Recitation 2 — Recurrences

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2013)

September 4, 2013

1 Announcements

- HW1 is due on Monday, September 9. Hopefully you have all started by now; if not, now would be a good time.
- If you are not able/do not want to use Piazza to contact the course staff, you may send email to

  `15210-staff@lists.andrew.cmu.edu.`

- Questions from lecture or homework?

2 Big-O, Big-Omega, and Big-Theta

In lecture yesterday, we reviewed Big-O notation for describing classes of functions. Today, we'll review some other useful classes of functions.

Remember the definition of Big-O from lecture:

Definition 2.1. $f \in O(g)$ if there exist $c > 0$ and $n_0 > 0$ such that $f(n) \leq c \cdot g(n)$ for all $n > n_0$.

Note that if $f \in O(g)$, g is asymptotically an "upper bound" of f. However, it may be a very weak upper bound. For example, you can easily show using the definition that $n \in O(n^n)$. This knowledge is not particularly useful, however. In general, we are interested in *tight* bounds (though technically correct, you would not get much credit on an exam for saying that mergesort does $O(n^5)$ work.) Fortunately, there are other classes of functions that can formalize this idea.

Definition 2.2. $f \in \Omega(g)$ if there exist $c > 0$ and $n_0 > 0$ such that $f(n) \geq c \cdot g(n)$ for all $n > n_0$.

Note that Ω ("Big-Omega") gives essentially the opposite of O: if $f \in \Omega(g)$, then f grows asymptotically *at least as fast* as g. This still does not give us the tight bounds we want, however. For this, we combine the two definitions.

Definition 2.3. $f \in \Theta(g)$ if there exist $c_1 > 0$, $c_2 > 0$ and $n_0 > 0$ such that $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n > n_0$.

Intuitively, Θ ("Big-Theta") says that f grows at least as fast and at least as slow as g, that is, they grow at approximately the same rate. This is a tight bound. Note that $f \in \Theta(g)$ if and only if $f \in O(g)$ and $f \in \Omega(g)$.

We will often be sloppy and use Big-O notation to give tight bounds. For example, when we say that the work of mergesort is $O(n \log n)$, we mean that it is $\Theta(n \log n)$. Make sure you understand this distinction and ask us if what we mean isn't clear.

Here are a few examples of simple functions and their asymptotic analyses.

f	g	Relations
n	$8n^2$	$f \in O(g)$
n^3	$12n^3 + 4n^2$	$f \in O(g), f \in \Omega(g), f \in \Theta(g)$
$2^{\log n}$	n	$f \in O(g), f \in \Omega(g), f \in \Theta(g)$
$n!$	$n^2 2^n$	$f \in \Omega(g)$

3 Substitution Method

Yesterday in lecture we went over the tree method, a good way to get started when solving a recurrence. We will review the tree method later. However, first we will look at another method: the substitution method. This is a good way of proving to yourself (and us) that the complexity class you've found is correct.

The substitution method uses a technique that may sound unusual, but will become easy with practice: guess an answer and prove it correct. In particular, let's start by solving a recurrence that should be familiar to all of you as a warmup:

$$W(n) = 2W(n/2) + O(n)$$

To solve this recurrence using the substitution method, we guess a function $g(n)$ and prove (by strong induction on n) the following theorem, which is equivalent to $W(n) \in O(g(n))$.

Theorem 3.1. *Let a constant c be given. If $W(n) \leq 2W(n/2) + c \cdot n$ for $n > 1$ and $W(1) \leq c$, then there exist constants k_1 and k_2 such that*

$$W(n) \leq k_1 \cdot g(n) + k_2$$

Because Big-O notation hides many details (that's why it's useful) and because this technique relies on guessing an answer, it is often tempting to be sloppy in writing these proofs. Occasionally, this can lead us to fool ourselves into believing an incorrect proof.

3.1 What can go wrong

Let's attempt to solve the above recurrence using an insufficiently formal application of the substitution method. Suppose $W(1) \in O(1)$. We claim that $W(n) \in O(n)$. Is this true? Let's try to prove it by induction.

Base case: Given.

Inductive hypothesis: For all $i < n$, $W(i) \in O(i)$.

Inductive case:

$$\begin{aligned} W(n) &= 2W(n/2) + O(n) \\ &= 2\left[O(n/2)\right] + O(n) \\ &\leq 2O(n) + O(n) \\ &= O(n) \end{aligned}$$

So, we proved that $W(n) \in O(n)$. Or did we?

3.2 A Closer Look

What went wrong? Recall the definition of Big-O.

Using Definition 2.1 we can prove the following lemma:

Lemma 3.2. *If $f \in O(n)$, there exist constants k_1, k_2 so that $f(n) \leq k_1 n + k_2, n \geq 0$*

Proof. By the definition of Big-O, there exist constants c and n_0 such that $f(n) \leq c \cdot n$ for $n > n_0$. Then $k_1 = c, k_2 = \max(f(i) : 0 \leq i < n_0)$ works. □

So, when we say $W'(n) \in O(n)$, we mean that there exist some n_0, c such that for all $n > n_0$, $W'(n) \leq c \cdot n$, and want to show that there exist constants k_1 and k_2 such that $W'(n) \leq k_1 n + k_2$ for all $n \geq 0$. This isn't the case in our proof of the inductive case:

$$\begin{aligned} W(n) &\leq 2W(n/2) + c \cdot n \\ &\leq 2\left[k_1 n/2 + k_2\right] + c \cdot n \\ &= (k_1 + c)n + 2k_2 \\ &\not\leq k_1 n + k_2 \end{aligned}$$

Do you see what went wrong?

Since $c > 0$, there is no fixed choice of c that makes this proof go through.

3.3 Doing It Correctly

Now let's try correctly proving $W(n) \in O(n \log n)$.

Proof. Let $k_1 = 2c$ and $k_2 = c$ (we need to show that k_1 and k_2 exist, and so guessing ones that work is perfectly valid.) For the base case ($n = 1$), we check that $W(1) \leq c = k_2$. For the inductive step ($n > 1$), we assume that

$$W(n/2) \leq k_1 \cdot \tfrac{n}{2} \log(\tfrac{n}{2}) + k_2,$$

(this is where we need strong induction, so we can assume the theorem is true for any $i < n$) and we'll show that $W(n) \leq k_1 \cdot n \log n + k_2$. To show this, we substitute an upper bound for

$W(n/2)$ from our assumption into the recurrence, yielding

$$\begin{aligned} W(n) &\leq 2W(n/2) + c \cdot n \\ &\leq 2(k_1 \cdot \tfrac{n}{2} \log(\tfrac{n}{2}) + k_2) + c \cdot n \\ &= k_1 n(\log n - 1) + 2k_2 + c \cdot n \\ &= k_1 n \log n + k_2 + (c \cdot n + k_2 - k_1 \cdot n) \\ &\leq k_1 n \log n + k_2, \end{aligned}$$

where the final step follows because $c \cdot n + k_2 - k_1 \cdot n \leq 0$ as long as $n > 1$. □

3.4 Big-Theta

In fact, the bound we have just found is tight. Can we prove this using the substitution method? We first prove another theorem that shows $W(n) \in \Omega(n \log n)$.

Theorem 3.3. *Let a constant c be given. If $W(n) \geq 2W(n/2) + c \cdot n$ for $n > 1$ and $W(1) \geq c$, then there exist constants k_1 and k_2 such that*

$$W(n) \geq k_1 \cdot g(n) + k_2$$

Proof. Let $k_1 = k_2 = c$. We show the inductive step, assuming that

$$W(n/2) \geq k_1 \cdot \tfrac{n}{2} \log(\tfrac{n}{2}) + k_2,$$

We show that $W(n) \geq k_1 \cdot n \log n + k_2$, as before.

$$\begin{aligned} W(n) &\geq 2W(n/2) + c \cdot n \\ &\geq 2(k_1 \cdot \tfrac{n}{2} \log(\tfrac{n}{2}) + k_2) + c \cdot n \\ &= k_1 n(\log n - 1) + 2k_2 + c \cdot n \\ &= k_1 n \log n + k_2 + (c \cdot n + k_2 - k_1 \cdot n) \\ &\geq k_1 n \log n + k_2, \end{aligned}$$

where the final step follows because $c \cdot n + k_2 - k_1 \cdot n \geq 0$ as long as $n > 1$. □

Now that we have shown that $W(n) \in O(n \log n)$ and $W(n) \in \Omega(n \log n)$, we know that $W(n) \in \Theta(n \log n)$, and thus the complexity bound we have given is tight.

3.5 Tree Method

Yesterday in lecture we went over the tree method, which is a good way of getting started solving a recurrence, or maybe getting a guess for the recurrence to use for the substitution method. In order to effectively use the tree method, we often need to solve a geometric series. Recall from math classes the formula for a geometric series of the form $a_0 + a_0 r + a_0 r^2 + ... + a_0 r^n$:

$$a_0 \frac{r^{n+1} - 1}{r - 1}$$

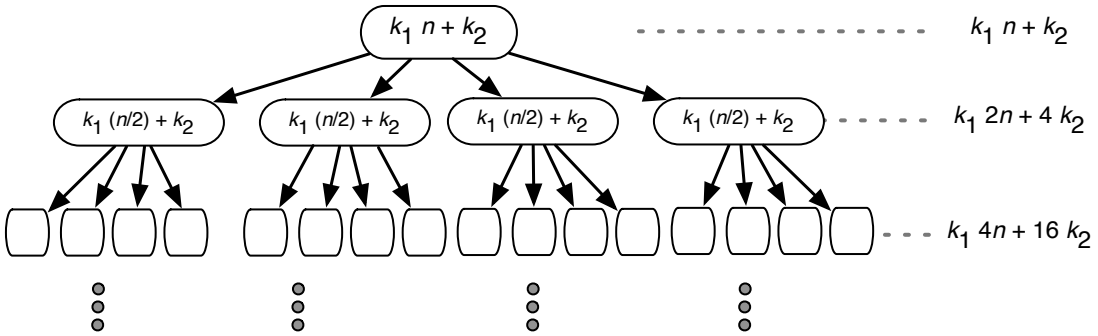
You're likely familiar with special cases of this formula in which $0 < r < 1$, in which case the sum converges even for $n = \infty$:

$$\frac{a_0}{1-r}$$

For example, $1 + \frac{1}{2} + \frac{1}{4} + ... = \frac{1}{1-\frac{1}{2}} = 2$.

Now, let's look at some examples using the tree method.

- For $W(n) = 4W(n/2) + O(n)$, the recursion tree is:



If we were to write out the total cost of the operation, we would have:

$$k_1 n + k_2 + 4(k_1(\frac{n}{2}) + k_2) + ... + 4^{\log n}(k_1(1) + k_2)$$

That is, we have at level i:

Problem Size	$n/2^i$
Node Cost	$\leq k_1(n/2^i) + k_2$
Number of Nodes	4^i

It is often more helpful to write the total cost using summation notation:

$$\Sigma_{i=0}^{\log n} 4^i (k_1(\frac{n}{2^i}) + k_2)$$

$$\Sigma_{i=0}^{\log n} 4^i k_1(\frac{n}{2^i}) + \Sigma_{i=0}^{\log n} 4^i k_2$$

$$\Sigma_{i=0}^{\log n} 2^i k_1(n) + \Sigma_{i=0}^{\log n} 4^i k_2$$

Using the formula for geometric series and log rules,

$$W(n) = k_1 \cdot n\frac{1 - 2^{\log n+1}}{1-2} + k_2\frac{1-4^{\log n+1}}{1-4}$$

$$W(n) = k_1 \cdot nO(n) + O(n^2) \in O(n^2)$$

- For $W(n) = W(3n/4) + O(n)$, we have at level i:

Problem Size	$(3/4)^i n$
Node Cost	$\leq k_1(3/4)^i n + k_2$
Number of Nodes	1

$$W(n) = k_1 \Sigma_{i=0}^{\log n} (3/4)^i n + k_2 \cdot \log n$$
$$W(n) = k_1 \frac{1 - O(1/n)}{1 - 3/4} + k2 \cdot \log n \in O(n)$$

- For $W(n) = 2W(n/2) + O(n)$, we have at level i:

Problem Size	$n/2^i$
Node Cost	$\leq k_1(n/2^i) + k_2$
Number of Nodes	2^i

$$W(n) = k_1 \Sigma_{i=0}^{\log n} 2^i (n/2^i) + k_2 \Sigma_{i=0}^{\log n} 2^i$$
$$W(n) = k_1 n \cdot \log n + k_2 \frac{1 - O(n)}{1 - 2} \in O(n \log n)$$

  Note that, in the first summation, since $r = 1$, we don't use the geometric series formula, but instead multiply the total cost at each level by the number of levels.