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

**Big O Notation**

Big O notation describes the time complexity of an algorithm in terms of input size. It helps analyze algorithm efficiency by estimating how runtime grows as data increases. It includes best-case (Ω), average-case (Θ), and worst-case (O) scenarios.

* 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

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

**Linear Search:** O(n), good for small or unsorted data.  
**Binary Search:** O(log n), fast but needs sorted data.

**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.

**Code**

using System;

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}";

    }

}

class Program

{

    // linear Search by product name

    static Product? LinearSearch(Product[] products, string name)

    {

        foreach (var product in products)

        {

            if (product.ProductName.Equals(name, StringComparison.OrdinalIgnoreCase))

            {

                return product;

            }

        }

    }

    // binary Search by product ID (array must be sorted)

    static Product? BinarySearch(Product[] products, int id)

    {

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

        while (left <= right)

        {

            int mid = (left + right) / 2;

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

                return products[mid];

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

                left = mid + 1;

            else

                right = mid - 1;

        }

        return null;

    }

    static void Main()

    {

        Product[] products = {

            new Product(103, "Phone", "Electronics"),

            new Product(101, "Shoes", "Fashion"),

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

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

        };

        // sort by ProductId for Binary Search

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

        Console.WriteLine("Linear Search for 'Laptop':");

        var result1 = LinearSearch(products, "Laptop");

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

        Console.WriteLine("\nBinary Search for ID 104:");

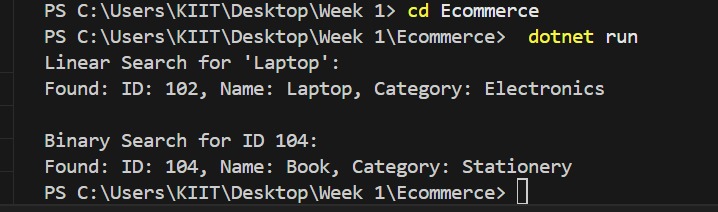
        var result2 = BinarySearch(products, 104);

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

    }

}

**OUTPUT**



**Exercise 7: Financial Forecasting**

**Recursion** is a programming technique where a method calls itself to solve smaller instances of a problem. Recursive algorithms are often used for problems that can be broken into similar sub-problems, such as mathematical computations, tree traversal, or forecasting.

In financial forecasting, recursion can help predict future values by repeatedly applying the growth rate to previous values. This mimics how investments grow over time through compound interest or percentage-based returns.

**Time Complexity of the Recursive Algorithm**

The time complexity of a recursive financial forecasting algorithm is **O(n)** because the function calls itself once for each year until it reaches the base case. Although this is efficient for small inputs, repeated recursive calls can lead to excessive memory usage or stack overflow for large input values. To optimize such recursion, it is recommended to use an **iterative approach**, which eliminates the overhead of multiple function calls, or implement **memoization**, where previously computed results are stored and reused to avoid redundant calculations. These methods help improve efficiency and ensure better performance in real-world applications.

**To optimize the recursive solution to avoid excessive computation.**

To optimize the recursive solution and avoid excessive computation, the recursion can be converted into an iterative approach, which eliminates the overhead of repeated function calls and prevents stack overflow. Another effective method is **memoization**, where previously computed results are stored and reused, reducing redundant calculations in recursive calls. These optimizations ensure better performance, especially when dealing with large inputs or deep recursive calls.

using System;

class FinancialForecast

{

    // recursive method to calculate future value

    static double PredictFutureValueRecursive(double initialValue, double growthRate, int years)

    {

        if (years == 0)

            return initialValue;

        return PredictFutureValueRecursive(initialValue, growthRate, years - 1) \* (1 + growthRate);

    }

    // iterative method to calculate future value

    static double PredictFutureValueIterative(double initialValue, double growthRate, int years)

    {

        double value = initialValue;

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

        {

            value \*= (1 + growthRate);

        }

        return value;

    }

    static void Main()

    {

        Console.Write("Enter initial investment amount: ");

        double initialValue = Convert.ToDouble(Console.ReadLine());

        Console.Write("Enter annual growth rate (in percent, e.g., 10 for 10%): ");

        double growthPercent = Convert.ToDouble(Console.ReadLine());

        double growthRate = growthPercent / 100.0;

        Console.Write("Enter number of years to forecast: ");

        int years = Convert.ToInt32(Console.ReadLine());

        double futureRecursive = PredictFutureValueRecursive(initialValue, growthRate, years);

        double futureIterative = PredictFutureValueIterative(initialValue, growthRate, years);

        Console.WriteLine($"\nForecasting result after {years} years:");

        Console.WriteLine($"Recursive prediction: {futureRecursive:F2}");

        Console.WriteLine($"Iterative prediction: {futureIterative:F2}");

    }

}

**OUTPUT**

