## Exercise 2: E-commerce Platform Search Function

### Step 1: Understand Asymptotic Notation

#### Explain Big O notation and how it helps in analyzing algorithms.

Big O notation is a mathematical representation used to describe the upper bound of an algorithm's time or space complexity. It provides a high-level understanding of the algorithm's performance by focusing on its growth rate as the input size increases. Big O notation helps in analyzing and comparing algorithms by indicating the worst-case scenario of their execution time or space requirements.

#### Best, Average, and Worst-Case Scenarios

* **Best Case**: The scenario where the algorithm performs the minimum number of operations. For example, in a search operation, the best case is when the target element is the first element in the list.
* **Average Case**: The scenario that represents the expected number of operations performed by the algorithm over all possible inputs. It provides a realistic measure of an algorithm's performance.
* **Worst Case**: The scenario where the algorithm performs the maximum number of operations. For a search operation, the worst case is when the target element is the last element in the list or not present at all.

### Step 2: Setup

#### Define the Product Class

// Java implementation

public 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;

}

// Getters

public String getProductId() {

return productId;

}

public String getProductName() {

return productName;

}

public String getCategory() {

return category;

}

}

### Step 3: Implementation

#### Linear Search Algorithm

// Java implementation of linear search

public class LinearSearchFunction {

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

for (Product product : products) {

if (product.getProductName().equalsIgnoreCase(targetName)) {

return product;

}

}

return null;

}

}

#### Binary Search Algorithm

// Java implementation of binary search

import java.util.Arrays;

import java.util.Comparator;

public class BinarySearchFunction {

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

Arrays.sort(products, Comparator.comparing(Product::getProductName));

int left = 0;

int right = products.length - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

int comparison = products[mid].getProductName().compareToIgnoreCase(targetName);

if (comparison == 0) {

return products[mid];

}

if (comparison < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

}

### Step 4: Analysis

#### Time Complexity Comparison

* **Linear Search**:
  + Best Case: O(1) (target element is the first element)
  + Average Case: O(n) (target element is somewhere in the middle)
  + Worst Case: O(n) (target element is the last element or not present)
* **Binary Search**:
  + Best Case: O(1) (target element is the middle element)
  + Average Case: O(log n) (each step reduces the search space by half)
  + Worst Case: O(log n) (target element is not present and search space is halved until it is empty)

#### Algorithm Suitability

For an e-commerce platform:

* **Linear Search** is simple and does not require the data to be sorted. However, it is inefficient for large datasets due to its O(n) time complexity.
* **Binary Search** is more efficient with O(log n) time complexity, making it suitable for large datasets. However, it requires the data to be sorted, which adds an additional overhead of O(n log n) for sorting.

**Recommendation**:

Use **binary search** for a more optimized search functionality, given that the data is sorted. Sorting the data can be done once and maintained with minor adjustments as new products are added or removed.