Topic 7: Recursion and Data Storage

Goals: By the end of this week, we will discuss…

- *what is recursion and how do we use it?*
- *why would we write a recursive function?*
- *what happens when functions are stored in memory?*

Acknowledgements: These class notes build on the content of my previous courses as well as the work of R. Jordan Crouser, and Jeffrey S. Castrucci.

Getting into recursion.

We have seen in previous weeks that functions can call other functions. So what if a function calls itself? Indeed a function can call itself and such a function is called a *recursive function*.

```
def nfact(n):
    if n == 0:  # Base case
        fact = 1
    else:       # Recursive case
        fact = n * nfact(n-1)
    return fact
```

We cannot just have a function call itself, or else it will loop forever. To prevent recursive functions for looping indefinitely, we add what is called a *base case*, which is a case for which the function will not call itself.

In the above example there is a call to the function `nfact` from inside the function `nfact`. Notice that when we call the function with a given *n*, the next call will be with parameter *n*-1.

Creating a recursive function

Generally when creating a recursive function we need to focus on two main steps:

- *Write the base case* - Every recursive function must have a *base case* that returns a value without performing a recursive call (i.e. stopping condition). A programmer may write that part of the function first, and then test.
- *Write recursive case* - The programmer then adds the recursive case to the function.

In the `nfact` example above, the base case is `n <= 0`. Designing a recursive function requires that you carefully choose a base case and make sure that every sequence of function calls eventually reaches a base case to prevent an infinite loop.

Let's look at another recursive example. This time we want to compute the factorial of n, when n is any integer value. The factorial of 5 (5!) for example is `5*4*3*2*1 = 120`.

```
def nfact(n):
    fact = 0
    if n == 1:  # Base case
        fact = 1
    else:
         print("Recursive call here")
    return fact

>>> nfact(5)
Recursive call here
0
```

Is this working as we expect it? (We want to make sure the base case works before writing the recursive case, so how would we test this?).

If we think this is working, let's see about the recursive call.

```
def nfact(n):
    if n == 1:  # Base case
        fact = 1
    else:       # Recursive case
        fact = n * nfact(n-1)
    return fact

>>> nfact(5)
120
```

Basic Structure of a recursive algorithm

- A **base case**: what to do in the simplest possible case (i.e. when you have a single disk)
- A **recursive step**: break the original problem into one or more smaller problems, and solve that (saving the intermediate result)

Recursion themes

- "Looping without a loop"
- "A function that calls itself as part of its definition"
- "Solving a problem by solving smaller instances"
- Key components of all three:
  - a recursive step (i.e. knowing when to split)
  - a "base case" (i.e. knowing when to stop)

Exercise:

Write a program to sum the numbers in a list.

Now rewrite the same program using recursion.

- What is the base case?
- What is the recursive step?

Now what about writing a program to add all the numbers in a list of list of lists (depth unknown)?

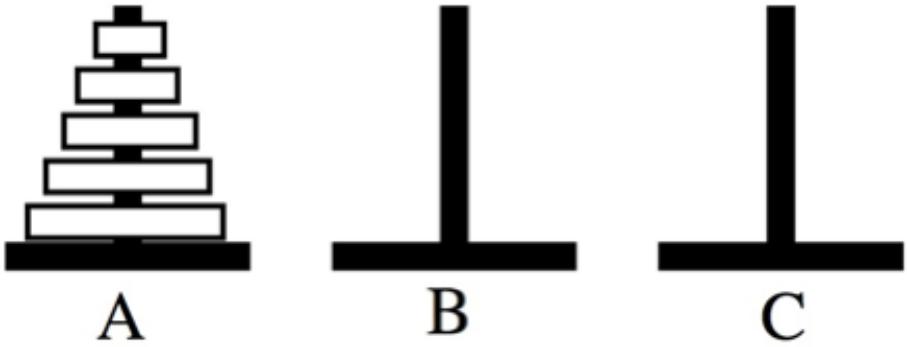
new What about a list of list of list.

Towers of Hanoi

The puzzle was invented by the French mathematician Édouard Lucas in 1883.

There is a story about an Indian temple in Kashi Vishwanath which contains a large room with three time-worn posts in it surrounded by 64 golden disks.
Brahmin priests, acting out the command of an ancient prophecy, have been moving these disks, in accordance with the immutable rules of the Brahma, since that time.
According to the legend, when the last move of the puzzle will be completed, the world will end.

The puzzle is therefore also known as the Tower of Brahma puzzle.



•You can only move one disk at a time
•You can only move a disk to a pole where it will be the smallest
(i.e. you can't put a disk on top of a larger one)
•You can only remove the smallest disk from a pole
(i.e. you can't lift up the stack to get a larger disk from below)

-- Go To Powerpoint Slides --

See this resource online: https://www.mathsisfun.com/games/towerofhanoi.html

How many moves does it take?

NumDisks	**1**	**2**	**3**	**4**	**5**	**6**	**7**
NumMoves	1	3	7	15	31	64	127

Notice any patterns?

$NumMoves = 2^{NumDisks} - 1$

So…when will the world end? -> Roughly 584 BILLION years after they started...we have time.

Towers of Hanoi implementation in Python:

```
def moveTower(nDisks, startingPole, endingPole, helperPole):
    if nDisks >= 1:
        moveTower(nDisks-1, startingPole, helperPole, endingPole)
        moveDisk(startingPole, endingPole)
        moveTower(nDisks-1, helperPole, endingPole, startingPole)

def moveDisk(startingPole, endingPole):
    print("moving disk from", startingPole, "to", endingPole)


moveTower(4,"A","B","C")
```

How would you solve Tower of Hanoi iteratively???

(Recall) Basic Structure of a recursive algorithm

- A **base case**: what to do in the simplest possible case (i.e. when you have a single disk)
- A **recursive step**: break the original problem into one or more smaller problems, and solve that (saving the intermediate result)

(Recall) Recursion themes

- "Looping without a loop"
- "A function that calls itself as part of its definition"
- "Solving a problem by solving smaller instances"
- Key components of all three:
  - a recursive step (i.e. knowing when to split)
  - a "base case" (i.e. knowing when to stop)

More Examples

Here is another example this time with characters:

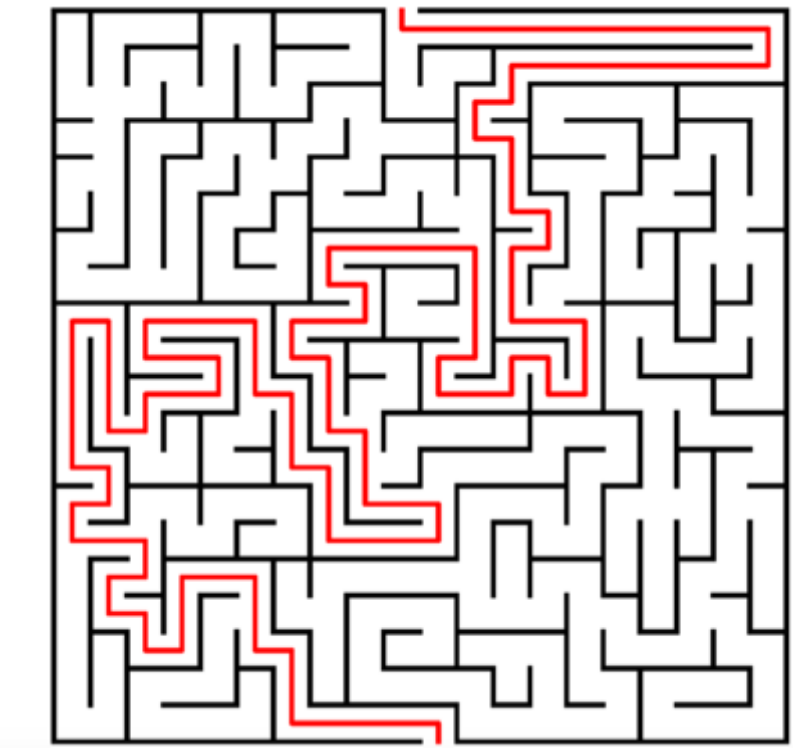
```
def backwards_alphabet(curr_letter):
    if curr_letter == 'a':
        print(curr_letter)
    else:
        print(curr_letter)
        prev_letter = chr(ord(curr_letter) - 1)
        backwards_alphabet(prev_letter)

starting_letter = 'f'
backwards_alphabet(starting_letter)
```

From our previous examples it might not be obvious why recursion is useful, since in many of those cases we could have accomplished the same result using iteration. Furthermore, recursion (sometimes) adds more complexity and is not easy to follow. There are however, situations where recursion can be very useful.

Exercise (Non-Programming)

How would you program a robot to solve a maze?



An Answer:

1. Mark your current location as visited
2. If you're at the end, you're done!
3. If not:
   A. If unmarked, go NORTH, solve maze. If not solved, go back and:
   B. If unmarked, go SOUTH, solve maze. If not solved, go back and:
   C. If unmarked, go EAST, solve maze. If not solved, go back and:
   D. If unmarked, go WEST, solve maze. If not solved, NO SOLUTION