What does 'upper bound' mean in context of BigO?

My computer science teacher says Big O has an upper bound but no lower bound. When I look at a graph of an algorithm mapped out using BigO though, there isn't an upper bound at all. The upper limit goes on forever. So what do it mean to say there is an upper bound in the context of BigO?

algorithms computer-science big-o

asked Sep 26 '14 at 19:18
chopper draw lion4
239 1 5

As an addition to Cragor's answer: Plain English explanation of Big O – informatik01 Sep 26 '14 at 21:48

1 Answer

BigO is, simply put, a measure of algorithm complexity. In a best case scenario, even the worst algorithm may be finished in one operation/iteration. However, different operations have different average- and worst-case complexities. These worst-case complexities could be considered the upper bound.

So, how can we figure out the upper bound?

An algorithm will have different running durations based on the size of the data it is handling. As an example, let's consider a simple data structure, which could be implemented in a number of ways (let's use an array) and we decide that we want to search the structure for a piece of data. The number of operations you have to do will be based on the size of the collection of data. Let's assume there are `n` elements in the structure.

A typical array will, at worst case, iterate through the entire collection for this, meaning that you will perform up to `n` operations, resulting in `O(n)` complexity, so you have an upper-bound of `n`.

Let's assume that the data is sorted: you could now perform a binary search, resulting in `log(n)` operations, reducing the complexity with an upper bound of `O(log(n))`. It could still be completed in one operation, and if `n` approached an infinite number, the complexity would approach infinite execution time. This is probably what you were seeing in class. Different algorithm complexities approach this infinite execution level at different rates (n! > n^2 > n > log(n) > 1).

Edit: As per the comments, it should be noted that a change in the amount of data will also be reflected by a change in execution time based on the complexity. O(1) algorithms will not change, logarithmic ones will change in a logarithmic manner, linear algorithms linearly, etc. Essentially, doubling the amount of data may NOT double your execution time. You may quadruple it, or increase it by some smaller or larger factor.

For a more complex example, more work is required to figure out the complexity of course, but this is the general idea:

The Upper Bound of an algorithm's complexity describes how the algorithm's execution time changes with a change in the amount of data being processed in the operation, meaning that a more complex algorithm will often take an increasingly (often not linearly) long time to execute as the amount of data is increased.

edited Sep 26 '14 at 20:30

answered Sep 26 '14 at 19:35
Cragor
156 3

To more specifically answer the question, it means that the algorithm will grow at most as quickly as the upper bound, ie if your algorithm is O(n), then if you double n (the size of the data), your algorithm will at most take twice as long. But it has no lower bound, in other words, it might actually take less, maybe even much less in most cases, than that. – HamHamJ Sep 26 '14 at 20:17

That's very true, and an oversight on my part. Do you mind if I edit something in to that regard? – Cragor Sep 26 '14 at 20:21

Go right ahead. – HamHamJ Sep 26 '14 at 20:23

@HamHamJ: "it means that the algorithm will grow at most as quickly as the upper bound" – Actually, that's Little-oh. Big-Oh is "will grow not significantly more quickly than the upper bound". – Jörg W Mittag Sep 26 '14 at 23:59