HarveyMuddX: CS005x CS For All: Introduction to Computer Science and Python Programming

courses.edx.org/courses/course-v1:HarveyMuddX+CS005x+2T2016/courseware/a1930c32b9a9464fbd5071d8f53f96e9/d8a40c0a68fe415c8154237b16d95dba/

Problem 1: Turtle

Problem 1: Python Turtles

In this problem, you'll start working with Python's turtle, a drawing library.

When you're finished with this assignment, submit your code at the bottom of this page.

Trying turtle out

Try these commands in the trinket below. You don't need the comments, but they shouldn't hurt if you paste them:

```
from turtle import *     # loads the turtle library...
width(5)         # make the turtle pen 5 pixels wide
shape('turtle') # use a turtle shape!
forward(100)     # turtle goes forward 100 steps
right(90)        # turtle turns right 90 degrees
up()             # turtle lifts its pen up off of the
paper
forward(100)     # turtle goes forward 100 steps
down()           # turtle puts its pen down on the paper
color("red")     # turtle uses red pen
circle(100)      # turtle draws circle of radius 100
color("blue")    # turtle changes to blue pen
forward(50)      # turtle moves forward 50 steps
```

If you'd like to check out all of the available turtle commands, the full turtle reference is available at this official turtle library page.

The `poly` Function

Start by making a copy of this trinket. You'll write all of the Homework 3, Problem 2 functions in this trinket.

To run this *septagonal* example, enter `poly( 7, 7 )` under the function and then run the trinket.

The `spiral` Function

The `poly` function is an example of `single-path` recursion: the recursive calls are made a single time, so that there is a single, step-by-step path taken—both by the code and by the turtle!

Next, you'll try another single-path recursive function on your own.

To start, underneath the `poly` function, write another named `spiral`. Here is its signature:

```
def spiral( initialLength, angle, multiplier
):
```

This `spiral` function should use the `turtle` drawing functions to create a spiral that:

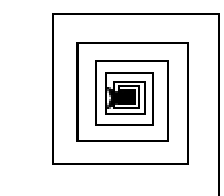
- has its first segment of length `initialLength` and
- whose neighboring segments form angles of `angle` degrees.
- The `multiplier` will be a float that will indicate how each segment changes in size from the previous one. For example, with a `multiplier` of `0.5` each side in the spiral should be *half* the length of the previous side.

Base cases!

The spiral should stop drawing when it has reached a side length of:

- less than 1 pixel or
- greater than 500 pixels

Here's a picture from the call `spiral( 100, 90, 0.9 )`



Try altering your spiral function with different drawing attributes—for example:

- different values of multiplier and angle
- include a different value of the line width in the code (or a changing value that depends on `initialLength`)
- a different, pre-programmed color within the function, or
- a random color for each segment of the spiral—by calling

```
c = choice( ['green','red','blue']
)
color( c )
```

The `chai` Function: Branching Recursion

Next, you'll build a *branching-recursion* example. It's in branching that recursion is at its most "magical"—but it's also true that in composing these functions, it can be the most mind-bending. What's remarkable is that the "magic" is, in the end, completely understandable.

Start by pasting the `chai` function from class:

```
def chai(size):
    """ our chai function!
"""
    if (size<9):
        return
    else:
        forward(size)
        left(90)
        forward(size/2.0)
        right(90)
        right(90)
        forward(size)
        left(90)
        left(90)
        forward(size/2.0)
        right(90)
        backward(size)
        return
```

Then, try running it with `chai( 100 )`.

Next, add one branch of recursion between the two calls to `right(90)`, by pasting a recursive call to `chai( size/2 )`.

Try it out!

Finally, add a second branch. It's really nothing more than a second branch, but because it's realized recursively, the resulting work (and visual intricacy) can be much greater!

In addition to the above call, between the two calls to `left(90)`, paste a recursive call to `chai( size/2 )`.

Try it out! Try it with some different parameters, as well. For example, you could:

- leave the first branching call at `chai( size/2 )` and change the second to `chai( size/3 )`
- or vice-versa
- try adding a color change (or two) within the code
- try adding a line-width change (or two) within the code

All of these example runs will build up intuition about how branching recursion works. In the end, it's simply creating a smaller version of the overall structure at ***more than one location*** within that structure.

The key to making "branching-recursion" work is *making sure that your turtle **ends** at the same location that it **begins***. That is how you know that the statements after the recursive calls are moving the turtle as expected.

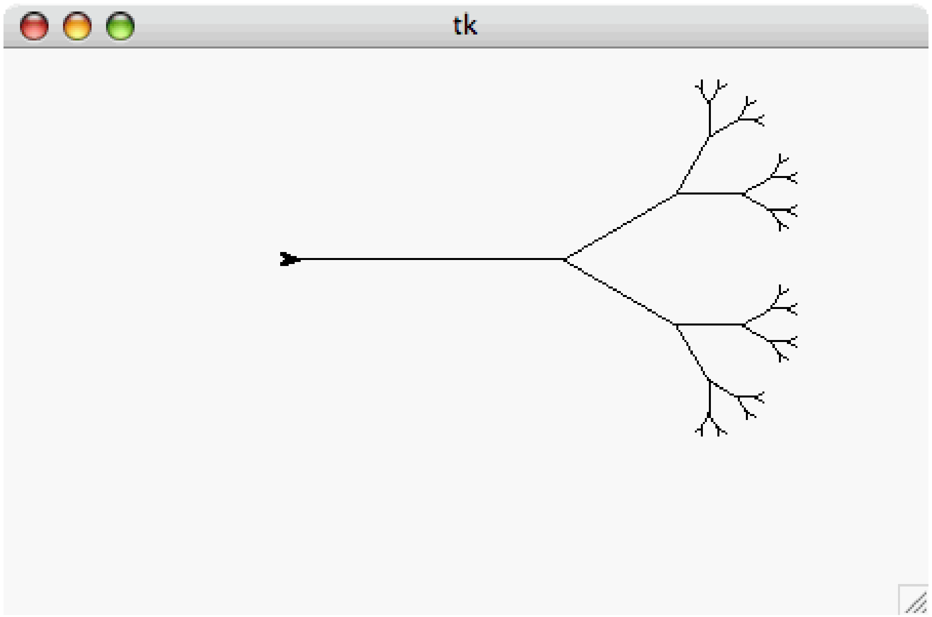
The `svtree` Function

Next, you'll write another branching example—here, "branching" seems like a particularly appropriate descriptor!

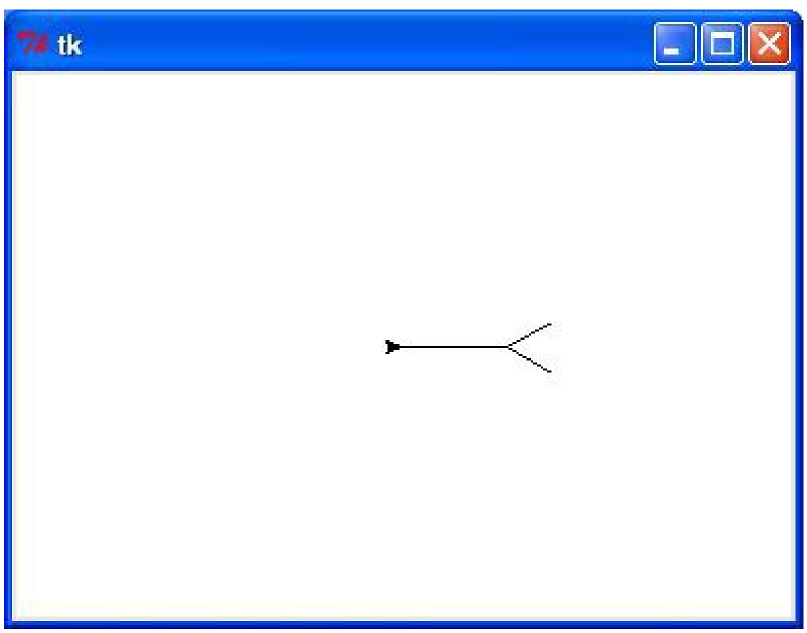
The idea here is to create a function that draws the *side-view* of a tree, hence `svtree`:

```
def svtree( trunklength, levels
)
```

Here is an example of the output from my function when `svtree( 128, 6 )` is run:

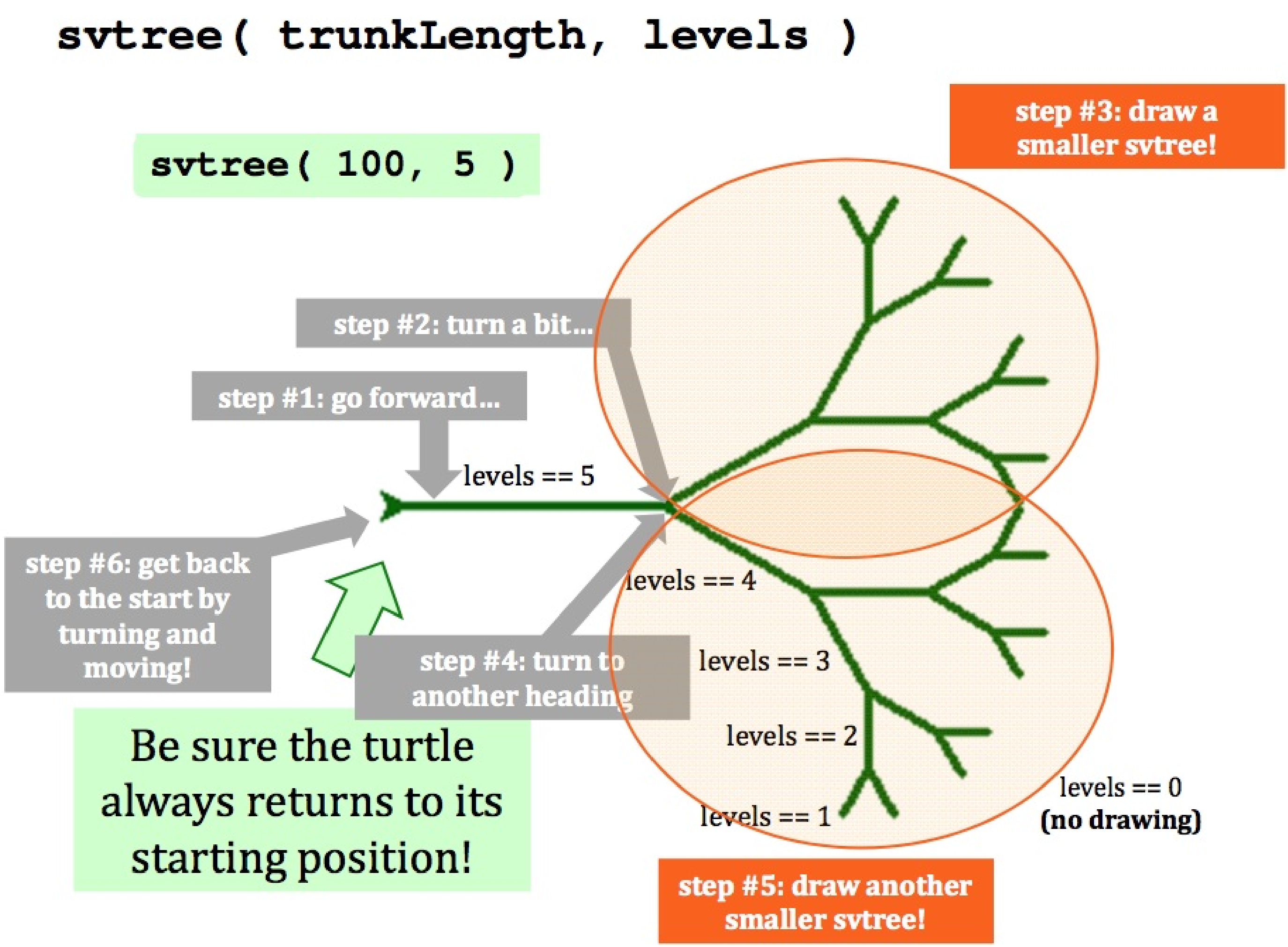


and another example of the output when `svtree( 50, 2 )` is run:



Note that these are **really** side view!

Calling `left(90); svtree(100,5)` will yield a more traditional tree pose! Also, here is a picture showing the self-similar breakdown of `svtree` (from the slides). *This is, in fact, an almost complete map of the* `svtree` *code!*



Hints

Consider the `chai` function above—and its recursive extension. It's is a good starting point for `svtree`.

The key to happiness with recursive drawing is this: ***the pen must be back at the start (root) of the tree at the end of the function call***! That way, each portion of the recursion "takes care of itself" relative to the other parts of the image.

Don't worry about the exact angle of branching or the amount of reduction of the `trunklength` in sub-branches, etc. However, you can design your own tree by making aesthetic choices for each of these, if you like!

Once you have the `svtree` function working, alter it so that it has **three or more branches**, instead of only two. You can get some very dense "foliage" very quickly, and even more "life-like" results are possible if you use non-identical branching angles and size multipliers!

Also, you could have the `width` or `color` depend on the value of `levels`. If you make the final "level" red, you can create an apple tree or, if you make that final "level" a random color, you can produce fall-foliage-type effects!

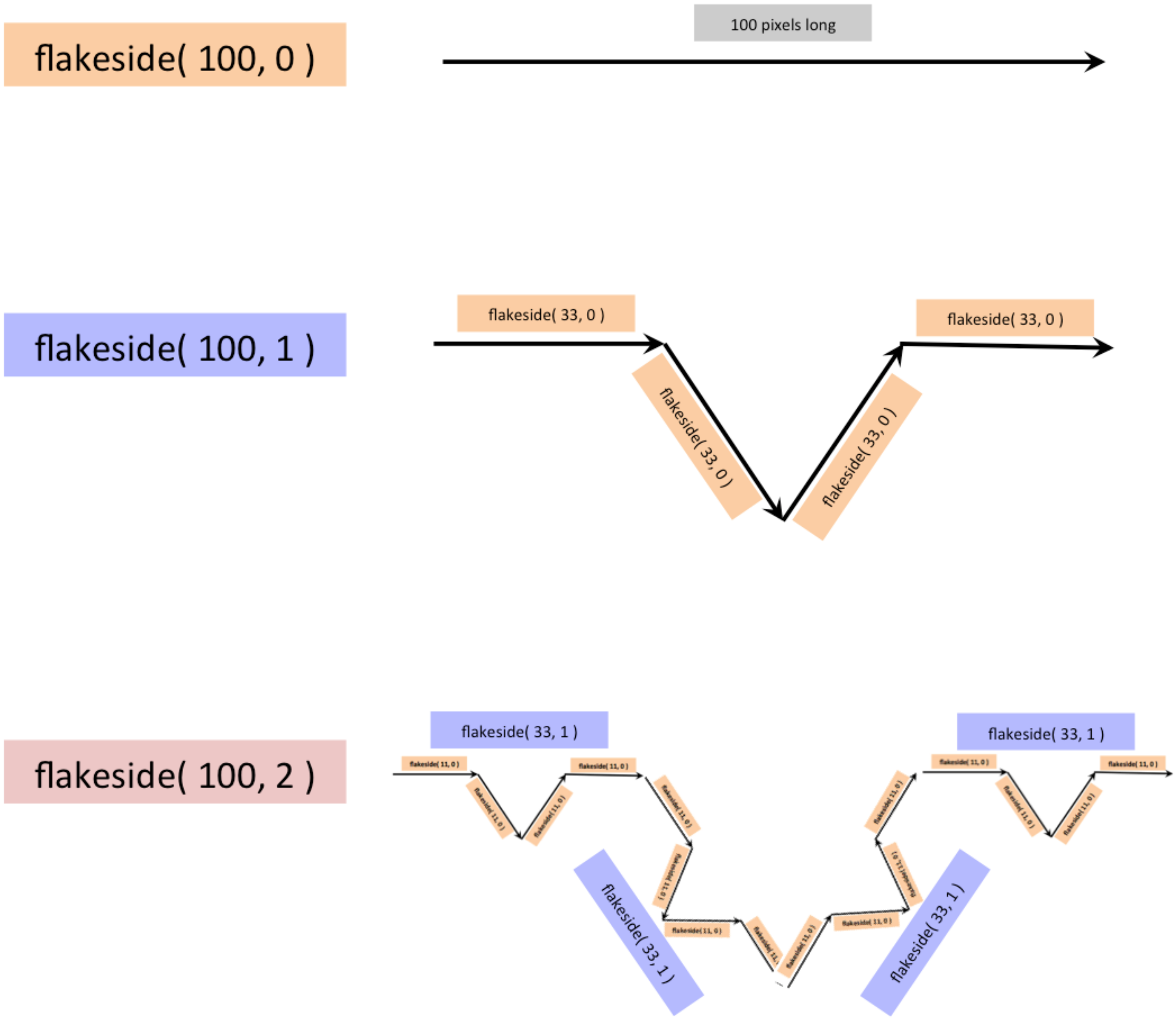
The `flakeside` Function

The Koch Snowflake is an example of very deeply branching recursion. Here, however, the branching is **inward** rather than outward, as in `svtree`.

The Koch snowflake is a fractal with three identical sides—it's the sides themselves that are defined recursively. Because of this, we provide the overall `snowflake` function for you to use—here it is:

```
def snowflake(sidelength, levels):
    """ fractal snowflake function
          sidelength: pixels in the largest-scale triangle
side
          levels: the number of recursive levels in each side
    """
    flakeside( sidelength, levels )
    left(120)
    flakeside( sidelength, levels )
    left(120)
    flakeside( sidelength, levels )
    left(120)
```

Your task is to implement `flakeside(sidelength, levels)` to complete the definition of the Koch Snowflake. First, here is a graphical summary of its structure:



Hints

A base-case Koch snowflake side is simply a straight line of length `sidelength`.

Each recursive level replaces the *middle third* of the snowflake's side with a "bump," i.e., two sides that would be part of a one-third-scale equilateral triangle.

See if you can determine the self-similar structure of the Koch snowflake! Some hints:

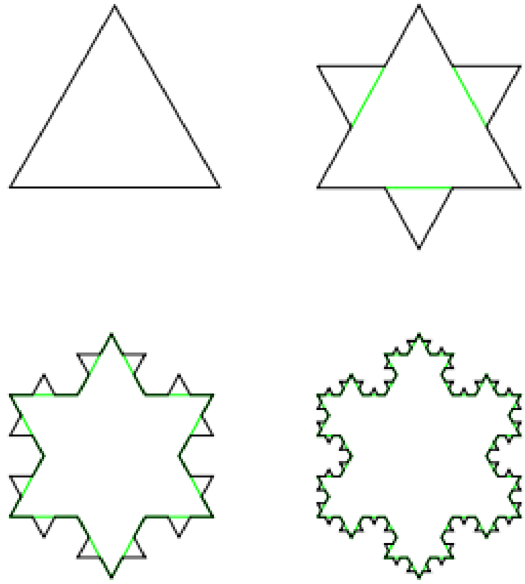
- If `levels` is zero, the base case, then `flakeside` should produce a single segment (base case!)
- If `levels` is zero, notice that each `flakeside` at level `3` contains four `flakesides` of level `2`
- Each `flakeside` at level `2` contains four `flakesides` of level `1` and so on, so `flakeside` will need to call itself ***four*** times!

Remember that `flakeside` is only creating one of the three sides of the snowflake! Because of this, it does ***not*** have to end in precisely the same location as it begins... (If it did, all three sides would be on top of one another...)

Here are images of four different values of `levels` for a snowflake, `0`, `1`, `2`, and `3`:

More information on this Koch fractal curve is here, among other places on the web.

Submit Homework 3, Problem 1



0 point possible (ungraded)

For this problem, your code will not be graded. However, we do still ask you to submit all of your code.

To submit Homework 3, Problem 1, you'll need to copy your code from your trinket and paste it into the box below. After you've pasted your code below, click the "Check" button.

IMPORTANT: Make sure that there aren't spaces at the beginning of your code, and that you copied all of the characters. If there are extra spaces or you are missing spaces, our server won't be able to run your code and we won't be able to give you any of the points you deserve for your hard work.

```
from turtle import *
```

```
import time

def poly(n, N):
    """ draws n sides of an N-sided regular polygon """
    if n == 0:
        return None
    else:
        forward( 50 )
        left( 360.0/N )
```

```
        poly( n-1, N )

def spiral(initialLength, angle, multiplier):
    """ draws spiral """
    if (initialLength < 1) or (initialLength > 500):
        return None
    else:
        forward(initialLength)
```

Press ESC then TAB or click outside of the code editor to exit
correct

correct

Test results
CORRECT See full outputSee full output

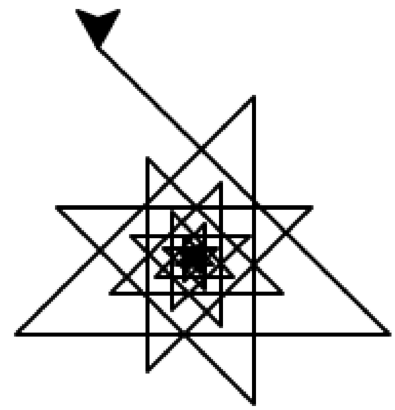
You have used 1 of 3 attempts Some problems have options such as save, reset, hints, or show answer. These options follow the Submit button.

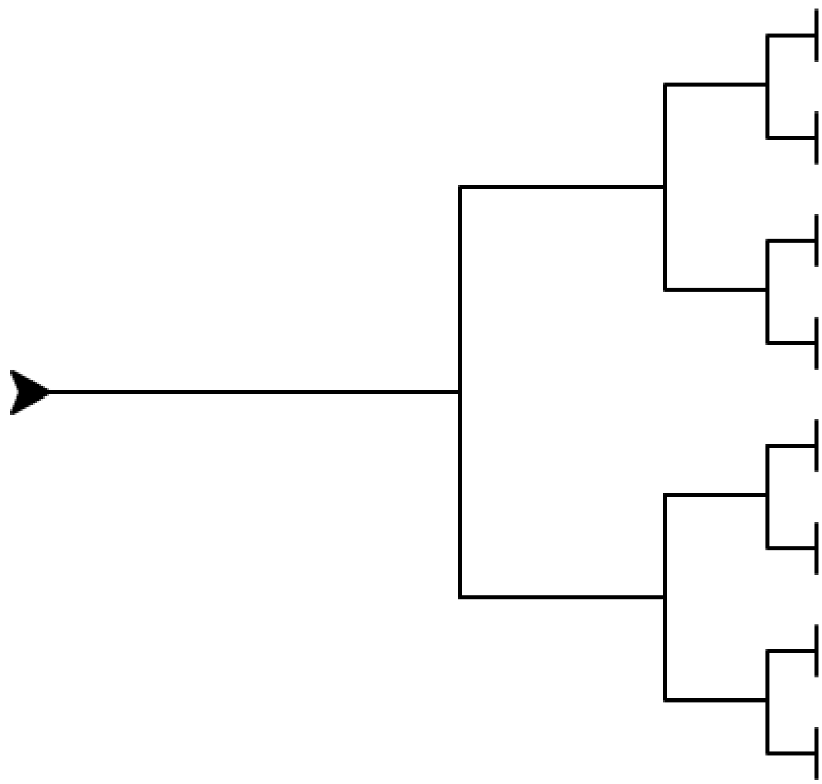
Homework 3, Problem 1 Questions

30.0/30.0 points (graded)

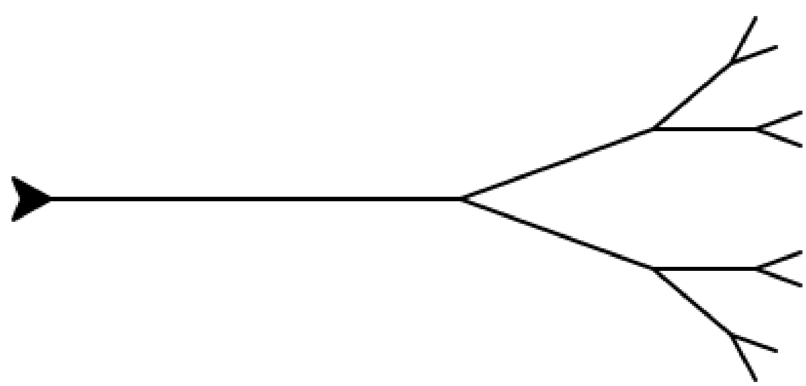
Does your `spiral` function produce a spiral similar to the one shown below when you run the line `spiral(2, 135, 1.1)`?

Does your `chai` function produce an image similar to the one shown below when you run the line `chai(100)`?





Does your `svtree` function produce a tree similar to the once shown below when you run the line `svtree(100, 4)` ?



Does the `snowflake` function produce a Koch snowflake similar to the image below when you run the line `snowflake(100, 2)` ?

You have used 1 of 3 attempts Some problems have options such as save, reset, hints, or show answer. These options follow the Submit button.