**Understanding Asymptotic Notation**

* **Big O Notation**

Big O notation is a mathematical concept used in computer science to describe the performance or complexity of an algorithm. Specifically, it provides an upper bound on the time complexity or space complexity of an algorithm as a function of the size of the input. It helps in analyzing how the runtime or space requirements of an algorithm grow with the input size, allowing us to compare the efficiency of different algorithms.

**Here’s a brief overview of Big O notation:**

**O(1)**: Constant time complexity. The runtime or space requirement is fixed and does not change with the input size. For example, accessing an element in an array by index.

**O(log n)**: Logarithmic time complexity. The runtime or space requirement grows logarithmically with the input size. For example, binary search in a sorted array.

**O(n)**: Linear time complexity. The runtime or space requirement grows linearly with the input size. For example, linear search in an array.

**O(n log n)**: Linearithmic time complexity. The runtime or space requirement grows proportionally to n \* logn. For example, efficient sorting algorithms like mergesort and heapsort.

**O(n²)**: Quadratic time complexity. The runtime or space requirement grows quadratically with the input size. For example, bubble sort or insertion sort.

**O(2^n)**: Exponential time complexity. The runtime or space requirement grows exponentially with the input size. For example, solving the traveling salesman problem using brute force.

**O(n!)**: Factorial time complexity. The runtime or space requirement grows factorially with the input size. For example, solving the traveling salesman problem using all possible permutations.

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

1. Comparison: Big O notation allows for the comparison of different algorithms and data structures based on their time and space requirements. This helps in choosing the most efficient solution for a given problem.

**Search Operations**

When analyzing search operations, it's useful to consider the best, average, and worst-case scenarios:

1. **Linear Search:**

Best-case: O(1). The element being searched is the first element in the list.

Average-case: O(n). On average, the search will need to check half of the elements in the list.

Worst-case: O(n). The element is either not in the list or is the last element.

2. **Binary Search (for a sorted array):**

Best-case: O(1). The middle element of the array is the target element.

Average-case: O(log n). The search repeatedly divides the array in half.

Worst-case: O(log n). The search has to go through the maximum number of divisions to find the element or determine it is not in the array.

* **What is more suitable in our application:**

We are using an array to store products. Arrays provide constant time random access that is not provided by some other data structures like Linked lists. In case we can sore the products in sorted order of their productId, then we can use **binary search** as it is a better candidate for search operations. Binary search also requires random access to different elements to make decision about its next move, which is provided by arrays.

Keeping a sorted array is helpful in searching but during insertion and deletion it may turn out to provide worse performance. If we are not concerned about those or can manage them correctly, then binary search is very useful, else Linear search is the only way, as it would not require us to keep a sorted array.