If there are fewer than 3 people in your group, merge your group with another group in the room. If your group has 6 or more students, you're welcome to split into two sub-groups and then sync up at the end. If you want two separate Pensieve documents for the two sub-groups, just have one sub-group add 1000 to their group number.

Now switch to Pensieve:

- **Everyone**: Go to pensieve.co, log in with your @berkeley.edu email, and **enter your group number** as the room number (which was in the email that assigned you to this discussion). As long as you all enter the same number (any number), you'll all be using a shared document.

Once you're on Pensieve, you don't need to return to this page; Pensieve has all the same content (but more features). If for some reason Penseive doesn't work, return to this page and continue with the discussion.

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.

If you didn't attend for a good reason (such as being sick), fill out this form (within 2 weeks of your discussion): attendance form

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.)

After Midterm 1, some students are looking for more effective ways to study. One great option is to meet up with your discussion group outside of class to review practice problems together. Now is a great time to schedule a time and place for some extra group practice of old Midterm 1 questions. This is optional and not everyone needs to come, but if there are Midterm 1 topics that haven't totally clicked yet, this weekend is a perfect time to review them.

Everything in this course builds on prior topics, and it's going to be hard to keep up if you don't have a solid understanding of Midterm 1 material.

Remember, it's ok if someone hasn't learned everything yet and needs more time to master the course material. The whole point of the course is for students to learn things they don't already know. Please support each other in the process.

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. Your goal should be to have all checks pass the first time you run them! If you need help, ask.

Q1: 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!)

```python
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:
        "*** YOUR CODE HERE ***"
```

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

Q2: Skip Factorial

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

```python
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 ___:
        return ___
    else:
        return ___
```

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.

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.

```python
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
    """
    "*** YOUR CODE HERE ***"
```

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`:

```python
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.

```python
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 ____

def odd(n):
    "*** YOUR CODE HERE ***"
```

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
```

Play a few games. Post the highest score your group reached on Discord.

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).

```python
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
        "*** YOUR CODE HERE ***"

    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

This question is quite difficult but the solution is short and elegant.

Q6: Karel the Robot

Karel the robot starts in the corner of an `n` by `n` square for some unknown number `n`. Karel responds to only four functions: - `move()` moves Karel one square forward if there is no wall in front of Karel and errors if there is. - `turn_left()` turns Karel 90 degrees to the left. - `front_is_blocked()` returns whether there is a wall in front of Karel. - `front_is_clear()` returns whether there is no wall in front of Karel.

Implement a `main()` function that will leave Karel stopped halfway in the middle of the bottom row. For example, if the square is 7 x 7 and Karel starts in position (1, 1), the bottom left, then Karel should end in position (1, 4) (three steps from either side on the bottom row). Karel can be facing in any direction at the end. If the bottom row length is even, Karel can stop in either position (1, `n // 2`) or (1, `n // 2 + 1`).

Important You can only write `if` or `if`/`else` statements and function calls in the body of `main()`. You may not write assignment statements, def statements, lambda expressions, or while/for statements.

For every two steps forward, take one step back to end up in the middle.