Recursion

Recursion is a technique for solving problems, such that the solution to the problem depends on solutions to smaller versions of the instance. Many problems can be solved either by a recursive technique or non-recursive (e.g. iterative) technique, and often these techniques are closely related. In the next few lectures, we’ll look at several examples. We’ll also see how recursion is closely related to mathematical induction.

Example 1: Factorial

The factorial function is defined as follows:

\[ n! = 1 \cdot 2 \cdot 3 \ldots (n - 1) \cdot n. \]

Here is an algorithm (written in Java) for computing it.

```java
int factorial(int n) { // assume n >= 1
    int result = 1;
    for (int i = 1; i <= n; i++)
        result *= i;
    return result;
}
```

Here is another way to define and compute \( n! \) which is more subtle, namely if \( n > 1 \), then

\[ n! = n \cdot (n - 1)! \]

and here is the corresponding algorithm coded in Java which is recursive. Note that the method `factorial` calls itself.

```java
int factorial(int n) { // algorithm assumes argument: n >= 1
    if (n == 1)
        return 1; // base condition
    else
        return n * factorial(n - 1);
}
```

Recursive algorithms can’t keep called themselves *ad infinitum*. Rather, they need to have a condition which says when to stop. This is called a *base condition*. For the `factorial` function, the base condition is that the argument is 1. Anytime you write a recursive algorithm, make sure you have a base condition and make sure you reach it. Typically this is ensured by having the parameter of the recursive call be smaller, *e.g.* \( n-1 \) rather than \( n \) in the case of `factorial`. 
Although I admit this is pedantic thing to do, let’s use mathematical induction to show that the factorial algorithm is correct. My point here is not to convince you it is indeed correct, but rather to make an explicit connection between mathematical induction and recursion.

**Statement:** The recursive factorial algorithm indeed computes \( n! \) for any input value \( n \geq 1 \).

**Proof:** First, the base case: If the parameter \( n \) is 1, then the algorithm returns 1.

Second, the induction step: Suppose \( \text{factorial}(k) \) indeed returns \( k! \). (This is the induction hypothesis). We want to show it follows that \( \text{factorial}(k+1) \) returns \( (k + 1)! \). But this is easy to see by inspection, since the algorithm returns \( (k + 1) \times k! \) which is just \( (k + 1)! \).

**Example 2: 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 function \( F(n) \) is defined in terms of itself, and so it is recursive.

Here is an iterative algorithm for computing the \( n \)-th Fibonacci number. We start at \( n = 0, 1 \) and move forward (assuming \( n > 0 \)).

```java
fib(n){
    if ((n == 0) | (n == 1))
        return n
    else{
        f0 = 0
        f1 = 1
        for i = 2 to n{
            f2 = f1 + f0
            f0 = f1 // set F(n) for next round
            f1 = f2 // set F(n+1) for next round
        }
        return f2
    }
}
```

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 is simpler to express:

```java
fib(n){  // assume n > 0
    if ((n == 0) || (n == 1))
        return n
    else
        return fib(n-1) + fib(n-2)
}
```
Here is a proof that this recursive algorithm for computing the nth Fibonacci number is correct. (Again, I realize this is bordering on pedantic, but I do want to emphasize the important connection between induction and recursion. If you can’t understand this connection for these examples, then more complex ones will be even more difficult to grasp.)

First, the base cases: If the parameter \( n \) is 0 or 1, then the algorithm returns 0 or 1, respectively, which is correct.

Second, the induction step: Suppose \( \text{fib}(k) \) and \( \text{fib}(k-1) \) indeed return the \( k \)th and \( (k-1) \)-th Fibonacci number, for any \( k \geq 1 \). (This is the induction hypothesis). We want to show it follows that \( \text{fib}(k+1) \) indeed returns the \( (k+1) \)th Fibonacci number. But again this is easy to see by inspection, since the algorithm returns the sum of Fibonacci numbers \( k-1 \) and \( k \) is indeed equal to the Fibonacci number \( k+1 \).

Interesting, the recursive version turns out to be very expensive since it ends up calling \( \text{fib} \) on the same parameter \( n \) many times, which is unnecessary. For example, suppose you are asked to compute the 247-th Fibonacci number. \( \text{fib}(247) \) calls \( \text{fib}(246) \) and \( \text{fib}(245) \), and \( \text{fib}(246) \) calls \( \text{fib}(245) \) and \( \text{fib}(244) \). But now notice that \( \text{fib}(245) \) is called twice. The problem here is that every time you want to compute \( \text{fib}(k) \) where \( k > 1 \), you need to do \( \text{two} \) recursive calls. This leads to an exponential explosion in the number of calls. (I haven’t provided a formal argument here, but hopefully get the idea. If not, see the picture in the slides.)

**Example 3: reversing a list**

Let’s next revisit a few algorithms for lists, and examine recursive versions. The first example is to reverse a list (see linked list exercises for an iterative version). The idea can be conveyed with the following picture. To reverse the list,

\[
(a \ b \ c \ d \ e \ f \ g)
\]

we can remove the first element \( a \) and reverse the remaining elements,

\[
a \ (b \ c \ d \ e \ f \ g) \quad \longrightarrow \quad a \ (g \ f \ e \ d \ c \ b)
\]

and then add the removed element at the end of the (reversed) list.

\[
(g \ f \ e \ d \ c \ b \ a)
\]

Here is the pseudocode:

```java
reverse(list){ // assume n > 0
    if list.size == 1 // base case
        return list
    else{
        firstElement = list.removeFirst()
        list = list.reverse() // list has only n-1 elements
        return list.addLast(firstElement)
    }
}
```

last updated: 4th Oct, 2016
and here is some Java code for a class that implements the List methods:

```java
public void reverse() {
    if (this.size > 1) {
        E e = this.removeFirst();
        this.reverse();
        this.addLast(e);
    }
}
```

**Example 4: sorting a list**

Recall the selection sort algorithm. The basic idea was to maintain two lists: a sorted list, and a ‘rest’ list which is unsorted. The algorithm loops repeatedly through the rest list, removes the minimum element each time, and adds it to the end of the sorted list. (Since the sorted list consists of elements that are all smaller than or equal to elements in the rest list, all the elements of the rest list will eventually be added after elements in the sorted list.) The recursive algorithm below is essentially the same idea. Remove the minimum element, sort the rest list (recursively), and add the minimum element to the front of the sorted rest list (that is, the sorted rest list will be after any minimum elements that have been removed). That may sound complicated, but look how simple the code is:

```java
sort(list) { // assumes list.size >= 1
    if list.size == 1
        return list // base case
    else {
        minElement = list.removeMin()
        list = sort(list)
        return list.addFirst(minElement)
    }
}
```

If you are not convinced this works, see [http://www.cim.mcgill.ca/~langer/250/SortTest.java](http://www.cim.mcgill.ca/~langer/250/SortTest.java) for java code that does roughly the above.

**Tower of Hanoi**

Let’s now turn to an example of a problem in which a recursive solution is very easy to express, and a non-recursive solution is very difficult to express (and I won’t even both with the latter). 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](http://en.wikipedia.org/wiki/Tower_of_Hanoi)). We start with the disks all on one stack, say stack 1, such that the size of disks on each stack increases from top to bottom. The objective is to move the disks from the starting stack (1) to one of the other two stacks, say 2, while obeying the following rules:

1. A larger disk cannot be on top of a smaller disk.
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 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) // i.e. finish.push( start.pop() )
    move from start to finish
    tower(n-1, other, finish, start)
end if

Here I will label the stacks $A$, $B$, $C$. For example, $tower(1,A,B,C)$ would produce the following sequence of instructions:

tower(0,A,C,B) // don’t do anything
move from A to B
tower(0,C,B,A) // don’t do anything

The two calls $tower(0,*,*,*)$ would do nothing since the condition $n > 0$ is not met.
What about $tower(2,A,B,C)$? This would produce the following sequence of instructions:

tower(1,A,C,B)
move from A to B
tower(1,C,B,A)

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 from A to C
move from A to B
move from C to B

Here are the states of the tower for $tower(3,A,B,C)$ and the corresponding print instructions. Notice that we need to do the following:

tower(2,A,C,B)
move from A to B
tower(2,C,B,A)

The initial state is:

```
* 
**
***
--- --- --- (initial)
```
So first we do `tower(2,A,C,B)`, which takes 3 moves:

```
**
*** *
--- --- ---  (after moving disk from A to B)

*** * **
--- --- ---

* *** **
--- --- ---  (after moving from B to C)
```

Next we do "move from A to B":

```
*
*** **
--- --- ---  (after moving from A to B)
```

Then we call `tower(2, C, B, A)` which does the following 3 moves:

```
* *** **
--- --- ---  (after moving from C to A)

**
*** *
--- --- ---  (after moving from C to B)

* **
*** ***
--- --- ---  (after moving from A to B)
```

and we are done!

Claim: 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 we move one disk at a time, and that the $n$ disks are eventually moved from the start to finish. The proof is by mathematical 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,

- $\text{tower}(k, \text{start}, \text{other}, \text{finish})$
- move from start to finish
- $\text{tower}(k, \text{other}, \text{finish}, \text{start})$

The first recursive call to $\text{tower}$ moves $k$ disks from start to other, while obeying the rules for these $k$ disks. (This is 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 $\text{tower}$ move $k$ disks from other to finish, while obeying the rules (again, by the induction hypothesis). This completes the proof.

Finally, let’s examine how many moves $\text{tower}(n, \ldots)$ takes? $\text{tower}(1, \ldots)$ takes 1 move. $\text{tower}(2, \ldots)$ takes 3 moves, namely two recursive calls to $\text{tower}(1, \ldots)$ which take 1 move each, plus one move. $\text{tower}(3, \ldots)$ makes two recursive calls to $\text{tower}(2, \ldots)$ which we just said takes 3 moves each, plus one move, for a total of $3*2 + 1 = 7$. Similarly, $\text{tower}(4, \ldots)$ makes two recursive calls to $\text{tower}(3, \ldots)$ which we just said takes 7 moves each, plus one move, for a total of $7*2 + 1 = 15$. Similarly, $\text{tower}(5, \ldots)$ takes $2*15 + 1 = 31$ moves, and $\text{tower}(6, \ldots)$ takes $2*31 + 1 = 63$ moves. In general, $\text{tower}(n, \ldots)$ makes two recursive calls to $\text{tower}(n-1, \ldots)$ plus one move. See if you can prove by induction that $\text{tower}(n, \ldots)$ takes $2^n - 1$ moves.