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

**ProductSearchSystem.java**

**package** mypackage;

**import** java.util.\*;

**public** **class** ProductSearchSystem {

// Step 2: Product class

**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;

}

**public** String toString() {

**return** "Product ID: " + productId + ", Name: " + productName + ", Category: " + category;

}

}

// Linear Search (unsorted array)

**public** **static** **void** linearSearch(Product[] products, **int** searchId) {

**boolean** found = **false**;

**for** (Product p : products) {

**if** (p.productId == searchId) {

System.***out***.println("Product found (Linear Search): " + p);

found = **true**;

**break**;

}

}

**if** (!found) {

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

}

}

// Binary Search (sorted array)

**public** **static** **void** binarySearch(Product[] products, **int** searchId) {

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

**boolean** found = **false**;

**while** (left <= right) {

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

**if** (products[mid].productId == searchId) {

System.***out***.println("Product found (Binary Search): " + products[mid]);

found = **true**;

**break**;

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

left = mid + 1;

} **else** {

right = mid - 1;

}

}

**if** (!found) {

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

}

}

**public** **static** **void** main(String[] args) {

Scanner sc = **new** Scanner(System.***in***);

// Step 2: Setup

System.***out***.print("Enter number of products: ");

**int** n = sc.nextInt();

sc.nextLine();

Product[] products = **new** Product[n];

**for** (**int** i = 0; i < n; i++) {

System.***out***.println("Enter details for Product " + (i+1));

System.***out***.print("Product ID: ");

**int** id = sc.nextInt();

sc.nextLine();

System.***out***.print("Product Name: ");

String name = sc.nextLine();

System.***out***.print("Category: ");

String category = sc.nextLine();

products[i] = **new** Product(id, name, category);

}

// Sorting the array for binary search

Arrays.*sort*(products, (a, b) -> a.productId - b.productId);

System.***out***.print("\nEnter Product ID to search: ");

**int** searchId = sc.nextInt();

// Step 3: Implementation

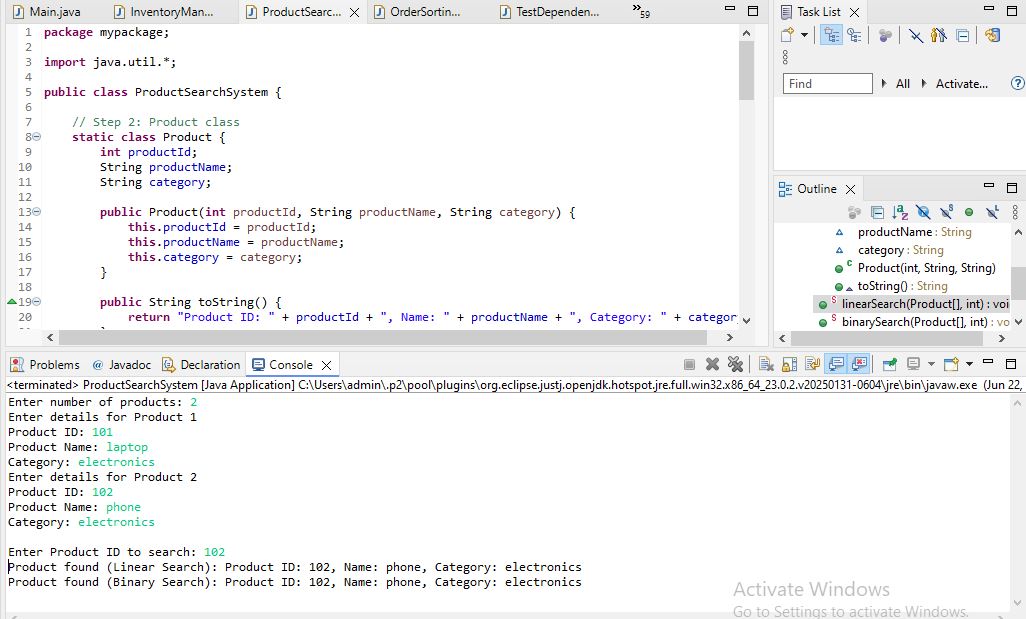
*linearSearch*(products, searchId);

*binarySearch*(products, searchId);

sc.close();

}

}



**1. Understand Asymptotic Notation**

**➔ What is Big O Notation?**

Big O notation is a mathematical notation that describes how an algorithm's **time or space complexity grows** as the input size increases. It focuses on:

* **Scalability**: How efficiently an algorithm handles large data.
* **Machine-independent**: Ignores hardware, language, or compiler-specific details.
* **Worst-case behavior**: Usually used to analyze the maximum time an algorithm can take.

**Example:**

* O(1) — constant time (very fast)
* O(n) — linear time (slower as data increases)
* O(log n) — logarithmic time (efficient even for large data)
* O(n²) — quadratic time (very slow for large data)

**➔ Why is it useful?**

* Helps us **compare algorithms** objectively.
* Helps select the most **efficient algorithm** for large datasets.
* Especially useful in search problems, sorting, and optimization tasks.

**➔ Best, Average, Worst-case for search operations**

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| **Linear Search** | O(1) (first element match) | O(n/2) ≈ O(n) | O(n) (last element or not found) |
| **Binary Search** | O(1) (middle element match) | O(log n) | O(log n) (worst case: many divisions) |

**2. Setup**

We define a class Product with the following attributes:

* productId (integer): unique identifier for each product.
* productName (String): name of the product.
* category (String): category to which the product belongs.

This structure allows us to search products based on their ID.

**3. Implementation**

**➔ Linear Search**

* The linear search algorithm scans the array **from start to end**.
* Compares each product's productId with the target productId.
* If found, returns the product; if not found, returns not found after checking all items.
* **Does not require sorting**.

**➔ Binary Search**

* Binary search requires the array to be **sorted by productId**.
* It divides the search space into halves:
  + If the middle element matches, we return it.
  + If the target is smaller, search in the left half.
  + If the target is larger, search in the right half.
* Much faster for large datasets compared to linear search.

**4. Analysis**

**➔ Time Complexity Comparison**

| **Algorithm** | **Time Complexity** |
| --- | --- |
| **Linear Search** | O(n) |
| **Binary Search** | O(log n) |

* Linear search checks each element one by one, making it inefficient for large data.
* Binary search significantly reduces search time by dividing the problem size repeatedly.

**➔ Which algorithm is more suitable for your platform?**

* In **small datasets**, both algorithms perform well.
* In **large-scale e-commerce platforms**, where millions of products exist:
  + Binary search is much more efficient (O(log n)).
  + However, binary search needs the data to be **sorted**.
  + In real-world e-commerce platforms, search is even further optimized using:
    - **Indexing** in databases.
    - **Search engines** like Elasticsearch.
    - **Hashing** or **B-trees** for even faster search.