|  |  |
| --- | --- |
| Cognizant Digital Nurture 4.0: Deep Skilling Hands-on | |
| Name: Rajatesh Paul | Superset ID: 6365356 |
| Week: 01 | Data Structures and Algorithms |

**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.

**Answer:**

**1. Understand Asymptotic Notation:**

Big O notation describes algorithm efficiency by quantifying worst-case time complexity relative to input size (n). For search operations:

* **Best-case**: Minimum operations (e.g., target found immediately).
* **Average-case**: Expected performance over random inputs.
* **Worst-case**: Maximum operations (e.g., target absent or at the end).

**2. Setup:**

**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;

}

}

**3. Implementation:**

**LinearSearch.cs**

public class LinearSearch{

public static Product Search(Product[] products, string name){

foreach (var product in products){

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

return product;

}

return null;

}

}

**BinarySearch.cs**

public class BinarySearch{

public static Product Search(Product[] products, string name){

int left = 0;

int right = products.Length - 1;

while (left <= right){

int mid = (left + right) / 2;

int comparison = string.Compare(products[mid].ProductName, name, 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;

class Program

{

static void Main()

{

Product[] products = new Product[]

{

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

new Product(2, "Shampoo", "Personal Care"),

new Product(3, "Book", "Stationery"),

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

};

// Sort products by name for binary search

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

Console.WriteLine("Linear Search: Looking for 'Phone'");

var result1 = LinearSearch.Search(products, "Phone");

Console.WriteLine(result1 != null ? $"Found: {result1.ProductName}" : "Not Found");

Console.WriteLine("Binary Search: Looking for 'Phone'");

var result2 = BinarySearch.Search(sortedProducts, "Phone");

Console.WriteLine(result2 != null ? $"Found: {result2.ProductName}" : "Not Found");

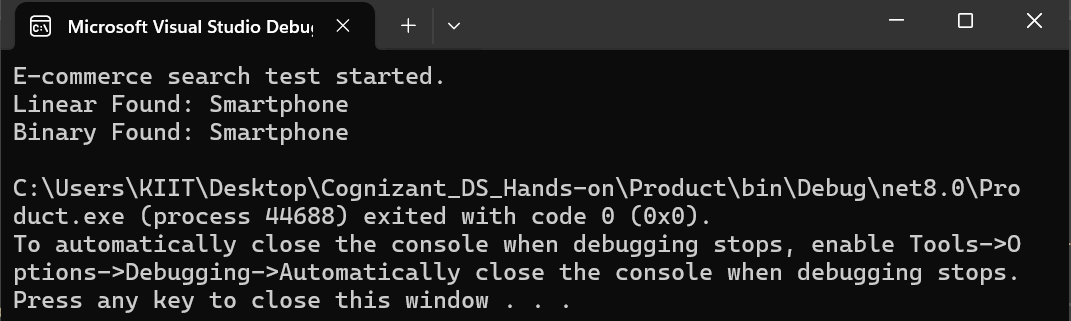
}

}

**4. Analysis-**

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

**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.

**Answer-**

1. **Understand Recursive Algorithms:**

Recursion is a programming technique where a function calls itself to solve a smaller part of a problem until a base case is reached. Why use Recursion? It simplifies problems that have a repetitive or self-similar structure, such as financial forecasts over time.

1. **Setup:**

FV(n)=FV(n−1)×(1+r)

Where:

* FV(n) = Future Value at year n
* r = Annual growth rate (e.g., 0.05 for 5%)
* n = Number of years
* FV(0) = Initial investment (base case)

1. **Implement:**

**FinancialForecast.cs**

using System;

class FinancialForecast

{

// Recursive method to calculate future value

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

{

// Base case: year 0

if (years == 0)

return initialValue;

// Recursive case

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

}

}

**Program.cs**

class Program

{

static void Main(string[] args)

{

double initialValue = 1000; // starting amount

double growthRate = 0.05; // 5% annual growth

int years = 5; // number of years

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

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

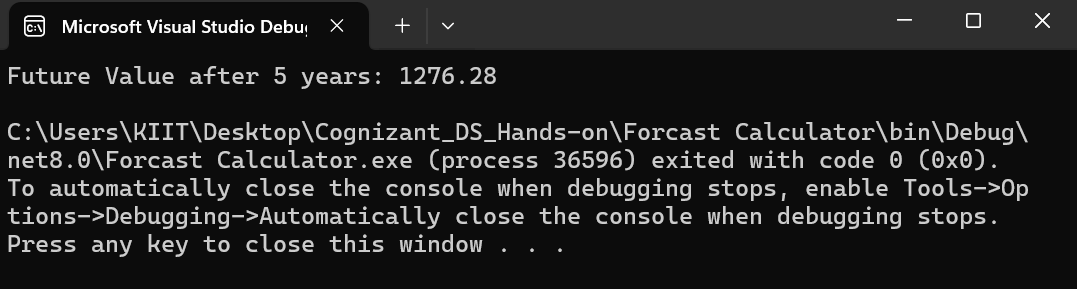
}

}

1. **Analysis:**

* Each recursive call reduces years by 1, so the function runs **n times** (where n = years).
* **Time complexity**: O(n)

**Output-**

****