CS 2302 Data Structures

Lab Report #1

Due: Sebtember 16th, 2020
Professor: Olac Fuentes
TA: Anindita Nath
Student: Manuel Gutierrez Jr

Introduction

For this lab, we were asked to work with recursion and Stacks and code methods that use this data structure to print out the specific geometric shapes. For this lab, it is key to remember the basics of recursion activation records and how Stacks work since both are required for half of the lab. The main objective of this lab is to get a deeper understanding of activation records and Stacks and being able to implement a recursive function using Stacks.

Proposed Solution Design and Implementation

Operation #1:

For this operation, I used the code given to us in the recursive slides available on BlackBoard where if a given set had a subset that summed up to a given value it would return true otherwise it would return false. I modified it so that instead of returning a boolean value it would return subsets that summed up to the given value. The algorithm I used to do this was iterating trough the array by comparing the sum of the first number and the second number in the array and seeing if they added to the given value if they did I would add the values to the subset by appending them. If the first number and the second number's sum was not the total of the given number it would increment the second number to the third number in the array this logic would repeat until the end of the array. After the first and last number's sum was totaled the first number was incremented to the second number and the first part of the algorithm's logic was repeated comparing the current number and all the numbers' sums adding to the subset only if their sum equaled the given number. The time complexity of this algorithm is O(n^2) because it is comparing every element twice when trying to create every possible subset combination. Some limitations of this implementation are that it will only create sets of two i took this constraint and made the assumption based on the examples provided that the biggest set that could be madrid a set of two positive integers

Operation #2:

For this operation, my solution was to apply a similar approach to the recursive examples given in,"drawing_recursion.py" but modifying it to be implemented with Stacks. The reason I would use a Stack is to take advantage of activation records. In the second method "draw_squares" the parameter provides four values ax,n,p,w but we don't need to record every value in the stack as some stay constant throughout the method call. I created a stack and store the values n and p given at the start, now while the length of the stack is greater than zero it would record the values n and p by appending them to the stack and then popping them to update the values this method was repeated throughout every single method that was covered in question 2. The basic algorithm used is to create a stack that would recorded the values where needed and constantly being updated. By using a stack we could update the values by popping them from the stack mimicking recursion.

Operation #3:

For this operation, my solution worked by implementing the recursive function "draw_four_squares" from the recursive function and plugging in the values used in the "four_circle method" I noticed a pattern in the implementation of both a curiously implemented the values. This leads me to discover that the implemented values where the needed ones this was because the values would create large square that could then be divided into smaller squares and those squares could then propagate more squares similar to cell division.

Operation #4:

For this operation, my solution was to apply a similar approach to operation #2. I looked at the recursive implementation and analyzed what values were being constantly updated and added them to the stack contrary to draw_four_cirlce this method used a for loop to print out every single smaller square in the bigger square this made it easier as we could use the same approach and add the updated values to the stack and every time the stack would pop the values would be updated so they could be printed.

Experimental Results

Operation #1:

For this operation, I decided to test with an empty set, An empty set was used as an opening test case as in the algorithm I use an empty set should return an empty subset followed by the given integer value. The test ran as expected.

Case 1 (mysubset_sum([],0))
Subset = [],0

For this operation, I decided to test the example given to us in the pdf, The given set was [1,3,4,6,7,9] that then becomes different subsets of two it was here that I noticed that if there ever was a set of three that could be made the algorithm would not catch that.

Case 2 (mysubset_sum([1,3,4,6,7,9],7))
Subset = [[1, 6], [3, 4], 7]

For this operation, I decided to test my hypothesis mentioned in case 2 and made a set that could be made by adding three digits, and as expected the set only returned combinations of two per subset. Notice how 6,4,3 could have made 13. This shows some limitations in the

implementation. To counter this a recursive implementation that checks every possible combination would have been best.

Case 3 (mysubset_sum([1,3,4,6,7,9],13))
Subset = [[6, 7], [7, 6], [9, 4], 13]

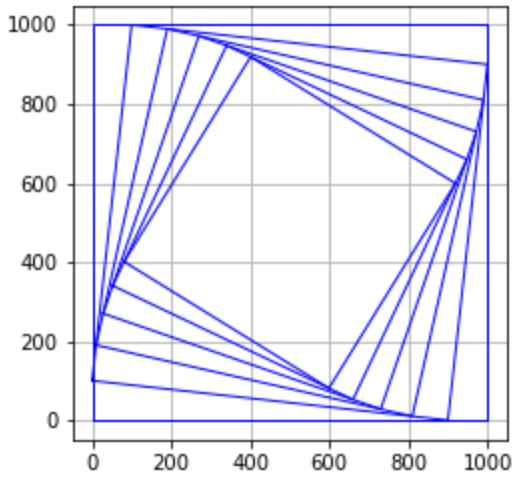
Operation #2

To test all the drawings covered in question two I used the same test cases as given in,"recursive_drawing" and copied them and used them to test the stack implementation of their respective counterparts. The variables passed down were all manipulated to match the requested drawings geometry.

Case 1:

```
fig, ax = plt.subplots()
draw_squares(ax,6,p,.1)
set_drawing_parameters_and_show(ax)
fig.savefig('squares1.png')
```

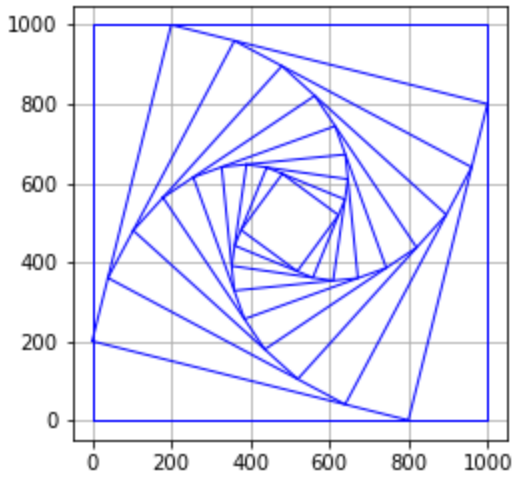
Drawing :



Case 2:

```
fig, ax = plt.subplots()
draw_squares(ax,10,p,.2)
set_drawing_parameters_and_show(ax)
fig.savefig('squares2.png')
```

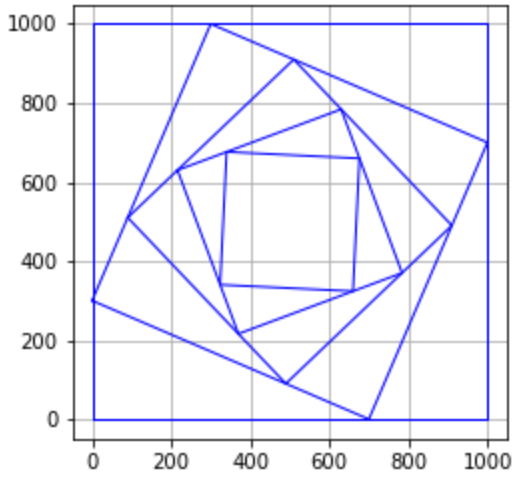
Drawing :



Case 3:

```
fig, ax = plt.subplots()
draw_squares(ax,5,p,.3)
set_drawing_parameters_and_show(ax)
fig.savefig('squares3.png')
```

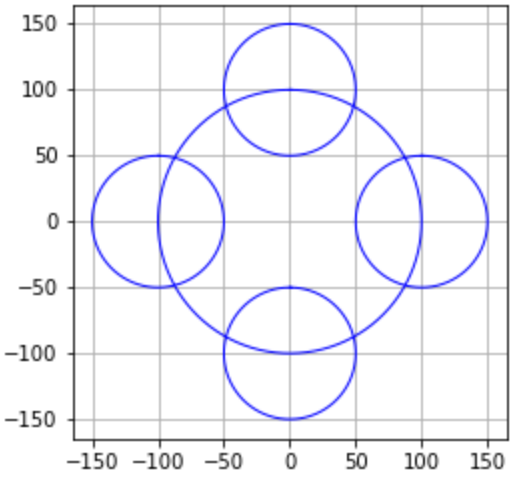
Drawing:



Case 4:

```
fig, ax = plt.subplots()
draw_four_circles(ax, 2, [0,0], 100)
set_drawing_parameters_and_show(ax)
fig.savefig('four_circles1.png')
```

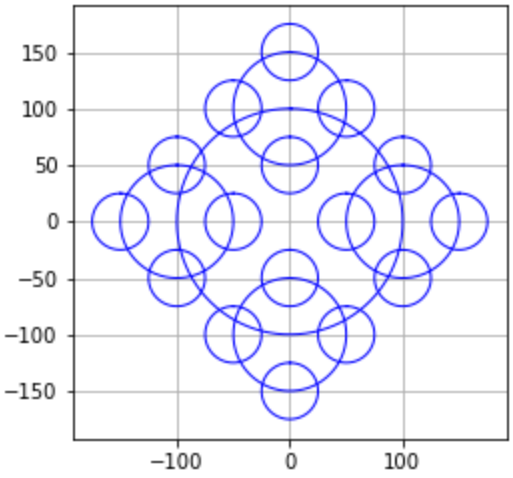
Drawing:



Case 5:

```
fig, ax = plt.subplots()
draw_four_circles(ax, 3, [0,0], 100)
set_drawing_parameters_and_show(ax)
fig.savefig('four_circles2.png')
```

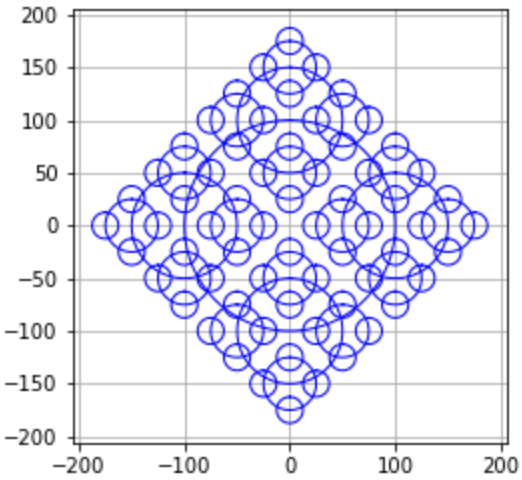
Drawing:



Case 6:

```
fig, ax = plt.subplots()
draw_four_circles(ax, 4, [0,0], 100)
set_drawing_parameters_and_show(ax)
fig.savefig('four_circles3.png')
```

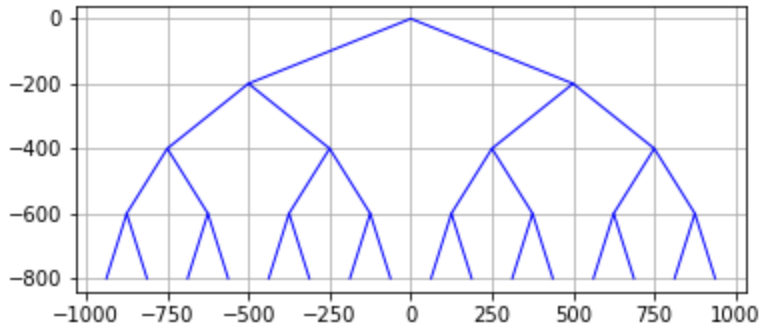
Drawing:



Case 7:

```
fig, ax = plt.subplots()
draw_tree(ax, 4, 0, 0, 500,200)
set_drawing_parameters_and_show(ax)
fig.savefig('tree1.png')
```

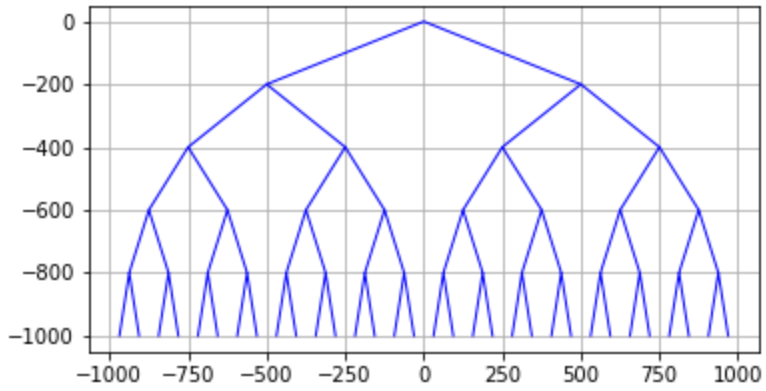
Drawing:



Case 8:

```
fig, ax = plt.subplots()
draw_tree(ax, 5, 0, 0, 500,200)
set_drawing_parameters_and_show(ax)
fig.savefig('tree2.png')
```

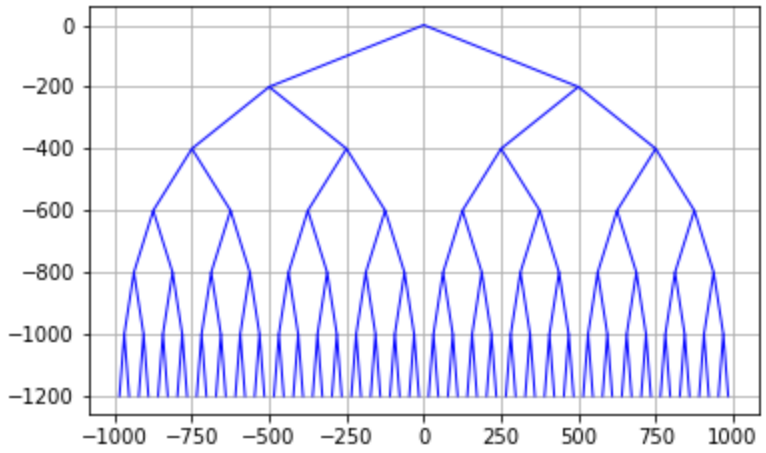
Drawing:



Case 9

```
fig, ax = plt.subplots()
draw_tree(ax, 6, 0, 0, 500,200)
set_drawing_parameters_and_show(ax)
fig.savefig('tree3.png')
```

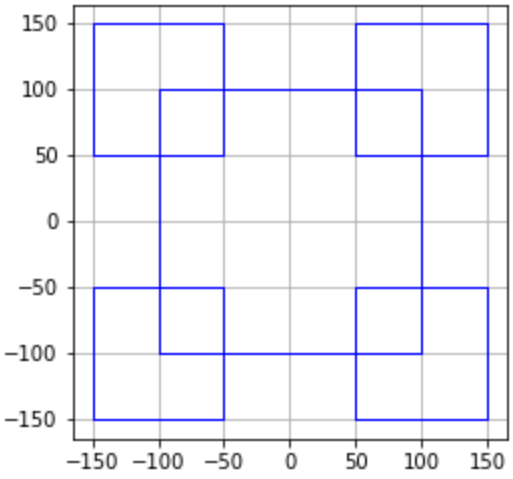
Drawing :



Case 10:

```
fig, ax = plt.subplots()
draw_four_squares(ax, 2, [0,0], 100)
set_drawing_parameters_and_show(ax)
fig.savefig('four_squres1.png')
```

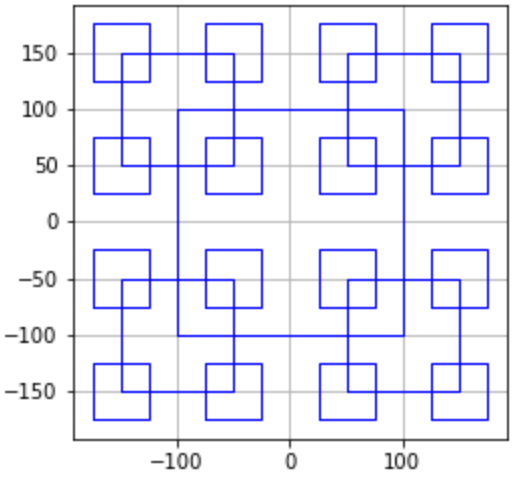
Drawing:



Case 11:

```
fig, ax = plt.subplots()
draw_four_squares(ax, 3, [0,0], 100)
set_drawing_parameters_and_show(ax)
fig.savefig('four_squares2.png')
```

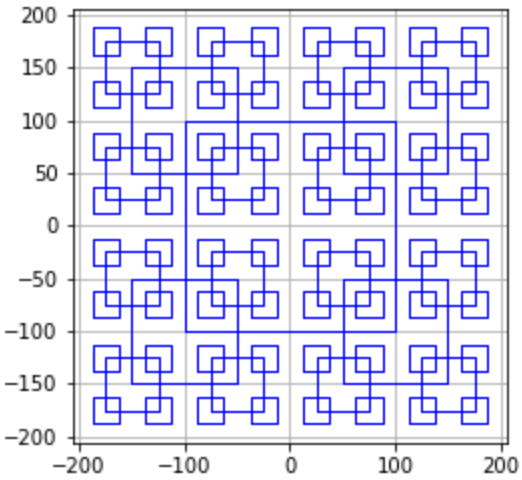
Drawing:



Case 12:

```
fig, ax = plt.subplots()
draw_four_squares(ax, 4, [0,0], 100)
set_drawing_parameters_and_show(ax)
fig.savefig('four_squares3.png')
```

Drawing

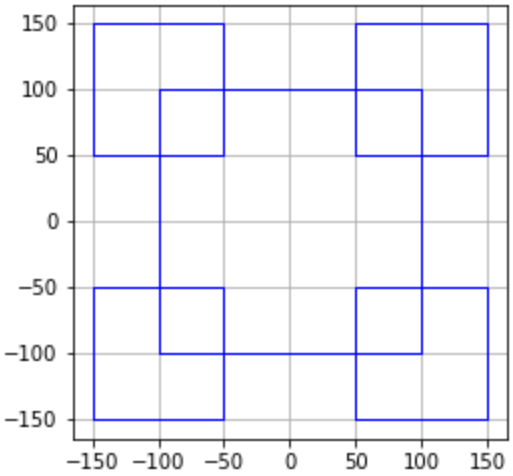


The following test cases are ment for the stack implementation of draw four squares eventough the results are the sam ehte implementation and cases does diffre slightly

Case 13:

```
fig, ax = plt.subplots()
draw_four_squares_stack(ax, 2, [0,0], 100)
set_drawing_parameters_and_show(ax)
fig.savefig('four_squres1_stack.png')
```

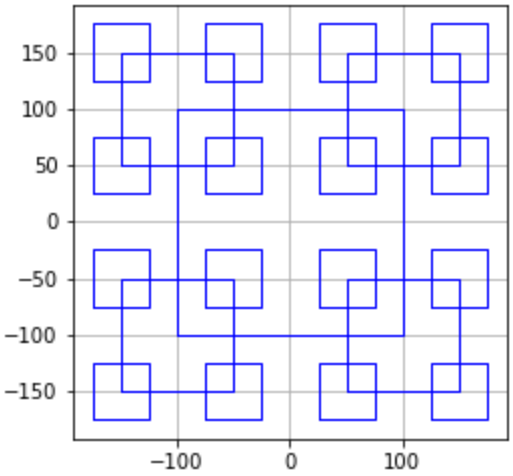
Drawing:



Case 14:

```
fig, ax = plt.subplots()
draw_four_squares_stack(ax, 3, [0,0], 100)
set_drawing_parameters_and_show(ax)
fig.savefig('four_squares2_stack.png')
```

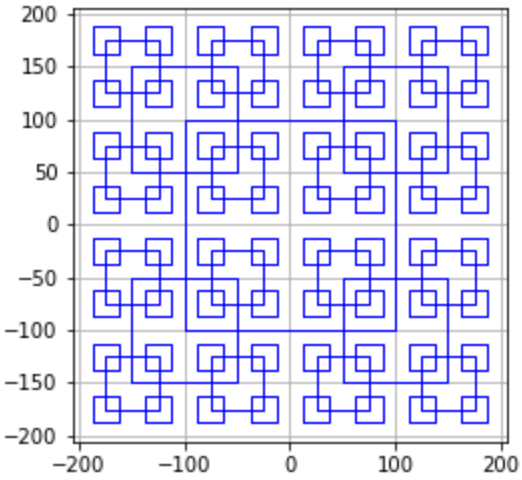
Drawing:



Case 15:

```
fig, ax = plt.subplots()
draw_four_squares_stack(ax, 4, [0,0], 100)
set_drawing_parameters_and_show(ax)
fig.savefig('four_squares3_stack.png')
```

Drawing:



Conclusion

This lab helped me understand that activation records are a record or screenshot of when that function was called this is very important for recursive implementation and when using stacks I understood what was happening I was taking a screenshot of the values and then updating the values when the single traversal of the while loop ended. using code and review the concepts from CS2 since we didn't work with code for Stacks as much as I would have liked to my knowledge of stacks was reinforced while still learning more of how to implement them in current ways I was able to get a clear view of how stacks and recursive activation are very similar theoretically but differ slightly in implementation. I believe that the operation we needed to code became easier hen we were given an example of how the stack was implemented. Before that example, I tough the stack implementation was rather difficult and spent a vast time looking at the other example stacks that were given to us.

Appendix

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon Sep 14 12:28:25 2020

@author: manuelgutierrez
"""
import numpy as np
import matplotlib.pyplot as plt
import math

def mysubset_sum(givenSet,givenNumber):
   subset = []
   returnset = []
   for i in range(len(givenSet)):
       for j in range(2,len(givenSet) - 1):
           # iterates by comapring the sum of the first number to all the other
numbers in the array
           # comparing if the sum equals the givenNumber
           if(givenSet[i] + givenSet[j] == givenNumber):
               subset.append(givenSet[i])
               subset.append(givenSet[j])
               returnset.append(subset)
               subset = []
   returnset.append(givenNumber)
   return returnset
```

```
def draw_squares(ax,n,p,w):
   stack = [[n,p]]
   while len(stack)>0:
       n,p = stack.pop()
       if n>0:
           ax.plot(p[:,0],p[:,1],linewidth=1.0,color='b') # Draw rectangle
           i1 = [1,2,3,0,1]
           stack.append([n-1,p*(1-w) + p[i1]*w])

#helper method that creates a circle with a given center and radius
def circle(center,radius):
   # Returns the coordinates of the points in a circle given center and radius
   n = int(4*radius*math.pi)
   t = np.linspace(0,6.3,n)
   x = center[0]+radius*np.sin(t)
   y = center[1]+radius*np.cos(t)
   return x,y

def draw_four_circles(ax,n,center,radius):
   stack = [[n,center,radius]]
   while len(stack)>0:
       n,center,radius = stack.pop()
       if n>0:
           x,y = circle(center,radius)
           ax.plot(x,y,linewidth=1.0,color='b')
           stack.append([n-1,[center[0],center[1]+radius],radius/2])
           stack.append([n-1,[center[0],center[1]-radius],radius/2])
           stack.append([n-1,[center[0]+radius,center[1]],radius/2])
           stack.append([n-1,[center[0]-radius,center[1]],radius/2])

def draw_tree(ax,n,x0,y0,dx,dy):
   stack = [[n,x0,y0,dx]] #dy is not needed because it never changes
   while len(stack)>0:
       n,x0,y0,dx = stack.pop()
       if n>0:
           x = [x0-dx,x0,x0+dx]
           y = [y0-dy,y0,y0-dy]
           ax.plot(x,y,linewidth=1.0,color='b')
```

```
            stack.append([n-1,x0-dx,y0-dy,dx/2])
            stack.append([n-1,x0+dx,y0-dy,dx/2])



def draw_four_squares(ax,n,center,size):
    if n>0:
        x = center[0] + np.array([-size,-size,size,size,-size])
        y = center[1] + np.array([-size,size,size,-size,-size])
        ax.plot(x,y,linewidth=1.0,color='b')
        for i in range(4):
            draw_four_squares(ax,n-1,[x[i],y[i]],size/2)



def draw_four_squares_stack(ax,n,center,size):
    stack = [[n,center,size]]
    while len(stack)>0:
        n,center,size = stack.pop()
        if n>0:
            x = center[0] + np.array([-size,-size,size,size,-size])
            y = center[1] + np.array([-size,size,size,-size,-size])
            ax.plot(x,y,linewidth=1.0,color='b')
            for i in range(4):
                stack.append([n-1,[x[i],y[i]],size/2])

def set_drawing_parameters_and_show(ax):
    show_axis = 'on'
    show_grid = 'True'
    ax.set_aspect(1.0)
    ax.axis(show_axis)
    plt.grid(show_grid)
    plt.show()



if __name__ == "__main__":
    print('Question 1')
    print(mysubset_sum([],0))
    print(mysubset_sum([1,3,4,6,7,9],7))
    print(mysubset_sum([1,3,4,6,7,9],10))
    print(mysubset_sum([1,3,4,6,7,9],13))
```

```
print()

print('Question 2')
plt.close("all") # Close all figures
orig_size = 1000.0
p = np.array([[0,0],[0,orig_size],[orig_size,orig_size],[orig_size,0],[0,0]])

fig, ax = plt.subplots()
draw_squares(ax,6,p,.1)
set_drawing_parameters_and_show(ax)
fig.savefig('squares1.png')

fig, ax = plt.subplots()
draw_squares(ax,10,p,.2)
set_drawing_parameters_and_show(ax)
fig.savefig('squares2.png')

fig, ax = plt.subplots()
draw_squares(ax,5,p,.3)
set_drawing_parameters_and_show(ax)
fig.savefig('squares3.png')

fig, ax = plt.subplots()
draw_four_circles(ax, 2, [0,0], 100)
set_drawing_parameters_and_show(ax)
fig.savefig('four_circles1.png')

fig, ax = plt.subplots()
draw_four_circles(ax, 3, [0,0], 100)
set_drawing_parameters_and_show(ax)
fig.savefig('four_circles2.png')

fig, ax = plt.subplots()
draw_four_circles(ax, 4, [0,0], 100)
set_drawing_parameters_and_show(ax)
fig.savefig('four_circles3.png')

fig, ax = plt.subplots()
draw_tree(ax, 4, 0, 0, 500,200)
```

```
    set_drawing_parameters_and_show(ax)
    fig.savefig('tree1.png')

    fig, ax = plt.subplots()
    draw_tree(ax, 5, 0, 0, 500,200)
    set_drawing_parameters_and_show(ax)
    fig.savefig('tree2.png')

    fig, ax = plt.subplots()
    draw_tree(ax, 6, 0, 0, 500,200)
    set_drawing_parameters_and_show(ax)
    fig.savefig('tree3.png')
    #by takeing th esame parameters as a circle i was able to recreate the square
drawings as the cordiantes would be simalar
    fig, ax = plt.subplots()
    draw_four_squares(ax, 2, [0,0], 100)
    set_drawing_parameters_and_show(ax)
    fig.savefig('four_squres1.png')

    fig, ax = plt.subplots()
    draw_four_squares(ax, 3, [0,0], 100)
    set_drawing_parameters_and_show(ax)
    fig.savefig('four_squares2.png')

    fig, ax = plt.subplots()
    draw_four_squares(ax, 4, [0,0], 100)
    set_drawing_parameters_and_show(ax)
    fig.savefig('four_squares3.png')

    fig, ax = plt.subplots()
    draw_four_squares_stack(ax, 2, [0,0], 100)
    set_drawing_parameters_and_show(ax)
    fig.savefig('four_squres1_stack.png')

    fig, ax = plt.subplots()
    draw_four_squares_stack(ax, 3, [0,0], 100)
    set_drawing_parameters_and_show(ax)
    fig.savefig('four_squares2_stack.png')
```

```
fig, ax = plt.subplots()
draw_four_squares_stack(ax, 4, [0,0], 100)
set_drawing_parameters_and_show(ax)
fig.savefig('four_squares3_stack.png')
print("Drawings....done")
```