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

**CODE:**

using System;

class Product : IComparable<Product> {

    public int ProductId;

    public string ProductName;

    public string Category;

    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 override string ToString() {

        return $"{ProductId}: {ProductName} ({Category})";

    }

}

class ECommerceSearch {

    // Linear search: O(n)

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

        foreach (var product in products) {

            if (product.ProductId == targetId)

                return product;

        }

        return null;

    }

    // Binary search: O(log n) (array must be sorted by ProductId)

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

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

        while (left <= right) {

            int mid = (left + right) / 2;

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

                return products[mid];

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

                left = mid + 1;

            else

                right = mid - 1;

        }

        return null;

    }

    public static void Main() {

        // Unsorted for linear search

        Product[] unsortedProducts = new Product[]

        {

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

            new Product(205, "Shirt", "Clothing"),

            new Product(150, "Laptop", "Electronics"),

            new Product(300, "Notebook", "Stationery")

        };

        Console.WriteLine("[\*] Linear Search (unsorted):");

        var linearResult = LinearSearch(unsortedProducts, 150);

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

        // Sorted array for binary search

        Product[] sortedProducts = (Product[])unsortedProducts.Clone();

        Array.Sort(sortedProducts); // sort by ProductId

        Console.WriteLine("\n[\*] Binary Search (sorted):");

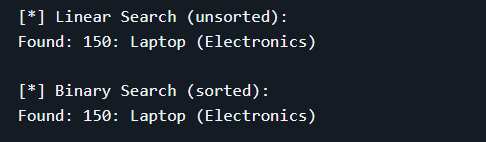
        var binaryResult = BinarySearch(sortedProducts, 150);

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

    }

}

**OUTPUT:**



**Exercise 7: Financial Forecasting**

**CODE:**

using System;

using System.Collections.Generic;

class FinancialForecasting {

// Recursive method to compute future value

public static double PredictFutureValue(double principal, double growthRate, int years) {

if (years == 0)

return principal;

return PredictFutureValue(principal, growthRate, years - 1) \* (1 + growthRate);

}

// Optimized version using memoization

public static double PredictFutureValueMemo(double principal, double growthRate, int years, Dictionary<int, double> memo) {

if (years == 0)

return principal;

if (memo.ContainsKey(years))

return memo[years];

double prev = PredictFutureValueMemo(principal, growthRate, years - 1, memo);

double result = prev \* (1 + growthRate);

memo[years] = result;

return result;

}

public static void Main() {

double principal = 1000.0;

double growthRate = 0.05; // 5% annual growth

int years = 10;

Console.WriteLine("[\*] Basic Recursive Forecast:");

double value = PredictFutureValue(principal, growthRate, years);

Console.WriteLine($"After {years} years: ${value:F2}");

Console.WriteLine("\n[\*] Optimized Recursive Forecast with Memoization:");

var memo = new Dictionary<int, double>();

double memoValue = PredictFutureValueMemo(principal, growthRate, years, memo);

Console.WriteLine($"After {years} years: ${memoValue:F2}");

}

}

**OUTPUT:**

