Exercise 2: E-commerce Platform Search Function

Scenario:

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

Steps:

1. **Understand Asymptotic Notation:**
    - o Explain Big O notation and how it helps in analyzing algorithms.
    - o Describe the best, average, and worst-case scenarios for search operations.
2. **Setup:**
    - ○ Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
    - o Implement linear search and binary search algorithms.
    - o Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
    - o Compare the time complexity of linear and binary search algorithms.
    - o Discuss which algorithm is more suitable for your platform and why.

Solution:-

1.

Big O Notation (O)

- Big O describes the **upper bound** on the time or space complexity of an algorithm.
- It helps in evaluating **scalability** as the input size (n) grows.

Best, Average, and Worst Cases

Algorithm	Best Case	Average Case	Worst Case
Linear Search	O(1)	O(n/2) ≈ O(n)	O(n)
Binary Search	O(1)	O(log n)	O(log n)

2.

```
package ecommerce;

public class Product {
    private int productId;
    private String productName;
    private String category;

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

    public int getProductId() {
        return productId;
    }
```

```java
    public String getProductName() {
        return productName;
    }

    public String getCategory() {
        return category;
    }

    public String toString() {
        return "Product ID: " + productId + ", Name: " + productName + ", Category: "
+ category;
    }
}
```

3. Linear Search

```java
package ecommerce;

public class LinearSearch {
    public static Product search(Product[] products, String targetName) {
        for (Product product : products) {
            if (product.getProductName().equalsIgnoreCase(targetName)) {
                return product;
            }
        }
        return null;
    }
}
```

Binary search

```java
package ecommerce;
import java.util.Arrays;
import java.util.Comparator;
public class BinarySearch {
    public static Product search(Product[] products, String targetName) {
        Arrays.sort(products, Comparator.comparing(Product::getProductName));
        int low = 0, high = products.length - 1;

        while (low <= high) {
            int mid = (low + high) / 2;
            String midName = products[mid].getProductName();

            int comparison = targetName.compareToIgnoreCase(midName);
            if (comparison == 0) {
                return products[mid];
            } else if (comparison < 0) {
                high = mid - 1;
            } else {
                low = mid + 1;
```

```
            }
        }

        return null;
    }
}
```

Test class

```
package ecommerce;
import java.util.Arrays;
import java.util.Comparator;
public class BinarySearch {
    public static Product search(Product[] products, String targetName) {
        Arrays.sort(products, Comparator.comparing(Product::getProductName));
        int low = 0, high = products.length - 1;
        while (low <= high) {
            int mid = (low + high) / 2;
            String midName = products[mid].getProductName();

            int comparison = targetName.compareToIgnoreCase(midName);
            if (comparison == 0) {
                return products[mid];
            } else if (comparison < 0) {
                high = mid - 1;
            } else {
                low = mid + 1;
            }
        }

        return null;
    }
}
```

4.

Criteria	Linear Search	Binary Search
Speed (Large Data)	Slower (O(n))	Faster (O(log n))
Data Sorted?	Works on any data	Needs sorted data
Simplicity	Very simple	Slightly more complex

Recommendation:

For **small or unsorted product lists**, use **Linear Search**.
For **large, sorted product lists**, use **Binary Search** for **better performance**.

Exercise 7: Financial Forecasting

Scenario:

You are developing a financial forecasting tool that predicts future values based on past data.

Steps:

1. **Understand Recursive Algorithms:**
   - Explain the concept of recursion and how it can simplify certain problems.
2. **Setup:**
   - Create a method to calculate the future value using a recursive approach.
3. **Implementation:**
   - Implement a recursive algorithm to predict future values based on past growth rates.
4. **Analysis:**
   - Discuss the time complexity of your recursive algorithm.
   - Explain how to optimize the recursive solution to avoid excessive computation.

Solution:-

1. What is Recursion?

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

- Every recursive function must have:
  - A **base case** to stop recursion.
  - A **recursive case** to reduce the problem.

Why Use Recursion?

- Natural fit for problems that can be broken down into subproblems (e.g., forecasting, Fibonacci, tree traversal).
- Makes code **elegant and easy to read**, but may need optimization for performance.

3.

```
package forecast;

public class FinancialForecast {
  public static double futureValueRecursive(double initialValue, double growthRate, int years) {
    if (years == 0) {
      return initialValue; // base case
    }
    return futureValueRecursive(initialValue, growthRate, years - 1) * (1 + growthRate);
  }

  public static void main(String[] args) {
    double initialInvestment = 10000; // ₹10,000
    double annualGrowthRate = 0.08;  // 8%
    int forecastYears = 5;
    double result = futureValueRecursive(initialInvestment, annualGrowthRate, forecastYears);
    System.out.printf("📈 Future Value after %d years = ₹%.2f%n", forecastYears, result);
  }
}
```

OUTPUT:-

Future Value after 5 years = ₹14693.28

4.

Time Complexity Analysis

- The recursive method calls itself once per year → **Time Complexity: O(n)**
- **Space Complexity** due to recursion stack: **O(n)**