**Algorithms\_Data Structures:-**

**Question 1:-**

E-commerce Platform Search Function

Scenario:

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

Steps:

1. Understand Asymptotic Notation:

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

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

2. Setup:

o Create a class Product with attributes for searching, such as productId, productName, and category.

3. Implementation:

o Implement linear search and binary search algorithms.

o Store products in an array for linear search and a sorted array for binary search.

4. Analysis:

o Compare the time complexity of linear and binary search algorithms.

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

**Solution:**

## Let’s see Asymptotic Notation

## Big O Notation

Big O notation is used to describe the **efficiency** of an algorithm by expressing its **upper bound time or space complexity** in terms of input size n.

**O(1)**: Constant time

**O(n)**: Linear time

**O(log n)**: Logarithmic time

**O(n log n), O(n²)**, etc.: Higher complexities

### Best, Average, and Worst-Case for Search

| **Case** | **Linear Search** | **Binary Search** |
| --- | --- | --- |
| Best Case | O(1) (found early) | O(1) (middle element) |
| Average | O(n/2) → O(n) | O(log n) |
| Worst Case | O(n) | O(log n) |

1. **Setting up: Product Class**

public class Product

{

public int ProductId { get; set; }

public string ProductName { get; set; }

public string Category { get; set; }

public Product(int productId, string productName, string category)

{

ProductId = productId;

ProductName = productName;

Category = category;

}

public override string ToString()

{

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

}

}

1. **Implementation:**

* **Linear Search:-**

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

{

foreach (var product in products)

{

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

{

return product;

}

}

return null;

}

* **Binary Search:-**

For binary search, the array must be sorted **alphabetically by product name**.

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

{

int left = 0;

int right = products.Length - 1;

while (left <= right)

{

int mid = (left + right) / 2;

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

if (comparison == 0)

{

return products[mid];

}

else if (comparison < 0)

{

left = mid + 1;

}

else

{

right = mid - 1;

}

}

return null;

}

**Program.cs:-**

using System;

using System.Linq;

public class Product

{

    public int ProductId { get; set; }

    public string ProductName { get; set; }

    public string Category { get; set; }

    public Product(int productId, string productName, string category)

    {

        ProductId = productId;

        ProductName = productName;

        Category = category;

    }

    public override string ToString()

    {

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

    }

}

class Program

{

    static void Main()

    {

        Product[] products = new Product[]

        {

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

            new Product(2, "Shoes", "Fashion"),

            new Product(3, "Watch", "Accessories"),

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

            new Product(5, "Bag", "Fashion")

        };

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

        var resultLinear = LinearSearch(products, "Camera");

        Console.WriteLine(resultLinear != null ? resultLinear.ToString() : "Not found");

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

        Console.WriteLine("\n🔍 Binary Search for 'Camera':");

        var resultBinary = BinarySearch(sortedProducts, "Camera");

        Console.WriteLine(resultBinary != null ? resultBinary.ToString() : "Not found");

    }

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

    {

        foreach (var product in products)

        {

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

                return product;

        }

        return null;

    }

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

    {

        int left = 0;

        int right = products.Length - 1;

        while (left <= right)

        {

            int mid = (left + right) / 2;

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

            if (comparison == 0)

                return products[mid];

            else if (comparison < 0)

                left = mid + 1;

            else

                right = mid - 1;

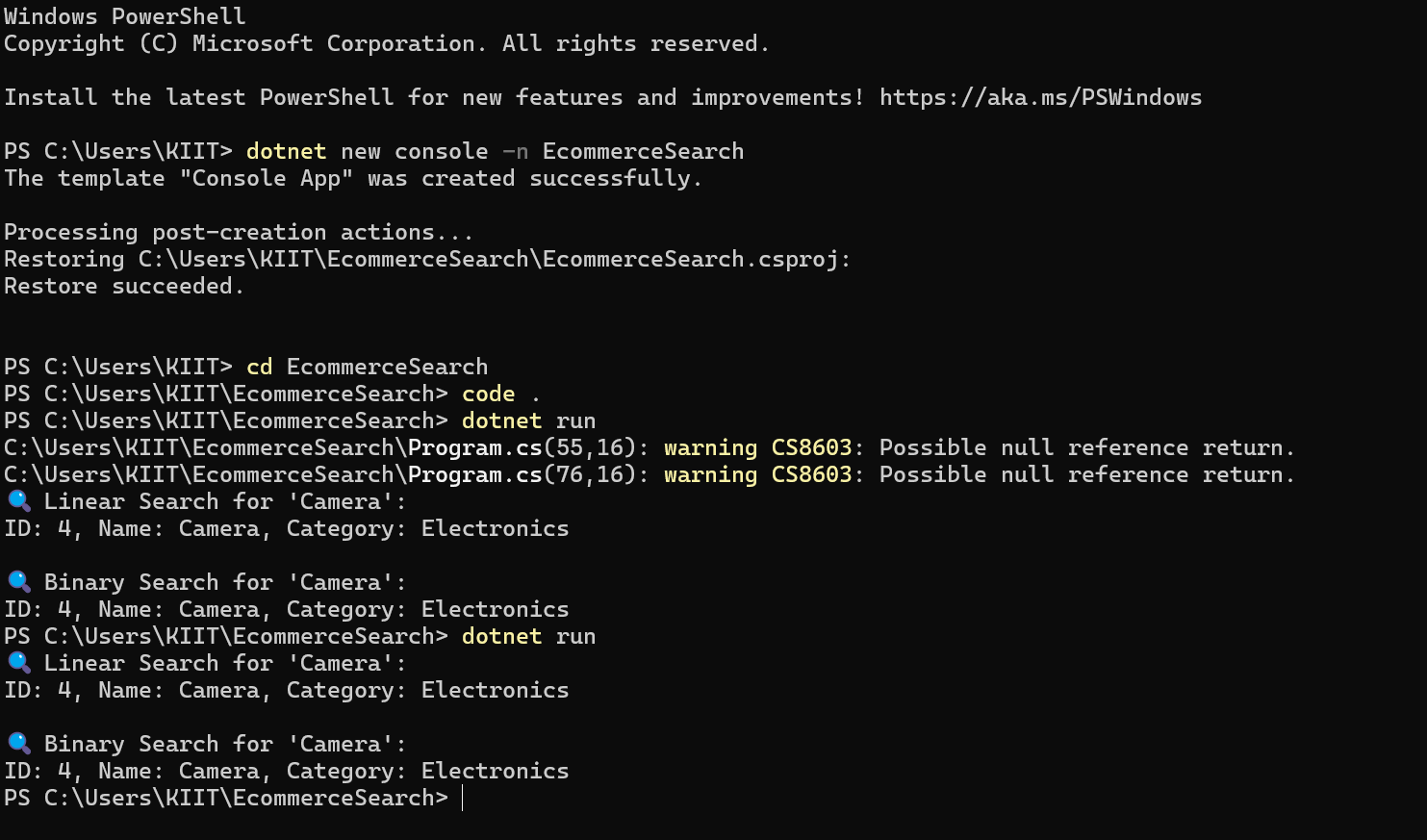
        }

        return null;

    }

}

**Output::-**

****

## 4. Analysis and Comparison

| **Criteria** | **Linear Search** | **Binary Search** |
| --- | --- | --- |
| Time Complexity | O(n) | O(log n) |
| Requires Sorted Data | No | Yes |
| Flexibility | Good for small unsorted arrays | Needs sorting before |
| Performance (Large Data) | Slower | Faster |

### **Conclusion:-**

Use **Linear Search** for:

Small datasets

Data not sorted or can’t be sorted easily

Use **Binary Search** for:

Large datasets

When search speed is critical and data can be kept sorted

For a real-world **e-commerce platform**, binary search is **more suitable** due to large product data and need for performance.

**Question 2:-**

Financial Forecasting

Scenario:

You are developing a financial forecasting tool that predicts future values based on past data.

Steps:

1. Understand Recursive Algorithms:

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

2. Setup:

o Create a method to calculate the future value using a recursive approach.

3. Implementation:

o Implement a recursive algorithm to predict future values based on past growth rates.

4. Analysis:

o Discuss the time complexity of your recursive algorithm.

o Explain how to optimize the recursive solution to avoid excessive computation.

**Solution:-**

## 1. Let’s Understand Recursive Algorithms

**Recursion** is a programming technique where a method calls itself to solve a smaller instance of the same problem.

### Example Use-Cases:

* Calculating factorial
* Traversing trees
* Fibonacci series
* Solving mathematical recurrence relations (like financial growth!)

### Why Use Recursion?

Recursion can make code shorter and more intuitive for problems that naturally fit a divide-and-conquer or self-similar pattern.

## 2. Seting up – Recursive Future Value Calculation

We assume a simple model where the value grows at a constant rate r per period.

The **recursive formula** for future value:

FV(n) = FV(n-1) \* (1 + r)

Where:

* FV(n) is the future value at year n
* r is the growth rate (e.g., 0.05 for 5%)
* Base case: FV(0) = initial amount

1. **Implementation:-**

**Program.cs:-**

using System;

class FinancialForecast

{

    public static double PredictFutureValue(int year, double initialAmount, double rate)

    {

        if (year == 0)

            return initialAmount;

        return PredictFutureValue(year - 1, initialAmount, rate) \* (1 + rate);

    }

    static void Main(string[] args)

    {

        Console.Write("Enter initial amount: ");

        double initialAmount = Convert.ToDouble(Console.ReadLine());

        Console.Write("Enter annual growth rate (in %): ");

        double ratePercent = Convert.ToDouble(Console.ReadLine());

        double rate = ratePercent / 100;

        Console.Write("Enter number of years to forecast: ");

        int years = Convert.ToInt32(Console.ReadLine());

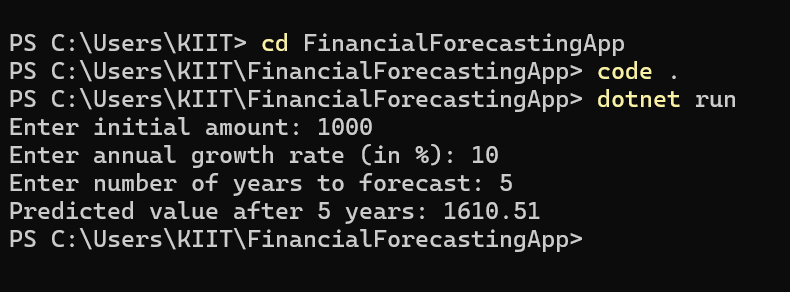
        double futureValue = PredictFutureValue(years, initialAmount, rate);

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

    }

}

**Output:-**

****

1. **Analysis:-**

### Time Complexity

The recursive function calls itself once for each year.

So, **Time Complexity** = **O(n)**, where n is the number of years.

### Space Complexity

Because recursion adds to the **call stack**, the **space complexity** is also **O(n)**.

## **For Optimization: We can do Memoization or Iteration**

To avoid redundant calculations and stack overflow in deeper recursions, we can use **memoization** or simply convert it to an **iterative** version:

**Iterative Version:-**

public static double PredictFutureValueIterative(int years, double initialAmount, double rate)

{

double futureValue = initialAmount;

for (int i = 1; i <= years; i++)

{

futureValue \*= (1 + rate);

}

return futureValue;

}

**Memoized Recursive Version:-**

public static double PredictFutureValueMemo(int year, double initialAmount, double rate, double[ ] memo)

{

if (year == 0)

return initialAmount;

if (memo[year] != 0)

return memo[year];

memo[year] = PredictFutureValueMemo(year - 1, initialAmount, rate, memo) \* (1 + rate);

return memo[year];

}

## Summary

| **Approach** | **Time Complexity** | **Space Complexity** | **Suitable For** |
| --- | --- | --- | --- |
| Basic Recursion | O(n) | O(n) | Educational, simple |
| Iterative | O(n) | O(1) | Efficient and preferred |
| Memoization | O(n) | O(n) | Recursive but faster |