Recursion

Mathematically, a recursive function is one that is defined in terms of other values of that same function. A couple (overused) examples are as follows:

Fibonacci numbers:

F(0)=0, F(1)=1, F(n) = F(n-1)+F(n-2), for all n>1. (F is defined for all non-negative integers here.)

Factorials:

Fact(0) = 1, Fact(n)=n*Fact(n-1), for all n>0. (Fact is also defined for all non-negative integers.)

Analogously, a recursive method is one where the method body contains a call to the method itself. (Remember, each instance a method is called, the computer allocates some memory for THAT instance to run. There is no reason why two DIFFERENT instances of the same method can not be in the middle of running at the same time.)

Incidentally, here are Java methods that compute the two recursive functions listed above:

```
public static int Fib(int n) {
        if (n<2)
                return n;
        else
                return Fib(n-1)+Fib(n-2);
}

public static int Fact(int n) {
        if (n ==0)
                return 1;
        else
                return n*Fact(n-1);
{
```

What would make both of these methods more robust?
What problem becomes more probable with recursion?

When dealing with recursion, there are essentially two issues to tackle:

1) How to trace through recursive code.
2) How to write recursive code.

Tracing through recursion

Clearly, the first is easier than the second, and you can't do the second if you can't do the first. So, let's talk about tracing through recursion first.

Really, there are no NEW rules to use when tracing through recursion, as compared to tracing through code that has method calls. Anytime a new method is called, you checkpoint and STOP running the calling method, and start executing the callee method. When the callee has terminated, it returns a value to the caller, and then the caller continues execution EXACTLY where it left off.

The stack trace illustration in many intro CS books uses is a fairly good model to use when tracing recursion. I often explain a trace using a physical stack of papers, where each paper is allowed to keep track of one method call. Let's trace through a couple simple examples of the two functions above.

Writing recursion

The toughest part with writing a recusive method ISN'T coding the method. Rather, the most difficult part is coming up with the recursive algorithm to solve the problem. Once you come up with the idea for the algorithm, the actual code is not so difficult to write. (As you can see, the mathematical definition of both fibonacci and factorial look amazingly similar to the code. Once you can get this mathematical definition, the code is not far off.)

The most difficult part of creating a recursive algorithm is finding a way to solve the given problem that involves a solution to a problem of the exact same nature. In both the Fibonacci and Factorial examples, the functions themselves are defined recursively, so we don't have that problem. Here is a problem that is not typically defined recursively:

Find the sum 1+2+3+...+n.

Your natural inclination is probably to write a loop to determine this sum. In order to come up with a recursive solution, you first have to think of a mathematical way to express the function above, recursively.

The big key is that 1+2+3+... = n + (1+2+3...+n-1)
Thus, if we let s(n)=1+2+3+...+n, then we can derive that

s(n) = n + s(n-1), for all n>1, s(1)=1.

Now using the information above, the recursive method that computes the function s above is apparent.

Once you have found a recursive algorithm to solve a problem, there is one other main issue to consider before coding a recursive method: For what values should the recursive algorithm be executed, and for what values should it NOT be executed, because the value to return can be easily computed?

This is known as the terminating condition. All recursive methods have to have a terminating condition in some shape or form. If they didn't, you would get infinite recursion. (This is very similar to an infinite loop.)

There are two basic constructs that most recursive methods take. Here they are:

if (terminating condition)
finish task
else
solve problem recursively

if (not terminating condition)
solve problem recursively

One other way to view coding a recursive problem is the following:

Imagine that you have been asked to write a method, but that someone else has already written the method and you are allowed to make as many calls to THAT method that you want to, as long as you don't pass it the same parameters that have been passed to you.

Thus, when computing Fact(n), for example, instead of doing all that multiplying yourself, you call the function your friend has written to calculate Fact(n-1). When he/she returns that answer to you, all you have to do to finish your job is multiply that answer by n and you are done!!!

Recursive Binary Search

A binary search is when we are given a sorted array and asked to find if a value is stored in that array. To write this recursively, we have to add two parameters to our method: the low index of our search and the high index of our search. Thus, the goal of our method is as follows:

Given a sorted array vals, a low index low, a high index high and a target t, determine if t is stored within the sub-array vals[low….high].

Here the key is that we have two terminating conditions: having no search space, and if the middle element (the one that is stored in the index halfway between index low and index high) is the target.

After that, based on our comparison with the middle element, we can make one of two recursive calls for a search in either the lower or upper half of the array.

```
public static boolean search(int vals[], int low, int high, int t) {
    if (low > high)
        return false;
    int mid = (low+high)/2;
    if (vals[mid] == t)
        return true;
    else if (vals[mid] < t)
        return search(vals, mid+1, high, t);
    else
        return search(vals, low, mid-1, t);
}
```

Introduction to Towers of Hanoi

The story goes as follows: Some guy has this daunting task of moving this set of golden disks from one pole to another pole. There are three poles total and he can only move a single disk at a time. Furthermore, he can not place a larger disk on top of a smaller disk. Our guy, (some monk or something), has 64 disks to transfer. After playing this game for a while, you realize that he's got quite a task. In fact, he will have to make 2^{64} - 1 moves total, at least. (I have no idea what this number is, but it's pretty big...)

Although this won't directly help you code, it is instructive to determine the smallest number of moves possible to move these disks. First we notice the following:

It takes one move to move a tower of one disk.

For larger towers, one way we can solve the problem is as follows:

1) Move the subtower of n-1 disks from pole 1 to pole 3.
2) Move the bottom disk to pole 2.
3) Move the subtower of n-1 disks from pole 3 to pole 2.

We can now use this method of solution to write a function that will print out all the moves necessary to transfer the contents of one pole to another. Here is the prototype for our function:

```
public static void towers(int n, int start, int end);
```

n is the number of disks being moved, start is the number of the pole the disks start on, and end is the number of the pole that the disks will get moved to. The poles are numbered 1 to 3. Here is the function:

```
public static void towers(int n, int start, int end) {

    if (n > 0) {
        int mid = 6 - start - end;
        towers(n-1, start, mid);
        System.out.print("Move disk "+n+" from tower ");
        System.out.println(start+" to tower "+end+".");
        towers(n-1,mid,end);
    }
}
```

Recursive Problem to Solve

Write a recursive method to return the sum of the digits of a non-negative integer n.

```
// Precondition: n >= 0.
public static int sumDigits(int n);
```