CMPS 10 Lecture Notes: Lecture 19 (3-8-2016)

Tower of Hanoi

- Remember, the idea is that we have some number of pegs (the default is usually three), and some number of discs (again, typically three in the basic scenario). The discs all start off of on one peg, and the goal it to move all of them to a different peg. The catch: a bigger disc can never be on top of a smaller disc.
- So, say, if the goal is to move all of them from peg A to peg B, we can tell that FIRST we need to move the biggest disc to B, as our first goal.
  - And to achieve that, it means that we need to "clean off" all of the discs on top of the bottom disc (so that it is free to be moved), and we need the destination peg to be clean (because if there was a smaller disc there already, we wouldn't be able to move the big disc there).
    - So our first steps are to move first disc from A to B, and then the second disc from A to C, and then move disc 1 from B to C.
      - Now peg B is free and open for disc 3 to go on top of it!

Given some code, takes four inputs

- First: number of discs
- Second: Source Peg
- Third: Destination Peg
- Fourth: Helper

The general idea:

- If you have some number of discs K, and you want to move them from a source to a destination (say source is A, destination is B)
  - To achieve this, you want to take k - 1 discs and move them from the source to the helper, and then move 1 disc from the source to the destination.
    - And THEN move the k - 1 discs from the helper to the destination, using the source as a helper.

Base Case: You have nothing to do! If discs is empty, you have nothing to do!
Note that when it says "move disc X" it didn't need to say the number, because the ONLY legal move is uniquely specified, because it is only ever "the top disc" that can be moved.

- But it is nifty that the code can still manage to keep track of the disc without it needing to be specified.

Analyzing how long it takes to solve Hanoi

The amount of "moves" it takes to solve a Hanoi problem of size N: it takes Two Times the amount of times it takes to solve a Hanoi problem of N - 1, plus one explicit move.

- The above in mathematical terms: H(n) = 2H(n 1) + 1
- And if we have H(0), then that takes 0 moves.
  - In English: For every n >= 1, the number of moves you must perform to solve a problem of size n, is the number of moves for a problem of size n - 1, + 1 move (which is for the bottom peg) + another number of moves of problem size n - 1
    - It's a similar to knowing that (x+y)^2 is equal to x^2 + y^2 + 2xy. It's an identity!

Let's go through it as an example. Let's say that we have three discs, so it is a problem of size three.

- Moves(3) = 2Moves(2) + 1 = 4(2Moves(0)+1) + 2 + 1
- 2[2Moves(1)+1] + 1 = 8Moves(0) + 4 + 2 + 1
- 4Moves(1) + 2 + 1 = 7
  - Another way to think of it: 2^numberOfDiscs - 1. (So when number of discs is 3, that is 2^3 - 1, or 8 - 1, or 7.

So the professor's two big claims:

- Moves(n) = 2Moves(n 1) + 1
- Moves(n) = 2^n - 1
  - So we have two values for Moves(n). If this is TRUE, then it means that they have to be equal to each other! We can verify that one quantity satisfies the formula! So we plug in 2^n minus 1 to our formula, like so:
    - 2^n minus 1 = 2(2^(n 1) + 1)
    - =2^n minus 2 + 1
    - =2^n minus 1
      - Aha! we did it!

Final Project Clues

Final project for the class: piece of code that performs towers of hanoi visually.

- Exactly the same code, but the only difference is that instead of that line being an explicit command like "say" or "put in list"...
  - You call some other procedure called move disc that will perform the visualization of moving the disc number BLAH from source to destination.
    - And in this example, that "move disc" function is simply the say that we had before. But for the final project, you'll need to fill that in with some beautiful graphical representation!
      - For code: the animation, the movement, doesn't matter. BUT showing the sequence of pegs is VERY important!
      - In order to do the visualization, **you need to KEEP TRACK of the state of each of the pegs.**
      - You need to keep track of which discs are on each peg, as a result of every move.
      - So if you have three discs, and you have your A peg, your B peg, and your C peg...
      - And you begin with A having 1,2,3, and B and C are empty...
      - And then the first command comes that says "move disc 1 from A"
      - Your code should REMOVE the disc from it's source list, and ADD it to the destination list.
      - So there are two ways of doing it: one way is just erasing the whole thing and redrawing the state each step (RECOMMENDED). The other is to literally erase the peg that you are moving from its source spot, and redraw it on the new peg (This is WAY HARDER).

This is optional in the following sense:

- 40 percent homework
- 30 percent final
- 30 percent midterm
- EXTRA 15 percent for doing this visualization.
  - So if you've already got a 100 percent from everything else, good for you.
    - But this is a way to bump up your grade.
      - In other words: it is extra credit.

Beautiful Recursion Example: Nice Tree

Then professor showed us a really beautiful tree recursion.

- If we run it multiple times, we get multiple different trees.
- Let's take a look at this code, because it is a fantastic example of recursion.
  - When green flag is clicked, do something. "Hide" makes the sprite go away. And "set size to 10 percent" makes your sprite a lot smaller. It just changes the size of the final leaf that gets placed.
  - Clear cleans the screen, pen up lifts the pen from the paper.
  - Go to x: y: just puts the cursor somewhere, point in direction goes in certain way.
  - We put the pen down on the paper, we set the pen color to brown (note that everything is brown except for the final stamp which is made with the sprite).

        * And it is kind of remarkable how our eye fools us.
            · We get the impression that a branch is uniformly thinning as we approach the leaves. But in actuality, it is a sequence of segments, and any given segment has a consistent thickness.
            · And then, finally, draw 10 tree with branch 67. THIS is the interesting part of the code.

So let's look at that last bit carefully.

- Draw tree with specified number of levels (stored in variable "levels"), and length stored in variable "length."
- Then it is pick two random numbers.
- And if you have gotten down to the 0 level, just place the stamp.
    - OTHERWISE, we go into the else. And if we ignore what happens in the blue things for a moment, what this is saying is that to draw a tree with a certain number of levels, (of levels one less than what we started with).
        * So this is actually a full binary tree of depth 10!
            · You always draw two subtrees of depth n - 1!
            · That's step 1! But then the question is, what are those two trees going to look like!

How do the trees look?

- Well, you draw a line, and then you turn left by some amount of randomness (that's our first random number), and then you just draw a new tree in this rotated direction!
- And then once you are done with that, you'll go back to where you were before, and turn right by some number.
    - The branches get SHORTER because length goes down a little bit each time.
    - The branches get THINNER because the pen size is set to whatever the current level is.

And then something happened in class that I (Ben) think is one of the most valuable takeaways from this entire course: the professor, Pardis, myself, and many voices in the room all worked together to try to figure out how to get a certain result from the code, and most of the time during this process we were confused and surprised by what was happening. Debugging is **the great equalizer** – it doesn't matter if this is your first time programming or if you've been doing it for years and years, sometimes your code will confuse and surprise you, and it **just takes time** to go through it and think about it rationally to figure out what the issue is. Sure, debugging can sometimes feel frustrating or confusing, but at least take solace in the fact that it is **normal**, and that **every programmer does it**, regardless of skill or experience level!

That was something I wish I had known when I was your age. So there you go.