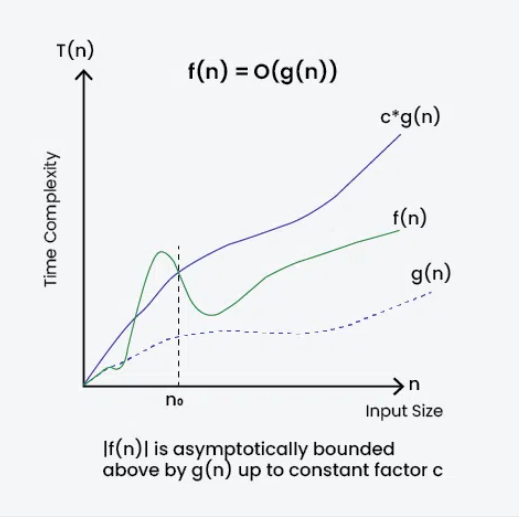
**Exercise 2: E-Commerce Platform Search Function**

1. Understanding the concepts :-
   1. Explain Big O notation and how it helps in analyzing algorithms.

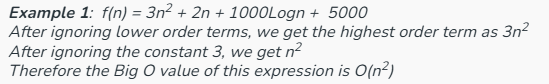
 p.c – Geeks for Geeks

Time complexity of a program or application is very important to analyze in order to maintain a clean, optimized and efficient application. In order to measure this time complexity we use the notation of Big O. Time complexity of an algorithm is the measure of the total time it takes to complete the whole operation and Big O is a way of expressing the upper bound or the worst case complexity of the algorithm.

Big O measures the asymptotic behavior of a function that is it gives the measure of growth or variance of time limit with respect to the size of input and not the exact actual time the program takes to complete the operations. It is also used to compare two different algorithms and their long term behavior that is with increasing size of input how they behave . It is denoted by the notation O(f(n)) where f(n) is a function that represents the number of operations that an algorithm performs to solve a problem of size n.   
  
The above diagram derives the definition of Big O as well as says how Big O analyses the time complexity of an algorithm.

**Definition** - Given two functions**f(n)** and **g(n)**, we say that**f(n)** is**O(g(n))** if there exist constants**c > 0** and **n0 >=**0 such that**f(n) <= c\*g(n)** for all **n >= n0**.

In simpler terms,**f(n)** is **O(g(n))** if**f(n)** grows no faster than**c\*g(n)** for all n >= n0 where c and n0 are constants.

Following example shows how Big O can be used to find complexity given a function f(n)  
p.c-GFG

So since we take up worst case complexity in Big O notation we represent f(n) with the highest degree of n it has within itself. For example in the above f(n) we have three different types of n – n,n^2,log n but among all these since n^2 is the highest power of n so taking up worst case complexity we represent f(n) =O(n^2).

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

In the context of search operations, best-case, average-case, and worst-case scenarios describe the different outcomes based on how the search algorithm performs with different inputs. The best case represents the most efficient outcome, the worst case represents the least efficient outcome, and the average case represents the typical performance across a range of inputs.

Best-Case Scenario:

* **Definition:**

The best case occurs when the search algorithm finds the target element in the fewest number of steps.

* **Example:**

For a linear search, the best case is when the target element is the first element in the list. For a binary search, the best case is when the target element is the middle element of the sorted list.

Time complexity of best case scenario for both the searches are O(1).

Average-Case Scenario:

* **Definition:**

The average case represents the typical performance of the search algorithm when considering a random distribution of target elements within the search space.

* **Example:**

For a linear search, the average case is when the target element is found roughly in the middle of the list. For a binary search, the average case is when the target element is found somewhere within the sorted list, not at either extreme.

Time complexity of average case scenario for binary search is O(log(n)) and linear search is O(n).

Worst-case Scenario :

* **Definition:**

The worst-case scenario represents the least efficient outcome for the search algorithm, meaning the maximum number of operations the algorithm might need to perform.

* **Example:**

For a linear search, the worst case is when the target element is the last element in the list or is not present at all. For a binary search, the worst case is when the target element is not present in the list and the algorithm has to search through the entire list.

Time Complexity of Worst case scenario for binary search is O(log(n)) and linear search is O(n).

4. Analysis

a. Discuss which algorithm is more suitable for your platform and why.

Now according to me both the algorithms are suitable based on how I am searching the product . As my program implements search by name of product through linear search since bubble sort program needs sorting of the products by the names first and then searching is done which is a bit harder and complex and then the searching can be done through linear search which just matches the given string with each name of product which seems to be more simple .  
  
In second case when I am searching the products by their id it is more efficient to do binary search because sorting through id and then searching is more efficient even if it takes a little more time from linear search . So basically if products can be someway stored in sorted or alphabetical order initially through some structures or collections then binary search will be efficient but if they can’t be sorted then products must be searched using linear search.

1. Compare the time complexity of linear and binary search.

Linear search is the most common and simple search in which in order to find an element from an array we traverse the array and stops where we find the element and return the index . Now what can be our worst case ? In worst case the element may not be present in the array and we traverse the whole array but become unsuccessful in finding the element or the element may be present at the last position , in that case also we have to traverse the whole array . So given an array of size n in worst case we traverse all the n elements and we either find the element at the last position or not find at all . **So assuming unit time to traverse to each element total time taken at the worst case is n times and hence the time complexity is O(n).**

In best case we can find the element at the first position only without needing to move further hence we can find the element at constant time if either we know where our element can be or if we find it at the very first position . **Hence time complexity for best case is O(1).**

For average case our element can be present somewhere in the middle of the array where we have to traverse only half the array lets say till n/2 elements. **Hence in average case also we will have time complexity as O(n)** since Big O takes highest function of n ignoring the coefficients.

Now for binary search we must have our array in sorted order .It is a bit complex than linear search and recursively performs division operation on the array and its subparts. It uses divide and conquer rule to find the element. Searching is done by dividing the array into two halves and again applying search on the part where the element can be present .

Best case of binary search is when the element is at the middle index of the array. It takes only one comparison to find the target element. So the best case complexity is **O(1)**.

**For average case :** Consider array **arr[]** of length **N** and element **X** to be found. There can be two cases:

Case1: Element is present in the array

Case2: Element is not present in the array.

There are N Case1 and 1 Case2. So total number of cases = N+1. Now notice the following:

An element at index N/2 can be found in 1 comparison

Elements at index N/4 and 3N/4 can be found in 2 comparisons.

Elements at indices N/8, 3N/8, 5N/8 and 7N/8 can be found in 3 comparisons and so on.

Based on this we can conclude that elements that require:

1 comparison = 1

2 comparisons = 2

3 comparisons = 4

x comparisons = 2x-1 where x belongs to the range [1, logN] because maximum comparisons = maximum time N can be halved = maximum comparisons to reach 1st element = logN.

So, total comparisons  
= 1\*(elements requiring 1 comparisons) + 2\*(elements requiring 2 comparisons) + . . . + logN\*(elements requiring logN comparisons)  
= 1\*1 + 2\*2 + 3\*4 + . . . + logN \* (2logN-1)  
= 2logN \* (logN - 1) + 1  
= N \* (logN - 1) + 1

Total number of cases = N+1.

**Therefore, the average complexity = (N\*(logN - 1) + 1)/N+1 = N\*logN / (N+1) + 1/(N+1). Here the dominant term is N\*logN/(N+1) which is approximately logN. So the average case complexity is O(logN)**

And finally the worst case will be when the element is present in the first position. As seen in the average case, the comparison required to reach the first element is logN**. So the time complexity for the worst case is O(logN).**