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

**1. Understand Asymptotic Notation:**

**a) Explain Big O notation and how it helps in analyzing algorithms.**

Big O Notation is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity.

In computer science, it's used to classify algorithms according to how their running time or space requirements grow as the input size grows. It provides an upper bound on the growth rate of an algorithm's complexity.

**How it helps in analyzing algorithms:**

* **Compares Algorithms**: It allows us to compare the efficiency of different algorithms independently of the hardware or specific implementation details. An algorithm with a better Big O complexity will generally perform better than one with a worse Big O complexity for large input sizes, even if the latter has a smaller constant factor.
* **Predicts Performance**: It helps predict how an algorithm will scale. If you know an algorithm is O(n), you can expect its running time to roughly double if the input size n doubles.
* **Identifies Bottlenecks**: By analyzing the Big O of different parts of a system, developers can identify which parts are likely to become performance bottlenecks as the data grows.

**Common Big O Complexities (from best to worst):**

* **O(1)** - Constant Time: The time taken is independent of the input size. (e.g., accessing an array element by index).
* **O(log n)** - Logarithmic Time: The time taken increases logarithmically with the input size. Often seen in algorithms that divide the problem space in half with each step (e.g., binary search).
* **O(n**) - Linear Time: The time taken grows linearly with the input size. (e.g., linear search).
* **O(n log n)** - Linearithmic Time: Common in efficient sorting algorithms (e.g., Merge Sort, Quick Sort).
* **O(n^2)** - Quadratic Time: The time taken grows quadratically with the input size. Often seen in algorithms with nested loops (e.g., bubble sort).
* **O(2^n)** - Exponential Time: The time taken doubles with each additional element. Highly inefficient for large inputs (e.g., some recursive algorithms without memoization).
* **O(n!)** - Factorial Time: The time taken grows extremely rapidly. Impractical for almost any useful input size (e.g., solving the traveling salesman problem with brute force).

**b) Describe the best, average, and worst-case scenarios for search operations.**

These scenarios describe the algorithm's performance under different conditions of input data. They are

* **Best-Case Scenario (Ω - Omega Notation):**
  + It describes the minimum possible time an algorithm could take to complete.
  + For a search operation: This occurs when the target element is found at the *very first position* the algorithm checks.
  + Example (Linear Search): The target element is the first element in the array.
  + Example (Binary Search): The target element is exactly the middle element on the first comparison.
* **Average-Case Scenario (Θ - Theta Notation):**
  + It describes the expected running time of an algorithm given a typical or random input. It's often the most practical measure as it reflects real-world performance.
  + **For a search operation**: This occurs when the target element is found somewhere in the *middle* of the data structure, or if the element is not present, the algorithm performs a typical number of comparisons.
  + **Example (Linear Search**): On average, the target element is found after checking about half of the elements.
  + **Example (Binary Search**): On average, the target element is found after log n comparisons, or it's determined to be absent after a similar number of comparisons.
* **Worst-Case Scenario (O - Big O Notation):**
  + It describes the maximum possible time an algorithm could take to complete. It provides an upper bound on performance and is often the most important for guarantees (e.g., ensuring a system doesn't bog down under specific conditions).
  + For a search operation: This occurs when the target element is found at the *last possible position* the algorithm checks, or when the element is *not present* in the data structure, requiring the algorithm to check all relevant possibilities.
  + Example (Linear Search): The target element is the last element in the array, or it is not present at all.
  + Example (Binary Search): The target element is the last element to be narrowed down, or it is not present, requiring log n comparisons to determine its absence.

**2. Setup:**

Create a class Product with attributes for searching, such as productId, productName, and category.

**Java code**

**Product.java**

package Ecommerce\_platform\_search;  
  
public class Product implements Comparable<Product> {  
 private String productId;  
 private String productName;  
 private String category;  
 private double price;  
  
 public Product(String productId, String productName, String category, double price) {  
 this.productId = productId;  
 this.productName = productName;  
 this.category = category;  
 this.price = price;  
 }  
  
 public String getProductId() {  
 return productId;  
 }  
  
 public String getProductName() {  
 return productName;  
 }  
  
 public String getCategory() {  
 return category;  
 }  
  
 public double getPrice() {  
 return price;  
 }  
  
 @Override  
 public String toString() {  
 return "Product{" +  
 "ID='" + productId + '\'' +  
 ", Name='" + productName + '\'' +  
 ", Category='" + category + '\'' +  
 ", Price=" + price +  
 '}';  
 }  
  
 @Override  
 public int compareTo(Product other) {  
 return this.productName.compareToIgnoreCase(other.productName);  
 }  
}

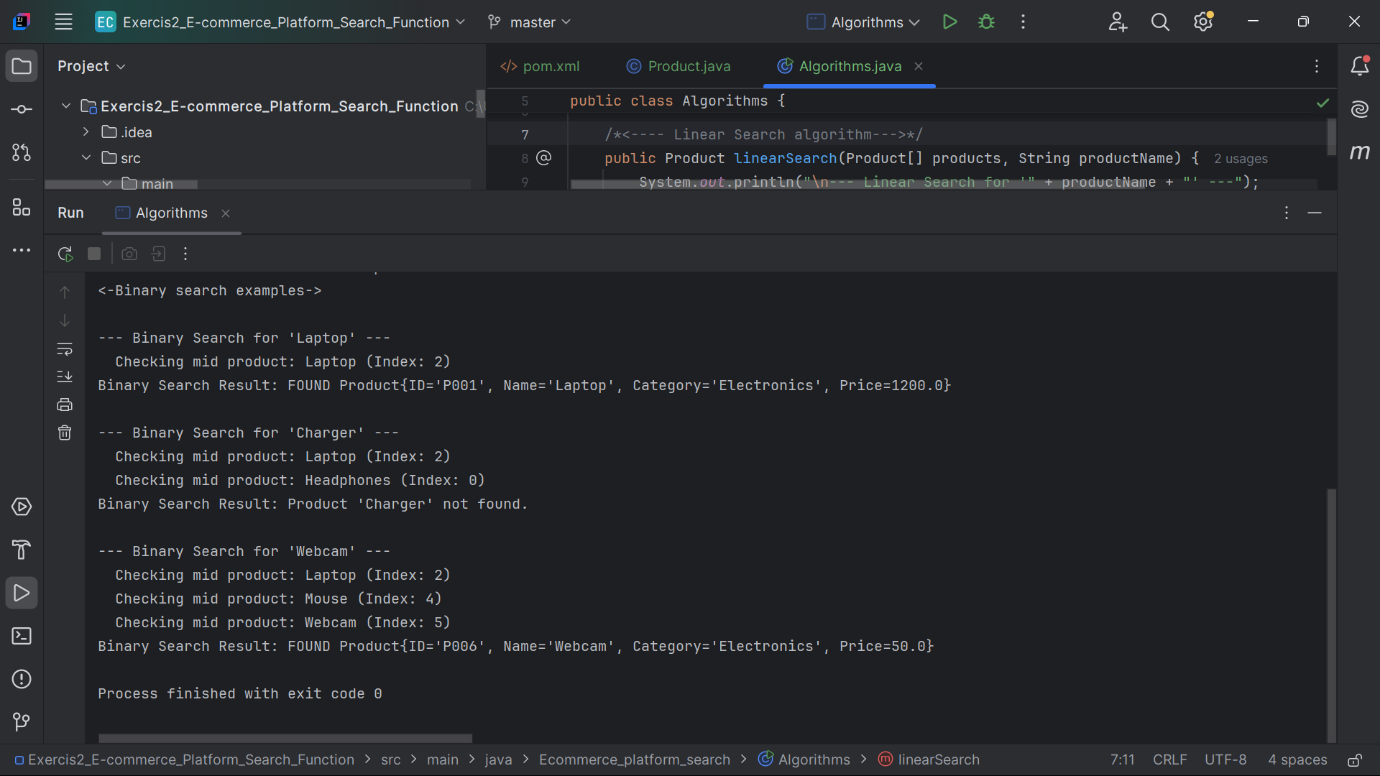
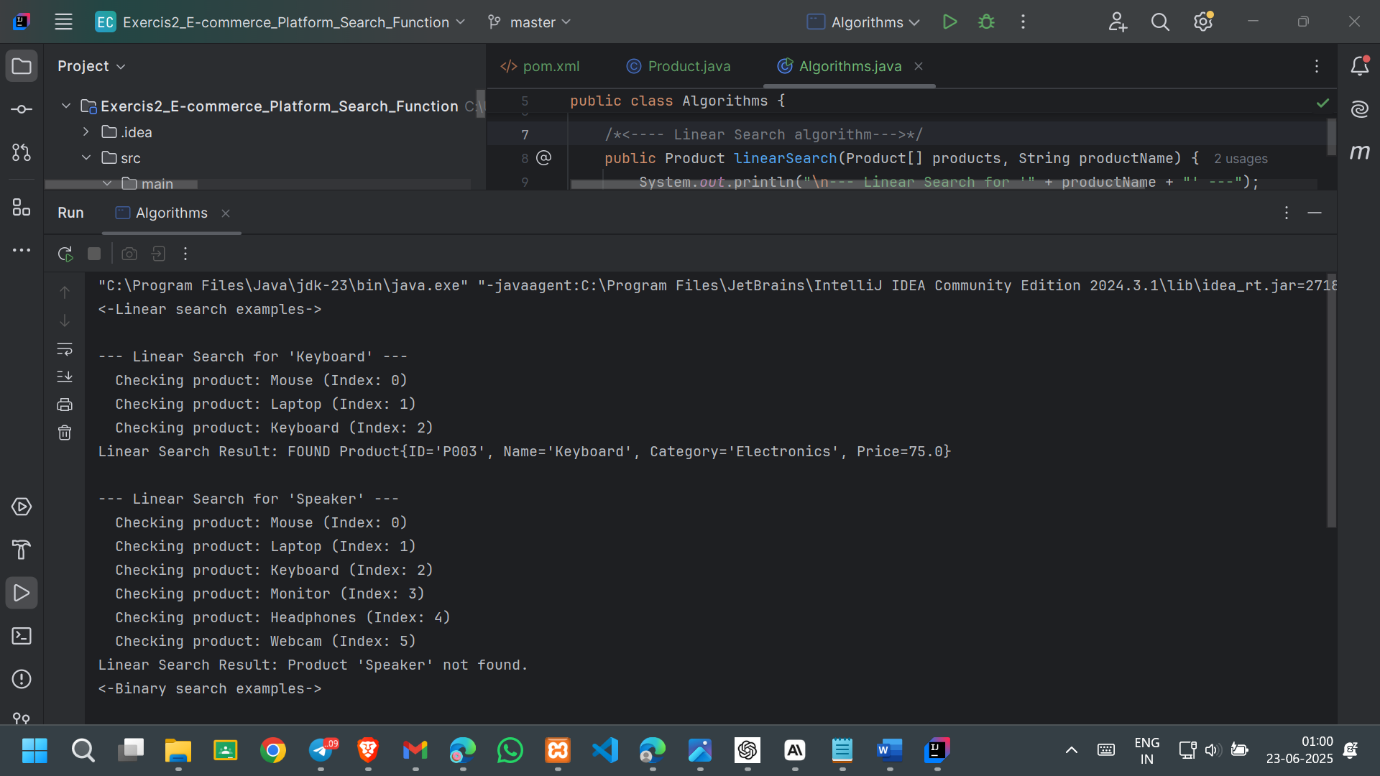
**3. Implementation**

**Java code**

Algorithms.java

package Ecommerce\_platform\_search;  
  
import java.util.\*;  
  
public class Algorithms {  
  
 /\*<---- Linear Search algorithm--->\*/  
 public Product linearSearch(Product[] products, String productName) {  
 System.*out*.println("\n--- Linear Search for '" + productName + "' ---");  
  
 for (int i = 0; i < products.length; i++) {  
 System.*out*.println(" Checking product: " + products[i].getProductName() + " (Index: " + i + ")");  
  
 if (products[i].getProductName().equalsIgnoreCase(productName)) {  
 return products[i];  
 }  
 }  
 return null;  
 }  
  
 /\*<---- Binary Search algorithm--->\*/  
 public Product binarySearch(Product[] products, String productName) {  
 System.*out*.println("\n--- Binary Search for '" + productName + "' ---");  
  
 int low = 0;  
 int high = products.length - 1;  
  
 while (low <= high) {  
 int mid = low + (high - low) / 2;  
 String midProductName = products[mid].getProductName();  
 System.*out*.println(" Checking mid product: " + midProductName + " (Index: " + mid + ")");  
  
 int compare = productName.compareToIgnoreCase(midProductName);  
  
 if (compare == 0) {  
 return products[mid];  
 } else if (compare < 0) {  
 high = mid - 1;  
 } else {  
 low = mid + 1;  
 }  
 }  
 return null; // Product not found  
 }  
  
 public static void main(String[] args) {  
  
 // --- Sample Products ---  
 Product[] unsortedProducts = {  
 new Product("P005", "Mouse", "Electronics", 25.00),  
 new Product("P001", "Laptop", "Electronics", 1200.00),  
 new Product("P003", "Keyboard", "Electronics", 75.00),  
 new Product("P002", "Monitor", "Electronics", 300.00),  
 new Product("P004", "Headphones", "Audio", 150.00),  
 new Product("P006", "Webcam", "Electronics", 50.00)  
 };  
  
 Algorithms searcher = new Algorithms();  
  
 // --- Demonstrate Linear Search ---  
 System.*out*.println("<-Linear search examples->");  
  
 Product foundProduct1 = searcher.linearSearch(unsortedProducts, "Keyboard");  
 if (foundProduct1 != null) {  
 System.*out*.println("Linear Search Result: FOUND " + foundProduct1);  
 } else {  
 System.*out*.println("Linear Search Result: Product not found.");  
 }  
  
 Product foundProduct2 = searcher.linearSearch(unsortedProducts, "Speaker");  
 if (foundProduct2 != null) {  
 System.*out*.println("Linear Search Result: FOUND " + foundProduct2);  
 } else {  
 System.*out*.println("Linear Search Result: Product 'Speaker' not found.");  
 }  
  
 // --- Prepare data for binary search ---  
 Product[] sortedProducts = Arrays.*copyOf*(unsortedProducts, unsortedProducts.length);  
  
 // Sort the array  
 Arrays.*sort*(sortedProducts);  
  
 // --- Demonstrate Binary Search ---  
 System.*out*.println("<-Binary search examples->");  
  
 Product foundProduct3 = searcher.binarySearch(sortedProducts, "Laptop");  
 if (foundProduct3 != null) {  
 System.*out*.println("Binary Search Result: FOUND " + foundProduct3);  
 } else {  
 System.*out*.println("Binary Search Result: Product not found.");  
 }  
  
 Product foundProduct4 = searcher.binarySearch(sortedProducts, "Charger");  
 if (foundProduct4 != null) {  
 System.*out*.println("Binary Search Result: FOUND " + foundProduct4);  
 } else {  
 System.*out*.println("Binary Search Result: Product 'Charger' not found.");  
 }  
  
 Product foundProduct5 = searcher.binarySearch(sortedProducts, "Webcam");  
 if (foundProduct5 != null) {  
 System.*out*.println("Binary Search Result: FOUND " + foundProduct5);  
 } else {  
 System.*out*.println("Binary Search Result: Product 'Webcam' not found.");  
 }  
 }  
}

**Result**



**4. Analysis**

**a) Comparing the time complexity of linear and binary search algorithms.**

**Linear Search:**

* **Best Case: O(1) -** When the target element is the first element in the array.
* **Average Case: O(n)** - On average, the algorithm checks about half of the elements.
* **Worst Case: O(n) -** The target element is the last element, or it's not present at all, requiring a scan of all n elements.
* **Requires:** No prior sorting of the data.
* **Space Complexity: O(1) -** Uses a constant amount of extra space.

**Binary Search:**

* **Prerequisite**: The data *must be sorted*. The time complexity of sorting (e.g., O(n log n)) should be considered if the data is not already sorted.
* **Best Case: O(1) -** The target element is exactly in the middle on the first attempt.
* **Average Case: O(log n) -** The search space is halved in each step.
* **Worst Case: O(log n) -** The target element is found after repeatedly halving the search space until only one element remains, or it's determined to be absent.
* **Space Complexity:** 
  1. **O(1)** for iterative implementation (like above);
  2. **O(log n)** for recursive implementations due to call stack.

**b) Discuss which algorithm is more suitable for your platform and why.**

For an e-commerce platform's search functionality, Binary Search is generally much more suitable than Linear Search, especially for a large number of products**.**

**Why is Binary Search more suitable:**

1. **Speed for Large Datasets: E-commerce platforms typically deal with millions of products.** 
   * **Linear Search (O(n)):** If you have 1,000,000 products, a linear search might require up to 1,000,000 comparisons in the worst case. This is unacceptably slow for a real-time user experience.
   * **Binary Search (O(log n)):** For 1,000,000 products, log₂ (1,000,000) is approximately 20. This means binary search would require at most around 20 comparisons, which is incredibly fast and provides a responsive user experience.
2. **User Expectation:** Users expect instant search results on e-commerce platforms. O(n) performance cannot deliver this for large catalogues.
3. **Scalability:** As the number of products grows (which it inevitably will on a successful platform), O(log n) algorithms scale gracefully, while O(n) algorithms quickly become a bottleneck.