Data Structures and Algorithms

Exercise 2: E-commerce Platform Search Function

Step 1: Understand Asymptotic Notation

Big O notation describes how algorithm performance scales with input size.

Linear Search:

Best Case: O(1), Average/Worst Case: O(n)

Binary Search:

Best Case: O(1), Average/Worst Case: O(log n)

Step 2: Setup - Product Class

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

Step 3: Linear Search Implementation

```
public static Product LinearSearch(Product[] products, string name)
{
    foreach (var product in products)
    {
        if (product.ProductName.Equals(name, StringComparison.OrdinalIgnoreCase))
        {
            return product;
        }
    }
    return null;
}
```

Step 3: Binary Search Implementation

```
public static Product BinarySearch(Product[] products, string name)
{
    int left = 0, right = products.Length - 1;
    while (left <= right)
    {
        int mid = (left + right) / 2;
        int cmp = string.Compare(products[mid].ProductName, name, StringComparison.OrdinalIgnoreCase);
        if (cmp == 0) return products[mid];
        else if (cmp < 0) left = mid + 1;
        else right = mid - 1;
    }
    return null;
}
```

Step 4: Main Method and Testing

```
class Program
{
    static void Main()
    {
        Product[] products = new Product[] {
            new Product(1, "Laptop", "Electronics"),
            new Product(2, "Shirt", "Apparel"),
            new Product(3, "Headphones", "Electronics"),
            new Product(4, "Shoes", "Footwear"),
            new Product(5, "Keyboard", "Electronics")
        };

        var sortedProducts = products.OrderBy(p => p.ProductName).ToArray();
        string searchName = "Keyboard";

        var linearResult = LinearSearch(products, searchName);
        Console.WriteLine("Linear Search Result: " + (linearResult != null ? linearResult.ToString() : "Not Found"));

        var binaryResult = BinarySearch(sortedProducts, searchName);
        Console.WriteLine("Binary Search Result: " + (binaryResult != null ? binaryResult.ToString() : "Not Found"));
    }
}
```

Step 5: Output Screenshot

```
Linear Search Result: 5 - Keyboard (Electronics)
Binary Search Result: 5 - Keyboard (Electronics)
```

Step 4: Analysis

Time Complexity Comparison:

- Linear Search: O(n)
- Binary Search: O(log n), requires sorted array

Binary Search is more suitable for optimized search performance, assuming the product list is sorted. For unsorted or real-time updated lists, linear search or dictionary-based lookup may be better.

Exercise 7: Financial Forecasting

1. Understanding Recursive Algorithms

Recursion is a technique where a function calls itself to solve smaller instances of a problem. It helps simplify code for problems that exhibit repetitive patterns, like computing financial forecasts over time.

2. Setup

We will build a tool to forecast future value based on an initial investment, growth rate, and time period.

3. Recursive Implementation

The following recursive method calculates the future value:

```
def future_value(principal, rate, years):
    if years == 0:
        return principal
    return future_value(principal * (1 + rate), rate, years - 1)

# Example usage:
print(future_value(10000, 0.05, 5))
```

4. Output

```
12762.815625
```

5. Analysis

Time Complexity: O(n), where n is the number of years.

To optimize, we can use memoization or convert to an iterative approach to avoid redundant calls.