Recursion = Stack + Loop

The following might not help you write recursive functions, but it should help you understand what happens when recursive functions are executed. We are going to simulate recursive calls by pushing and popping a stack.

The data on the stack will be a *Frame* for the current call. A frame holds 3 kinds of data:

A. Arguments for the current call.
B. Local variables for the current call.
C. A marker I call "nextInstruction" for where to resume executing when we return from the call.

Clicker Q: Consider a recursive function that returns something, like

```
    int foo(int n) {
        …
        int x = foo(n-1)
```

Where does the program return to when it has the value of foo(n-1) ?

A. To foo(n)
B. To the middle of the line int x = foo(n-1)
C. To the line after int x = foo(n-1)
D. To the end of the program

Consider the factorial function:

```
public int fact(int n) {
        // marker 'A'
        if (n == 0)
                return 1;
        else {
                int f = fact(n-1);
                // marker 'B'
                return f*n;
        }
}
```

We do this in a while loop. If our current position is at marker 'A', we test n for 0 and either simulate returning 1 or recurse on n-1. If we are at marker 'B', we get the value that was just returned, save it in the current frame's variable f, then simulate returning the current frame's f times the current frame's n.

The only variables for our simulator are the stack, the current position and a variable to hold the value just returned. Everything else is save in the stack frames, so we can have multiple copies of them.

Here are the protocols for simulating calls and returns.

TO SIMULATE A CALL: Push a new stack frame with the arguments and the appropriate next instruction marker. Make the current instruction marker be the start of the function.

TO SIMULATE A RETURN: Copy the nextInstruction marker from the frame at the top of the stack. Put the value to be returned in the return variable. Pop the stack.

For example, three lines from our factorial function are

```
int f = fact(n-1);
// marker 'B'
return f*n;
```

We will simulate the call with

```
Frame top = stack.top();
stack.push( new Frame(top.n-1, 'B') );
currentInstruction = 'A'
```

If we are at marker 'B' we first put the return value into f, then return f*N:

```
top.f = returnValue;
currentInstruction = top.currentInstruction;
returnValue = top.f*top.n;
stack.pop();
```

At the top level we push a frame with a special marker that means "really return the return value."

```
static int stackFact(int n) {
        Stack<Frame> stack = new Stack<Frame>();
        char nextInstruction = 'A';
        int returnValue=0;
        stack.push( new Frame(n, 'Z'));

        while (nextInstruction != 'Z') {
                Frame top = stack.top();
                if (nextInstruction == 'A') {
                        if (top.n == 0) {
                                nextInstruction = top.nextInstruction;
                                returnValue = 1;
                                stack.pop();
                        }
                        else {
                                stack.push( new Frame(top.n-1, 'B'));
                                nextInstruction = 'A';
                        }
                }
```

```
          else  if (nextInstruction == 'B') {
                    top.f = returnValue;
                    nextInstruction = top.nextInstruction;
                    returnValue = top.f*top.n;
                    stack.pop();
          }
}   // ends while(nextInstruction != 'Z')  {
return returnValue;
```

Here is the loop for

```
public int Fib(int n) {
        // Marker 'A'
        if (n == 0)
                return 0;
        else if (n == 2)
                return 1;
        else {
                int t1 = Fib(n-1);
                // Marker 'B'
                int t2 = Fib(n-2);
                // Marker 'C'
                return t1+t2;
        }
}
```

```
while (nextInstruction != 'Z') {
        Frame top = stack.top();
        if (nextInstruction == 'A') {
                if (top.n == 0) {
                        nextInstruction = top.nextInstruction;
                        returnValue = 0;
                        stack.pop();
                }
                else if (top.n == 1) {
                        nextInstruction = top.nextInstruction;
                        returnValue = 1;
                        stack.pop();
                }
                else {
                        stack.push( new Frame(top.n-1, 'B'));
                        nextInstruction = 'A';
                }
        }
```

```
          else  if (nextInstruction == 'B') {
                    top.t1 = returnValue;
                    stack.push( new Frame(top.n-2, 'C'));
                    nextInstruction = 'A';
          }
          else if (nextInstruction == 'C') {
                    top.t2 = returnValue;
                    returnValue = top.t1 + top.t2;
                    nextInstruction = top.nextInstruction;
                    stack.pop();
          }
} //  ends while (currentInstruction != 'Z')
return returnValue;
```

I will post complete programs that give stack-based implementations of the Factorial, Fibonacci, and Towers of Hanoi functions.