Big-O Notation - What is it good for?

by Limbic~Region

on Sep 15, 2006 at 14:07 UTC (#573138=perltutorial: print w/ replies, xml)

All,
If you have ever felt that people debating the big-oh notation of some algorithm sound like they are speaking a foreign language, then this tutorial is for you. You may have even decided to educate yourself by checking out the Wikipedia entry and have been convinced it was a foreign language. You are not alone.

You may have already read An informal introduction to O(N) notation by dws which is excellent. This tutorial will repeat much of the same information in a much more elementary manner as well as go into detail about how useful *or not* the notation is. It is likely I won't be completely accurate about a number of things in the tutorial. My hope is that by the end, your understanding of the topic will be sufficient enough that you can understand the corrections others make (*as I am sure they will*).

What Is The Big-O

This tutorial covers the Big-O as it relates to computer science. If you were thinking of something else (perhaps Fridays in the Chatterbox), you can stop reading now. Simply put, it describes how the algorithm scales (*performs*) in the worst case scenario as it is is run with more input. Since my simple explanation may not be simple enough - let me give an example. If we have a sub that searches an array item by item looking for a given element, the scenario that the Big-O describes is when the target element is last (or not present at all). This particular algorithm is O(N) so the same algorithm working on an array with 25 elements should take approximately 5 times longer than an array with 5 elements.

It is easy to lose sight of the fact that there is more to consider about an algorithm other than how fast it runs. The Big-O can also be used to describe other behavior such as memory consumption. We often optimize by trading memory for time. You may need to choose a slower algorithm

because it also consumes less of a resource that you need to be frugal with.

What The Big-O Is Not

Constants: The Big-O is not concerned with factors that do not change as the input increases. Let me give an example that may be suprising. Let's say we have an algorithm that needs to compare every element in an array to every other element in the array. A simple implementation may look like:

```
for my $i (0 .. $#array) {
    for my $j (0 .. $#array) {
        next if $j == $i;
        # Compare $i, $j
    }
}
```
[download]

This is O(N^2). After a little bit of testing we decide that this is far too slow, so we make a little optimization.

```
for my $i (0 .. $#array - 1) {
    for my $j ($i + 1 .. $#array) {
        # Compare $i, $j
    }
}
```
[download]

We have just cut our run time in half - YAY! Guess what, the Big-O has stayed the same O(N^2). This is because N^2 / 2 only has one variable part. The *divided by 2* remains the same (constant) regardless of the input size. There are valid mathematical reasons for doing this but it can be frustrating to see two algorithms with the exact same Big-O that results in one running twice as fast as the other.

Implementation Details: The Big-O is an uncaring cold-hearted jerk. It does not care if you can't afford to buy the extra RAM needed for your problem and have to resort to tying your hash to disk. You are on your own. It also doesn't care that the data structure you would need to implement to achieve O(Log Log N) is so complex you will never be able to maintain it. In a nutshell, the Big-O lives in the land of theory and doesn't care very much about the real world.

What The Big-O Is Good For

The good news is that the Big-O belongs to an entire family of notation. This tutorial will not cover it but family members include describing the average and best cases. It also serves as a good indicator of what algorithm to use once you take your individual circumstances into consideration. Let me give a contrived example:

Let's consider using cacheing as an optimization. In theory, the Big-O is going to ignore it saying your input is all different and you will never

benefit from it. In reality, you test it and discover that you have a 60% hit rate. You do a little more experimenting and discover that the input size required for a more complex algorithm to be faster is larger than your *real* maximum input size. This all despite the more complex alorithm having a more favorable Big-O.

In a nutshell, the Big-O of a given algorithm combined with the specific problem knowledge is a great way to choose the best algorithm for your situation.

What Do Those Symbols Mean?

So by this point you should realize that Big-O (theory) without context (real world) is not very useful. You are now armed with the knowledge necessary to start using Big-O as the mercenary it is. Ok Big-O, what exactly do you mean that algorithm is O(N Log N)? I am going to duck at this point and suggest you read the node by dws or the Wikipedia entry I linked to earlier. You may now be wondering if Big-O is really inanimate, perhaps even an abstract concept and not at all real as I have made it out to be. If so, how then can you determine the Big-O of a given algorithm? Analysis of algorithms is not for the faint of heart, so I must once again duck.

I have not really added anything to any of the other links I referenced. I do hope however that I have put it in plain enough english to be understood by even the most extreme novice. I welcome those more knowledgeable than myself to add corrections as well as provide additional content. I would only ask that you do so in the same spirit of this tutorial (*understandable by non-CS majors*).

Update: Removed an incorrect analogy regarding slopes thanks to blokhead.
Also see Sorting a list of IP addresses (aka Why I hate Big O) by jeffa

Cheers - L~R

Comment on Big-O Notation - What is it good for?
Select or Download Code

Re: Big-O Notation - What is it good for?
by blokhead on Sep 15, 2006 at 17:54 UTC

I think this is a very nice introduction to the topic. But I'd like to clear up a few minor nits which I frequently see, which in the grand scheme of things are not too important. If I'm lucky, I'll be able to clear them up while still keeping things down-to-earth (**Update:** you were probably wise to gloss over these details, since it took me this long to address them)! You've said:

> *... it [big-O] describes how the algorithm scales (performs) in the worst case scenario as it is is run with more input ...*

> *... Big-O belongs to an entire family of notation*
> *... family members include describing the average and best cases ...*

There is a common misconception that Big-O means worst-case, Big-Omega means best-case, Big-Theta means average-case, etc. Well, kinda. In many common cases, that is often what is meant. But this can be an imprecise mental model. Let me propose a different interpretation.

For now let's ignore the distinction between best-case, worst-case, average-case running times. So imagine an algorithm that takes exactly the same amount of time on all inputs of the same size (best-case, worst-case and average-case all the same). Then:

- Big-O gives an *upper-bound*. That is, if the algorithm's running time is O(n^2), it says that the algorithm takes *at most* (some constant times) n^2 steps on these inputs.
- Big-Omega gives an *lower-bound*. That is, if the algorithm's running time is Ω(n), it says that the algorithm takes *at least* (some constant times) n steps on these inputs.

(I'm intentionally omitting Big-Theta here to keep things non-technical)

So **why is Big-O commonly associated with worst-case running times, and why is that imprecise?** It's because when considering the worst possible case, it is natural to give a limit on how bad that worst case can be, not how good it can me. That is, we want to give an *upper bound* on its degree of badness. Similarly, we often want to give a *lower bound* on how good the best-case is (i.e, even on good inputs, there's still a limit on how fast the algorithm can go; what is that limit?), so Big-Omega gets associated with best-case.

That's why Big-O gets associated with worst-case running times, and Big-Omega with best-case. And it's true that if someone just says "the running time" is O(n^2), then n^2 is indeed "closer" to the worst-case running time than to the best-case running time, in the sense that n^2 is "bigger" than all possible running times, and the worst-case running time is "bigger" than the best-case running time. But **O(n^2) doesn't mean that the worst-case running time actually *is* n^2, just that it is at most n^2.**

And it's not wrong or illegal for me to express (for instance) a lower bound on the worst-case running time (i.e, "the worst-case is *at least* this bad"). In fact, this kind of bound can be quite useful. Big-O and friends are just expressing bounds/limits on a function. I can give both upper bounds and

lower bounds for any function, whether that function expresses the worst-case, best-case, or average-case running time of an algorithm.

My second point, which I already hinted at, deals with **how close the upper/lower bounds are to the actual running time**. Remember, Big-O expresses an upper limit over your running time, but it doesn't need to be *close* to the running time! Here's an example. Again, for simplicity, let's just assume that average-case = best-case = worst-case running times.

I have two algorithms. My local CS major tells me that algorithm A's runtime is O(n^3), but algorithm B's runtime is O(n^2). Which is better?

Well, all I know is that algorithm A takes no more than n^3 steps, and algorithm B takes no more than n^2 steps. It may be the case that algorithm A takes *much fewer* than n^3 steps. Maybe it was a difficult algorithm to analyze, and it actually takes exactly n steps, but our CS major could only prove that it took less than n^3. The upper bound of O(n^3) is still absolutely correct. And it may be the case that algorithm B takes exactly n^2 steps on its inputs. It was an easy algorithm to analyze and the CS major could exactly count the number of steps. The upper bound here of O(n^2) is still correct.

In this example, algorithm B has the more "attractive" big-O, but algorithm A is *always* faster. (Update: fixed this sentence according to jpearl's comment)

So while Big-O gives you an upper bound, it does not tell you how good that bound is. It may be a very very rough/sloppy over-estimate. So it's nice to be able to have corresponding lower bounds as well. If the upper and lower bounds for some running time are close, you know that the actual running time must live in the small space between these two bounds. This is why I mentioned that asking for a *lower bound* on the *worst-case* running time could be a useful thing. This is why we get things like Big-Omega and Big-Theta, but for that you'll have to look elsewhere. This reply is already way too long ;)

---

blokhead

[reply]

Re^2: Big-O Notation - What is it good for?
by Anonymous Monk on Jul 10, 2008 at 14:34 UTC

Coming from a background "unbounded" by CS, and more often approaching a mathematical context, the

tutorial explanation, and even moreso the comment aftewards, are amazingly AWESOME! I'm usually left stuck between programmers who dislike math, and mathematicians who just don't get computers, and getting any kind of consensus and mutually beneficial discussion is impossible... A big thank-you to both authors for your time and effort, and if anyone is aware of additional resources for such topics (CS/math and math/CS crossovers, or explanations of theories behind practical applications and, of particular interest to me, practical applications of theories), PLEase share!

[reply]

Re^2: Big-O Notation - What is it good for?
by jpearl on Mar 13, 2009 at 23:44 UTC

Hrm, in your last bolded sentence, do you have A and B switched? Very good intro though :-)

[reply]

Re^3: Big-O Notation - What is it good for?
by Anonymous Monk on Jun 23, 2009 at 01:07 UTC

Thank you. I didn't even have that mental model. Having it really clears things up. Great work!

[reply]

Re: Big-O Notation - What is it good for?
by jrtayloriv on Sep 29, 2009 at 04:24 UTC

It's also important to note that complexity analysis does not consider how a certain algorithm will interact with virtual memory and CPU caches, which is also very important in determining how the algorithm will perform in reality.

--Jesse Taylor

[reply]

Back to Tutorials