Big O Notation

Using not-boring math to measure code's efficiency

The idea behind big O notation

Big O notation is the language we use for talking about how long an algorithm takes to run. It's how we compare the efficiency of different approaches to a problem.

It's like math except it's an **awesome, not-boring kind of math** where you get to wave your hands through the details and just focus on what's *basically* happening.

With big O notation we express the runtime in terms of—brace yourself—*how quickly it grows relative to the input, as the input gets arbitrarily large*.

Let's break that down:

1. **how quickly the runtime grows**—It's hard to pin down the *exact runtime* of an algorithm. It depends on the speed of the processor, what else the computer is running, etc. So instead of talking about the runtime directly, we use big O notation to talk about *how quickly the runtime grows*.
2. **relative to the input**—If we were measuring our runtime directly, we could express our speed in seconds. Since we're measuring *how quickly our runtime grows*, we need to express our speed in terms of...something else. With Big O notation, we use the size of the input, which we call "n." So we can say things like the runtime grows "on the order of the size of the input" ($O(n)$) or "on the order of the square of the size of the input" ($O(n^2)$).
3. **as the input gets arbitrarily large**—Our algorithm may have steps that seem expensive when n is small but are eclipsed eventually by other steps as n gets huge. For big O analysis, we care most about the stuff that grows fastest as the input grows, because everything else is quickly eclipsed as n gets very large. (If you know what an asymptote is, you might see why "big O analysis" is sometimes called "asymptotic analysis.")

If this seems abstract so far, that's because it is. Let's look at some examples.

Some examples

Python 2.7 ▾

```
def print_first_item(items):
    print items[0]
```

This function runs in $O(1)$ time (or "constant time") relative to its input. The input list could be 1 item or 1,000 items, but this function would still just require one "step."

Python 2.7 ▾

```
def print_all_items(items):
    for item in items:
        print item
```

This function runs in $O(n)$ time (or "linear time"), where n is the number of items in the list. If the list has 10 items, we have to print 10 times. If it has 1,000 items, we have to print 1,000 times.

Python 2.7 ▾

```
def print_all_possible_ordered_pairs(items):
    for first_item in items:
        for second_item in items:
            print first_item, second_item
```

Here we're nesting two loops. If our list has n items, our outer loop runs n times and our inner loop runs n *times for each iteration of the outer loop*, giving us n^2 total prints. Thus **this function runs in $O(n^2)$ time (or "quadratic time")**. If the list has 10 items, we have to print 100 times. If it has 1,000 items, we have to print 1,000,000 times.

N could be the *actual* input, or the *size* of the input

Both of these functions have $O(n)$ runtime, even though one takes an integer as its input and the other takes a list:

Python 2.7 ▾

```
def say_hi_n_times(n):
    for time in xrange(n):
        print "hi"


def print_all_items(items):
    for item in items:
        print item
```

So sometimes n is an *actual number* that's an input to our function, and other times n is the *number of items* in an input list (or an input map, or an input object, etc.).

Drop the constants

This is why big O notation *rules*. When you're calculating the big O complexity of something, you just throw out the constants. So like:

Python 2.7 ▾

```
def print_all_items_twice(items):
    for item in items:
        print item

    # Once more, with feeling
    for item in items:
        print item
```

This is $O(2n)$, which we just call $O(n)$.

Python 2.7 ▾

```
def print_first_item_then_first_half_then_say_hi_100_times(items):
    print items[0]

    middle_index = len(items) / 2
    index = 0
    while index < middle_index:
        print items[index]
        index += 1

    for time in xrange(100):
        print "hi"
```

This is $O(1 + n/2 + 100)$, which we just call $O(n)$.

Why can we get away with this? Remember, for big O notation we're looking at what happens **as n gets arbitrarily large**. As n gets really big, adding 100 or dividing by 2 has a decreasingly significant effect.

Drop the less significant terms

For example:

Python 2.7 ▾

```
def print_all_numbers_then_all_pair_sums(numbers):
    print "these are the numbers:"
    for number in numbers:
        print number

    print "and these are their sums:"
    for first_number in numbers:
        for second_number in numbers:
            print first_number + second_number
```

Here our runtime is $O(n + n^2)$, which we just call $O(n^2)$. Even if it was $O(n^2/2 + 100n)$, it would still be $O(n^2)$.

Similarly:

- $O(n^3 + 50n^2 + 10000)$ is $O(n^3)$ * Only consider most time-consuming step
- $O((n + 30) * (n + 5))$ is $O(n^2)$

Again, we can get away with this because the less significant terms quickly become, well, less significant as n gets big.

We're usually talking about the "worst case"

Often this "worst case" stipulation is implied. But sometimes you can impress your interviewer by saying it explicitly.

Sometimes the worst case runtime is significantly worse than the best case runtime:

Python 2.7 ▾

```
def contains(haystack, needle):

    # Does the haystack contain the needle?
    for item in haystack:
        if item == needle:
            return True

    return False
```

Here we might have 100 items in our haystack, but the first item might be the needle, in which case we would return in just 1 iteration of our loop.

In general we'd say this is $O(n)$ runtime and the "worst case" part would be implied. But to be more specific we could say this is worst case $O(n)$ and best case $O(1)$ runtime. For some algorithms we can also make rigorous statements about the "average case" runtime.

Space complexity: the final frontier

Sometimes we want to optimize for using less memory instead of (or in addition to) using less time. Talking about memory cost (or "space complexity") is very similar to talking about time cost. We simply look at the total size (relative to the size of the input) of any new variables we're allocating.

This function takes $O(1)$ space (we use a fixed number of variables):

Python 2.7 ▾

```
def say_hi_n_times(n):
    for time in xrange(n):
        print "hi"
```

This function takes $O(n)$ space (the size of `hi_list` scales with the size of the input):

Python 2.7 ▾

```
def list_of_hi_n_times(n):
    hi_list = []
    for time in xrange(n):
        hi_list.append("hi")
    return hi_list
```

Usually when we talk about space complexity, we're talking about *additional space*, so we don't include space taken up by the inputs. For example, this function takes constant space even though the input has n items:

Python 2.7 ▾

```
def get_largest_item(items):
    largest = float('-inf')
    for item in items:
        if item > largest:
            largest = item
    return largest
```

Sometimes there's a tradeoff between saving time and saving space, so you have to decide which one you're optimizing for.

Big O analysis is awesome except when it's not

You should make a habit of thinking about the time and space complexity of algorithms *as you design them*. Before long this'll become second nature, allowing you to see optimizations and potential performance issues right away.

Asymptotic analysis is a powerful tool, but wield it wisely.

Big O ignores constants, but **sometimes the constants matter**. If we have a script that takes 5 hours to run, an optimization that divides the runtime by 5 might not affect big O, but it still saves you 4 hours of waiting.

Beware of premature optimization. Sometimes optimizing time or space negatively impacts readability or coding time. For a young startup it might be more important to write code that's easy to ship quickly or easy to understand later, even if this means it's less time and space efficient than it could be.

But that doesn't mean startups don't care about big O analysis. A great engineer (startup or otherwise) knows how to strike the right *balance* between runtime, space, implementation time, maintainability, and readability.

You should develop the *skill* to see time and space optimizations, as well as the *wisdom* to judge if those optimizations are worthwhile.

Share Tweet Share

← course home

Next up: Data Structures →