Week -1 Data structures and Algorithms

Q2.**E-commerce Platform Search Function**

Step 1: Understanding Asymptotic Notation

Big O describes **how the runtime of an algorithm grows** as the input size increases:

· O(1) – Constant time

· O(n) – Linear time

· O(log n) – Logarithmic time

· O(n log n) – Linearithmic time

· O(n^2) – Quadratic time

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

    {

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

        {

            foreach (var product in products)

            {

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

                    return product;

            }

            return null;

        }

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

        {

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

            while (left <= right)

            {

                int mid = (left + right) / 2;

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

                if (comparison == 0)

                    return sortedProducts[mid];

                else if (comparison < 0)

                    left = mid + 1;

                else

                    right = mid - 1;

            }

            return null;

        }

        static void Main(string[] args)

        {

            Product[] products = {

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

                new Product(2, "Shampoo", "Personal Care"),

                new Product(3, "Notebook", "Stationery"),

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

                new Product(5, "Pen", "Stationery")

            };

            Console.WriteLine("Linear Search for 'Phone':");

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

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

            Console.WriteLine("\nSorting for Binary Search...");

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

            Console.WriteLine("Binary Search for 'Phone':");

            var binaryResult = BinarySearch(sortedProducts, "Phone");

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

            Console.WriteLine("\nTime Complexity:");

            Console.WriteLine("Linear Search: O(n) — Simple but slow for large data.");

            Console.WriteLine("Binary Search: O(log n) — Fast, but needs sorted data.");

        }

    }

}

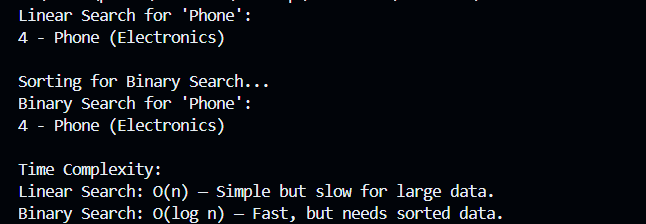
Step 5: Analysis

· **Linear Search** is simple but **slow for large datasets** (O(n)).

· **Binary Search** is **faster (**O(log n)**), but requires sorted data**.

· Use **Linear Search** for **unsorted or small datasets**.

· Use **Binary Search** when **data is sorted and performance matters**, like in product search for an e-commerce platform.

Question 2 OUTPUT

Q7.**Financial Forecasting**

Step 1: Understand Recursion

**Recursion** is when a method calls itself to solve smaller sub-problems. It's useful in problems where the solution depends on results from previous steps — such as **predicting future financial values from past growth**

using System;

using System.Collections.Generic;

class FinancialForecast

{

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

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

    {

        if (year == 0)

            return initialValue;

        // Check if value already calculated

        if (memo.ContainsKey(year))

            return memo[year];

        double prevValue = PredictFutureValue(year - 1, initialValue, growthRate);

        double futureValue = prevValue \* (1 + growthRate);

        memo[year] = futureValue;

        return futureValue;

    }

    static void Main(string[] args)

    {

        double initialInvestment = 1000.0;

        double annualGrowthRate = 0.08; // 8%

        int forecastYears = 10;

        Console.WriteLine("📈 Financial Forecast:");

        for (int year = 0; year <= forecastYears; year++)

        {

            double value = PredictFutureValue(year, initialInvestment, annualGrowthRate);

            Console.WriteLine($"Year {year}: ${value:F2}");

        }

        Console.WriteLine("\n✅ Time Complexity:");

        Console.WriteLine("- Without memoization: O(2^n) due to repeated calls.");

        Console.WriteLine("- With memoization: O(n) since each year is computed once.");

    }

}

QUESTION ANALYSIS:

Without memoization, recursion would recompute values for each year multiple times (O(2^n)).  
With memoization (using Dictionary<int, double>), each unique year is computed only once (O(n)), greatly improving performance.

Question 7 OUTPUT

