E- commerce Platform Search Function-EX 2

1. Understanding Asymptotic Notation

Big O Notation

Big O notation is a mathematical notation used to describe the upper bound of an algorithm's runtime or space requirements as a function of the input size. It provides a high-level understanding of the algorithm's efficiency, allowing us to compare different algorithms regardless of hardware or specific implementation details.

Best, Average, and Worst-Case Scenarios:

- **Best Case**: The scenario where the algorithm performs the minimum number of operations. For a search operation, this could be finding the element on the first try.
- **Average Case**: The expected scenario considering all possible inputs, giving a measure of typical performance.
- **Worst Case**: The scenario where the algorithm performs the maximum number of operations. This often provides the most crucial insight into an algorithm's efficiency, as it shows the maximum possible time or space complexity.

2. Setup

The setup involves defining a class Product with attributes for searching, such as productId, productName, and category. This class encapsulates the essential information about a product.

3. Implementation

The code provided implements both linear and binary search algorithms and stores products in an array.

Product Class

Defines the properties of a product:

```
class Product {
   private String productId;
   private String productName;
   private String category;


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

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

    public String getProductName() {
        return productName;
    }

    public String getCategory() {
        return category;
    }
}
```

Linear Search

Iterates through each element in the array to find the target product by name.

```
class LinearSearch {
    public static int linearSearch(Product[] products, String productName) {
        for (int i = 0; i < products.length; i++) {
            if (products[i].getProductName().equalsIgnoreCase(productName)) {
                return i;
            }
        }
        return -1;
    }
    // Time complexity: O(n)
}
```

Binary Search

Assumes the array is sorted and uses a divide-and-conquer approach to find the target product.

```
class BinarySearch {
```

```
    public static int binarySearch(Product[] products, String productName) {
        int left = 0;
        int right = products.length - 1;

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

            if (result == 0) {
                return mid;
            }

            if (result > 0) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return -1;
    }
    // Time complexity: O(log n)
}
```

Sorting the Products

Sorting is necessary for binary search. The example uses bubble sort for simplicity, though more efficient algorithms like quicksort or mergesort are preferred for larger datasets.

```
class SortProducts {
    public static void sortProducts(Product[] products) {
```

```
        for (int i = 0; i < products.length - 1; i++) {
            for (int j = 0; j < products.length - 1 - i; j++) {
                if (products[j].getProductName().compareToIgnoreCase(products[j + 1].getProductName()) > 0)
{
                    Product temp = products[j];
                    products[j] = products[j + 1];
                    products[j + 1] = temp;
                }
            }
        }
    }
}
```

4. Analysis

Time Complexity Comparison

- **Linear Search**:
    - Best Case: O(1) (when the target is the first element)
    - Average Case: O(n/2) = O(n)
    - Worst Case: O(n) (when the target is the last element or not present)
- **Binary Search**:
    - Best Case: O(1) (when the target is the middle element)
    - Average Case: O(log n)
    - Worst Case: O(log n) (when the target is not present)

Suitability for the Platform

For an e-commerce platform where the dataset can be large, binary search is more suitable due to its logarithmic time complexity. This efficiency is crucial for ensuring quick response times, especially when the number of products is large. However, binary search requires the data to be sorted, which adds a preprocessing step.