

Algorithms Analysis:

Study of *nature of algorithms* for the *time taken* and the *amount of memory* used in its execution. **Essentially it is the analysis of effect of inputs (size and the nature) on the time taken and memory consumed by the algorithm.**

There are two aspects of Complexity Analysis:

1. Algorithmic aspect - involves counting of the number of operations (for time complexity analysis) and the total number of bytes (for space complexity analysis) in terms of the input size.
2. Mathematical aspect - involves finding the BigO, Omega and Theta from the function formulated by counting the steps/memory.

Since it involves mathematical analysis, it can get quite complicated for algorithms.

Steps for performing the algorithmic analysis (Time complexity analysis for the time being. Space complexity will also have the same steps except the count will be the memory count)

1. Count the number of steps.
2. Formulate a mathematical function f(n) from the count, where n is the input size.
3. If possible, make the function f(n) compact that involves only multiplications.
4. Find a function u(n), an upper bound to the function f(n) that is u(n) > f(n) for all n.
5. Find a function l(n), a lower bound to the function f(n) that is l(n) < f(n) for all n.
6. Perform asymptotic analysis on u(n), which results in f(n) = O(<some>).
7. Perform asymptotic analysis on l(n), which results into f(n) = Ὠ(<some>).
8. If <some> in O() and Ὠ() are same, then it is called as ϴ(<some>) else ϴ does not exist.

*Note that for any recursive algorithm, the space required is proportional to the maximum depth of the recursion tree, because that is the maximum number of elements that can be present in the implicit function call stack.*

Important points:

* The asymptotic analysis is performed on functions u(n) and l(n) not on f(n).
* In step 3, we are trying to make the function compact (reduce to strictly increasing or strictly decreasing, if it involves functions like sines, cosines then it is hard to find upper limit and the lower limit function). If successful, then both u(n) and l(n) can be defined as constant multiples of f(n), That is ,

u(n) = p.f(n)

l(n) = q.f(n) , where p and q are constants such that u(n) > f(n) and q(n) < f(n) respectively.

With this simplification, it looks like the asymptotic analysis is performed on f(n) which is an illusion.

* Generally, we care only about BigO complexities because that provides the upper bound.
* The entire analysis not only depends upon the input size n but also depends on the permutation of the n inputs provided. Each of the permutations has its BigO complexities. The relevant complexity out of all the permutations depends upon the use case. If the worst-case permutation is more frequent than the others, then it can be attributed as the complexity, otherwise the average case permutation complexity can be more relevant. For example, Insertion sort and Selection sort. (Refer Brilliant.org)
* Tip: In the case of an algorithm with nested loops, calculate the number of times the inner loop runs for each iteration of the outer loop.
* During the Step1 - Counting of the steps, if the bottleneck operation is identified then we can directly consider the count of that operation and ignore the rest.
* **[Important]** The time complexity represented by any of the asymptotic notations are meaningful only for large values of ‘n’, the input size. The concept is explained below.

Let’s say the time complexity of an algorithm is O(n2).

It implies the following:

The bottleneck operation which defines the cost runs in a time proportional to n2, and the runtime, ‘t’ of the algorithm can be described as

t = C.n2

If the algorithm is timed for different input sizes then the time and the input size will be related by a quadratic polynomial.

The constant factor, C depends upon the implementation and numerous other factors. If two algorithms with quadratic time complexity are compared then, the constant C will be different for them. Moreover, it is also possible that a less efficient algorithm (with greater/worse time complexity) can run faster than the more efficient algorithm (smaller/better time complexity) for smaller values of the input size. (Refer Brilliant.org Algorithms course for more details). But for the large enough values of n, the former will perform better.

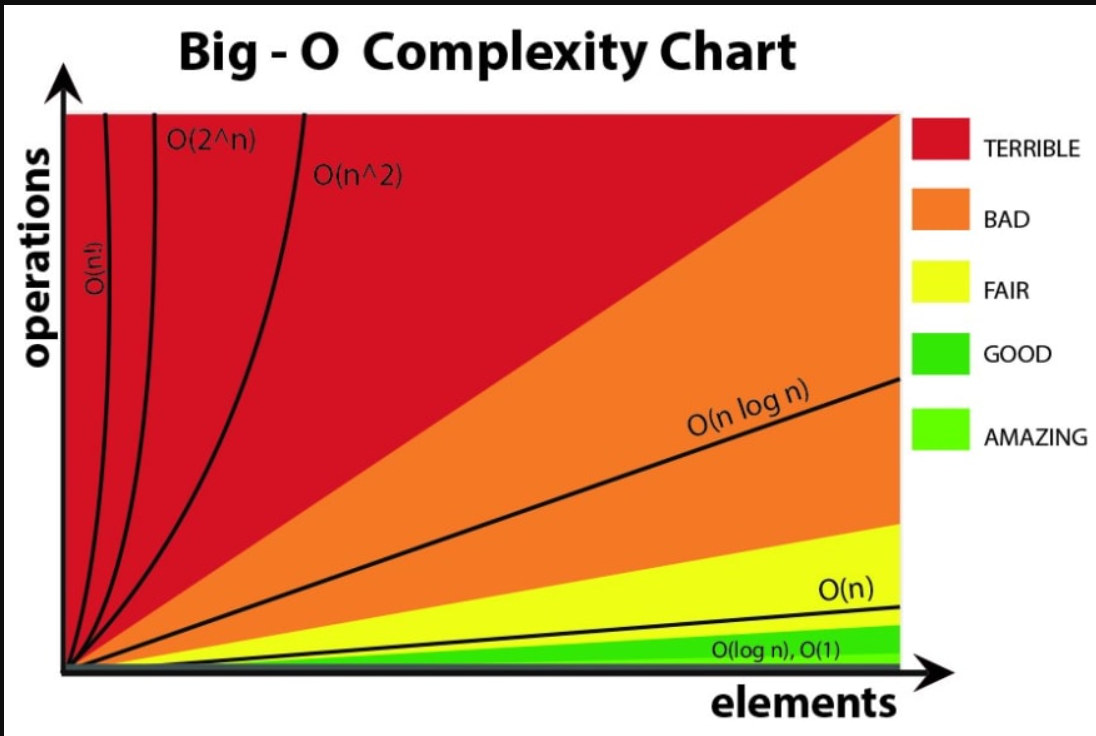
Generally, we do not have to worry about the value of C, because comparing the algorithms for large input size makes sense as for the smaller values, the difference in performance is insignificant.

* In space complexity, we are generally concerned about the auxiliary space complexity, that is the extra space used by the algorithm apart from the inputs, rather than the overall space complexity.

Common time complexities

O(1) > O(logn) > O(n) > O(nlogn) > O(n2) > O(2n) > O(n!)

99% of all the cases will result in these complexities.



References:

[How to find time complexity of an algorithm - Stack Overflow](https://stackoverflow.com/questions/11032015/how-to-find-time-complexity-of-an-algorithm)