Lab Assignment 3

Before you start, please keep in mind that it is always good to decompose your whole code into a few functions that each of them will take care of a specific smaller problem/task.

Important: In this lab, you should not import any other function from other packages except the package **random**. You may import the function sqrt() from math, but that will NOT give you full mark.

Problem 1: Who are the best friends?

We have 4 very good friends Albert(A), Billy(B), Carl(C) and David(D), and we want to see which pair of them is the ultimate best friends to each other. We have a machine that can test a pair of person how "matching" they are and give a score.

In order to find the best pair, we have to try every pair, namely,

AB, AC, AD, BC, BD and CD.

So we have to repeat the test **6** times because there are six different pairs. (And the pairs AB and BA are counted as one single pair only.) How many times do we have to do if we have *n* friends instead of 4? And what if we want to find the best matching *team* with *k* persons instead of a pair of 2 when the machine can measure ***k*** persons at the same time? How many times do we have to repeat the test?

In fact, this number is called "*n* choose *k*", or the ***binomial coefficient***, in combinatorics, or simply ${}_nC_k$ or $\binom{n}{k}$.

The formula is $\binom{n}{k} = \frac{n!}{k!(n-k)!}$

a. Write a function **`nChooseK(n,k)`** WITHOUT using any recursion.
b. Write a recursive version of **`nChooseK_recursive(n,k)`** to compute the binomial coefficient by using recursion **without** using any factorial functions or loops. The binomial coefficient can be expressed in another form:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k},$$

for $\binom{n}{n} = \binom{n}{0} = 1$.

Problem 1 (cont.)

Sample output:

```
>>> print(nChooseK(6,2))
15
>>> print(nChooseK(10,5))
252
>>> print(nChooseK_recursive(10,5))
252
>>>
```

c. Find the largest number (output) that your functions are able to compute within 1 minute in the above two parts. You don't have to find an exact tight bound; just something close is good enough. Which version is better and in what ways? Write your thoughts in the comment part of your .py file.

A final note for this part, your output of your function MUST be **INTEGERS**.

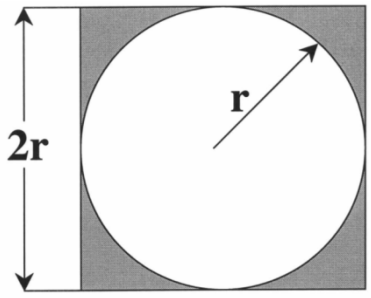
Problem 2: Approximating π

The constant π is used in mathematics, physics and other related fields such as engineering extensively. In this exercise, you are going to compute an approximation of the constant π .

Scenario

Gambler Jack is no mathematician. His friends laugh at him and make a bet that Jack does not know the number π, not even the first three most significant digits. Jack is going to lose the bet but his girlfriend is an accountant and she is going to help. She suggests Jack the followings.

See? You see that dartboard on the wall with the frame?



The circle has a radius ***r*** and the frame is a square with length 2***r***. What we will do is, we will throw a lot of darts randomly onto the board. And the probability ***p*** of the dart hitting the circle will be:

$$p = \frac{\text{Area of Circle}}{\text{Area of Square}} = \frac{\pi r^2}{(2r)^2} = \frac{\pi}{4}$$

So, if you throw a lot of darts within the frame and some of them hit the circle randomly, the probability is the same as ***p*** and we have

$$\frac{\text{No. of dart in circle}}{\text{Total no. of darts}} = p = \frac{\text{Area of Circle}}{\text{Area of Square}} = \frac{\pi r^2}{(2r)^2} = \frac{\pi}{4}$$

So, we can calculate π by the above formula! For example, if I throw 10 darts and 8 of them landed inside the circle, then $\pi \approx$ (8/10) x 4 = 3.2 .

Your task

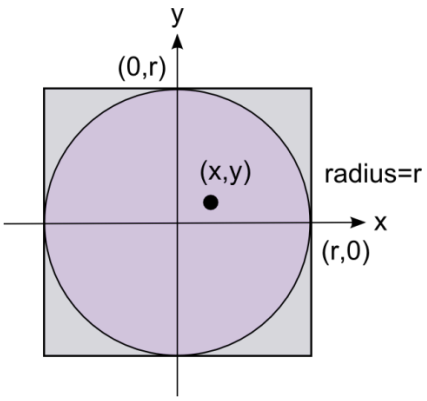
Your task is to implement a function, **monte_carlo_pi(n)** which returns the approximation of π by throwing the darts n times. So the more darts you throw, the more accurate your $\boldsymbol{\pi}$ is.

Sample output:

```
>>> monte_carlo_pi(10)
3.6
>>> monte_carlo_pi(100)
3.32
>>> monte_carlo_pi(1000)
3.084
>>> monte_carlo_pi(10000)
3.1348
>>> monte_carlo_pi(100000)
3.1412
>>> monte_carlo_pi(1000000)
3.140808
>>> monte_carlo_pi(10000000)
3.1413096
>>>
```

Steps:

The algorithm to approximate the value of π using the method described earlier is as follows.



1. Let the radius of the circle be r and the length of the square be 2r. (Is it ok to just simplify r = 1?)
2. Generate two random numbers x, y, such that $-r \leq x, y \leq r$. The position (x, y) is our dart position. *Hint: Use the function in the package `random` that starts with the letter 'u'.*
3. Calculate the distance $\boldsymbol{d}$ from the dart position to the center of the circle.
4. If $d \leq r$. This implies that the point, (x, y) is within the circle. Otherwise, (x, y) falls outside of the circle, but in the square. Remember, you cannot use the function sqrt(), how do you get around with this?
5. Repeat the procedure n times, and keep track of the number of points that fall inside the circle, k.
6. After you have completed n trials, compute $p = \frac{k}{n}$.
7. For large n, the value of π can be approximated by $4p$.