WEEK-1 ASSIGNMENTS

SKILL:-DATA STRUCTURE AND ALGORITHM

Exercise 2: E-commerce Platform Search Function

CODE:-

product.cs:-

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;

    }

}

Program.cs:-

using System;

class Program

{

    static void Main()

    {

        Product[] products = new Product[]

        {

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

            new Product(2, "Phone", "Electronics"),

            new Product(3, "Shoes", "Footwear"),

            new Product(4, "Watch", "Accessories")

        };

        Console.WriteLine("Searching using Linear Search:");

        var linearResult = SearchAlgorithms.LinearSearch(products, "Phone");

        Console.WriteLine(linearResult != null ? $"Found: {linearResult.ProductName}" : "Not Found");

        Array.Sort(products, (x, y) => x.ProductName.CompareTo(y.ProductName));

        Console.WriteLine("Searching using Binary Search:");

        var binaryResult = SearchAlgorithms.BinarySearch(products, "Phone");

        Console.WriteLine(binaryResult != null ? $"Found: {binaryResult.ProductName}" : "Not Found");

    }

}

SearchAlgorithms.cs:-

using System;

public static class SearchAlgorithms

{

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

    {

        foreach (var product in products)

        {

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

                return product;

        }

        return null;

    }

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

    {

        int left = 0;

        int right = products.Length - 1;

        while (left <= right)

        {

            int mid = (left + right) / 2;

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

            if (comparison == 0)

                return products[mid];

            else if (comparison < 0)

                left = mid + 1;

            else

                right = mid - 1;

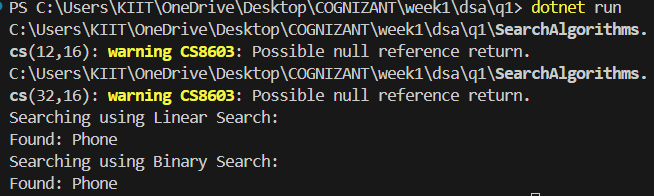
        }

        return null;

    }

}

OUTPUT:-



1. **Understand Asymptotic Notation:**
   * Explain Big O notation and how it helps in analyzing algorithms.

Ans:- **Big O notation** is a mathematical way to describe how the performance of an algorithm changes as the size of the input increases. It focuses on the growth rate rather than exact execution time, making it easier to compare the efficiency of different algorithms. By using Big O, developers can estimate how well an algorithm will scale and whether it’s suitable for large datasets. For example, an algorithm with O(n) complexity will take twice as long if the input size doubles, while one with O(log n) grows much slower and is more efficient for large inputs.

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

Ans:-

**Best Case:** This is when the target item is found immediately or very early in the search. For linear search, this happens if the item is the first element. For binary search, it's when the item is exactly in the middle of the sorted list.

**Average Case:** This represents a typical or expected scenario, where the item is found somewhere in the middle of the data or after a few comparisons. It's useful for estimating how the algorithm performs under normal conditions.

**Worst Case:** This occurs when the item is not in the list or is located at the very end (in linear search), or after the maximum number of divisions (in binary search). Worst-case analysis is important because it shows the algorithm’s maximum time requirement, ensuring the system can handle the slowest outcomes.

1. **Analysis:**
   * Compare the time complexity of linear and binary search algorithms.

Ans:-Linear search has a **time complexity of O(n)**, where n is the number of elements in the collection. This means that in the worst case, the algorithm may need to check every single item to find the target or determine it's not present. On the other hand, binary search has a **time complexity of O(log n)**, which is much faster because it repeatedly divides the search space in half. However, binary search only works efficiently when the data is already **sorted**.

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

Ans:-For an e-commerce platform that handles a **large number of products**, **binary search** is generally more suitable because of its superior performance. Since product data can be sorted by name, ID, or category in advance, binary search allows users to find items quickly, even in a massive catalog. While linear search is simpler and doesn’t require sorting, its slower performance makes it less ideal for platforms where speed and scalability are critical for user experience.

Exercise 7: Financial Forecasting

CODE:-

Forecast.cs:-

public static class Forecast

{

    public static double CalculateFutureValueRecursive(double presentValue, double growthRate, int years)

    {

        if (years == 0)

            return presentValue;

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

    }

    public static double CalculateFutureValueIterative(double presentValue, double growthRate, int years)

    {

        for (int i = 0; i < years; i++)

        {

            presentValue \*= (1 + growthRate);

        }

        return presentValue;

    }

}

Program.cs:-

using System;

class Program

{

    static void Main()

    {

        double presentValue = 10000.0;

        double growthRate = 0.08;

        int years = 5;

        double futureValueRecursive = Forecast.CalculateFutureValueRecursive(presentValue, growthRate, years);

        Console.WriteLine($"[Recursive] Future value after {years} years: ₹{futureValueRecursive:F2}");

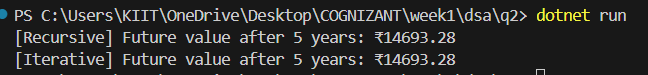
        double futureValueIterative = Forecast.CalculateFutureValueIterative(presentValue, growthRate, years);

        Console.WriteLine($"[Iterative] Future value after {years} years: ₹{futureValueIterative:F2}");

    }

}

OUTPUT:-



1. **Understand Recursive Algorithms:**
   * Explain the concept of recursion and how it can simplify certain problems.

Ans:-Recursion is a programming concept where a method calls itself to solve a smaller version of a problem. It simplifies problems that have a repetitive or self-similar structure, such as computing factorials, Fibonacci sequences, or navigating nested data.

1. **Analysis:**
   * Discuss the time complexity of your recursive algorithm.

Ans:-The recursive algorithm to calculate future value has a **time complexity of O(n)**, where n is the number of years. This is because the function makes one recursive call for each year, reducing the remaining time by one at each step until it reaches zero.

* + Explain how to optimize the recursive solution to avoid excessive computation.

Ans:-To avoid excessive computation and improve performance, the recursive solution can be **converted to an iterative version**. This eliminates the overhead of multiple function calls and stack usage, making the algorithm more efficient and easier to manage, especially for large input sizes. In real-world applications, an **iterative approach** is preferred for problems involving simple repetitive calculations like compound growth.