

Recursion

In the lecture notes from last lecture, we saw an algorithm for computing the factorial function. There were two ways to compute it. One used a for loop¹ and the other used recursion. There are many other problems for which there is a non-recursive and a recursive solution. Sometimes the recursive way is more natural for expressing what you want to do, and other times the non-recursive way is more natural. Another issue is the computational cost. For factorial, the costs of the two algorithms are similar in that both run in time proportional to n . Let's now look at another problem in which the non-recursive vs. recursive solutions have quite different time costs.

Fibonacci numbers

Consider the Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

$$F(n) = F(n - 1) + F(n - 2),$$

where $F(0) = 0, F(1) = 1$. The standard way to calculate the Fibonacci numbers is just to iterate, starting at $n = 0$. Suppose you wanted the n^{th} Fibonacci number, where $n > 0$.

ALGORITHM: fib(n)

```

if ((n == 0) | (n == 1))
    return n
else{
    fibiMinus2 = 0
    fibiMinus1 = 1
    for i = 2 to n{
        fibi = fibiMinus1 + fibiMinus2
        fibiMinus2 = fibiMinus1
        fibiMinus1 = fibi
    }
    return fibi
}

```

The method requires n passes through the “for loop”. Each pass takes a small (fixed) number of operations. So we would expect the number of steps to be about cn for some constant c .

A *recursive algorithm* for computing the n th Fibonacci number goes like this:

```

Algorithm: fib(n)    // assume n > 0
// Input:  the index of the Fibonacci number to be computed
// Output: the n-th fibonacci number
//
if ((n == 0) || (n == 1))
    return n
else
    return fib(n-1) + fib(n-2)

```

¹not discussed in class, but it was mentioned in notes

The trouble with this algorithm is that you end up calling `fib` on the same parameter *many* times. For example, suppose you are asked to compute the 247-th Fibonacci number. `fib(247)` calls `fib(246)` and `fib(245)`, and `fib(246)` calls `fib(245)` and `fib(244)`. But now notice that `fib(245)` is called twice. Similar redundancies occur each step of the way until you reach `fib(1)` and `fib(2)`. As we will see a few lectures from now, the number of steps required in the computational grows like 2^n , which takes a lot longer than the iterative (for loop) algorithm.

Towers of Hanoi

Let's now turn to an example in which recursion allows us to express a solution in a very simple manner. Unlike in the previous example, the issue here is expressibility, rather than computational efficiency. (The algorithm here still takes a long time for large n , as we'll see a few lectures from now.)

The problem is called *Tower of Hanoi*. There are three stacks (towers) and a number n of disks of different radii. (See http://en.wikipedia.org/wiki/Tower_of_Hanoi). We start with the disks all on one stack, say stack 1, and such that the size of disks decreases from bottom to top. The objective is to move the disks from the starting stack to one of the other two stacks, say stack 2, while obeying the following rules:

1. A larger disk cannot be on top of a smaller disk (at any time).
2. Each move consists of popping a disk from one stack and pushing it onto another stack, or more intuitively, taking the disk at the top of one stack and putting it on another stack.

The (remarkably simple) recursive algorithm for solving the problem goes as follows. The three stacks are labelled s_1, s_2, s_3 . One of the stacks is where the disks "start". Another stack is where the disks should all be at the "finish". The third stack is the only remaining one.

```
Tower(n,start, finish, other)
  if n>0 then
    Tower(n-1, start, other, finish)
    move from start to finish          // i.e. finish.push( start.pop() )
    Tower(n-1, other, finish, start)
  end if
```

For example, `Tower(1,s1,s2,s3)` would produce to the following sequence of instructions:

```
Tower(0,s1,s3,s2)
move from s1 to s3
Tower(0,3,2,1)
```

The two calls `Tower(0,*,*,*)` would do nothing since the condition $n > 0$ is not met, and so the three instructions would collapse to one: instruction:

```
move from 1 to 2
```

What about `Tower(2,1,2,3)` ? This would produce the following sequence of instructions:

```
Tower(1,1,3,2)
move from 1 to 2
Tower(1,3,2,1)
```

and the two calls `Tower(1,*,*,*)` would each move one disk, similar to the previous example (but with different parameters). So, in total there would be 3 moves:

```
move disk from 1 to 3
move disk from 1 to 2
move disk from 3 to 2
```

Here are the states of the tower for `Tower(3,1,2,3)` and the corresponding print instructions.

```
*
**
***
---      ---      ---      (initial)

**
***      *
---      ---      ---      (after moving disk from 1 to 2)

***      *      **      (after moving disk from 1 to 3)
---      ---      ---

***      **      *
---      ---      ---      (after moving from 2 to 3)
```

which completes the `Tower(2, 1, 3, 2)` call.

```
*
***      **
---      ---      ---      (after moving from 1 to 2)
```

and now we call `Tower(2, 3, 2, 1)`

```
*      ***      **
---      ---      ---      (after moving from 3 to 1)
```

```

      **
*     ***
---   ---   ---   (after moving from 3 to 2)

```

```

      *
      **
      ***
---   ---   ---   (after moving from 1 to 2)

```

and we are done!

For any $n \geq 0$, Towers of Hanoi algorithm is correct for n disks

For the algorithm to be “correct”, we need to ensure that a larger disk is never place on top of a smaller disk, and that the n disks are moved from the start tour to the finish tour. The proof is by induction.

Base case: The rule is obviously obeyed if $n = 1$ and the algorithm simply moves the one disk from *start* to *finish*.

Induction step: Suppose the algorithm is correct if there are $n = k$ disks *in the initial tower*. This is the induction hypothesis. We need to show that the algorithm is therefore correct if there are $n = k + 1$ disks *in the initial tower*. For $n = k + 1$, the algorithm has three steps, namely,

- **Tower**(k , *start*, *other*, *finish*)
- move (disk $k + 1$) from *start* to *finish*
- **Tower**(k , *other*, *finish*, *start*)

The first recursive call to **Tower** moves k disks from *start* to *other*, while obeying the rule for these k disks (by the induction hypothesis). The second step moves the biggest disk ($k + 1$) from *start* to *finish*. This also obeys the rule, since *finish* does not contain any of the k smaller disks (because these smaller disks were all moved to the *other* tower). Finally, the second recursive call to **Tower** move k disks from *other* to *finish*, while obeying the smaller-on-bigger rule (by the induction hypothesis). This completes the proof.

ASIDE: In the lecture, I got a bit confused and said that this proof was not good enough. That was wrong. The proof is fine.

In class I also discussed the time required by the algorithm. I won't present it here. We'll see it a few lectures from now.