**ALORITHMS DATASTRUCTURES**

**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.

**DESCRIPTION:**

This program demonstrates **linear** and **binary search** on an array of e-commerce products.  
It searches by **product name** using linear search and by **product ID** using binary search after sorting.

**PROGRAM:**

import java.util.Arrays;

import java.util.Comparator;

public class EcommerceSearchDemo {

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 "[" + productId + ", " + productName + ", " + category + "]";

}

}

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

for (Product p : products) {

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

return p;

}

}

return null;

}

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

int left = 0;

int right = products.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

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

return products[mid];

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

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

public static void main(String[] args) {

Product[] products = {

new Product(102, "Laptop", "Electronics"),

new Product(205, "Shoes", "Footwear"),

new Product(301, "Book", "Stationery"),

new Product(150, "Headphones", "Electronics")

};

System.out.println("Linear Search by Product Name:");

Product result1 = linearSearch(products, "Shoes");

System.out.println(result1 != null ? result1 : "Product not found");

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

System.out.println("\n Binary Search by Product ID:");

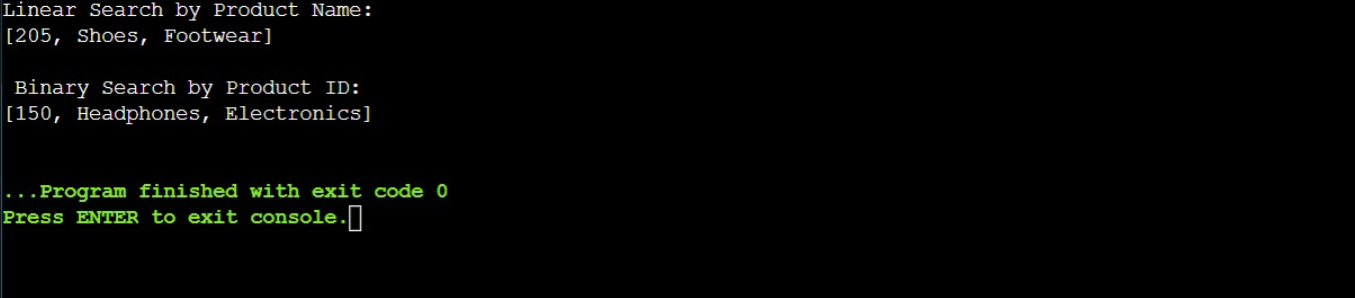
Product result2 = binarySearch(products, 150);

System.out.println(result2 != null ? result2 : "Product not found");

}

}

**OUTPUT:**

****

**ANALYSIS:**

The recursive algorithm decreases the years parameter by 1 in each call, resulting in n total recursive calls, where *n* is the number of years.  
Therefore, the time complexity is O(n).  
Since each recursive call adds a new frame to the call stack, the space complexity is also O(n).

**WAYS TO OPTIMIZE:**

Use iteration instead of recursion to eliminate call stack overhead.

Memoization is unnecessary here as there are no overlapping subproblems.