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

**Scenario:**

You are 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 :**

**Product.java**  
public class Product {

private int productId;

private String productName;

private String category;

public Product(int productId, String productName, String category) {

this.productId = productId;

this.productName = productName;

this.category = category;

}

public int getProductId() {

return productId;

}

@Override

public String toString() {

return "Product{id=" + productId + ", name='" + productName + "', category='" + category + "'}";

}

}

**SearchDemo.java**

public class SearchDemo {

// Linear Search: Works on any array (unsorted or sorted)

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

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

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

return i; // Found at index i

}

}

return -1; // Not found

}

// Binary Search: Requires a sorted array

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;

int midId = products[mid].getProductId();

if (midId == targetId) {

return mid; // Found at mid index

} else if (midId < targetId) {

left = mid + 1; // Search right half

} else {

right = mid - 1; // Search left half

}

}

return -1; // Not found

}

public static void main(String[] args) {

// Sample products (unsorted for linear search)

Product[] linearSearchProducts = {

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

new Product(203, "Blender", "Kitchen"),

new Product(145, "Smartphone", "Electronics"),

new Product(567, "Desk Chair", "Furniture")

};

// Sample products sorted by ID for binary search

Product[] binarySearchProducts = {

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

new Product(145, "Smartphone", "Electronics"),

new Product(203, "Blender", "Kitchen"),

new Product(567, "Desk Chair", "Furniture")

};

// Test searches

int targetId = 203;

// Linear Search

int linearResult = linearSearch(linearSearchProducts, targetId);

System.out.println("Linear Search Result: " +

(linearResult != -1 ? linearSearchProducts[linearResult] : "Product not found"));

// Binary Search

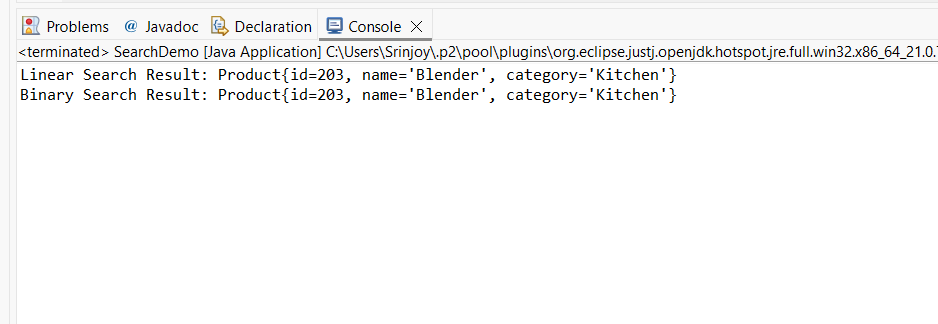
int binaryResult = binarySearch(binarySearchProducts, targetId);

System.out.println("Binary Search Result: " +

(binaryResult != -1 ? binarySearchProducts[binaryResult] : "Product not found"));

}

}

****

**Output :**

**Algorithm Analysis**

* **Time Complexity**:
  + **Linear Search**: O(n) in worst/average cases. Checks each element sequentially.
  + **Binary Search**: O(log n) in worst/average cases. Halves search space each iteration.
* **Suitability for E-commerce Platform**:
  + **Binary Search** is optimal for large catalogs due to logarithmic scalability. For example, searching 1 million products requires only ~20 comparisons vs. 500,000 on average with linear search.
  + **Precondition**: Products must be sorted by productId. This is feasible during data ingestion or updates using efficient sorting algorithms (O(n log n)), which is amortized over many searches.