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:**
   - Explain Big O notation and how it helps in analyzing algorithms.
   - 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:**
   - Implement linear search and binary search algorithms.
   - Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
   - Compare the time complexity of linear and binary search algorithms.
   - Discuss which algorithm is more suitable for your platform and why.

→ Step 1: Understand Asymptotic Notation

Big O Notation

- **Big O notation** describes the **upper bound** of an algorithm's running time in terms of input size n.
- It helps evaluate algorithm performance **independent of machine specifications**.

Big O Scenarios for Searching

Case	**Linear Search**	**Binary Search**
Best	O(1) – First element	O(1) – Middle element
Average	O(n/2) ≈ O(n)	O(log n)
Worst	O(n)	O(log n)

Binary Search requires the array to be sorted, whereas linear search does not.

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

Step 3: Implementation

Linear Search

```
public class SearchAlgorithms {
    public static Product linearSearch(Product[] products, String name) {
        for (Product p : products) {
            if (p.productName.equalsIgnoreCase(name)) {
                return p;
            }
        }
        return null;
    }
```

Binary Search (Assumes sorted array by productName)

```
    public static Product binarySearch(Product[] products, String name) {
        int low = 0;
        int high = products.length - 1;
 while (low <= high) {
            int mid = (low + high) / 2;
            int compare = products[mid].productName.compareToIgnoreCase(name);
            if (compare == 0) return products[mid];
            else if (compare < 0) low = mid + 1;
```

```
            else high = mid - 1;
        }
        return null;
    }
}
```

Example Main Class

```
import java.util.Arrays;
import java.util.Comparator;
public class Main {
    public static void main(String[] args) {
        Product[] products = {
            new Product(101, "Keyboard", "Electronics"),
            new Product(102, "Mouse", "Electronics"),
            new Product(103, "Shampoo", "Personal Care"),
            new Product(104, "Notebook", "Stationery")
        };
        Product found1 = SearchAlgorithms.linearSearch(products, "Mouse");
        System.out.println("Linear Search Found: " + (found1 != null ? found1 : "Not Found"));
        Arrays.sort(products, Comparator.comparing(p -> p.productName));
        Product found2 = SearchAlgorithms.binarySearch(products, "Mouse");
        System.out.println("Binary Search Found: " + (found2 != null ? found2 : "Not Found"));
    }
}
```

Step 4: Analysis

Time Complexities

Search Type	**Time Complexity**	**When to Use**
Linear Search	O(n)	Small datasets or unsorted arrays
Binary Search	O(log n)	Sorted arrays and large datasets

Recommendation

- **Binary Search** is preferred for large and sorted datasets due to logarithmic performance.

- If data isn’t sorted or updates frequently, **Linear Search** may be used for simplicity.
- For real-world e-commerce, using **hash-based indexing** or **search trees (e.g., Trie, B-trees)** is optimal.

OUTPUT: