**SUBMITTED BY: SUPERSET ID-6361172**

**Week 1\_Algorithms\_Data\_Structures**

**Exercise 2: E-commerce Platform Search Function**

**Steps:**

1. **Understand Asymptotic Notation:**
   1. **Explain Big O notation and how it helps in analyzing algorithms.**

Big O notation is a mathematical way to describe how the performance of an algorithm changes as the size of the input increases. It is used to express the **time complexity** or **space complexity** of an algorithm. Time complexity tells us how the running time grows with the input, while space complexity tells us how the memory usage grows.

Big O notation uses symbols like:

* + **O(1)** - constant time: the operation takes the same time no matter how big the input is.
  + **O(n)** - linear time: the time increases directly with the input size.
  + **O(log n)** - logarithmic time: time increases slowly as input grows.
  + **O(n^2)** - quadratic time: time increases very fast, especially with nested loops.

Big O helps in **analyzing algorithms** by allowing developers to predict how their programs will perform on large data sets without needing to test on every possible case. For example, in a search function, using **linear search** takes O(n) time because it checks each item one by one. But using **binary search** on a sorted list takes O(log n) time, which is much faster for large inputs.

* 1. Describe the best, average, and worst-case scenarios for search operations.

In search operations, we analyze performance using three cases:

* **Best Case**: The item is found immediately.
  + - For linear search: O(1) (first element)
    - For binary search: O(1) (middle element)
* **Average Case**: The item is somewhere in the middle or randomly placed.
  + - Linear search: O(n)
    - Binary search: O(log n)
* **Worst Case**: The item is not found or is the last one.
  + - Linear search: O(n) (check all items)
    - Binary search: O(log n) (divide until empty)

1. **Setup:**
   1. Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.  
      namespace EcommerceSearch

{

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}";

}

}

}

1. **Implementation:**
   1. Implement linear search and binary search algorithms.
   2. Store products in an array for linear search and a sorted array for binary search.

using System;

namespace EcommerceSearch

{

public class SearchEngine

{

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

{

foreach (var product in products)

{

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

{

return product;

}

}

return null;

}

public static Product BinarySearch(Product[] sortedProducts, string targetName)

{

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

while (left <= right)

{

int mid = (left + right) / 2;

int compare = string.Compare(sortedProducts[mid].ProductName, targetName, true);

if (compare == 0)

return sortedProducts[mid];

else if (compare < 0)

left = mid + 1;

else

right = mid - 1;

}

return null;

}

}

}

1. **Analysis:**
2. **Compare the time complexity of linear and binary search algorithms.**

|  |  |  |  |
| --- | --- | --- | --- |
| **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) |

1. **Discuss which algorithm is more suitable for your platform and why.**

For an e-commerce platform, binary search is a better choice if the product list is sorted. It is much faster and works well when there are a lot of products because it cuts the search area in half each time. Its time complexity is O(log n).

But if the product list is not sorted or keeps changing often, then linear search can be used. It is easier to use and does not need sorting, but it is slower, with time complexity O(n).

So, overall, binary search is best when you want fast searching on a large and sorted product list.

**CODE:  
Program.cs**

//Submitted by: Superset ID- 6361172

//Exercise 2: E-commerce Platform Search Function

using System;

using System.Linq;

namespace EcommercePlatformSearch

{

class Program

{

static void Main(string[] args)

{

Product[] products = {

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

new Product(2, "Shirt", "Clothing"),

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

new Product(4, "Shoes", "Footwear"),

new Product(5, "Book", "Stationery")

};

Console.WriteLine("Available Products:");

foreach (var product in products)

{

Console.WriteLine($"- {product.ProductName}");

}

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

string searchName = Console.ReadLine();

// LINEAR SEARCH

Console.WriteLine("\nUsing LINEAR SEARCH:");

var foundLinear = SearchEngine.LinearSearch(products, searchName);

Console.WriteLine(foundLinear != null ? foundLinear.ToString() : "Product not found");

// BINARY SEARCH

Console.WriteLine("\nUsing BINARY SEARCH:");

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

var foundBinary = SearchEngine.BinarySearch(sortedProducts, searchName);

Console.WriteLine(foundBinary != null ? foundBinary.ToString() : "Product not found");

Console.ReadLine();

}

}

}

**SearchEngine.cs**

//Submitted by: Superset ID- 6361172

//Exercise 2: E-commerce Platform Search Function

using System;

namespace EcommercePlatformSearch

{

public class SearchEngine

{

// Linear Search

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

{

foreach (var product in products)

{

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

{

return product;

}

}

return null;

}

// Binary Search

public static Product BinarySearch(Product[] sortedProducts, string targetName)

{

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

while (left <= right)

{

int mid = (left + right) / 2;

int compare = string.Compare(sortedProducts[mid].ProductName, targetName, true);

if (compare == 0)

return sortedProducts[mid];

else if (compare < 0)

left = mid + 1;

else

right = mid - 1;

}

return null;

}

}

}

**Product.cs**//Submitted by: Superset ID- 6361172

//Exercise 2: E-commerce Platform Search Function

namespace EcommercePlatformSearch

{

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()

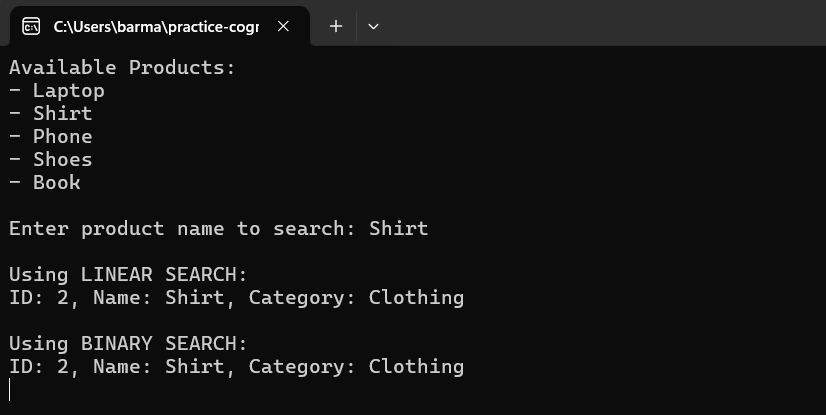
{

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

}

}

}

**OUTPUT:  
**

**Exercise 7: Financial Forecasting**

**Steps:**

1. Explain the concept of recursion and how it can simplify certain problems.  
   Recursion is a programming technique where a method calls itself to solve smaller parts of a problem. It is useful for problems that can be broken down into similar subproblems, such as calculating factorials, Fibonacci numbers, or predicting future values based on patterns.  
   Recursion simplifies the code by using repeated function calls instead of loops. However, it can also lead to performance issues if not handled carefully, especially when there are many repeated calculations.
2. **Setup:**

* Create a method to calculate the future value using a recursive approach.

public static double PredictFutureValue(double currentValue, double growthRate, int years)

{

if (years == 0)

return currentValue;

return PredictFutureValue(currentValue \* (1 + growthRate), growthRate, years - 1);

}

This method keeps calling itself until **years == 0**, calculating compound growth for each year recursively.

1. **Implementation:**

* Implement a recursive algorithm to predict future values based on past growth rates.

public class FinancialForecast  
{  
 public static double PredictFutureValue(double currentValue, double growthRate, int years)  
 {  
 if (years == 0)  
 return currentValue;  
   
 return PredictFutureValue(currentValue \* (1 + growthRate), growthRate, years - 1);  
 }  
}

1. **Analysis:**
   * Discuss the time complexity of your recursive algorithm.

The time complexity of the recursive future value prediction algorithm is O(n), where n is the number of years. This is because the method calls itself once per year until it reaches year 0.

* + Explain how to optimize the recursive solution to avoid excessive computation.  
    **1. Memoization**  
    Store already computed results to avoid repeating the same calculation. Useful for problems with overlapping subproblems (like Fibonacci).

**2. Use Iteration Instead**  
Convert recursion to a loop (e.g., for or while) to reduce function call overhead and avoid stack overflow.

**3. Limit Recursion Depth**  
Avoid deep recursion when n is large. Use iterative methods or divide the problem into smaller parts.

**CODE:**

**FinancialForecast.cs**//Submitted by: Superset ID- 6361172

//Exercise 7: Financial Forecasting

using System;

namespace FinancialForecasting

{

public class FinancialForecast

{

public static double PredictFutureValue(double currentValue, double growthRate, int years)

{

if (years == 0)

return currentValue;

return PredictFutureValue(currentValue \* (1 + growthRate), growthRate, years - 1);

}

}

}

**Program.cs**//Submitted by: Superset ID- 6361172

//Exercise 7: Financial Forecasting

using System;

using System.Globalization;

namespace FinancialForecasting

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("----- Financial Forecasting Tool -----");

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

double currentValue = Convert.ToDouble(Console.ReadLine(), CultureInfo.InvariantCulture);

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

double growthRate = Convert.ToDouble(Console.ReadLine(), CultureInfo.InvariantCulture);

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

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

double futureValue = FinancialForecast.PredictFutureValue(currentValue, growthRate, years);

Console.WriteLine($"\nPredicted future value after {years} years: {futureValue:C2}");

Console.ReadLine();

}

}

}

**OUTPUT:**

