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

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

Big O notation is a mathematical way to describe the efficiency of an algorithm in terms of time or space required, based on the input size n. It expresses the upper bound of an algorithm’s growth rate, allowing us to understand how the algorithm scales as the size of the input increases.

Big O helps us:

* Compare different algorithms based on performance.
* Predict execution time for large inputs.
* Choose the most optimal solution in terms of speed and memory usage.

For example:

* O(1) – constant time (fastest and independent of input size).
* O(n) – linear time (slower as input grows).
* O(log n) – logarithmic time (very efficient for large data sets).

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

In search operations (like finding a product in a list or array), the performance varies depending on where the target is located (or not found at all). Here's how:

* Best Case:  
  The target element is found immediately (e.g., first element in linear search, middle element in binary search).

Linear Search: O(1),Binary Search: O(1)

* Average Case:  
  The target is somewhere in the middle. This is what usually happens in practice.

Linear Search: O(n),Binary Search: O(log n)

* Worst Case:  
  The element is not in the array or is at the last position (for linear), or it's in the smallest partition after many comparisons (for binary).

Linear Search: O(n),Binary Search: O(log n)

**Code:**

**Product Class: -**

public 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 void setProductId(int productId) {  
 this.productId = productId;  
 }  
  
 public String getProductName() {  
 return productName;  
 }  
  
 public void setProductName(String productName) {  
 this.productName = productName;  
 }  
  
 public String getCategory() {  
 return category;  
 }  
  
 public void setCategory(String category) {  
 this.category = category;  
 }  
}

**Search Class: -**

import java.util.Arrays;  
import java.util.Comparator;  
  
public class Search {  
 public static Product linearSearch(Product[] products,int targetId){  
 for(Product p:products){  
 if(p.getProductId()==targetId){  
 return p;  
 }  
 }  
 return null;  
 }  
 public static Product binarySearch(Product[] products,int targetId){  
 int low=0,high=products.length-1;  
 while(low<=high){  
 int mid=(low+high)/2;  
 int id=products[mid].getProductId();  
 if(id==targetId) return products[mid];  
 else if(id<targetId) low=mid+1;  
 else high=mid-1;  
 }  
 return null;  
 }  
 public static void sortProductsById(Product[] products){  
 Arrays.*sort*(products,Comparator.*comparingInt*(Product::getProductId));  
 }  
}

**Main Class: -**

public class Main {  
 public static void main(String[] args){  
 Product[] products ={  
 new Product(105,"Camera","Photography"),  
 new Product(101,"Laptop","Electronics"),  
 new Product(104,"Table","Furniture"),  
 new Product(102,"Mobile","Electronics"),  
 new Product(103,"Chair","Furniture")  
 };  
 Product linearSearchAnswer=Search.*linearSearch*(products,104);  
 if(linearSearchAnswer==null) System.*out*.println("No such product found");  
 else{  
 System.*out*.println("<-----Product Found(by Linear Search)----->");  
 System.*out*.println(linearSearchAnswer.getProductId()+"."+ linearSearchAnswer.getProductName()+"("+linearSearchAnswer.getCategory()+")");  
 }  
 System.*out*.println();  
 Search.*sortProductsById*(products);  
 Product binarySearchAnswer=Search.*binarySearch*(products,102);  
 if(binarySearchAnswer==null) System.*out*.println("No such product found");  
 else{  
 System.*out*.println("<-----Product Found(by Binary Search)----->");  
 System.*out*.println(binarySearchAnswer.getProductId()+"."+ binarySearchAnswer.getProductName()+"("+binarySearchAnswer.getCategory()+")");  
 }  
 }  
}

**Output:**

**A screenshot of a computer program

AI-generated content may be incorrect.**

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

1.Linear Search:

* Best Case: O(1), which occurs when the target element is found at the very beginning of the array.
* Average Case: O(n/2), which is approximately O(n), assuming the target could be anywhere in the list.
* Worst Case: O(n), which happens when the element is at the end of the array or not present at all.

2. Binary Search:

* Best Case: O(1), which happens when the target element is exactly at the middle of the array.
* Average Case: O(log n), since the algorithm divides the search space in half with each step.
* Worst Case: O(log n), occurring when the target is in one of the smallest remaining partitions or not present at all.

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

For an e-commerce platform with a potentially large number of products, binary search is more suitable due to its significantly better performance with large datasets. With a time complexity of O(log n), binary search is much faster than linear search’s O(n), especially as the inventory grows.

However, binary search requires that the product list be sorted by the search key (e.g., productId). This small overhead is acceptable in exchange for the much faster search performance.

Therefore, binary search is the preferred algorithm for the platform, provided that the products are stored in a sorted array or collection. If sorting is not feasible or the dataset is small and unsorted, linear search can be used as a simple alternative.