Lecture 15:
Recursion
(Sections 5.8-5.10)

CS 1110
Introduction to Computing Using Python

[E. Andersen, A. Bracy, D. Gries, L. Lee, S. Marschner, C. Van Loan, W. White]
Recursion

Recursive Function:
A function that calls itself

(see also Recursive Function)

Two parts to every recursive function:
1. A simple case: can be solved easily
2. A complex case: can be made simpler (and simpler, and simpler... until it looks like the simple case)
Russian Dolls!
What is the simple case that can be solved easily?

A: The case where the doll has a seam and another doll inside of it.
B: The case where the doll has no seam and no doll inside of it.
C: A & B are both simple
D: I do not know
import russian

d1 = russian.Doll("Dmitry", None)
d2 = russian.Doll("Catherine", d1)
def open_doll(d):
    '''Input: a Russian Doll
    Opens the Russian Doll d '''
    print("My name is " + d.name)
    if d.hasSeam:
        inner = d.innerDoll
        open_doll(inner)
    else:
        print("That's it!")
Examples

- Russian Dolls
- Blast Off!
- Towers of Hanoi
Blast Off!

blast_off(5) # must be a positive int
5
4
3
2
1
BLAST OFF!

blast_off(0)
BLAST OFF!
Blast Off!

```
blast_off(5) # must be a positive int
5
4
3
2
1
BLAST OFF!
```

What is the simple case that can be solved easily?

```
<table>
<thead>
<tr>
<th></th>
<th>A: negative n</th>
<th>B: positive n</th>
<th>C: n == 0</th>
<th>D: n == 1</th>
<th>E: I do not know.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
blast_off(0)
BLAST OFF!
```
def blast_off(n):
    """Input: a positive int
    Counts down from n to Blast-Off!
    """
    if (n == 0):
        print("BLAST OFF!")
    else:
        print(n)
        blast_off(n-1)
Tower of Hanoi

- Three towers: *left*, *middle*, and *right*
- $n$ disks of unique sizes on *left*
- **Goal**: move all disks from *left* to *right*
- Cannot put a larger disk on top of a smaller disk
1 Disc: Easy!

1. Move from *left* to *right*

*Solving for 1 tower is easy! That's the simple case!*
2 Discs: Step 1

1. Move from *left* to *middle*

*Thought: If I could get Disk 1 off of Disk 2, I could move Disk 2 to where it's supposed to go…. Moving 1 disk is easy!*
2 Discs: Step 2

1. Move from left to middle
2. Move from left to right

Thought: Now that Disk 1 is gone, I can move Disk 2 to where it's supposed to go.
2 Discs: Step 3 (final)

1. Move from *left* to *middle*
2. Move from *left* to *right*
3. Move from *middle* to *right*

---

**Thought:** Now that Disk 2, is where it's supposed to be, all I have to do is move Disk 1. Moving 1 disk is easy!
Thought: If I could get Disks 1 & 2 off of Disk 3, I could move Disk 3 to where it's supposed to go.... And I know how to move 2 Disks from the previous slide!
3 Discs: Moving **Disks 1 & 2** off of **Disk 3** (1)

1. Move from *left* to *right*
3 Discs: Moving **Disks 1 & 2** off of **Disk 3** (2)

1. Move from *left* to *right*
2. Move from *left* to *middle*
3 Discs: Moving **Disks 1 & 2** off of **Disk 3** (3)

1. Move from *left* to *right*
2. Move from *left* to *middle*
3. Move from *right* to *middle*
3 Discs: Move **Disk 3** to the Goal

1. Move from *left* to *right*
2. Move from *left* to *middle*
3. Move from *right* to *middle*
4. Move from *left* to *right*
3 Discs: Moving **Disks 1 & 2** to the Goal (1)

1. Move from *left* to *right*
2. Move from *left* to *middle*
3. Move from *right* to *middle*
4. Move from *left* to *right*
5. Move from *middle* to *left*
3 Discs: Moving Disks 1 & 2 to the Goal (2)

1. Move from left to right
2. Move from left to middle
3. Move from right to middle
4. Move from left to right
5. Move from middle to left
6. Move from middle to right
3 Discs: Moving **Disks 1&2** to the Goal (3)

1. Move from *left* to *right*
2. Move from *left* to *middle*
3. Move from *right* to *middle*
4. Move from *left* to *right*
5. Move from *middle* to *left*
6. Move from *middle* to *right*
7. Move from *left* to *right*
4 Discs: Oh, boy...

Thought: If I could get Disks 1&2&3 off of Disk 4, I could move Disk 4 to where it's supposed to go…. And I know how to move 3 Disks from the previous slide!
Rely on the solution for the simpler case

\[ \text{Hanoi}(4,L \to R) \]

\[ \text{Hanoi}(3,L \to M) \]
(uncover the big one)

move the big one

\[ \text{Hanoi}(3,M \to R) \]
(covers the big one)
solve_hanoi(n, start, goal, temp)

"""Prints instructions for how to move n disks (sorted small to large, going down) from the start peg to the goal peg, using the temp peg if needed. """

if n == 1:
    print("move from "+start+" to "+goal)
else:
    # need to move top n-1 disks from start to temp so that I can move
    # the bottom disk to goal... luckily, I have a function that does that!
    solve_hanoi(n-1, start, temp, goal)

    # move the bottom disk from start to goal
    print("move from "+ start +" to "+ goal)

    # now put everything back on the last disk at goal
    solve_hanoi(n-1, temp, goal, start)
Divide and Conquer

**Goal**: Solve really big problem P

**Idea**: Split into simpler problems, solve, combine

**3 Steps:**
1. Decide what to do for simple cases
2. Decide how to break up the task
3. Decide how to combine your work
Recursion vs Iteration

• Recursion is *provably equivalent* to iteration
  - Iteration includes *for-loop* and *while-loop* (later)
  - Anything can do in one, can do in the other
• But some things are easier with recursion
  - And some things are easier with iteration
• Will **not** teach you when to choose recursion
• We just want you to *understand the technique*