**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 is used to describe the performance or complexity of an algorithm. It provides an upper bound on the time or space complexity of an algorithm in terms of the size of the input data, denoted as nnn. Big O notation helps in understanding the worst-case scenario for an algorithm's performance.

**Best, Average, and Worst-Case Scenarios:**

* **Best Case**: The scenario where the algorithm performs the minimum number of operations. For example, in a search algorithm, this would be finding the target element on the first try.
* **Average Case**: The scenario that represents the expected number of operations an algorithm will perform over a typical set of inputs.
* **Worst Case**: The scenario where the algorithm performs the maximum number of operations. For example, in a search algorithm, this would be having to search through the entire data set to find the target element or determining that it is not present

**SETUP AND IMPLEMENTATION:**

**LINEAR AND BINARY SEARCH:**

package searchingdsa;

import java.util.Arrays;

public class SearchingDSA {

public static void main(String[] args) {

Product[] products = {

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

new Product(2, "Smartphone", "Electronics"),

new Product(3, "Tablet", "Electronics"),

new Product(4, "Headphones", "Accessories"),

new Product(5, "Keyboard", "Accessories")

};

// Linear search example

int targetId = 3;

Product result = SearchAlgorithms.linearSearch(products, targetId);

System.out.println("Linear Search Result: " + result);

// Binary search example

Arrays.sort(products, (a, b) -> a.getProductId() - b.getProductId()); // Ensure array is sorted

result = SearchAlgorithms.binarySearch(products, targetId);

System.out.println("Binary Search Result: " + result);

}

}

class Product {

private int productId;

private String productName;

private String category;

public Product(int productId, String productName, String category) {

this.productId = productId;

this.productName = productName;

this.category = category;

}

public int 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 + '\'' +

'}';

}

}

class SearchAlgorithms {

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

for (Product product : products) {

if (product.getProductId() == targetId) {

return product;

}

}

return null;

}

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

int left = 0;

int right = products.length - 1;

while (left <= right) {

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

if (products[mid].getProductId() == targetId) {

return products[mid];

}

if (products[mid].getProductId() < targetId) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

}

**ANALYSIS**

**Time Complexity:**

* **Linear Search**: O(n)O(n)O(n)
  + Best Case: O(1)O(1)O(1) (if the element is found at the beginning)
  + Average Case: O(n)O(n)O(n)
  + Worst Case: O(n)O(n)O(n) (if the element is found at the end or not found at all)
* **Binary Search**: O(logn)O(log n)O(logn)
  + Best Case: O(1)O(1)O(1) (if the element is found at the middle)
  + Average Case: O(loGn)O(\log n)O(logn)
  + Worst Case: O(loGn)O(Log n)O(logn)