WEEK 1 : Algorithms_Data Structures

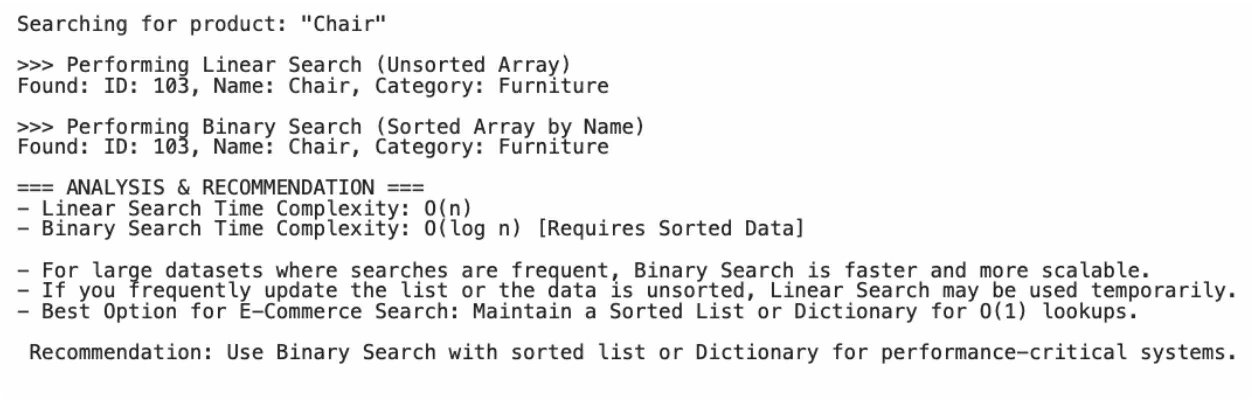
Code: E-commerce Platform Search Function

```
using System;

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}";

   }

}


public class SearchEngine

{

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

   {

      foreach (var product in products)

      {

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

```
            return product;
        }
        return null;
    }
    public static Product BinarySearch(Product[] products, string searchName)
    {
        int left = 0;
        int right = products.Length - 1;

        while (left <= right)
        {
            int mid = left + (right - left) / 2;
            int compare = string.Compare(products[mid].ProductName, searchName, true);

            if (compare == 0)
                return products[mid];
            else if (compare < 0)
                left = mid + 1;
            else
                right = mid - 1;
        }

        return null;
    }
}
class Program
{
    static void Main(string[] args)
    {
        // Sample Product Dataset
        Product[] products = new Product[]
```

```
{
  new Product(101, "Laptop", "Electronics"),
  new Product(102, "Shampoo", "Personal Care"),
  new Product(103, "Chair", "Furniture"),
  new Product(104, "Book", "Stationery"),
  new Product(105, "Mouse", "Electronics"),
  new Product(106, "Table", "Furniture"),
  new Product(107, "Soap", "Personal Care"),
  new Product(108, "Pen", "Stationery"),
  new Product(109, "Monitor", "Electronics"),
  new Product(110, "Bag", "Accessories")
};
string searchTerm = "Chair";
Console.WriteLine($"Searching for product: \"{searchTerm}\"\n");

// LINEAR SEARCH DEMO
Console.WriteLine(">>> Performing Linear Search (Unsorted Array)");
var linearResult = SearchEngine.LinearSearch(products, searchTerm);
Console.WriteLine(linearResult != null ? $"Found: {linearResult}" : "Not found");

// BINARY SEARCH DEMO
// Step 1: Sort by ProductName before binary search
Array.Sort(products, (a, b) => string.Compare(a.ProductName, b.ProductName));

Console.WriteLine("\n>>> Performing Binary Search (Sorted Array by Name)");
var binaryResult = SearchEngine.BinarySearch(products, searchTerm);
Console.WriteLine(binaryResult != null ? $"Found: {binaryResult}" : "Not found");



Console.WriteLine("\n=== ANALYSIS & RECOMMENDATION ===");
Console.WriteLine(
        @"- Linear Search Time Complexity: O(n)

- Binary Search Time Complexity: O(log n) [Requires Sorted Data]


- For large datasets where searches are frequent, Binary Search is faster and more scalable.

- If you frequently update the list or the data is unsorted, Linear Search may be used temporarily.

- Best Option for E-Commerce Search: Maintain a Sorted List or Dictionary for O(1) lookups.

Recommendation: Use Binary Search with sorted list or Dictionary for performance-critical systems.");

  }

}
```

Output :



```
Searching for product: "Chair"

>>> Performing Linear Search (Unsorted Array)
Found: ID: 103, Name: Chair, Category: Furniture

>>> Performing Binary Search (Sorted Array by Name)
Found: ID: 103, Name: Chair, Category: Furniture

=== ANALYSIS & RECOMMENDATION ===
- Linear Search Time Complexity: O(n)
- Binary Search Time Complexity: O(log n) [Requires Sorted Data]

- For large datasets where searches are frequent, Binary Search is faster and more scalable.
- If you frequently update the list or the data is unsorted, Linear Search may be used temporarily.
- Best Option for E-Commerce Search: Maintain a Sorted List or Dictionary for O(1) lookups.

 Recommendation: Use Binary Search with sorted list or Dictionary for performance-critical systems.
```

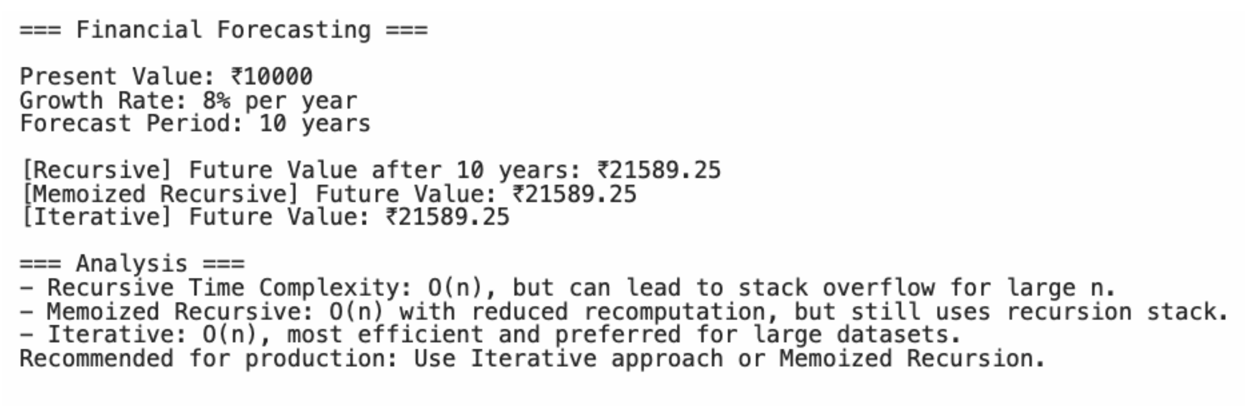
Code: Financial Forecasting

```
using System;

using System.Collections.Generic;

class FinancialForecast

{

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

    {

        if (years == 0)

            return presentValue;


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

    }

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


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

    {

        if (years == 0)

            return presentValue;


        if (memo.ContainsKey(years))

            return memo[years];


        double result = (1 + growthRate) * PredictFutureValueMemoized(presentValue, growthRate, years - 1);

        memo[years] = result;

        return result;

    }
```

```
public static double PredictFutureValueIterative(double presentValue, double growthRate, int years)
{
    double result = presentValue;
    for (int i = 1; i <= years; i++)
    {
        result *= (1 + growthRate);
    }
    return result;
}
static void Main(string[] args)
{
    double presentValue = 10000;
    double annualGrowthRate = 0.08;
    int forecastYears = 10;

    Console.WriteLine("=== Financial Forecasting ===\n");

    Console.WriteLine($"Present Value: ₹{presentValue}");
    Console.WriteLine($"Growth Rate: {annualGrowthRate * 100}% per year");
    Console.WriteLine($"Forecast Period: {forecastYears} years\n");

    double futureValueRecursive = PredictFutureValueRecursive(presentValue, annualGrowthRate, forecastYears);
    Console.WriteLine($"[Recursive] Future Value after {forecastYears} years: ₹{futureValueRecursive:F2}");

    double futureValueMemoized = PredictFutureValueMemoized(presentValue, annualGrowthRate, forecastYears);
    Console.WriteLine($"[Memoized Recursive] Future Value: ₹{futureValueMemoized:F2}");

    double futureValueIterative = PredictFutureValueIterative(presentValue, annualGrowthRate, forecastYears);
    Console.WriteLine($"[Iterative] Future Value: ₹{futureValueIterative:F2}");
```

```
        Console.WriteLine("\n=== Analysis ===");

        Console.WriteLine(@"- Recursive Time Complexity: O(n), but can lead to stack overflow for large n.

- Memoized Recursive: O(n) with reduced recomputation, but still uses recursion stack.

- Iterative: O(n), most efficient and preferred for large datasets.

Recommended for production: Use Iterative approach or Memoized Recursion.");

    }

}
```

Output:



```
=== Financial Forecasting ===

Present Value: ₹10000
Growth Rate: 8% per year
Forecast Period: 10 years

[Recursive] Future Value after 10 years: ₹21589.25
[Memoized Recursive] Future Value: ₹21589.25
[Iterative] Future Value: ₹21589.25

=== Analysis ===
- Recursive Time Complexity: O(n), but can lead to stack overflow for large n.
- Memoized Recursive: O(n) with reduced recomputation, but still uses recursion stack.
- Iterative: O(n), most efficient and preferred for large datasets.
Recommended for production: Use Iterative approach or Memoized Recursion.
```