Objective. Writing recursive functions and develop “recursive thinking.”

Recursion is a powerful design technique, but it can be a difficult concept to master. In this lab, we will concentrate on a variety of recursion problems. The goal of this lab is to practice writing recursive programs and to train your brain to think recursively.

When writing a recursive function, always start with one or more base cases. These are problems that are so small that we can solve them directly. Then, write the recursive cases. These are simpler problems that take you closer to the base case. Make sure there is progress, otherwise you may never get to the base case. When you get answers to the subproblems, think carefully how these results can be used to construct the answer to the problem at hand.

Getting Started. Clone the lab resources from the gitlab repository, as usual:

```
git clone https://evolene.cs.williams.edu/cs134-labs/22xyz3/lab06.git ~/cs134/lab06
```

where your CS username replaces 22xyz3. You will the find five Python files gap.py, bedtime.py, vortex.py, shrub.py and quilt.py each corresponding to Tasks 1 to 5 respectively.

Required tasks. This week, you must attempt two of the following programs. Some are textual, while others are examples of graphical recursion, completed using the turtle package. For Tasks 1 & 2, we expect you to write new doctests as part of solving the Task. We encourage you to read through all the descriptions before you choose.

Task 1. Recursion over Python lists.
In this task, you will write the `firstGap` function in the file gap.py. Suppose you have a list of integers, `intList`. Given a non-negative integer parameter `limit`, we want `firstGap` to return the value of the first difference (or “gap”) it finds between two consecutive integers in `intList` that is strictly greater than `limit`. If no such gap exists, `firstGap` should return `-1`. This function must be implemented recursively; no loops should be used.

Here are examples of how your function should behave:

```
>>> firstGap([12,2,34,11], 13)
32
>>> firstGap([1,2,6], 4)
-1
>>> firstGap([1,1,2,6], 3)
4
>>> firstGap([], 17)
-1
```

1This lab assignment has been partially adapted from The Sampler Quilt by Julie Zelenski & Eric Roberts (1999 - 2001, Nifty Assignments).
Remember to comment, write new doctests, and thoroughly test your function before committing your solution.

Task 2. Accumulating recursion.
In this task, you will write a function called `bedtimeStory` which, given a list of character names as strings, produces a list of strings representing a bedtime story. We then provide the procedure `formatPrint` which prints out this list of of “story strings” in a nicely readable format. The bedtime story idea we follow here is a common pattern across cultures where a simple phrase repeats multiple times in a nested fashion. Your final implementation must be recursive and cannot use any loops.

Running the file `bedtime.py` as a Python script takes, as command line arguments, the set of characters you’d like in the story. For instance,

```
python3 bedtime.py moose bear reindeer
```

will produce the following story:

```
So the mother of the moose told them a story to fall asleep about a bear...
So the mother of the bear told them a story to fall asleep about a reindeer...
   and then the bear fell asleep.
and then the moose fell asleep.
```

Note that the last character in the list (reindeer) in the above example, is just the object of a story for the bear, not a subject of its own story.

You can test out the `bedtimeStory` function directly in interactive Python; here is is one test you may want to try:

```
>>> from bedtime import bedtimeStory
>>> bedtimeStory(['parrot', 'budgie', 'flamingo', 'heron'])
['So the mother of the parrot told them a story to fall asleep about a budgie...',
 'So the mother of the budgie told them a story to fall asleep about a flamingo...',
 'So the mother of the flamingo told them a story to fall asleep about a heron...',
 'and then the flamingo fell asleep.',
 'and then the budgie fell asleep.',
 'and then the parrot fell asleep.]
```

See the doctests in `bedtime.py` for some additional examples. When you are writing your new doctests, note that newlines between elements of lists can cause doctests to fail. Please make the tests as short as you can, but you can go over 80 characters if needed (as in the test provided with the starter code).
Recursive Graphics with the Turtle module.

Students using Windows, you have a couple additional steps to take to be able to fully execute the turtle questions. If you encounter any issues, please ask one of the instructors!

1. Make sure you have run the instructions on Pages 7 & 8 of the Setting up your Computer document.

2. Each time you restart your computer, you need to do the XLaunch steps. Go to the Search bar in the bottom left and type XLaunch and open the app. Click through until you get to the Extra Settings page and the make sure to check the Disable access control box. If, after you press Finish, you get a warning from the Windows Defender Firewall, make sure you click both checkboxes (Private networks and Public networks), and then press Allow access.

3. Open a new Ubuntu window. Run the command export DISPLAY=:0 which you will need to do each time you open a new Ubuntu window for this lab. If this does not work, you may need to run export DISPLAY=localhost:0 instead.

To test whether you can effectively use the turtle package and view the turtle window, go into interactive Python (by typing python3) and run the following commands:

```python
>>> from turtle import *
>>> forward(100)
```

You should see a window popup with a turtle, and the turtle should move forward by 100 units, as shown in lecture. If this does not generate an error, you should be good to go!

Task 3. Draw a Vortex.
In this task, you will create a vortex of color made up of squares. Look at these cool examples!

```python
drawVortex(200, 5, [ORANGE, GRAY])
drawVortex(250, 25, [RED, GREEN, BLUE])
drawVortex(500, 10, [RED, ORANGE, GOLD, GREEN, BLUE, PURPLE, GRAY])
```
In `vortex.py`, write the function `drawVortex(width, border, colorList)` where the input parameters are:

- **width**, the side length of each square that makes up the vortex. This will change as each subsequent square is drawn inside of the other
- **border**, the amount of vertical and horizontal space between square \( n \) and square \( n+1 \); in other words, each square should always be offset to the east and to the north border units from the square drawn before it.
- **colorList**, the colors in the vortex. The list can have 1 or more colors. Once colors run out, your implementation should loop back around to pick the first color in the list again. (There are some colors specified as global variables at the top of the file. Feel free to experiment with your own “hex” colors!)

The following helper functions have been defined for you in `vortex.py`.

- **drawRectangle(width, length, color)**. With the turtle starting in the bottom lefthand corner, pointing east, this routine draws a rectangle of with `width` unit top and bottom, and `length` unit sides, filled with color `color`.
- **initializeTurtle(size)**. Sets up a screen of size slightly bigger than the pattern, resets the turtle, and sets its position to be lower-left endpoint of the pattern, that is, \((-size/2, -size/2)\). The turtle points east.
- **testdrawVortex(width, border, colorList)**. Calls the `initializeTurtle` function followed by the `drawVortex` function and saves the resulting figure as an `.ps` file.

Students using Windows, to open the `.ps` file generated by `drawVortex(200, 5, [ORANGE, GRAY])`, you would run the command:

```
$ gv drawVortex-200-5-orange-gray.ps
```

The only turtle commands you will need to use in your function are `forward`, `left`, and `right`.

**Testing your function.** You can test your function by uncommenting the provided function calls to the `testdrawVortex` function in the `if __name__ == '__main__':` block and comparing the output to the examples provided here. You may change the speed of the turtle in the `initializeTurtle` function to adjust the speed of the animation.

**Extra thinking about vortex.**

1. How would your implementation need to change if you wanted to make a vortex of rectangles (i.e., where the length and width are not the same)? What about circles? Triangles?

2. How would your implementation need to change to accommodate `border` being a randomly chosen positive integer?
Task 4. Recursive Shrubs.
In this task, you will write a fruitful recursive function named `shrub` in the file `shrub.py` that draws a tree pattern and returns a tuple of values, described below.

The function `shrub(trunkLength, angle, shrinkFactor, minLength)` takes in four parameters:

- `trunkLength`, the length of the vertical branch at the base of the shrub
- `angle`, the angle between a trunk and its right and left branches
- `shrinkFactor`, the length of the right and left branches relative to their trunk. Specifically, the length of the right branch is `shrinkFactor` times the `trunkLength`, and the trunk of the left branch is `shrinkFactor * shrinkFactor` times the `trunkLength`
- `minLength`, the minimum branch length in the shrub.

The `shrub` function (in addition to drawing the shrub) must return a pair of items, where

- the first item is the total number of branches in the shrub, including the trunk
- the second item is the total length of branches in the shrub, including the trunk

The following helper functions have been defined for you in `shrub.py`:

- `initializeTurtle()`. Sets up the screen, resets the turtle, and positions it at an appropriate spot, and orients its pointer to face north.

- `testShrub(trunkLength, angle, shrinkFactor, minLength)`. Calls the `initializeTurtle` function followed by the `shrub` function, prints the tuple returned and saves the figure generated.

Please limit your `shrub` to using `forward`, `backward`, `left`, and `right` turtle commands.

See the sample invocations of the shrub function on the next page, with the tuple after the function call `->` indicating the value returned by that invocation.

Testing your `shrub` function. You can test your function by uncommenting the provided function calls to the `testShrub` function in the `if __name__ == '__main__':` block and comparing the output to the examples here and the expected return values provided in comments. You may change the speed of the turtle in `initializeTurtle` function to adjust speed of the animation.
Task 5. Draw a Williams Quilt.
In this task, you will (re)visit the pattern idea, similar to Task 3, but this time you will build a quilt! Some of the helper functions are similar between Task 3 and Task 5, but be on the look out for some subtle differences if you attempt both questions.

In quilt.py, you will define the function `drawQuilt(size, level, color1=PURPLE, color2=GOLD)` where the input parameters are described below:

- **size** denotes the side length of the whole quilt (that is, the largest square); the side length of each successive square goes down by half
- **level** determines how many recursive subpatterns the quilt will have—a quilt of level $\ell$ has a quilt of level $\ell - 1$ as its upper-left and lower-right subpattern. In particular, $\text{level} = 0$ means nothing is drawn, $\text{level} = 1$ means the entire quilt is one solid square, $\text{level} = 2$ means the quilt has two
level 1 quilts as subpatterns, and so on.

- **color1** and **color2** denote the colors in the quilt, set by default to **PURPLE** and **GOLD**, respectively. (Here **PURPLE** and **GOLD** are global variables that have been predefined with the corresponding HEX color codes.) The colors of the quilt alternate between **color1** and **color2**, starting with **color1**.

The following helper functions have been defined for you in quilt.py:

- **drawSquare(size, color)**. Draws a square of side length *size* filled with color *color*, with the turtle's starting position as one of the endpoints. Your function must call **drawSquare** only once.

- **initializeTurtle(size)**. Sets up a screen of size slightly bigger than the pattern, resets the turtle, and sets its position to be lower-left endpoint of the pattern, that is, \((-size/2, -size/2)\).

- **testDrawQuilt(size, level, color1 = PURPLE, color2 = GOLD)**. Calls the **initializeTurtle** function followed by the **drawQuilt** function and saves the resulting figure as an ps file.

Aside from the single call to **drawSquare**, the remaining pattern must be drawn using recursion. The only turtle commands you can use in your function are **forward**, **backward**, **left**, and **right**.

Consider the following invocations of **drawQuilt**, which describe how your method should behave.
Submitting your work.
When you’re finished, add, commit, and push the Python files corresponding to the tasks you completed to the server as you did in previous labs. For Tasks 1 & 2, a reminder that you are required to add new doctests. For Tasks 3, 4, and 5, in addition to the Python file, you must add, commit, and push the figure generated by the following function calls:

- If attempting Task 3, submit the figure generated by `drawVortex(500, 10, [RED, ORANGE, GOLD, GREEN, BLUE, PURPLE, GRAY])`.
- If attempting Task 4, submit the figure generated by `shrub(100, 15, 0.8, 10)`.
- If attempting Task 5, submit the figure generated by `drawQuilt(500, 4)`.

Remember that you must certify that your work is your own, by typing out the Honor Code statement in the `honorcode.txt` file, committing and pushing it along with your work.

Perpetual Grading Expectations. In addition to the lab-specific grading guidelines contained above and in `GradeSheet.txt`, remember that you are assessed on whether your code meets the general course expectations. For full coverage of what style entails, please see the [Style Guide](#).

As a guide to avoiding common errors, make sure to follow these guidelines:

- Always sign the honor code. It protects your work.
- Follow the 80-character rule. This makes your code more readable.
- Use meaningful variable names. Help the reader understand your code.
- Comment your code. Logic that was hard to develop is often hard to understand.
- Write entirely new tests of your own design. Ours can be improved.
- Every doctest should be meaningfully different. Address standard use and unusual “corner” cases.
- When done, review the lab handout and make sure you’ve addressed the requirements.