# CS1210 Lecture 22 Oct. 13, 2021

- A new item has been added to the "Tests" group of the ICON grades.
  - Quiz1and2adjustment indicates how many additional points have been added to your total to reflect the "replace quiz 1 score with quiz 2 score" option I discussed after quiz 1. Thus, if you got 10 on quiz 1 and 16 on quiz 2, the adjustment would be 4 (because 16/20 on quiz2 converts to 14/18 for quiz 1)
- Again, note the course grading scale published on the website. As DS4, 5, 6, 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. I'll say more about this soon.
- HW 5 is due Friday
- Next quiz is Oct. 29, and will cover recursion (our current topic) and objects/ classes/inheritance

Last time

- Continued recursion - Chapter 16

Today

- A few words about CS courses after this one
- Finish recursion - Chapter 16
- A short introduction to exceptions - Chapter 13

# 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.

# 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

# 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 - e.g. [1,2,3,'a'] -> 0 []-> 0
  - 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 set in the loop (i.e. we saw no lists), return 0
    # otherwise return 1 more than max
```

# Handling errors, raising errors, monitoring presumptions

- try/except
- raise
- assert – good practice to ”sprinkle” asserts throughout your code. Catch cases of presumed conditions not being met before they result in mysterious, hard-to-track down errors

Try/except and raise covered well in basic Python documentation

Textbook Ch 13 covers exceptions. I won’t test you on this material but it is very useful.

exceptions.py

# Next Time

- Start next topic - classes, objects, and object-oriented programming - Chapters 17-19