**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.

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

Big O notation is a mathematical concept used to describe the efficiency of an algorithm—specifically, how its runtime or space requirements grow as the size of the input increases.

**Purpose of Big O Notation**

* It gives a **high-level understanding** of an algorithm’s performance.
* It focuses on the **worst-case** scenario (unless stated otherwise).
* It helps compare different algorithms **independently of hardware or programming language**.

Big O describes the **relationship between input size (n)** and the number of **steps (operations)** an algorithm takes.

| **Big O** | **Description** | **Example Algorithm** |
| --- | --- | --- |
| **O(1)** | Constant time | Accessing an array element |
| **O(log n)** | Logarithmic time | Binary Search |
| **O(n)** | Linear time | Linear Search |
| **O(n log n)** | Linearithmic time | Merge Sort, Quick Sort |
| **O(n²)** | Quadratic time | Bubble Sort, Nested Loops |

**Describe the best, average, and worst-case scenarios for search operations.**

**Best, Average, and Worst-Case Scenarios in Search Operations**

When analyzing **search algorithms**, it's important to consider how they perform in **different scenarios** depending on the **position of the target element** or whether it's **found at all**.

**✅ 1. Linear Search**

**Definition**: Searches each element one by one in an **unsorted** list.

| **Case** | **Scenario** | **Time Complexity** |
| --- | --- | --- |
| **Best Case** | Target is the **first element** | **O(1)** |
| **Average Case** | Target is **in the middle** | **O(n/2)** ≈ **O(n)** |
| **Worst Case** | Target is the **last element** or **not present** | **O(n)** |

**2. Binary Search**

**Definition**: Works on a **sorted** list by repeatedly dividing the search range in half.

| **Case** | **Scenario** | **Time Complexity** |
| --- | --- | --- |
| **Best Case** | Target is the **middle element** at first check | **O(1)** |
| **Average Case** | Target is **somewhere in the list** | **O(log n)** |
| **Worst Case** | Target is **not in the list** or last to be checked | **O(log n)** |

**Code:**

import java.util.Arrays;

import java.util.Scanner;

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;

    }

}

public class ECommerceSearch {

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

        for (int i = 0; i < products.length; i++) {

            if (products[i].productName.equalsIgnoreCase(targetName)) {

                return products[i];

            }

        }

        return null;

    }

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

        int left = 0, 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;

    }

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        Product[] products = {

            new Product(101, "Laptop", "Electronics"),

            new Product(102, "Shoes", "Fashion"),

            new Product(103, "Phone", "Electronics"),

            new Product(104, "Watch", "Accessories"),

            new Product(105, "Shirt", "Fashion")

        };

        System.out.print("Enter product name to search: ");

        String searchTerm = scanner.nextLine();

        Product linearResult = linearSearch(products, searchTerm);

        System.out.println("\nLinear Search Result:");

        System.out.println(linearResult != null ? linearResult : "Product not found");

        Arrays.sort(products, (a, b) -> a.productName.compareToIgnoreCase(b.productName));

        Product binaryResult = binarySearch(products, searchTerm);

        System.out.println("\nBinary Search Result:");

        System.out.println(binaryResult != null ? binaryResult : "Product not found");

        scanner.close();

    }

}

**Output:**

**A screen shot of a computer

AI-generated content may be incorrect.**

**Compare the time complexity of linear and binary search algorithms.**

Comparison Table

| Feature | Linear Search | Binary Search |
| --- | --- | --- |
| Time (Best) | O(1) | O(1) |
| Time (Average) | O(n) | O(log n) |
| Time (Worst) | O(n) | O(log n) |
| Data Required | Unsorted | Must be sorted |
| Performance | Slower on large data | Much faster on large data |

**Discuss which algorithm is more suitable for your platform and why.**

Best Choice for E-Commerce Platform: Binary Search

1. **Large product catalogs** (thousands to millions of products)
2. **Frequent search queries** from users
3. **Low-latency expectations** (fast results needed)
4. Can pre-sort or index data for efficiency