DATA C88C
Spring 2025

Tree Recursion
Discussion 5: February 25, 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):
    """
    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
        2
        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
    0
    >>> max_subseq(12345, 1)
    5
    """
    if n == 0 or t == 0:
        return 0
    with_last = max_subseq(n // 10, t - 1) * 10 + n % 10
    without_last = max_subseq(n // 10, t)
    return max(with_last, without_last)
```

```
def max_subseq(n, t):
    """
    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
        2
        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
    0
    >>> max_subseq(12345, 1)
    5
    """
    if n == 0 or t == 0:
        return 0
    with_last = max_subseq(n // 10, t - 1) * 10 + n % 10
    without_last = max_subseq(n // 10, t)
    return max(with_last, without_last)
```

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.

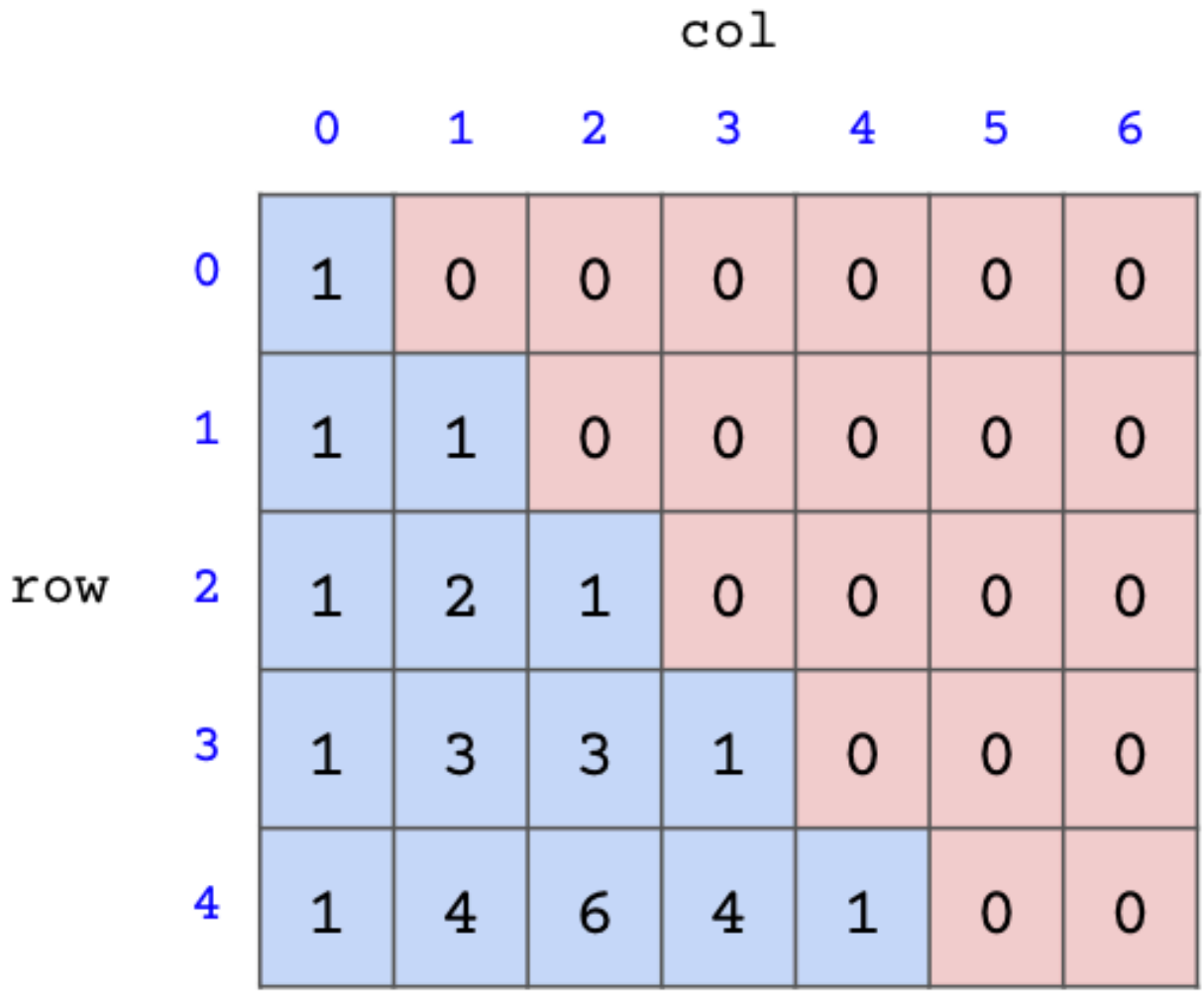
For example, if `f` adds 1 and `g` doubles, then it is possible to reach 25 from 5 in four calls: `f(g(g(f(5))))`.

```
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"))
    True
    >>> can_reach_string("peach", "folding", 4)   # Not possible
    False
    """
    def can_reach(x, y, limit):
        if limit < 0:
            return False
        elif x == y:
            return True
        else:
            return can_reach(f(x), y, limit - 1) or can_reach(g(x), y, limit - 1)
    return can_reach
```

```
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"))
    True
    >>> can_reach_string("peach", "folding", 4)   # Not possible
    False
    """
    def can_reach(x, y, limit):
        if limit < 0:
            return False
        elif x == y:
            return True
        else:
            return can_reach(f(x), y, limit - 1) or can_reach(g(x), y, 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:



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)
    1
    >>> pascal(0, 5)    # Empty entry; outside of Pascal's Triangle
    0
    >>> pascal(3, 2)    # Row 3 (1 3 3 1), Column 2
    3
    >>> pascal(4, 2)     # Row 4 (1 4 6 4 1), Column 2
    6
    """
    if column == 0:
        return 1
    elif row == 0:
        return 0
    else:
        above = pascal(row - 1, column)
        above_left = pascal(row - 1, column - 1)
        return above + above_left
    # First base case: every number in the leftmost column of the triangle is 1.
    # Second base case: There is only one number in the topmost row, which is already
    # accounted for by the first base case.
```

For any position that satisifies `row >= 1` and `column >= 1`, the recursive definition of Pascal's Triangle works. But for any position where `row == 0` or `column == 0`, there is no position "above and to the left of it", so the recursive definition won't work! That is why we need the base cases described above.

Your solution may have different base cases, such as a base case for all positions that satisfy `row < column`. This is perfectly okay! As long as all positions in the first row or first column are treated as base cases, there is nothing wrong with treating additional positions as base cases.

Document the Occasion

Please all fill out the attendance form (one submission per person per week).