Topics for next two classes

- Introduction to recursion
- Recursive definitions
- Recursive functions
  - How to approach formulation and implementation

Reading: Zelle, Chapter 13.2

What is recursion?

- Have you had someone tell you that you can’t use a word in its own definition? This is a *circular* definition.
- A description of something that refers to itself, has a base case, and is defined on smaller instances is called a *recursive definition*.
- A function calling itself is called a *recursive function*
- Why use recursive functions?
  - Recursion allows simple definitions
  - Powerful programming technique
Recursive definitions

- **Factorial**

\[
\begin{align*}
    n! &= \begin{cases} 
    1 & \text{if } n = 0 \\
    n(n-1)! & \text{otherwise}
    \end{cases} \\
    n! &= n(n-1)(n-2)\ldots(1)
\end{align*}
\]

- \( \text{fac}(n) = n \times \text{fac}(n-1) \)

- Recurrences like \( t_1 = 5; \ t_n = 2t_{n-1} + 5 \)

- Recursive definitions need a termination condition

More on recursion

- Recursive programming is directly related to mathematical induction, a common proof technique

- Termination condition = base case

- Recursive definition are not circular because they work on smaller and smaller problem sizes and finally reach the base case

- When the base case is encountered, the expression can be computed

- Functional programming languages (e.g., Scheme, Haskell, Lisp) use recursion for iteration
Clicker question

1. \( \text{sum}(0) = 1 \)
   \[ \text{sum}(n+1) = \text{sum}(n) + 1/(n+1)! \]

2. \( \text{sum}(0) = 0 \)
   \[ \text{sum}(1) = 3 \]
   \[ \text{sum}(n) = \text{sum}(n+1) + \text{sum}(n-2) \]

3. \( \text{sum}(0) = 1 \)
   \[ \text{sum}(n) = \text{sum}(n-2) + n^2 \]

Which ones are incorrect recursive definitions?

A. 2
B. 2 and 3
C. 3
D. All are incorrect
E. All are correct

Two functions computing \( n! \)

```python
def non-rec-fact(n):
    fact = 1
    for factor in range(n, 1, -1):
        fact = fact * factor
    return fact

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

factorials.py
Recursive factorial

>>> fact(4)
24
>>> fact(10)
3628800
>>> fact(100)
9332621544394415268169923885626670049071596826438
162146859296389521759999322991560894146397615651
82862536979208272237582511852109168640000000000000
0000000000000L

Remember: each call to a function starts that function anew, with its own copies of local variables and parameters.
Recursive fact(n)

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

Not a particularly useful program as the iterative version is more efficient and $n!$ grows too quickly to use in computations

Recursive definitions: Fibonacci

- **Fibonacci Numbers**
  - $F(n) = F(n-1) + F(n-2)$, $F(0)=0$ and $F(1) = 1$
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711 …
  - named after Leonardo of Pisa, but first described by an Indian mathematician (about 300BC)
  - Closed form solution exists
    $$F(n) = \frac{\varphi^n - (-\varphi)^{-n}}{\sqrt{5}},$$
    where $\varphi$ is the golden ratio; i.e., a root of $x^2 = x + 1$, $x^2 - x - 1 = 0$