## **WEEK 1 Mandatory Hands On**

## Algorithms\_Data Structures

## Question:

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:

o Explain Big O notation and how it helps in analyzing algorithms.

o Describe the best, average, and worst-case scenarios for search operations.

2. Setup:

o Create a class Product with attributes for searching, such as productId, productName, and category.

3. Implementation:

o Implement linear search and binary search algorithms.

o Store products in an array for linear search and a sorted array for binary search.

4. Analysis:

o Compare the time complexity of linear and binary search algorithms.

o Discuss which algorithm is more suitable for your platform and why.

## Solution:

## Big O notation is a mathematical representation used to describe the efficiency of an algorithm — specifically, its time complexity and space complexity — in terms of how the algorithm performs as the input size n grows towards infinity.

## Rather than giving exact timings (which can vary based on hardware, programming language, etc.), Big O focuses on the growth rate of an algorithm’s operations as a function of input size.

Given two functions**f(n)** and **g(n)**, we say that**f(n)** is**O(g(n))** if there exist constants**c > 0** and **n0 >=**0 such that**f(n) <= c\*g(n)** for all **n >= n0**.

In simpler terms,**f(n)** is **O(g(n))** if**f(n)** grows no faster than**c\*g(n)** for all n >= n0 where c and n0 are constants.

**Best, Average, and Worst-Case Scenarios for Search Operations**

When analyzing **search algorithms**, it’s not enough to know just their general time complexity. We must also understand how they behave in different **input conditions** — namely, the **best**, **average**, and **worst-case scenarios**.

| **Scenario** | **Meaning** |
| --- | --- |
| **Best Case** | The most optimal input — the algorithm finishes quickly. |
| **Average Case** | The expected time taken across all possible inputs (statistical mean). |
| **Worst Case** | The most time-consuming input — maximum effort required by the algorithm. |

Each case tells us something different:

* Best case = "How fast can it go?"
* Average case = "What usually happens?"
* Worst case = "What’s the worst that could happen?"

**Linear Search**: Searches each element one by one in an **unsorted array**.

| **Scenario** | **Time Complexity** | **Explanation** |
| --- | --- | --- |
| **Best Case** | O(1) | The target element is the **first item** in the array. |
| **Average Case** | O(n) | The target is **somewhere in the middle** of the array. |
| **Worst Case** | O(n) | The target is the **last item**, or **not present at all**. |

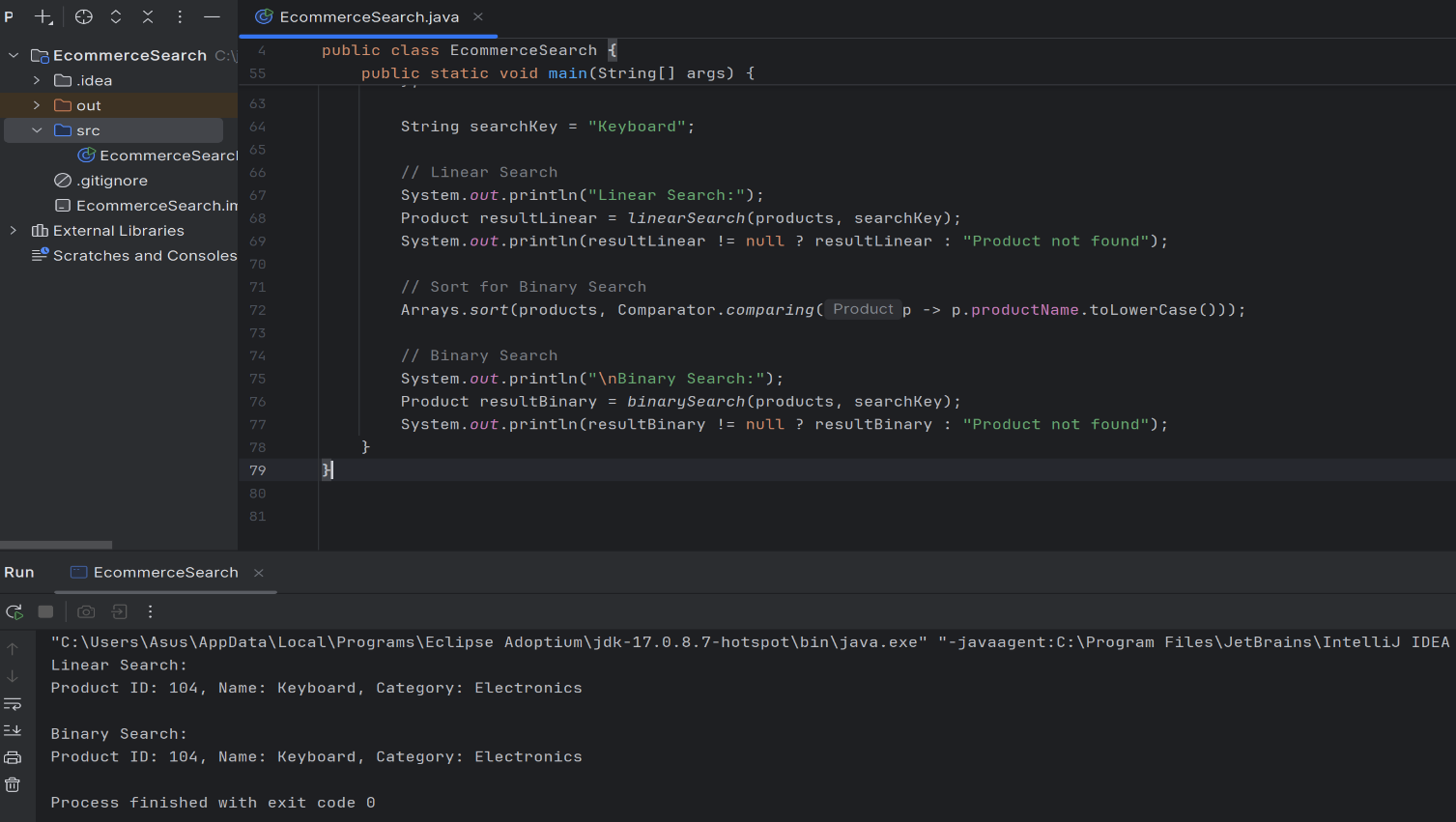
**Binary Search**: Searches by repeatedly dividing a **sorted array** in half.

| **Scenario** | **Time Complexity** |  | **Explanation** |
| --- | --- | --- | --- |
| **Best Case** | O(1) |  | The middle element is the target. |
| **Average Case** | O(log n) |  | Logarithmic halving — on average, log₂(n) comparisons. |
| **Worst Case** | O(log n) |  | Target is in the last position to be searched. |

CODE: EcommerceSearch.java

**import java.util.Arrays;  
import java.util.Comparator;  
  
public class EcommerceSearch {  
  
 // Product class definition  
 static 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 "Product ID: " + productId + ", Name: " + productName + ", Category: " + category;  
 }  
 }  
  
 // Linear Search implementation  
 public static Product linearSearch(Product[] products, String targetName) {  
 for (Product p : products) {  
 if (p.productName.equalsIgnoreCase(targetName)) {  
 return p;  
 }  
 }  
 return null;  
 }  
  
 // Binary Search implementation  
 public static Product binarySearch(Product[] products, String targetName) {  
 int left = 0, right = products.length - 1;  
  
 while (left <= right) {  
 int mid = (left + right) / 2;  
 int cmp = products[mid].productName.compareToIgnoreCase(targetName);  
  
 if (cmp == 0) {  
 return products[mid]; // Found  
 } else if (cmp < 0) {  
 left = mid + 1; // Search right half  
 } else {  
 right = mid - 1; // Search left half  
 }  
 }  
  
 return null; // Not found  
 }  
  
 // Main method to test both searches  
 public static void main(String[] args) {  
 Product[] products = {  
 new Product(101, "Laptop", "Electronics"),  
 new Product(102, "Shoes", "Footwear"),  
 new Product(103, "Mug", "Kitchen"),  
 new Product(104, "Keyboard", "Electronics"),  
 new Product(105, "T-shirt", "Clothing")  
 };  
  
 String searchKey = "Keyboard";  
  
 // Linear Search  
 System.*out*.println("Linear Search:");  
 Product resultLinear = *linearSearch*(products, searchKey);  
 System.*out*.println(resultLinear != null ? resultLinear : "Product not found");  
  
 // Sort for Binary Search  
 Arrays.*sort*(products, Comparator.*comparing*(p -> p.productName.toLowerCase()));  
  
 // Binary Search  
 System.*out*.println("\nBinary Search:");  
 Product resultBinary = *binarySearch*(products, searchKey);  
 System.*out*.println(resultBinary != null ? resultBinary : "Product not found");  
 }  
}**

OUTPUT:



**Comparing Linear vs Binary Search**

Let’s say we are searching for a product in a list of n items.

* Linear Search:
  + Worst-case: We check all items ⇒ O(n)
  + As n doubles, the time also doubles.
* Binary Search:
  + Requires sorted data.
  + Divides the list in half each step ⇒ O(log n)
  + As n doubles, steps increase only slightly.

| **n** | **Linear Search Steps** | **Binary Search Steps** |
| --- | --- | --- |
| 10 | 10 | 4 |
| 100 | 100 | 7 |
| 1,000,000 | 1,000,000 | ~20 |

**Binary Search** is better for my platform because the search needs to be optimized for fast performance and binary search is:

Faster on Large Datasets:  
Binary search operates in **O(log n)** time, making it significantly faster than linear search for large product catalogs.

Predictable Performance:  
Binary search offers **consistent, predictable performance**, even in the worst-case scenario.

User Experience:  
A fast search leads to **lower bounce rates** and **better user satisfaction**.