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

**1. Understand Asymptotic Notation:**

**Big O Notation:**

* **Definition: Big O notation describes the upper bound of the time complexity of an algorithm. It gives an idea of the algorithm’s performance in the worst-case scenario as the input size grows.**
* **Purpose: Helps in analyzing and comparing the efficiency of algorithms, particularly how they scale with increasing data sizes**

**Linear Search:**

* **Best Case: O(1) - The target element is the first element in the array.**
* **Average Case: O(n) - The target element is somewhere in the middle of the array.**
* **Worst Case: O(n) - The target element is the last element or not in the array.**

**Binary Search:**

* **Best Case: O(1) - The target element is at the middle of the sorted array.**
* **Average Case: O(log n) - The target element is somewhere in the array, and each step halves the search space.**
* **Worst Case: O(log n) - The target element is either at the very end or not present, and the search space is halved each time.**

**2. Setup**

**// Product.java**

**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;**

**}**

**@Override**

**public String toString() {**

**return "Product{" +**

**"productId='" + productId + '\'' +**

**", productName='" + productName + '\'' +**

**", category='" + category + '\'' +**

**'}';**

**}**

**}**

**3. Implementation**

**// LinearSearch.java**

**public class LinearSearch {**

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

**for (Product product : products) {**

**if (product.getProductId().equals(searchTerm) ||**

**product.getProductName().equals(searchTerm) ||**

**product.getCategory().equals(searchTerm)) {**

**return product;**

**}**

**}**

**return null;**

**}**

**}**

**// BinarySearch.java**

**import java.util.Arrays;**

**public class BinarySearch {**

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

**int left = 0;**

**int right = products.length - 1;**

**while (left <= right) {**

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

**Product midProduct = products[mid];**

**if (midProduct.getProductId().equals(searchTerm) ||**

**midProduct.getProductName().equals(searchTerm) ||**

**midProduct.getCategory().equals(searchTerm)) {**

**return midProduct;**

**}**

**if (midProduct.getProductId().compareTo(searchTerm) < 0) {**

**left = mid + 1;**

**} else {**

**right = mid - 1;**

**}**

**}**

**return null;**

**}**

**public static void main(String[] args) {**

**// Example of usage**

**Product[] products = {**

**new Product("1", "Laptop", "Electronics"),**

**new Product("2", "Phone", "Electronics"),**

**new Product("3", "Shoes", "Fashion")**

**};**

**// Sort the products by ID or name before binary search**

**Arrays.sort(products, (p1, p2) -> p1.getProductId().compareTo(p2.getProductId()));**

**Product foundProduct = binarySearch(products, "2");**

**System.out.println(foundProduct);**

**}**

**}**

**4. Analysis**

**Time Complexity Comparison:**

* **Linear Search: O(n) - Scans through each element sequentially.**
* **Binary Search: O(log n) - Efficiently reduces the search space by half with each comparison.**

**For an e-commerce platform where search speed is critical and the dataset is large, binary search is more efficient if the products are sorted. Otherwise, consider maintaining a sorted list or using additional data structures (like hash-based indices) to optimize search performance further.**