Computer Science CS134 (Fall 2020)

Daniel Aalberts, Duane Bailey, & Molly Feldman

Laboratory 7

Fun with Recursion[1] *(due Thursday at 5pm)*

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/lab07.git ~/cs134/lab07
```

where your CS username replaces 22xyz3. You will the find five Python files `girlscouts.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`[2] package. We encourge you to read through all the descriptions before you choose.

Task 1. Recursion over Python lists.

In this task, you will write the `totalCookieOrder` function in the file `girlscouts.py`. This question asks you to help out a Girl Scout (circa the 1990s) with their cookie order for Girl Scout cookie season. This particular Girl Scout decided to only sell Tagalongs (Peanut Butter Patties), so we do not have to worry about keeping track of many types of cookies.

Back then, a Girl Scout had order forms and would ask people for both the number of cookies they would want and to pass along order forms to other people (their neighbors, etc.). In our set up, each holder of an order form has a list reflecting the orders they helped initiate. This list can contain integers, the number of boxes ordered, and other lists, reflecting orders made using the order forms they passed out.

In this set up, the function `totalCookieOrder` takes as input a list `orders` as input and computes and returns the total number of boxes of Tagalongs the Girl Scout should order. Your function must be recursive and should not use any loops.

[1] This lab assignment has been partially adapted from Wellesley Fall 2018 course materials as well as The Sampler Quilt by Julie Zelenski & Eric Roberts (1999 - 2001, Nifty Assignments).

[2] You can find a video tutorial on the `turtle` package at `https://youtu.be/1gs2POBsTFs` or off the course pages.

Here are examples of how your function should behave:

```
>>> totalCookieOrder([])
0
>>> totalCookieOrder([1, 3, 2])
6
>>> totalCookieOrder([4, [5, 6], 10, [[1, 1], 1]])
28
```

In each example, the total order corresponds to the sum of all the integers that appear in the lists, no matter how deeply nested.

Task 2. Accumulating recursion.
In this task, you will write a function called `bedtimeStory` which, given a list of character names as strings, prints out a bedtime story. This bedtime story follows 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 moose's mother told them a story to fall asleep about a bear...
So the bear's mother 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. If you want to import the function into interactive Python, we recommend testing it by running `print(bedtimeStory(lst))`, where `lst` contains your list of test characters. For example, here is one test you may want to try:

```
>>> print(bedtimeStory(['parrot', 'budgie', 'flamingo', 'heron']))
So the parrot's mother told them a story to fall asleep about a budgie...
So the budgie's mother told them a story to fall asleep about a flamingo...
So the flamingo's mother 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, including where there should be newlines and spaces. This function is a lot about string formatting!

Recursive Graphics with the `Turtle` module.

The next three tasks make use of the `turtle` module in Python. The module uses `tkinter` for the underlying graphics, so you need a version of Python installed with `Tk` support. It is likely your installation of Python is sufficent.

To test whether your version can use the `turtle` package, go into interactive Python (by typing `python3`) and run the following import command:

```
>>> from turtle import *
```

If this does not generate an error you should be good to go—to see the turtle graphics window, you can further type `forward(20)` which will pop up the window and move the turtle forward by 20 units.

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. `Tk` support likely is not pre-installed on the Windows Subsystem for Linux. You can install all the tools we need opening up your Ubuntu installation and then run the following command:

   ```
   sudo apt install python3-tk ghostscript gv
   ```

   Press `y` to approve the install. Remember your password for sudo on Ubuntu is different than the password for your computer at-large! Close Ubuntu.

2. Go to `https://sourceforge.net/projects/vcxsrv/` and download the VcXsrv Windows X Server. Follow all the default installer instructions; for instance, you *do* want it to be downloaded on "normal" Windows (e.g. `C:\ProgramFiles(x86)\ ...`)

3. Once that succeeds, `vcXsrv` is now a Windows app. We need to set some settings, so 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 then make sure to check the `Disable access control` box.

4. If after you press `Finish` you get a warning from the Windows Defender Firewall, make sure to click both checkboxes (yes for `Private networks` and `Public networks`).

5. Now open Ubuntu! 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.

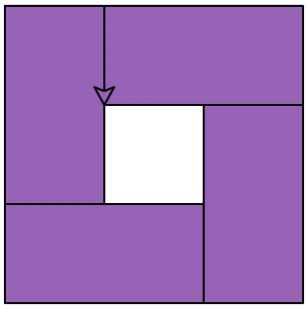
6. Navigate to your cs134 folder and you should be able to type `python3 quilt.py` and a turtle window should pop up (although it will close quite quickly!)

7. You should also be able to use the `gv` command to view `.ps` and `.pdf` files, as well.

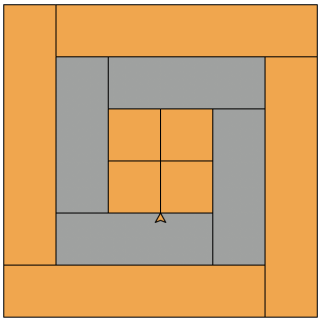
If you want the graphics window to only close after a mouse click, use the turtle command `exitonclick()`.

Task 3. Draw a Vortex.

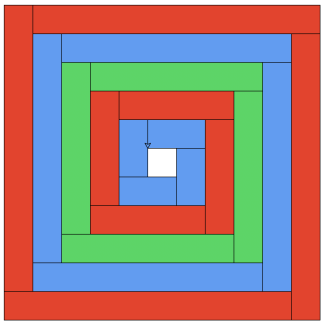
In this task, you will create a vortex of color made up of offset bars. Look at these cool examples!



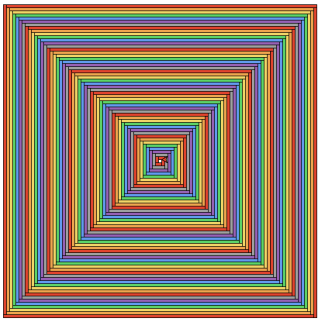
`drawVortex(50, 100, [purple])`



`drawVortex(50, 250, [orange, gray])`



`drawVortex(50, 500, [red, blue, green])`



`drawVortex(5, 500, [red, orange, gold, green, blue, purple, gray])`

In `vortex.py`, write the function `drawVortex(width, length, colorList)` where the input parameters are:

- `width`, the side length of each "bar" that makes up the vortex. According to our pattern, this value remains the same for the whole vortex. The smaller the width, the more recursive calls we will get!
- `length`, the length of the long side of each "bar". This changes as the pattern moves forward.
- `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`.

- `drawBar(width, length, color)`. Draws a bar (a rectangle) of short side length `width` and long side length `length` filled with color `color`, with the turtle's starting position as one of the endpoints.
- `initializeTurtle(size)`. Sets up a screen of size slightly bigger than the pattern, resets the turtle, and sets its position to be in the lower right of the screen.
- `testdrawVortex(width, length, colorList)`. Calls the `initializeTurtle` function followed by the `drawVortex` function and saves the resulting figure as an `ps` file.

The only turtle commands you will need to use in your function are `forward`, `backward`, `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. For what values of `width` and `length` will the pattern have a hole in the middle versus be completely filled in?

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`:

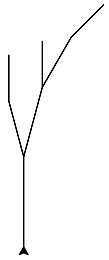
- `initializeTurtle()`. Sets up the screen, resets the turtle, and positions it at an appropriate spot, and orients its pointer to face north.
- `testShrub(trunkLength, angle, shrinkFactor, minLength)`. Calls the `initializeTurtle` function followed by the `shrub` function, prints the tuple returned and saves the figure generated.

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

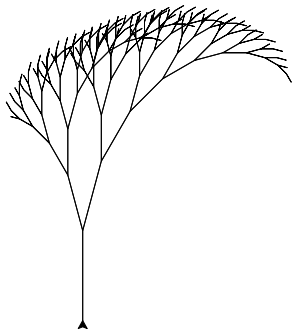
See the sample invocations of the shrub function on the next page, with the tuple after the function call -> indicating the value returned by that invocation.

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.

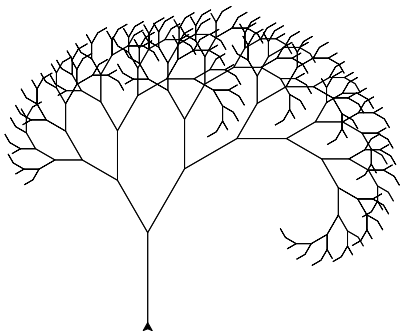
`shrub(100, 15, 0.8, 60) ->`
`(4, 308.0)`



`shrub(100, 15, 0.8, 50) ->`
`(7, 461.6)`



`shrub(100, 15, 0.8, 10) ->`
`(12, 666.4000000000001)`



`shrub(100, 30, 0.82, 10) ->`
`(376, 6386.440567704483)`

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.

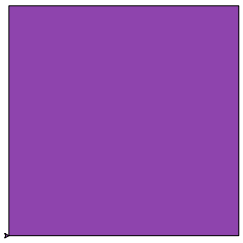
In `quilt.py`, you will define the function `drawQuilt(size, level, color1=purple, color2=gold)` where the input parameters are described below:

- `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 ℓ has a quilt of level $\ell - 1$ as its upper-left and lower-right subpattern. In particular, `level = 0` means nothing is drawn, `level = 1` means the entire quilt is one solid square, `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`.

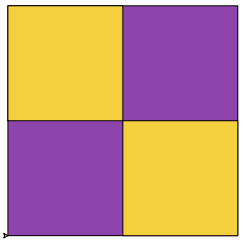
The following helper functions have been defined for you in `quilt.py`:

- `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.

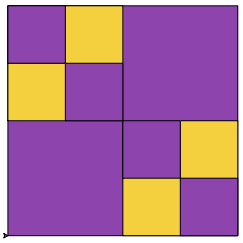
Aside from the single call to `drawSquare`, the remaining pattern must be drawn using recursion. The only turtle commands you can use in your function are `forward`, `backward`, `left`, and `right`.
Consider the following invocations of `drawQuilt`, which describe how your method should behave.



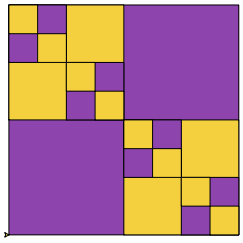
`drawQuilt(500, 1)`



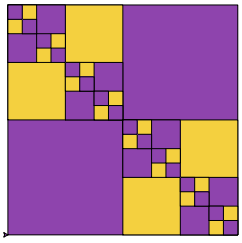
`drawQuilt(500, 2)`



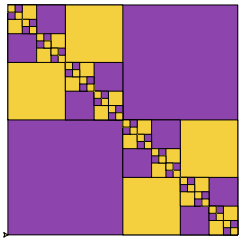
`drawQuilt(500, 3)`



`drawQuilt(500, 4)`



`drawQuilt(500, 5)`



`drawQuilt(500, 6)`

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 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(50, 500, [orange, 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.

⋆