**Data Structures and Algorithms:**

**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.

**Steps:**

1. **Understand Asymptotic Notation:**
   * Explain Big O notation and how it helps in analyzing algorithms.
   * Describe the best, average, and worst-case scenarios for search operations.
2. **Setup:**
   * Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
   * Implement linear search and binary search algorithms.
   * Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
   * Compare the time complexity of linear and binary search algorithms.
   * Discuss which algorithm is more suitable for your platform and why.

**Big O Notation**

Big O notation is used to describe the time and space complexity of an algorithm in terms of how it scales with input size. It represents the worst-case performance, ignoring constants and lower-order terms.

Importance:

* Helps in analyzing algorithm efficiency.
* Predicts performance for large inputs.
* Enables comparison between different algorithms.

**Complexity analysis for search operations**

**1. Best-Case Scenario**

Definition: The condition where the target element is found immediately.

Example: In linear search, the target is the first element.

Time Complexity:

* Linear Search: O(1)
* Binary Search: O(1) (target is at the middle index)

**2. Average-Case Scenario**

Definition: The expected number of operations assuming the target is in a random position.

Example: In linear search, the target is somewhere in the middle.

Time Complexity:

* Linear Search: O(n/2) → simplifies to O(n)
* Binary Search: O(log n)

**3. Worst-Case Scenario**

Definition: The condition where the target is not found or is at the last position.

Example:

* Linear search checks all elements.
* Binary search keeps dividing until one element is left.

Time Complexity:

* Linear Search: O(n)
* Binary Search: O(log n)

**Implementation:**

**Code:**

using System;

using System.Linq;

class Product

{

    public int ProductID;

    public string ProductName;

    public string Category;

    public Product(int productID, string productName, string category)

    {

        this.ProductID = productID;

        this.ProductName = productName;

        this.Category = category;

    }

    public override string ToString()

    {

        return $"[{ProductID}, {ProductName}, {Category}]";

    }

}

class Search

{

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

    {

        foreach (Product p in products)

        {

            if (p.ProductID == targetId)

            {

                return p;

            }

        }

        return null;

    }

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

    {

        int left = 0;

        int right = products.Length - 1;

        while (left <= right)

        {

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

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

            {

                return products[mid];

            }

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

            {

                left = mid + 1;

            }

            else

            {

                right = mid - 1;

            }

        }

        return null;

    }

}

class MainClass

{

    public static void Main(string[] args)

    {

        Product[] products = {

            new Product(4, "Apple", "Fruits"),

            new Product(1, "Onion", "Vegetables"),

            new Product(2, "Atta", "Grocery"),

            new Product(3, "Orange", "Fruits")

        };

        Product linear = Search.LinearSearch(products, 2);

        Console.WriteLine("Linear search: " + linear);

        Array.Sort(products, (a, b) => a.ProductID.CompareTo(b.ProductID));

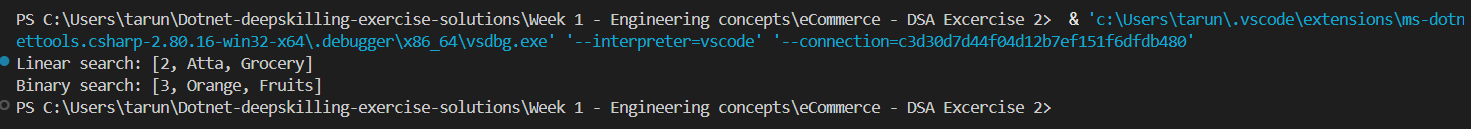
        Product binary = Search.BinarySearch(products, 3);

        Console.WriteLine("Binary search: " + binary);

    }

}

**Output:**



**Time complexity:**

* Linear search : O(n)
* Binary search : O(logn)

Binary Search is more suitable for an e-commerce platform because:

* Binary search performs significantly faster on large, sorted datasets.
* As the number of products grows, binary search remains efficient (logarithmic time).

**Exercise 7: Financial Forecasting**

**Scenario:**

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

**Steps:**

1. **Understand Recursive Algorithms:**
   * Explain the concept of recursion and how it can simplify certain problems.
2. **Setup:**
   * Create a method to calculate the future value using a recursive approach.
3. **Implementation:**
   * Implement a recursive algorithm to predict future values based on past growth rates.
4. **Analysis:**
   * Discuss the time complexity of your recursive algorithm.
   * Explain how to optimize the recursive solution to avoid excessive computation.

**Recursion**

Recursion is a programming technique where a function calls itself directly or indirectly to solve a problem. Each recursive call works on a smaller subproblem, and the solution is built upon the results of these subproblems.

Structure of a Recursive Function:

* Base Case – Stops the recursion when a specific condition is met.
* Recursive Case – The function calls itself with a smaller input

How Recursion Simplifies Problems:

* Breaks complex problems into smaller, manageable subproblems.
* Offers clean and elegant solutions (especially for divide-and-conquer problems).
* Reduces the need for explicit loops and stack handling in certain cases.

**Implementation:**

**Code:**

using System;

public class FinancialForecast

{

    public static double CalculateFutureValue(double presentValue, double growthRate, int years)

    {

        if (years == 0)

        {

            return presentValue;

        }

        return (1 + growthRate) \* CalculateFutureValue(presentValue, growthRate, years - 1);

    }

    public static void Main(string[] args)

    {

        double presentValue = 10000;

        double growthRate = 0.08;

        int years = 5;

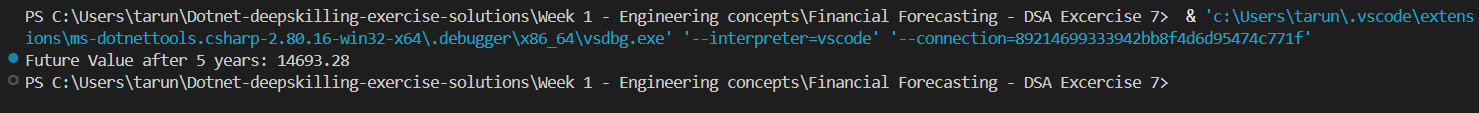
        double futureValue = CalculateFutureValue(presentValue, growthRate, years);

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

    }

}

**Output:**



**Analysis:**

**Time Complexity:**

The recursive function calls itself once for each year until years == 0. This creates a linear chain of recursive calls.

* Total number of recursive calls: years + 1
* Each call performs a constant-time calculation: (1 + growthRate) \* result
* Time Complexity: O(n) where n is the number of years.

**Optimization :**

* Recursive functions can lead to stack overflow for large n, and function calls add overhead.
* Solution: Use Iteration (Tail Recursion or Loop)

We can replace recursion with iteration to avoid excessive memory usage. This approach avoids recursion and is more efficient for large input size