E-commerce Platform Search Function

Understanding Big O Notation

Big O notation is a mathematical concept used in computer science to describe how the performance of an algorithm changes with the size of its input. It helps us analyze how quickly an algorithm runs or how much space it uses as the input grows. Below are common time complexities:

O(1) – *Constant time:* The operation takes the same amount of time regardless of input size. *Example: Accessing an element in an array by its index.*

O(log n) – *Logarithmic time:* The input size is repeatedly divided, which reduces the amount of work each step. *Example: Binary search in a sorted array.*

O(n) – *Linear time:* The time grows in direct proportion to the input size. *Example: Searching through an unsorted list.*

O(n log n) – *Linearithmic time:* A combination of linear and logarithmic behavior. *Example: Efficient sorting algorithms like mergesort or heapsort.*

O(n^2) – *Quadratic time:* Time increases significantly with input size. *Example: Inefficient sorting algorithms like bubble sort.*

Search Algorithms in an E-commerce Platform

1. Linear Search

Linear search goes through each product one by one until it finds the desired item.

```
public int linearSearch(int productId) {
   for (int i = 0; i < products.length; i++) {
      if (products[i].getProductId() == productId) {
         return i;
      }
   }
   return -1;
}
```

Best-case time complexity: O(1) – The product is found at the first position.

Average-case time complexity: O(n) – The product is somewhere in the middle.

Worst-case time complexity: O(n) – The product is last in the list or not present at all.

2. Binary Search

Binary search is more efficient but requires the product list to be sorted. It checks the middle element and narrows the search range by half each time.

```
public int binarySearch(int productId) {
   int low = 0;
   int high = products.length - 1;
   while (low <= high) {
      int mid = low + (high - low) / 2;
      if (products[mid].getProductId() == productId) {
         return mid;
      } else if (products[mid].getProductId() > productId) {
         high = mid - 1;
      } else {
         low = mid + 1;
      }
   }
   return -1;
}
```

Best-case time complexity: O(1) – The product is exactly at the middle position.

Average-case time complexity: O(log n) – The list is divided in half each step.

Worst-case time complexity: O(log n) – The product is near the beginning or end.

Conclusion

Both linear and binary search methods are useful depending on the situation:

Linear search is straightforward and works on any list but becomes inefficient as the number of products increases.

Binary search is significantly faster but only works when the product list is sorted.

In scenarios where the product list is large and sorted, binary search is clearly the more efficient and preferred option.