**Exercise 2: E-commerce Platform Search Function**

**Big O notation:**

It describes how the time or space requirements of an algorithm increase as the input size grows. Instead of providing precise values, it emphasizes the upper bound of an algorithm's performance, especially in the worst-case scenario. Denoted as O(f(n)), where f(n) indicates the number of operations in relation to the input size n, it facilitates the comparison of the efficiency of algorithms or data structures by showcasing their long-term behavior as the input scales. Consequently, Big O serves as an essential tool for analyzing and enhancing algorithmic performance.

**Search Case Scenarios**

* **Best Case:** Search finds the target early.
* **Average Case:** Search hits the middle of the dataset.
* **Worst Case:** Search finds the target at the end or not at all.

**SOURCE CODE:**

**i)Program.cs**

using System;

using System.Collections.Generic;

using System.Linq;

class Program

{

    static void Main()

    {

        var products = new List<Product>

        {

            new Product(101, "Glasses", "Accessories"),

            new Product(102, "Skirt", "Apparel"),

            new Product(103, "Watch", "Accessories"),

            new Product(104, "Bag", "Luggage")

        };

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

        Console.WriteLine("Linear Search Found: " + (linearResult != null ? linearResult : "Not Found"));

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

        var binaryResult = SearchAlgorithms.BinarySearch(sortedProducts, "Skirt");

        Console.WriteLine("Binary Search Found: " + (binaryResult != null ? binaryResult : "Not Found"));

    }

}

**ii) Searching.cs**using System;

using System.Collections.Generic;

public static class SearchAlgorithms

{

    public static Product LinearSearch(List<Product> products, string searchName)

    {

        foreach (var product in products)

        {

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

                return product;

        }

        return null;

    }

    public static Product BinarySearch(List<Product> products, string searchName)

    {

        int low = 0;

        int high = products.Count - 1;

        while (low <= high)

        {

            int mid = (low + high) / 2;

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

            if (comparison == 0)

                return products[mid];

            else if (comparison < 0)

                low = mid + 1;

            else

                high = mid - 1;

        }

        return null;

    }

}

**iii) Product class**

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()

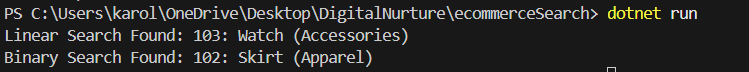
    {

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

    }

}

**CODE OUTPUT:**

****

**ANALYSIS:**Linear Search is suitable for small/unsorted lists. Time complexity: O(n)

Binary Search is faster (O(log n)) than linear search but needs sorted data.

**Exercise 7: Financial Forecasting**

**SOURCE CODE:**

**i)Program.cs**

using System;

class Program

{

    static void Main()

    {

        double presentValue = 50000;

        double annualRate = 0.08;

        int years = 7;

        double futureValue = FinancialForecast.CalculateFutureValue(presentValue, annualRate, years);

        Console.WriteLine($"Present value: {presentValue}");

        Console.WriteLine($"Annual Growth Rate: {annualRate}");

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

    }

}

**ii) Forecasting.cs**

public static class FinancialForecast

{

    public static double CalculateFutureValue(double presentValue, double rate, int years)

    {

        if (years == 0)

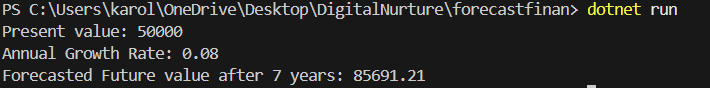
            return presentValue;

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

    }

}

**CODE OUTPUT:**

****