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

**Scenario:**

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.

**SOLUTION**

**Step-1:**  
**What is Big O Notation?**

Big O notation is a way to describe **how fast or slow an algorithm grows** as the size of the input increases.

Imagine you're searching for a product in a list. If the list has 10 items, it’s easy. But what if it has 1,000 or 1 million? Big O helps us **measure the time or space** an algorithm will take, not in actual seconds or memory, but in terms of **how it grows** as input gets bigger.

So, Big O doesn’t tell you **exactly how long** something takes. Instead, it tells you **how the time grows** as the amount of data grows.

For example:

* If a search takes time directly based on the number of items, we call it **O(n)** (linear time).
* If it divides the work in half each time (like binary search), it’s **O(log n)** (logarithmic time).

**🔹 Why is Big O Important?**

Let’s say your app works fine now, but as your customer base grows, suddenly it gets **slow** or even **crashes**. If you had analyzed the algorithm with Big O, you could have chosen a more efficient one early on.

Big O helps you:

* Compare two different algorithms and choose the faster one.
* Predict performance for **larger inputs**.
* Identify bottlenecks in your program.

**🔹 Best, Average, and Worst-Case Scenarios (for Search)**

When you talk about search algorithms like **linear search** or **binary search**, their behavior can change based on **where the item is found** (or if it isn’t found at all). That’s where best, average, and worst-case come in:

**🔸 Best Case:**

This is the **ideal** situation — the item is found immediately.

* Example: In linear search, if the item you're looking for is the **first one**, it only takes **1 check**.
* In Big O, linear search best case is **O(1)** (constant time).

**🔸 Average Case:**

This is the **most common** situation — the item is somewhere in the middle.

* In linear search, if you randomly search through a list, on average, it might take **half the list size** to find the item.
* Still considered **O(n)**, because in Big O we focus on **how it grows**, not actual counts.

**🔸 Worst Case:**

This is the **most time-consuming** case — either the item is at the **end**, or it’s **not there at all**.

* For linear search, this means checking **every single item**.
* So its worst case is **O(n)**.

For binary search:

* Best case is **O(1)** (if the item is at the center).
* Worst case is **O(log n)** — it cuts the list in half every time.

**Step-2:**

**Product.java**

public 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 + ")";

}

}

**Step-3:**

**LinearSearch.java**

public class LinearSearch {

public static Product linearSearch(Product[] products, String name) {

for (Product p : products) {

if (p.productName.equalsIgnoreCase(name)) {

return p;

}

}

return null;

}

}

**BinarySearch.java**

import java.util.Arrays;

import java.util.Comparator;

public class BinarySearch {

public static Product binarySearch(Product[] products, String name) {

int left = 0, right = products.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

int cmp = products[mid].productName.compareToIgnoreCase(name);

if (cmp == 0) return products[mid];

else if (cmp < 0) left = mid + 1;

else right = mid - 1;

}

return null;

}

public static void sortProductsByName(Product[] products) {

Arrays.sort(products, Comparator.comparing(p -> p.productName.toLowerCase()));

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

Product[] products = {

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

new Product(102, "T-shirt", "Clothing"),

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

new Product(104, "Shoes", "Footwear"),

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

};

// Linear Search

Product foundLinear = LinearSearch.linearSearch(products, "Shoes");

System.out.println("Linear Search Result: " + (foundLinear != null ? foundLinear : "Not Found"));

// Binary Search

BinarySearch.sortProductsByName(products);

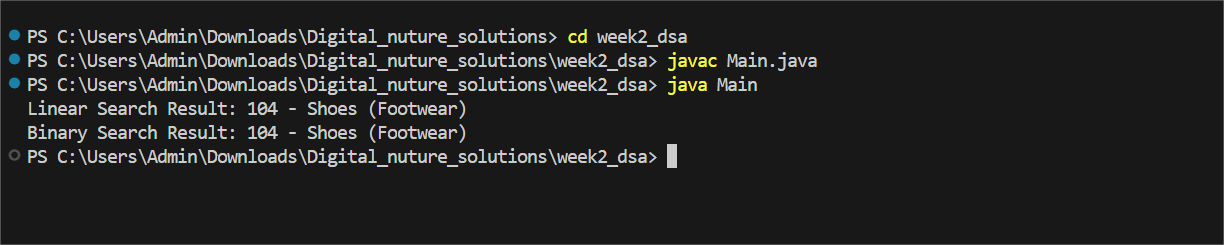
Product foundBinary = BinarySearch.binarySearch(products, "Shoes");

System.out.println("Binary Search Result: " + (foundBinary != null ? foundBinary : "Not Found"));

}

}

**OUTPUT:**



Step-4:

**Linear Search**

* **How it works:** Goes through the list one by one, checking each product until it finds a match.
* **Time Complexity:**
  + **Best case:** O(1) – Item is the first one in the list.
  + **Average case:** O(n/2) ≈ O(n) – Item is somewhere in the middle.
  + **Worst case:** O(n) – Item is last or not in the list.
* **Space Complexity:** O(1) – It doesn’t use extra memory.

**Binary Search**

* **How it works:** Works only on **sorted** data. It checks the middle element and eliminates half of the list in each step.
* **Time Complexity:**
  + **Best case:** O(1) – Item is exactly at the middle.
  + **Average and Worst case:** O(log n) – List is divided in half repeatedly until the item is found or not found.
* **Space Complexity:** O(1) if iterative; O(log n) if implemented recursively.

**Which Is More Suitable for an E-Commerce Platform?**

**Binary Search is more suitable. Here’s why:**

* **Performance Matters**: E-commerce platforms handle **thousands or millions of products**. You can’t afford to go through every item one by one (as linear search does). Binary search is much faster because it **cuts the search space in half** each time.
* **Sorted Data is Manageable**: Most product databases are **already sorted or indexed** by name, category, or ID. This makes binary search a natural fit.

**Scalability**: As the number of products grows, binary search scales far better. For example : (i)Searching 1,000,000 products with linear search might take up to 1,000,000 steps. (ii) With binary search, it takes only about 20 steps (log2(1,000,000) ≈ 20).

* **User Experience**: Fast search means **instant results**. In a real-world app, you might use even faster structures (like hash maps or tries), but for basic search, binary search is the best choice over linear.