Big O

Margaret M. Fleck

12 October 2009

This lecture introduces asymptotic analysis of function growth and big-O notation.

1 Announcements

There's a quiz coming up a week from Wednesday (21 October). Similar in format to the first quiz. Expect study materials later this week, tell me about conflicts, etc.

The student ACM chapter's big annual "Reflections/Projections" conference is this Friday through Sunday.

This week and next week, we'll be talking about how analyze how fast simple computer programs run. This week, we'll be doing some of the basic techniques required (big-O analysis and solving recurrences). Next week, we'll get to the actual programs.

2 Running times of programs

An important aspect of designing a computer programs is figuring out how well it run, in a range of likely situations. Designers need to estimate how fast it will run, how much memory it will require, how reliable it will be, and so forth. In this class, we'll concentrate on speed issues.

Designers for certain small platforms sometimes develop very detailed models of running time, because this is critical for making complex applications work with limited resources. E.g. making God of War run on your Iphone. However, such programming design is increasingly rare, because

computers are getting fast enough to run most programs without hand optimization.

More typically, the designer has to analyze the behavior of a large C or Java program. It's not feasible to figure out exactly how long such a program will take. The transformation from standard programming languages to machine code is way too complicated. Only rare programmers have a clear grasp of what happens within the C or Java compiler. Moreover, a very detailed analysis for one computer system won't translate to another programming language, another hardware platform, or a computer purchased a couple years in the future. It's more useful to develop an analysis that abstracts away from unimportant details, so that it will be portable and durable.

This abstraction process has two key components:

- Ignore multiplicative constants
- Ignore behavior on small inputs, concentrating on how well programs handle large inputs. (Aka asymptotic analysis.)

Multiplicative constants are extremely sensitive to details of the implementation, hardware platform, etc.

Behavior on small inputs is ignored, because programs typically run fast enough on small test cases. Or will do so soon, as computers become faster and faster. Hard-to-address problems more often arise when a program's use expands to larger examples. For example, a small database program developed for a community college might have trouble coping if deployed to handle (say) all registration records for U. Illinois.

3 Function growth: the ideas

So, suppose that you model the running time of a program as a function $F(n)$, where n is some measure of the size of the input problem. E.g. n might be the number of entries in a database application. For a numerical program, n might be the magnitude or the number of digits in an input number. Then, to compare the running times of two programs, we need to compare the growth rates of the two running time functions.

So, suppose we have two functions f and g, whose inputs and outputs are real numbers. Which one has "bigger outputs"?

Suppose that $f(x) = x$ and $g(x) = x^2$. For small positive inputs, x^2 is smaller. For the input 1, they have the same value, and then g gets bigger and rapidly diverges to become much larger than f. We'd like to say that g is "bigger," becausee it has bigger outputs for large inputs.

Because we are only interested in the running times of algorithms, we'll only consider behavior on positive inputs. And we'll only worry about functions whose output values are positive. Or whose output values become positive as the input value gets big, e.g. the log function.

Because we don't care about constant multipliers, we'll consider functions such as $3x^2$, $47x^2$, and $0.03x^2$ to all grow at the same rate. Similarly, functions such as $3x$, $47x$, and $0.03x$ will be treated as growing at the same, slower, rate. The functions in each group don't all have the same slope, but their graphs have the same shape as they head off towards infinity. That's the right level of approximation for analyzing computer programs.

Finally, when a function is the sum of faster and slower-growing terms, we'll only be interested in the faster-growing term. For example, $0.3x^2 + 7x + 105$ will be treated as equivalent to x^2. As the input x gets large, the behavior of the function is dominated by the term with the fastest growth (the first term in this case).

4 Primitive functions

Let's look at some basic functions and try to put them into growth order.

Any constant function grows more slowly than a linear function (i.e. because a constant function doesn't grow!). A linear polynomial grows more slowly than a quadratic. For large numbers, a third-order polynomial grows faster than a quadratic.

Earlier in the term (as an example of an induction proof), we showed that $2^n \leq n!$ for every integer $n \geq 4$. Informally, this is true because 2^n and $n!$ are each the product of n terms. For 2^n, they are all 2. For $n!$ they are the first n integers, and all but the first two of these are bigger than 2. Although we only proved this inequality for integer inputs, you're probably prepared to believe that it also holds for all real inputs ≥ 4.

In a similar way, you can use induction to show that $n^2 < 2^n$ for any integer $n \geq 4$. And, in general, for any exponent k, you can show that $n^k < 2^n$ for any n above some suitable lower bound. And, again, the intermediate real input values follow the same pattern. You're probably familiar with how

fast exponentials grow. There's a famous story about a judge imposing a doubling-fine on a borough of New York, for ignoring the judge's orders. It took the borough officials a few days to realize that this was serious bad news, at which point a settlement was reached.

So, 2^n grows faster than any polynomial in n, and $n!$ grows yet faster. If we use 1 as our sample constant function, we can summarize these facts as:

$$1 \prec n \prec n^2 \prec n^3 \ldots \prec 2^n \prec n!$$

I've used curly $\prec$ because this ordering isn't standard algebraic $\leq$. The ordering only works when n is large enough.

For the purpose of designing computer programs, only the first three of these running times are actually good news. Third-order polynomials already grow too fast for most applications, if you expect inputs of non-trivial size. Exponential algorithms are only worth running on extremely tiny inputs, and are frequently replaced by faster algorithms (e.g. using statistical sampling) that return approximate results.

Now, let's look at slow-growing functions, i.e. functions that might be the running times of efficient programs. Next week, we'll see that algorithms for finding entries in large sets often have running times proportional to $\log n$. If you draw the log function and ignore its strange values for inputs smaller than 1, you'll see that it grows, but much more slowly than n.

Algorithms for sorting a list of numbers (as we'll see) have running times that grow like $n \log n$. If n is large enough, $1 < \log n < n$. So $n < n \log n < n^2$. We can summarize these relationships as:

$$1 \prec \log n \prec n \prec n \log n \prec n^2$$

It's well worth memorizing the relative orderings of these basic functions, since you'll see them again and again in this and future CS classes.

5 The formal definition

Let's write out the formal definition. Suppose that f and g are functions whose domain and co-domain are subsets of the real numbers. Then $f(x)$ is $O(g(x))$ (read "big-O of g) if and only if

> There are real numbers c and k such that $|f(x)| \leq c|g(x)|$ for every $x \geq k$.

The constant c in the equation models the fact that we don't care about multiplicative constants in comparing functions. The restriction that the equation only holds for $x \geq k$ models the fact that we don't care about the behavior of the functions on small input values.

Common unimportant variations on this definition involve requiring f and g to have positive output values and dropping the absolute value signs, or requiring that $x > k$ rather than $x \geq k$ at the end of the definition.

So, for example, $3x^2$ is $O(2^x)$. $3x^2$ is also $O(x^2)$. But $3x^2$ is not $O(x)$. So this big-O relationship is an ordering relation similar to $\leq$, i.e. it includes equality.

6 Applying the definition

To show that a big-O relationship holds, we need to produce suitable values for c and k. For any particular big-O relationship, there are a wide range of possible choices. First, how you pick the multiplier c affects where the functions will cross each other and, therefore, what your lower bound k can be. Second, there is no need to minimize c and k. Since you are just demonstrating existence of suitable c and k, it's entirely appropriate to use overkill values.

For example, to show that $3x$ is $O(x^2)$, we can pick $c = 3$ and $k = 1$. Then $3x \leq cx^2$ for every $x \geq k$ translates into $3x \leq 3x^2$ for every $x \geq 1$, which is clearly true. But we could have also picked $c = 100$ and $k = 100$.

Overkill seems less elegant, but it's easier to confirm that your chosen values work properly, especially in situations like exams. Moreover, slightly overlarge values are often more convincing to the reader, because the reader can more easily see that they do work.

To take a more complex example, let's show that $3x^2 + 7x + 2$ is $O(x^2)$. If we pick $c = 3$, then our equation would look like $3x^2 + 7x + 2 \leq 3x^2$. This clearly won't work for large x.

So let's try $c = 4$. Then we need to find a lower bound on x that makes $3x^2 + 7x + 2 \leq 4x^2$ true. To do this, we need to force $7x + 2 \leq x^2$. This will be true if x is big, e.g. ≥ 100. So we can choose $k = 100$.

The above is a bit of a mess, rather like what you might have on your scratch paper. To write out a formal proof in logical order, we need to reverse our steps. In our proof, we pull k and c out of thin air and show carefully that our equation works.

Proof: Consider $c = 4$ and $k = 100$. Then for any $x \geq k$, $x^2 \geq 100x \geq 7x + 2$. Therefore, for any $x \geq k$, $3x^2 + 7x + 2 \leq 3x^2 + x^2 = 4x^2 = cx^2$. So $3x^2 + 7x + 2$ is $O(x^2)$.

7 More notation

Very, very annoyingly, for historical reasons, the statement $f(x)$ is $O(g(x))$ is often written as $f(x) = O(g(x))$. This looks like a sort of equality, but it isn't. The notation is misleading. Sadly, such notation is very common in computer science and you just have to learn that it is representing a type of inequality.

For numbers, we have a set of relations: $\leq$, $\geq$, and $=$. We have three similar relationships in big-O notation. We've seen that O is the analog of $\leq$. Using it, we can define the two other relationships:

- $g(x)$ is $\Omega(f(x))$ if and only if $f(x)$ is $O(g(x))$. (I.e. it's like $\geq$.)
- $f(x)$ is $\Theta(g(x))$ if and only if $g(x)$ is $O(f(x))$ and $f(x)$ is $O(g(x))$. (I.e. it's like equality.)

In other classes, you may see $O(f(x))$ used to indicate a tight bound, i.e. a bound that's as low as possible, i.e. equal rates of growth. This is quite common outside theory classes. Strictly speaking, however, this is not correct. In this class, use Θ when you mean to say that two functions grow at the same rate or when you mean to give a tight bound.