CS 315 - Day 04, Iteration and Recursion

This is a great textbook, comprehensive and thorough. It has the "classic" sorting and searching algorithms, and the "new wave" algorithms focused on internet and data science applications.
Be sure to read it closely. https://www.facultyfocus.com/articles/teaching-and-learning/what-textbook-reading-teaches-students/ . But, I'll warn you that reading the textbook requires effort and engagement. It won't be easy, but it's worth it.

What is an algorithm? ***A well-defined set of instructions which transform <u>input</u> into <u>output</u>.*** An algorithm is ***not*** a program, we implement algorithms using code, and we will analyze the expected performance of the algorithm and compare it to the actual performance of the implementation. There are several implementation details which could affect performance, such as the compiler, the type of CPU, the amount of memory, and so on.

Real-world metrics for: Peanut Butter, Football, Selection Sort…

For the most part, the ***metric*** we use for analyzing our algorithms will be expected run-time. At a few points in the semester we will consider other metrics, like memory usage and access. And we are interested in ***asymptotic efficiency*** of the algorithms, which is "*how the running time of an algorithm increases with the size of the input, **in the limit**.*" Generally an algorithm that is more efficient asymptotically is the best choice for all but the smallest inputs.

Why does it matter? Computers are so fast now.
Well… what are some sorting algorithms you recall from CS 240? ***n^2*** vs. ***n*lg(n)***. Note on $\log_{10}()$ vs. $\log_2()$ vs. lg().
By the way, the logarithm of any finite base ***b*** is related to the logarithm of any other base ***c*** by a constant. Basically it doesn't matter what base we use when considering asymptotic behavior. In this class we will use `log`$_2$`()`, which we write as `lg()`.
$log_2(x) = \frac{log_{10}(x)}{log_{10}(2)} = c \cdot \ log_{10}(x)$, where the constant $c = \ log_{10}(2)$

When you develop software professionally, you typically have to specify its performance before writing the code.

One of the techniques we will use in this class is recursion. It is one of the most used techniques for algorithms.

A recursive routine calls itself, has a stopping criteria (base case), the recursive call should be simpler. The recursion is solving smaller versions of the same problem, "divide and conquer."

By the way, Mathematical Induction (MA 116) is analogous to recursion. There is a base case, and an inductive step, but generally inductive proofs are involve a predicate $P(k)$ and the next largest (rather than smaller) predicate $P(k+1)$.

Recall the factorial, $n!$ It's defined as $n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$.

So $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$

How could you frame 5! recursively? $5! = 5 \cdot 4!$ But $4! = 4 \cdot 3!$, and so on, all the way to $1! = 1$ which is the base case.

How would you implement this? `factorial.py`

```
def fact(n):
    if n==0:
        return 1
    else:
        return n*fact(n-1)
```

In-class group work will be an element in this course. The pedagogical idea is "***Social Constructivism***," the idea that we learn by working with others (*learning from / teaching to*).

So, brainstorm with two or three of your neighbors, and propose algorithms for forming in-class groups. Before starting, each share (1) name, (2) POE, (3) reason for taking this class, and (4) pick someone who will report out the algorithm.

Inputs used in your group-forming algorithm should be <u>easily</u> accessible:

- number of students in class.
- number of desired teams (or size of teams), "counting off" is fine.
- student's identifying information (easy, no lookups):
    - student id, residence hall, initials, birth month, birth day of month, etc.

Implement one of the algorithms to form new groups.

Before starting, each share (1) name, (2) POE, (3) reason for taking this class, and (4) pick someone who will report out.

Here is the question for your group:
<u>How could you determine the number of rows in Alumni Hall (A100) if:</u>

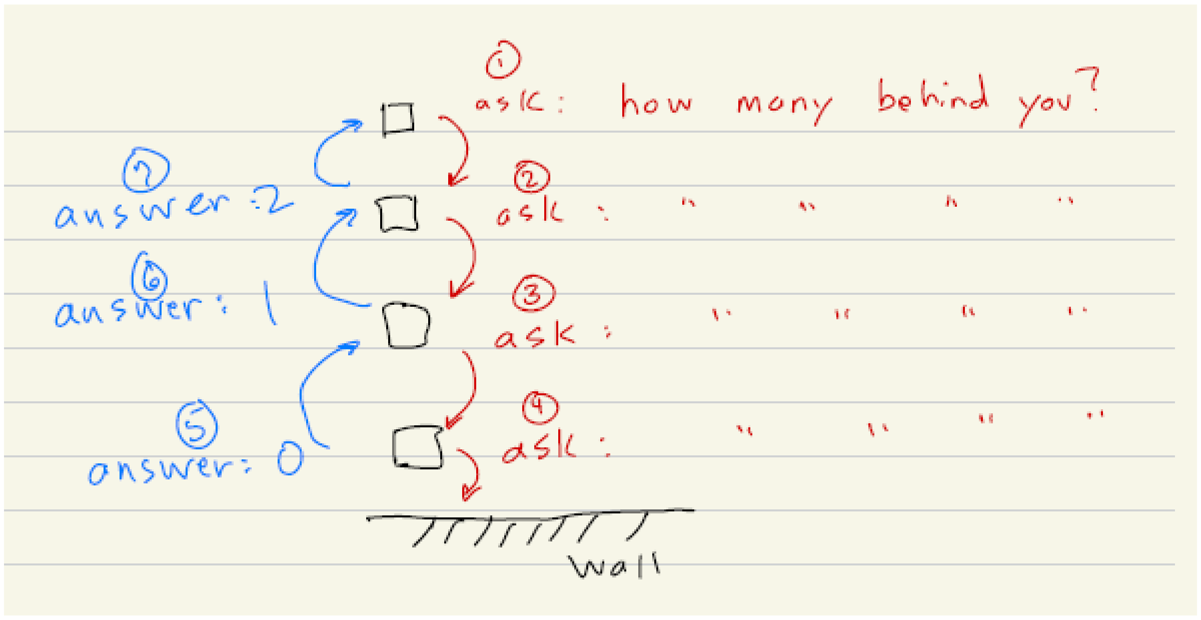
- The room is full.
- You are aware that you are in the front row, and there are no rows in front of you.
- The students in the back row are aware that there are no other rows and the wall is behind them.
- Students in Alumni Hall can only look back (or look forward) one row.
- Students in Alumni Hall can only talk to the person in front of them or behind them.
- There is no counting allowed, and there are no row or seat numbers.
- No one can look around, or walk up and down, and count rows.

Think recursively.

Report out.

Try a solution in class.

Here's a solution:



So, it ends up that the person in the front is told that the person right behind them has two people behind them. They know to add one more for the person behind them and one more for themselves, in order to answer the question, "How many are in this row?"

```
ask_row_behind():
    if (nothing):
        return 0
    else:
        return ask_row_behind() + 1
```

Lab time.