WEEK 1 Data Structures and Algorithm Hands On

Exercise 2: E-commerce Platform Search Function

**Explanation of Big O Notation:**

Big O notation describes how the runtime of an algorithm grows as the size of the input increases. It helps us predict performance and analyse optimisation.

| **Case** | **Description** | **Search** |
| --- | --- | --- |
| **Best** | Minimum work needed | Item is at the beginning |
| **Average** | On average | Item is somewhere in the middle |
| **Worst** | Most work needed | Item is not present or at the end |

**Code:**

using System;

class Product

{

    public int ProductId;

    public string ProductName;

    public string Category;

    public Product(int id, string name, string category)

    {

        ProductId = id;

        ProductName = name;

        Category = category;

    }

    public void Display()

    {

        Console.WriteLine($"ID: {ProductId}, Name: {ProductName}, Category: {Category}");

    }

}

class SearchDemo

{

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

    {

        foreach (var product in products)

        {

            if (product.ProductName == name)

            {

                return product;

            }

        }

        return null;

    }

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

    {

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

        while (left <= right)

        {

            int mid = (left + right) / 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;

    }

    static void Main()

    {

        Product[] productsLinear = {

            new Product(101, "Laptop", "Electronics"),

            new Product(102, "Chair", "Furniture"),

            new Product(103, "Shoes", "Apparel"),

            new Product(104, "Mobile", "Electronics")

        };

        Product[] productsBinary = {

            new Product(102, "Chair", "Furniture"),

            new Product(101, "Laptop", "Electronics"),

            new Product(104, "Mobile", "Electronics"),

            new Product(103, "Shoes", "Apparel")

        };

        string searchName = "Mobile";

        Console.WriteLine("Linear Search:");

        var resultLinear = LinearSearch(productsLinear, searchName);

        if (resultLinear != null)

            resultLinear.Display();

        else

            Console.WriteLine("Product not found.");

        Console.WriteLine("\nBinary Search (on sorted list):");

        var resultBinary = BinarySearch(productsBinary, searchName);

        if (resultBinary != null)

            resultBinary.Display();

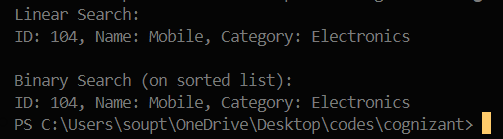
        else

            Console.WriteLine("Product not found.");

    }

}

**Output:**

****

**Analysis:**

| **Search Method** | **Time Complexity** | **When to Use** |
| --- | --- | --- |
| **Linear Search** | **O(n)** | **Small lists or unsorted data** |
| **Binary Search** | **O(log n)** | **Large, sorted lists** |

Binary Search is faster, but only works if your list is sorted. Linear Search is flexible but slower for large datasets.

Exercise 7: Financial Forecasting

Recursion is when a function calls itself to solve smaller parts of a problem. It makes code shorter and easier to understand, especially for repetitive calculations.

**Code:**

using System;

class FinancialForecast

{

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

    {

        if (years == 0)

            return presentValue;

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

    }

    static void Main()

    {

        double presentValue = 1000;

        double annualGrowthRate = 0.05;

        int years = 5;

        double futureValue = PredictFutureValue(presentValue, annualGrowthRate, years);

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

    }

}

**Output:**

****

* The time complexity is O(n), because the function calls itself once for every year.
* Optimization Idea:
  + If the same year’s result is calculated repeatedly (not in this simple case, but in others), it causes redundant work.
  + To optimize, we can:
  + Use Memoization to store results.
* Or convert to an iterative solution (loop-based).