**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.

**ANSWERS**

**Step 1: Understand Asymptotic Notation**

Big O notation is a mathematical notation used to **describe the time complexity of algorithms** as the input size increases. It helps estimate how an algorithm performs under different conditions, particularly with **large datasets**.

Big O abstracts away machine-specific details and allows us to evaluate **algorithm scalability and efficiency**.

Big O gives us a **worst-case upper bound**, allowing developers to:

* Predict how performance changes as data grows.
* Compare algorithms objectively.
* Choose the best algorithm for the situation.

For example, an algorithm with a time complexity of **O(n)** means the time taken increases linearly with input size. Big O notation ignores constant factors and lower-order terms to focus purely on the growth rate. This makes it particularly useful in comparing algorithms' performance, especially for large datasets, without being affected by the underlying hardware or programming language.

### **Best, Average, and Worst Cases**

|  |  |  |
| --- | --- | --- |
| **Case** | **Linear Search** | **Binary Search** |
| Best Case | O(1) — first element | O(1) — middle element |
| Average Case | O(n/2) ≈ O(n) | O(log n) |
| Worst Case | O(n) — last element | O(log n) — max depth |

Search operations can behave differently based on the position of the target element and the search algorithm used. The **best-case scenario** occurs when the target is found immediately—for example, the first element in a linear search or the middle element in a binary search—resulting in **O(1)** time complexity. The **average-case scenario** considers the expected performance over various possible positions of the target element. For linear search, it averages to **O(n/2)** (still treated as O(n)), while for binary search, it is **O(log n)**. The **worst-case scenario** occurs when the target is either not present or found last in the search sequence. For linear search, the worst case is **O(n)**, and for binary search, it is still **O(log n)** due to halving the search space repeatedly. These cases help assess the reliability and efficiency of search methods under different conditions.

**2. Setup:**

**Product class**

**File Name: Product.java**

**package** myproject.search;

**public** **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;

}

@Override

**public** String toString() {

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

}

}

**3. Implementation:**

Created 3 classes named:

* Product.java
* Search.java
* SearchTest.java
* Search class – for linear and binary search algorithms

**File Name: Search.java**

**package** myproject.search;

**public** **class** Search {

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

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

**if** (product.productName.equalsIgnoreCase(searchName)) {

**return** product;

}

}

**return** **null**;

}

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

**int** left = 0, right = products.length - 1;

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

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

**int** cmp = products[mid].productName.compareToIgnoreCase(searchName);

**if** (cmp == 0) {

**return** products[mid];

} **else** **if** (cmp < 0) {

left = mid + 1;

} **else** {

right = mid - 1;

}

}

**return** **null**;

}

}

**File Name: SearchTest.java**

**package** myproject.search;

**import** java.util.Arrays;

**import** java.util.Comparator;

**import** java.util.Scanner;

**public** **class** SearchTest {

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

Scanner scanner = **new** Scanner(System.***in***);

Product[] products = {

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

**new** Product(102, "Shoes", "Footwear"),

**new** Product(103, "Mobile", "Electronics"),

**new** Product(104, "Chair", "Furniture"),

**new** Product(105, "Watch", "Accessories")

};

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

String target = scanner.nextLine();

Product res1 = Search.*linearSearch*(products, target);

System.***out***.println("Linear Search Result: " + (res1 != **null** ? res1 : "Not Found"));

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

Product res2 = Search.*binarySearch*(products, target);

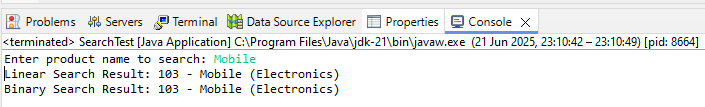
System.***out***.println("Binary Search Result: " + (res2 != **null** ? res2 : "Not Found"));

scanner.close();

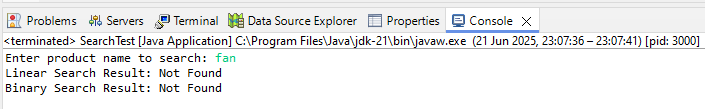
}

}

**Output: Found case:**



**Not Found Case:**



**5. Analysis:**

### **Time Complexity**

|  |  |  |
| --- | --- | --- |
| **Algorithm** | **Time Complexity** | **Sorted Input Required** |
| Linear Search | **O(n)** | No |
| Binary Search | **O(log n)** | Yes |

**Linear search** has a time complexity of **O(n)** in the worst and average cases, as it may require checking every element in the list. It is simple and works without requiring the data to be sorted. In contrast, **binary search** offers a much better time complexity of **O(log n)** for both average and worst-case scenarios, as it repeatedly divides the search space in half. However, binary search requires that the array be **sorted** beforehand. For small datasets or unsorted arrays, linear search may suffice, but for large, sorted datasets—like those found in e-commerce systems—binary search is significantly more efficient and scalable.