**Week 1 - Data structures and Algorithms - Hands-on**

DEBANJAN KAURI(Superset ID-6362388)

**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.

To achieve this:

1. Explain Big O notation and the best, average, and worst-case complexities for search.

2. Create a Product class with productId, productName, and category.

3. Implement both linear and binary search algorithms.

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

5. Compare their time complexities and justify the better choice for the platform.

**Explanation:**

Big O notation is a mathematical representation used to describe the **time and space complexity** of an algorithm. It characterizes the **upper bound** of an algorithm's growth rate as the input size increases — essentially telling us how the algorithm performs at scale.

Time Complexity:

- Best Case: The scenario where the algorithm performs the least work.

- Average Case: The expected performance for a random input.

- Worst Case: The scenario where the algorithm performs the most work.

For example:

- Linear Search: Best: O(1), Average: O(n), Worst: O(n)

- Binary Search: Best: O(1), Average: O(log n), Worst: O(log n)

**Code:**

using System;

namespace ECommerceSearchFunction

{

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 $"{ProductId} - {ProductName} ({Category})";

}

}

public class Program

{

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[] sortedProducts, string targetName)

{

int left = 0, right = sortedProducts.Length - 1;

while (left <= right)

{

int mid = (left + right) / 2;

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

if (comparison == 0)

return sortedProducts[mid];

else if (comparison < 0)

left = mid + 1;

else

right = mid - 1;

}

return null;

}

public static void Main()

{

Console.WriteLine("--- E-commerce Search Function ---\n");

Product[] products = new Product[]

{

new Product(101, "Laptop", "Electronics"),

new Product(102, "Shoes", "Fashion"),

new Product(103, "Book", "Education"),

new Product(104, "Smartphone", "Electronics"),

new Product(105, "Watch", "Accessories")

};

Console.Write("Enter product name to search: ");

string? searchName = Console.ReadLine()?.Trim();

var linearResult = LinearSearch(products, searchName ?? "");

Console.WriteLine("\n[Linear Search Result]");

if (linearResult != null)

Console.WriteLine("Found: " + linearResult);

else

Console.WriteLine("Product not found.");

Array.Sort(products, (p1, p2) => string.Compare(p1.ProductName, p2.ProductName, StringComparison.OrdinalIgnoreCase));

var binaryResult = BinarySearch(products, searchName ?? "");

Console.WriteLine("\n[Binary Search Result]");

if (binaryResult != null)

Console.WriteLine("Found: " + binaryResult);

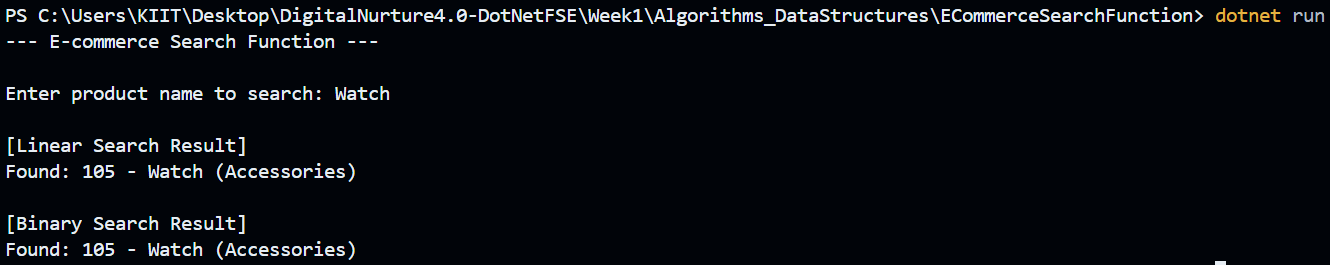
else

Console.WriteLine("Product not found.");

}

}

}

**OUTPUT:**

**Analysis:**

Linear Search:

- Time Complexity: O(n)

- Use when data is small or unsorted..

Binary Search:

- Time Complexity: O(log n)

- Requires sorted input but much faster for large data sets.

Conclusion:

Binary Search is more efficient for sorted product lists, making it more suitable for real-world e-commerce platforms.

**Exercise 7: Financial Forecasting**

Scenario:

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

To achieve this:

1. Explain recursion and how it simplifies problem-solving.

2. Create a method to calculate future value recursively.

3. Implement a recursive algorithm using past values and growth rate.

4. Analyze the time complexity of the recursive approach  
5. Explain how to optimize the solution to reduce redundant computations.

**Explanation:**

Recursion is a technique where a function calls itself to solve smaller subproblems. It simplifies code for repetitive or mathematical problems, especially those based on recurrence patterns like growth or factorial. It reduces manual logic by allowing a function to call itself .

- Base Case: Defines the condition when recursion ends.

- Recursive Case: Breaks the problem down and calls itself.

**Code:**

using System;

using System.Collections.Generic;

namespace FinancialForecasting

{

public class Program

{

public static double CalculateFutureValue(double principal, double rate, int years)

{

if (years == 0)

return principal;

return (1 + rate) \* CalculateFutureValue(principal, rate, years - 1);

}

public static double CalculateFutureValueMemo(double principal, double rate, int years, Dictionary<int, double> memo)

{

if (years == 0)

return principal;

if (memo.ContainsKey(years))

return memo[years];

double value = (1 + rate) \* CalculateFutureValueMemo(principal, rate, years - 1, memo);

memo[years] = value;

return value;

}

public static void Main()

{

Console.WriteLine("--- Financial Forecasting using Recursion ---");

Console.Write("Enter present value (e.g., 1000): ");

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

Console.Write("Enter annual growth rate (e.g., 0.05 for 5%): ");

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

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

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

Console.WriteLine("\n[Recursive Calculation]");

double futureValue = CalculateFutureValue(principal, rate, years);

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

Console.WriteLine("\n[Memoized Recursive Calculation]");

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

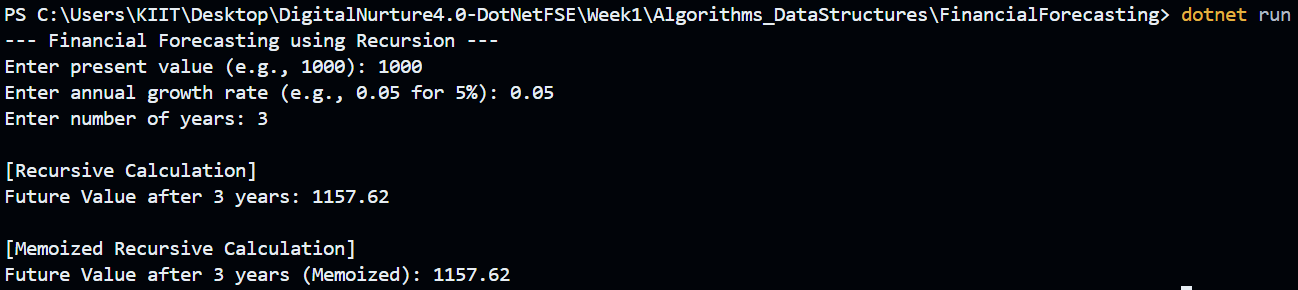
double futureValueMemo = CalculateFutureValueMemo(principal, rate, years, memo);

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

}

}

}

**OUTPUT:**

**Analysis:**

Time Complexity:

* Basic recursion: O(n) time, O(n) space.
* Memoized Recursive: O(n) time, O(n) space (avoids recalculations).

Optimization Strategy:

* Use memoization to store and reuse results of previously calculated years.
* Prevents redundant computation in recursive branches.
* Handles large input sizes more safely and efficiently.

Conclusion:

Recursion is well-suited for financial forecasting based on patterns. Memoization further improves performance for practical use.