Recursion

Recursion is a technique for solving problems in which the solution to the problem of size n is based on solutions to versions of the problem of size smaller than n. 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 argue that recursion is closely related to mathematical induction.

Example 1: Factorial

The factorial function is defined as follows:

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

Unlike the sum of numbers to n which we discussed last class, there is no formula that gives us the answer of taking the product of numbers from 1 to n, and we need to compute it. Here is an algorithm (written in Java) for computing it.

```
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.

```
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`.

Let's use mathematical induction to convince ourselves that the factorial algorithm is correct.

Claim: 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: The induction hypothesis is that `factorial(k)` indeed returns $k!$. We want to show it follows that `factorial(k+1)` returns $(k+1)$!. But this is easy to see by inspection, since the induction hypothesis implies that the algorithm returns $(k+1) * k!$, which is just $(k+1)!$.

Example 2: Fibonacci numbers

Consider the Fibonnacci 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 $n > 0$).

```
fibonacci(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 nth Fibonacci number is simpler to express:

```
fibonacci(n){    //   assume n > 0
   if ((n == 0) || (n == 1))
      return n
   else
      return fibonacci(n-1) + fibonacci(n-2)
}
```

Here is a proof that this recursive algorithm for computing the nth Fibonaccie number is correct. First, the base case: If the parameter `n` is 0 or 1, then the algorithm returns 0 or 1, respectively, which is correct.

Second, the induction step: the induction hypothesis is that `fibonacci(k)` and `fibonacci(k-1)` returns the kth and $(k-1)$-th Fibonacci number, for any $k \geq 1$. We want to show that this implies `fibonacci(k+1)` returns the $(k+1)$th Fibonacci number. But again this is easy to see by inspection, since the algorithm returns the sum of `k-1` and `k` Fibonacci numbers which is the `k+1` Fibonacci number.

The recursive version turns out to be very slow since it ends up calling `fibonacci` on the same parameter n *many* times, which is unnecessary. For example, suppose you are asked to compute the 247-th Fibonacci number. `fibonacci(247)` calls `fibonacci(246)` and `fibonacci(245)`, and `fibonacci(246)` calls `fibonacci(245)` and `fibonacci(244)`. But now notice that `fibonacci(245)` is called twice. The problem here is that every time you want to compute `fibonacci(k)` where `k > 1`, you need to do *two* recursive calls. This leads to a combinatorial explosion in the number of calls. (I haven't provided a formal calculation here in exactly how many calls would be made, but hopefully you get the idea that many repetitions of the same calls occur. 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)  ---->   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. I have written it so that `list` is an argument to the various methods that are called, which is different notation from what we used in the list lectures.

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

And here is some Java code for a class that implements the List methods:

```
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:

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

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`). 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 $s1, s2, s3$. 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
```

Here I will label the stacks `A, B, C`. For example, `tower(1,A,B,C)` would produce to 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 disk from A to C)
      ---       ---         ---


                            *
      ***                   **       (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,

- `tower(k,start,other,finish))`
- `move 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 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 `tower` move k disks from `other` to `finish`, while obeying the rules (again, by the induction hypothesis). This completes the proof.

[ASIDE: In Sec. 001 lecture, I mistakenly skipped the slides for the following calculation.]

How many moves does `tower(n, ...)` take? `tower(1, ...)` takes 1 move. `tower(2, ...)` takes 3 moves, namely two recursive calls to `tower(1, ...)` which take 1 move each, plus one move. `tower(3, ...)` makes two recursive calls to `tower(2, ...)` which we just said takes 3 moves each, plus one move, for a total of 3*2 + 1 = 7. Similarly, `tower(4, ...)` makes two recursive calls to `tower(3, ...)` which we just said takes 7 moves each, plus one move, for a total of 7*2 + 1 = 15. And so on... `tower(5, ...)` takes 2*15 + 1 = 31 moves, and `tower(6, ...)` takes 2*31 + 1 = 63 moves. In general, `tower(n, ...)` makes two recursive calls to `tower(n-1, ...)` plus one move. So one can prove by induction that `tower(n, ...)` takes $2^n - 1$ moves.

Recursion and the Call Stack

Back in lecture 8, we discussed stacks and I mentioned the "call stack". Each time a method calls another method (or a method calls itself, in the case of recursion), the computer needs to do some administration to keep track of the "state" of method at the time of the call. This information is called a "stack frame". You will learn more about this in COMP 273, but it is worth mentioning now to take some of the mystery out of how recursion is implemented in the computer.

As an example, suppose the program calls `factorial(6)`. This leads to a sequence of recursive calls and subsequent returns from these calls. For example, right before *returning* from the `factorial(3)` call, we have made the following sequence of calls and returns:

```
factorial(6), factorial(5), factorial(4), factorial(3), factorial(2),
factorial(1), return from factorial(1), return from factorial(2)
```

the call stack looks like this,

```
frame for factorial(3):  [factn = 6, n=3] <---- top of stack
frame for factorial(4):  [factn = 0, n=4]
frame for factorial(5):  [factn = 0, n=5]
frame for factorial(6):  [factn = 0, n=6] <---- bottom of stack
```

Using the Eclipse debugger and setting breakpoint within the `factorial()` method, you can see how the stack evolves. I strongly recommend that you do this and verify how this works.

`http://www.cim.mcgill.ca/~langer/250/TestFactorial.java`

I would do the same for the Tower of Hanoi.

`http://www.cim.mcgill.ca/~langer/250/TestTowerOfHanoi.java`

See slides for screen shots.

I should emphasize that the call stack is not some abstract idea, but rather it is a real data structure that the software that runs your Java program (called the "Java Virtual Machine"). The stack consists of *stack frames*, one for each method that is called. In the case of recursion, there is one stack frame for each time the method is called.

The stack frame contains all the information that is needed for that method. This includes local variables declared and used by that method, parameters that are passed to the method, and information about where the method returns when it is done, that is, who called the method.

You will learn about the call stack and stack frames work in much more detail in COMP 273. I mention it here because I want you to get familiar with the idea, and because I want you to be aware that the call stack and stack frame really exist. Indeed most decent IDEs will allow you to examine the call stack (see slides) and at least the current stack frame, i.e. the frame on top of the call stack.