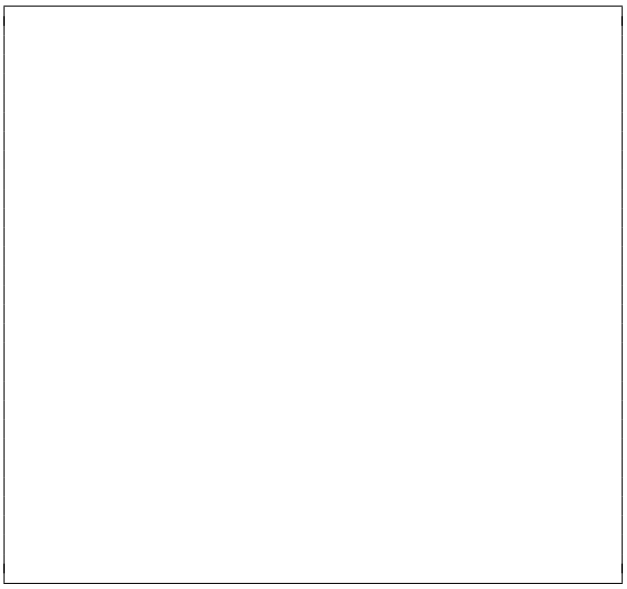
15-112 Final Exam Review

Up to 75 minutes. No calculators, no notes, no books, no computers. Show your work!
Do not use try/except on this review.

1. (points) CT1

```
class A(object):
    def __init__(self):
        self.b = "Puppy"

    def f(self):
        print("Dog")

    def g(self):
        print("Cat")

    def h(self):
        print(f"{self.b}: Pony")
        self.g()

class B(A):
    def __init__(self, s):
        self.b = s
        super().__init__()

    def g(self):
        print("Falcon")
        super().g()

    def h(self, s):
        super().h()
        print(f"{s}: Tiger")

def ct1():
    b = B("Sheep")
    b.h("Moo")

ct1()
```



2. (points) CT2

```
import copy

def ct2(L):
    M = copy.copy(L)
    N = copy.deepcopy(L)
    O = L[:]
    M[0].append(3)
    N[0].append(4)
    O += [3]
    L.insert(0, L.pop())
    print(f"M: {M}")
    print(f"N: {N}")
    print(f"O: {O}")

L = [[1], [2]]
ct2(L)
print(f"L: {L}")
```

3. (points) **Free Response**: OOPy Wars

Write the classes ForceUser and Jedi to pass the test cases shown below. Do not hardcode against the testcase values, though you can assume the testcases cover needed functionality. For full credit you must use inheritance appropriately, including not needlessly duplicating code.

Consider the following test cases:

```
# A ForceUser has a name. It also has a lightsaber,
# which can either be None (no lightsaber) or a
# string (the color of the lightsaber).
# The default lightsaber is None.
assert(str(ForceUser('Luke')) == 'Force User Luke')
assert(str(ForceUser('Yoda', 'green')) == 'Force User Yoda, green lightsaber')
# You can give a lightsaber to a ForceUser,
# but only if they don't have one already
rey = ForceUser('Rey')
assert(str(rey) == 'Force User Rey')
assert(rey.giveLightsaber('blue') == 'Rey got a lightsaber!')
assert(str(rey) == 'Force User Rey, blue lightsaber')
assert(rey.giveLightsaber('red') == 'Rey already has a lightsaber!')
assert(str(rey) == 'Force User Rey, blue lightsaber')
# Here are a variety of things that should
# work for a ForceUser
luke = ForceUser('Luke')
assert(luke == ForceUser('Luke', None))
assert(luke != ForceUser('Leia'))
assert(luke != 'do not crash here!')
s = set()
assert(luke not in s)
s.add(luke)
assert(ForceUser('Luke') in s)
assert(ForceUser('Luke', 'green') not in s)
# A Jedi is a ForceUser who can become a ghost
# if you strike them down.
# But not all Jedi are ghosts!
obiWan = Jedi('Obi-Wan', 'blue')
assert(isinstance(obiWan, ForceUser) == True)
assert(obiWan.giveLightsaber('purple') == 'Obi-Wan already has a lightsaber!')
assert(obiWan.isGhost() == False)
obiWan.strikeDown()
assert(obiWan.isGhost() == True)
# you can't strike down a non-Jedi, because
# ForceUser doesn't implement strikeDown.
crashed = False
try: rey.strikeDown()
except: crashed = True
assert(crashed == True)
```

If you need more space, the next page is blank.

Extra space for Problem 3.

4. (10 points) **Free Response**: Slidy Numers

   Do not use strings, lists, dictionaries, sets, try/except, or recursion on this problem. If you do, you will receive a 0.

   We'll say that an integer is a slidyNumber (coined term) if it is a positive integer such that it has at least 3 digits and the digits are in descending order. For example, 210, 320, and 8320 are all slidy numbers. 2110 is *not* a slidy number.

   Write the function `slidyNumbersCount(n)` that returns the count of slidyNumbers between 0 and n.

   Hint: You should almost certainly break this problem into two parts: `isSlidyNumber(a)` *and* `slidyNumbersCount(n)`

   Consider the following test cases:

```
assert slidyNumbersCount(-5) == None
assert slidyNumbersCount(100) == 0
assert slidyNumbersCount(220) == 1 #210
assert slidyNumbersCount(320) == 3 #210, 310, 320
assert slidyNumbersCount(500) == 10 #210, 310, 320, 321, 410, 420, 421, 430, 431, 432
```

5. (points) **Recursive Free Response: recursiveFind**

   Write a **recursive** function `recursiveFind(L, elem, start)` that behaves like the find built-in function of lists in Python: it returns a list of indices in L where element `elem` occurs, starting at index `start`.

   Your solution must use recursion. If you use any loops or iterative functions, you will receive no points on this problem.
   Your solution should NOT use any built-in functions that imply iteration, in particular: `index`, `in`, and (obviously) `find`.
   You can use slicing ONLY to remove the first element (as usually done in recursive functions)
   Hint: You can create a recursive helper function (with more useful arguments) that you call from the main function.

   Some examples:

```
assert(recursiveFind(["a","42","15112","test", "15112"], 0, '15112') == [2,4])
assert(recursiveFind([42,1,0,42], 0,  42) == [0, 3]) # 42s at index 0 and 3
assert(recursiveFind([42,1,0,42], 2,  42) == [3]) # Start from 2, only a 42 at index 3
assert(recursiveFind([42,1,42,0], 3,  42) == []) # The last 42 occurs before index 3
assert(recursiveFind(["one","two", "three"], 0, 'four') == []) # No elements found
assert(recursiveFind(['a', 'a', 'a', 'a', 'a'], 0, 'a') == [0,1,2,3,4])
assert(recursiveFind(['a', 'a', 'a', 'a', 'a'], 4, 'a') == [4])
```

6. (points) **Free Response: Invert Dictionary**

Write the non-destructive function `invertDictionary(d)` that takes a dictionary d that maps keys to values and returns a dictionary of its inverse, that maps the original values back to their keys. This has a complication: There can be duplicate values in the original dictionary. That is, there can be keys k1 and k2 such that `d[k1] == v` and `d[k2] == v` for the same value v. In that case, invertDictionary should map the original value back to the set of all the keys that originally mapped to it. Consider the following example:

```
d = {'cat':5, 'dog':6, 'bunny':5, 'bird':7}
print(invertDictionary(d))
# prints: {5: {'cat', 'bunny'}, 6: 'dog', 7: 'bird'}
```

Note: You may assume that all keys and values used will be immutable.

You should use sets and/or dictionaries to do this faster than $O(N^2)$

You should also specify the Big-O of your solution in the box below: