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

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

=> Big O notation describes the **upper bound** of an algorithm's running time in terms of input size n. It helps us analyze and compare algorithms based on their growth rates.

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

* **Linear Search:**
  + Best Case: O(1) – when the element is the first item.
  + Average Case: O(n)
  + Worst Case: O(n) – when the element is not found or at the end.
* **Binary Search:**
  + Best Case: O(1) – when the element is at the middle.
  + Average Case: O(log n)
  + Worst Case: O(log n) – the list is divided repeatedly.

**Code:**

Product.cs:

using System;

public class Product

{

public int ProductId { get; set; }

public string ProductName { get; set; }

public string Category { get; set; }

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

{

ProductId = productId;

ProductName = productName;

Category = category;

}

}

Program.cs:

using System;

class Program

{

static void Main()

{

Product[] products = {

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

new Product(102, "Smartphone", "Electronics"),

new Product(103, "Shoes", "Fashion"),

new Product(104, "Book", "Education"),

};

// Linear Search

var result1 = LinearSearch(products, 103);

Console.WriteLine(result1 != null ? $"Found: {result1.ProductName}" : "Product not found.");

// Sort array by ProductId for Binary Search

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

// Binary Search

var result2 = BinarySearch(products, 103);

Console.WriteLine(result2 != null ? $"Found: {result2.ProductName}" : "Product not found.");

}

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

{

foreach (var product in products)

{

if (product.ProductId == productId)

return product;

}

return null;

}

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

{

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

while (left <= right)

{

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

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

return products[mid];

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

left = mid + 1;

else

right = mid - 1;

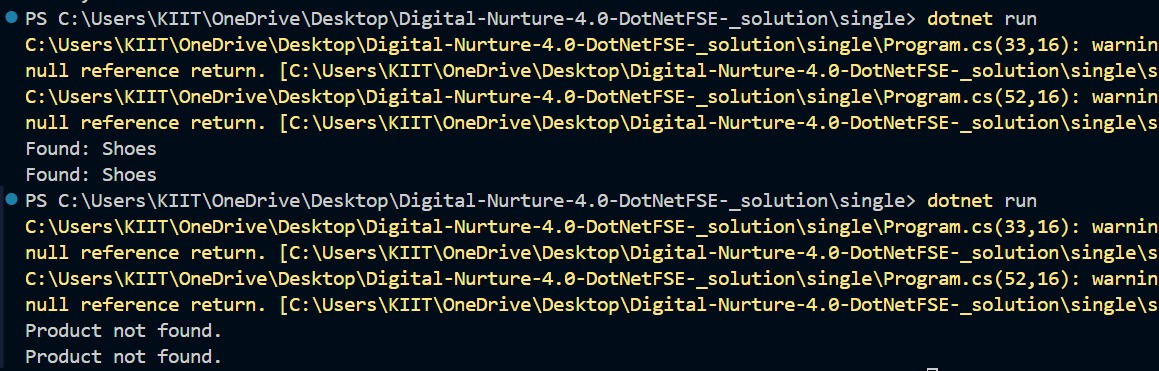
}

return null;

}

}

Output:



**Analysis**

| **Search Type** | **Time Complexity** | **Best Case** | **Average Case** | **Worst Case** |  |
| --- | --- | --- | --- | --- | --- |
| Linear Search | O(n) | O(1) | O(n) | O(n) |  |
| Binary Search | O(log n) | O(1) | O(log n) | O(log n) |  |

**Exercise 7: Financial Forecasting**

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

=> Recursion is a programming technique where a method **calls itself** to solve a problem by breaking it into smaller sub-problems.

**Example:**

* To calculate the **n-th future value**, you can use the previous value and apply the growth rate repeatedly.
* Recursive problems usually have:
  + **Base Case:** Stops further recursive calls.
  + **Recursive Case:** Reduces the problem size and calls itself.

Code:

Program.cs:

using System;

class FinancialForecast

{

public static void Main()

{

double initialValue = 1000; // Initial investment

double growthRate = 0.10; // 10% annual growth

int periods = 5; // Forecast 5 years into the future

double futureValue = ForecastFutureValue(initialValue, growthRate, periods);

Console.WriteLine($"The forecasted future value after {periods} years is: {futureValue:F2}");

}

// Recursive method to calculate future value

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

{

// Base Case

if (periods == 0)

return currentValue;

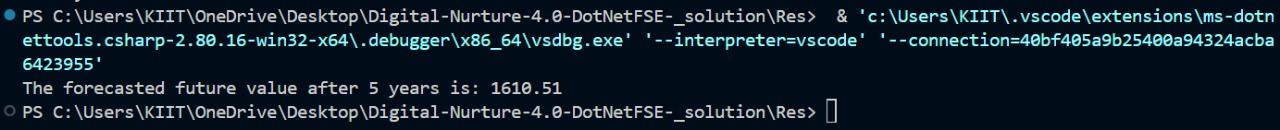
// Recursive Case

return ForecastFutureValue(currentValue \* (1 + growthRate), growthRate, periods - 1);

}

}

Output:



Q. Discuss the time complexity of your recursive algorithm.

=> **Time Complexity:**

* Each recursive call reduces periods by 1.
* Total recursive calls = n (number of periods)
* **Time Complexity: O(n)**  
  Each call does a simple multiplication and makes one recursive call.

Q.Explain how to optimize the recursive solution to avoid excessive computation.

=> Although this recursion is already **O(n)** and does not have overlapping subproblems (like Fibonacci), here are ways to improve:

**Convert to Iterative Approach:**

This removes the function call overhead.

**Code:**

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

{

double futureValue = initialValue;

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

{

futureValue \*= (1 + growthRate);

}

return futureValue;

}