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

**Big O Notation**

Big O notation is used to analyze the efficiency of an algorithm. It describes the worst-case performance of an algorithm as the size of the input data.

- Best Case: The minimum number of operations. For a search, this is finding the item on the first attempt.

- Average Case: The expected performance for a typical input.

- Worst Case: The maximum number of operations. For a search, this means checking every item, or the item is not present at all.

using System;

using System.Diagnostics;

public class Product : IComparable<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 int CompareTo(Product other)

    {

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

    }

}

public class ProductSearch

{

    //Linear search

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

    {

        foreach (var product in products)

        {

            if (product.ProductId == targetId)

            {

                return product;

            }

        }

        return null;

    }

    //Binary search

    public static Product 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 sortedProducts[mid];

            }

            if (sortedProducts[mid].ProductId < targetId)

            {

                left = mid + 1;

            }

            else

            {

                right = mid - 1;

            }

        }

        return null;

    }

}

public class SearchTest

{

    public static void Main(string[] args)

    {

        Product[] products = {

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

            new Product(157, "Headphones", "Electronics"),

            new Product(121, "Keyboard", "Electronics"),

            new Product(199, "Book", "Books"),

            new Product(135, "T-Shirt", "Clothing"),

            new Product(105, "Mouse", "Electronics"),

            new Product(142, "Coffee Mug", "Kitchenware")

        };

        Console.WriteLine("Linear Search");

        Product found1 = ProductSearch.LinearSearch(products, 135);

        Console.WriteLine($"Found product: {(found1 != null ? found1.ProductName : "Not Found")}");

        Array.Sort(products);

        Console.WriteLine("\nBinary Search");

        Product found2 = ProductSearch.BinarySearch(products, 135);

        Console.WriteLine($"Found product: {(found2 != null ? found2.ProductName : "Not Found")}");

    }

}

**Time Complexity Comparison**

**Linear Search:**

Time Complexity: **O(n)**

**Binary Search:**

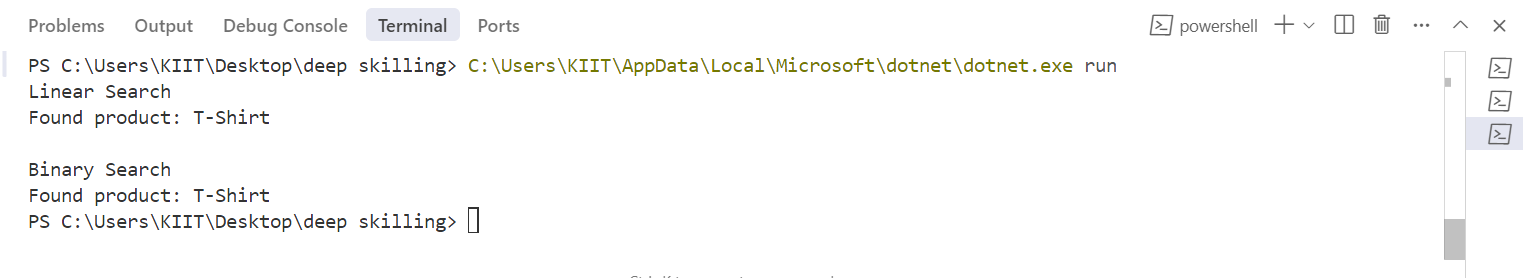
Time Complexity: **O(log n)**

**Suitability for E-commerce Platform**

For an e-commerce platform, Binary Search is far more suitable.

As an e-commerce site can have millions of products. A linear O(n) search would be unacceptably slow and lead to a poor user experience. A binary O(log n) search will return results almost instantly, even for massive datasets.

**OUTPUT**



**Exercise 7: Financial Forecasting**

Concept of Recursion

Recursion is a programming technique where a method calls itself to solve a problem. A recursive function breaks a problem down into smaller, identical sub-problems until it reaches a simple, solvable case called the "base case."

A recursive method must have two parts:

1. Base Case: A condition that stops the recursion and returns a value without

   making another recursive call. Without a base case, the recursion would continue indefinitely, leading to a "stack overflow" error.

2. Recursive Step: The part of the method that calls itself, but with a modified

   input that moves it closer to the base case.

using System;

using System.Collections.Generic;

public class FinancialForecasting

{

    public static double CalculateFutureValueRecursive(double currentValue, double growthRate, int periods)

    {

        if (periods == 0)

        {

            return currentValue;

        }

        double nextValue = currentValue \* (1 + growthRate);

        return CalculateFutureValueRecursive(nextValue, growthRate, periods - 1);

    }

    //Optimized version

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

    {

        double futureValue = initialValue;

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

        {

            futureValue \*= (1 + growthRate);

        }

        return futureValue;

    }

}

public class ForecastTest

{

    public static void Main(string[] args)

    {

        double initialInvestment = 1000.00;

        double annualGrowthRate = 0.05;

        int yearsToForecast = 10;

        Console.WriteLine($"Financial Forecast");

        Console.WriteLine($"Initial Investment: ${initialInvestment:N2}");

        Console.WriteLine($"Annual Growth Rate: {annualGrowthRate:P1}");

        Console.WriteLine($"Period: {yearsToForecast} years\n");

        //RecursiveCalculation

        Console.WriteLine("Using Recursive Method");

        double futureValueRecursive = FinancialForecasting.CalculateFutureValueRecursive(initialInvestment, annualGrowthRate, yearsToForecast);

        Console.WriteLine($"Predicted Future Value: ${futureValueRecursive:N2}");

        //OptimizedIterativeCalculation

        Console.WriteLine("\nUsing Optimized Iterative Method");

        double futureValueIterative = FinancialForecasting.CalculateFutureValueIterative(initialInvestment, annualGrowthRate, yearsToForecast);

        Console.WriteLine($"Predicted Future Value: ${futureValueIterative:N2}");

    }

}

**Time Complexity of the Recursive Algorithm**

The time complexity of the `CalculateFutureValueRecursive` method is **O(n)**, where 'n' is the number of periods.

For each period from 'n' down to 1, the function makes one recursive call. Each call performs a constant amount of work. Therefore, the total work is directly proportional to the number of periods.

While the recursive solution is clear and maps well to the problem's definition, it has drawbacks for large 'n'. Each function call adds a new frame to the call stack, which consumes memory and can be slower than a direct iterative approach due to function call overhead.

Iteration (The Best Optimization for this Problem):

The most effective optimization for this particular scenario is to convert the recursion into a simple loop (iteration). An iterative approach avoids the function call overhead and the risk of stack overflow for a very large number of periods. It is more memory-efficient and generally faster. The method ‘CalculateFutureValueIterative’ demonstrates this superior approach.

It achieves the same **O(n)** time complexity but with better real-world performance.

**OUTPUT**

