CS 145 Homework 5 - due Sunday October 26, 2025
See go/cs145/ for starter code

1. **Sum the positive numbers in a list**

   Write a recursive function sum_pos(numbers) that returns the sum of the positive values in a list.
   For example:

```
>>> sum_pos([2, -3, 6, -1])
8
>>> sum_pos([])
0
```

2. **Compute the greatest common divisor of two positive integers**

   One of the oldest algorithms, dating back to the 5th century BCE, is Euclid's algorithm for computing the greatest common divisor of two integers. It works as follows:

   - Let a, b be positive integers.
   - Let rem be the remainder of a / b.
   - If rem is 0, then gcd(a,b) is b.
   - Otherwise, gcd(a,b) is the same as gcd(b,rem).

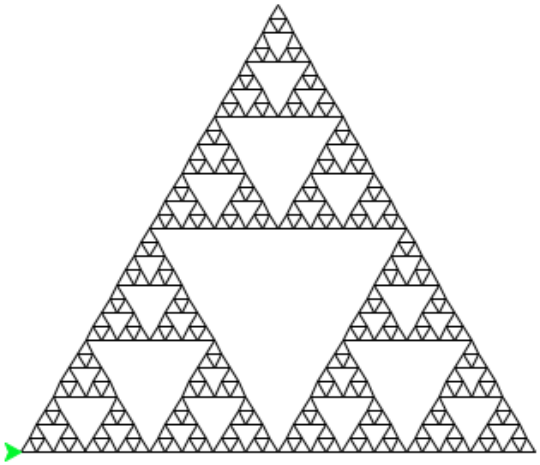
   Write a recursive function gcd(a,b) that implements Euclid's algorithm using a recursive strategy. Your function should take the two integers as parameters and should return a result. You can call your function from the shell to test it. Your gcd() function should work for any two positive integers sent as parameters.

   For example:

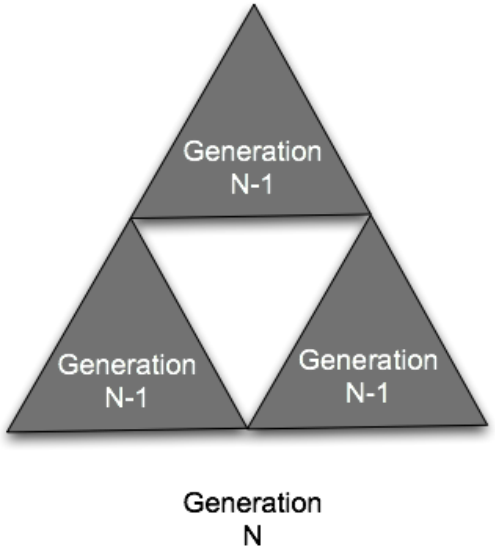
```
>>> gcd(48, 18)
6
>>> gcd(1, 2)
1
```

3. **Draw the Sierpinski triangle**

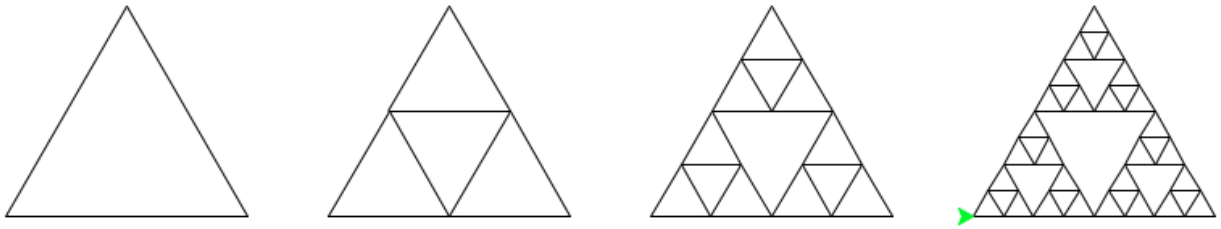
   Next we'll draw a fractal shape called the Sierpinski triangle. Here is what it looks like taken out to six generations:



There are many ways to draw this shape, but we are going to use the same technique that we used for the Koch curve. Rather than taking a line and breaking it into thirds, we are going to draw half size triangles. Here is an abstraction of how Generation N of the Sierpinski Triangle is built:



More simply, here are the first generations 1, 2, 3, 4:



Here is one of the easiest ways to draw the Sierpinski Triangle. Start with the assumption that when you draw the triangle, the turtle will be back in the same position and orientation it started from. Given that, drawing a generation N triangle just involves drawing three N-1 triangles that are half the size. Draw the shape, move to the location of the next one, draw again, move to the last triangle and draw that one, then restore the turtle to the original location and orientation. For generation 0, don't do anything (think of the base case of the spiral rather than the Koch curve).

All of this should go in a function called sierpinski(length, generations). length will be the length of the base of the figure and generations will tell us how many generations to make.

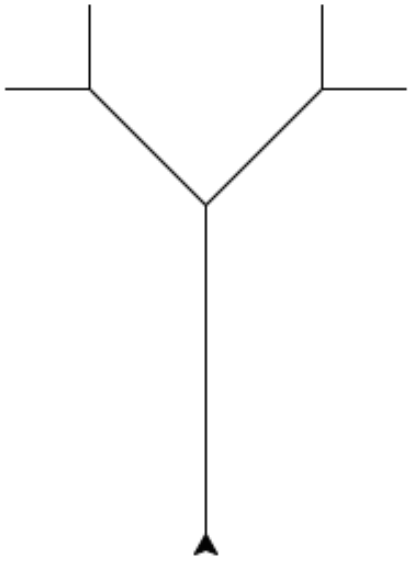
4. Draw a recursive tree with added randomization

In class we experimented with code to draw a self-similar tree. It has the important feature that the turtle is left in exactly the same position and orientation after drawing a tree. Using this, the basic tree shape can be accomplished by following these steps:

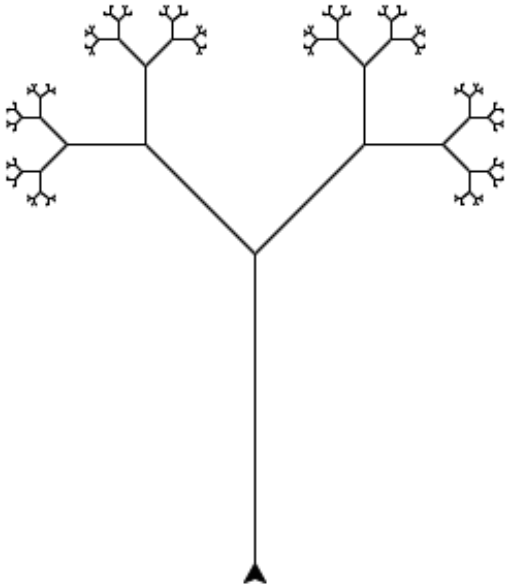
- Move forward by some amount (this draws the trunk)
- Turn left by 45 degrees
- Draw a smaller version of the tree (e.g., perhaps half size)
- Turn right by 90 degrees (you should actually think of this as undo-ing the previous 45 degrees and then turning right 45 degrees)

- Draw a smaller version of the tree
- Turn left by 45 degrees (i.e., undo the turn to the right)
- Move backward by the length of the trunk (the turtle should now be in its original position and orientation)

Here is what that tree looks like taken out to three generations:



And here it is taken out to eight generations:



Complete the definition of the function draw_tree(length, levels) that follows the above algorithm, as we did in class.

Once you have the tree drawing correctly, and you have tested a couple of generations, we are going to start making it more tree-like. Use random.randint to randomize the tree. There are two main things we can randomize: the length of a particular branch, and the angle with respect to the trunk.

Some ideas:

To randomize the trunk length, you could add a random value, say between -length//10 and length//10 to length at the start of your function. In essence, this would add or subtract something within a 10th of the trunk length.

You could play the same trick to randomize the angle. This is harder because you need to be able to undo the angle and it would be good if the left and right angles were not identical. One way to solve this would be to add a variable and break up the turn between the two branches into two separate turns.

For example, you could:

- Create a new angle variable and set it equal to your base angle (currently 45) plus some random amount between -45 and +45
- Rather than the 90 degree turn to the right, turn right by this angle to undo the left turn
- Immediately after this right turn, calculate a new random angle
- Use this angle to turn right
- After drawing the second smaller tree, use the new angle to turn left (the turtle should now be facing the original direction)

Now you should take some time to play with the tree code. Here are some things you could try (complete at least 2 for full points)

- Tinker with the generations to get a nice full tree.
- Changing the base angle from 45 will give you trees with quite different characters. Try some different angles (or even mix them within the same tree) to get a tree you like.
- Adjust the width of the turtle's lines (using turtle.width()) based on the current length so the tree tapers (i.e., make the width a function of the length).
- Rather than shrinking the length in half every time, experiment with other size reductions like one third, or three quarters.
- Add color, perhaps even based on the current generation.

Make sure the function includes a docstring enumerating the changes you have made.

When you are done:

1. Make sure your name is at the top of your code and all your functions have complete docstrings.
2. Run drawing_demo() to generate output for both your sierpinski and tree code. Take and save a screenshot of this window.
3. Submit both hw5_recursion.py and your screenshot to Gradescope.