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

**Big O notation:**

**Big O notation** is a mathematical way to describe the **upper bound** of an algorithm's running time or space requirement as a function of input size **n**.

It expresses the **worst-case scenario** of an algorithm's growth rate — **how fast the time or memory usage increases** as the input grows larger.

Examples with Search Algorithms:

**Linear Search(Scans each item one by one.):**

**Best Case**: Target is first element ⇒ O(1)

**Worst Case**: Target is last or not found ⇒ O(n)

**Average Case**: Element is in the middle ⇒ O(n)

**Binary Search(Requires a sorted array and divides the search space in half at each step**.**):**

**Best Case**: Middle element is the target ⇒ O(1)

**Worst & Average Case**: Each step halves the array ⇒ O(log n)

**Setup and Implementation:**

using System;

using System.Linq;

using System.Diagnostics;

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}";

}

}

public class Program

{

public static Product LinearSearch(Product[] products, int targetId)

{

foreach (var product in products)

{

if (product.ProductId == targetId)

return product;

}

return null;

}

public static Product BinarySearch(Product[] products, int targetId)

{

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

while (left <= right)

{

int mid = (left + right) / 2;

if (products[mid].ProductId == targetId)

return products[mid];

else if (products[mid].ProductId < targetId)

left = mid + 1;

else

right = mid - 1;

}

return null;

}

public static void Main()

{

int targetId = 103;

Product[] unsortedProducts = {

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

new Product(105, "Shoes", "Footwear"),

new Product(103, "Watch", "Accessories"),

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

new Product(102, "Shirt", "Clothing")

};

Product[] sortedProducts = unsortedProducts.OrderBy(p => p.ProductId).ToArray();

Stopwatch stopwatch = new Stopwatch();

stopwatch.Start();

var linearResult = LinearSearch(unsortedProducts, targetId);

stopwatch.Stop();

long linearTime = stopwatch.ElapsedTicks;

Console.WriteLine("=== LINEAR SEARCH (Unsorted Array) ===");

Console.WriteLine(linearResult != null ? $"Found: {linearResult}" : "Product not found.");

Console.WriteLine($"Time Taken: {linearTime} ticks");

stopwatch.Restart();

var binaryResult = BinarySearch(sortedProducts, targetId);

stopwatch.Stop();

long binaryTime = stopwatch.ElapsedTicks;

Console.WriteLine("\n=== BINARY SEARCH (Sorted Array) ===");

Console.WriteLine(binaryResult != null ? $"Found: {binaryResult}" : "Product not found.");

Console.WriteLine($"Time Taken: {binaryTime} ticks");

Console.WriteLine("\n=== TIME COMPLEXITY SUMMARY ===");

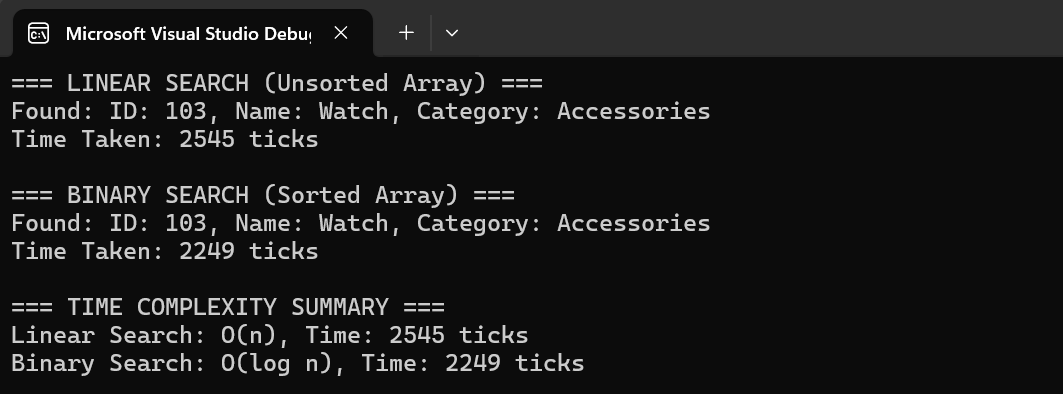
Console.WriteLine("Linear Search: O(n), Time: {0} ticks", linearTime);

Console.WriteLine("Binary Search: O(log n), Time: {0} ticks", binaryTime);

}

}

**Output:**

****

**Analysis:**

### ****Time Complexity Comparison:****

**Linear Search** has a time complexity of **O(n)**, which means it checks each item one by one until it finds a match or reaches the end of the array. In the best case, the item is found at the first position (O(1)), but in the worst case, it may be the last item or not present at all (O(n)). This approach becomes slower as the number of products increases.

**Binary Search**, on the other hand, has a time complexity of **O(log n)**. It works by repeatedly dividing the search space in half and checking the middle element. This results in significantly faster performance, especially as the product list grows. However, binary search only works when the data is sorted, which adds an initial cost of sorting if not already done.

### ****Which Algorithm Is More Suitable and Why?****

For an e-commerce platform, **binary search is generally more suitable**, especially when the product catalog is large. Most real-world e-commerce systems have thousands of products, and fast search is essential for a good user experience. Since product data (like ProductId or ProductName) can be sorted once and reused for many search operations, the one-time sorting cost is justified by the improved performance.

However, if the platform deals with a small number of products or if products are frequently added, removed, or updated in real time, **linear search may be acceptable** or even preferred for simplicity—especially when data isn't sorted.

**Exercise 7: Financial Forecasting**

**Concept of Recursion:**

**Recursion** is a programming technique where a function **calls itself** to solve smaller subproblems of the original problem. This approach continues until it reaches a **base case**, which provides a direct answer without further recursion.

The core idea is to break down a complex problem into simpler versions of itself. This is especially useful in situations where the same pattern repeats over time — such as in **financial forecasting**.

### ****Recursion in Financial Forecasting:****

In financial forecasting, you're often trying to **predict the future value** of an investment or asset based on its past performance and a consistent growth rate. This is a **naturally recursive process**, because:

* The value in **year** n depends on the value in **year** n-1.
* The pattern continues until reaching **year 0**, which is the starting or base amount.

This structure fits recursion perfectly.

**Setup and Implementation:**

using System;

using System.Collections.Generic;

class FinancialForecast

{

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

{

if (years == 0)

return baseValue;

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

}

public static double PredictFutureValueMemo(int years, double baseValue, double growthRate, Dictionary<int, double> memo)

{

if (years == 0)

return baseValue;

if (memo.ContainsKey(years))

return memo[years];

memo[years] = PredictFutureValueMemo(years - 1, baseValue, growthRate, memo) \* (1 + growthRate);

return memo[years];

}

static void Main()

{

int futureYears = 5;

double baseValue = 10000.0;

double growthRate = 0.05;

Console.WriteLine("Recursive Forecast (No Memoization):");

Console.WriteLine($"Year {futureYears}: Rs.{PredictFutureValue(futureYears, baseValue, growthRate):F2}");

Console.WriteLine("\nOptimized Recursive Forecast (With Memoization):");

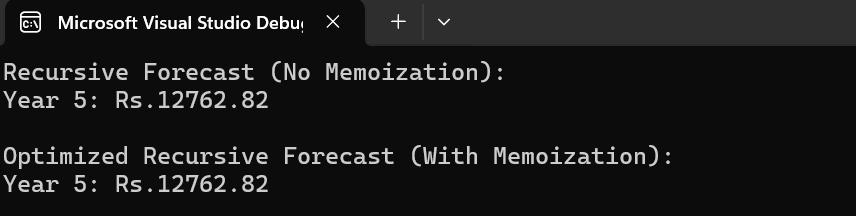
var memo = new Dictionary<int, double>();

Console.WriteLine($"Year {futureYears}: Rs.{PredictFutureValueMemo(futureYears, baseValue, growthRate, memo):F2}");

}

}

**Output:**

****

**Time Complexity:**

Let n be the number of years.

The function calls itself **once** for each year:

* *PredictFutureValue(n*) calls *PredictFutureValue(n - 1)*
* which calls *PredictFutureValue(n - 2)*, and so on
* until *PredictFutureValue(0)*

So, it performs exactly **n** recursive calls.

Therefore,  
**Time Complexity = O(n)**  
**Space Complexity (due to call stack) = O(n)**

**Optimization of solution:**

**The recursive solution can be optimized by using dynamic programming implementation(memoization).**

**Memoization stores results of already-computed years in a dictionary or array.  
So when the function is asked to compute the value for a year it has already seen, it simply returns the stored result instead of recalculating.The optimized solution has also been implemented in the code above.**

****Optimized solution:****

public static double PredictFutureValueMemo(int years, double baseValue, double growthRate, Dictionary<int, double> memo)

{

if (years == 0)

return baseValue;

if (memo.ContainsKey(years))

return memo[years];

memo[years] = PredictFutureValueMemo(years - 1, baseValue, growthRate, memo) \* (1 + growthRate);

return memo[years];

}