Lab 1 Adrian Monreal

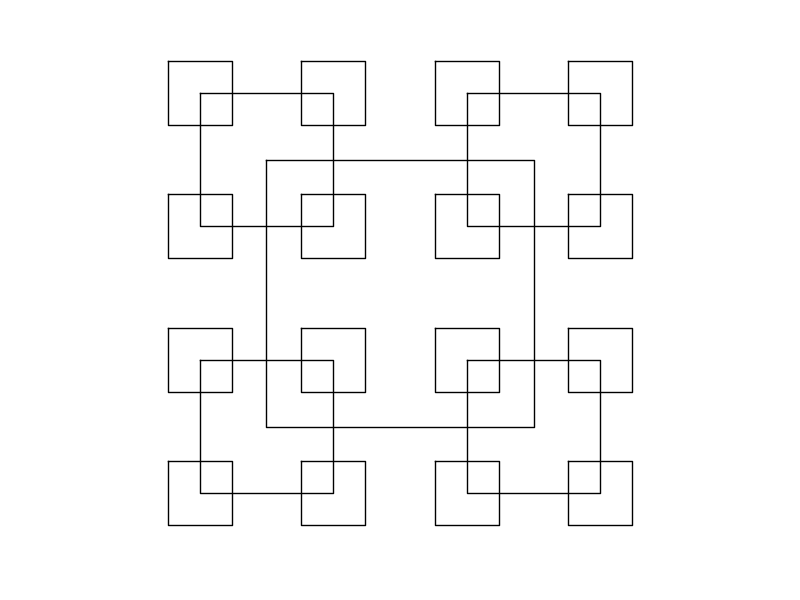
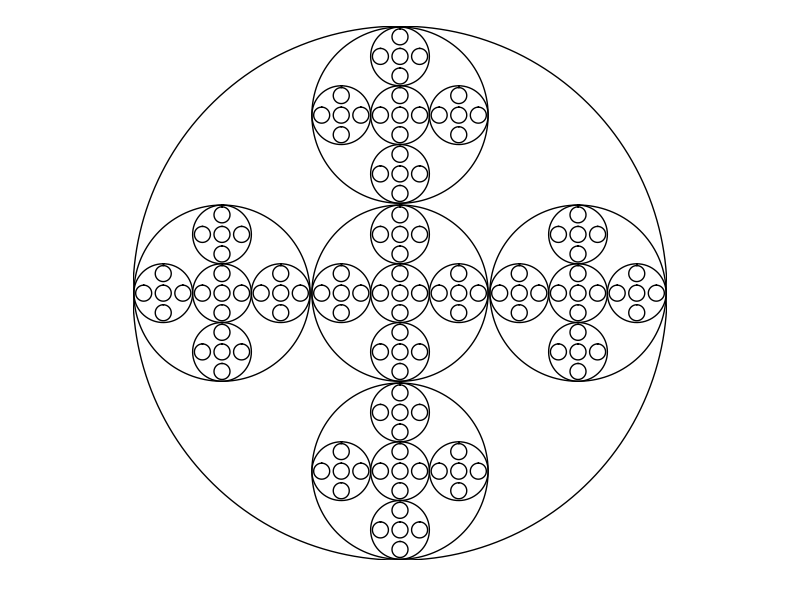
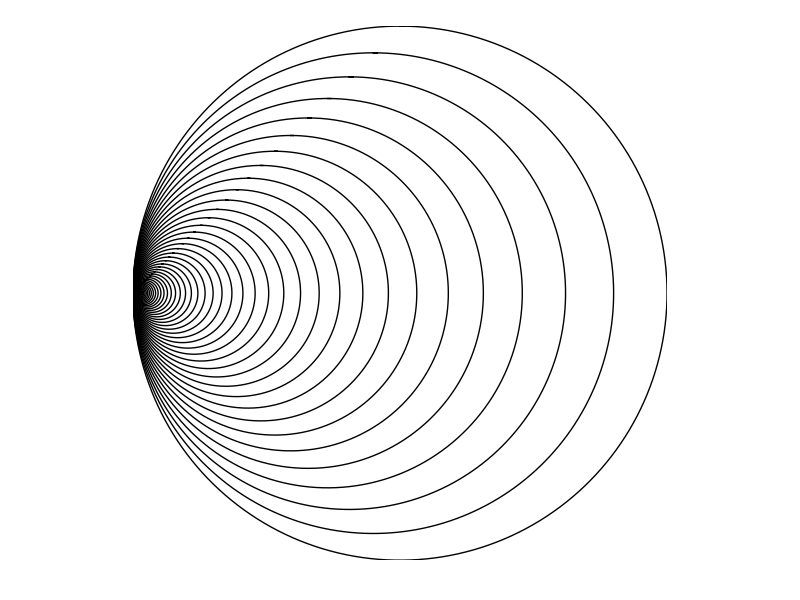
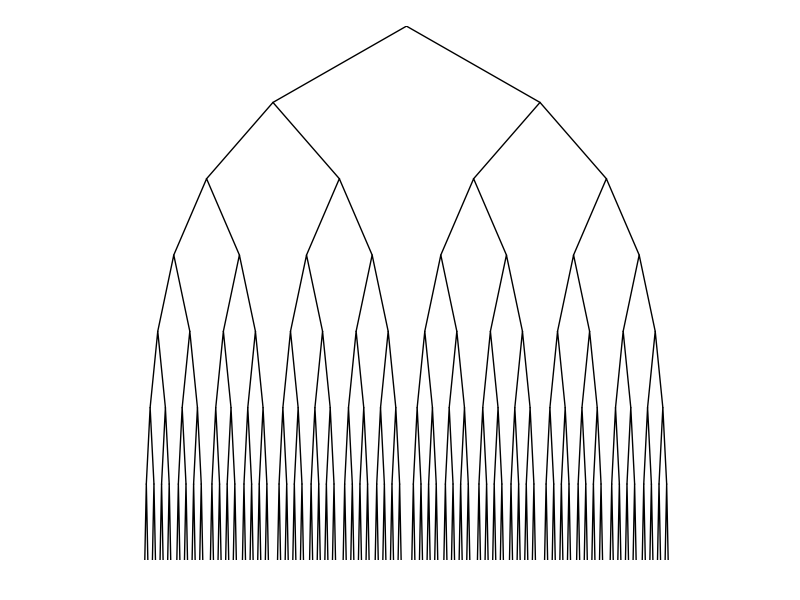
In this lab Professor Fuentes gave us four patterns that he wanted us to replicate by using recursive methods.He also gave us two more patterns however, these were already solved he had given us the building blocks to use when completing the lab.

The first pattern we had to duplicate was a square that had a square in all four corners, so the function was a recursive function that called itself four times with new coordinates in relation to the first square. My initial approach I failed because I was hardcoding, how I saw the problem and how the shape was plotted in relation to the y axis I thought that okay it had to be a fraction higher or lower from all sides when in reality you must look at the center point of the original square to form four more. In explanation you know the radius and the center coordinate because you are passing them when calling the function, use those points to either add or subtract using the coordinates as a starting point. To complete this function I broke down the problem into two functions one function would just draw the square given the center coordinates it would either add or subtract the x and the y of the center coordinates and it would make four recursive calls.

The second pattern was a tunnel pattern this was the easiest of the all the functions, my initial approach was successful, when looking at this problem I noticed that much like the function that was given to us the center point did not move just like this one so all I did was move the x on the center coordinate over all the way to the left using the radius as a guide.

The third pattern was a function that draws a tree of some sort, my initial approach I hard coded using a decimal to treat it as a percentage of what the x and y coordinate would move, after many unsuccessful attempts, using the change of x and y in many ways such as using the coordinates as a way to divide to move the point I became stuck it was brought to my attention that what if the change of y was a constant that was always dividing by a decreasing number such as the n that was passed as a parameter, and the change of x was the length of the whole grid divided by four or .25 of the whole grid, and every time it is called recursively it is divided by 2. In this function there were two recursive calls one for each branch one of the first thoughts I had about drawing the trees was what if I drew the whole left side of the tree then in the recursive call on the way out draw the right side, however due to the limits of python it was very difficult.

The fourth pattern that we were told to follow was a function that drew a circle with 5 other circles inside, this was very difficult for me I understood that each circle was drawn one at a time filled with as many circles as was called for, I also understood how many were supposed to fit in the circle, I was dumfounded by the logic and how the measurements worked, for this problem to be solved I also created two different functions that would work together one would just draw the circle given the center coordinate making a messy part of the problem invisible, now the other function was a little less messy I starting thinking okay if there are three circles on the x axis and three on the y axis why not move the center in relation to the center coordinate, and for the radius divide it by 3, since there are 3 in the line.



I learned that you must look at the problem not just from a pseudocode or a code point of view but using fractions and how things relate to solve your problem may be very beneficial.

#Adrian Monreal 80570881#  
#Lab 1#  
#DUE DATE 02/08/2019#  
  
import math  
import numpy as np  
import matplotlib.pyplot as plt  
  
def draw\_squares(ax, n, p, w):  
 if n>0:  
 #has to call again until there are no more sqaures in the corner  
 i1 = [1, 2, 3, 0, 1]  
 q = p\*w + p[i1]\*(1-w)  
 ax.plot(p[:,0],p[:,1],color='k')  
 draw\_squares(ax,n-1,q,w)  
  
plt.close("all")  
orig\_size = 800  
p = np.array([[0,0],[0,orig\_size],[orig\_size,orig\_size],[orig\_size,0],[0,0]])  
fig, ax = plt.subplots()  
draw\_squares(ax, 15, p, .8)  
ax.set\_aspect(1.0)  
ax.axis('off')  
plt.show()  
fig.savefig('squares.png')  
  
  
  
def circle(center, rad):  
 n = int(4 \* rad \* math.pi)  
 t = np.linspace(0, 6.3, n)  
 x = center[0] + rad \* np.sin(t)  
 y = center[1] + rad \* np.cos(t)  
 return x, y  
  
  
def draw\_circles(ax, n, center, radius, w):  
 if n > 0:  
 x, y = circle(center, radius)  
 ax.plot(x, y, color='k')  
 draw\_circles(ax, n - 1, center, radius \* w, w)  
  
  
plt.close("all")  
fig, ax = plt.subplots()  
draw\_circles(ax, 100, [100, 0], 100, .9)  
ax.set\_aspect(1.0)  
ax.axis('off')  
plt.show()  
fig.savefig('circles.png')  
  
def draw\_corner\_squares(center, radius):  
#this function is a helper function for corner squares, this draws the square  
#so that when corner\_squares calls it, it will draw a square everytime corner squares does a recursive call  
#it creates an array of points in relation to the center point that are essentially the corners  
  
 corners = np.array([[center[0]-radius, center[1]+ radius],[center[0]+radius, center[1]+radius],  
 [center[0]+radius, center[1]-radius],[center[0]-radius,center[1]-radius],  
 [center[0]-radius,center[1]+radius]])  
 ax.plot(corners[:,0],corners[:,1],color='k')  
 #draw\_corner\_squares(center,radius)  
  
def corner\_squares(ax, n,center,radius ):  
 #this function has 4 recursive calls each of which uses the corner in relation to the center coordinate  
 #there names are as follows  
 #the radius shrinks by 1/2 because the amount space is cut in half  
 if n>0:  
 topL= [center[0]-radius,center[1]+radius]  
 topR= [center[0]+radius,center[1]+radius]  
 botL= [center[0]-radius,center[1]-radius]  
 botR= [center[0]+radius,center[1]-radius]  
 draw\_corner\_squares(center,radius)  
 corner\_squares(ax,n-1,topL,radius/2)  
 corner\_squares(ax,n-1,topR,radius/2)  
 corner\_squares(ax,n-1,botL,radius/2)  
 corner\_squares(ax,n-1,botR,radius/2)  
  
plt.close("all")  
  
  
  
fig, ax = plt.subplots()  
corner\_squares(ax, 3,[0,0],50)  
ax.set\_aspect(1.0)  
ax.axis('off')  
plt.show()  
fig.savefig('corner\_squares.png')  
  
  
  
def draw\_tunnel(ax, n, center, radius,w):  
 #the logic behind this one was if you move the center of the circle to the one of the sides of the  
 #circle it will create a tunnel effect, so i basically added the center to the radius  
 #to have the circle create a circle around the edge  
  
  
 if n > 0:  
 x, y = circle(center, radius)  
 x = x + radius  
 ax.plot(x, y, color='k')  
 draw\_tunnel(ax, n - 1, center, radius \* w,w)  
  
plt.close("all")  
fig, ax = plt.subplots()  
draw\_tunnel(ax, 100, [100, 0], 100,.9)  
ax.set\_aspect(1.0)  
ax.axis('off')  
plt.show()  
fig.savefig('tunnel.png')  
  
def draw\_trees(ax, n, x,y,delta\_y,delta\_x):  
  
 #in this fuinction it creates an array of coordinates to plot the tree  
 #the first coordinate is the left most branch the middle is the top point and  
 #the last is the vertex of the right branch  
 #the 2 recursive calls are for the 2 branches one to the right of the point  
 # and the other to the left of the point but both with y\_delta which is the change in Y  
  
 if n>0:  
  
 tree\_array = np.array([[x-delta\_x, y-delta\_y],[x, y], [x+delta\_x, y-delta\_y]])  
 ax.plot(tree\_array[:, 0], tree\_array[:, 1], color='k')  
 draw\_trees(ax,n-1,x-delta\_x,y-delta\_y,delta\_y,delta\_x/2)  
 draw\_trees(ax,n-1,x+ delta\_x,y-delta\_y,delta\_y,delta\_x/2)  
  
plt.close("all")  
fig, ax = plt.subplots()  
num\_trees= 7  
deltaY= 100  
deltaX= (num\_trees\*deltaY)/4  
draw\_trees(ax,num\_trees,0,0,deltaY,deltaX)  
  
ax.set\_aspect(1.0)  
ax.axis('off')  
plt.show()  
fig.savefig('trees.png')  
  
def draw\_circle\_inside(center, rad):  
  
 #this function is similar to the draw squares function this function helps split the problem  
 #into 2 functions this one just creates the circle  
  
 n = int(4 \* rad \* math.pi)  
 t = np.linspace(0, 6.3, n)  
 x = center[0] + rad \* np.sin(t)  
 y = center[1] + rad \* np.cos(t)  
 ax.plot(x, y, color='k')  
  
def circle\_inside(ax,n,center,radius):  
  
 #this is the second half of the problem this actually gets the points of the  
 #center of the circle so it can create the radius in relationto the point  
 # the radius is divided by 3 because of how many circles you fit  
 # along the axis either x or y its always 3  
 # it creates 4 different arrays of the circles to correspond to the center of where  
 #the circle would be in correlation of the first circle center point  
  
 if n > 0:  
 #five calls x- 2/3 radius y-+ 2/3 or 1/3 of radius )  
 #radius is shrunk by 1/3  
 leftCircle = [center[0]-(.6666\*radius),center[1]]  
 rightCircle = [center[0]+(.6666\*radius),center[1]]  
 topCircle = [center[0],center[1]+(.6666\*radius)]  
 bottomCircle = [center[0],center[1]-(.6666\*radius)]  
  
 draw\_circle\_inside(center, radius)  
 circle\_inside(ax, n - 1, center, radius/3)  
 circle\_inside(ax, n - 1,leftCircle , radius/3 )  
 circle\_inside(ax, n - 1, rightCircle, radius/3)  
 circle\_inside(ax, n - 1, topCircle, radius/3 )  
 circle\_inside(ax, n - 1, bottomCircle, radius/3 )  
  
  
plt.close("all")  
fig, ax = plt.subplots()  
circle\_inside(ax, 4, [100, 0], 100)  
ax.set\_aspect(1.0)  
ax.axis('off')  
plt.show()  
fig.savefig('circles\_inside.png')

I Adrian Monreal here by sign that this code is mine I did not copy or steal from anyone.