Introduction to Computer Science - Java

Recursion

Simply put, recursion is when a function calls itself. That is, in the course of the function definition there is a call to that very same function. At first this may seem like a never ending loop, or like a dog chasing its tail. It can never catch it. So too it seems our method will never finish. This might be true is some cases, but in practise we can check to see if a certain condition is true and in that case exit (return from) our method. The case in which we end our recursion is called a **base case** . Additionally, just as in a loop, we must change some value and incrementally advance closer to our base case.

Consider this function.

```
void myMethod( int counter)
{
if(counter == 0)
     return;
else
       {
       System.out.println(""+counter);
       myMethod(--counter);
       return;
       }
}
```

This recursion is not infinite, assuming the method is passed a positive integer value.
What will the output be?
Consider this method:

```
void myMethod( int counter)
{
if(counter == 0)
     return;
else
       {
       System.out.println("hello" + counter);
       myMethod(--counter);
       System.out.println(""+counter);
       return;
       }
}
```

If the method is called with the value 4, what will the output be? Explain.
The above recursion is essentially a loop like a for loop or a while loop. When do we prefer recursion to an iterative loop? We use recursion when we can see that our problem can be reduced to a simpler problem that can be solved after further reduction.

Every recursion should have the following characteristics.

1. A simple base case which we have a solution for and a return value.

2. A way of getting our problem closer to the base case. I.e. a way to chop out part of the problem to get a somewhat simpler problem.
3. A recursive call which passes the simpler problem back into the method.

The key to thinking recursively is to see the solution to the problem as a smaller version of the same problem. The key to solving recursive programming requirements is to imagine that your method does what its name says it does even before you have actually finish writing it. You must pretend the method does its job and then use it to solve the more complex cases. Here is how.

Identify the base case(s) and what the base case(s) do. A base case is the simplest possible problem (or case) your method could be passed. Return the correct value for the base case. Your recursive method will then be comprised of an if-else statement where the base case returns one value and the non-base case(s) recursively call(s) the same method with a **smaller** parameter or set of data. Thus you decompose your problem into two parts: (1) The simplest possible case which you can answer (and return for), and (2) all other more complex cases which you will solve by returning the result of a second calling of your method. This second calling of your method (recursion) will pass on the complex problem but reduced by one increment. This decomposition of the problem will actually be a complete, accurate solution for the problem for all cases other than the base case. Thus, the code of the method actually has the solution on the first recursion.
Let's consider writing a method to find the factorial of an integer. For example 7! equals 7*6*5*4*3*2*1 .
But we are also correct if we say 7! equals 7*6!.
In seeing the factorial of 7 in this second way we have gained a valuable insight. We now can see our problem in terms of a simpler version of our problem and we even know how to make our problem progressively more simple. We have also defined our problem in terms of itself. I.e. we defined 7! in terms of 6!. This is the essence of recursive problem solving. Now all we have left to do is decide what the base case is. What is the simplest factorial? 1!. 1! equals 1.

Let's write the factorial function recursively.

```
int myFactorial( int integer)
{
if( integer == 1)
     return 1;
else
       {
       return(integer*(myFactorial(integer-1);
       }
}
```

Note that the base case (the factorial of 1) is solved and the return value is given. Now let us imagine that our method actually works. If it works we can use it to give the result of more complex cases. If our number is 7 we will simply return 7 * the result of factorial of 6. So we actaully have the exact answer for all cases in the top level recursion. Our problem is getting smaller on each recursive call because each time we call the method

we give it a smaller number. Try running this program in your head with the number 2. Does it give the right value? If it works for 1 then it must work for two since 2 merely returns 2 * factorial of 1. Now will it work for 3? Well, 3 must return 3 * factorial of 2. Now since we know that factorial of 2 works, factorial of 3 also works. We can prove that 4 works in the same way, and so on and so on.
Food for thought: ask yourself, could this be written iteratively?
Note: make it your habit of writing the base case in the method as the first statement.

Note: Forgetting the base case leads to infinite recursion.
However, in fact, your code won't run forever like an infinite loop, instead, you will eventually run out of stack space (memory) and get a run-time error or exception called a stack overflow. There are several significant problems with recursion. Mostly it is hard (especially for inexperienced programmers) to think recursively, though many AI specialists claim that in reality recursion is closer to basic human thought processes than other programming methods (such as iteration). There also exists the problem of stack overflow when using some forms of recursion (head recursion.) The other main problem with recursion is that it can be slower to run than simple iteration. Then why use it? It seems that there is always an iterative solution to any problem that can be solved recursively. Is there a difference in computational complexity? No.
Is there a difference in the efficiency of execution? Yes, in fact, the recursive version is usually less efficient because of having to push and and pop recursions on and off the run-time stack, so iteration is quicker. On the other hand, you might notice that the recursive versions use fewer or no local variables.

So why use recursion? The answer to our question is predominantly because it is easier to code a recursive solution once one is able to identify that solution. The recursive code is usually smaller, more concise, more elegant, possibly even easier to understand, though that depends on ones thinking style. But also, there are some problems that are very difficult to solve without recursion. Those problems that require backtracking such as searching a maze for a path to an exit or tree based operations (which we will see in semester 2) are best solved recursively. There are also some interesting sorting algorithms that use recursion.

Towers of Hanoi

This problem comes from history, monks in Vietnam were asked to carry 64 gold disks from one tower (stack) to another. Each disk is of a different size. There are 3 stacks, a source stack, a destination stack and an intermediate stack. A disk is placed on one of three stacks but **no disk can be placed on top of a smaller disk**. The source tower holds 64 disks. How will the monks solve this problem? How long will it take them?

The easiest solution is a recursive one. The key to the solution is to notice that to move any disk, we must first move the smaller disks off of it, thus a recursive definition. Another way to look at it is this, if we had a method to move the top three disks to the middle position, we could put the biggest disk in its place. All we need to do is assume we have this method and then call it.

Lets start with 1 disk (our base case): Move 1 disk from start tower to destination tower and we are done.
To move 2 disks:
Move smaller disk from start tower to intermediate tower, move larger disk from start tower to final tower, move smaller disk from intermediate tower to final tower and we are done.
To move n disks (or think of, say, 3 disks):
Solve the problem for n - 1 disks (i.e. 2 disks) using the intermediate tower instead of the final tower (i.e. get 2 disks onto the intermediate tower). Then , move the biggest disk from start tower to final tower. Then again solve the problem for n - 1 disks but use the intermediate tower instead of the start tower (i.e. get the 2 disks onto the final tower using the start tower as the intermediate tower).

Tail Recursion

Tail recursion is defined as occuring when the recursive call is at the end of the recursive instruction. This is not the case with my factorial solution above. It is useful to notice when ones algorithm uses tail recursion because in such a case, the algorithm can usually be rewritten to use iteration instead. In fact, the compiler will (or at least should) convert the recursive program into an iterative one. This eliminates the potential problem of stack overflow.

This is not the case with head recursion, or when the function calls itself recursively in different places like in the Towers of Hanoi solution. Of course, even in these cases we could also remove recursion by using our own stack and essentially simulating how recursion would work.

In my example of factorial above the compiler will have to call the recursive function before doing the multiplication because it has to resolve the (return) value of the function before it can complete the multiplication. So the order of execution will be "head" recursion, i.e. recursion occurs before other operations.

To convert this to tail recursion we need to get all the multiplication finished and resolved before recursively calling the function. We need to force the order of operation so that we are not waiting on multiplication before returning. If we do this the stack frame can be freed up.

The proper way to do a tail-recursive factorial is this:

```
int factorial(int number) {
    if(number == 0) {
            return 1;
        }
        factorial_i(number, 1);
}

int factorial_i(int currentNumber, int sum) {
    if(currentNumber == 1) {
```

```
        return sum;
    } else {
        return factorial_i(currentNumber - 1, sum*currentNumber);
    }
}
```

Notice that in the call return factorial_i(currentNumber - 1, sum*currentNumber); both parameters are immediately resolvable. We can compute what each parameter is without waiting for a recursive function call to return. This is not the case with the previous version of factorial. This streamlining enables the compiler to minimize stack use as explained above. *Thanks to Jon Bartlett for the example.*

Some definitions (types of recursion):

- Tail Recursion: A call is tail-recursive if nothing has to be done after the call returns. I.e. when the call returns, the returned value is immediately returned from the calling function. More simply, tail recursion is when the recursive call is the last statement in the function. See Tail Recursion.
    - advantages of tail recursion
    - advantages of tail recursion (2)
- Head Recursion: A call is head-recursive when the first statement of the function is the recursive call.
- Middle or Multi Recursion: A call is mid-recursive when the recursive call occurs in the middle of the function. I.e. there are other statements before and after the recursive call. If one or more of these statements is another recursive call, then the function is multi-recursive. There is no essential difference between Head Recursion, Middle Recursion and Multi Recursion from the standpoint of efficiency and algorithm theory.
- Mutual Recursion: Function X and Y are called mutually-recursive if function X calls function Y and function Y in turn calls function X. This is also called indirect recursion because the recursion occurs in two steps instead of directly. See Mutual Recursion.

Some Links

More explanations about recursion, Towers of Hanoi, etc.