By Sachin Ray (SuperSet ID: 6364957)

Exercise 2: E-commerce Platform Search Function

Exercise 2: 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:**
    - Explain Big O notation and how it helps in analyzing algorithms.
    - Describe the best, average, and worst-case scenarios for search operations.
2. **Setup:**
    - Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
    - Implement linear search and binary search algorithms.
    - Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
    - Compare the time complexity of linear and binary search algorithms.
    - Discuss which algorithm is more suitable for your platform and why.

SOLUTION :
1--> Answer: Big O notation describes how the performance (time or space) of an algorithm changes as the size of the input increases. It is used to analyze and compare algorithms by focusing on their efficiency, especially in the worst-case scenario, and helps developers choose the most appropriate algorithm for a given problem.

For search operations, the best-case scenario occurs when the target element is found immediately, such as at the beginning of the list, which takes constant time (O(1)) for both linear and binary search. The average-case scenario considers the typical position of the target, which for linear search means looking through about half of the elements (O(n)), while for binary search it remains much faster at O(log n) because it repeatedly halves the search space. The worst-case scenario happens when the target is either not present or is at the last position; this requires checking every element for linear search (O(n)), but only log n steps for binary search (O(log n)), provided the data is sorted. Overall, binary search is much more efficient than linear search for large, sorted datasets.

--> 2,3,4. Setup and Implementation with Analysis

1. Product.cs :

```
using System;

namespace ECommerceSearchSystem
{
    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}";
        }
    }
}
```

2. SearchEngine.cs:

```
using System;
using System.Collections.Generic;

namespace ECommerceSearchSystem
{
    public static class SearchEngine
    {
        // Linear search: O(n)
        public static List<Product> LinearSearchByName(Product[] products, string searchTerm)
        {
            var results = new List<Product>();
            foreach (var product in products)
            {
                if (product.ProductName.Contains(searchTerm,
StringComparison.OrdinalIgnoreCase))
                    results.Add(product);
            }
            return results;
        }
        // Binary search: O(log n), for exact match
        public static Product BinarySearchByName(Product[] sortedProducts, string exactName)
        {
            int left = 0, right = sortedProducts.Length - 1;
            while (left <= right)
            {
                int mid = left + (right - left) / 2;
                int cmp = string.Compare(sortedProducts[mid].ProductName, exactName,
StringComparison.OrdinalIgnoreCase);
                if (cmp == 0) return sortedProducts[mid];
```

```
                if (cmp < 0) left = mid + 1;
                else right = mid - 1;
            }
            return null;
        }
        // Binary search for partial matches (returns all matches)
        public static List<Product> BinarySearchPartialMatch(Product[] sortedProducts, string
searchTerm)
        {
            var results = new List<Product>();
            int left = 0, right = sortedProducts.Length - 1;
            int firstMatch = -1;
            // Find the first matching index
            while (left <= right)
            {
                int mid = left + (right - left) / 2;
                if (sortedProducts[mid].ProductName.Contains(searchTerm,
StringComparison.OrdinalIgnoreCase))
                {
                    firstMatch = mid;
                    right = mid - 1;
                }
                else if (string.Compare(sortedProducts[mid].ProductName, searchTerm,
StringComparison.OrdinalIgnoreCase) < 0)
                {
                    left = mid + 1;
                }
                else
                {
                    right = mid - 1;
                }
            }
            // Collect all subsequent matches
            if (firstMatch != -1)
            {
                for (int i = firstMatch; i < sortedProducts.Length; i++)
                {
                    if (sortedProducts[i].ProductName.Contains(searchTerm,
StringComparison.OrdinalIgnoreCase))
                        results.Add(sortedProducts[i]);
                    else
                        break;
                }
            }
            return results;
        }
    }
}
```

3. Main File: Program.cs :

```
using System;
using System.Linq;
using System.Diagnostics;

namespace ECommerceSearchSystem
{
    class Program
```

```
    {
        static void Main(string[] args)
        {
            Console.WriteLine("=== E-COMMERCE SEARCH FUNCTIONALITY ===\n");
            // Sample data
            var products = GenerateProductData();
            var sortedProducts = products.OrderBy(p => p.ProductName).ToArray();
            // Demonstrate search scenarios
            DemonstrateSearchScenarios(products, sortedProducts);
            // Performance analysis
            PerformanceAnalysis(products, sortedProducts);
            Console.WriteLine("\nPress any key to exit...");
            Console.ReadKey();
        }
        static Product[] GenerateProductData()
        {
            return new Product[]
            {
                new Product(1, "Apple iPhone 14", "Electronics"),
                new Product(2, "Samsung Galaxy S23", "Electronics"),
                new Product(3, "Dell Laptop", "Computers"),
                new Product(4, "Nike Running Shoes", "Sports"),
                new Product(5, "Adidas T-Shirt", "Clothing"),
                new Product(6, "Sony Headphones", "Electronics"),
                new Product(7, "HP Printer", "Office"),
                new Product(8, "Canon Camera", "Electronics"),
                new Product(9, "Puma Sneakers", "Sports"),
                new Product(10, "Levi's Jeans", "Clothing"),
                new Product(11, "Microsoft Surface", "Computers"),
                new Product(12, "Bose Speaker", "Electronics"),
                new Product(13, "Under Armour Hoodie", "Clothing"),
                new Product(14, "Logitech Mouse", "Computers"),
                new Product(15, "Apple Watch", "Electronics"),
                new Product(16, "Reebok Training Shoes", "Sports"),
                new Product(17, "Epson Scanner", "Office"),
                new Product(18, "JBL Earbuds", "Electronics"),
                new Product(19, "Champion Shorts", "Clothing"),
                new Product(20, "Acer Monitor", "Computers")
            };
        }
        static void DemonstrateSearchScenarios(Product[] products, Product[] sortedProducts)
        {
            Console.WriteLine("=== SEARCH DEMONSTRATIONS ===");
            // Linear search
            var linearResults = SearchEngine.LinearSearchByName(products, "iPhone");
            Console.WriteLine("Linear Search for 'iPhone':");
            foreach (var product in linearResults)
                Console.WriteLine($"  Found: {product}");
            // Binary search (exact)
            var binaryResult = SearchEngine.BinarySearchByName(sortedProducts, "Apple iPhone
14");
            Console.WriteLine("\nBinary Search for 'Apple iPhone 14':");
            if (binaryResult != null)
                Console.WriteLine($"  Found: {binaryResult}");
            else
                Console.WriteLine("  Not found");
            // Linear search by category
            var electronics = products.Where(p => p.Category.Equals("Electronics",
StringComparison.OrdinalIgnoreCase)).ToList();
            Console.WriteLine("\nLinear Search for 'Electronics' category:");
            foreach (var product in electronics)
```

```
                Console.WriteLine($"  - {product}");
        }
        static void PerformanceAnalysis(Product[] products, Product[] sortedProducts)
        {
            Console.WriteLine("\n=== PERFORMANCE ANALYSIS ===");
            string[] searchTerms = { "Apple", "Samsung", "Nike", "Electronics" };
            foreach (var term in searchTerms)
            {
                var stopwatch = new Stopwatch();
                // Linear search
                stopwatch.Start();
                var linearResults = SearchEngine.LinearSearchByName(products, term);
                stopwatch.Stop();
                var linearTime = stopwatch.ElapsedTicks;
                // Binary search
                stopwatch.Restart();
                var binaryResult = SearchEngine.BinarySearchByName(sortedProducts, term);
                stopwatch.Stop();
                var binaryTime = stopwatch.ElapsedTicks;
                Console.WriteLine($"\nSearch Term: '{term}'");
                Console.WriteLine($"Linear Search: {linearTime} ticks, Results:
{linearResults.Count}");
                Console.WriteLine($"Binary Search: {binaryTime} ticks, Result:
{(binaryResult != null ? 1 : 0)}");
                if (binaryTime > 0)
                    Console.WriteLine($"Binary Search is {(double)linearTime / binaryTime:F2}x
faster than Linear Search");
            }
        }
    }
}
```

OUTPUT :

```
PS C:\Users\KIIT\OneDrive\Desktop\Cognizant_DeepSkilling_.NET_solutions\1_2_DSA\EcommerceSearchSystem> dotnet run
C:\Users\KIIT\OneDrive\Desktop\Cognizant_DeepSkilling_.NET_solutions\1_2_DSA\EcommerceSearchSystem\SearchEngine.cs(3
rence return.
=== E-COMMERCE SEARCH FUNCTIONALITY ===

=== SEARCH DEMONSTRATIONS ===
Linear Search for 'iPhone':
  Found: ID: 1, Name: Apple iPhone 14, Category: Electronics

Binary Search for 'Apple iPhone 14':
  Found: ID: 1, Name: Apple iPhone 14, Category: Electronics

Linear Search for 'Electronics' category:
  - ID: 1, Name: Apple iPhone 14, Category: Electronics
  - ID: 2, Name: Samsung Galaxy S23, Category: Electronics
  - ID: 6, Name: Sony Headphones, Category: Electronics
  - ID: 8, Name: Canon Camera, Category: Electronics
  - ID: 12, Name: Bose Speaker, Category: Electronics
  - ID: 15, Name: Apple Watch, Category: Electronics
  - ID: 18, Name: JBL Earbuds, Category: Electronics

=== PERFORMANCE ANALYSIS ===

Search Term: 'Apple'
Linear Search: 55 ticks, Results: 2
Binary Search: 8 ticks, Result: 0
Binary Search is 6.88x faster than Linear Search

Search Term: 'Samsung'
Linear Search: 67 ticks, Results: 1
Binary Search: 8 ticks, Result: 0
Binary Search is 8.38x faster than Linear Search

Search Term: 'Nike'
Linear Search: 33 ticks, Results: 1
Binary Search: 4 ticks, Result: 0
Binary Search: 4 ticks, Result: 0
Binary Search is 8.25x faster than Linear Search
```

```
Search Term: 'Electronics'
Linear Search: 1652 ticks, Results: 0
Binary Search: 5 ticks, Result: 0
Binary Search is 330.40x faster than Linear Search

Press any key to exit...



Binary Search: 4 ticks, Result: 0
Binary Search is 8.25x faster than Linear Search

Search Term: 'Electronics'
Linear Search: 1652 ticks, Results: 0
Binary Search: 5 ticks, Result: 0
Binary Search is 330.40x faster than Linear Search

Press any key to exit...


Binary Search: 4 ticks, Result: 0
Binary Search is 8.25x faster than Linear Search

Search Term: 'Electronics'
Linear Search: 1652 ticks, Results: 0
Binary Search: 5 ticks, Result: 0
Binary Search is 330.40x faster than Linear Search

Press any key to exit...
Binary Search: 4 ticks, Result: 0
Binary Search is 8.25x faster than Linear Search

Search Term: 'Electronics'
Linear Search: 1652 ticks, Results: 0
Binary Search: 5 ticks, Result: 0
Binary Search is 330.40x faster than Linear Search
Binary Search: 4 ticks, Result: 0
Binary Search is 8.25x faster than Linear Search


 Search Term: 'Electronics'
 Linear Search: 1652 ticks, Results: 0
 Binary Search: 5 ticks, Result: 0
 Binary Search: 4 ticks, Result: 0
 Binary Search is 8.25x faster than Linear Search

 Search Term: 'Electronics'
 Binary Search: 4 ticks, Result: 0
 Binary Search is 8.25x faster than Linear Search

 Binary Search: 4 ticks, Result: 0
 Binary Search is 8.25x faster than Linear Search

 Binary Search is 8.25x faster than Linear Search


 Search Term: 'Electronics'
 Search Term: 'Electronics'
 Linear Search: 1652 ticks, Results: 0
 Binary Search: 5 ticks, Result: 0
 Binary Search is 330.40x faster than Linear Search

 Press any key to exit...
```

Concluison :

Binary search is optimal for e-commerce platforms with large product catalogs due to its logarithmic scalability. However, these conditions are expected to meet to use this :

- Pre-sorting products by ProductName (O(n log n) initial cost).

-Maintaining sorted order during inventory updates (O(n) per insertion).

Hybrid approach:

-Use binary search for primary product searches.

-Apply linear search for dynamic filters (e.g., category-based searches), as implemented for Electronics category lookups1.

This balances efficiency with flexibility, leveraging each algorithm's strengths. For partial matches (e.g., "phone" in "iPhone"), the provided BinarySearchPartialMatch method extends binary search efficiency to fuzy queries.

Binary search consistently outperforms linear search (e.g., 2–100x faster for large datasets) but when it comes to small datasets or unsorted, Linear serach can be effective to use.

Exercise 7: Financial Forecasting

Exercise 7: Financial Forecasting

Scenario:

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

Steps:

1. **Understand Recursive Algorithms:**
    - Explain the concept of recursion and how it can simplify certain problems.
2. **Setup:**
    - Create a method to calculate the future value using a recursive approach.
3. **Implementation:**
    - Implement a recursive algorithm to predict future values based on past growth rates.
4. **Analysis:**
    - Discuss the time complexity of your recursive algorithm.
    - Explain how to optimize the recursive solution to avoid excessive computation.

SOLUTION :

1. Recursion is a programming technique where a function calls itself to solve a problem. It breaks a big problem into smaller, similar subproblems until it reaches a simple base case that can be solved directly. Recursion can simplify problems like calculating factorials, traversing trees, or computing compound interest, because it allows us to write clear and concise code that closely matches the way the problem is defined.

2 & 3.

Program.cs code:

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace FinancialForecastingTool
{
    public class FinancialForecaster
    {
        private Dictionary<string, double> memoCache = new Dictionary<string, double>();
```

```
        private List<double> historicalData = new List<double>();
        public void SetHistoricalData(double[] data)
        {
            historicalData = data.ToList();
            memoCache.Clear();
        }
        // Simple recursive calculation for compound growth
        public double CalculateFutureValueRecursive(double presentValue, double growthRate,
int periods)
        {
            if (periods == 0)
                return presentValue;
            return CalculateFutureValueRecursive(presentValue * (1 + growthRate), growthRate,
periods - 1);
        }
        // Memoized version for efficiency
        public double CalculateFutureValueMemoized(double presentValue, double growthRate, int
periods)
        {
            string key = $"{presentValue}_{growthRate}_{periods}";
            if (memoCache.ContainsKey(key))
                return memoCache[key];
            double result;
            if (periods == 0)
                result = presentValue;
            else
                result = CalculateFutureValueMemoized(presentValue * (1 + growthRate),
growthRate, periods - 1);
            memoCache[key] = result;
            return result;
        }
        // Recursive average growth rate calculation
        public double CalculateAverageGrowthRateRecursive(double[] data, int index = 1, double
sum = 0, int count = 0)
        {
            if (index >= data.Length)
                return count > 0 ? sum / count : 0;
            double growthRate = (data[index] - data[index - 1]) / data[index - 1];
            return CalculateAverageGrowthRateRecursive(data, index + 1, sum + growthRate,
count + 1);
        }
        // Predict future values given historical data
        public double[] PredictFutureValues(int periodsToForecast)
        {
            if (historicalData.Count < 2)
                throw new InvalidOperationException("At least 2 historical data points are
required");
            double avgGrowthRate =
CalculateAverageGrowthRateRecursive(historicalData.ToArray());
            double lastValue = historicalData.Last();
            double[] predictions = new double[periodsToForecast];
            for (int i = 0; i < periodsToForecast; i++)
            {
                predictions[i] = CalculateFutureValueMemoized(lastValue, avgGrowthRate, i + 1);
            }
            return predictions;
        }
    }
    class Program
    {
        static void Main(string[] args)
```

```
        {
            var forecaster = new FinancialForecaster();
            // Example: Set historical data
            double[] historicalRevenue = { 50000, 55000, 58000, 62000, 68000, 72000, 78000 };
            forecaster.SetHistoricalData(historicalRevenue);
            // Calculate average growth rate
            double avgGrowth =
forecaster.CalculateAverageGrowthRateRecursive(historicalRevenue);
            Console.WriteLine($"Average Growth Rate: {avgGrowth:P2}");
            // Predict next 5 years' revenue
            double[] predictions = forecaster.PredictFutureValues(5);
            Console.WriteLine("Future Revenue Predictions:");
            for (int i = 0; i < predictions.Length; i++)
            {
                Console.WriteLine($"Year {historicalRevenue.Length + i + 1}:
Rs.{predictions[i]:F2}");
            }
            // Simple compound growth calculation
            double futureValue = forecaster.CalculateFutureValueRecursive(10000, 0.08, 10);
            Console.WriteLine($"\nFuture Value (Recursive, 10 years at 8%):
Rs.{futureValue:F2}");
        }
    }
}
```

OUTPUT:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS                    powershell - FinancialForecasting

PS C:\Users\KIIT\OneDrive\Desktop\Cognizant_DeepSkilling_.NET_solutions\1_2_DSA\FinancialForecasting> dotnet run
Average Growth Rate: 7.71%
Future Revenue Predictions:
Year 8: Rs.84011.75
Year 9: Rs.90486.84
Year 10: Rs.97460.99
Year 11: Rs.104972.67
Year 12: Rs.113063.30

Future Value (Recursive, 10 years at 8%): Rs.21589.25
PS C:\Users\KIIT\OneDrive\Desktop\Cognizant_DeepSkilling_.NET_solutions\1_2_DSA\FinancialForecasting>
```

4**. Time Complexity of the Recursive Algorithm:**

The time complexity of the basic recursive algorithm for predicting future values is **O(n)**, where n is the number of periods or steps. This is because each recursive call reduces the problem by one period, and the function is called once for each period until it reaches the base case.

-How to Optimize the Recursive Solution:

To avoid excessive computation and make the recursive algorithm efficient, you can use **memoization**. Memoization means storing the results of previous calculations in a cache (like a dictionary) so that if the same calculation is needed again, we can reuse the stored result instead of recalculating it. This reduces repeated work and improves the performance of the recursive function, especially when the same inputs occur multiple times.