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

**What is Asymptotic Notation?**

Asymptotic notation is used to describe the **efficiency** of an algorithm as the input size increases. It provides a mathematical way to express the **growth rate** of an algorithm's running time or memory usage, helping developers understand how their code will perform on large inputs. Rather than exact timings, it focuses on how performance scales—ignoring machine speed, programming language, or implementation details.

**Big O Notation (O)**

Big O is the most commonly used asymptotic notation. It represents the **worst-case scenario**, or the maximum time an algorithm can take for any input size. It’s especially important in systems where performance guarantees matter. For example, in linear search, if the target element is not in the list, every item must be checked—resulting in a time complexity of **O(n)**. In contrast, binary search only checks a logarithmic number of elements, yielding a complexity of **O(log n)**

**Other Asymptotic Notation**

**Omega Notation (Ω)**

Omega notation represents the **lower bound** of an algorithm’s running time. It tells us the **fastest** an algorithm can complete its task under the **most favorable input conditions**. For example, in linear search, if the target is the first element, it takes only one step, giving a time complexity of **Ω(1)**. Although useful, Omega doesn't reflect typical or worst-case behavior.

**Theta Notation (Θ)**

Theta notation defines the **tight bound**, meaning it describes the **average-case performance** or the exact growth rate of an algorithm. It lies between the best and worst cases. For example, in binary search, the target is typically found after a few halving steps, resulting in **Θ(log n)**. Theta provides a realistic view of algorithm efficiency under **normal input conditions**.

**How Big O Notation Helps in Analyzing Algorithms**

Big O notation is a fundamental concept in algorithm analysis that offers several practical benefits. Here's how it helps:

 **1. Evaluates Scalability**  
Big O shows how an algorithm's running time or space usage increases as the input size grows, helping assess whether it can handle large datasets efficiently.

 **2. Compares Algorithms**  
It allows developers to compare different algorithms based on their performance, making it easier to choose the most efficient one for a given problem.

 **3. Hardware Independence**  
Big O focuses on the algorithm’s behavior rather than exact timings, making performance evaluation consistent across different machines and environments.

 **4. Simplifies Complexity Analysis**  
By focusing only on the dominant term and ignoring constants or minor variations, Big O simplifies the process of analyzing and discussing algorithm performance.

**Search Operations**

Search operations are fundamental to computer science and are used to **locate a specific element** within a collection of data. In the context of an e-commerce platform, for example, search operations help users find products by name, category, or ID. The efficiency of these operations greatly affects system performance, especially as data grows.

There are two common types of search algorithms:

 **1. Linear Search:**  
Also known as sequential search, it checks each element one by one until the target is found or the list ends. It works on both sorted and unsorted data and is simple to implement but inefficient for large datasets.

 **2. Binary Search:**  
This algorithm works only on sorted data. It repeatedly divides the list in half, comparing the target with the middle element. It is much faster than linear search for large datasets, with logarithmic time complexity.

**1. Linear Search**

Linear search checks each element one by one until it finds the target or reaches the end of the list.

* 🔸 **Best Case (Ω(1))**  
  The best case occurs when the target element is at the first position. Only one comparison is needed, making it the fastest scenario.
* 🔸 **Average Case (Θ(n))**  
  On average, the target might be located somewhere in the middle of the list. This results in scanning about half the elements, giving a time complexity of Θ(n).
* 🔸 **Worst Case (O(n))**  
  The worst case happens when the target is the last element or not present at all. In such a case, all n elements must be checked, resulting in a time complexity of O(n).

### ****2. Binary Search****

Binary search works on a **sorted list** and repeatedly divides the search range in half to locate the target.

* 🔸 **Best Case (Ω(1))**  
  The best case is when the target is found in the **first comparison**, i.e., it's the **middle element** of the array. Only one step is needed.
* 🔸 **Average Case (Θ(log n))**  
  Typically, the target is found after a few divisions. Since the list is halved each time, the number of steps grows **logarithmically**, resulting in **Θ(log n)** time complexity.
* 🔸 **Worst Case (O(log n))**  
  The worst case occurs when the target is at the **start or end** of the list, or not present at all. Even then, binary search will take at most **log n** steps, making it very efficient compared to linear search.

**SourceCode:**

import java.util.Arrays;

import java.util.Comparator;

public class ProductSearch {

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;

}

@Override

public String toString() {

return "Product ID: " + productId +

", Name: " + productName +

", Category: " + category;

}

}

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) {

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

int left = 0;

int right = products.length - 1;

while (left <= right) {

int mid = (left + right) / 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;

}

public static void main(String[] args) {

Product[] products = {

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

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

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

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

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

};

String target = "Mobile";

System.out.println("Linear Search Result:");

Product foundLinear = linearSearch(products, target);

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

System.out.println("\nBinary Search Result:");

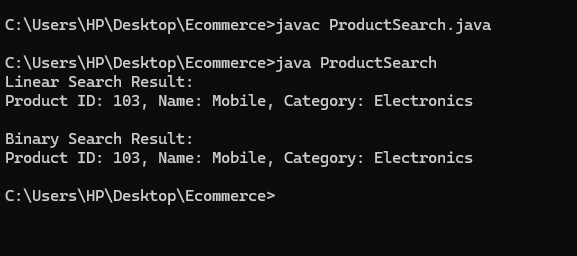
Product foundBinary = binarySearch(products, target);

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

}

}

**Output:**

****

**Analysis**

Search functionality in an e-commerce platform must be fast, scalable, and reliable. To choose the right search algorithm, we compare **Linear Search** and **Binary Search** based on multiple performance and design criteria.

**Detailed Time Complexity & Feature Comparison**

| **Aspect** | **Linear Search** | **Binary Search** |
| --- | --- | --- |
| **Best Case (Ω)** | Ω(1) – first element match | Ω(1) – middle element match |
| **Average Case (Θ)** | Θ(n) – check half the array | Θ(log n) – log₂n comparisons |
| **Worst Case (O)** | O(n) – last or no match | O(log n) – item not found |
| **Array Requirement** | Works on unsorted arrays | Requires sorted array |
| **Implementation Complexity** | Very simple | Slightly complex (needs sorting) |
| **Performance on Large Data** | Poor – scales linearly | Excellent – logarithmic growth |
| **Scalability** | Not suitable | Highly scalable |

### ****Suitability for E-commerce****

In modern e-commerce platforms, search functionality must be fast and scalable. With thousands of products, performance becomes critical — slow searches can lead to poor user experience and reduced conversions.

Since product data is often **sorted or indexed, Binary Search** is highly suitable. It offers **logarithmic time complexity**, making it efficient even for large datasets. Unlike Linear Search, which checks every element, Binary Search reduces the search space by half at each step, ensuring quick and consistent results.

Its performance, scalability, and low overhead make Binary Search ideal for real-time features like product lookup, filters, and search suggestions. In short, for platforms handling large product inventories**, Binary Search is the preferred choice** due to its speed, efficiency, and reliability.

**Conclusion**

While both algorithms have their uses**, Binary Search is the better choice** for an e-commerce platform due to its superior time complexity and performance on large, sorted datasets. It ensures faster search results, which leads to better user experience and platform responsiveness.