**E-commerce Platform Search Function**

**Big O Notation and Its Importance**

Big O Notation is a mathematical way to describe the **efficiency of an algorithm** in terms of **time or space**, relative to the size of the input n. It helps developers understand how an algorithm performs when the size of input grows — regardless of machine speed or programming language.

**Why it matters:**

* Helps **compare algorithms** efficiently.
* Focuses on the **growth rate** of operations.
* Highlights the **scalability** of code.
* Guides in choosing the **most optimized approach** for large datasets.

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

**Linear Search (Unsorted array)**

* **Best Case**: O(1) – target is at the first index.
* **Average Case**: O(n/2) ≈ O(n) – target is in the middle.
* **Worst Case**: O(n) – target is at the end or not found.

**Binary Search (Sorted array)**

* **Best Case**: O(1) – target is at the middle.
* **Average Case**: O(log n) – logarithmic comparisons.
* **Worst Case**: O(log n) – target not found after all divisions.

**Code:**

**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;

}

}

**SearchService.java**

import java.util.Arrays;

import java.util.Comparator;

public class SearchService {

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

for (Product p : products) {

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

return p;

}

}

return null;

}

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

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

int low = 0, high = products.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

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

if (cmp == 0) {

return products[mid];

} else if (cmp < 0) {

high = mid - 1;

} else {

low = mid + 1;

}

}

return null;

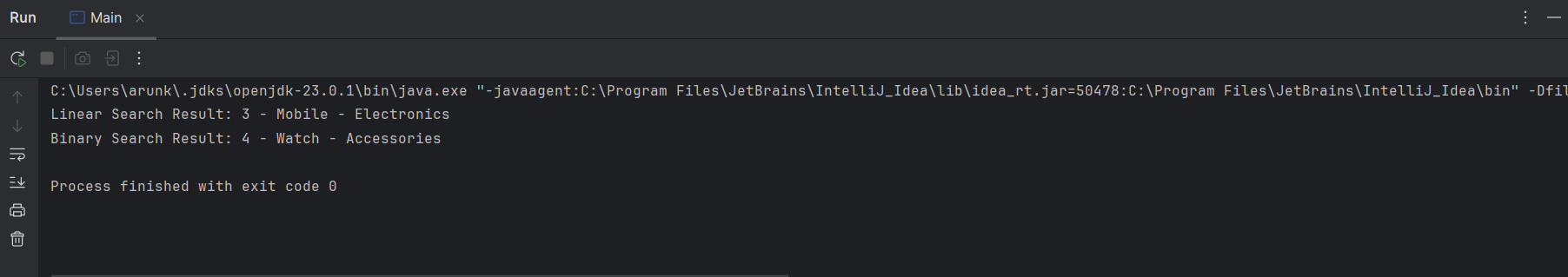
}

}

**Main.java**

public class Main {  
 public static void main(String[] args) {  
 Product[] products = {  
 new Product(1, "Laptop", "Electronics"),  
 new Product(2, "Shoes", "Footwear"),  
 new Product(3, "Mobile", "Electronics"),  
 new Product(4, "Watch", "Accessories")  
 };  
  
 Product linearResult = SearchService.*linearSearch*(products, "Mobile");  
 System.*out*.println("Linear Search Result: " + (linearResult != null ? linearResult : "Not Found"));  
  
 Product binaryResult = SearchService.*binarySearch*(products, "Watch");  
 System.*out*.println("Binary Search Result: " + (binaryResult != null ? binaryResult : "Not Found"));  
 }  
}

**Output:**

****

**4. Analysis**

**Time Complexity Comparison**

| **Search Method** | **Best Case** | **Average Case** | **Worst Case** | **Needs Sorting** |
| --- | --- | --- | --- | --- |
| Linear Search | O(1) | O(n) | O(n) | No |
| Binary Search | O(1) | O(log n) | O(log n) | Yes |

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

**Binary Search** is more suitable for **fast and scalable search** if the dataset is **already sorted** and does not change frequently.

* It offers **logarithmic performance**, making it ideal for large product catalogs.
* However, it requires the list to be **sorted**, which adds overhead if products are frequently added or updated.

**Linear Search** is more flexible for **small, dynamic datasets** where sorting is not practical, but it becomes inefficient as the number of products grows.

**Final Recommendation:**

* Use **binary search** for **static or infrequently updated, sorted product lists**.
* In real-world systems, for fast and flexible search, consider:
  + **HashMaps** for exact match lookups (O(1) time).