Introduction:

We are given two source files where we can draw figures recursively. However, they are supposed to be modified to create different kinds of fractals by replicating the images that were provided.

Proposed solution design and implementation:

Some of the pictures provided to replicate were completely different from what the source code could really do. Therefore, understanding how the images given were generated and doing reverse engineering on the source code was essential for approaching a solution. Decomposing the image by its figures’ positions was my first step. Then understanding the pattern on each image was easier because of the previous step. Secondly, on a separate sheet of paper the images were redrawn to notice in which directions would the pattern repeat.

For example, when the first image (square recursion) was redrawn, a pattern was noticed on the direction of 4 axis-northeast, southeast, southwest, and northwest. These types of repetitions on a certain direction would help determine how many recursive calls would be needed and what parameters to insert when creating the method. Same applied to the third and fourth recursive problems, but the second problem was a little different (circle recursion over the same point). Redrawing the figure made everything simpler to understand how the circles would transform inwards. The circle would transform based on an origin point so only having a recursive call making the changes of the transformation was evident. All there was left to do was to put a weight value below 1 inside the radius change parameter.

Experimental results:

1st recursion problem:

Experiment: At the beginning of creating the method, I would mess around with the values from the recursive calls to figure out how to displace a figure on each axis.

Result: Several squares displacing in multiple directions, weird rotations, and decrementing in size.

Experiment: During the process of the method, the equation modifying the “p” array’s values gave limited transformation, so I created a new array to be able to access its 4 vertices. Tried hardcoding different values into each element and then used the same value for all of them.

Result: Would move all vertices randomly, then moved figure completely.

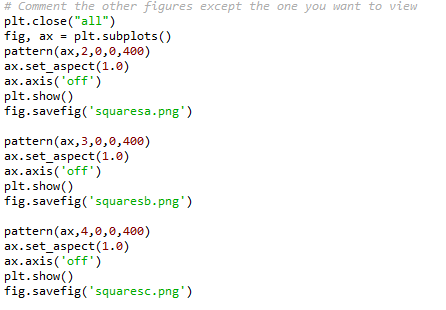
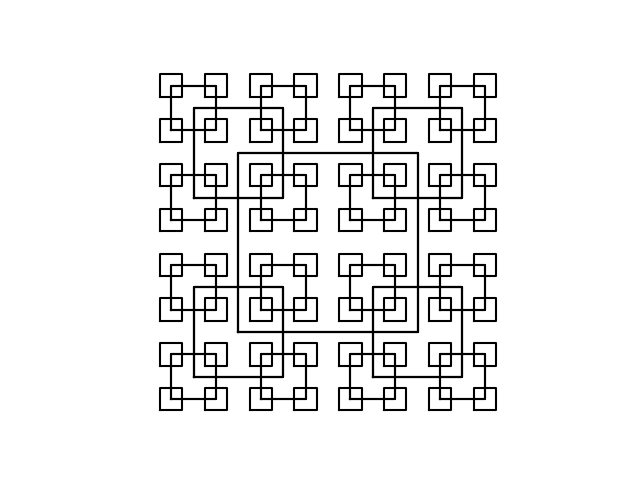
Experiment: Found the center of each vertex and implemented them in each array element.

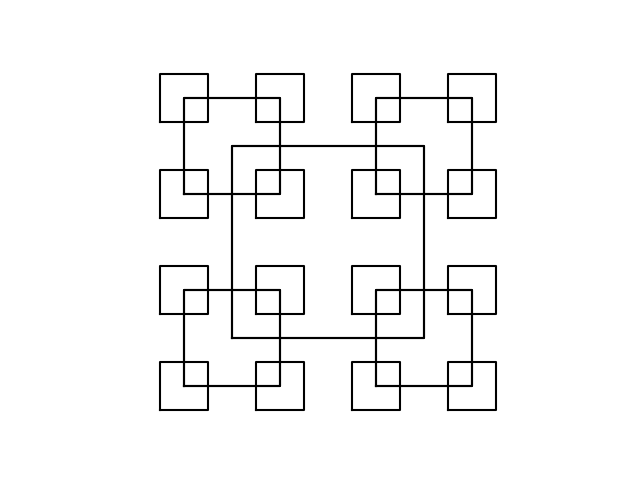
Result: No changes, although the figure would shift to a different position.

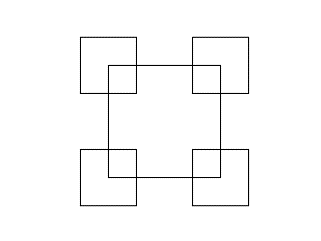
Experiment: Made recursive call with multiple iterations and changed size of next square by ½.

Result: Created exact representation of the figures provided for problem 1.

Input:



Outputs:

Running times (for different depths of the same recursive figure):

1st = 0.16150975227355957s

2nd = 0.20910882949829102s

3rd = 0.3789980411529541s

2nd recursion problem:

Experiment: Used the provided method and modified its inputs.

Result: Increased number of inner circles, increased size of all circles, and weight.

Experiment: Changed operations for ‘x’ inside of the “circles” method. Switched ‘+’ signs to ‘-‘and ‘\*’ to ‘/’.

Result: No major changes other than shifting the complete figure to random positions.

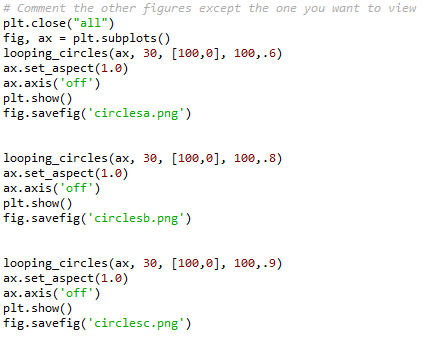
Experiment: Added a constant to the equation of ‘x’ variable.

Result: Shifted all the circles towards the left by a minimal amount

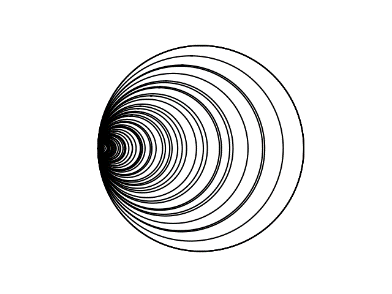
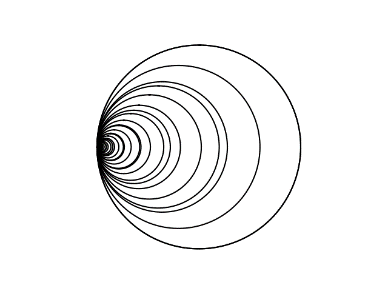
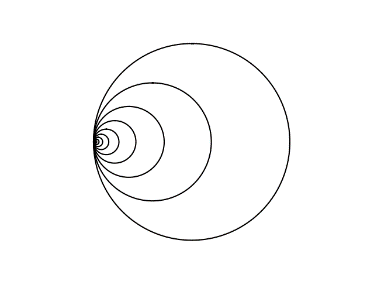
Experiment: Added by the length of the radius to the ‘x’ equation

Result: Set an anchor point to the left most side of the circle.

Input:



Outputs:



Running times:

1st = 0.33664679527282715s

2nd = 0.20442843437194824s

3rd = 0.25952887535095215s

3rd recursion problem:

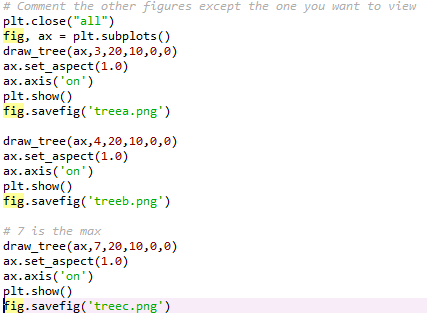
Experiment: Transferred handwritten pseudo code to code editor. Added only one recursive call dividing the x-axis displacement by half and subtracting its origin point by the x displacement of the next circle origin.

Result: Only left branch of the image was produced.

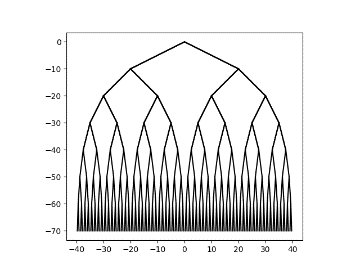
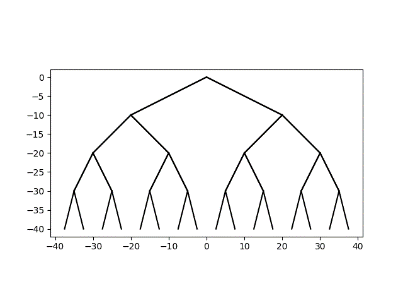
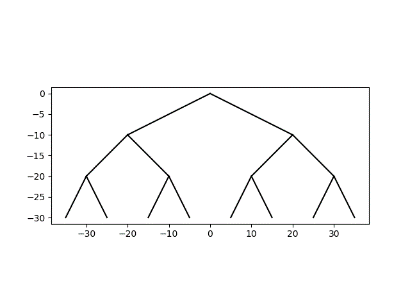
Experiment: Added second recursive call by adding x-axis displacement divided by 2 and added its origin point by the x displacement of the next circle origin.

Result: Both branches displayed like the image provided.

Input:



Output:



Running times:

1st = 0.17054390907287598

2nd = 0.3460698127746582s

3rd = 0.8221855163574219s

4th recursion problem:

Experiment: Created method with center coordinates of the circle, and with shifts to change the positions of circles. Had one recursive method with y coordinates being subtracted from the shift.

Result: Made a series of circles going downwards.

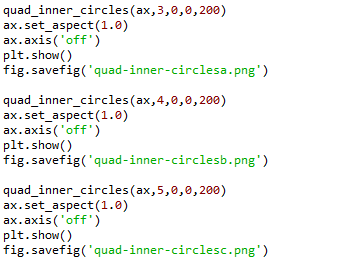
Experiment: Added 2nd, 3rd and 4th methods. Each had different directions to display the figures.

Result: Created circles around the circumference of each circle, but was missing middle circle for each circle created recursively

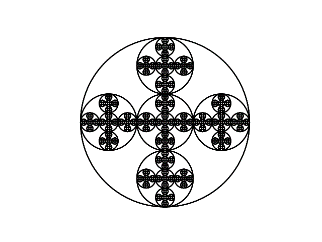
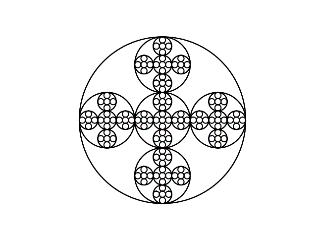
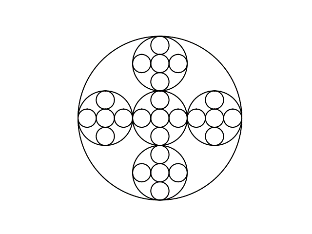
Experiment: Added a 5th method with middle coordinates to display at the center of every circle created recursively

Result: Recreated the image provided from the problem.

Input:



Output:



Running times:

1st = 0.2263932228088379s

2nd = 0.4358673095703125s

3rd = 2.423243284225464

Conclusion:

Analyzing the problem in depth helps destroy the tunnel vision of just wanting to figure out a solution when stuck. Breaking down the problem into parts to solve helps break from the “stuck” moments, making coders more efficient when figuring out a solution.

Appendix

Draw\_circles.py

*# Course: 2302-001*

*# Author: Esteban Retana*

*# Assignment: Create figures or fractals based on the knowledge of recursion*

*# Instructor: Olac Fuentes*

*# TA: Eduardo Lara*

*# Date of last modification:2/8/19*

*# Purpose: Practice creating fractals with recursion*

import matplotlib.pyplot as plt

import numpy as np

import math

import time

def circle(center,rad):

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

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

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

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

return x,y

def circle2(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 looping\_circles(ax,n,center,radius,w):

if n>0:

x,y = circle(center,radius)

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

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

def quad\_inner\_circles(ax,n,middle\_x,middle\_y,r):

if n>0:

middle = [middle\_x,middle\_y]

x,y = circle2(middle,r)

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

shift = r \* (2/3)

left = middle\_x + shift

right = middle\_x - shift

up = middle\_y + shift

down = middle\_y - shift

r /= 3

quad\_inner\_circles(ax,n-1,left,middle\_y,r)

quad\_inner\_circles(ax,n-1,right,middle\_y,r)

quad\_inner\_circles(ax,n-1,middle\_x,up,r)

quad\_inner\_circles(ax,n-1,middle\_x,down,r)

quad\_inner\_circles(ax,n-1,middle\_x,middle\_y,r)

*# Comment the other figures except the one you want to view*

start\_time = time.time()

plt.close("all")

fig, ax = plt.subplots()

*#looping\_circles(ax, 30, [100,0], 100,.6)*

*#ax.set\_aspect(1.0)*

*#ax.axis('off')*

*#plt.show()*

*#fig.savefig('circlesa.png')*

*#looping\_circles(ax, 30, [100,0], 100,.8)*

*#ax.set\_aspect(1.0)*

*#ax.axis('off')*

*#plt.show()*

*#fig.savefig('circlesb.png')*

*#looping\_circles(ax, 30, [100,0], 100,.9)*

*#ax.set\_aspect(1.0)*

*#ax.axis('off')*

*#plt.show()*

*#fig.savefig('circlesc.png')*

*#quad\_inner\_circles(ax,3,0,0,200)*

*#ax.set\_aspect(1.0)*

*#ax.axis('off')*

*#plt.show()*

*#fig.savefig('quad-inner-circlesa.png')*

*#quad\_inner\_circles(ax,4,0,0,200)*

*#ax.set\_aspect(1.0)*

*#ax.axis('off')*

*#plt.show()*

*#fig.savefig('quad-inner-circlesb.png')*

quad\_inner\_circles(ax,5,0,0,200)

ax.set\_aspect(1.0)

ax.axis('off')

plt.show()

fig.savefig('quad-inner-circlesc.png')

end\_time = time.time() - start\_time

print(end\_time)

Draw\_squares.py

*# Course: 2302-001*

*# Author: Esteban Retana*

*# Assignment: Create figures or fractals based on the knowledge of recursion*

*# Instructor: Olac Fuentes*

*# TA: Eduardo Lara*

*# Date of last modification:2/8/19*

*# Purpose: Practice creating fractals with recursion*

import numpy as np

import matplotlib.pyplot as plt

import time

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

if n>0:

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)

*# Creates square pattenr based on each vertex as a median point*

def pattern(ax,n,x,y,size):

if n>0:

right\_x = x+size

left\_x = x-size

lower\_y = y-size

upper\_y = y+size

*# Creates new square*

q = np.array([[left\_x,lower\_y],[left\_x,upper\_y],[right\_x,upper\_y],[right\_x,lower\_y],[left\_x,lower\_y]])

ax.plot(q[:,0],q[:,1],color='k')

*# bottom left*

pattern(ax,n-1,left\_x,lower\_y,size/2)

*# upper left*

pattern(ax,n-1,left\_x,upper\_y,size/2)

*# upper right*

pattern(ax,n-1,right\_x,upper\_y,size/2)

*# bottom right*

pattern(ax,n-1,right\_x,lower\_y,size/2)

*# Comment the other figures except the one you want to view*

plt.close("all")

fig, ax = plt.subplots()

pattern(ax,2,0,0,400)

ax.set\_aspect(1.0)

ax.axis('off')

plt.show()

fig.savefig('squaresa.png')

pattern(ax,3,0,0,400)

ax.set\_aspect(1.0)

ax.axis('off')

plt.show()

fig.savefig('squaresb.png')

pattern(ax,4,0,0,400)

ax.set\_aspect(1.0)

ax.axis('off')

plt.show()

fig.savefig('squaresc.png')

draw\_tree.py

*# Course: 2302-001*

*# Author: Esteban Retana*

*# Assignment: Create figures or fractals based on the knowledge of recursion*

*# Instructor: Olac Fuentes*

*# TA: Eduardo Lara*

*# Date of last modification:2/8/19*

*# Purpose: Practice recursion by creating fractals*

import numpy as np

import matplotlib.pyplot as plt

import time

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

if n>0:

q = np.array([[x,y],[x-x\_shift,y-y\_shift]])

q1 = np.array([[x,y],[x+x\_shift,y-y\_shift]])

ax.plot(q[:,0],q[:,1],color='k')

ax.plot(q1[:,0],q1[:,1],color='k')

draw\_tree(ax,n-1,x\_shift/2,y\_shift,x-x\_shift,y-y\_shift)

draw\_tree(ax,n-1,x\_shift/2,y\_shift,x+x\_shift,y-y\_shift)

*# Comment the other figures except the one you want to view*

start\_time = time.time()

plt.close("all")

fig, ax = plt.subplots()

draw\_tree(ax,3,20,10,0,0)

ax.set\_aspect(1.0)

ax.axis('on')

plt.show()

fig.savefig('treea.png')

*#draw\_tree(ax,4,20,10,0,0)*

*#ax.set\_aspect(1.0)*

*#ax.axis('on')*

*#plt.show()*

*#fig.savefig('treeb.png')*

*# 7 is the max*

*#draw\_tree(ax,7,20,10,0,0)*

*#ax.set\_aspect(1.0)*

*#ax.axis('on')*

*#plt.show()*

*#fig.savefig('treec.png')*

end\_time = time.time() - start\_time

print(end\_time)