Big O, Recurrences

Margaret M. Fleck

10 March 2010

This lecture finishes discussion of big-O notation and starts talking about solving recurrences. This material is in sections 3.2 and 7.1 of Rosen.

1 Announcements

There's a quiz coming next Wednesday (15 March). Study materials will be posted soon.

2 Recap big-O

Last class, we saw the idea of asymptotic analysis, in which functions (e.g. the running times of computer programs) are classified on the basis of their behavior for large inputs, ignoring multiplicative constants.

We saw that a function f is $O(g)$ if f grows no faster than g. So $3x$ is $O(x^2)$. $3x$ also $O(x)$, since we don't care about the constant 3. But $3x^2$ isn't $O(x)$, because $3x^2$ grows faster than x when x gets large.

We saw the following big-O ordering for selected basic functions:

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

We also saw the formal definition for big-O notation. 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 $x \geq k$.

And then

- $g(x)$ 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 $O(g(x))$ or (alternatively, if $g(x)$ is $O(f(x))$ and $g(x)$ is $\Omega(g(x))$. (Θ is like equality.)

For example, x^3 is $\Omega(3x^2)$ and $\Omega(x^3)$, because x^3 grows at least as fast as both of these functions.

x^3 is also $O(x^3)$, so x^3 is $\Theta(x^3)$. But x^3 is not $O(3x^2)$, so it's not $\Theta(3x^2)$.

3 Example big-O proof

Show that $3x^2 + 8x \log x$ is $O(x^2)$.

Notice that $x \log x \leq x^2$ for any $x \geq 1$. So $3x^2 + 8x \log x \leq 11x^2$. So if we set $c = 11$ and $k = 1$, our definition of big-O is satisfied.

Writing this out neatly, we get:

> Consider $c = 11$ and $k = 1$. Suppose that $x \geq k$. Then $x \geq 1$. So $\log x \leq x$. So (since x is positive), $x \log x \leq x^2$. So then $3x^2 + 8x \log x \leq 11x^2 = cx$, which is what we needed to show.

4 Sample disproof

Claim: x^3 is not $O(7x^2)$.

> Proof by contradiction. Suppose x^3 were $O(7x^2)$. Then there are c and k such that $x^3 \leq c7x^2$ for every $x \geq k$. But $x^3 \leq c7x^2$ implies that $x \leq 7c$. But this fails for values of x that are greater than $7c$. So we have a contradiction.

5 More abstract example

Let's do a more abstract proof about big-O relationships. To keep things simple, we'll assume that all functions return positive outputs. (It's easy

to upgrade the proof to add absolute values and thus handle other sorts of functions.)

Claim: Suppose $f(x)$ is $O(g(x))$ and $p(x)$ is $O(r(x))$, then $f(x)p(x)$ is $O(g(x)r(x))$.

> Proof: Suppose $f(x)$ is $O(g(x))$ and $p(x)$ is $O(r(x))$. Then (definition of big-O), there are positive real numbers c, k, c', k' such that $f(x) \leq cg(x)$ for all $x \geq k$ and $p(x) \leq c'r(x)$ for all $x \geq k'$.
>
> Then, for every $x \geq \max(k, k')$, $f(x)p(x) \leq cc'g(x)r(x)$. If we set $K = \max(k, k')$ and $C = cc'$ then, we have $f(x)p(x) \leq Cg(x)r(x)$, for all $x \geq K$. So $f(x)p(x)$ is $O(g(x)r(x))$.

Suppose $f(x)$ is $O(g(x))$ and $p(x)$ is $O(r(x))$. Show that $f(x) + p(x)$ is $O(g(x) + r(x))$

We didn't do the following proof in class. It's here in case you are curious.

> Proof: Suppose $f(x)$ is $O(g(x))$ and $p(x)$ is $O(r(x))$. Then (definition of big-O), there are positive real numbers c, k, c', k' such that $f(x) \leq cg(x)$ for all $x \geq k$ and $p(x) \leq c'r(x)$ for all $x \geq k'$.
>
> Then, for every $x \geq \max(k, k')$, $f(x) + p(x) \leq cg(x) + c'r(x)$. Let $C = \max(c, c')$. Then $f(x) + p(x) \leq C(g(x) + r(x))$. So we have that $f(x) + p(x)$ is $O(g(x) + r(x))$.

It's often the case that one of the two terms in the righthand sum dominates the other. For example, if $g(x)$ is $O(r(x))$, then we can simplify the conclusion to: $f(x) + p(x)$ is $O(r(x))$.

6 Recurrences

We've seen recursively defined numerical functions, aka recurrences, last week. For example, we might have a function $T : \mathbb{N} \to \mathbb{Z}$ defined by

- $T(0) = 1$
- $T(n) = T(n-1) + 3n$ (for $n \geq 1$)

If a recursively-defined function has an equivalent closed form (i.e. a formula that doesn't use recursion), this is extremely useful for understanding the function's behavior.

Sometimes you can just look at a recursive function definition and guess the right closed form. Or you might work out the values of the function for the first few input numbers and then guess the right closed form. This method is called "inspection." Sometimes it works; often it doesn't, especially for beginners.

In this course, we'll see a couple techniques for attacking recurrences that you can't solve by inspection. One is called "unrolling" and the other involves inspecting a tree representation of the recurrence. You'll see other techniques in later courses, most notably algorithms.

Once you have the correct closed form, you can prove it's correct using induction. This is easy once you have had some practice using induction. (It probably doesn't seem easy to you right this minute, because you are still working on the first induction problem set.)

7 Unrolling

The idea behind unrolling is to substitute a recurrence into itself, so as to re-express $T(n)$ in terms of $T(n-2)$ rather than $T(n-1)$. We keep doing this, expressing $T(n)$ in terms of the value of T for smaller and smaller inputs, until we can see the pattern required to express $T(n)$ in terms of n and $T(0)$.

Let's do this to our example function T above.

$$\begin{aligned}
T(n) &= T(n-1) + 3n \\
&= (T(n-2) + 3(n-1)) + 3n \\
&= (T(n-3) + 3(n-2)) + 3(n-1) + 3n \\
&= (T(n-4) + 3(n-3)) + 3(n-2) + 3(n-1) + 3n \\
&\dots \\
&= T(n-k) + 3(n-k+1) + \ldots + 3n
\end{aligned}$$

The first few lines of this are mechanical substitution. To get to the last line, you have to imagine what the pattern looks like after k substitutions.

Now, determine when the call to T hits the base case. That is when $n - k = 0$, which is when $n = k$. Substituting this value for k into our equation, we get:

$$\begin{aligned} T(n) &= T(n-k) + 3(n-k+1) + \ldots + 3n \\ &= T(0) + 3 \cdot 1 + \ldots + 3n \end{aligned}$$

We can now compress this using summation notation. Since the key summation is a familiar one, we can solve for a relatively simple closed form:

$$\begin{aligned} T(n) &= T(0) + 3 + 6 + \ldots + 3(n-3) + 3(n-2) + 3(n-1) + 3n \\ &= T(0) + \sum_{k=1}^{n} 3k \\ &= 1 + \sum_{k=1}^{n} 3k \\ &= 1 + 3\sum_{k=1}^{n} k \\ &= 1 + 3\frac{n(n+1)}{2} \\ &= O(n^2) \end{aligned}$$

Sometimes we need the exact solution (the second to last line). For analyzing algorithm running times, we often just the big-O version of the solution (last line).