

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

{

foreach (var product in products)

{

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

{

return product;

}

}

return null;

}

// Binary Search by Product Name (Array must be sorted)

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

{

int left = 0, right = products.Length - 1;

while (left <= right)

{

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

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

if (comparison == 0)

return products[mid];

else if (comparison < 0)

left = mid + 1;

else

right = mid - 1;

}

return null;

}

}

using System;

class Program

{

static void Main(string[] args)

{

Product[] products = new Product[]

{

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

new Product(102, "T-Shirt", "Apparel"),

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

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

};

// Sort products by ProductName for binary search

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

Console.WriteLine("Searching for 'Laptop'...");

var resultLinear = SearchDemo.LinearSearch(products, "Laptop");

Console.WriteLine("Linear Search Result: " + (resultLinear != null ? resultLinear.ToString() : "Not found"));

var resultBinary = SearchDemo.BinarySearch(products, "Laptop");

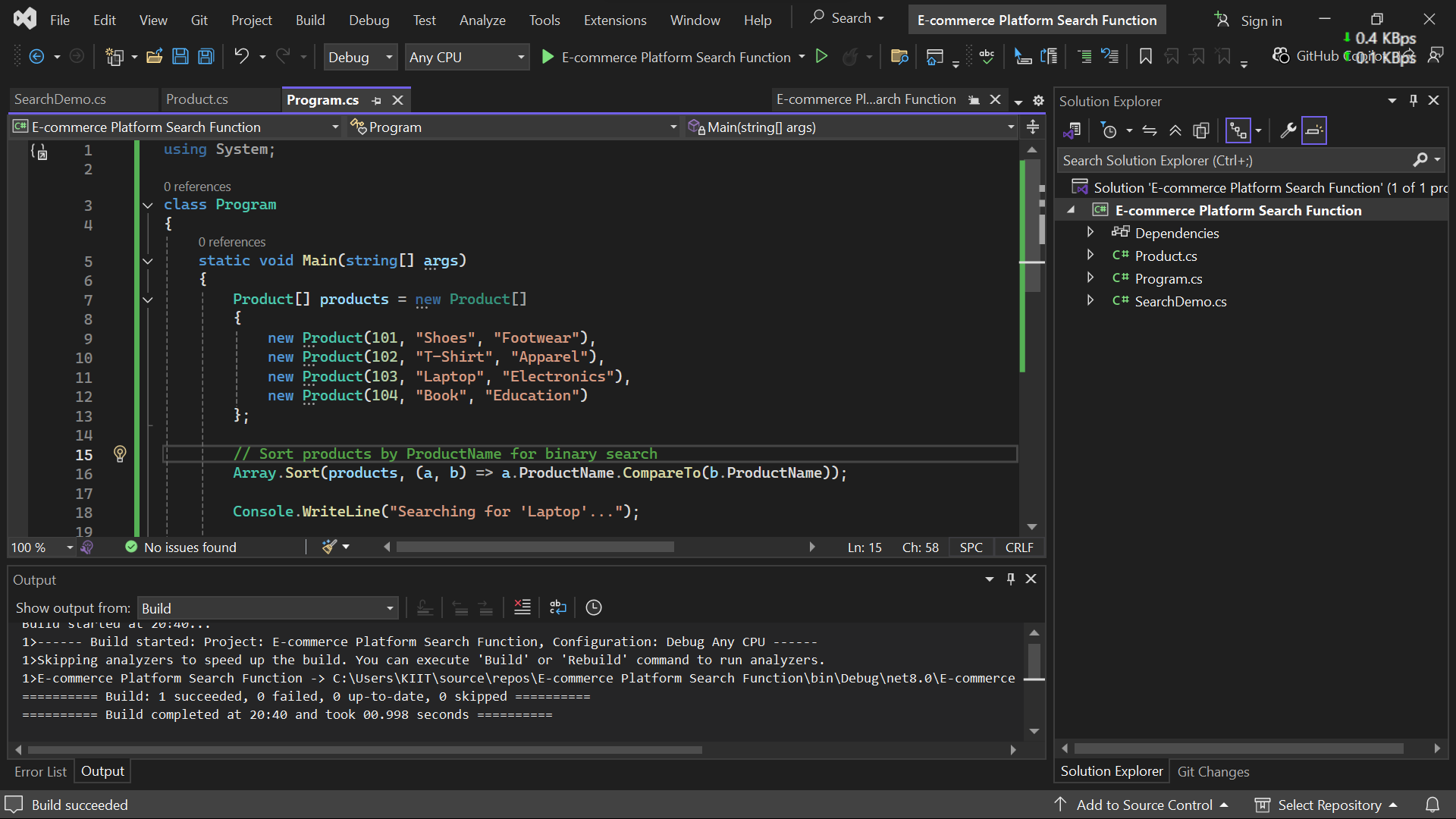
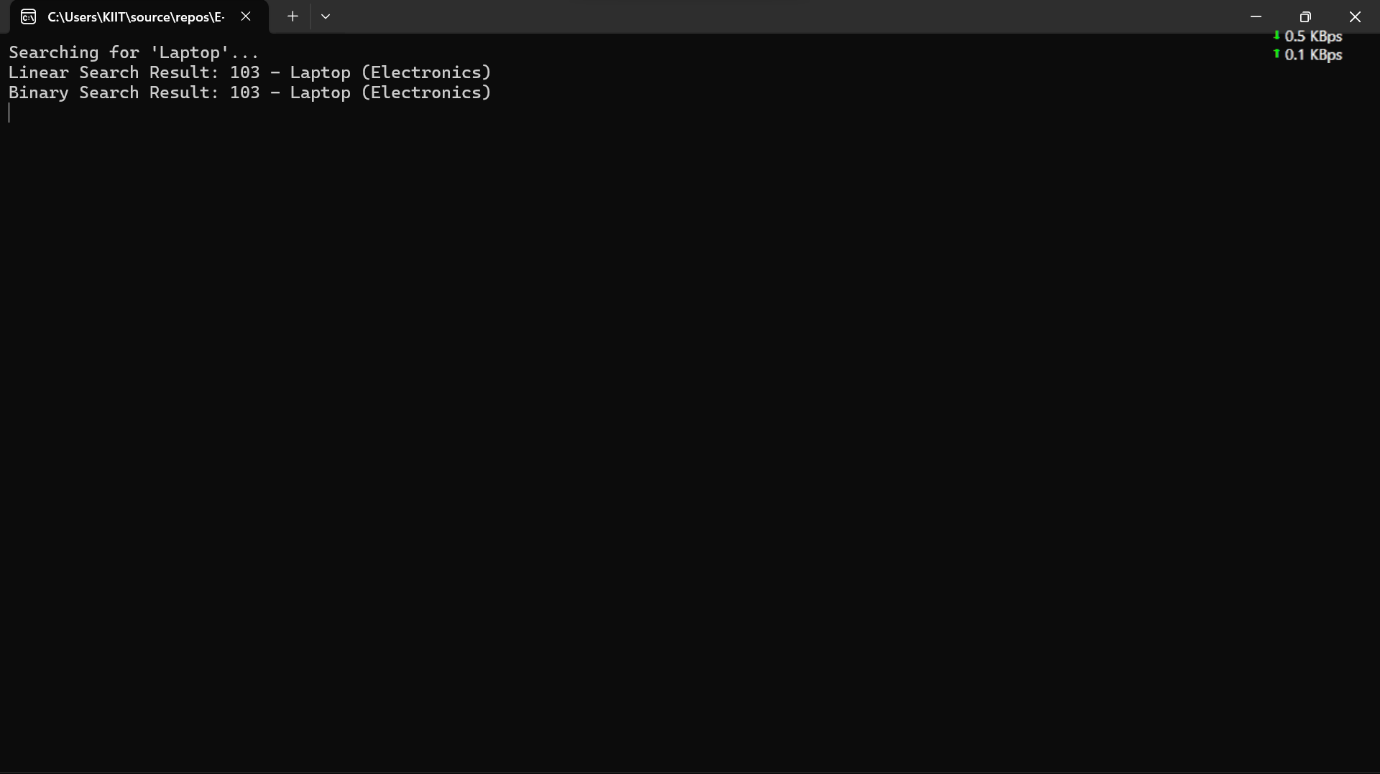
Console.WriteLine("Binary Search Result: " + (resultBinary != null ? resultBinary.ToString() : "Not found"));

Console.ReadLine();

}

}

**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:**

using System;

namespace FinancialForecasting

{

class Forecast

{

// Recursive method to calculate future value

public static double PredictFutureValue(double initialValue, double growthRate, int years)

{

if (years == 0)

return initialValue;

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

}

// Optimized version using memoization

public static double PredictWithMemo(double initialValue, double growthRate, int years, double[] memo)

{

if (years == 0)

return initialValue;

if (memo[years] != 0)

return memo[years];

memo[years] = PredictWithMemo(initialValue, growthRate, years - 1, memo) \* (1 + growthRate);

return memo[years];

}

}

class Program

{

static void Main(string[] args)

{

double initial = 1000;

double rate = 0.10; // 10%

int years = 5;

Console.WriteLine("Recursive Forecast:");

double result = Forecast.PredictFutureValue(initial, rate, years);

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

Console.WriteLine("\nOptimized Recursive Forecast (Memoization):");

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

double optimizedResult = Forecast.PredictWithMemo(initial, rate, years, memo);

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

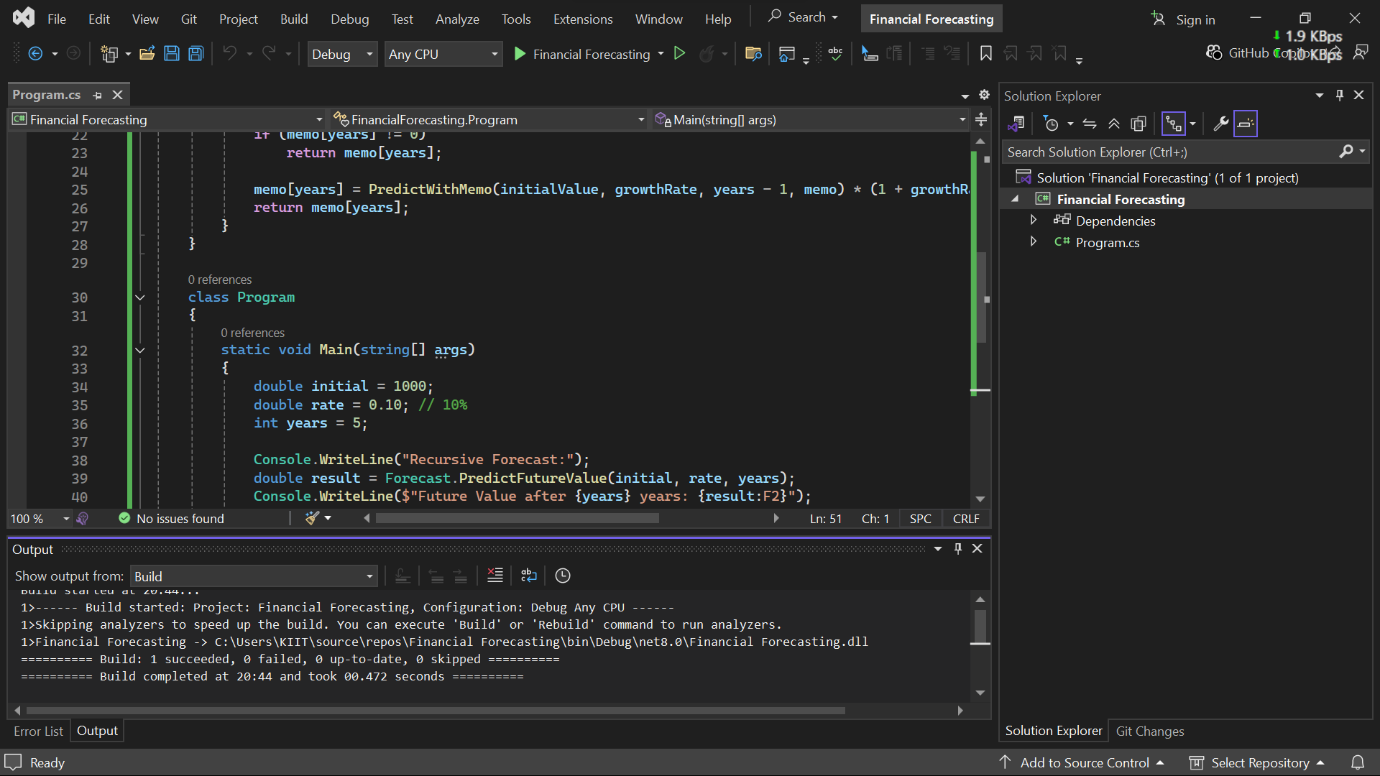
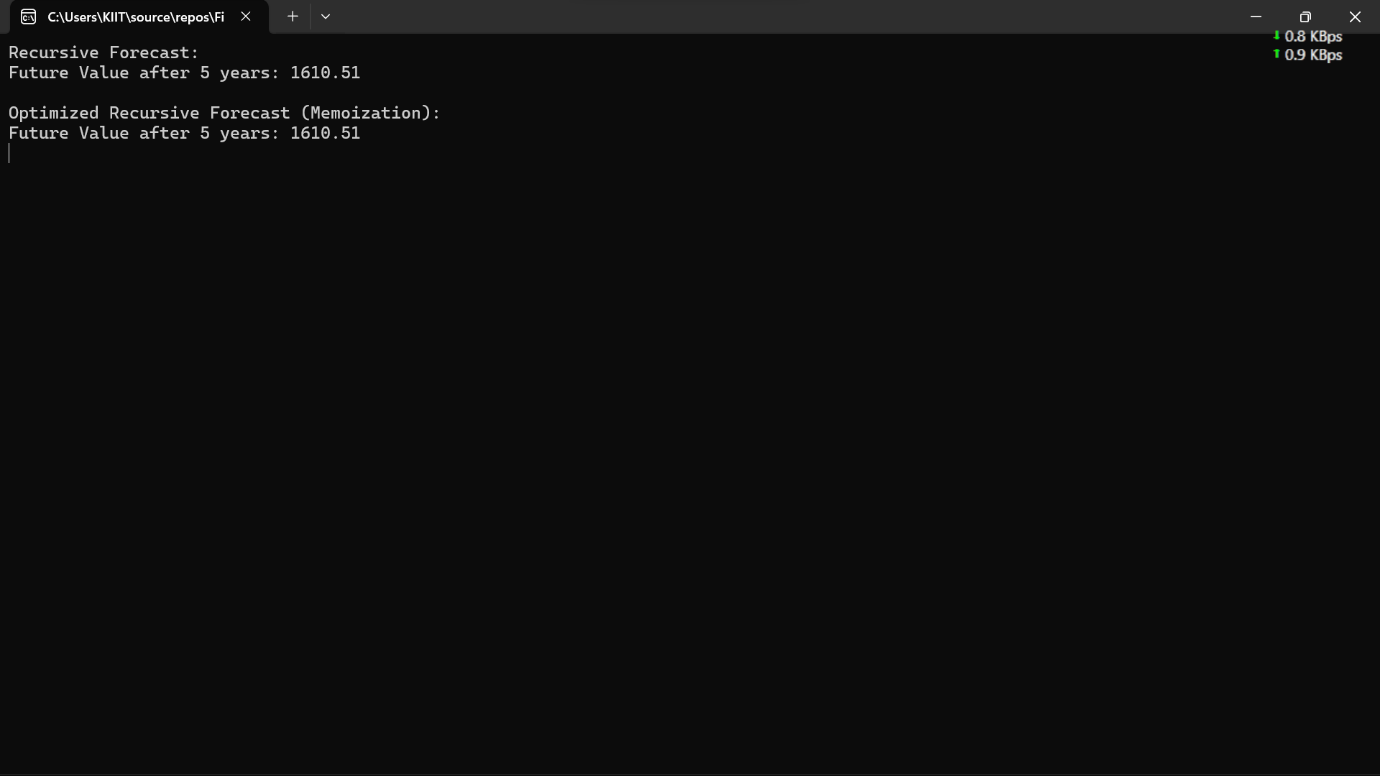
Console.ReadLine();

}

}

}

**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.