Exercise 2: E-commerce Platform Search Function

**Step-1: Understand Asymptotic Notation**

**Big O Notation**

Big O notation is a mathematical representation used to describe the time complexity or space complexity of an algorithm. It expresses the upper bound of the running time in terms of the input size *n*, allowing developers to evaluate the efficiency and scalability of algorithms regardless of hardware or implementation details.

**Common Examples:**

* **O(1) – Constant Time:**  
  The algorithm takes the same amount of time regardless of the input size.
* **O(n) – Linear Time:**  
  The algorithm’s execution time grows linearly with the input size.
* **O(log n) – Logarithmic Time:**  
  The time increases logarithmically as the input size increases. This is typical of divide-and-conquer algorithms like binary search.

Big O notation is essential for selecting the most efficient algorithm, particularly for large-scale applications where performance is critical, such as search functionality in an e-commerce platform.

**Best, Average, and Worst Case Scenarios for Search Operations**

When analyzing search operations, it is important to consider three different scenarios to fully understand an algorithm's behavior:

|  |  |
| --- | --- |
| **Case Type** | **DESCRIPTION** |
| **Best Case** | When the target is found immediately |
| **Average Case** | Typical number of operations on average |
| **Worst Case** | Maximum possible operations required |

These cases help in assessing how an algorithm will perform under different conditions and are directly linked to the Big O analysis.

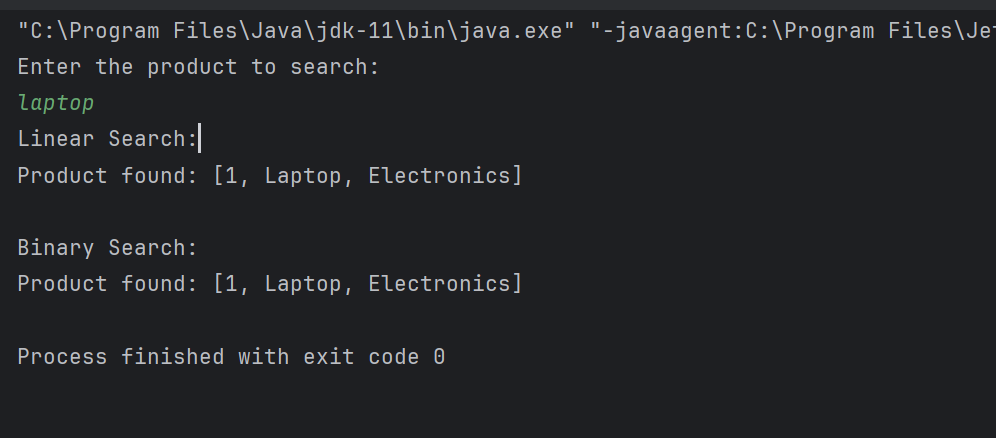
**Step-2: setup and Implementation**

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;  
 }  
 @Override  
 public String toString() {  
 return "[" + productId + ", " + productName + ", " + category + "]";  
 }  
 }

public class ProductSearch {  
 public static Product linearSearch(Product[] products, String targetName) {  
 for (Product product : products) {  
 if (product.productName.equalsIgnoreCase(targetName)) {  
 return product;  
 }  
 }  
 return null;  
 }  
 public static Product binarySearch(Product[] products, String targetName) {  
 int left = 0;  
 int right = products.length - 1;  
 while (left <= right) {  
 int mid = left + (right - left) / 2;  
 int comparison = products[mid].productName.compareToIgnoreCase(targetName);  
 if (comparison == 0) {  
 return products[mid];  
 } else if (comparison < 0) {  
 left = mid + 1;  
 } else {  
 right = mid - 1;  
 }  
 }  
 return null;  
 }  
}

import java.util.\*;  
public class Main {  
 public static void main(String[] args) {  
 Product[] products = {  
 new Product(1, "Laptop", "Electronics"),  
 new Product(2, "Shoes", "Footwear"),  
 new Product(3, "Watch", "Accessories"),  
 new Product(4, "Phone", "Electronics"),  
 new Product(5, "Backpack", "Travel"),  
 new Product(6, "Camera", "Electronics"),  
 new Product(7, "T-Shirt", "Clothing"),  
 new Product(8, "Keyboard", "Electronics"),  
 new Product(9, "Sunglasses", "Accessories"),  
 new Product(10, "Book", "Stationery"),  
 new Product(11, "Mouse", "Electronics"),  
 new Product(12, "Charger", "Electronics"),  
 new Product(13, "Water Bottle", "Travel"),  
 new Product(14, "Jacket", "Clothing"),  
 };  
 Scanner sc=new Scanner(System.*in*);  
 System.*out*.println("Enter the product to search: ");  
 String targetName = sc.nextLine();  
 System.*out*.println("Linear Search:");  
 Product linearResult = ProductSearch.*linearSearch*(products, targetName);  
 if (linearResult != null) {  
 System.*out*.println("Product found: " + linearResult);  
 } else {  
 System.*out*.println("Product not found.");  
 }  
 Arrays.*sort*(products, Comparator.*comparing*(p -> p.productName.toLowerCase()));  
 System.*out*.println("\nBinary Search:");  
 Product binaryResult = ProductSearch.*binarySearch*(products, targetName);  
 if (binaryResult != null) {  
 System.*out*.println("Product found: " + binaryResult);  
 } else {  
 System.*out*.println("Product not found.");  
 }  
 }  
}

**OUTPUT:**

****

**Step-3: Analysis**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Search Type** | **Best Case Time** | **Average Case Time** | **Worst Case Time** | **Data Requirement** |
| **Linear Search** | O(1) | O(n) | O(n) | Works on unsorted data |
| **Binary Search** | O(1) | O(log n) | O(log n) | Requires sorted data |

**Linear Search**

**Description:**  
Linear search checks each element in the list one by one until the target value is found or the end of the list is reached.

**Advantages:**

* Simple to implement.
* Does not require the array to be sorted.
* Works on all types of data structures.

**Disadvantages:**

* Inefficient for large datasets.
* Performance decreases linearly as the number of elements increases.

**Binary Search**

**Description:**  
Binary search operates on sorted arrays. It repeatedly divides the search interval in half and compares the target value to the middle element.

**Advantages:**

* Highly efficient for large datasets.
* Requires significantly fewer comparisons compared to linear search.

**Disadvantages:**

* Requires the data to be sorted beforehand.
* Slightly more complex to implement.

In an e-commerce platform where performance and speed are critical, especially as the product catalog grows, binary search is the preferred approach due to its logarithmic time complexity. Although it requires sorted data, the performance benefits far outweigh the overhead of sorting, particularly when searches are frequent. However, binary search would be more appropriate in a real-world application for its scalability and efficiency.