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

Big O notation describes how the runtime or space requirements of an algorithm grow relative to input size.

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

Best Case: Quickest outcome

Average Case: Typical performance.

Worst Case: Slowest outcome

**Code**

In Product.cs-

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}";

}

}

In SearchUtility.cs-

using System;

public static class SearchUtility

{

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

{

foreach (var product in products)

{

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

return product;

}

return null;

}

public static Product BinarySearch(Product[] products, string name)

{

int left = 0;

int right = products.Length - 1;

while (left <= right)

{

int mid = (left + right) / 2;

int comparison = string.Compare(products[mid].ProductName, name, StringComparison.OrdinalIgnoreCase);

if (comparison == 0)

return products[mid];

else if (comparison < 0)

left = mid + 1;

else

right = mid - 1;

}

return null;

}

}

In Program.cs-

using System;

using System.Linq;

namespace ECommerceSearch

{

class Program

{

static void Main(string[] args)

{

Console.Write("Enter number of products: ");

int n = int.Parse(Console.ReadLine());

Product[] products = new Product[n];

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

{

Console.WriteLine($"\nEnter details for Product #{i + 1}:");

Console.Write("Product ID: ");

int id = int.Parse(Console.ReadLine());

Console.Write("Product Name: ");

string name = Console.ReadLine();

Console.Write("Category: ");

string category = Console.ReadLine();

products[i] = new Product(id, name, category);

}

Console.Write("\nEnter product name to search: ");

string searchName = Console.ReadLine();

Console.WriteLine("\n--- Linear Search Result ---");

var linearResult = SearchUtility.LinearSearch(products, searchName);

Console.WriteLine(linearResult != null ? linearResult.ToString() : "Product not found.");

Console.WriteLine("\n--- Binary Search Result ---");

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

var binaryResult = SearchUtility.BinarySearch(sortedProducts, searchName);

Console.WriteLine(binaryResult != null ? binaryResult.ToString() : "Product not found.");

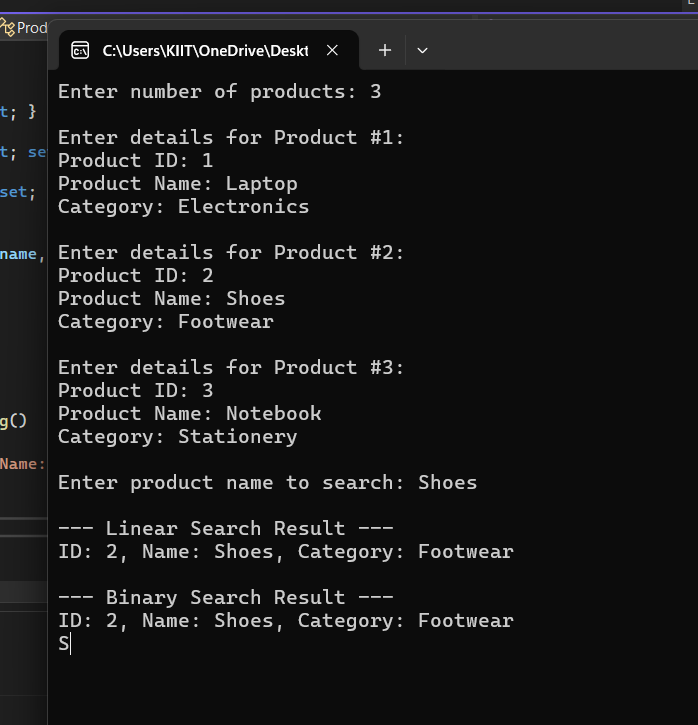
Console.ReadLine();

}

}

}

**Output**



Linear search has a time complexity of O(n) and works on unsorted data, making it suitable for small datasets or fallback scenarios, while binary search, with a faster O(log n) complexity, requires sorted data and is ideal for large datasets, which is why real-world e-commerce platforms typically use binary search or indexing structures like B-trees or B+ trees for optimized performance.  
  
  
**7: Financial Forecasting**

Recursion is a programming technique where a method calls itself to solve a problem in smaller sub-problems.

In financial forecasting, recursion can model future values based on compound growth and each year's value depends on the previous year's.

**Code**

In FinancialForecast.cs-

public static class FinancialForecast

{

// Recursive method to calculate future value

public static double PredictFutureValue(int year, double initialValue, double growthRate)

{

if (year == 0)

return initialValue;

return (1 + growthRate) \* PredictFutureValue(year - 1, initialValue, growthRate);

}

}

In Program.cs-

using System;

namespace FinancialForecasting

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Financial Forecasting Tool (Recursive)");

Console.Write("Enter initial value: ");

double initialValue = double.Parse(Console.ReadLine());

Console.Write("Enter annual growth rate (in %): ");

double ratePercent = double.Parse(Console.ReadLine());

double growthRate = ratePercent / 100.0;

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

int years = int.Parse(Console.ReadLine());

double futureValue = FinancialForecast.PredictFutureValue(years, initialValue, growthRate);

Console.WriteLine($"\nPredicted future value after {years} years: {futureValue:F2}");

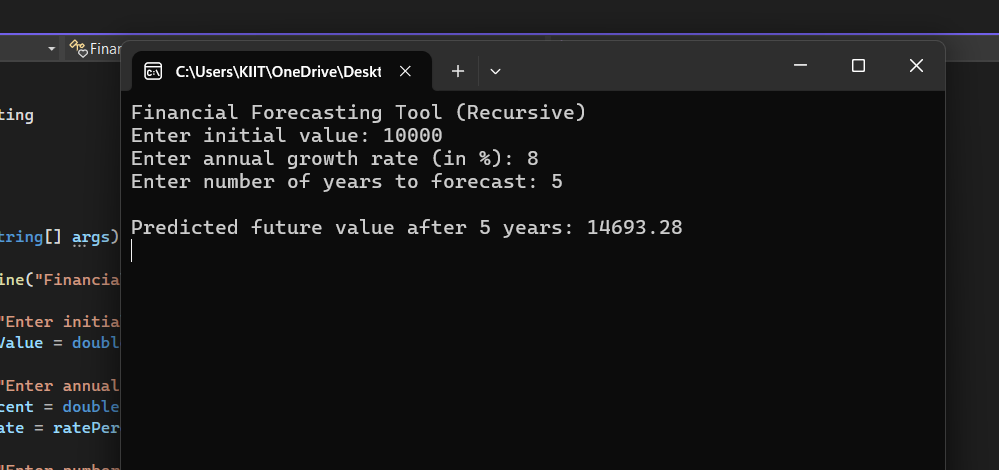
Console.ReadLine();

}

}

}

**Output**



The recursive algorithm simplifies the prediction logic by modeling future value as a repeated application of growth over time, with a time complexity of O(n); however, for large inputs, recursion can lead to stack overflow or redundant calculations, so an optimized approach like iteration is recommended.