CMPS 10 Lecture Notes: Lecture 17 (3-1-2016)

IF YOU LEARN NOTHING ELSE FROM THE CLASS LEARN THIS:

- When you have an experiment that involves n coins flips, the probability of deviation from the average, the correct measure, is NOT percent. If you flip a coin four times, you have 16 total possibilities.
  - And of those 16, only 6 of them are "balanced" (i.e. an equal number of heads and tails).
    - So if you flip a few coins, the probability that it will be balanced, or close to balanced, becomes huge.
    - But as you increase the number of trails, you can ask how within square root of n you are away from the average.
      - Saying that you are going to be within 1 percent of average is actually not as precise as you can be, since 0.01 * n is much much bigger than the square root of n.

RECURSION

Now we are going to talk about RECURSION!
Can someone remind the professor what is a function: An input gives you an output.

- A mapping from a domain to a range.
- And usually, f is a mapping from one set to another (or f: D >R) where D is the Domain set, R is the Range set.
- So in general, we provide an input (something from the domain) and the function gives us an output (something from the range)
  - It's totally fine for multiple things in the domain to point to the same thing in the range. It is VERY BAD for a single thing in the domain to point to multiple things in the range.

A function represents a definite relationship between input and output.

- Function can be stupid. It can be indifferent to the input.

We have been primed to think of input like this: f(x) = 2x^2 + 5x - 7

- When we see this, why are we sure that it is a function.
  - Example of not a function: f(x) = "The number whose square is x" this is no good, because if we plugged in 9 there would be two answers, +3 and negative 3, the "the" is cheating.
    - Definiteness is very important.

So let's define an example recursive function!

- We'll work with the Natural numbers, N (that is, the numbers 0,1,2,3,....)
- So we'll say that our function, f, has the domain N (so the only inputs we can give it are the natural numbers)
- And then we specify that if the input is 0, we return 1: f(0) = 1
- Similarly, we specify that if the input is 1, we return 2: f(1) = 2
- So that covers the case for our input being 0 and 1. But for All N greater than or equal 2, we'll figure it out with this function: **f(n) = f(n - 1) + f(n - 2)**

And so to solve for, say, f(5), we just PLUG IT IN!

f(5) = f(5 - 1) + f(5 - 2) = f(4) + f(3) And so we see that in order to solve f(5), we first need to solve for f(4) and f(3).

f(4) = f(4 - 1) + f(4 - 2) = f(3) + f(2) And now HERE we see that to solve for f(4) we need to solve for f(3) (which we were gonna do anyway), but also for f(2)

f(3) = f(3 - 1) + f(3 - 2) = f(2) + f(1) To solve f(3), we'll also need to solve for f(2). We also need to solve for f(1), BUT that was just told to us in the function description: f(1) = 2.

f(2) = f(2 - 1) + f(2 - 2) = f(1) + f(0) And here we see that f(2) breaks down to f(1) and f(0), which we know is 2 and 1! And now that we know all of this, we can add everything up for the initial question of f(5)

And now we have successfully broken everything down into it's tiny parts! We can add it up!

In order for something to be a function, it must be well specified. A function is a contract. The contract has a specification. You are only allowed to give me sets of this input.

- A function does not have to be specified directly.
  - It doesn't necessarily need to be such that you just specify it directly.
    - As we just saw with the recursive function above, we figure out any given value by knowing the two values that came before it.

Let's take a look at a nice little algorithm.

- Imagine that we have an ARRAY of numbers, that we'll call A. A has n numbers inside of it.
- Now imagine that we have a nice little function called SORT (I) that always takes an array as input.
  - If the length of A is less than or equal to 2, then check if they are in order, and swap is needed.
  - But the interesting part comes next:
    - OTHERWISE, split evenly into Ileft, and Iright
    - SORT(Ileft)
    - SORT(Iright)
    - MERGE(Ileft, and Iright).

So what do we mean by MERGE?

- Well, if we have two arrays, and they are sorted...
- It's pretty easy to produce a single array that contains all of the numbers in the two arrays in sorted order.
  - The way we do it is to start at the beginning of both arrays, and compare the two elements, write down the smaller one, and then move the pointer of the array it came from by one.
- And what's kind of nice is that it isn't THAT hard to express the action of merging in code.
  - So we can sort an array of arbitrary size
    - Even though we never REALLY talk about sorting.

What would this all look like?

- Let's say we have an array of size 16.
  - we keep on splitting it up by half each time (so we end up with arrays of length 8, and then 4, and then 2).
    - When we get to two, sorting is trivial (it is just a single check!)
    - And then we simply merge the pieces back up the tree again!
    - And we don't even need to know ahead of time how many times we'll have to split! We can just say "keep splitting!"