Recurrences

Margaret M. Fleck

15 March 2010

This lecture does more examples of unrolling recurrences and shows how to use recursion trees to analyze divide-and-conquer recurrences. We'll emphasize finding big-O solutions rather than exact solutions. This material is in section 7.1 and 7.3 of Rosen. However, Rosen takes a slightly more abstract/general approach to divide-and-conquer recurrences in 7.3, so look at that section only if you are curious.

1 Announcements

There's a quiz coming Wednesday (17 March). Study materials are available on the web.

There has been a rash of laptop thefts in/near Siebel. Keep careful watch on your laptop. Also, I need to close the door to my office on short errands (e.g. to pick up a printout). So, if you find my door closed, don't immediately conclude I'm far away.

2 Recap

Last class, we saw how to solve a recurrence using a technique called "unrolling." Here's another example. Suppose that you deposit $10,000 and your bank gives you 11% interest each year (which isn't very likely, is it?). The function M for how much money you'll have in n years is given by

- $M(0) = 10000$
- $M(n) = 1.11M(n-1)$.

Unrolling this recurrence, we get

$$\begin{aligned} M(n) &= 1.11M(n-1) \\ &= 1.11(1.11M(n-2)) \\ &= 1.11(1.11(1.11M(n-3))) \\ &\cdots \\ &= (1.11)^n M(0) \\ &= (1.11)^n (10,000) \\ M(30) &= 228,922.97 \end{aligned}$$

3 A harder example

Let's do a more complex example:

- $T(1) = 1$
- $T(n) = 2T(n-1) + 3$

A note on terminology: If we call this a "recursive definition," we usually refer to $T(1) = 1$ as the "base case." If we call this a "recurrence relation," we usually call $T(1) = 1$ the "initial condition." This is just another of those annoying synonyms that come about because different subfields adopt slightly different terminology.

If we unroll this function, we get

$$\begin{aligned} T(n) &= 2T(n-1) + 3 \\ &= 2(2T(n-2) + 3) + 3 \\ &= 2(2(2T(n-3) + 3) + 3) + 3 \\ &= 2^3 T(n-3) + 2^2 \cdot 3 + 2 \cdot 3 + 3 \\ &= 2^4 T(n-4) + 2^3 \cdot 3 + 2^2 \cdot 3 + 2 \cdot 3 + 3 \\ &\cdots \\ &= 2^i T(n-i) + \sum_{k=0}^{i-1} 3(2^k) \end{aligned}$$

We reach the base case when $n - i = 1$ That is, when $i = n - 1$. So, substituting $i = n - 1$ and $T(1) = 1$ into the above formula, we get:

$$T(n) = 2^{n-1} + \sum_{k=0}^{n-2} 3(2^k) = 2^{n-1} + 3\sum_{k=0}^{n-2}(2^k) = 2^{n-1} + 3(2^{n-1} - 1) = 4(2^{n-1}) - 3 = 2^{n+1} - 3$$

In computer science, it's often the case that we don't need an exact solution for our recurrence but only its big-O description. If so, we can summarize the growth of our function T as: T is $O(2^n)$. We don't care about the constant term because it's dominated by the exponential. Nor do we care about the exponent in 2^{n-1} because adding/subtracting constants to the exponent just multiplies the whole thing by a constant.

4 Divide and conquer

Many important algorithms involve dividing a big problem of size n into a sub-problems, each of size n/b. This general method is called "divide and conquer." Analyzing such algorithms involves recurrences that look like:

$$S(n) = aS(n/b) + f(n)$$

The term $f(n)$ is the work involved in dividing up the big problem and/or merging together the solutions for the smaller problems.

Such recurrences can have a wide range of closed forms, depending on what a, b, and f are. This term, we'll look at some of the more important special cases.

For example, if we break up our problem into two half-size problems, and the dividing/merging takes time proportional to the size of the problem, we get the recurrence

$$S(n) = 2S(n/2) + n$$

Let's suppose the initial condition is $S(1) = C$, where C is a constant.

Unrolling this, we get

$$\begin{aligned} S(n) &= 2S(n/2) + n \\ &= 2(2S(n/4) + n/2) + n \\ &= 4S(n/4) + n + n \\ &= 8S(n/8) + n + n + n \\ &\cdots \\ &= 2^i S(\frac{n}{2^i}) + in \end{aligned}$$

When do we hit the base case? That is, when does $\frac{n}{2^i} = 1$? Well, as we've stated things literally, it might not. This will only work out neatly if n is a power of 2. And, in fact, the entire recursive definition of S won't work right for other values of n.

To make such a recurrence work on inputs that aren't powers of two, we would need to wrap the floor or the ceiling function around the fraction $n/2$ in the recursive call. Analyzing functions involving floors or ceilings is a big pain.

Fortunately, if we care only about the big-O behavior of a recurrence, not a precise solution to the recurrence, it's normally safe to analyze the recurrence's behavior for inputs that are powers of two (or powers of whatever the divisor is, if it isn't two for some other recurrence).

So, assuming that n is a power of two, S will hit its base case when $\frac{n}{2^i} = 1$ i.e. when $i = \log n$ (i.e. log base 2, which is the normal convention for algorithms applications). Substituting in this value for i and the base case value $S(1) = C$, we get

$$S(n) = 2^i S(\frac{n}{2^i}) + in = 2^{\log n} C + n \log n = Cn + n \log n$$

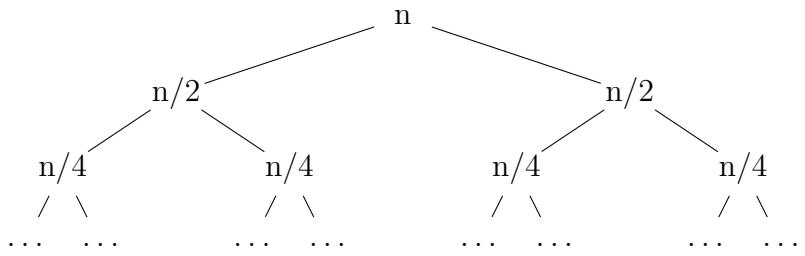
So $S(n)$ is $O(n \log n)$.

Notice that this process could get very messy if we tried to work out the precise details of S and the precise closed-form. It's important to learn how to cut corners when a big-O solution is all that's required.

5 Tree picture

This kind of recurrence can be illustrated with a picture, called a "recursion tree". The top node in the tree represents $S(n)$ and contains everything in

the formula for $S(n)$ **except the recursive calls to** S. The two nodes below it represent two copies of the computation of $S(n/2)$. Again, the value in each node contains the non-recursive part of the formula for computing $S(n/2)$. The value of $S(n)$ is then the sum of the values in all the nodes in the recursion tree.

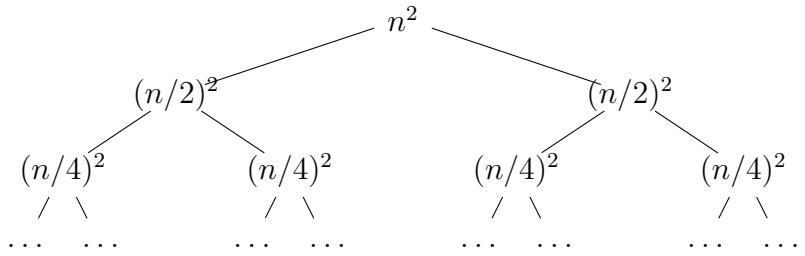


In the tree picture, we ask two questions. First, how high is this tree, i.e. how many levels do we need to expand before we hit the base case $n = 1$? Second, for each level of the tree, what is the sum of the values in all nodes at that level?

In this example, the height of the tree is $O(\log n)$. At each level of the tree, the node values sum to n. So the sum of all the node values in the tree is $O(n \log n)$.

6 Another example

Now, let's suppose that we have the recurrence $P(n) = 2P(n/2) + n^2$. Again, we assume that the base case is that $P(1)$ is some constant. Let's draw its recursion tree:



The height of the tree is again $O(\log n)$. The sums of all nodes at the top level is n^2. The next level down sums to $n^2/2$. And then we have sums: $n^2/4$, $n^2/8$, $n^2/16$, and so forth. So the sum of all nodes at level k is $n^2\frac{1}{2^k}$.

The solution to our recurrence is the sum of all levels in tree, i.e.

$$P(n) = \sum_{k=0}^{\log n} n^2 \frac{1}{2^k} = n^2 \sum_{k=0}^{\log n} \frac{1}{2^k} = n^2(2 - \frac{1}{2^{\log n}}) = n^2(2 - \frac{1}{n}) = 2n^2 - n = O(n^2)$$