Armando Ortiz

Lab Report 1

**Introduction**

This first lab was meant to draw different shapes using Python recursively. The four shapes are squares, two different types of circles, and a tree. The squares contain even smaller squares on each corner of the prior larger square. The first type of circle contains concentric circles that are stuck to the left side of the largest circle, while getting smaller, but never exiting the circle. The second type of circle contains a pattern resembling a plus sign of 5 circles within each circle. Each smaller circle contains the pattern as well. The final design is a binary tree with branches coming out of the left and right side, and these branches come out of every branch for as many levels as necessary. These patterns all must be drawn recursively.

**Proposed Solution Design and Implementation**

All of these different shapes needed to be tackled differently as far as designing the algorithm. Understanding the code provided by the professor was the first important part. The program to draw nested squares assisted with the square figures, and the circle program helped with the two circle figures. Drawing the tree had to be done in a different way than these shapes. The main way to tackle these problems was by breaking them down to their own problem, function by function. The rest of the program was simple, because it would just be calling the drawing functions using test variables.

The main difficulty for the squares was figuring out how to make sure the midpoint of the new square would be on each corner of the old square. Recursively drawing a square on each corner of the larger square was becoming a problem, but the breakthrough came with the idea that each recursive call would be for one corner of the square. So, there are four recursive calls, one for each corner. The function to draw the squares contains 5 inputs: ax to plot the points and draw the square, the number of recursive calls, an origin number for each coordinate, the 2D-array containing the coordinates for the square, and the width to make the square smaller each recursive call. The actual function has q be a list that is the result of the multiplication of the array with the width, making the new square smaller. Then, four new square arrays are made by adding x and y coordinates of the array to the origin number. Square one, the top right corner, is found by adding the origin number to all coordinates. Square two, bottom left is found by subtracting the origin number from all coordinates. Square three, top left, is found by subtracting the origin from the x-coordinates and adding the origin to the y-coordinates. Square four, bottom right is by adding the origin to the x-coordinates, and subtracting from the y. Then, these squares are plotted each recursive call, and the only variable changed is n, the number of iterations. Each corner is its own recursive call.

The tree function is found by bringing in an array of two coordinates, x,y, del y, and n. X and Y are the length of each line, and del y is the same as the number of iterations. This was going to be a number to divide the height of each branch, but that removed from the calls. This was leftover as a mistake. The function then has two separate variables for each branch. The left branch is created by subtracting x and y from the indices of the array, and the right is found by adding x to the zero index and subtracting y from index 1. These are then plotted by having the matplotlib plot draw from the original array, to the indices of the newly created lines. Finally, there are recursive calls for each branch, with the x variable being divided by 2 to make them shorter each iteration.

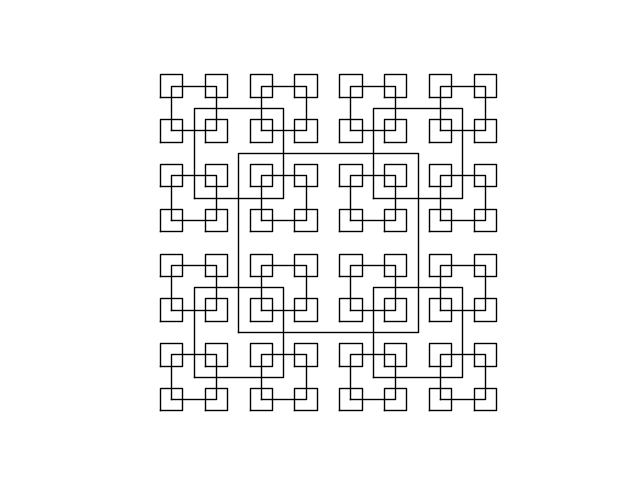
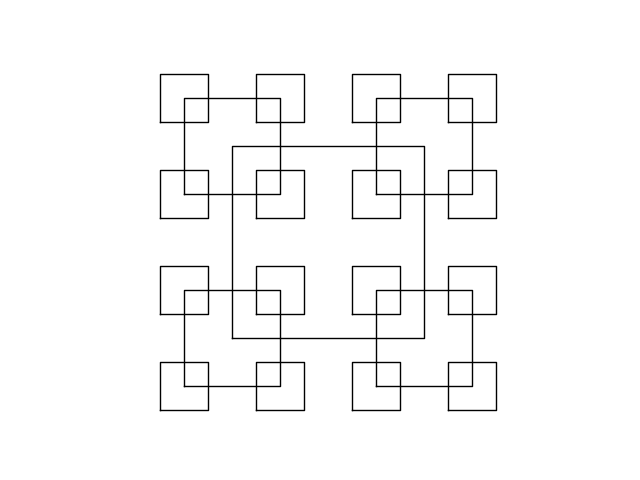
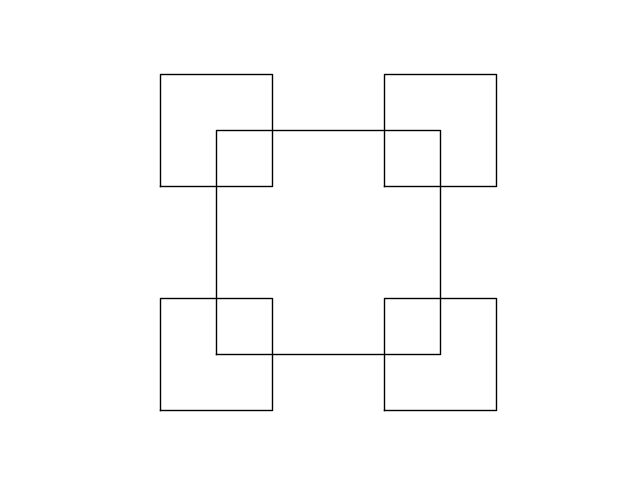
The first circle function, the one that sticks to the left, brings in a center, a radius, and a width. First, the circle is created by calling the circle creation function provided by the professor. Then, the x value created by the circle function is moved to the right by adding it with the radius. This slightly moves each circle to the right. This circle is then drawn by the plot function. The function only has one recursive line, but the radius is multiplied by the width each call, making the movement to the right smaller each time. This keeps each smaller circle within the larger circles. Finally, the different calls of this function from the main have different w-variables to better match the examples.

The second circle function, the pattern of circles inside of larger circles, mixed the problems of the circle and square functions. The circles needed to stay inside the bigger circles, while staying in their own section of the larger circle. This method had the same variables in the header as the other circle function, but the main change came with the recursive calls. This function has five recursive calls, one for each circle. The call for the center circle has the radius multiplied by the width to make it smaller. The right and top circle add the radius to their respective x and y coordinates to move them up and right, but subtract the radius times the width again, to keep them within their areas. Meanwhile, the left and bottom circles subtract the radius from the center variables, but add the radius times the width.

**Experimental Results**

The experiments for this program were trying to recreate the designs from the lab worksheet. The square figure was tested using N = 2, N = 3, and N = 4 in the function call from the main function. The main way of testing the recursive functions is by changing the number of iterations to increase the number of figures drawn.

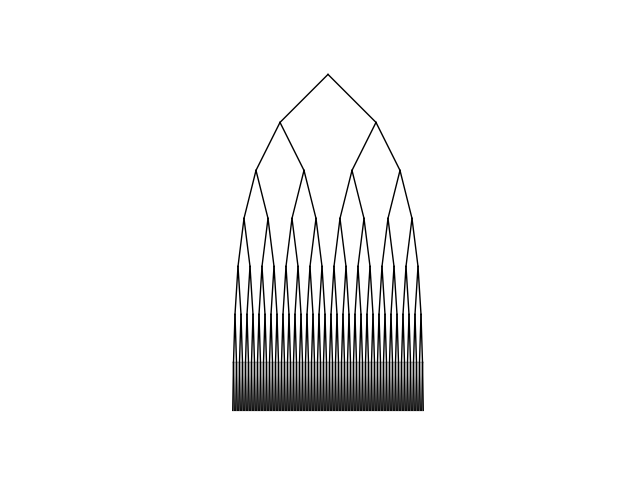
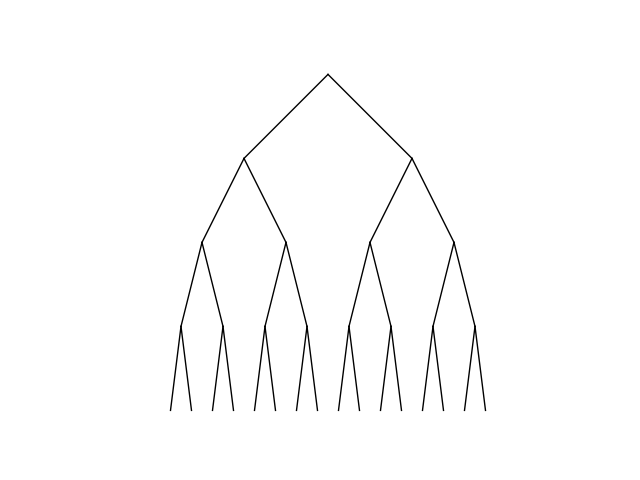
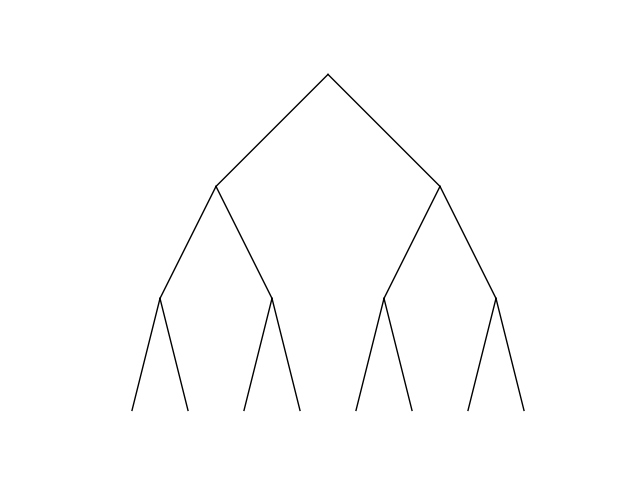
**draw\_squares**



N = 2 N = 3 N = 4

The output for each of the squares did end up matching the ones from the example, but it runs slowly as the input becomes larger due to the number of recursive calls.

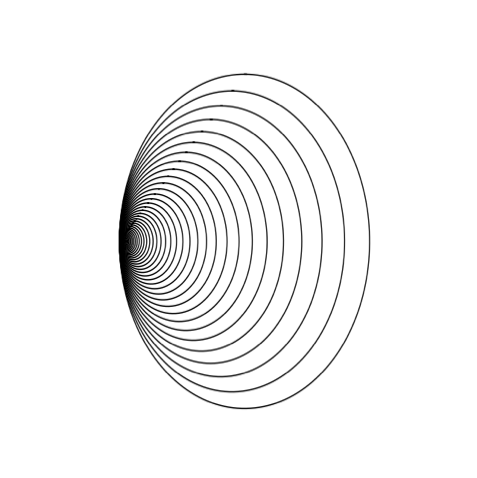
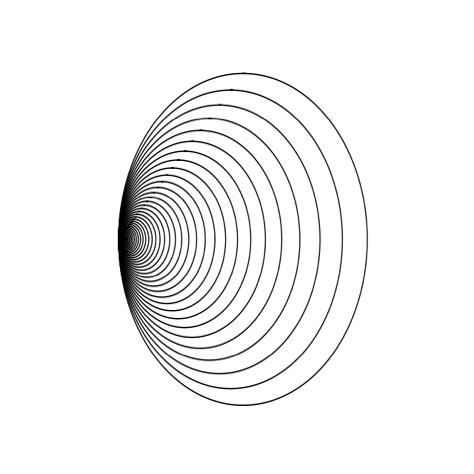
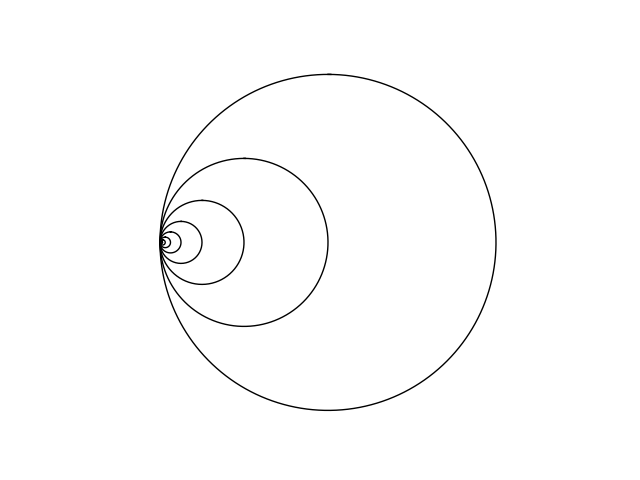
**draw\_trees**



N = 3 N = 4 N = 5

This did not match perfectly with the example from the lab sheet. Mainly with the length of the branches, as they’re supposed to be shorter further down, but they did split up correctly.

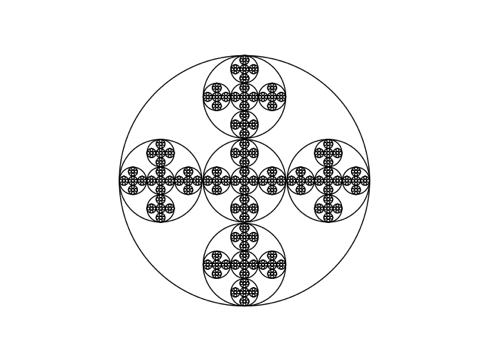
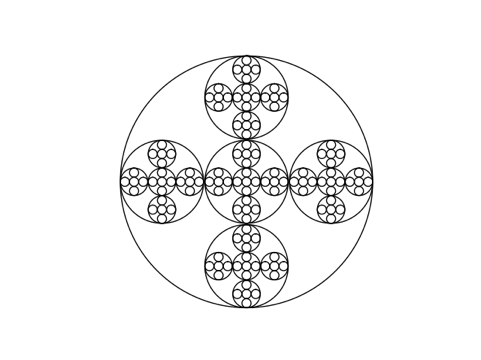
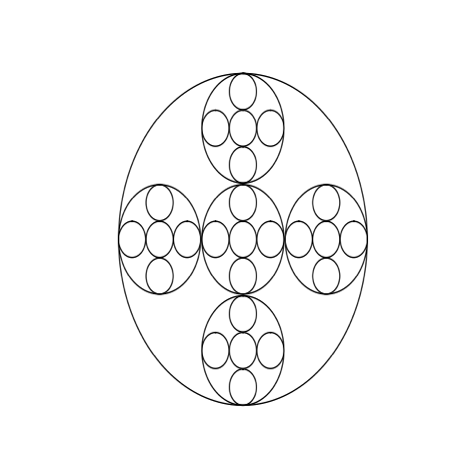
**draw\_circles**



N = 9 N = 50 N = 100

This function did create circles similar to the lab sheet, but in order for it to exactly like it the widths had to change for the different calls.

**drawcirclePattern**



These drawings looked exactly like the worksheet, but run slowly the higher the number of calls, similarly to the square function.

**Conclusions**

This lab proved to be challenging for various reasons. Recursion can often be tricky and performing recursion in a new language with a different library, syntax, and functions is even trickier. Though it did prove helpful to practice recursion, and learn how to draw using Python. It also showed how complex ideas can be performed simply using recursion as none of these functions have many lines of code, but can be difficult come up with conceptually.

**Appendix**

﻿"""

Author: Armando Ortiz

Course: CS 2302 Data Structures

Assignment: Lab 1

Instructor: Olac Fuentes

T.A.: Anindita Nath, Maliheh Zargaran

Last Modification: Feb 10, 2019

Purpose: The purpose of this program is to recursively draw different figures.

"""

import numpy as np

import matplotlib.pyplot as plt

import math

#Draw Squares

def draw\_squares(ax,n,orig,p,w):

if n>0:

q = p\*w #Lowers size of square

#Builds Squares

square1 = q+orig #Top right square

square2 = q-orig #Bottom left square

square3 = np.array([[q[0][0]-orig, q[0][1]+orig], #Top left square

[q[1][0]-orig, q[1][1]+orig],

[q[2][0]-orig,q[2][1]+orig],

[q[3][0]-orig, q[3][1]+orig],

[q[4][0]-orig,q[4][1]+orig]])

square4 = np.array([[q[0][0]+orig, q[0][1]-orig], #Bottom right square

[q[1][0]+orig, q[1][1]-orig],

[q[2][0]+orig,q[2][1]-orig],

[q[3][0]+orig, q[3][1]-orig],

[q[4][0]+orig,q[4][1]-orig]])

ax.plot(p[:,0], p[:,1], linewidth=1, color='k')#Draws Square

#Recursive calls for squares

draw\_squares(ax,n-1,orig,square1,w) #Top Right

draw\_squares(ax,n-1,orig,square2,w) #Bottom Left

draw\_squares(ax,n-1,orig,square3,w) #Top Left

draw\_squares(ax,n-1,orig,square4,w) #Bottom Right

#Draw Tree

def draw\_tree(ax,line,x,y,dely,n):

if n>0:

#Builds Branches

left = [line[0]-x,line[1]-y] #Builds left branch

right = [line[0]+x,line[1]-y] #Builds right branch

ax.plot([line[0],left[0]],[line[1],left[1]], #Draws left branch

[line[0],right[0]],[line[1],right[1]],linewidth=1,color='k') #Draws right branch

#Recursive Calls for branches

draw\_tree(ax,left,x/2,y,dely,n-1) #Left branches

draw\_tree(ax,right,x/2,y,dely,n-1) #Right branches

#Creates Circle

def circle(cent,rad):

n = int(4\*rad\*math.pi)

t = np.linspace(0,6.3,n)

x = cent[0]+rad\*np.sin(t)

y = cent[1]+rad\*np.cos(t)

return x, y

#Draws Circles

def draw\_circles(ax,n,center,radius,w):

if n>0:

x,y = circle(center,radius)

x = x + radius

ax.plot(x,y,linewidth=1,color='k')

draw\_circles(ax,n-1,center,radius\*w,w)

#Draws pattern of circles

def drawcirclePattern(ax,n,c,r,w):

if n>0:

x,y = circle(c,r) #Builds Circle

ax.plot(x, y, linewidth=1, color='k') #Draws Circle

#Recursive calls for circles

drawcirclePattern(ax,n-1,c,r\*w,w)#Middle circle

drawcirclePattern(ax,n-1,[(c[0]+r)-(r\*w),c[1]],r\*w,w) #Right Circle

drawcirclePattern(ax,n-1,[c[0],(c[1]+r)-(r\*w)],r\*w,w) #Top Circle

drawcirclePattern(ax,n-1,[(c[0]-r)+(r\*w),c[1]],r\*w,w) #Left Circle

drawcirclePattern(ax,n-1,[c[0],(c[1]-r)+(r\*w)],r\*w,w) #Bottom Circle

#Draw Squares

orig\_size = 1000

mid = np.array([0,0])

square = np.array([[-orig\_size,-orig\_size], [-orig\_size,orig\_size], [orig\_size,orig\_size],[orig\_size,-orig\_size], [-orig\_size,-orig\_size]])

#First Square

plt.close("all")

fig, ax = plt.subplots()

draw\_squares(ax,2,orig\_size,square,.5)

ax.set\_aspect(1.0)

ax.axis('off')

plt.show()

fig.savefig('square1.png')

#Second Square

plt.close("all")

fig, ax = plt.subplots()

draw\_squares(ax,3,orig\_size,square,.5)

ax.set\_aspect(1.0)

ax.axis('off')

plt.show()

fig.savefig('square2.png')

#Third Square

plt.close("all")

fig, ax = plt.subplots()

draw\_squares(ax,4,orig\_size,square,.5)

ax.set\_aspect(1.0)

ax.axis('off')

plt.show()

fig.savefig('square3.png')

#Draw Tree

line = np.array([0,0])

x = 1000

y = 1000

#First Tree

plt.close("all")

fig, ax = plt.subplots()

draw\_tree(ax,line,x,y,3,3)

ax.set\_aspect(1.0)

ax.axis('off')

plt.show()

fig.savefig('tree1.png')

#Second Tree

plt.close("all")

fig, ax = plt.subplots()

draw\_tree(ax,line,x,y,4,4)

ax.set\_aspect(1.0)

ax.axis('off')

plt.show()

fig.savefig('tree2.png')

#Third Tree

plt.close("all")

fig, ax = plt.subplots()

draw\_tree(ax,line,x,y,7,7)

ax.set\_aspect(1.0)

ax.axis('off')

plt.show()

fig.savefig('tree3.png')

#Draw Circles

#First Circle

plt.close("all")

fig, ax = plt.subplots()

draw\_circles(ax, 9, [0,0], 100,.5)

ax.set\_aspect(1.0)

ax.axis('off')

plt.show()

fig.savefig('circles1.png')

#Second Circle

plt.close("all")

fig, ax = plt.subplots()

draw\_circles(ax, 50, [0,0], 100,.9)

ax.set\_aspect(1.0)

ax.axis('off')

plt.show()

fig.savefig('circle2.png')

#Third Circle

plt.close("all")

fig, ax = plt.subplots()

draw\_circles(ax, 100, [0,0], 100,.9)

ax.set\_aspect(1.0)

ax.axis('off')

plt.show()

fig.savefig('circles3.png')

#drawCirclePattern

#First Pattern

plt.close("all")

fig, ax = plt.subplots()

drawcirclePattern(ax, 3, [0,0], 100,.33)

ax.set\_aspect(1.0)

ax.axis('off')

plt.show()

fig.savefig('circlepattern1.png')

#Second Pattern

plt.close("all")

fig, ax = plt.subplots()

drawcirclePattern(ax, 4, [0,0], 100,.33)

ax.set\_aspect(1.0)

ax.axis('off')

plt.show()

fig.savefig('circlepattern2.png')

#Third Pattern

plt.close("all")

fig, ax = plt.subplots()

drawcirclePattern(ax, 5, [0,0], 100,.33)

ax.set\_aspect(1.0)

ax.axis('off')

plt.show()

fig.savefig('circlepattern3.png')