15-122: Principles of Imperative Computation

Recitation 7

Josh Zimmerman, Jonathan Clark

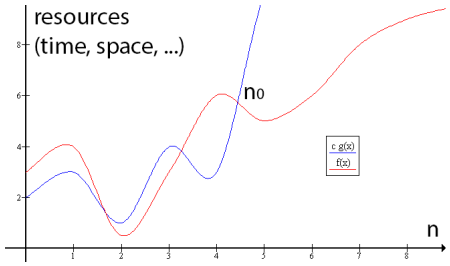
Big-O

The definition of Big-O has a lot of mathematical symbols in it, and so can be very confusing at first. Let's familiarize ourselves with the formal definition and get an intuition behind what it's saying.

First of all, $O(g(n))$ is a *set* of functions. The formal definition is:

$f(n) \in O(g(n))$ if and only if there is some $c \in \mathbb{R}^+$ and some $n_0 \in \mathbb{R}$ such that for all $n > n_0$, $f(n) \leq c * g(n)$.

That's a bit formal and possibly confusing, so let's look at the intuition. Here's a diagram, courtesy of Wikipedia.



To the left of n_0, the functions can do anything. To its right, $c * g(n)$ is always greater than or equal to $f(n)$.

Something that's very important to note about this diagram is that there are infinitely many functions that are in $O(g(n))$: If $f(n) \in O(g(n))$, then $\frac{1}{2}f(n) \in O(g(n))$ and $\frac{1}{4}f(n) \in O(g(n))$ and $2f(n) \in O(g(n))$. In general, for any constant k, $k * f(n) \in O(g(n))$. In problem 3, we prove this.

Something that will come up often with big-O is the idea of a *tight* bound on the runtime of a function.

It's technically correct to say that binary search, which takes around $\log(n)$ steps on an array of length n, is $O(n!)$, since $n! > \log(n)$ for all $n > 0$ but it's not very useful. If we ask for a *tight* bound, we want the closest bound you can give. For binary search, $O(\log(n))$ is a tight bound because no function that grows more slowly than $\log(n)$ provides a correct upper bound for binary search.

Unless we specify otherwise, we want a tight bound!

Checkpoint 0

Rank these big-O sets from left to right such that every big-O is a subset of everything to the right of it. (For instance, $O(n)$ goes farther to the left than $O(n!)$ because $O(n) \subset O(n!)$.) If two sets are the same, put them on top of each other.

$O(n!)$ $O(n)$ $O(4)$ $O(n\log(n))$ $O(4n+3)$ $O(n^2+20000n+3)$ $O(1)$ $O(n^2)$ $O(2^n)$
$O(\log(n))$ $O(\log^2(n))$ $O(\log(\log(n)))$

Solution:

$O(4)$ $O(\log(\log(n)))$ $O(\log(n))$ $O(\log^2(n))$ $O(n)$ $O(n\log(n))$ $O(n^2+20000n+3)$ $O(2^n)$ $O(n!)$
$O(1)$ $O(4n+3)$ $O(n^2)$

Checkpoint 1

Using the formal definition of big-O, prove that $n^3 + 300n^2 \in O(n^3)$.

Solution: $n^3 + 300n^2 \leq n^3 + 300n^3$ for all $n > 1$. $n^3 + 300n^3 = 301n^3$. So, for all $n > 1$, $n^3 + 300n^2 \leq 301n^3$. We have $n_0 = 1$, $c = 301$ if we want to plug back in to the formal definition.

Unit testing

It's really important to always test your code as you write it. Why, you ask? Well, it's effectively impossible to write bug-free code all of the time. However, you want your code to work. So, testing your code is essential to help you find and eliminate bugs.

Why should you do it as you write it? The sooner you catch a bug, the easier it will be to fix. If you wait a long time between writing code and testing it, you might not remember exactly how your code is supposed to work. Not understanding exactly how your code is supposed to work makes debugging much harder.

In addition, if you write a buggy function `foo` and then write a function `bar` that calls `foo`, you might see incorrect behavior in `bar` that is actually caused by the bug in `foo`. This kind of situation would make tracking down the actual source of the bug far more difficult than it has to be.

So, unit test your program! If you test just one function at a time, it'll be far easier to determine where the problem in your program lies.

Here are some major important points about unit testing.

- Use unit tests to look at edge cases!
  - Edge cases are inputs that the specification for your function allows but that might be tricky to handle. They're a common source of bugs because you may need to handle edge case inputs differently from normal cases.
  - Some common edge cases with `int`s are `int_min()`, `int_max()`, 0, -1, and 1. (These are not all of the cases! Depending on what function you are writing there may be other inputs that are edge cases.)
  - For arrays, some edge cases you can run into are the empty array, an array of length 1, and very long arrays. Again, these aren't all of the edge cases — what the edge cases are depends on what the function you wrote is.
- Use unit tests to help you narrow down where exactly bugs in your program lie.
  - If all that you know is that your program produces the incorrect result, it's essentially impossible to debug it. If you write good unit tests for all of your functions, you'll be able to run those and narrow down the bug to specific function/functions, thus making your code far easier to debug.
  - Related to this, you should write tests that exercise all of your code: If you have an `if` statement with two branches, make sure you have a test for each branch. If you don't, one of them might be incorrect and you'd never know it. (Or you'd have a hard time tracking it down if you think whatever bug there is comes from somewhere else.)

- Use unit tests to make sure your contracts pass when they should pass and fail when they should fail
  - Contracts are very useful, but only if they're correct. Incorrect contracts could lead you to thinking that part of your code is wrong when it actually is right, or vice versa. If you test your contracts, you'll be able to make sure that they are correct and won't mislead you.

Taking time to write good tests can save massive amounts of debugging time, since good tests will tell you exactly where in your code the bug lies.

When testing your code, you should start with the assumption that it's incorrect for some input, and that you want to find that input. It's critical to think about corner cases and edge cases — those are things that are out of the ordinary, but are allowed by the preconditions of your function.

If you're trying to test code, it's really important to come at it with the attitude that you want to break it. Pretend your worst enemy wrote the code, and that you want to show them all of the problems with it to get them back for that thing they did, even the pedantic weird cases that probably won't come up in practice. (This is necessary because in the real world you will have to deal with edge cases: if you don't, your code will break for at least some people [but probably thousands], possibly making whatever you wrote unusable for them.)

Checkpoint 2

Using the formal definition of big-O, prove that if $f(n) \in O(g(n))$, then $k * f(n) \in O(g(n))$ for $k > 0$.

One interesting consequence of this is that $O(\log_i(n)) = O(\log_j(n))$ for all i and j (as long as they're both greater than 1), because of the change of base formula:

$$\log_i(n) = \frac{\log_j(n)}{\log(i)}$$

But $\frac{1}{\log(i)}$ is just a constant! So, it doesn't matter what base we use for logarithms in big-O notation.

Solution: Since $f(n) \in O(g(n))$, we know that there exist some $n_0 \in \mathbb{R}$ and $c \in \mathbb{R}^+$ such that $f(n) \leq c * g(n)$ for all $n > n_0$.

We can multiply both sides by k to obtain $k * f(n) \leq k * c * g(n)$ for all $n > n_0$.

So, if we set $c_1 = k*c$, then we know that $k*f(n) \leq c_1*g(n)$ for all $n > n_0$. Thus, $k*f(n) \in O(g(n))$.