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

**Code**

**Project.java**

package com.example.search;

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;

}

public String toString() {

return productId + " - " + productName + " (" + category + ")";

}

}

**SearchTest.java**

package com.example.search;

import java.util.Arrays;

import java.util.Comparator;

public class SearchTest {

public static void main(String[] args) {

Product[] products = {

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

new Product(102, "Shoes", "Footwear"),

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

new Product(104, "T-Shirt", "Clothing"),

new Product(105, "Book", "Stationery")

};

String searchKey = "Phone";

Product result1 = SearchUtility.*linearSearch*(products, searchKey);

System.*out*.println("Linear Search Result: " + (result1 != null ? result1 : "Not Found"));

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

Product result2 = SearchUtility.*binarySearch*(products, searchKey);

System.*out*.println("Binary Search Result: " + (result2 != null ? result2 : "Not Found"));

}

}

**SearchUtility.java**

package com.example.search;

public class SearchUtility {

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

for (Product product : products) {

if (product.productName.equalsIgnoreCase(key)) {

return product;

}

}

return null;

}

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

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

while (low <= high) {

int mid = (low + high) / 2;

int compare = products[mid].productName.compareToIgnoreCase(key);

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

else if (compare < 0) low = mid + 1;

else high = mid - 1;

}

return null;

}

}

**Analysis**

In the context of an e-commerce platform, implementing efficient search functionality is crucial for performance and user experience. To evaluate the efficiency of search algorithms, we use Big O notation, which describes the upper bound of an algorithm’s running time as the input size increases. It helps in understanding how scalable and efficient an algorithm is under different conditions.

For linear search, the best-case time complexity is O(1) when the element is at the beginning, average-case is O(n/2), and the worst-case is O(n), where *n* is the number of products. It is simple and works on unsorted data, but it becomes inefficient as the dataset grows. Binary search, on the other hand, is much faster with a time complexity of O(log n) for average and worst cases, but it requires the array to be sorted beforehand. While binary search adds the overhead of sorting, it offers significantly better performance for large datasets. For an e-commerce platform dealing with thousands of products and requiring frequent searches, binary search is more suitable, especially when used in conjunction with pre-sorted data or efficient data structures like trees or indexes. Overall, binary search scales better and is the preferred choice for real-time search operations in large-scale applications.