**Week 1:**

**Data Structures and Algorithm:**

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

**Q1.Explain Big O notation and how it helps in analyzing algorithms.**

Big O notation is a mathematical way to describe how the running time or space requirements of an algorithm grow as the input size increases. It focuses on the dominant factor that affects performance for large inputs, ignoring constants and less significant terms.

How it helps in analyzing algorithms:

It helps compare the efficiency of different algorithms.Predicts how an algorithm will perform as the dataset grows, which is important for scalability and resource management.Guides developers to choose or design algorithms that remain efficient even with large data.

**Q2.Describe the best, average, and worst-case scenarios for search operations.**

* Best Case:  
  The fastest possible outcome, such as finding the target in the first position. For linear search, this is O(1) time.
* Average Case:  
  The typical scenario over many runs, assuming all input positions are equally likely. For linear search, on average, half the elements are checked, so the time is about O(n/2), which simplifies to O(n).
* Worst Case:  
  The slowest scenario, such as when the target is at the last position or not present at all. For linear search, this means checking every element, so the time is O(n).

**Code:**

[**Product.cs**](http://product.cs)

**using System;**

**namespace ECommerceSearch**

**{**

**public class Product : IComparable<Product>**

**{**

**public int ProductId { get; set; }**

**public string ProductName { get; set; }**

**public string Category { get; set; }**

**public decimal Price { get; set; }**

**public Product(int id, string name, string category, decimal price)**

**{**

**ProductId = id;**

**ProductName = name;**

**Category = category;**

**Price = price;**

**}**

**public int CompareTo(Product other)**

**{**

**return ProductId.CompareTo(other.ProductId);**

**}**

**public override string ToString()**

**{**

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

**}**

**}**

**}**

[**SearchAlgorithms.cs**](http://searchalgorithms.cs)

**using System;**

**using System.Diagnostics;**

**namespace ECommerceSearch**

**{**

**public static class SearchAlgorithms**

**{**

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

**{**

**for (int i = 0; i < products.Length; i++)**

**{**

**if (products[i].ProductId == targetId)**

**return i;**

**}**

**return -1;**

**}**

**public static int BinarySearch(Product[] sortedProducts, int targetId)**

**{**

**int left = 0;**

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

**while (left <= right)**

**{**

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

**if (sortedProducts[mid].ProductId == targetId)**

**return mid;**

**if (sortedProducts[mid].ProductId < targetId)**

**left = mid + 1;**

**else**

**right = mid - 1;**

**}**

**return -1;**

**}**

**public static int LinearSearchByName(Product[] products, string productName)**

**{**

**for (int i = 0; i < products.Length; i++)**

**{**

**if (products[i].ProductName.Equals(productName, StringComparison.OrdinalIgnoreCase))**

**return i;**

**}**

**return -1;**

**}**

**}**

**}**

[**Program.cs**](http://program.cs)

**using System;**

**using System.Diagnostics;**

**namespace ECommerceSearch**

**{**

**class Program**

**{**

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

**{**

**Product[] products = {**

**new Product(5, "Laptop", "Electronics", 999.99m),**

**new Product(2, "Phone", "Electronics", 699.99m),**

**new Product(8, "Book", "Education", 29.99m),**

**new Product(1, "Headphones", "Electronics", 199.99m),**

**new Product(6, "Shirt", "Clothing", 39.99m),**

**new Product(3, "Shoes", "Clothing", 89.99m),**

**new Product(9, "Watch", "Accessories", 299.99m),**

**new Product(4, "Tablet", "Electronics", 499.99m)**

**};**

**Product[] sortedProducts = new Product[products.Length];**

**Array.Copy(products, sortedProducts, products.Length);**

**Array.Sort(sortedProducts);**

**Console.WriteLine("=== E-Commerce Search Performance Analysis ===\n");**

**Console.WriteLine("Available Products:");**

**for (int i = 0; i < products.Length; i++)**

**{**

**Console.WriteLine($"{i + 1}. {products[i]}");**

**}**

**TestSearchPerformance(products, sortedProducts);**

**Console.WriteLine("\n--- Large Dataset Performance Comparison ---");**

**CompareLargeDatasetPerformance();**

**}**

**static void TestSearchPerformance(Product[] products, Product[] sortedProducts)**

**{**

**int targetId = 6;**

**Console.WriteLine($"\n--- Searching for Product ID: {targetId} ---");**

**Stopwatch sw = Stopwatch.StartNew();**

**int linearResult = SearchAlgorithms.LinearSearch(products, targetId);**

**sw.Stop();**

**Console.WriteLine($"Linear Search: {(linearResult != -1 ? $"Found at index {linearResult}" : "Not found")}");**

**Console.WriteLine($"Time: {sw.ElapsedTicks} ticks");**

**sw.Restart();**

**int binaryResult = SearchAlgorithms.BinarySearch(sortedProducts, targetId);**

**sw.Stop();**

**Console.WriteLine($"Binary Search: {(binaryResult != -1 ? $"Found at index {binaryResult}" : "Not found")}");**

**Console.WriteLine($"Time: {sw.ElapsedTicks} ticks");**

**}**

**static void CompareLargeDatasetPerformance()**

**{**

**const int dataSize = 100000;**

**Product[] largeDataset = new Product[dataSize];**

**Product[] sortedLargeDataset = new Product[dataSize];**

**for (int i = 0; i < dataSize; i++)**

**{**

**largeDataset[i] = new Product(i + 1, $"Product{i + 1}", "Category", 99.99m);**

**sortedLargeDataset[i] = largeDataset[i];**

**}**

**int targetId = 75000;**

**Stopwatch sw = Stopwatch.StartNew();**

**SearchAlgorithms.LinearSearch(largeDataset, targetId);**

**sw.Stop();**

**long linearTime = sw.ElapsedTicks;**

**sw.Restart();**

**SearchAlgorithms.BinarySearch(sortedLargeDataset, targetId);**

**sw.Stop();**

**long binaryTime = sw.ElapsedTicks;**

**Console.WriteLine($"Dataset Size: {dataSize:N0} products");**

**Console.WriteLine($"Linear Search Time: {linearTime:N0} ticks");**

**Console.WriteLine($"Binary Search Time: {binaryTime:N0} ticks");**

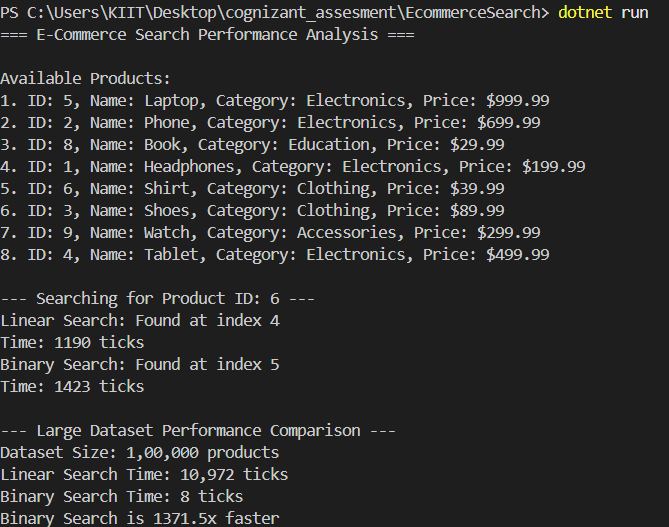
**Console.WriteLine($"Binary Search is {(double)linearTime / binaryTime:F1}x faster");**

**}**

**}**

**}**

**Output:**

****

**Q3.Compare the time complexity of linear and binary search algorithms**

## **Linear Search:**

* Time Complexity: O(n)
* Best Case: O(1) - target found at first position
* Average Case: O(n/2) ≈ O(n) - target found in middle
* Worst Case: O(n) - target at end or not found
* Space Complexity: O(1)

## **Binary Search**

* Time Complexity: O(log n)
* Best Case: O(1) - target found at middle position
* Average Case: O(log n)
* Worst Case: O(log n) - maximum log₂(n) comparisons
* Space Complexity: O(1)

**Q4.Discuss which algorithm is more suitable for your platform and why.**

Binary search is more suitable for an e-commerce platform than linear search because it is much faster and more efficient for large, sorted product lists. Binary search has a time complexity of O(log n), meaning it can find a product in just a few steps even among thousands or millions of items, while linear search is O(n) and becomes slow as the product list grows.

**2. Exercise 7: Financial Forecasting :**

**Q1.Explain the concept of recursion and how it can simplify certain problems.**

Recursion is a programming technique where a function calls itself to solve a problem by breaking it down into smaller, simpler subproblems. Each recursive function has two main parts:

Base case: The simplest situation, which ends the recursion (prevents infinite calls).

Recursive case: The function calls itself with a modified input, moving closer to the base case each time.

Recursion allows you to write elegant solutions for complex problems by handling one small piece at a time. For example, instead of writing loops to process nested structures (like directories or trees), a recursive function can visit each level naturally. Classic examples include calculating factorials, traversing file systems, and searching tree-like data.

**Q2.Create a method to calculate the future value using a recursive approach.**

static double ForecastRecursive(double initialValue, double growthRate, int periods)

{

if (periods == 0)

return initialValue;

return ForecastRecursive(initialValue, growthRate, periods - 1) \* (1 + growthRate); // Recursive case

}

**Code:**

[**Program.cs**](http://program.cs)

**using System;**

**class Program**

**{**

**static double ForecastRecursive(double initialValue, double growthRate, int periods)**

**{**

**if (periods == 0)**

**return initialValue;**

**return ForecastRecursive(initialValue, growthRate, periods - 1) \* (1 + growthRate); // Recursive case**

**}**

**static void Main()**

**{**

**double initialValue = 1000.0;**

**double growthRate = 0.08;**

**int periods = 7;**

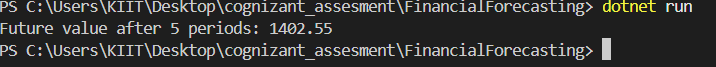
**double forecast = ForecastRecursive(initialValue, growthRate, periods);**

**Console.WriteLine($"Future value after {periods} periods: {forecast:F2}");**

**}**

**}**

**Output:**

****

**Q3.Discuss the time complexity of your recursive algorithm.**

The recursive algorithm has O(n) time complexity, where n is the number of periods, because it makes one call per period.Space complexity is also O(n).

**Q4. Explain how to optimize the recursive solution to avoid excessive computation.**

Optimized approach: we will use an iterative method because it will prevents stack overflow and is suitable for very large period counts as shown below

public static double ForecastIterative(double initialValue, double growthRate, int periods)

{

double value = initialValue;

for (int i = 0; i < periods; i++)

{

value \*= (1 + growthRate);

}

return value;

}