Recurrences

We have seen many algorithms thus far. For each one we have tried to express how many basic operations are required as a function of some parameter $n$ which is typically the size of the input e.g. the number of elements in a list.

For algorithms that involve for loops, we can often write the number of operations of the loop component as a power of $n$, which corresponds to the number of nested loops. For example, if we have two nested for loops, each of which run $n$ times, then these loops take time proportional to $n^2$. The sorting algorithms from lecture 6 and the grade school multiplication algorithm are a few examples.

For recursive algorithms, it is less obvious how to express the number of operations as a function of the size of the input. We have given some examples of recursive algorithms in the last few lectures. For these examples and others, we would like to express in a more general way how the time it takes to solve the problem depends on $n$. That is what we will do next and next lecture. In each case, we express a function $t(n)$ in terms of $t(\ldots)$ where the argument depends on $n$ but it is a value smaller than $n$. Such a recursive definition of $t(n)$ is called a recurrence relation.

Example 1: reversing a list

Let $t(n)$ be the time it takes to reverse a list with $n$ elements. Recall how this is done. You remove the first element of the list. Then, you take the remaining $n - 1$ element list and recursively reverse them. Then you add the element you removed to the end of the reversed list.

Each recursive call reduces the problem from size $n$ to size $n - 1$. This suggests a relationship:

$$t(n) = c + t(n - 1)$$

where the constant $c$ is the time it takes in total to remove the first element from a list plus the time it takes to add that same element to the end of a list. We are not saying what $c$ is. All that matters is that it is constant: it doesn’t depend on $n$. (By the way, I am making an assumption here that the first element can be removed in constant time. If we are using an array list, then this is not so.)

To obtain an expression for $t(n)$ that is not recursive, we repeatedly substitute on the right side, as follows:

$$t(n) = c + t(n - 1)$$
$$= c + c + t(n - 2)$$
$$= c + c + c + t(n - 3)$$
$$= \ldots$$
$$= cn + t(0).$$

This method is called backwards substitution. Note $t(0)$ is the base case of the recursion and done in constant time.

Informally, we say that $t(n)$ is $O(n)$ since the time it takes is roughly proportional to $n$. A few lectures from now, we’ll say more formally what we mean by $O(\ )$. 

last updated: 12th Oct, 2017 at 08:37
[ASIDE: One often writes such a recurrence in a slightly simpler way ($c=1$):

$$t(n) = 1 + t(n-1).$$

The idea is that since the constant $c$ has no "units" anyhow, its meaning is unspecified except for the fact that it is constant, so we just treat it as a unit (1) number of instructions.]

**Example 2: sorting a list by finding the minimum element**

Recall the recursive algorithm for sorting which found the smallest element in a list and removed it, then sorted the remaining $n-1$ elements, and finally added the removed element to the front of the list. We express the time taken, using the recurrence:

$$t(n) = c \cdot n + t(n-1).$$

As I discussed in the lecture, we could write the recurrence more precisely as

$$t(n) = c_1 + c_2 \cdot n + t(n-1)$$

since in each recursive call there is some constant $c_1$ amount of work, plus some amount of work $c_1 n$ that depends linearly on the size $n$ of the list in that call. But let’s keep it simple and consider just the first recurrence.

Solving by back substitution:

$$t(n) = c \cdot n + t(n-1)$$

$$= c \cdot n + c \cdot (n-1) + t(n-2)$$

$$= \ldots$$

$$= c \{ n + (n-1) + (n-2) + \cdots + (n-k) \} + t(n-k-1)$$

$$= c \{ n + (n-1) + (n-2) + \cdots + 2 + 1 \} + t(0)$$

$$= \frac{cn(n+1)}{2} + t(0)$$

This is $O(n^2)$ since the largest term here that depends on $n$ is $n^2$.

**Example 3: Tower of Hanoi**

Recall the Tower of Hanoi problem. Let $t(n)$ be the number of disk moves. The recurrence relation is:

$$t(n) = c + 2 \cdot t(n-1).$$

The $c$ on the right side refers to the work that is done with the call to `tower`. There is the single disk move that is done in each call, and there is also some administrative work that is done for each
recursive call (making a stack frame, pushing it on the call stack, and then popping the call stack when the method exits). All this constant time work is bundled together as a single constant $c$. This work is added to a term $2 \, t(n-1)$ which is the time needed for the two recursive calls on the problem of size $n-1$.

Proceeding by back substitution, we get

$$t(n) = c + 2t(n-1)$$
$$= c + 2(c + 2t(n-2))$$
$$= c(1 + 2) + 4t(n-2)$$
$$= c(1 + 2) + 4(c + 2t(n-3))$$
$$= c(1 + 2 + 4) + 8t(n-3)$$
$$= ...$$
$$= c(1 + 2 + 4 + 8 + \cdots + 2^{k-1}) + 2^k \, t(n-k)$$
$$= c(1 + 2 + 4 + 8 + \cdots + 2^{n-1}) + 2^nt(0)$$
$$= c(2^n - 1) + 2^nt(0)$$

where we have used the familiar geometric series (recall lecture 2)

$$\sum_{i=0}^{n-1} x^i = \frac{x^n - 1}{x - 1}$$

for the case that $x = 2$.

Since the largest term that depends on $n$ is $2^n$, we say that $t(n)$ is $O(2^n)$.

**Example 4: binary search**

Recall the recursive binary search algorithm. We assume we have an ordered list of elements, and we would like to find a particular element $e$ in the list. The algorithm computes the mid index and compares the element $e$ to the element at that mid index. The algorithm then recursively calls itself, searching for $e$ either in the lower or upper half of the list. Since the recursive call is on a list that is only half the size, we can express the time using the recurrence:

$$t(n) = c + t(n/2) .$$

We suppose that $n$ is a power of 2, i.e. $n = 2^k$, where $k = \log_2 n$.

$$t(n) = c + t(n/2)$$
$$= c + c + t(n/4)$$
$$= c + c + \cdots + t(n/2^k)$$
$$= c + c + \cdots + c + t(n/n)$$
$$= c \log_2 n + t(1)$$

So we say that binary search is $O(\log_2 n)$ since the largest term that depends on $n$ is $\log_2 n$.
Logarithms: some review, some new

When you signed up for COMP 250, you probably didn’t expect that you would need to think so much about logarithms. Surprise! As we saw last lecture, logarithms are very important in understanding how long different algorithms take. Because you may be rusty on logarithms, here I will review some basic and important properties that you will need to know or at least be familiar with in COMP 250.

Let’s begin with the definition. The logarithm is the inverse function of the exponential. If we consider take the exponential function for some base \( b > 0 \), namely

\[
y = b^x,
\]

then the inverse function

\[
\log_b y \equiv x
\]

is called the logarithm of that base \( b \). Thus,

\[
\log_b(b^x) \equiv x
\]

\[
b^{\log_b y} \equiv y.
\]

Note that this is the definition of the log function, but this definition assumes that \( b^x \) is meaningful. I think you all agree that \( b^x \) is meaningful when \( x \) is an integer, but you are probably not so clear on exactly what it means when \( x \) is not an integer. For the latter case, indeed it is not so clear and you would need to arguments from MATH 242 Real Analysis to say exactly what it means. Alas, those not taking that course will just have to just trust the good people in the math department.

Here are a few basic properties of logs. We assume \( a, b, c > 0 \).

- \( a^{b+c} = a^b a^c \)
- \((a^b)^c = a^{bc}\)
- \(\log_b(a^c) = c \log_b a\)
- \(\log_b(ac) = \log_b a + \log_b c\)

The next two properties compare logarithms of two different bases, that is, \( \log_b x \) versus \( \log_a x \).

- \(\log_b x = (\log_b a)(\log_a x)\) Here is the proof:

\[
\log_b x = \log_b(a^{\log_a x}) = (\log_b x)(\log_b a)
\]

- \(a^{\log_b c} = c^{\log_b a}\) Here is the proof:

\[
(\log_b c)(\log_b a) = (\log_b a)(\log_b c)
\]

and so

\[
\log_b(a^{\log_b c}) = \log_b(c^{\log_b a})
\]

and so we can cancel the \( \log_b \) on both sides and we’re done.