**DATA STRUCTURES AND ALGORITHMS**

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

Big O notation describes the upper bound of an algorithm's time or space complexity in terms of input size n. It tells us how an algorithm scales as the input grows.

It helps compare algorithms independently of hardware and input specifics, giving a mathematical basis for optimization.

The Best, Average and Worst Case scenarios for the search feature are:

**BEST CASE:** O(1) FOR LINEAR SEARCH (WHEN THE ELEMENT IS AT THE BEGINNING) AND BINARY SEARCH (WHEN ELEMENT IS AT THE MIDDLE)

**AVERAGE CASE:** O(N) FOR LINEAR SEARCH (WHEN THE ELEMENT IS AT THE MIDDLE) AND O(log N) FOR BINARY SEARCH (DIVIDE THE ARRAY EVERYTIME)

**WORST CASE:** O(N) FOR LINEAR SEARCH AND O(log N) FOR BINARY SEARCH (IF ELEMENT IS NOT FOUND)

**PRODUCT CLASS :**

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 + "]";

}

}

**LINEAR SEARCH (FOR UNSORTED PRODUCTS):**

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

for (Product p : products) {

if (p.productId == targetId) { // ITERATE THROUGH THE WHOLE ARRAY TO FIND THE PRODUCT

return p;

}

}

return null; // IF PRODUCT IS NOT FOUND

}

**BINARY SEARCH (FOR PRODUCTS SORTED BY productId):**

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

int left = 0;

int right = products.length - 1;

while (left <= right) {

int mid = left + right / 2;

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

return products[mid]; // RETURN IF THE MIDDLE ONE IS THE PRODUCT TO BE FOUND.

} else if (products[mid].productId < targetId) {

left = mid + 1; // FIND IN THE RIGHT HALF

} else {

right = mid - 1; // FIND IN THE LEFT HALF

}

}

return null; // IF PRODUCT IS NOT FOUND

}

// Sort array before binary search

public static void sortProductsById(Product[] products) {

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

}

**TEST CLASS:**

public class Main {

public static void main(String[] args) {

Product[] products = {

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

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

new Product(105, "Notebook", "Stationery"),

new Product(102, "Pen", "Stationery"),

new Product(104, "Monitor", "Electronics")

};

// Linear Search Test

System.out.println("Linear Search for ID 105:");

Product result1 = SearchOperations.linearSearch(products, 105);

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

// Sort before Binary Search

SearchOperations.sortProductsById(products);

// Binary Search Test

System.out.println("Binary Search for ID 105:");

Product result2 = SearchOperations.binarySearch(products, 105);

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

}

}



**THE BEST, AVERAGE AND WORST CASE TIME COMPLEXITIES FOR BOTH LINEAR AND BINARY SEARCH ARE DISCUSSED ABOVE.**

**BINARY SEARCH WILL BE EFFICIENT AS THE DATASET IS TOO LARGE.**