David Amparan

February 10th, 2019

Lab #1

Computer Sciene 2302

Dr. Fuentes

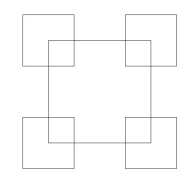
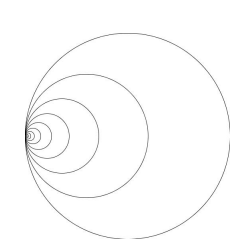
**Introduction**

With a re-run of recursion, the first lab tasked us with completing recursive methods that drew certain geometric shapes. In total we had 4 figures to draw which included a square with a new square in each corner, circles that weigh to the left most side, a binary tree, and a circle with 5 other circles within itself. The code for creating the squares and circles was provided by Dr. Fuentes and was easy to understand (within the wide scope) and to modify to complete the task. In general, we passed an axis to draw onto the methods, a list/array of coordinates, a weight (to calculate average), and the amount of times ran (n). The given points were then plotted, modified with the average and then recursively passed to draw until n is 0.

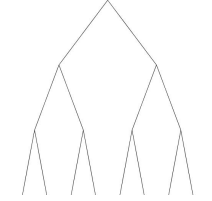
Therefore I have 4 solutions since 4 figures are needed

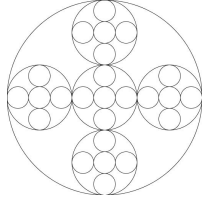
* Square with square within each corner
  + My solution for this figure was to take the center of each coordinate and use that center to calculate the coordinates of the square assigned with that center. Therefore, giving us four recursive calls in its method
* Circle weighed to the left
  + The simple solution was to multiply the center by a weight of 50% and increase the amount of n’s to produce more circles
* Binary Tree
  + Create 3 coordinates to make a triangle without the bottom side and recursively send each daughter cell to add 2 more cells to that cell. This will go on until the desired length is attained, since n will be the size
* Circle with 5 within
  + My solution was after drawing the original circle, we have five recursive calls but the radius for all will be divided by 3 and multiply the radius by the weight. The radius is divided by 3 since we want to fit 3 circles horizontally and vertically.

**Solution Proposals and Design**

Like I mentioned above, my code was compromised of 4 main methods (4 shapes requested) and a few sub methods which were implemented only on certain methods. Beginning with the squares with consecutive corner squares algorithm was broken down into 2 parts, pointsSquares and the\_squares methods. The\_squares method is our recursive method that runs an *n* amount of times and only takes the center coordinates (array) and the length desired. Before we make any recursive calls we call pointsSquares so we can receive these points and plot them. PointsSquare treats each square independently since it only needs the length and center of the square that will be drawn. I chose this method since it was the easiest way to see that the squares in the corners share an equality of its center and the corner from which it is called from, be it top right, bottom left, etc. After calculating the points, we return them to the the\_sqaure method, plot them, and finally make 4 recursive calls (4 corners).

Next up was the circle with a left shift. This method utilized both methods given to us by Dr. Fuentes which includes the draw\_circles and circle (which I renamed the\_hole). By calling the circle method within the\_hole we get the set of points to plot by calculating these with the formula for circumference. After this was computed the change need for shift the consecutive circles was to multiply the X coordinate of the center (center [0]) by the weight variable. Depending on the percentage given we could get the other circles how we were asked (at the left most x point). My choice of design came easy since no major modifications were need nor was a new method needed to complete the objective.

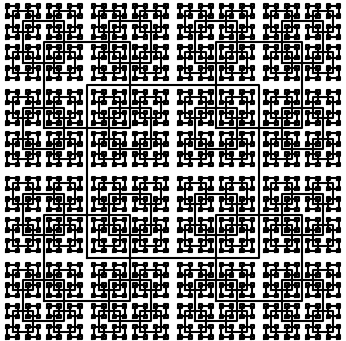
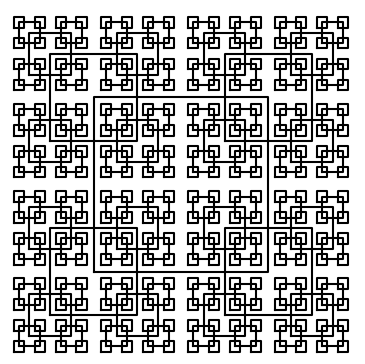
Following the circle with the left shift is the binary tree. While thinking about this shape its obvious that it has a triangular shape without the bottom edge and that the length from the daughter cells is half in the next iteration. Plus, with each iteration the daughter cells themselves become the new roots from which more cells will sprout. Now the value that n can represent is the height of the tree, the larger the value of n the deeper the tree shall go. With these ideals in mind, we only need one method with parameters of the root (coordinates), length, and n. Simply put we take in the root and coordinates, plot them, divide the length by two, and divide n by 2 as well (n can also be subtracted by 1 it makes no major difference).

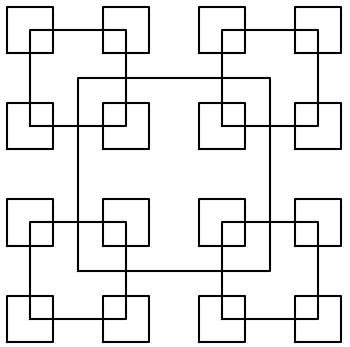
Finally, the circles within a circle figure can be perceived as the toughest out of all four. Which to be fair is the one I took the longest to solve. My first approach was to simply divide the radius for the consecutive circles (recursive calls) and mod the center values to attain either the horizontal circles or the vertical. This approach worked until I increased the size of n, by increasing the size caused my circles to act randomly (so I thought). The main issue with my approach was that increasing the number of recursive calls caused the center for each circle to be modified over and over leading to extreme expansions beyond the original circle. My next approach came from a thread by one of the T. A’s through slack. All we needed to compute (since I used the same code Dr. Funtes provided) was the ratio of the radius divided by 3 times the weight (fixed to 1.95) minus the center X or Y within each recursive call.

Now when the program is running, it will display 3 results for each figure with a change in its amounts of recursive calls or n. Therefore, a total of 12 figures will be opened or displayed (depending on the settings) to satisfy the request to display all in the first run.

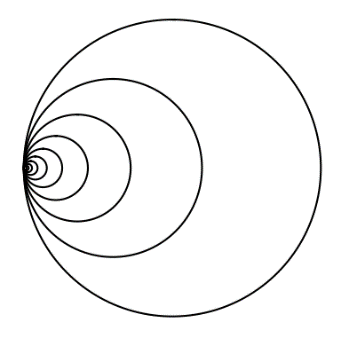
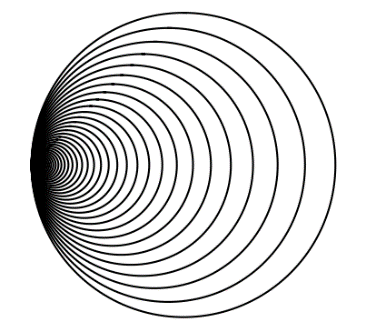
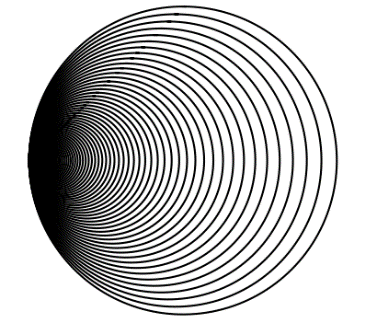
**Experimental Results**

Testing the durability and accuracy for my program was relatively simple since n was the premise of the assignment (recursive calls and determine running time). Therefore, by increasing the value of n we can gauge the durability and possible running time of each method. Now I cannot increase n by some crazy figure because of the speed of my desktop as well. Most of the tests performed can be done on a moderate system. My system runs with 4GBs of RAM and a Intel Pentium CPU running at 3.00 GHz

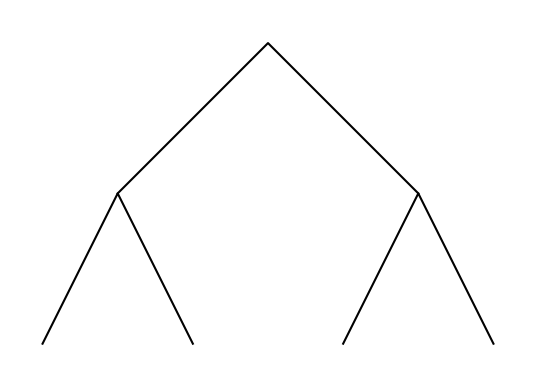
* Testing figure 1: We will let n equal to 2 then 4 and finally 6. This then causes the squares figure to recurse 2, 4, and 6 times respectively. By conducting these tests, we can see that the growth rate as n increases at a high rate which can indicate an exponential T(n)
  + In ascending order 2, 4, 6

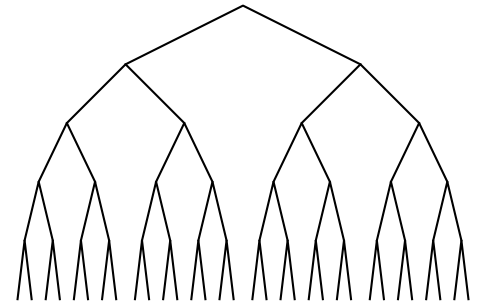


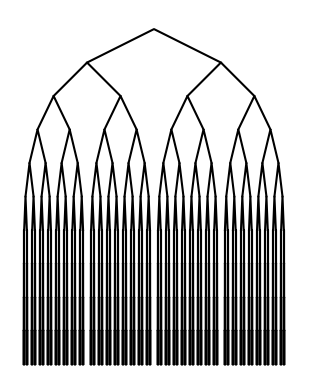
* Testing figure 2: Once again we will be increasing n since it is responsible for the growth rate. For figure 2, n will be equal to 10, 50, 100 and the output as well as running times will be given below. As we can see by increasing n, we get more circles to the point it looks like a hypnotize tool. Also, the growth is very low as we can see since the running time grows slightly each time.





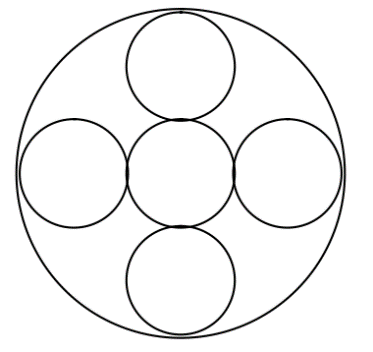
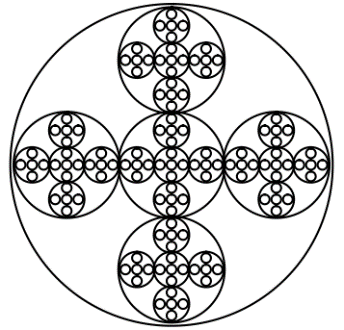
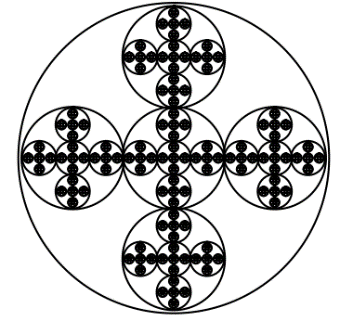
* Testing figure 3: For the binary tree n, is responsible for the depth of the tree therefore we will increase n and achieve a deeper tree with each increase on n. For this test n shall be equal to 2, 5, 10. Two things can be observed as n grows and that is the size of the tree and the limitation (for my PC) of the number of n’s I can input. As n grows the ability to see each root and daughter cells becomes increasingly more difficult.





* Testing figure 4: Finally, for the circles within we once again only mod the size of n. By increasing n, we should see more and more circles within each that is drawn and a spike

with the time taken since a total of five recursive calls are made. So, for our test n will be 2, 4, 6 and the results will be displayed underneath. After running the tests, we see that the growth rate for n greater than six takes too long to display and times out the figure causing the image to crash. The growth rate is also big since it jumps exponentially much like figure 2.



**Conclusions**

In conclusion this lab was useful in various ways such as showing the pros of python over java, a touch up on the advantages of recursion, and finally the importance of optimizing an algorithm to a manageable growth rate. Python is a smoother language to code in since it does not have as much requirements and can allow one to code at a faster rate when you know what you’re doing. Furthermore, recursion allows us to partition problems into sub problems that can be passed on and solved by a base case or multiple base cases. Lastly, optimizing algorithms is an important and essential since it can drastically help reduce the growth rate of your code. This can save time and memory from the system and help you optimize the rest of the code.

**Appendix**

**Code**

|  |
| --- |
| Author: David Amparan |
|  | CS 2302 |
|  | Last Date Modified: February 8, 2019 |
|  | Instructor: O.Fuentes |
|  | Assingment: Lab 1 |
|  | TA: Anindita Nath & Maliheh Zaragan |
|  | Purpose: This code will serve as practice for recursion as well as an introduction to the |
|  | benefits of python, various geometric shapes will be drawn and each method will be ran 3 times creating 3 different outputs |
|  | to show the full functionality of them |
|  | """ |
|  | import numpy as np |
|  | import matplotlib.pyplot as graph |
|  | import math |
|  |  |
|  | """ |
|  | Method Name: draw circle | Parameters: ax, n, center, radius, w |
|  | Functionality: This is the method that Professor Fuentes provided except with five recursive calls to |
|  | succesfully draw the last difure requested. This is done by attaining the ratio of the radius divided 3 times the correct |
|  | weight |
|  | """ |
|  | def draw\_circles(ax,n,center,radius,w): |
|  | if n>0: |
|  | x,y = circle(center,radius) |
|  | ax.plot(x,y,color='k') |
|  | #first call for middle circle |
|  | draw\_circles(ax,n-1,center,radius//3,w) |
|  | #x is the ratio of the radius divided by 3 since you want three circles |
|  | x = radius/3 |
|  | #depending on which you want you either add the ratio or subtract |
|  | draw\_circles(ax, n-1,[(center[0]-(x\*w)), center[1]], radius//3, w ) |
|  | draw\_circles(ax, n-1, [(center[0]+(x\*w)), center[1]], radius//3, w) |
|  | draw\_circles(ax, n-1, [center[0], (center[1]-(x\*w))], radius//3, w) |
|  | draw\_circles(ax, n-1, [center[0], (center[1] + (x\*w))], radius//3, w) |
|  |  |
|  |  |
|  | """ |
|  | Method Name: attainKids| Parameters (root, height, length) |
|  | Functionality: will pass the root and produce the daughter cells of these, |
|  | The daughter roots will be based on half of the length and 50% of the remaning height |
|  | """ |
|  | def attainKids(root, height, length): |
|  | #here we will give the coordinates for the left and right children |
|  | left, right = np.zeros((2,2)) |
|  | left[0] = root[0]-(length//2) |
|  | right[0] = root[0]+(length//2) #Here all we do is take half of the given measurments and either add or sub |
|  | left[1] = root[1] - (height\*.50) |
|  | right[1] = root[1]-(height\*.50) |
|  | return left, root, right |
|  |  |
|  | """ |
|  | Method Name: binary\_tree | Parameters: ax, height, root, length, toMulti |
|  | Functionality: Binary tree will take in the ax to draw onto as well as the height, root, and length |
|  | to impliment on attainKids and plot the returned points |
|  | """ |
|  | def binary\_tree(ax, height, root, length, toMulti): |
|  | if height>0: |
|  | #i attain the coordinates for the daughter cells then put them in arrau |
|  | left\_right = attainKids(root, toMulti, length) |
|  | coor = np.zeros((3,2)) |
|  | #convert to array to avoid any error when slicing to plot |
|  | for i in range(3): |
|  | coor[i] = left\_right[i] |
|  |  |
|  | ax.plot(coor[:,0], coor[:,1], color = 'k') |
|  | ax.plot() |
|  | #recursive call |
|  | binary\_tree(ax, height//2, coor[0], length//2, toMulti) |
|  | binary\_tree(ax, height//2, coor[2], length//2, toMulti) |
|  |  |
|  |  |
|  | """ |
|  | Method Name: the\_hole | Parameters:ax,n,center,radius,width| |
|  | Functionality: This method will take in the paramets which will be used for the |
|  | composition of a circle figure, then will be ploted. THIS METHOD ALSO INTEGRATES PARTS |
|  | OF THE SHARED CODE BY PROFESSOR FUENTES |
|  | """ |
|  | def the\_hole(ax,n,center,radius, weight): |
|  | if n>0: |
|  | x,y = circle(center,radius) |
|  | ax.plot(x,y,color='k') |
|  | #here we modify the center as we begin to push left |
|  | center[0] = center[0]\*weight |
|  | #recursive call |
|  | the\_hole(ax, n-1, center, radius\*weight, weight) |
|  | """ |
|  | Method Name: circle | Parameters: center, radius| Returns an xy array |
|  | Functionality: circle will calculate the circles circumference |
|  | and create a set of points for that circumeference |
|  | """ |
|  | 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 |
|  |  |
|  |  |
|  | """ |
|  | Method Name: the\_squares | Parameters: axis, times it will iterate, the origin, length of side |
|  | Functionality: The method will draw the squares, these will be ploted |
|  | by the pairs from the 2D array. The radius parameter also indicates the size |
|  | for the consecutive corner squares which will also use a center |
|  | """ |
|  | def the\_squares(ax,n,origin,length): |
|  | if n>=0: |
|  | #call method to attain coordinates an begin plotting |
|  | p = pointsSquare(origin,length) |
|  | #by adding the for loop we allow the coordinates to become an array not a tuple |
|  | coor = np.zeros((5,2)) |
|  | for i in range(len(p)): |
|  | coor[i] = p[i] |
|  |  |
|  | #iterator = [1,2,3,0,1] |
|  | ax.plot(coor[:,0],coor[:,1], color='k') |
|  | ax.plot() |
|  | #RECURSIVE CALL |
|  | the\_squares(ax,n-1, coor[0], length//2) |
|  | the\_squares(ax,n-1, coor[1], length//2) |
|  | the\_squares(ax,n-1, coor[2], length//2) |
|  | the\_squares(ax,n-1, coor[3], length//2) |
|  | """" |
|  | Method Name: pointsSquare | Parameters: center, length | Return Type: Pairs |
|  | Center squares will return the points for the corresponding origin. It will use the length |
|  | to give the correct points to attain a symmetrical square |
|  | """ |
|  | def pointsSquare(center,length): |
|  | #here we reduce the size to get even points |
|  | radius = length//2 |
|  | #the top left and bottom right coordinates have to modified carefully since they involve |
|  | #the oppostie operation for each element in the center array |
|  |  |
|  | x=center[0] #x and y must be created to help with the pass by reference issue, thus we create new array |
|  | y=center[1] |
|  | topLeft = np.array([x-radius,y+radius]) |
|  | bottomRight = np.array([x+radius, y-radius]) |
|  |  |
|  | #once we return we have the coordiantes for the correspinding center |
|  | return center-radius, topLeft, center+radius, bottomRight, center-radius |
|  |  |
|  | #adding the for loop allows us to repeat the demonstration three times |
|  | squares = 1 |
|  | for i in range(3): |
|  | length = 100 |
|  | origin = np.array([0,0]) |
|  | #graph.close("all") |
|  | fig, ax = graph.subplots() |
|  | the\_squares(ax, squares, origin, length) |
|  | ax.set\_aspect(1.0) |
|  | ax.axis("off") |
|  | graph.show() |
|  | #fig.savefig('squares.png') |
|  | squares+=1 |
|  |  |
|  | #loop for the circles |
|  | totalC = [10, 50, 100] |
|  | radi = [.6, .9, .95] |
|  |  |
|  | circles = 1 |
|  | for i in range(3): |
|  | #graph.close("all") |
|  | fig, ax = graph.subplots() |
|  | the\_hole(ax, totalC[i], [100,0], 100, radi[i]) |
|  | ax.set\_aspect(1.0) |
|  | #ax.axis('off') |
|  | graph.show() |
|  | #fig.savefig('circles.png') |
|  |  |
|  | #loop for the binary tree |
|  | #variables to change values |
|  | heights = [4, 10, 40] |
|  | for i in range(3): |
|  | #graph.close("all") |
|  | fig, ax = graph.subplots() |
|  | binary\_tree(ax, heights[i], [0,100], 100, 100) |
|  | ax.set\_aspect(1.0) |
|  | #ax.axis("off") |
|  | graph.show() |
|  | #fig.savefig('circles.png') |
|  | #for loop for the draw circles |
|  | tripCircles = [3,4,5] |
|  | for i in range(3): |
|  | #graph.close("all") |
|  | fig, ax = graph.subplots() |
|  | draw\_circles(ax, tripCircles[i], [100,100], 100, 1.95) |
|  | ax.set\_aspect(1.0) |
|  | ax.axis("off") |
|  | graph.show() |
|  | #fig.savefig('tripCircles.png') |