CS/Math 240: Introduction to Discrete Mathematics 3/1/2011

Lecture 12 : Recurrences

Instructor: Dieter van Melkebeek Scribe: Dalibor Zelený

DRAFT

Last few classes we talked about program correctness. We discussed correctness of non-recursive as well as of recursive algorithms. In both cases, the proofs of correctness went by induction.

As part of a proof of correctness, we show that the algorithm terminates. For example, we prove that a certain loop has to terminate after some number of iterations, or that the longest chain of recursive calls is finite.

In practice, we usually want to know more than that an algorithm terminates. We want it to terminate in a reasonable amount of time. Thus, we want to prove termination in a stronger sense. We not only want to know that the algorithm terminates, but we also want to quantify how fast it terminates. This brings us to the question of program efficiency. In today's lecture we use recurrences to argue efficiency of programs.

12.1 Efficiency of Algorithms

When we analyze efficiency of an algorithm, we are concerned with the number of operations of a certain kind performed by that algorithm, and how this number grows as a function of the input size. We call this function the *complexity* of the algorithm.

We usually count the number of some *elementary operations*, that is, the simplest operations performed by the algorithm. Examples of such operations are additions of two integers, bit shifts, recursive calls, or comparisons of two values. The elementary operation we choose depends on the situation. We just need to make sure that we choose an elementary operation that tells us something meaningful about the behavior of the algorithm. For example, proving that solving a multiplication problem requires one multiplication does not tell us anything useful. On the other hand, showing that it takes n bit shifts to multiply two numbers whose binary representations consist of n bits tells us something. Today's examples use different kinds of elementary operations that are commonly used.

We mentioned that we were interested in expressing the number of operations in terms of some input size, n. This value n could be simply the value of an integer input, but also the length of an array, or the length of a binary description of an integer.

12.2 Recurrences

A recurrence is an inductive definition of a sequence. It specifies the first few terms explicitly, and then expresses the n-th term in terms of earlier terms in the sequence.

We've already seen an example, the Fibonacci numbers. We specify explicitly that the first two terms, F_1 and F_2, are both 1. For future terms, that is, for $n \geq 3$, we define $F_n = F_{n-1} + F_{n-2}$.

12.2.1 Towers of Hanoi

Recall the rules and the overall strategy for the towers of Hanoi game.

We have three pegs labeled A, B and C, and n disks of increasing size which are stacked on peg A with the largest disk on the bottom and smallest disk on top. The goal is to bring all disks from peg A to peg C. In each step, we can move one disk from one peg to another peg, but we can never move a larger disk on top of a smaller disk.

We discussed a recursive solution last time. We first move the top $n-1$ disks from peg A to peg B. Afterwards, peg A has only the largest disk on it, and peg C is empty, so we move the largest disk from peg A to peg C. To complete the solution, we move the $n-1$ disks from peg B to peg C.

Last class we described a recursive algorithm that returns a sequence of moves that solves the problem with n disks. We repeat this algorithm below for completeness.

```
Algorithm TOWERS($n, A, B, C$)
  Input: $n \in \mathbb{N}$, pegs $A$, $B$ and $C$, where $n$ disks are on peg $A$
  Output: Sequence of moves that brings the disks from peg $A$ to peg $C$, using $B$ as an
          intermediate peg.
(1) if $n = 0$ then return
(2)           else return TOWERS($n$-1,$A$,$C$,$B$)
                          Move top disk of $A$ to $C$
                          TOWERS($n$-1,$B$,$A$,$C$)
```

We now analyze the complexity of `TOWERS` in terms of the number of disks, n. Recall that we analyze the complexity in terms of the number of times the algorithm performs some elementary operation. Some options for picking an elementary operation are the number of recursive calls made or the number of moves (i.e., times we say "Move top disk of x to y") necessary to solve the problem with n disks. We pick the latter option for our analysis.

Let $M(n)$ be the number of moves needed to move n disks from peg A to peg C. Let's find a recurrence that describes $M(n)$.

We see $M(0) = 0$ since there are no disks to move.

We also have $M(1) = 1$ since we just move the only disk from A to C right away. Since we are looking for a recurrence relation that describes $M(n)$, let's find $M(1)$ by tracing through the call to `TOWERS` with $n = 1$ and see if that leads to a recurrence relation for $M(n)$. When `TOWERS` gets called with $n = 1$, we get in the else clause (line 2). There, we make a call to `TOWERS` with $n = 0$, which returns right away and produces no moves, then we make one move, and make another call to `TOWERS` with $n = 0$, which produces no additional moves. Thus, the total number of moves is indeed 1

The trace through the recursive calls on input $n = 1$ shows a pattern. There are three parts to a recursive solution to the problem on n moves.

1. Move the top $n-1$ disks to peg B. This takes $M(n-1)$ moves.

2. Spend one move on moving the largest disk to peg C.

3. Move the $n-1$ disks from peg B to peg C. This uses $M(n-1)$ moves.

Thus, $M(n) = M(n-1) + 1 + M(n-1) = 2M(n-1) + 1$, and we define $M(n)$ as follows.

$$M(0) = 0 \tag{12.1}$$
$$M(n) = 2M(n-1) + 1, \quad n \geq 1 \tag{12.2}$$

This expression is nice, but we would like to know how $M(n)$ behaves as a function of n. In other words, we want an expression for $M(n)$ that only uses n, and does not use $M()$ anywhere.

Intuitively, we see that the number of steps roughly doubles as n increases by 1, so we would expect an expression that contains 2^n somewhere in it (or perhaps 2^{n-1}, 2^{2n}, or something else with 2 in the exponent). To make this intuition clearer, consider, instead, the sequence

$$\begin{aligned} M'(1) &= 1 \\ M'(n) &= 2M'(n-1), \quad n \geq 2 \end{aligned}$$

After writing down the first few terms, we conjecture that $M'(n) = 2^{n-1}$. This is indeed the case, and you can prove it by induction on n. We omit the proof.

Now let's get back to finding an expression for $M(n)$. Let's start by writing down a few values. Since we suspect there is some dependence on powers of two, we also list those. See Table 12.1.

n	0	1	2	3	4	5	6
$M(n)$	0	1	3	7	15	31	63
2^n	1	2	4	8	16	32	64

Table 12.1: A visual aid for finding an expression for $M(n)$ in terms of n.

We observe Table 12.1 and conjecture that $M(n) = 2^n - 1$. This conjecture agrees with what we said earlier about $M(n)$ doubling when n increases by one. Observe that $2^n - 1$ roughly doubles when n increases by one as well.

We now prove our conjecture rigorously by induction on n.

For the base case, $M(0) = 0$ by (12.1) and $2^0 - 1 = 1 - 1 = 0$, so our conjecture holds for $n = 0$.

For the induction step, assume $M(n) = 2^n - 1$. Since $n + 1 \geq 1$, we can use (12.2) and the induction hypothesis to get $M(n+1) = 2M(n) + 1 = 2(2^n - 1) + 1 = 2^{n+1} - 2 + 1 = 2^{n+1} - 1$.

That completes the proof that $M(n) = 2^n - 1$.

12.2.2 Review of the Analysis Technique

The analysis of the towers of Hanoi algorithm shows a typical use of a recurrence as a tool for analyzing the complexity of an algorithm. We take three steps when finding an expression for the complexity $M(n)$. First we use the structure of the algorithm to find a recurrence relation for $M(n)$. Afterwards, we write down a few terms until we see a pattern. We make this pattern our conjectured expression for $M(n)$ and then prove that our conjecture is correct, usually by induction.

The process of finding an expression for $M(n)$ only in terms of n, starting from a recurrence for $M(n)$, is called *solving the recurrence*.

12.2.3 Grade School Multiplication Algorithm - Iterative Version

Our next two examples focus on the grade school multiplication algorithm. First, let's analyze the iterative algorithm from lecture 9. We restate it as Algorithm 1.

The quantity we choose for our analysis of complexity is the number of iterations of the loop on line 2. Let $L(a, b)$ be the number of loop iterations on input (a, b).

Algorithm 1: Multiplication algorithm

```
Input: $a, b$ - positive integers we want to multiply
Output: $ab$ - product of $a$ and $b$
(1) $x \leftarrow a$; $y \leftarrow b$; $p \leftarrow 0$
(2) while $y > 0$ do
(3)     if $y$ is odd then $p \leftarrow p + x$
(4)     $y \leftarrow \lfloor y/2 \rfloor$
(5)     $x \leftarrow 2x$
(6) end
(7) return $p$
```

We see there are no iterations if $b = 0$, so $L(a, b) = 0$ if $b = 0$.

Now suppose $b \geq 1$. The situation becomes more complicated here since y is getting rounded down, so let's look at a simpler case first. If y is a power of 2, no rounding happens until the last step, and y is cut in half in all other iterations. Hence, $L(a, b) = k + 1$ if $b = 2^k$. When b is not an exact power of 2, in particular, if $2^k \leq b < 2^{k+1}$, we get $L(a, b) = k + 1$ as well. In other words, the smallest value of b for which we have k iterations is 2^{k-1}. We can prove this rigorously by induction, but omit the proof here and instead do a similar proof for the recursive version of the algorithm.

We make two remarks about our conjecture about $L(a, b)$. First, every positive integer n satisfies $2^k \leq n < 2^{k+1}$ for some k, so our conjecture covers all possible positive integers. Second, another way to express our quantity $k + 1$ so that it only uses b is $L(a, b) = \lfloor \log_2 b \rfloor + 1$.

12.2.4 Grade School Multiplication Algorithm - Recursive Version

Now let's analyze the recursive version of the grade school multiplication algorithm which we describe below as function `MULT`.

Algorithm MULT(a,b)

```
Input: $a, b \in \mathbb{N}$
Output: $a \cdot b$
(1) if $b = 0$ then return 0
(2) if $b$ is even then return $2 \cdot$ MULT($a$, $\lfloor b/2 \rfloor$)
(3)                 else return $2 \cdot$ MULT($a$, $\lfloor b/2 \rfloor$) $+ a$
```

We can no longer choose the number of iterations of a loop as our measure of complexity because there are no loops in `MULT`. Recall from last time that multiplication by two in binary corresponds to a left shift of the digits and putting a zero at the end. We choose the number of shifts as our complexity measure.

Let $S(a, b)$ be the number of shifts when multiplying a and b. We have $S(a, b) = 0$ when $b = 0$. The only shifts that happen in the recursive part come from calls to `MULT` and from multiplications by two on the last two lines. Multiplication by two takes one shift, so $S(a, b) = S(a, \lfloor b/2 \rfloor) + 1$.

Now we try to find a pattern again. See Table 12.2 for some values of $S(a, b)$. Also notice that $S(a, b)$ only depends on b.

b	0	1	2	3	4	7	8	9
$S(a,b)$	0	1	2	2	3	3	4	4

Table 12.2: A visual aid for finding an expression for $S(a,b)$ in terms of b.

We conjecture that $S(a,b)=0$ if $b=0$, and $S(a,b)=k+1$ if $2^k \leq b < 2^{k+1}$. We prove our conjecture by strong induction on b.

For the base we have $S(a,0)=0$ since the algorithm returns zero right away.

For the induction step, we assume that our conjecture holds for all $b \leq B$, and we show it holds for $b = B+1$. Since $B \geq 0$, there is some integer $k \in \mathbb{N}$ such that $2^k \leq B+1 < 2^{k+1}$. Now we apply our recurrence and get

$$S(a, B+1) = S(a, \lfloor (B+1)/2 \rfloor) + 1. \tag{12.3}$$

We would like to apply the induction hypothesis and say that $S(a, \lfloor (B+1)/2 \rfloor) = k$. For this, we need to show that if $2^k \leq B+1 < 2^{k+1}$, then $2^{k-1} \leq \lfloor (B+1)/2 \rfloor < 2^k$. We do this in two steps. First we show the lower bound and then the upper bound.

If $2^k \leq B+1$, we have $2^{k-1} \leq (B+1)/2$. Moreover, 2^{k-1} is an integer, so rounding $(B+1)/2$ down to the nearest integer cannot round down below 2^{k-1}. This proves the lower bound.

For the upper bound, note that the largest $B+1$ can be is $2^{k+1}-1$. Thus, $(B+1)/2 < 2^k$, and rounding down a value that is less than 2^k gives us an integer that is less than 2^k. It follows that $\lfloor (B+1)/2 \rfloor < 2^k$.

Hence, we now have $2^{k-1} \leq \lfloor (B+1)/2 \rfloor < 2^k$, and $\lfloor (B+1)/2 \rfloor \leq B$, so our induction hypothesis applies, and implies that $S(a, \lfloor (B+1)/2 \rfloor) = k$ as desired. Substituting into (12.3) then yields $S(a, B+1) = k+1$ where $2^k \leq B+1 < 2^k$, which is what we wanted to show.

12.2.5 Greatest Common Divisor Algorithm

Today's last example is the recursive algorithm for finding greatest common divisors from lecture 10.

```
Algorithm GCD(a, b)
    Input: $a, b \in \mathbb{N}$, $a > 0$, $b > 0$
    Output: gcd$(a, b)$
(1) if $a = b$ then return $a$
(2) if $a < b$ then return GCD($a, b - a$)
(3) if $a > b$ then return GCD($a - b, b$)
```

We measure the complexity by counting the number of recursive calls. Let $R(a,b)$ be the number of recursive calls on input (a,b). We point out that the dependence on the number of recursive calls on the input is more complicated because the roles of a and b in this algorithm are entirely symmetric. This is unlike the multiplication algorithm where the complexity depended only on one of the parameters.

It turns out that it is too difficult to find a good expression for $R(a,b)$. Therefore, to simplify matters, we define $R'(n)$ to be the number of recursive calls on input (a,b) when $a, b \leq n$. This is reasonable to do because we define complexity in terms of the input size, and we could view the input size to be the largest of a and b. Note that $(\forall a, b \leq n)\ R(a,b) \leq R'(n)$.

We now find an expression for $R'(n)$ using the same strategy as in the previous examples.

The value of $R'(0)$ is undefined since Algorithm GCD doesn't work on such inputs.

We have $R'(1) = 0$ because the only possibility here is that $a = b = 1$, and the algorithm returns right away in that case.

For $R'(2)$, the possible inputs are $(1, 1)$, $(1, 2)$, $(2, 1)$, and $(2, 2)$. The numbers of calls are 0, 1, 1, and 0, respectively. Let's briefly justify this statement. For the inputs $(1, 1)$ and $(2, 2)$, the algorithm returns right away. On input $(1, 2)$, the algorithm makes a recursive call to itself with input $(1, 2 - 1) = (1, 1)$, and that call takes $R'(1) = 0$ recursive calls. This means the call to the algorithm on input $(1, 1)$ is the only recursive call in this case. We would argue similarly for the input $(2, 1)$.

The possible inputs we need to consider in order to find the value of $R'(3)$ are $(1, 3)$, $(2, 3)$, $(3, 3)$, $(3, 2)$, $(3, 1)$, and the inputs we considered for $R'(2)$. Each of the new inputs takes either zero (on input $a = b = 3$) or 2 (all the other inputs with 3 in them) recursive calls. We know the number of recursive calls for all other inputs with $a, b \leq 3$ from our analysis of $R'(2)$, and see that $R'(3) = 2$.

Based on the first few values, we conjecture that $R'(n) = n-1$. Some intuition behind this is that the sequence of inputs to recursive calls to `GCD` on input $(n, 1)$ is $(n-1, 1), (n-2, 1), \ldots, (2, 1), (1, 1)$, which is a total of $n - 1$ recursive calls.

We prove by this conjecture in two steps. We use induction to show that $R'(n) \geq n - 1$ and then that $R'(n) \leq n - 1$. Combining those two inequalities yields $R'(n) = n - 1$.

For the first inequality, we need an input for which the algorithm makes at $n-1$ recursive calls. The goal is to find an input on which the algorithm makes as little progress as possible with each recursive call. The input $(n, 1)$ works. We leave the rigorous proof to the reader, and only appeal to the observation we made earlier.

So now we know the number of recursive calls can be as "bad" as $n - 1$ for some input (a, b) with $a, b \leq n$. Before we show by strong induction show that it cannot get worse than $n - 1$, let's find a recurrence for $R'(n)$.

We already know that $R'(1) = 0$.

Now consider an input (a, b) such that $a, b \leq n$. If $a = b$, the algorithm returns right away, so there are no recursive calls. If $a < b \leq n$, we have $R(a, b) = 1 + R(a, b - a)$. Note that $b - a \leq n - 1$ because $b \leq n$ and $a \geq 1$. Thus, $R(a, b) = 1 + R(a, b - 1) \leq 1 + R'(n - 1)$. Since this inequality holds for every input (a, b) such that $a, b \leq n$, it holds for the input that maximizes $R(a, b)$ over all $a, b \leq n$, which means we have $R'(n) \leq 1 + R'(n - 1)$.

So now let's prove that $R'(n) \leq n - 1$. We proceed by induction on n.

For the base case, note $R'(1) = 0 = 1 - 1$.

For the induction step, our recurrence tells us that $R'(n + 1) \leq 1 + R'(n)$. By the induction hypothesis, $R'(n) \leq n - 1$, so $R'(n + 1) \leq 1 + n - 1 = n$. This completes the proof.

So we have $R(a, b) \leq R'(\max(a, b)) = \max(a, b) - 1$. Next time we will analyze the greatest common divisor algorithm from your fourth homework and see that its running time is bounded above roughly by $c \log_2(\max(a, b))$ where c is some positive real number. This is a much better bound than the one we obtained today.