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

**What is Big O Notation?**

Big O notation describes **how the time or space complexity of an algorithm grows** as the input size increases. It ignores constant factors and focuses on the dominant term affecting performance.  
It helps developers:

* Analyze algorithms independently of hardware.
* Compare different approaches for efficiency.
* Predict scalability as data size increases.

**Best, Average, Worst Cases for Search**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Best Case** | **Average Case** | **Worst Case** |
| **Linear Search** | O(1) (first element) | O(n) | O(n) |
| **Binary Search** | O(1) (middle element) | O(log n) | O(log n) |

* **Linear Search:** Simple but slower for large datasets.
* **Binary Search:** Faster but requires the data to be **sorted**.

**Product Class**

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;

}

}

**LinearSearch.java**

public class LinearSearch {

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

for (int i = 0; i < products.length; i++) {

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

return i;

}

}

return -1; // Not found

}

}

**BinarySearch.java**

import java.util.Arrays;

import java.util.Comparator;

public class BinarySearch {

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

int left = 0;

int right = products.length - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

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

return mid;

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

left = mid + 1;

} else {

right = mid - 1;

}

}

return -1; // Not found

}

public static void sortProducts(Product[] products) {

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

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

Product[] products = {

new Product(3, "Keyboard", "Electronics"),

new Product(1, "Laptop", "Electronics"),

new Product(2, "Mouse", "Electronics"),

new Product(5, "T-Shirt", "Apparel"),

new Product(4, "Shoes", "Footwear")

};

int targetId = 5;

int linearIndex = LinearSearch.linearSearch(products, targetId);

if (linearIndex != -1) {

System.out.println("Linear Search: Found " + products[linearIndex].productName);

} else {

System.out.println("Linear Search: Product not found.");

}

BinarySearch.sortProducts(products);

int binaryIndex = BinarySearch.binarySearch(products, targetId);

if (binaryIndex != -1) {

System.out.println("Binary Search: Found " + products[binaryIndex].productName);

} else {

System.out.println("Binary Search: Product not found.");

}

}

}

**Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| **Search Method** | **Time Complexity** | **Pros** | **Cons** |
| **Linear Search** | O(n) | Easy to implement, works on **unsorted data** | Slow for large lists |
| **Binary Search** | O(log n) | Much faster for large data | Needs **sorted data** |

**Which is more suitable?**

For **large product lists**, **binary search** is more efficient because of its logarithmic time complexity — but only works if your product list is sorted by the key you’re searching for (e.g., productId).

**Expected Output**

