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

**1. Asymptotic Notation**

**Big O Notation:**

Big O notation is a formal way to describe the performance and scalability of an algorithm in terms of time or space complexity. It specifically represents the upper bound of an algorithm's growth rate relative to the size of the input data. This notation helps developers and system architects make informed decisions when choosing between different algorithms, especially in high-performance or data-intensive applications.

Understanding Big O notation allows us to compare algorithms without implementing them and provides insight into how they will perform as data volume increases. While actual execution time may vary depending on hardware, Big O focuses on the algorithm's fundamental behavior, abstracting away implementation details.

Common Big O notations include:

* **O(1)**: Constant time – performance remains the same regardless of input size.
* **O(log n)**: Logarithmic time – performance improves significantly as input grows.
* **O(n)**: Linear time – performance decreases proportionally with input size.
* **O(n log n)**: Log-linear time – commonly seen in efficient sorting algorithms.
* **O(n^2)**: Quadratic time – performance degrades rapidly; often seen in nested loops.

**Best, Average, and Worst Case for Search Operations:**

When evaluating search operations, it is important to understand their behavior in three scenarios:

| **Search Type** | **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**: The desired element is found immediately.
* **Average Case**: The element is somewhere in the middle or evenly distributed.
* **Worst Case**: The element is not in the list or is located at the end.

This categorization aids in predicting how a search algorithm will perform in practical situations.

**2. Setup**

To enable search functionality on an e-commerce platform, we must define a class to represent products with key attributes used in searching and filtering. These typically include:

* Product ID
* Product Name
* Product Category

A class definition allows us to organize and manage product data efficiently.

**Product Class:**

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 productId + " - " + productName + " (" + category + ")";

}

}

This class will serve as the foundation for storing product data and applying search algorithms.

**3. Implementation**

We will implement two types of search algorithms:

* Linear Search (unsorted array)
* Binary Search (sorted array by product name)

**Linear Search:**

public static Product linearSearch(Product[] products, String targetName) {

for (Product p : products) {

if (p.productName.equalsIgnoreCase(targetName)) {

return p;

}

}

return null;

}

Linear search is simple and does not require sorting. It checks each element one-by-one and works well with small or unsorted datasets.

**Sorting for Binary Search:**

Before using binary search, we must ensure the array is sorted based on the product name.

import java.util.Arrays;

import java.util.Comparator;

public static void sortProductsByName(Product[] products) {

Arrays.sort(products, Comparator.comparing(p -> p.productName.toLowerCase()));

}

This uses Java's Arrays.sort and a lambda comparator to sort products alphabetically by name.

**Binary Search:**

public static Product binarySearch(Product[] products, String targetName) {

int left = 0;

int right = products.length - 1;

while (left <= right) {

int mid = (left + right) / 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;

}

Binary search repeatedly divides the search interval in half and is much faster than linear search for sorted datasets.

**4. Analysis**

**Time Complexity Comparison:**

| **Algorithm** | **Time Complexity** | **Requires Sorted Data** | **Space Complexity** |
| --- | --- | --- | --- |
| Linear Search | O(n) | No | O(1) |
| Binary Search | O(log n) | Yes | O(1) |

Linear search scales poorly with large datasets as every element may need to be checked. Binary search is more efficient but only applicable when the data is sorted.

**Algorithm Suitability for E-commerce:**

In e-commerce applications, performance and scalability are crucial. Binary search provides optimal performance for search queries, but requires upfront sorting and maintenance of sorted order, which can add complexity when the dataset is dynamic (e.g., when products are frequently added, removed, or updated).

For smaller datasets or infrequent searches, linear search offers simplicity and flexibility. For larger datasets with frequent access and read-heavy workloads, binary search (along with auxiliary structures like indexes or hash maps) offers significant performance benefits.

In real-world systems, search functionality is often implemented using specialized data structures (e.g., B-trees, tries) or full-text search engines (e.g., Elasticsearch), but binary search remains an important foundational concept.

**Conclusion**

Understanding the difference between linear and binary search helps in selecting the right approach based on dataset size, search frequency, and performance requirements. While linear search is easy to implement and suitable for unsorted data, binary search is preferred for sorted and large datasets due to its logarithmic time complexity.

Grasping Big O notation is essential for evaluating and comparing algorithm performance. Choosing the appropriate algorithm depends on the specific needs and structure of the e-commerce platform. The balance between implementation complexity and runtime efficiency should guide the selection of the search method.

**Output**



