Justin Li
1/19/23

COEN 379 HW 1

1. Assume that we generate the values x and y so that:
   x is a random value between 0 and 2*root(2)
   y is a random value between 0 and 4

   Thus, the probability that the values we generate an ordered pair that exists within the curve is equal to the area of the curve divided by the area of the square of length 2*root(2) * 4.

   We know that the values x and y are in the curve if they follow this inequality:
   (x^2 + y^2 - 4)^3 - 108y^2 <= 0 (the equation given)

   Thus, we estimate the area of the curve by running a simulation with x and y 10 million times in order to get the probability of a random value being inside the curve, and then multiplying that value by the area of the square and multiplying by 4 (for each quadrant) to get the total area.

   Simulation Code:

```
import math
import random as rand

x = 0
y = 0
count = 0
iterations = int(input("How many iterations would you like to run?
"))
for i in range(0, iterations):
    x = rand.uniform(0, 2*math.sqrt(2))
    y = rand.uniform(0, 4)
    if (x**2 + y**2 - 4)**3 - 108*y**2 <= 0:
        count+=1
prob = count/iterations
print(prob)
print(4 * prob * 4 * 2*math.sqrt(2))
```

Simulation Results:

```
EN 379/COEN379HW1.py"
How many iterations would you like to run? 10000000
0.8330432
37.69923172744585
```

(Probability of being in curve, area of curve)

2. Let Ck represent the number of integers generated at least once by k independent calls to rand(n).
   Let us also assume that every integer from 0 to n-1 has an equal likelihood of being generated for each call.

   E|Ck| represents the expected value of the number of integers that will be generated at least once by k independent calls to rand(n).

   If we use I as a random variable for each of the values in Ck with the outcomes 1 (if occurs >= 1 time) or 0 (if occurs 0 times), we can then find a formula for E|Ck| using E|I|:

   E|Ck| = Summation (from 0 to n-1) of E|I|

   E|I| in this case is equal to the probability that an integer in Ck will be generated at least once in k amount of calls. To find this value, we subtract the probability that this integer does not occur at least once from the total value of 1, thus:

   E|I| = 1 - [(n - 1)/n]^k

   since (n - 1)/n is the chance that the value does not occur for each call.

   Then, because there are n total values:

   E|Ck| = n * (1 - [(n - 1)/n]^k)

   which gives us the final closed-form formula we are looking for.

   Simulation Code:

```
import random as rand
count = 0
n = int(input("Provide a number n representing the amount of
possible integers: "))
k = int(input("Provide a number k representing the amount of
iterations desired: "))
a = []
for i in range(0, n):
    a.append(0)
```

```
for i in range(0, k):
    r = rand.randint(0, n-1)
    a[r] = 1
for i in range(0, n):
    count += a[i]
print("Expected value: " + str(n*(1-((n-1)/n)**k)))
print("Simulation Value: " + str(count))
```

Simulation Results:

```
Provide a number n representing the amount of possible integers: 100
Provide a number k representing the amount of iterations desired: 100
Expected value: 63.396765872677086
Simulation Value: 66
```

```
Provide a number n representing the amount of possible integers: 1000
Provide a number k representing the amount of iterations desired: 1000
Expected value: 632.3045752290362
Simulation Value: 649
```

```
Provide a number n representing the amount of possible integers: 1000000
Provide a number k representing the amount of iterations desired: 1000000
Expected value: 632120.7427789335
Simulation Value: 631613
```

3. Scramble Search:
   In doing this, we have two procedures which need to be done:
   Firstly, we need to randomly swap around the elements of an array.
   Next, we need to search in this array for a specific value x, which may or may not exist.

   We randomly swap the elements of the array array by swapping each element with another element in the array at a random index, thus running in O(n).

   Next, we do a linear search within this scrambled array, which also runs in O(n).

   Altogether, because both operations run in linear time, the worst-case and expected running time are both still O(n), whether x is in the array or not.