**WEEK1\_DATA STRUCTURE AND ALGORITHM**

**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.

***code***

using System;

namespace EcommerceSearchSystem

{

public class Product : IComparable<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 int CompareTo(Product other)

{

return ProductId.CompareTo(other.ProductId);

}

public override string ToString()

{

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

}

}

public class SearchEngine

{

private Product[] productsLinear;

private Product[] productsBinary;

public SearchEngine()

{

// Sample data for linear search (unsorted)

productsLinear = new Product[]

{

new Product(3, "Laptop", "Electronics"),

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

new Product(2, "Book", "Books")

};

// Sorted data for binary search

productsBinary = new Product[productsLinear.Length];

Array.Copy(productsLinear, productsBinary, productsLinear.Length);

Array.Sort(productsBinary);

}

public Product LinearSearch(int productId)

{

foreach (var product in productsLinear)

{

if (product.ProductId == productId)

{

Console.WriteLine($"Found by Linear Search: {product}");

return product;

}

}

Console.WriteLine($"Linear Search: Product ID {productId} not found.");

return null;

}

public Product BinarySearch(int productId)

{

int left = 0;

int right = productsBinary.Length - 1;

while (left <= right)

{

int mid = left + (right - left) / 2;

int comparison = productsBinary[mid].ProductId.CompareTo(productId);

if (comparison == 0)

{

Console.WriteLine($"Found by Binary Search: {productsBinary[mid]}");

return productsBinary[mid];

}

else if (comparison < 0)

{

left = mid + 1;

}

else

{

right = mid - 1;

}

}

Console.WriteLine($"Binary Search: Product ID {productId} not found.");

return null;

}

}

public class SearchTest

{

public static void Main(string[] args)

{

SearchEngine engine = new SearchEngine();

engine.LinearSearch(1);

engine.LinearSearch(4);

engine.BinarySearch(2);

engine.BinarySearch(4);

Console.ReadLine();

}

}

}

***Code Analysis***

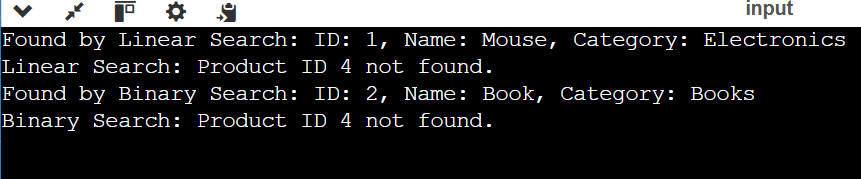
· **Time Complexity Comparison**:

* **Linear Search**: O(n) in all cases (best, average, worst). It checks each element sequentially, making it simple but inefficient for large datasets.
* **Binary Search**: O(log n) in all cases (best, average, worst), requiring a sorted array. It halves the search space with each step, offering better scalability.

· **Suitability for the Platform**:

* **Binary Search** is more suitable for an e-commerce platform with a large product catalog (e.g., thousands of items). The requirement for a sorted array can be maintained by sorting on insertion or using a pre-sorted database index. This ensures faster search times, critical for user experience.
* **Linear Search** is better for small, unsorted datasets or when data is frequently updated, avoiding the overhead of maintaining sorted order. However, for an e-commerce platform, the performance gain of O(log n) outweighs this convenience.
* **Optimization**: Use a balanced binary search tree (e.g., AVL or Red-Black Tree) or a hash table (O(1) average case) for dynamic data, though this requires additional implementation complexity.

***OUTPUT :***



**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.

**CODE**

using System;

namespace FinancialForecastingTool

{

public class ForecastEngine

{

public double CalculateFutureValue(double initialValue, double growthRate, int years)

{

if (years == 0)

return initialValue;

return CalculateFutureValue(initialValue \* (1 + growthRate), growthRate, years - 1);

}

}

public class ForecastTest

{

public static void Main(string[] args)

{

ForecastEngine engine = new ForecastEngine();

double initialValue = 1000.0;

double growthRate = 0.05;

int years = 3;

double futureValue = engine.CalculateFutureValue(initialValue, growthRate, years);

Console.WriteLine($"Future Value after {years} years: {futureValue:C2}");

Console.ReadLine();

}

}

}

***Code Analysis***

· **Time Complexity**: The recursive algorithm has a time complexity of O(n), where n is the number of years. Each recursive call reduces the problem size by 1 (years - 1), and the function makes n recursive calls, each performing O(1) operations (multiplication and addition).

· **Space Complexity**: O(n) due to the call stack depth, as each recursive call adds a frame to the stack until the base case is reached.

· **Optimization**:

* **Memoization**: Store intermediate results in a cache (e.g., array or dictionary) to avoid recalculating values for the same inputs, reducing time complexity to O(1) for repeated calculations. However, this is less relevant for a simple linear recursion like this.
* **Iterative Approach**: Convert the recursion to a loop to eliminate stack space usage, reducing space complexity to O(1). For example:

public double CalculateFutureValueIterative(double initialValue, double growthRate, int years)

{

double value = initialValue;

for (int i = 0; i < years; i++)

{

value \*= (1 + growthRate);

}

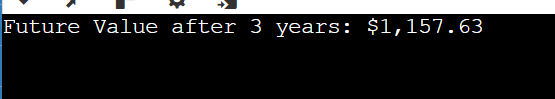
return value;

}

· **Tail Recursion Optimization**: If the compiler supports tail call optimization (not guaranteed in C#), the recursive call could be optimized to use constant stack space. However, C# does not reliably perform this optimization, so the iterative approach is preferred.

· **Precomputation**: For fixed growth rates and multiple forecasts, precompute values for common year ranges to serve as a lookup table.

***OUTPUT :***



**Superset id 6363566**