

Attendance

Your TA will come around during discussion to check you in. You can start on the worksheet before being checked in; you don't need to wait for your TA to get started.

Getting Started

Say your name and share a food that you really liked as a child. (It's ok if you still like that food now.)

Recursion

Many students find this discussion challenging. Everything gets easier with practice. Please help each other learn. It's also long. If you don't finish, don't worry.

VERY IMPORTANT: In this discussion, don't check your answers until your whole group is sure that the answer is right. Figure things out and check your work by *thinking* about what your code will do. If you need help, ask.

Q1: Skip Factorial

Define the base case for the `skip_factorial` function, which returns the product of **every other positive** integer, starting with `n`.

```
def skip_factorial(n):
    """Return the product of positive integers n * (n - 2) * (n - 4) * ...

    >>> skip_factorial(5) # 5 * 3 * 1
    15
    >>> skip_factorial(8) # 8 * 6 * 4 * 2
    384
    """
    if n <= 2:
        return n
    else:
        return n * skip_factorial(n - 2)
```

If `n` is even, then the base case will be 2. If `n` is odd, then the base case will be 1. Try to write a condition that handles both possibilities.

Q2: Swipe

Implement `swipe`, which prints the digits of argument `n`, one per line, **first backward then forward**. The left-most digit is printed only once. **Do not use** `while` or `for` or `str`. (Use recursion, of course!)

```
def swipe(n):
    """Print the digits of n, one per line, first backward then forward.

    >>> swipe(2837)
    7
    3
    8
    2
    8
    3
    7
    """
    if n < 10:
        print(n)
    else:
        print(n % 10)
        swipe(n // 10)
        print(n % 10)
```

First `print` the first line of the output, then make a recursive call, then `print` the last line of the output.

Q3: Is Prime

Implement `is_prime` that takes an integer `n` greater than 1. It returns `True` if `n` is a prime number and `False` otherwise. Try following the approach below, but implement it recursively without using a `while` (or `for`) statement. You will need to define another “helper” function (a function that exists just to help implement this one). Does it matter whether you define it within `is_prime` or as a separate function in the global frame? Try to define it to take as few arguments as possible.

```
def is_prime(n):
    """Returns True if n is a prime number and False otherwise.
    >>> is_prime(2)
    True
    >>> is_prime(16)
    False
    >>> is_prime(521)
    True
    """
    def check_all(i):
        "Check whether no number from i up to n evenly divides n."
        if i == n:      # could be replaced with i > (n ** 0.5)
            return True
        elif n % i == 0:
            return False
        return check_all(i + 1)
    return check_all(2)
```

Define an inner function that checks whether some integer between `i` and `n` evenly divides `n`. Then you can call it starting with `i=2`:

```
def is_prime(n):
    def f(i):
        if i == n:
            return ____
        elif ____:
            return ____
        else:
            return f(____)
    return f(2)
```

Documentation: Come up with a one sentence docstring for the helper function that describes what it does. Don't just write, “it helps implement `is_prime`.” Instead, describe its behavior. If you need help, ask!

Q4: Recursive Hailstone

Recall the `hailstone` function from [Homework 2](#). First, pick a positive integer `n` as the start. If `n` is even, divide it by 2. If `n` is odd, multiply it by 3 and add 1. Repeat this process until `n` is 1. Complete this recursive version of `hailstone` that prints out the values of the sequence and returns the number of steps.

```
def hailstone(n):
    """Print out the hailstone sequence starting at n,
    and return the number of elements in the sequence.
    >>> a = hailstone(10)
    10
    5
    16
    8
    4
    2
    1
    >>> a
    7
    >>> b = hailstone(1)
    1
    >>> b
    1
    """
    print(n)
    if n % 2 == 0:
        return even(n)
    else:
        return odd(n)

def even(n):
    return 1 + hailstone(n // 2)

def odd(n):
    if n == 1:
        return 1
    else:
        return 1 + hailstone(3 * n + 1)
```

An even number is never a base case, so `even` always makes a recursive call to `hailstone` and returns one more than the length of the rest of the hailstone sequence.

An odd number might be 1 (the base case) or greater than one (the recursive case). Only the recursive case should call `hailstone`.

Recommended: Once your group has converged on a solution, it's time to practice your ability to describe your own code. Nominate someone to describe how your solution works for the group. If you'd like, you can even share your description with your TA.

Q5: Sevens

The Game of Sevens: Players in a circle count up from 1 in the clockwise direction. (The starting player says 1, the player to their left says 2, etc.) If a number is divisible by 7 or contains a 7 (or both), switch directions. Numbers must be said on the beat at **60 beats per minute**. If someone says a number when it's not their turn or someone misses the beat on their turn, the game ends.

For example, 5 people would count to 20 like this:

```

Player 1 says 1
Player 2 says 2
Player 3 says 3
Player 4 says 4
Player 5 says 5
Player 1 says 6 # All the way around the circle
Player 2 says 7 # Switch to counterclockwise
Player 1 says 8
Player 5 says 9 # Back around the circle counterclockwise
Player 4 says 10
Player 3 says 11
Player 2 says 12
Player 1 says 13
Player 5 says 14 # Switch back to clockwise
Player 1 says 15
Player 2 says 16
Player 3 says 17 # Switch back to counterclockwise
Player 2 says 18
Player 1 says 19
Player 5 says 20

```

Then, implement `sevens` which takes a positive integer `n` and a number of players `k`. It returns which of the `k` players says `n`. You may call `has_seven`.

An effective approach to this problem is to simulate the game, stopping on turn `n`. The implementation must keep track of the final number `n`, the current number `i`, the player `who` will say `i`, and the current `direction` that determines the next player (either increasing or decreasing). It works well to use integers to represent all of these, with `direction` switching between 1 (increase) and -1 (decreasing).

```

def sevens(n, k):
    """Return the (clockwise) position of who says n among k players.

    >>> sevens(2, 5)
    2
    >>> sevens(6, 5)
    1
    >>> sevens(7, 5)
    2
    >>> sevens(8, 5)
    1
    >>> sevens(9, 5)
    5
    >>> sevens(18, 5)
    2
    """
    def f(i, who, direction):
        if i == n:
            return who
        if i % 7 == 0 or has_seven(i):
            direction = -direction
        who = who + direction
        if who > k:
            who = 1
        if who < 1:
            who = k
        return f(i + 1, who, direction)
    return f(1, 1, 1)

def has_seven(n):
    if n == 0:
        return False
    elif n % 10 == 7:
        return True
    else:
        return has_seven(n // 10)

```

First check if i is a multiple of 7 or contains a 7, and if so, switch directions. Then, add the direction to `who` and ensure that `who` has not become smaller than 1 or greater than k .

Optional Question

Q6: Y combinator

The recursive factorial function can be written as a single expression by using a [conditional expression](#).

```
>>> fact = lambda n: 1 if n == 1 else mul(n, fact(sub(n, 1)))
>>> fact(5)
120
```

However, this implementation relies on the fact (no pun intended) that `fact` has a name, to which we refer in the body of `fact`. To write a recursive function, we have always given it a name using a `def` or assignment statement so that we can refer to the function within its own body. In this question, your job is to define `fact` recursively without giving it a name!

There's actually a general way to do this that uses a function `Y` defined

```
def Y(f):
    return f(lambda: Y(f))
```

Using this function, you can define `fact` with an assignment statement like this:

```
fact = Y(?)
```

where `?` is an expression containing *only* lambda expressions, conditional expressions, function calls, and the functions `mul` and `sub`. That is, `?` contains no statements (no assignments or `def` statements in particular), and no mention of `fact` or any other identifier defined outside `?` other than `mul` or `sub` from the `operator` module. You are also allowed to use the equality (`==`) operator. Find such an expression to use in place of `?`.

```
from operator import sub, mul

def Y(f):
    """The Y ("paradoxical") combinator."""
    return f(lambda: Y(f))

def Y_tester():
    """
    >>> tmp = Y_tester()
    >>> tmp(1)
    1
    >>> tmp(5)
    120
    >>> tmp(2)
    2
    """
    return Y(lambda f: lambda n: 1 if n==0 else n * f()(n-1))
```