COSC 1P03 Lab 4
Mar. 10-14, 2014

Exercise 1
Towers of Hanoi

Estimated Time: 30 min

Towers of Hanoi is a pretty old problem, that you've probably seen several times before. You have three pegs, with discs stacked on the first peg in descending sequence of size, and wish to move all of the discs to the third peg, with the following restrictions:

1. You may only move one disc at a time
2. You may only move the top disc of any peg (and may only place it on the top of any stack)
3. You may never place a larger disc onto a smaller disc

There are different ways to solve it (including simply trial and error, for smaller numbers of discs), but the recursive solution is particularly handy.
When trying to move ***n*** discs from one peg to another:

- If ***n*** is 1, simply move ***1*** disc from the *source* to the *destination*.
- Otherwise:
  - Move ***n-1*** discs from the *source* peg to the *other* peg (the one that's neither the source nor the destination).
  - Move ***1*** disc from the *source* peg to the *destination* peg.
  - Move ***n-1*** discs from the *other* peg to the *destination* peg.

You should be able to see the trivial case fairly easy here, as well as the recursive call(s).
Download the sample code from the course webpage, and fill in the recursive solution.

Tips:

- You might notice that the sample code uses a Stack to represent each peg (which is logical, as you can only add and remove from the same end anyway); that there are three Stacks (one for each peg); and that the Stacks are held in a Vector. This is likely overkill, but a nice reminder of how to use Generics.
- The initial call will instruct it to move N from 0 to 2 (the left to the right). Of course, in that case, the middle peg is 1. However, that will not always be the case; clearly you can't hardcode the 'other' to be 1. So, the tip? Collectively, the pegs will always be one each of 0, 1, and 2. They'll always add up to a sum of 3. So, 3 minus the sum of 'from' and 'to' will always yield your 'other' peg's index.
- It *might* seem like you should have two recursive calls within the solve method (two calls with size n-1); actually, have three. For the middle step, actually still have it call itself with size 1. It'd be entirely functional either way, but this limits the redundancy in the code.

```
Moving 3 from 0 to 2.
Moving 2 from 0 to 1.
Moving 3 from 2 to 1.
Moving 1 from 0 to 2.
Moving 3 from 1 to 0.
Moving 2 from 1 to 2.
Moving 3 from 0 to 2.
Done!
```

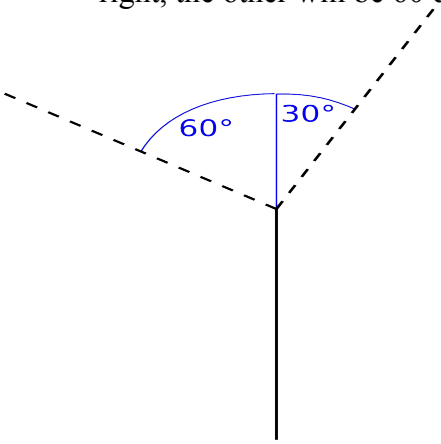
Exercise 2
Trees!

Estimated Time: 30 min

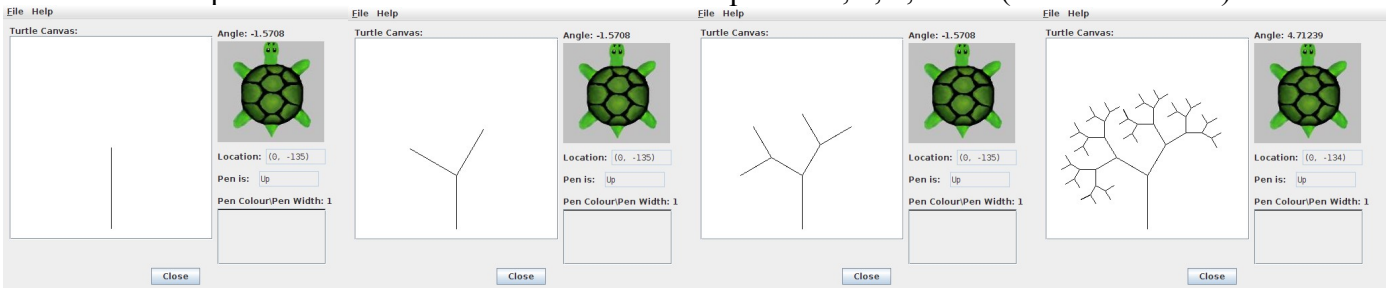
In class, we looked at Koch curves (the snowflakes). There are lots of interesting ways to draw things recursively, though. All we need to do is to decide on the base case, and the lines to draw (if any) and orientation in the recursive case. For example, let's try drawing a tree!
(Note: In case you aren't familiar with Turtle Graphics, a basic skeleton is premade for you; feel free to download it from the course website. You'll still likely have to refer to the documentation, though)

We can say that there are two possible shapes to draw in our tree:

- When we're drawing with a depth of 0, we're simply drawing a line segment of the specified size.
- When we're drawing with any larger depth than 0, we're drawing a line segment of 2/3 the specified size, and drawing two additional branches (with one fewer depth); one will be 30 degrees (or $\pi/6$) towards the right, the other will be 60 degrees (or $\pi/3$) towards the left.



Below are examples of how it should look with initial depths of 0, 1, 2, and 5 (all with size 120).



Tips:

- By default, the turtle speed is set to 'fast' in the code. Setting it to 0 makes it *far* faster (and also inappropriate, from a style standpoint), but don't do that until you have it working. Until it draws correctly, you'll want to be able to trace what's really happening.
- For the trivial case, don't forget to move forward *and return backward*; do the same for the recursive case (though the return trip is obviously by a different amount).
- Once you get it working, try it out with a depth of, say, 15!

Exercise 3
Ackermann Functions

Estimated Time: 30 min

Now that we have all of the icky fun stuff out of the way, it's time for everyone's favourite subject: math!

We talked about the Ackermann function in class, but didn't really try it out. There are a few different versions of this function, so let's use the Wolfram MathWorld version:

$$A(x,y) \equiv \begin{cases} y+1 & \text{if } x \text{ is } 0 \\ A(x-1,1) & \text{if } y \text{ is } 0 \\ A(x-1,A(x,y-1)) & \text{otherwise} \end{cases}$$

You can read more here: http://mathworld.wolfram.com/AckermannFunction.html

For this exercise, you have two tasks to perform:

1. Write a class that includes an implementation of the Ackermann function. Try it out on a few values
   - Note, it should return its computed value; not print it! (Do the printing in your constructor)
   - Once you have it written, see if it works by testing on a few values (refer to the tips below!)
2. Once you have it written the function, add a counter to see how many times the function is being called. Print both the computed value and the number of calls to the screen (see example below).

Tips:

- You might want to write an additional method (e.g. test(x,y)) that just resets the counter, calls your Ackermann function and also handles printing.
- This function can grow surprisingly fast! For example, with an x of just 3, you might be able to get away with a y value of 10 or 11, but probably not much more than that.
  - The cause for this is the *stack overflow* that occurs when it tries to recurse too deep
  - There's little you can do to fix this (you *could* try to adjust the memory allocated to the Java Virtual Machine, but no matter how large you made it, you'd still be hitting it pretty soon)
- Your counter is simply keeping track of how many times the Ackermann function is being called; it doesn't directly indicate depth. If you have spare time after completing the lab, you might want to try adding a third parameter that lets you keep track of depth. Of course, this could also speed up how soon you hit a stack overflow (as each stack frame would then be just a bit larger).

```
Testing on (0,0): 1 [1]
Testing on (1,0): 2 [2]
Testing on (0,1): 2 [1]
Testing on (1,3): 5 [8]
Testing on (2,20): 43 [945]
Testing on (2,100): 203 [20705]
Testing on (3,1): 13 [106]
Testing on (3,5): 253 [42438]
Testing on (3,8): 2045 [2785999]
Testing on (3,9): 4093 [11164370]
Testing on (3,10): 8189 [44698325]
Testing on (3,11): 16381 [178875096]
```