Discussion 5: July 8, 2025

Tree Recursion

For the following questions, don't start trying to write code right away. Instead, start by describing the recursive case in words. Some examples: - In fib from lecture, the recursive case is to add together the previous two Fibonacci numbers. - In double_eights from lab, the recursive case is to check for double eights in the rest of the number. - In count_partitions from lecture, the recursive case is to partition n-m using parts up to size m and to partition n using parts up to size m-1.

Q1: Maximum Subsequence

A *subsequence* of a number is a series of digits from the number (not necessarily contiguous). For example, 12345 has subsequences like 123, 234, 124, 245, etc. Your task is to find the **largest subsequence** that is under a specified length.

Hint: To add a digit (d) to an existing number (n), calculate: n * 10 + d. For instance, to add 8 to 15 to get 158, compute 15 * 10 + 8.

```
def max_subseq(n, t):
0.00
Return the maximum subsequence of length at most t that can be found in the given
number n.
For example, for n = 2012 and t = 2, we have that the subsequences are
     0
     1
     2
     20
     21
     22
     01
     02
     12
and of these, the maxumum number is 22, so our answer is 22.
>>> max_subseq(2012, 2)
22
>>> max_subseq(20125, 3)
225
>>> max_subseq(20125, 5)
20125
>>> max_subseq(20125, 6) # note that 20125 == 020125
20125
>>> max_subseq(12345, 3)
345
>>> max_subseq(12345, 0) # 0 is of length 0
>>> max_subseq(12345, 1)
"*** YOUR CODE HERE ***"
```

Q2: Making Onions

Write a function make_onion that takes in two one-argument functions, f and g. It returns a function that takes in three arguments: x, y, and limit. The returned function returns True if it is possible to reach y from x using up to limit calls to f and g, and False otherwise.

For example, if f adds 1 and g doubles, then it is possible to reach 25 from 5 in four calls: f(g(g(f(5)))).

```
def make_onion(f, g):
"""Return a function can_reach(x, y, limit) that returns
whether some call expression containing only f, g, and x with
up to limit calls will give the result y.
>>> up = lambda x: x + 1
>>> double = lambda y: y * 2
>>> can_reach = make_onion(up, double)
>>> can_reach(5, 25, 4)  # 25 = up(double(double(up(5))))
True
>>> can_reach(5, 25, 3)
                           # Not possible
False
>>> can_reach(1, 1, 0)
                           # 1 = 1
True
>>> add_ing = lambda x: x + "ing"
>>> add_end = lambda y: y + "end"
>>> can_reach_string = make_onion(add_ing, add_end)
>>> can_reach_string("cry", "crying", 1) # "crying" = add_ing("cry")
True
>>> can_reach_string("un", "unending", 3)  # "unending" = add_ing(add_end("un"))
>>> can_reach_string("peach", "folding", 4)
                                             # Not possible
False
def can_reach(x, y, limit):
     if limit < 0:</pre>
        return ____
     elif x == y:
        return ____
     else:
        return can_reach(___, ___, limit - 1) or can_reach(___, ___, limit - 1)
return can_reach
```

Q3: Pascal's Triangle

Pascal's triangle is a recursively defined mathematical structure. Here are the first five rows of Pascal's triangle:

		col						
		0	1	2	3	4	5	6
row	0	1	0	0	0	0	0	0
	1	1	1	0	0	0	0	0
	2	1	2	1	0	0	0	0
	3	1	3	3	1	0	0	0
	4	1	4	6	4	1	0	0

Pascal's triangle, as a grid.

Every number in Pascal's triangle is defined as the sum of the number above it and the number above and to the left of it. Rows and columns are zero-indexed; that is, the first row is row 0 instead of row 1 and the first column is column 0 instead of column 1. For example, the number at row 2, column 1 in Pascal's triangle is 2.

Define the function pascal, which takes a row and column and finds the value of the number at that position in Pascal's triangle. Note that row and column will always be nonnegative.

Hint: For which positions can we find the corresponding number in Pascal's triangle without recursion? Remember that positions are zero-indexed!

```
def pascal(row, column):
"""Returns the value of the item in Pascal's Triangle
whose position is specified by row and column.
>>> pascal(0, 0)  # The top left (the point of the triangle)
>>> pascal(0, 5)  # Empty entry; outside of Pascal's Triangle
>>> pascal(3, 2)
                 # Row 3 (1 3 3 1), Column 2
>>> pascal(4, 2)  # Row 4 (1 4 6 4 1), Column 2
"*** YOUR CODE HERE ***"
```