Lab 1 : Recursion Functionality

Steven Robles *
University of Texas, El Paso, TX

February 8, 2019

This paper aims to demonstrate the functionality of recursive functions and analyze their efficiency. To demonstrate and analyze the functionality, four main test were derived to sample the functions. Each individual test compromised plotting a series of figures in Python by manipulated singular points of an array and utilizing the numpy library. Each sample were given three test inputs to exhibit the various number of recession calls. The analysis of the functions is preformed by timing the executions of each sample. Concluded in this paper, the efficiency of the recursive functions decreases as the number of recursive calls within the function itself increases.

;

Contents

I. Introduction

In the world of programming, there are two common approaches for iterations; Loops and Recursions. For simple applications loops can be concluded as the more practical solution for the task at had. To accomplish more complicated applications although, loop iterations may not always be best solution. With the inability to backtrack between each iteration, loops may take longer times to complete certain tasks compared to that of recursion iterations. The ability for the recursion functions to backtrack or what is known as tail recursion between each iteration reduces the amount of time executed on the task thus increasing the overall inefficiency.

To demonstrate the functionality of recursive functions four tasks are given to preform. Each of these tasks consists of drawing polynomials in a specific pattern. The number of polynomials to be plotted in the same figure is dependant on the depth of the given pattern. To meet the criteria of the specified pattern, each new polynomial must be modified and adjusted in accordance to its previous counterparts and as a result, it showcases the functionality of the recursion statements in a physical manner.

*ID Number: 80678755

II. Design

To complete each task, each problem is broken down to its fundamental parts. The fundamental parts are identified by analyzing the targeted polynomial and the target pattern. To begin with, each polynomial is made up of points connected by utilizing the numpy array provided in Python. The number of points needed for each polynomial is :

$$N_{Points} = N_{Edges} + 1 \tag{1}$$

Note that N_{Edges} only account for definitive straight edges. Circles will be conducted as a singular center point and its radius. By modifying the x and y coordinates of the points on a *Cartesian Plane*, a polynomial is constructed. Since each problem has a unique polynomial and pattern, each modification is different and is depended on previous point coordinates.

A. Part 1 : Squares

For the first problem of the lab the pattern requires draw a square at the four corners of the previous square. This signifies that there will be four fundamental parts, or in other words four recursive calls to the function; one to construct the top right, another to construct the top left, another to construct bottom left, and a last one to construct to bottom right. This approach is was is often called "devinding and conquering". By carefully analyzing the sizes of each new square and comparing to it previous from the pattern, I concluded that each new length is half the size of its previous length therefor effectively reducing all x and y point coordinates by half as shown below:

$$P_n(x, y) = P_{n-1}(x * 0.50, y * 0.50) \tag{2}$$

After the size of current square has been properly reduced, it must be moved to one of the four locations. To keep the recursive function accurately shift the current n square, the amount shifted is determined by using the previous n coordinates as well. It is important to keep in mind that since the current n square has already changed it side lengths, there will be new existing coordinates for x and y. From this comparison, you can clearly tell that each shift is compromised of two relations from the previous coordinate; 75% and 25%. The equations below provided a formula for the new locations to draw the square.

$$P_{n_{Top\_\_\_Right}}(x, y) = P_n(x + (0.75 * x_{n-1}), y + (0.75 * y_{n-1}) \tag{3}$$

$$P_{n_{Top\_\_\_Left}}(x, y) = P_n(x - (0.25 * x_{n-1}), y + (0.75 * y_{n-1}) \tag{4}$$

$$P_{n_{Bottom\_Left}}(x, y) = P_n(x - (0.75 * x_{n-1}), y - (0.75 * y_{n-1}) \tag{5}$$

$$P_{n_{BottomRight}}(x, y) = P_n(x + (0.75 * x_{n-1}), y - (0.25 * y_{n-1}) \tag{6}$$

Since there four new squares at the n level of recursion, each recursion has to have for call to its self; one for each corner. This process is followed until we reached the targeted pattern provided in the problem.

B. Part 2: Left Circles

The purpose of Part 2 is to draw circles with only its left-most point overlapping the previous left most point. Identifying the relation between each center points of the recursion levels is key to accomplish the specific pattern. The amount shifted is solely dependant on the percentage of how much the radius shrinks as show below:

$$P_n(x, y) = (x * radius_{n-1}, y) \tag{7}$$

A design challenge that was encountered in this part is that the amount of percent in which the radius is reduced varies as the number of expected recursions changes. For this reason, there is no user input in this part or in any of the others. The only user interface allowed in in this lab is when the number of recursions is adjusted when timing the efficiency's of each part.

C. Part 3: Tree

As a result of the required polynomial and pattern combined, this part is one of the most difficult sections of the lab. The polynomial is similar to that of part 1 with only half of the square being plotted, but the shifting is quite more complex. Since it is a tree and each leaf is designed to have two children at each recursion level, this means that the angle between each siblings decreases ever so slightly to prevent from *"criss-crossing"* at lower levels. Changing only the angles can proved to be quite difficult, so instead we can focused on the decreasing distance between each sibling. This is considerably easier to model as shown in equations 8 and 9.

$$P_{Left}(x, y) = (x - (width_{n-1} * 0.50), y - ((width_{n-1} + 1) * 0.50)) \tag{8}$$

$$P_{Right}(x, y) = (x + ((width_{n-1} + 1) * 0.50), y + (width_{n-1} * 0.50)) \tag{9}$$

Where the width is the previous left x endpoint coordinate subtracted from the previous mid-point x coordinate. Since this section requires the addition of two children for each parent node, the recursion function is called twice, hence the two equations above. Each call contains one of the equation, plots it, and continues onto the next level before plotting the other child.

D. Part 4: Circles

This last section models a similar pattern to that of Part 1. Although there only exists one point in the problem, there will be five recursive calls to achieve the target pattern. A challenge surface when printing each circle after it shifted. In order to prevent mirrored images, the center point must be shifted back to its origin point before proceeding to plot the next n circle. The $\pm$ in the following three equation represent the back and fourth shifting.

$$P_{X_Shift}(x, y) = (x \pm (radius_{n-1} * 0.66), y) \tag{10}$$

$$P_{y_Shift}(x, y) = (x, y \pm (radius_{n-1} * 0.66)) \tag{11}$$

The *radius* is multiplied by 0.66 to create equal spacing, since each direction there are three circles that must fit within the previous one. Of course this section does not only require to move the center points, but reduce the radius itself which was calculated as 33% of its original length.

III. Results

In order to analyze the designs of each recursion, the plotted figure must be shown. Additionally, each part had to output three different test cases as well as undergo run time testing. This demonstrates how each particular recursion function can handle deep levels of recursion and untimely showcasing their functionality. The following figures display each results from three particular test cases:

Figure 1: Part 1: Squares

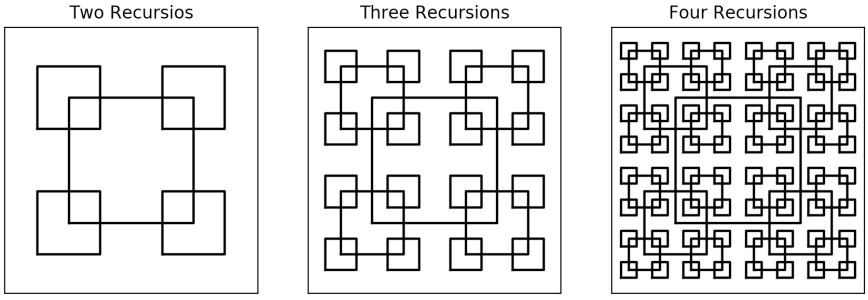


Figure 2: Part 2 : Left Circles

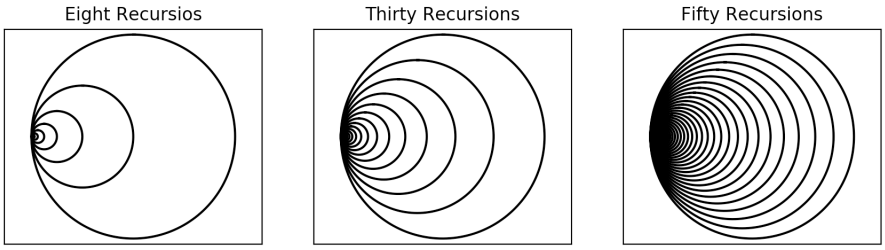


Figure 3: Part 3: Tree

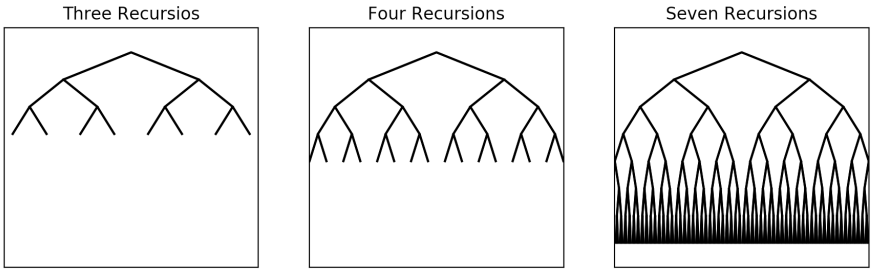
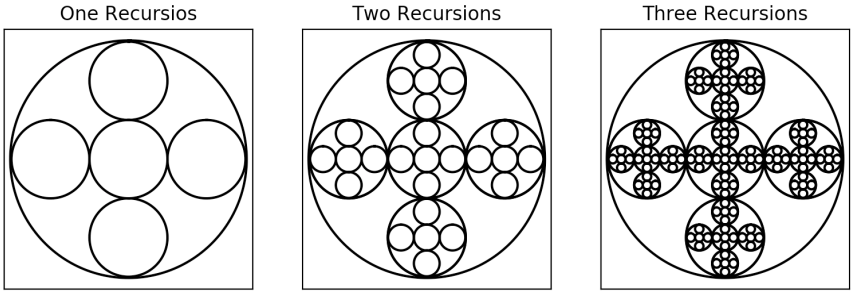


Figure 4: Part 4 : Circles



The three separate cases for each problem not only test the recursion function itself but also demonstrates the different depth levels each polynomial is shown. Notice on how some recursion structures might be more complex than others. This also has a direct effect on its run times. The Figures 5 - 8 displays the results of each part with different depth levels.

Figure 5

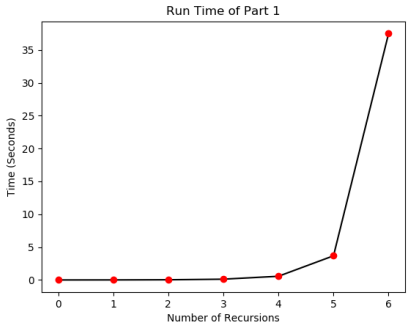


Figure 6

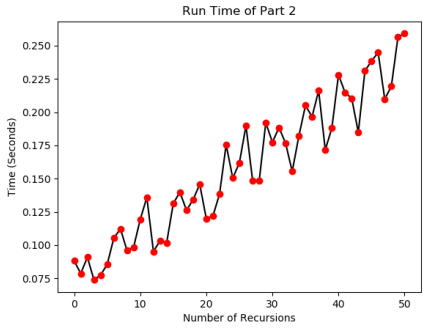


Figure 7

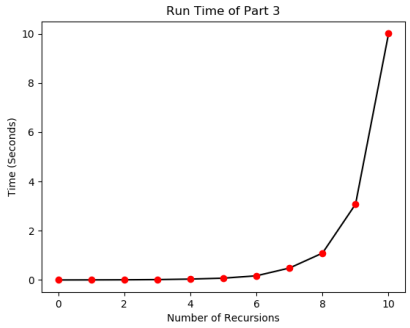
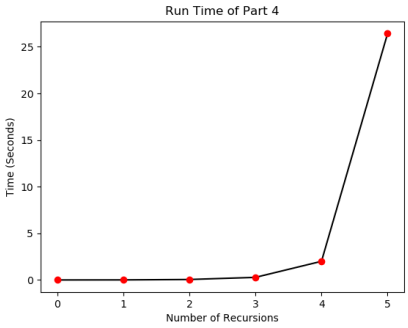


Figure 8



Each figures shows that the test runs are measured in seconds, but have different maximum depth levels being tested. Figures 5 and 8 hold the least number number of depth levels since both functions call itself the most; 4 and 5 times respectively. This causes the time taken to finish executed the recursion function as shown in the increment of the graph. Figure 7 shows the time runs of part and is the second fastest after Part 2 shown in Figure 6. Since Part 2 only call itself once, the recursion only has to execute once instance for each recursion therefore achieving the least amount of time for each depth level.

IV. Discussion

Overall, the given problems and their solutions demonstrate the functionality of recursive functions profoundly. It shows that recursive functions are deigned to break down the problem into smaller solvable parts in what is knows as "divide and conquer". A negative effect is that is also creates many instances of the object being manipulated thus increasing the run times shown in Figures 5 - 8. The complexity of the recursive function although, all depends on the number of parts the function intends to break down.

V. Source Code

```
#
    ********************************************************************************************
#Author: Steven J. Robles
#Class: CS 2302 Data Structures III
#Instructor: Olac Fuentes
#TA: Anindita Nath And Maliheh Zargaran
#Last Modified: February 7, 2019
#Discreption: Lab 1:
#    The purpose of this program is to expierament with recrusive functions by plotting
    several figures. When each
#    recursive function is called, the figure should be plotted, but it will be modified.
    Modification consists of
#    shrinking and shifting the firgure respectivly to the previous firgures. All of this is
     done through points
#    created by the numpy library. The Labs is devided into four sectoin with new drawings
    within each section. They
#    are seperated into four different files accessed from below.
#
    ********************************************************************************************


from Squares import Section1
from Squares import RuntimeCheck1
from LeftCircles import Section2
from LeftCircles import RuntimeCheck2
from Tree import Section3
from Tree import RuntimeCheck3
from Circles import Section4
from Circles import RuntimeCheck4

#Section1()
#Section2()
#Section3()
#Section4()

n1 = 7
n2 = 51
n3 = 11
n4 = 6

#the section below are the test run time call functions to test the run times of 0 to n
#recursive inputs of each section. They are edited by providing the numbers above.

RuntimeCheck1(n1)
#RuntimeCheck2(n2)
#RuntimeCheck3(n3)
#RuntimeCheck4(n4)
```

```

#
    ********************************************************************************************
#Author: Steven J. Robles
#Class: CS 2302 Data Structures III
#Instructor: Olac Fuentes
#TA: Anindita Nath And Maliheh Zargaran
#Last Modified: February 7, 2019
#Discreption: Lab 1 - Part 1:
#    The purpose of this program is to expierament with recrusive functions by plotting a
    series of squares. Each
#    is reduced by half repsectivley form the previous square and is shifted to all four
    corners of the previous
#    square.
#
    ********************************************************************************************
```

```
import matplotlib.pyplot as plt
import numpy as np
import timeit

#recurse fucntion of which shrinks, shifts, and plots the squares
def draw_squares(ax,n,p,w, length):
    if n>0:

        #the followin shifts the squares 75% and/or 25% of the previous length of the square
        bigShift = length*.75
        smallShift = length*.25

        points = [0, 1 , 2, 3, 4]

        #top left square
        q = p*w
        q[points, 0] -= smallShift
        q[points, 1] += bigShift
        ax.plot(p[:,0],p[:,1],color='k')
        draw_squares(ax,n-1,q,w, length)

        #top right squares
        q = p*w
        q[points] += bigShift
        ax.plot(p[:,0],p[:,1],color='k')
        draw_squares(ax,n-1,q,w, length)

        #Bottom Left Square
        q = p*w
        q[points] -= smallShift
        ax.plot(p[:,0],p[:,1],color='k')
        draw_squares(ax,n-1,q,w, length)

        #Bottom Right Square
        q = p*w
        q[points, 0] += bigShift
        q[points, 1] -= smallShift
        ax.plot(p[:,0],p[:,1],color='k')
        draw_squares(ax,n-1,q,w, length)

#This definition is called from the main program of which initiates the recursive call. It also sets up and
#show the plots produced in a pair of three.
def Section1():
    plt.close("all")
    orig_size = 524288
    p = np.array([[0,0],[0,orig_size],[orig_size,orig_size],[orig_size,0],[0,0]])
    fig, (ax1, ax2, ax3) = plt.subplots(1, 3, sharex = True,sharey = True)
    ax1.axis('on')
    ax1.set_aspect(1.0)
    ax1.yaxis.set_major_locator(plt.NullLocator())
    ax1.xaxis.set_major_locator(plt.NullLocator())
    ax1.set_title('Two Recursios')
    ax2.set_title('Three Recursions')
    ax3.set_title('Four Recursions')
    draw_squares(ax1,2,p, .50, orig_size)
    draw_squares(ax2,3,p, .50, orig_size)
    draw_squares(ax3,4,p, .50, orig_size)
    plt.show()
    fig.savefig('Squares.png')

#The following definition is called to check the runtime of the program when there are 0 recursion called up to 7
# and plots the results
def RuntimeCheck1(n):

    #sets the size of the array of which the run times are going to be stored
    times = [0] * n

    orig_size = 2147483648
```

```
    p = np.array([[0,0],[0,orig_size],[orig_size,orig_size],[orig_size,0],[0,0]])
    fig, ax = plt.subplots()


    #this loop is Not recursice. It calls the recursive function from 0 to n times and stores the time onto the previous array
    for i in range(len(times)):
        start = timeit.default_timer() # starts timer

        draw_squares(ax,i,p, .50, orig_size)

        stop = timeit.default_timer() # ends timer
        times[i] = stop - start #stores the lapsed time

    #proceeds to plot the results
    plt.close("all")
    plt.title('Run Time of Section 1')
    plt.xlabel('Number of Recursions')
    plt.ylabel('Time (Seconds)')
    x = np.arange(0,n,1)
    plt.plot(x, times, 'k', x, times, 'ro')
    plt.savefig('1_Runtime')
    plt.show()
```

```
#*****************************************************************************************

#Author: Steven J. Robles
#Class: CS 2302 Data Structures III
#Instructor: Olac Fuentes
#TA: Anindita Nath And Maliheh Zargaran
#Last Modified: February 7, 2019
#Discreption: Lab 1 - Part 2:
#    The purpose of this program is to expierament with recrusive functions by plotting a series of circles which are
#    reduced by specific percent and shift so thhe left most point of the new circle is aligned with the previous left
#    most point of the cirlce. The percantage of the radius reduced and the amount of shifting depends on the number of
#    recersions which is already provide in the code.
#
#*****************************************************************************************

import matplotlib.pyplot as plt
import numpy as np
import math
import timeit

#The following creats the cirlce with the given center point and radius
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

#The following function is the recrusive function of which plots the cricle, and reduces and shifts the next one.
def draw_circles(ax,n,center,radius,w):
    if n>0:

        center[0] = radius * .9999  # Shifts the next circle .0001% to the right of the left side
                                    # so you can tell its a new circle
        x,y = circle(center,radius)
        ax.plot(x,y,color='k')
        draw_circles(ax,n-1,center,(radius*w),w)

#This definition is called from the main program of which initiates the recursive call. It also sets up and
```

```
#show the plots produced in a pair of three
def Section2():
    fig, (ax1, ax2, ax3) = plt.subplots(1, 3, sharex = True,sharey = True)
    ax1.axis('on')
    ax1.set_aspect(1.0)
    ax1.yaxis.set_major_locator(plt.NullLocator())
    ax1.xaxis.set_major_locator(plt.NullLocator())
    ax1.set_title('Eight Recursios')
    ax2.set_title('Thirty Recursions')
    ax3.set_title('Fifty Recursions')
    draw_circles(ax1, 8, [500, 0], 500, .50)
    draw_circles(ax2, 30, [500, 0], 500, .75)
    draw_circles(ax3, 50, [500, 0], 500, .90)
    plt.show()
    fig.savefig('LeftCircles.png')

#The following definition is called to check the runtime of the program when there are 0 recursion called up to 50
# and plots the results
def RuntimeCheck2(n):

    #sets the size of the array of which the run times are going to be stored
    times = [0] * n
    fig, ax = plt.subplots()

    #this loop is Not recursice. It calls the recursive function from 0 to n times and stores the time onto the previous array
    for i in range(len(times)):


        start = timeit.default_timer() # starts timer

        draw_circles(ax, n, [500, 0], 500, .90)

        stop = timeit.default_timer() # ends timer

        times[i] = stop - start #stores the lapsed time

    #proceeds to plot the results
    plt.close("all")
    plt.title('Run Time of Part 2')
    plt.xlabel('Number of Recursions')
    plt.ylabel('Time (Seconds)')
    x = np.arange(0,n,1)
    plt.plot(x, times, 'k', x, times, 'ro')
    plt.savefig('2_Runtime')
    plt.show()
```

```
#
    ********************************************************************************************************

#Author: Steven J. Robles
#Class: CS 2302 Data Structures III
#Instructor: Olac Fuentes
#TA: Anindita Nath And Maliheh Zargaran
#Last Modified: February 7, 2019
#Discreption: Lab 1 - Section 3:
#    The purpose of this program is to expierament with recrusive functions by plotting a series of trees which
#    are applieds a new level to the tree each recursive call.
import numpy as np
import matplotlib.pyplot as plt
import timeit

#the following is the recursive function which creates a new level by adding two children to the bottom leafs
def draw_trees(ax,n,p,w, h, wt):
    if n>0:

        #the followin adds two children to the left leaf
```

```
        endPoints = [0,2]
        q =  p * 1.0001
        q[1] = p[0]     #sets the midle point of the new leafs to the end of the previous one.
        #The folloiwng to lines makes sure the length of the new leafs in the x direction is half from the previous one
        q[0,0] -= w*wt
        q[2,0] -= (1+w)*wt
        q[endPoints,1] -= h
        wt1 = q[1,0]- q[0,0]
        ax.plot(p[:,0],p[:,1],color='k')
        draw_trees(ax,n-1,q,w, h, wt1)

        #the following adds children to the right leaf
        q =  p * 1.0001
        q[1] = p[2]     #sets the midle point of the new leafs to the end of the previous one.
        #The folloiwng to lines makes sure the length of the new leafs in the x direction is half from the previous one
        q[0,0] += (1+w)*wt
        q[2,0] += w*wt
        q[endPoints,1] -= h
        ax.plot(p[:,0],p[:,1],color='k')
        draw_trees(ax,n-1,q,w, h, wt1)

#This definition is called from the main program of which initiates the recursive call. It also sets up and
#show the plots produced in a pair of three.
def Section3():
  fig, (ax1, ax2, ax3) = plt.subplots(1, 3, sharex = True,sharey = True)
  ax1.axis('on')
  ax1.set_aspect(1.0)
  ax1.yaxis.set_major_locator(plt.NullLocator())
  ax1.xaxis.set_major_locator(plt.NullLocator())
  ax1.set_title('Three Recursios')
  ax2.set_title('Four Recursions')
  ax3.set_title('Seven Recursions')
  height = 800
  width = 2000
  p = np.array([[100,100],[2100,900],[4100,100]])
  draw_trees(ax1,3,p,.5, height, width)
  draw_trees(ax2,4,p,.5, height, width)
  draw_trees(ax3,7,p,.5, height, width)
  plt.show()
  fig.savefig('Trees.png')

#The following definition is called to check the runtime of the program when there are 0 recursion called up to 50
# and plots the results
def RuntimeCheck3(n):

    #sets the size of the array of which the run times are going to be stored
    times = [0] * n
    fig, ax = plt.subplots()
    height = 800
    width = 2000
    p = np.array([[100,100],[2100,900],[4100,100]])

    #this loop is Not recursice. It calls the recursive function from 0 to n times and stores the time onto the previous array
    for i in range(len(times)):

        start = timeit.default_timer() # starts timer

        draw_trees(ax,i,p,.5, height, width)

        stop = timeit.default_timer() # ends timer

        times[i] = stop - start #stores the lapsed time

```

```
    #proceeds to plot the results
    plt.close("all")
    plt.title('Run Time of Part 3')
    plt.xlabel('Number of Recursions')
    plt.ylabel('Time (Seconds)')
    x = np.arange(0,n,1)
    plt.plot(x, times, 'k', x, times, 'ro')
    plt.savefig('3_Runtime')
    plt.show()
```

```
#
    ********************************************************************************************

#Author: Steven J. Robles
#Class: CS 2302 Data Structures III
#Instructor: Olac Fuentes
#TA: Anindita Nath And Maliheh Zargaran
#Last Modified: February 7, 2019
#Discreption: Lab 1 - Section 4:
#    The purpose of this program is to expierament with recrusive functions by plotting a
    series of circles which are
#    by 66% and placed in the middle, top, right, bottom and left of the previous cirlce.
    This will cause a patter of
#    cirlces within circles which demonstrates the effectivness of recersion algrorythims
#
    ********************************************************************************************

import matplotlib.pyplot as plt
import numpy as np
import math
import timeit

#The following constructs the circle given the center point and its radius
def circle(center,rad):
    n = int(10*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

#The recursive function reduces and shifts 4 circles accroding to their destinations
    respective to the previous cricle
def draw_circles(ax,n,center,radius,w):
    if n>0:

        #plots the origional sized cricle "Parent"
        x,y = circle(center,radius)
        ax.plot(x,y,color='k')

        #plots the reduced circle in the middle
        x,y = circle(center,radius*w)
        ax.plot(x,y,color='k')
        draw_circles(ax,n-1,center,(radius*w),w)
        #plots the reduced circle in the to to the right
        center2 = center
        center2[0] -= radius*.66    #shifts the center point to the right
        x,y = circle(center2,radius*w)
        ax.plot(x,y,color='k')
        draw_circles(ax,n-1,center,(radius*w),w)
        center2[0] += radius*.66   #shifts the center point back to its oriigional locatoins
    -- same with all of the rest

        #plots the reduced circle in the to to the left
        center2 = center
        center2[0] += radius*.66
        x,y = circle(center2,radius*w)
        ax.plot(x,y,color='k')
        draw_circles(ax,n-1,center,(radius*w),w)
        center2[0] -= radius*.66

```

```
    #plots the reduced circle in the to the top
        center2 = center
        center2[1] -= radius*.66
        x,y = circle(center2,radius*w)
        ax.plot(x,y,color='k')
        draw_circles(ax,n-1,center,(radius*w),w)
        center2[1] += radius*.66

    #plots the reduced circle in the to the bottom
        center2 = center
        center2[1] += radius*.66
        x,y = circle(center2,radius*w)
        ax.plot(x,y,color='k')
        draw_circles(ax,n-1,center,(radius*w),w)
        center2[1] -= radius*.66

#This definition is called from the main program of which initiates the recursive call. It also sets up and
#show the plots produced in a pair of three
def Section4():
    fig, (ax1, ax2, ax3) = plt.subplots(1, 3, sharex = True,sharey = True)
    ax1.axis('on')
    ax1.set_aspect(1.0)
    ax1.yaxis.set_major_locator(plt.NullLocator())
    ax1.xaxis.set_major_locator(plt.NullLocator())
    ax1.set_title('One Recursios')
    ax2.set_title('Two Recursions')
    ax3.set_title('Three Recursions')
    draw_circles(ax1, 1, [0, 0], 20,.33 )
    draw_circles(ax2, 2, [0, 0], 20,.33 )
    draw_circles(ax3, 3, [0, 0], 20,.33 )
    plt.show()
    fig.savefig('Circles.png')

#The following definition is called to check the runtime of the program when there are 0 recursion called up to 50
# and plots the results
def RuntimeCheck4(n):

    #sets the size of the array of which the run times are going to be stored
    times = [0] * n
    fig, ax = plt.subplots()

    #this loop is Not recursice. It calls the recursive function from 0 to n times and stores the time onto the previous array
    for i in range(len(times)):

        start = timeit.default_timer() # starts timer

        draw_circles(ax, i, [0, 0], 20,.33 )

        stop = timeit.default_timer() # ends timer

        times[i] = stop - start #stores the lapsed time

    #proceeds to plot the results
    plt.close("all")
    plt.title('Run Time of Part 4')
    plt.xlabel('Number of Recursions')
    plt.ylabel('Time (Seconds)')
    x = np.arange(0,n,1)
    plt.plot(x, times, 'k', x, times, 'ro')
    plt.savefig('4_Runtime')
    plt.show()
```