**Submitted By :** 6364888

**Data Structures and Algorithms**

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

1.Understand Asymptotic Notation:

i.Explain Big O notation and how it helps in analyzing algorithms.

Big O notation is a way to describe the performance of an algorithm by measuring how much its execution time or memory usage grows with the input size. It considers the worst-case scenario and ignores constant factors, which helps programmers understand the scalability of their code. With algorithm analysis using Big O, we can choose the best performing algorithm, especially for large datasets.

Big O helps by:  
 a.Measuring performance as the size of input increases.  
 b.Comparing different algorithms for one function.  
 c.Identifying bottlenecks and optimizing code.  
 d.Technical interviews and system design preparation.  
 e.Estimating scalability in real-world applications.

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

Best, Average, and Worst Cases for Search Operations:

|  |  |  |  |
| --- | --- | --- | --- |
| Algorithm | Best Case | Average Case | Worst Case |
| Linear Search | O(1) | O(n) | O(n) |
| Binary Search | O(1) | O(logn) | O(logn) |

2.Product Class (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;

    }

public override string ToString()

    {

        return $"ID: {ProductId}, Name: {ProductName}, Category: {Category}";

    }

}

## **3.** Search Algorithms (SearchAlgorithms.cs)

public static class SearchAlgorithms

{

    public static Product? LinearSearch(Product[] products, string targetName)

    {

        foreach (var product in products)

        {

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

                return product;

        }

        return null;

    }

public static Product? BinarySearch(Product[] products, string targetName)

    {

        int low = 0, high = products.Length - 1;

  while (low <= high)

        {

            int mid = (low + high) / 2;

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

if (cmp == 0) return products[mid];

            else if (cmp < 0) low = mid + 1;

            else high = mid - 1;

        }

  return null;

    }

}

## 4. Program.cs

class Program

{

    static void Main(string[] args)

    {

        Product[] products = new Product[]

        {

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

            new Product(102, "Shirt", "Clothing"),

            new Product(103, "Book", "Stationery"),

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

            new Product(105, "Shoes", "Footwear")

        };

string searchTerm = "Phone";

Console.WriteLine("🔍 Linear Search:");

        var linearResult = SearchAlgorithms.LinearSearch(products, searchTerm);

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

Array.Sort(products, (p1, p2) => p1.ProductName.CompareTo(p2.ProductName));

        Console.WriteLine("\n🔍 Binary Search (sorted by ProductName):");

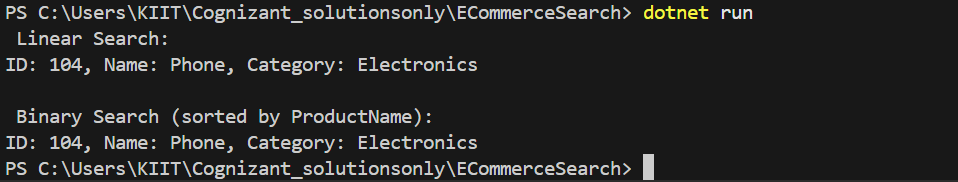
        var binaryResult = SearchAlgorithms.BinarySearch(products, searchTerm);

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

    }

}

## 5. Output Screenshot



## 6. Analysis

Time Complexity Comparison:Time Complexity

|  |  |
| --- | --- |
| Algorithm | Time Complexity |
| Linear Search | O(n) |
| Binary Search | O(log n) |

Conclusion:  
Binary search is more efficient for large, sorted datasets, while linear search is suitable for small or unsorted data.

# **Exercise 7: Financial Forecasting**

## Understanding Recursive Algorithms

1. Explain the concept of recursion and how it can simplify certain problems.

Recursion is a programming technique where a function calls itself to solve smaller instances of a problem.  
It is useful in problems like factorial, Fibonacci, tree traversal, and financial forecasting where each future value depends on the previous one.

## 2. Recursive Method Setup

The following method calculates the future value of an investment using a recursive approach.

### FinanceForecast.cs

public static class FinanceForecast

{

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

    {

        if (years == 0)

            return initialValue;

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

    }

}

## 3. Full Implementation (Program.cs)

class Program

{

    static void Main(string[] args)

    {

        double initialValue = 1000; // starting value

        double rate = 0.05;         // 5% growth rate

        int years = 5;

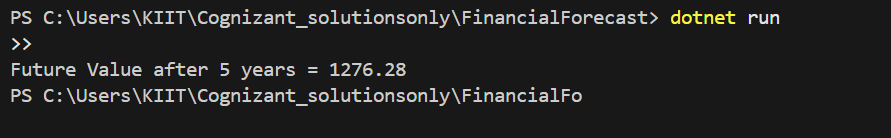
        double futureValue = FinanceForecast.PredictFutureValue(initialValue, rate, years);

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

    }

}

## Output Screenshot



## 5. Analysis

Time Complexity: O(n), where n is the number of years.

Each recursive call computes the value for one year, making the recursion depth equal to the number of years.

Optimization Tip: For large inputs, recursion may be inefficient. Use an iterative version or memoization to reduce stack overhead.