DATA STRUCTURES AND ALGORITHMS

EXERCISE-2:E-commerce Platform Search Function

Step 1: Understand Asymptotic Notation

Big O Notation Explanation:

Big O notation describes the **upper bound** of an algorithm's running time. It provides a **high-level understanding of the time or space complexity** of an algorithm as the input size grows.

- **Why it's useful:** Helps you **predict performance** and **choose efficient algorithms**.

Search Operation Scenarios:

Let's consider an array of n products:

- **Best Case:**
  - *Linear Search:* Item found at the beginning → **O(1)**
  - *Binary Search:* Item found at middle in first check → **O(1)**
- **Average Case:**
  - *Linear Search:* Item somewhere in the middle → **O(n/2)** ⇒ simplified to **O(n)**
  - *Binary Search:* Cut half repeatedly → **O(log n)**
- **Worst Case:**
  - *Linear Search:* Item not found (entire list searched) → **O(n)**
  - *Binary Search:* Logarithmic checks until failure/success → **O(log n)**

Step 2: Setup - Product Class

```
public class Product {
    int productId;
```

```
    String productName;
    String category;

    public Product(int productId, String productName, String category) {
        this.productId = productId;
        this.productName = productName;
        this.category = category;
    }

    @Override
    public String toString() {
        return productId + " - " + productName + " (" + category + ")";
    }
}
```

Step 3: Implementation - Linear and Binary Search

```
import java.util.Arrays;
import java.util.Comparator;

public class SearchEngine {

    // Linear Search
    public static Product linearSearch(Product[] products, String targetName) {
        for (Product product : products) {
            if (product.productName.equalsIgnoreCase(targetName)) {
                return product;
            }
        }
        return null;
    }

    // Binary Search (products must be sorted by productName)
    public static Product binarySearch(Product[] products, String targetName) {
        int left = 0;
        int right = products.length - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;
            int cmp = products[mid].productName.compareToIgnoreCase(targetName);

            if (cmp == 0)
                return products[mid];
            else if (cmp < 0)
```

```
            left = mid + 1;
        else
            right = mid - 1;
    }

    return null;
}

// Sample test
public static void main(String[] args) {
    Product[] productList = {
        new Product(101, "Laptop", "Electronics"),
        new Product(102, "Shampoo", "Personal Care"),
        new Product(103, "Sneakers", "Footwear"),
        new Product(104, "Chair", "Furniture"),
        new Product(105, "Phone", "Electronics")
    };

    // Linear Search (unsorted array)
    Product result1 = linearSearch(productList, "Phone");
    System.out.println("Linear Search Result: " + (result1 != null ? result1 : "Not Found"));

    // Sort products for binary search
    Arrays.sort(productList, Comparator.comparing(p -> p.productName));

    // Binary Search (sorted array)
    Product result2 = binarySearch(productList, "Phone");
    System.out.println("Binary Search Result: " + (result2 != null ? result2 : "Not Found"));
}
}
```

Step 4: Analysis

Search Type	Time Complexity	Space Complexity	Sorted Required?	Use Case
Linear Search	O(n)	O(1)	No	Small data or unsorted list
Binary Search	O(log n)	O(1)	Yes	Large, sorted datasets

Which is more suitable?

- For an **e-commerce platform**, where **thousands or millions of products** are searched frequently, **Binary Search** is more suitable **if the data is sorted or indexed**.
- In real systems, advanced structures like **HashMaps**, **B-Trees**, or **Search Indexes** (like Elasticsearch) are used for even better performance.

Conclusion:

- **Linear Search** is simple and flexible but **slow for large datasets**.
- **Binary Search** is significantly **faster (O(log n))**, ideal when data can be kept sorted.
- **For real-world applications**, use **indexing/search libraries or databases** for optimal performance.

Exercise 7: Financial Forecasting

Step 1: Understand Recursive Algorithms

What is Recursion?

Recursion is a programming technique where a function calls **itself** to solve smaller instances of a problem until a **base case** is reached.

Why Use Recursion?

- Simplifies problems that have **repetitive substructure** (like tree traversal, factorial, Fibonacci).
- Great for breaking down complex calculations such as **compound interest** or **future value prediction** where results build on previous steps.

Step 2: Setup — Define the Recursive Method Structure

We will forecast future financial values using this formula:

FV=PV×(1+r)nFV = PV \times (1 + r)^n

Where:

- `FV` = future value
- `PV` = present value
- `r` = growth rate (e.g. 0.05 for 5%)
- `n` = number of periods

Step 3: Implementation in Java

```
public class FinancialForecast {


    // Recursive method to calculate future value
    public static double calculateFutureValue(double presentValue, double rate, int years) {
        // Base case: no years left
        if (years == 0) {
            return presentValue;
        }


        // Recursive case: grow the value year by year
        return calculateFutureValue(presentValue * (1 + rate), rate, years - 1);
    }


    public static void main(String[] args) {
        double initialValue = 10000;  // Initial investment
        double annualGrowthRate = 0.07;  // 7% per year
        int forecastYears = 10;  // Forecast over 10 years
```

```
        double futureValue = calculateFutureValue(initialValue, annualGrowthRate, forecastYears);

        System.out.printf("Future value after %d years: %.2f%n", forecastYears, futureValue);

    }

}
```

Step 4: Analysis

Time Complexity:

- Each call reduces `years` by 1 ⇒ **O(n)** time where `n` is the number of years.
- Stack space also grows with `n` ⇒ **O(n)** space complexity (due to recursive call stack).

Problem:

Recursive depth grows linearly with `years`, which can be risky (stack overflow) if `n` is large.

Optimization: Use Iteration or Memoization

Optimized Iterative Version:

```
public static double calculateFutureValueIterative(double presentValue, double rate, int years) {

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

        presentValue *= (1 + rate);

    }

    return presentValue;

}
```

- **Time Complexity:** O(n)
- **Space Complexity:** O(1)
- Avoids deep recursion and is safer for large inputs.

Summary

Feature	Recursive Version	Iterative Version
Simplicity	Elegant, matches formula	Explicit, step-wise
Time Complexity	O(n)	O(n)
Space Complexity	O(n) (call stack)	O(1)
Risk	Stack overflow for large n	Safe for large n

Final Thought

Recursion simplifies the logic of future value calculations but should be optimized or converted to iteration for real-world financial forecasting tools, especially with large time spans.