**DATA STRUCTURES AND ALGORITHMS**

**EXERCISE 1:- E-commerce Platform Search Function**

**1. Understanding Asymptotic Notation:-**

### What is Big O Notation?

**Big O notation** describes the **upper bound** of an algorithm's time or space complexity as the input size grows (denoted as n).

| **Notation** | **Meaning** | **Example** |
| --- | --- | --- |
| O(1) | Constant time | Accessing array element |
| O(n) | Linear time | Linear search |
| O(log n) | Logarithmic time | Binary search |
| O(n²) | Quadratic time | Nested loops |

**Best, Average, Worst-Case for Search**

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| **Linear Search** | O(1) (first match) | O(n/2) ≈ O(n) | O(n) (last/no match) |
| **Binary Search** | O(1) | O(log n) | O(log n) |

1. **Product Class Setup**

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;

}

public override string ToString()

{

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

}

}

**3. Search Implementations**

### Linear Search

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

{

foreach (var product in products)

{

if (product.ProductId == productId)

return product;

}

return null;

}

**Binary Search (requires sorted array)**

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

{

int left = 0;

int 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;

}

**4. Test and Analysis**

### Main Program to Test Both

using System;

using System.Linq;

class Program

{

static void Main()

{

Product[] products = new Product[]

{

new Product(5, "Keyboard", "Electronics"),

new Product(3, "Laptop", "Electronics"),

new Product(9, "Chair", "Furniture"),

new Product(1, "Book", "Stationery"),

new Product(7, "Monitor", "Electronics"),

};

int searchId = 7;

Console.WriteLine("➡️ Linear Search:");

var resultLinear = LinearSearch(products, searchId);

Console.WriteLine(resultLinear ?? "Product not found.");

Console.WriteLine("\n➡️ Binary Search (requires sorted array):");

var sortedProducts = products.OrderBy(p => p.ProductId).ToArray();

var resultBinary = BinarySearch(sortedProducts, searchId);

Console.WriteLine(resultBinary ?? "Product not found.");

}

// Including both search methods

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;

int 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;

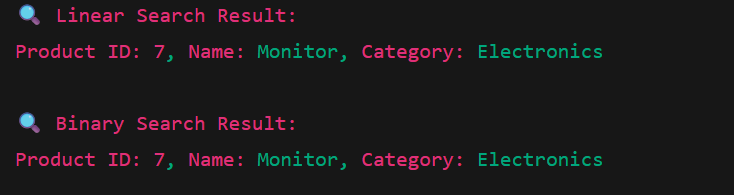
}

return null;

}

}

**OUTPUT**



1. **Time Complexity Analysis**

| **Method** | **Time Complexity** | **Space Complexity** | **Requirements** |
| --- | --- | --- | --- |
| **Linear Search** | O(n) | O(1) | Unsorted array |
| **Binary Search** | O(log n) | O(1) | Must be **sorted** array |

**Which is More Suitable?**

| **Platform Type** | **Recommendation** | **Why** |
| --- | --- | --- |
| Small dataset (e.g., < 1000 items) | **Linear Search** | Simpler, no need to sort |
| Large or frequent searches | **Binary Search** + sorting | Fast performance at scale |
| Dynamic insertions/removals | Consider **HashTable / Tree** | Faster inserts/updates/searches |

**Conclusion**

Use **Binary Search** when you can afford to keep data **sorted** (or pre-sort it).

Use **Linear Search** for small, unsorted datasets or one-time searches.

For real-world large platforms, consider **more advanced structures** like:

* Dictionary<int, Product> for O(1) lookup by ID
* Trie for name-based search
* Search Indexes or Lucene.NET for full-text search.

**EXERCISE 2:-Financial Forecasting**

**1. Understand Recursive Algorithms**

### What is Recursion?

**Recursion** is a programming technique where a method calls itself to solve a smaller instance of the same problem.

### When to Use Recursion?

When a problem can be divided into **smaller subproblems** of the same type.

Common examples: factorial, Fibonacci numbers, traversals, and **financial projections** over time.

**2. Setup: Define the Problem**

Let’s assume we want to forecast the **future value** of an investment using **compound growth**, based on a fixed growth rate and number of years.

### Formula:

FutureValue = PresentValue \* (1 + GrowthRate)^Years

1. **Implementation: Recursive Algorithm in C#**

**Recursive Forecasting Method**

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

{

if (years == 0)

return presentValue;

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

}

1. **Test Example**

using System;

class Program

{

static void Main()

{

double presentValue = 1000; // Initial investment

double growthRate = 0.05; // 5% annual growth

int years = 5;

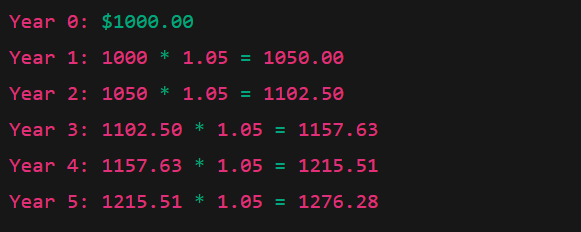
double futureValue = ForecastValueRecursive(presentValue, growthRate, years);

Console.WriteLine($"Future value after {years} years: {futureValue:C2}");

}

}

**OUTPUT**





**4. Analysis of Recursive Algorithm**

### Time Complexity

The function performs **one call per year**, so:

Time Complexity: O(n), where n = number of years

Space Complexity: O(n), due to recursion call stack

1. **Optimizing the Recursive Solution**

Use **Tail Recursion** or Replace with Iteration

#### Iterative Equivalent (More Efficient):

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

{

double result = presentValue;

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

{

result \*= (1 + growthRate);

}

return result;

}

#### Or Use Math.Pow (Best):

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

{

return presentValue \* Math.Pow(1 + growthRate, years);

}

**Summary: Recursive vs. Iterative vs. Math**

| **Method** | **Time Complexity** | **Space Complexity** | **Notes** |
| --- | --- | --- | --- |
| Recursive | O(n) | O(n) | Clear logic, but risk of stack overflow |
| Iterative | O(n) | O(1) | Safer and more efficient |
| Using Math.Pow | O(1) | O(1) | Fastest and most optimized |

**Conclusion**

For **simple educational use**, recursion is okay.  
But for **production-level financial tools**, prefer:

->Math.Pow for performance.  
->**Iterative loop** for full control and better stability.