CS1210 Lecture 21 Oct. 11, 2021

- Quiz 2 scores and comments available
  - Median and average around 14/20 or 70%
  - Not an high number of scores at the median, though.
    - A lot of scores at 50% or below - 70+ people
    - A lot of 20/20 - 28 people!
- Note the course grading scale published on the website. As DS4 and 5, and HW4 get graded, you'll start to get a good idea of where you stand. Please let me know if you want to discuss your current score/grade outlook.
- HW 5 is due Friday
- DS6 will be published this afternoon and will be due at 8pm tomorrow. DS attendance is not required but the TAs will be available in person and on zoom to help. The DS6 assignment consists of two short, easy recursion problems.
- Next quiz is Oct. 29, and will cover recursion (our current topic) and objects/classes/ inheritance

Last time

- Another image manipulation example
- Started recursion - Chapter 16

Today

- More on recursion - Chapter 16

Recursion – Ch 16

- Very important and useful concept
- Not just for programming, but math and even everyday life, legal definitions, etc.
- Has undeserved reputation among some people: “recursion is bad – recursive programs are inefficient” Yes, one can write very bad recursive programs but this is true of non-recursive programs as well. And recursion *can* be super useful.

Recursion

- Recursive function: function whose definition contains references to/calls to itself

- For example, math's factorial (3! = 3 * 2 * 1)
  - You might be familiar with informal definition like: n! = n * (n-1) * … * 2 * 1
  - But more precise mathematical definition is:

    factorial(1) = 1

    factorial(n) = n * factorial (n-1), for all n > 1

- Programming-wise, can very directly translate recursive mathematical definitions into code:

```
def factorial(n):
    if (n == 1):
        return 1
    else:
        return n * factorial (n + 1)
```

- DON'T let the function call, factorial(n-1), scare you. It's just a function call. If you draw stack frames like we did in earlier lectures, it all works out fine.
- DO need to think carefully when writing/analyzing recursive programs though …

Important rules for recursive functions

- When writing a recursive function:
  - MUST have *base case*(s), situations when code *does not* make recursive call.
  - MUST ensure that recursive calls *make progress toward base cases.* I.e. you need to convince yourself that recursive call is "closer to" base case than the original problem you are working on
  - SHOULD ensure you *don't unnecessarily repeat work*. Ignoring this contributes to recursion's bad reputation. E.g. direct recursive implementation of Fibonacci is extremely and unnecessarily inefficient

Recursion

More basic recursion examples (lec20.py)

- Print the items of a list, one per line
- Print the items of a list, one per line, in reverse order
  - Idea?
  - Consider list as: theFirstItem <the rest of the list>
  - Reverse is: reverse(<the rest of the list>) theFirstIte
  - Consider list as: <list from start to near end> theLastItem
  - Reverse is: theLastItem reverse(<list from start to near end>)
- return a string that is the reverse of the given string
- sum the items in a list
- return True/False depending on whether given string is a palindrome (e.g. Was it a car or a cat I saw?)
- return num of digits in an integer
- return sum of digits in an integer
- return string with each occurrence of a particular character replaced with a new character
- return count of number of substrings that have same first and last characters
- compute nth Fibonacci number:
  - 1, 1, 2, 3, 5, 8, 13, …
  - Definition: fib(1) = 1, fib(2) = 1,

    fib(n) = fib(n-1) + fib(n-2) for n > 2

Ch16: Recursion and stack frames

```
# Fibonacci numbers: 1,1, 2, 3, 5, 8, 13, …
# variables fnm1, fnm2, result are used just so that it's a little easier to follow
# what's going on when stepping through execution/viewing stack frames
#
def fib(n):
    if (n == 0) or (n == 1):
        result = 1
    else:
        fnm1 = fib(n-1)
        fnm2 = fib(n-2)
        result = fnm1 + fnm2
    return result

>>> fib(4)
3
```

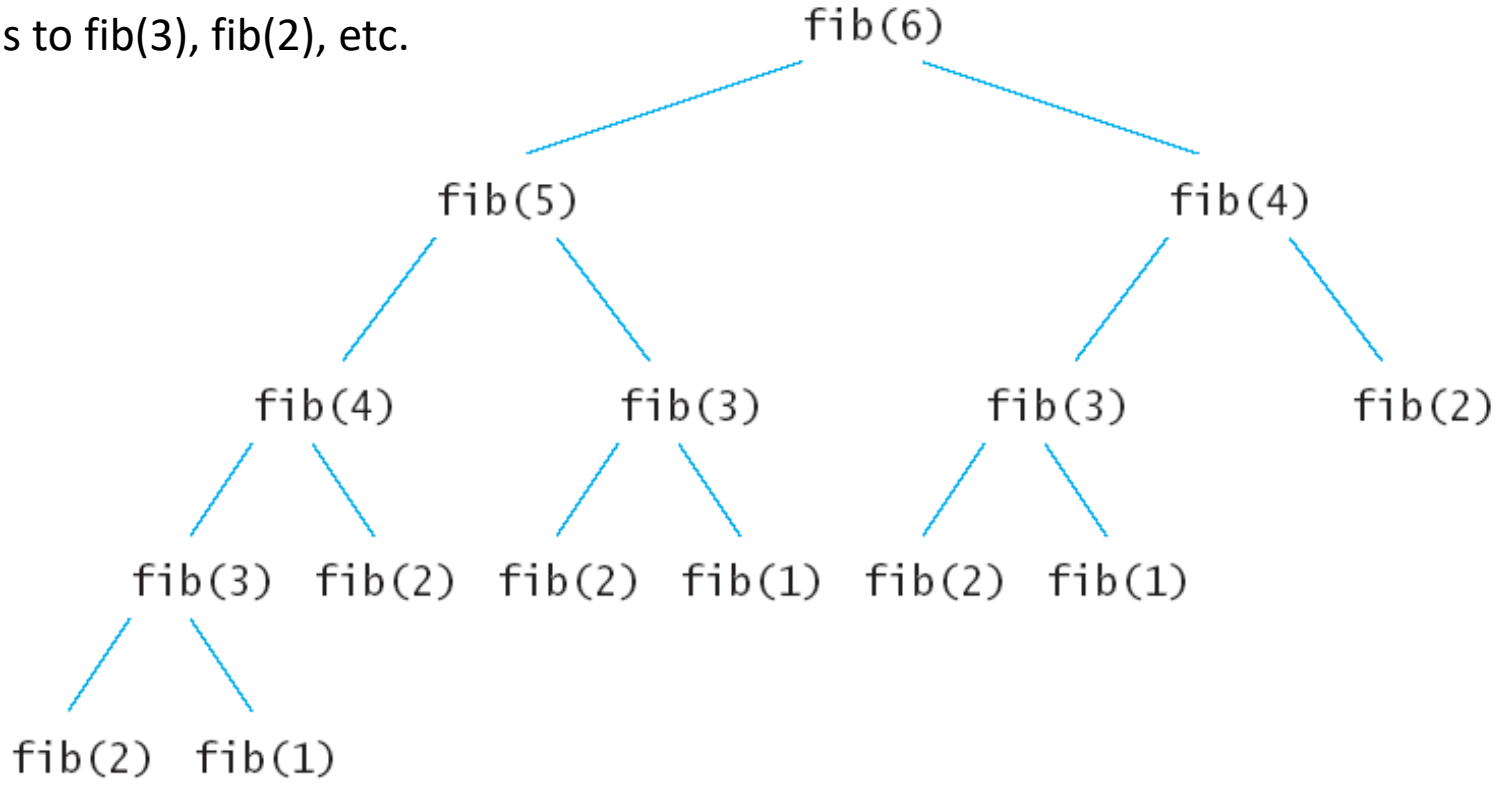
Copy/paste code above and execute step-by-step on Pythontutor.com to watch variables and stack diagrams

Simplest direct recursive implementation of Fibonacci violates third rule of designing recursive function:

- redoes many calculations many times
  - For fib(6), fib(2) called 5 times, fib(1) called 3 times.

This can be easily fixed (while maintaining basic recursive structure). Technique is called "memoization" and just means that once you calculate something store the value in a table. Whenever about to calculate something, check first if value is already in table

- Thus, after fib(5) calls fib(4), fib(4)'s value, 3, is stored in a table. When fib(6) needs the value fib(4), it will be retrieved from the table rather than generating calls to fib(3), fib(2), etc.



More recursion examples

- Given a list of numbers and a number n, determine if there is a subset of numbers from that list that sums to n. E.g. findSum(10, [3, -2, 1, 5, 99, 2]) should yield [3,5,2] findSum.py
- "flatten" a list lec21.py
  - E.g. [[[[[3,[2,4]]], 0], ['a']], 23] -> [3,2,4,0,'a', 23]
- Towers of Hanoi problem ToHcomplete.py

(These next three will be done next time)

- generate a string pattern lec21.py
- nesting depth lec21.py
- Drawing shapes

Towers of Hanoi solution

- Rods/towers A, B, C. Goal is to move disks from A to C, one at a time, never allowing larger disk to be on top of smaller disk
- Algorithm:
  - Move n-1 disks from A -> B (by this algorithm!)
  - Move final disk from A -> C
  - Move n-1 disks from B -> C (by this algorithm!)

ToHcomplete.py

Nesting depth

- Def of nestingDepth of a list
  - 0 if list has no lists as elements
  - 1 more than maximum nestingDepth among all items of the list that are lists
- Examples
  - [] -> 0
  - [[]] -> 1
  - [[[]]] -> 2
  - [[[]], 1, 2] -> 2, because [[]] is 1 so the whole list is 1 + 1 = 2
  - [[[]], 1, 2, [[[100]]]] -> 3, because [[]] is 1, [[[100]]] is 2, max is 2, so whole list is 2 + 1 = 3

```
def nestingDepth(inList):
    # initialize a variable for max nesting level seen (among sublists)
    # iterate over items in inList
    #     if item is not a list, just skip it
    #     if item is a list, recursively compute its nesting level and
    #       compare that item's nesting level with max seen so far,
    #       updating max if nec
    # if max has not been updated in the loop, return 0
    # otherwise return 1 more than max
```

Next Time

- Finish recursion: Towers of Hanoi plus some other problems, including use of recursion in turtle graphics
- Start next topic - a short look at randomization