**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.

**1. Understand Asymptotic Notation**

* **Big O Notation**

Big O notation describes the upper bound of an algorithm's running time or space requirement in terms of input size n. It tells us how the performance (running time or memory usage) of an algorithm increases as the size of input grows, especially for large inputs.

It focuses on the worst-case scenario, helping developers understand how an algorithm scales.

| **Notation** | **Description** | **Example** |
| --- | --- | --- |
| O(1) | Constant time | Accessing array |
| O(log n) | Logarithmic time | Binary search |
| O(n) | Linear time | Linear search |
| O(n log n) | Log-linear time | Merge sort |
| O(n²) | Quadratic time | Bubble sort |

* **Best, Average, and Worst-Case Scenarios for Search**
* Linear Search (Unsorted):
* Best Case: O(1) – The element is found at the first position.
* Average Case: O(n) – On average, the element is found around the middle.
* Worst Case: O(n) – The element is at the last position or not present at all.
* Binary Search (Sorted):
* Best Case: O(1) – The element is exactly at the middle.
* Average Case: O(log n) – The search space is halved in each step.
* Worst Case: O(log n) – The element is not found but still requires log n comparisons.

**2. Setup**

* **Class Product :**

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;

}

}

**3. Implementation**

public class ProductSearch {

**// Linear Search by productId**

public static Product linearSearch(Product[] products, int targetId) {

for (Product product : products) {

if (product.productId == targetId) {

return product;

}

}

return null;

}

**// Binary Search by productId (array must be sorted)**

public static Product binarySearch(Product[] products, int targetId) {

int low = 0, high = products.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

if (products[mid].productId == targetId) return products[mid];

else if (products[mid].productId < targetId) low = mid + 1;

else high = mid - 1;

}

return null;

}

public static void main(String[] args) {

Product[] products = {

new Product(103, "Book", "Education"),

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

new Product(105, "Laptop", "Electronics"),

new Product(104, "T-Shirt", "Fashion"),

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

};

**// Linear Search**

System.out.println("Linear Search for productId = 104:");

Product result1 = linearSearch(products, 104);

**System.out.println(result1 != null ? result1 : "Product not found");**

**// Binary Search (after sorting)**

Arrays.sort(products, Comparator.comparingInt(p -> p.productId));

System.out.println("\nBinary Search for productId = 105:");

Product result2 = binarySearch(products, 105);

**System.out.println(result2 != null ? result2 : "Product not found");**

}

}

**Output**

Linear Search for productId = 104:

104: T-Shirt (Fashion)

Binary Search for productId = 105:

105: Laptop (Electronics)

**Note:**

* This solution searches products by “productid”, which is typically unique and numerically efficient for search operations. Binary Search is used after sorting the product array by “productid”, achieving O(log n) time complexity.
* The same approach can be adapted to search by “productName”, especially for

user-driven search experiences. However, from a backend perspective, using

“productid” is more efficient and reliable.

**4. Analysis**

* Linear Search has a time complexity of O(n). In the worst-case scenario, it may need to check every element in the array one by one until it finds the target or reaches the end.
* Binary Search has a time complexity of O(log n). It works by repeatedly dividing the sorted array in half and checking the middle element, significantly reducing the number of comparisons needed.

**Note:**

Binary Search only works on sorted arrays. While sorting initially takes O(n log n) time, this cost is justified when multiple searches are performed on the same sorted data, as each search then benefits from the faster O(log n) performance.

**Which Algorithm is More Suitable and Why ?**

Binary Search is more suitable for our platform, and here are the few reasons:

1. For large datasets (hundreds or thousands of products), binary search is significantly faster than linear search due to its logarithmic time complexity leading to high performance.
2. In a real e-commerce platform, the product list is often stored in sorted order (by productId or name), making binary search feasible and efficient.
3. Scalability - Binary search scales well with increasing data, which is crucial for an online store with a growing inventory.
4. Searching by productId ensures no duplicates and fast numeric comparison, making it ideal for backend systems.