Recurrences

In the last two lectures, we examined the running time $t(n)$ of an algorithm in terms of the input "size" n. In this lecture and the next one, we specifically address recursive algorithms.

Example 1: Factorial

Let $t(n)$ be the time it takes to compute $n!$. You should have an intuition that $t(n)$ is $O(n)$. This is easy to see if you use an iterative algorithm to compute $n!$ since we have a `for` loop which we iterate n times. What about if we use a recursive algorithm?

```
Algorithm: Factorial(n)
  if $n > 1$ then
    return $n*$ Factorial($n - 1$)
  end if
```

The recursive algorithm for computing $n!$ involves a method call and return and a multiplication. These operations can each be done in constant time. Moreover, each method call reduces the problem from size n to size $n - 1$. This suggests a relationship:

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

namely the time it takes to compute $n!$ is some constant plus the time it takes to compute $(n - 1)!$. Such a relationship, in which $t(n)$ is expressed in terms of $t(*)$ where the argument is value smaller than n is called a *recurrence relation*.

Repeatedly substituting on the right side yields:

$$\begin{aligned} t(n) &= 1 + 1 + t(n - 2) \\ &= \ldots \\ &= n - 1 + t(1). \end{aligned}$$

This is called *backwards substitution* because we substitute starting at $t(n)$ and working our way back to $t(1)$. Note $t(1)$ is the base case of the recursion and done in constant time. So, $t(n) = n$ which is obviously $O(n)$.

You may be a bit disturbed by the use of "1" as a constant. I am using "1" to refer to a set of operations that takes a constant time i.e. independent of n. I do this because ultimately I am only interested in asympotic running time and the particular constants will play no role. What if I had written a different constant in the recurrence? For example, what if I tried to distinguish between the constant time c_1 that is taken at each step of the recursion for the multiplication and method call from the constant c_2 that is taken at the base when $n = 1$.

$$t(n) = c_1 + t(n - 1)$$

and

$$t(1) = c_2$$

Solving the recurrence, I would get:

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

which is still $O(n)$, as you can easily verify.

Example 2: Tower of Hanoi

The Tower of Hanoi problem is presented on p. 151 of the GT textbook and was briefly discussed at the end of lecture 10. It is basically a mathematical puzzle. The "tower" consists of three rods, and a number of disks of different sizes which can slide onto any rod. (See "Tower of Hanoi" entry in wikipedia.) We start with the disks stacked in order of size on one rod. The objective of the puzzle is to move the entire stack to another rod, obeying the following rules:

1. Only one disk may be moved at a time.
2. Each move consists of taking the upper disk from one of the rods and sliding it onto another rod, on top of the other disks that may already be present on that rod.
3. No disk may be placed on top of a smaller disk.

Here is the standard recursive algorithm for solving the problem. The three rods are labelled *start*, *finish*, *tmp*.

```
Algorithm: Tower(n,start, finish, tmp)
Input: a number $n$ of disks at a start position
Output: sequence of instructions that move $n$ disks from start to finish
  if $n > 0$ then
    Tower($n-1, start, tmp, finish$)
    print "move the top disk from " start " to " finish
    Tower($n-1, tmp, finish, start$)
  end if
```

Why is the algorithm correct ?

How can we be sure that the algorithm does not violate rule 3 ? We need to prove this. The proof is by induction.
Base case: Rule 3 is obviously obeyed if $n = 1$ since the algorithm simply moves the one disk from *start* to *finish*.
Induction step: Suppose rule 3 is obeyed if $n = k$ (induction hypothesis). We need to show it is obeyed for $n = k + 1$. For $n = k + 1$, the solution has three steps, namely,

- **Tower**$((k + 1) - 1, start, tmp, finish)$
- print "move the top disk from " *start* " to " *finish*
- **Tower**$((k + 1) - 1, tmp, finish, start)$

The two (recursive) calls to **Tower** move $k = k + 1 - 1$ disks each. which obeys rule 3 by the induction hypothesis. The middle step in which we move the biggest disk from *start* to finish also obeys rule 3, since rod *finish* is empty just before that step (because we previously moved all the smaller disks to *tmp*). This completes the proof.

Running time

Here we analyze the running time of the algorithm The recurrence relation is:

$$t(n) = 1 + 2t(n-1)$$

and $t(1) = 1$. The "1" on the right side refers to a constant time needed to do the print statement and to check if $n > 0$. The $2t(n-1)$ is the time needed for the two recursive calls.

Proceeding by back substitution, we get

$$\begin{aligned} t(n) &= 1 + 2t(n-1) \\ &= 1 + 2(1 + 2t(n-2)) = 1 + 2 + 4t(n-2) \\ &= 1 + 2 + 4(1 + 2t(n-3)) \\ &= 1 + 2 + 4 + 8t(n-3) \\ &= 1 + 2 + 4 + 8 + \cdots + 2^{n-1} + 2^n t(0) \\ &= 2^n - 1 + 2^n t(0) \end{aligned}$$

where we use the formula for the geometric series

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

for the case that $a = 2$.

Notice that $t(0)$ is a different constant from "1", since the latter requires both the check whether $n > 0$ and also the print statement. In particular, note that

$$t(n) \ \ is \ \ O(2^n).$$

You may be wondering why I was careful to put a factor "2" in the recurrence equation. I have said that we are ignoring constant factors in the asympototic analysis, so you might have thought we could drop the "2" and write the recurrence as

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

However, notice that the latter recurrence is that same as we saw for computing $n!$, which was $O(n)$ rather than $O(2^n)$. Quite a big difference! What's going on here (you may well ask) ?

To see what's going on, compare the two recurrences:

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

versus

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

The first recurrence says that it takes c extra steps to reduce the size of the problem by 1. This implies that the total number of steps behaves as cn which is $O(n)$. The second recurrence says that it takes c times as many steps to reduce the problem size by 1. This implies that the total number of steps behaves as c^n which is $O(c^n)$. Note the difference.

Notation: floor and ceiling

The recurrence equations we will work with have arguments that are positive integers. If we have a fractional number and we wish to round it down (floor) or up (ceiling) to the nearest integer, then we use the following notation:

$\lfloor x \rfloor$ is the largest integer that is less than or equal to x. It is called the “floor” operator.

$\lceil x \rceil$ is the smallest integer that is greater than or equal to x. It is called the “ceiling” operator.

Example 3: converting decimal to binary

Recall the algorithm for converting a decimal number n to binary. Here we write the algorithm recursively, and use the “floor” operator (instead of the Java integer “/” operator which we used previously).

```
Algorithm: DecimalToBinary(n)
Input: a decimal number $n$
Output: sequence of bits from least to most significant, representing $n$ in binary
  if $n \geq 1$ then
    print $n\%2$
    DecimalToBinary($\lfloor n/2 \rfloor$)
  end if
```

What is the asymptotic running time of this algorithm? We can write a recurrence relation as follows:

$$t(n) = 1 + t(\lfloor n/2 \rfloor).$$

The “floor” operator is a bit annoying, so we bound the recurrence as follows. Let $k = \lceil \lg n \rceil$, where $\lg n \equiv \log_2 n$. That is, k is the smallest integer such that $n \leq 2^k$. In particular,

$$\lfloor n/2 \rfloor \leq 2^{k-1}.$$

Thus,

$$t(n) \leq 1 + t(2^{k-1}).$$

Backwards substitution for $t(2^{k-1})$ gives:

$$t(n) \ \leq \ 1 + t(2^{k-1}) \ \leq \ 1 + 1 + t(2^{k-2}) \ = \ \ldots \ \leq \ k + t(2^0) \ = \ k + 1 + t(0).$$

Since $k \leq \lg n + 1$, it follows that

$$t(n) \text{ is } O(\lg n).$$

Again, note that if I had written a different constant in the recurrence, and tried to distinguish between the constant time c_1 that is taken at each step of the recursion from the constant c_2 that is taken at the base when $n = 0$, then I would end up with

$$t(n) \leq c_1 k + c_2$$

which again is $O(\lg n)$.

Finally, you should not be surprised by the result, if you consider that the number of bits in the binary representation of n is $\lfloor \lg n \rfloor + 1$. A constant number of operations are required for each bit.

n (in decimal)	n (in binary)	floor(lg n) + 1
1	1	1
2	10	2
3	11	2
4	100	3
5	101	3
:		
8	1000	4
9	1001	4
:	:	:
15	1111	4
16	10000	5
:	:	:
31	11111	5
32	100000	6
:	:	: