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

### **Big O Notation:**

**Big O notation describes how an algorithm performs **as input size grows**. It focuses on the **upper bound (worst-case)** performance.**

**BEST,AVERAGE,WORST CASE :**

| **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)** |  |

**CODE:-**

**Program.cs:-**

**using System;**

**using System.Linq;**

**namespace ECommerceSearch**

**{**

**class Program**

**{**

**static void Main(string[] args)**

**{**

**Product[] products = new Product[]**

**{**

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

**new Product(2, "T-Shirt", "Apparel"),**

**new Product(3, "Smartphone", "Electronics"),**

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

**new Product(5, "Watch", "Accessories"),**

**};**

**// Linear Search (unsorted)**

**Console.WriteLine("Linear Search for 'Watch':");**

**var linearResult = SearchEngine.LinearSearch(products, "Watch");**

**Console.WriteLine(linearResult != null ? linearResult.ToString() : "Not found");**

**// Binary Search (sorted)**

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

**Console.WriteLine("\nBinary Search for 'Watch':");**

**var binaryResult = SearchEngine.BinarySearch(sortedProducts, "Watch");**

**Console.WriteLine(binaryResult != null ? binaryResult.ToString() : "Not found");**

**Console.ReadLine();**

**}**

**}**

**}**

**CODE:-**

**SearchEngine.cs:-**

**using System;**

**namespace ECommerceSearch**

**{**

**public class SearchEngine**

**{**

**public static Product? LinearSearch(Product[] products, string name)**

**{**

**foreach (var product in products)**

**{**

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

**{**

**return product;**

**}**

**}**

**return null;**

**}**

**public static Product? BinarySearch(Product[] products, string name)**

**{**

**int left = 0;**

**int right = products.Length - 1;**

**while (left <= right)**

**{**

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

**int compare = string.Compare(products[mid].ProductName, name, true);**

**if (compare == 0)**

**return products[mid];**

**else if (compare < 0)**

**left = mid + 1;**

**else**

**right = mid - 1;**

**}**

**return null;**

**}**

**}**

**}**

**CODE:-**

**Product.cs:-**

**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()**

**{**

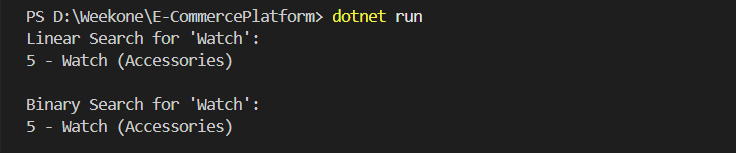
**return $"{ProductId} - {ProductName} ({Category})";**

**}**

**}**

**}**

**OUTPUT SCREENSHOT:-**



## Analysis:

## Time Complexity:

| **Search Method** | **Time Complexity** | **Sorted Data Required?** |
| --- | --- | --- |
| Linear Search | O(n) | No |
| Binary Search | O(log n) | Yes |

**WHICH ALGORITHM IS BETTER :**

* **Binary Search is faster, but only works on sorted data.**
* **For small datasets or unsorted arrays, linear search is easier.**
* **For large datasets, binary search is much faster and preferred.**

.

**Exercise 7: Financial Forecasting:-**

**Recursion is when a method calls **itself** to solve smaller instances of the same problem.**

**It simplifies problems like:**

* **Tree traversal**
* **Fibonacci series**
* **Forecasting based on repeated growth**

**Let’s assume:**

* **We have a base amount (e.g. ₹10,000)**
* **An annual growth rate (e.g. 10%)**
* **We want to forecast the value after N years**

**CODE:-**

**Program.cs:-**

**using System;**

**namespace FinancialTool**

**{**

**class Program**

**{**

**static void Main(string[] args)**

**{**

**double initialAmount = 10000; // ₹10,000**

**double annualGrowthRate = 0.10; // 10%**

**int years = 5;**

**double result = FinancialForecast.ForecastValue(initialAmount, annualGrowthRate, years);**

**Console.WriteLine($"Forecasted value after {years} years: Rs {result:F2}");**

**}**

**}**

**}**

**CODE:-**

**FinancialForecast.cs:-**

**namespace FinancialTool**

**{**

**public class FinancialForecast**

**{**

**public static double ForecastValue(double initialAmount, double growthRate, int years)**

**{**

**if (years == 0)**

**return initialAmount;**

**return ForecastValue(initialAmount, growthRate, years - 1) \* (1 + growthRate);**

**}**

**}**

**}**

**OUTPUT SCREENSHOT:-**



## **Time Complexity Analysis:-**

****Recursive Calls**: 1 call per year -> O(n) time**

****Space Complexity**: O(n) due to call stack**

## O**ptimization: Use Memoization**

**If the same years are calculated repeatedly, **memoization** avoids recomputing.**

**CODE FOR THE SAME:-**

**public static class OptimizedForecast**

**{**

**private static Dictionary<int, double> cache = new();**

**public static double ForecastValueMemo(double initialAmount, double growthRate, int years)**

**{**

**if (years == 0)**

**return initialAmount;**

**if (cache.ContainsKey(years))**

**return cache[years];**

**double result = ForecastValueMemo(initialAmount, growthRate, years - 1) \* (1 + growthRate);**

**cache[years] = result;**

**return result;**

**}**

**}**