Estimated Cycles of a Pots & Coins Game Using Simulation Methods

Georgia Tech - ISYE 6644

Evelyn Shen, Syefira Shofa

Abstract

This work provides an in-depth analysis of the expected length of the "Pots & Coins" game via analytical, statistical, and simulation techniques. The goal is to understand how many "cycles" the Pots & Coins game (described in detail in the section below) is expected to last. After an elementary analysis of the game by conducting a real-world test, discussing analytical concepts, and observing basic transition probabilities for the first few cycles, the game is simulated in Python and output analysis via independent replications is used to determine a confidence interval for the expected number of cycles the game will last. In addition, the game balance is examined to see if certain players can start the game with an advantage based on the order that they play.

Background

Description of Pots & Coins Game

As outlined in Project 14 in the ISYE-6644 Spring 2022 Projects list, the Pots & Coins game describes a scenario where two players, A and B, each start with 4 coins. The game includes a pot that initially contains 2 coins in it.

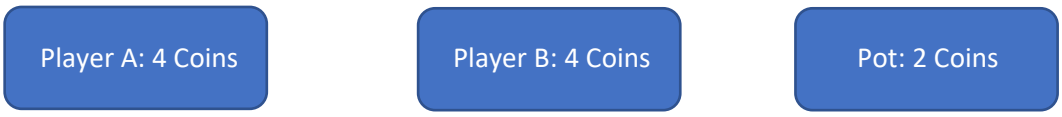
With player A as the starting player, the players each take turns tossing a 6-sided die. The following actions are taken based on the result of the 6-sided die:

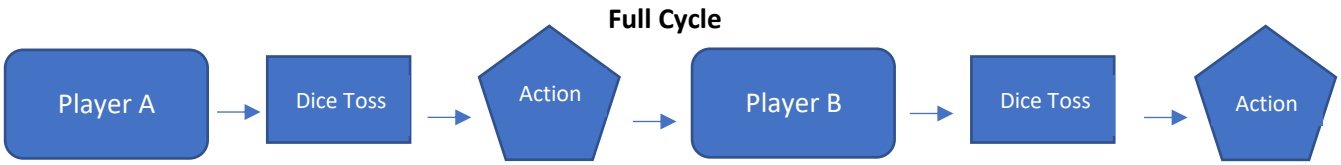
Die Toss Result	Action
1	Nothing
2	Player takes all coins in pot
3	Player takes half of the coins in the pot (rounded down)
4	Player puts a coin in the pot
5	Player puts a coin in the pot
6	Player puts a coin in the pot

The game ends when a player reaches a point where they have 0 coins and need to place a coin in the pots after rolling a 4, 5, or 6. This player is considered the game's loser.

A "cycle" is defined as both players completing their turns, with Player A going first. The exception is the final cycle: the final cycle always counts as 1 cycle even if only Player A is able to start their turn.

Game Start Conditions





Application Area

This project aims to determine how many cycles, on average, a simple game of Pots & Coins is expected to last. This is important because the length of a game is critical in predicting its popularity. A game should only take the amount of time necessary to deliver an intended experience[1]. Players often choose games that are worthwhile to play given how much time it takes to finish them. In general, if a game can deliver its intended experience in a shorter amount of time, it will be more popular. With a straightforward game such as Pots & Coins, an initial guess is that players should be able to complete the game in a relatively small number of cycles.

Generally, players will choose to participate in a game if there is motivation. Motivation comes from areas such as interest and competency[2]. Players are more attracted to games in which there is not a clear advantage towards one side. This paper will dissect aspects of the game in order to analyze the experience for both players.

Elementary Analysis

Before coding the simulation in Python, several methods can be used to perform observations of the game to gain a better understanding of potential expected cycle length.

Collecting Real-Life Data

Because Pots & Coins has a straightforward setup, playing the game organically several times is the simplest way to begin collecting data. Following the rules of the game, the outcome of 10 games was:

Game	1	2	3	4	5	6	7	8	9	10	**Mean**	**Median**
# Cycles	22	8	18	13	13	5 (min)	34 (max)	12	7	8	**14**	**12.5**

From these 10 real-life games alone, an observation can be made that there is a right-skewed distribution because the mean is greater than the median.

Minimum Number of Cycles

The probability that Pots & Coins ends at a specific cycle is mainly dependent on whether any player enters a cycle with 0 coins. Recall that a player "loses" the game when they have 0 coins and roll a 4, 5, or 6. Since both players start with the same number of coins, the minimum number of cycles the game will last can be determined by calculating the number of cycles until one player begins a cycle with 0 coins. This change in states can be represented with the matrix:

		Player loses (rolls a 4, 5, 6)	Player does not lose (rolls a 1, 2, 3)
# coins player has at start of cycle	0 coins	0.5	0.5
	1+ coins	0	1

To determine how long it would take any player to reach 0 cycles, a similar matrix can be used:

		-1 change in coin count	0+ change in coin count
# coins	any	0.5	0.5

At any cycle in the game, it is only possible for a player to decrease their coin count by 1. Because a player begins cycle 1 with 4 coins, the smallest number of coins a player can have at the start of cycle 2 is 3 coins, at start of cycle 3 is 2 coins, and at start of cycle 4 is 1 coin. Finally, if a player rolls a 4, 5, or 6 in cycle 4, they would start cycle 5 with 0 coins. Only at 5 cycles would it be possible for a player to lose by rolling another 4, 5, or 6. This corroborates the minimum number of cycles from the real-life exercise above. This concept is revisited and explored further in the Python simulation results.

Main Findings

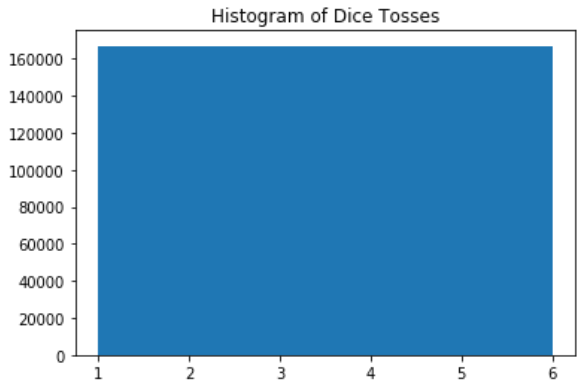
Dice Toss

To begin the simulation in Python, the game's dice toss is generated in Python:

```
def dice_toss():
    return math.ceil(6*np.random.uniform())
```

The formula to simulate the dice toss mechanism can be interpreted as "6 multiplied by a random number from a random distribution with range 0 to 1." To ensure the dice toss results follow a uniform distribution (i.e., that they represent a fair die), the dice toss is generated 1 million times. The plot of results should resemble a uniform distribution:

```
dice_toss_output= []
for i in range(0, 1000000):
  dice_toss_output.append(dice_toss())
plt.hist(dice_toss_output, bins=6)
```



The generated dice toss values are captured in a list so that they can also be tested for uniformity using the Chi-Square Goodness-of-Fit Test and for independence using a Runs Test "Above and Below the Mean."

Chi-Square Goodness-of-Fit Test for Uniformity

The null hypothesis for the Chi-Square Goodness-of-Fit Test is that the generated variables are uniform. To confirm this, the interval from 1 through 6, which are the possible values from the dice toss generator, can be split into *k* = 6 equal increments. If the variables are uniform, the probability that they will fall into each increment is 1/6. Thus, the expected number in each interval should be 1 million divided by 6. The differences of the observed and expected variables in each interval are squared and summed up, then divided by the expected number to form the Chi-Square test statistic, X_0^2:

$$X_0^2 = \frac{(O_i - E_i)^2}{E_i} \text{ , compare to } X_{\alpha,k-1}^2$$

$$\text{where } E_i = \frac{n}{k} \text{ and } O_i = observations\ in\ intervals$$

If the X_0^2 is less than or equal to $X_{\alpha,k-1}^2$, the null hypothesis of uniformity is not rejected. In Python:

```
# find the number of generated dice values in each bucket 1 through 6
(unique, counts) = np.unique(dice_toss_output, return_counts=True)
{x:y for x,y in zip(unique, counts)}

#perform Chi-Square Goodness of Fit Test ###
expected = [1000000/6] * 6
observed = list(counts)

### Chi-Square test statistic
stats.chisquare(f_obs=observed, f_exp=expected)
```

The unique generated values are [1 2 3 4 5 6] and the frequency for these values is [166622 166868 166347 166765 167580 165818]. The $X_{\alpha,k-1}^2$value for α = 0.05 and 5 degrees of freedom is 11.07 and the X_0^2 is 10.25. Since 10.25 is less than 11.07, the null hypothesis is not rejected, indicating that the generated variables are uniform.

Runs Test "Above and Below the Mean" for Independence

The null hypothesis for the Runs Test is that the generated variables are independent. To run a Runs Test, the mean for the expected output is calculated and each variable is assigned a positive or negative sign depending on whether it is above or below the mean, respectively. A "run" is defined as any sequence of observations with the same sign. The number of total runs (*n*), positive-sign variables (n_1), and negative-sign variables (n_2) are used to calculate the Z-statistic, Z_0:

$$Z_0 = \frac{B - E(B)}{\sqrt{Var(B)}}, \text{ compare to } z_{\alpha/2}$$

$$\text{where E(B)} = 2n_1n_2/\text{n} + 0.5, Var(B) = 2n_1n_2(2n_1n_2 - n)/n^2/(n-1)$$

If the absolute value of $Z_0$is less than or equal to $z_{\alpha/2}$, the null hypothesis of independence is not rejected. To implement in Python:

```
def runsTest(n, n_mean):
    runs, n1, n2 = 0, 0, 0
    # Checking for start of new run
    for i in range(len(n)):
        # no. of runs
        if (n[i] >= n_mean and n[i-1] < n_mean) or (n[i] < n_mean and n[i-1] >= n_mean):
            runs += 1
        # no. of positive values
        if (n[i]) >= n_mean:
            n1 += 1
        # no. of negative values
        else:
            n2 += 1
    exp_runs = 2*n1*n2/len(n)+0.5
    std_dev = math.sqrt(2*n1*n2*(2*n1*n2-len(n))/(len(n)**2)/(len(n)-1))
    z_stat = (runs-exp_runs)/std_dev
    return runs, n1, n2, len(n), exp_runs, std_dev, z_stat, abs(z_stat)
```

The $z_{\alpha/2}$ value for α = 0.05 is 1.96, and the test Z-statistic is 1.07. Since the absolute value of 1.07 is less than 1.96, the null hypothesis that the variables are independent is not rejected.

Game Cycle Simulation

Based on the rules of Pots & Coins, the Python function to track player actions after each dice toss can be codified as:

```
def player_cycle(player_coins, pot_coins):
    player_dice_toss = dice_toss()
    if player_dice_toss == 2:
        player_coins = player_coins + pot_coins
        pot_coins = 0
    elif player_dice_toss == 3:
        half_pot_coins = math.floor(pot_coins/2)
        player_coins = player_coins + half_pot_coins
        pot_coins = pot_coins - half_pot_coins
    elif player_dice_toss in [4,5,6]:
        player_coins = player_coins - 1
        pot_coins = pot_coins + 1
    return player_coins, pot_coins
```

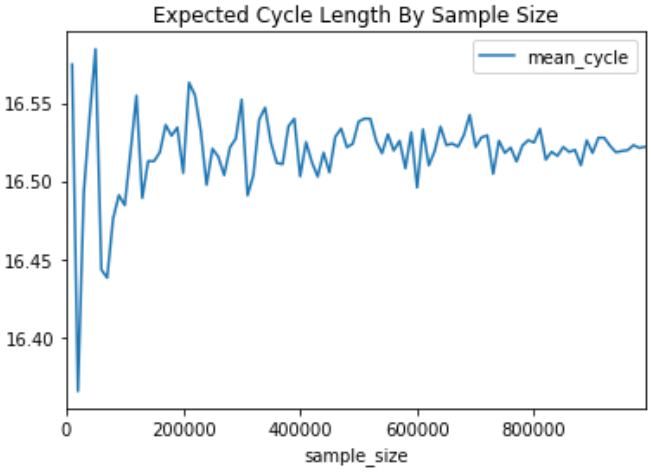
The following function tracks each game cycle:

```
def game_cycle(cycles, player_a_coins, player_b_coins, pot_coins):
    #player a goes first
    player_a_coins, pot_coins = player_cycle(player_a_coins, pot_coins)
    #player b goes second
    if player_a_coins >= 0:
        player_b_coins, pot_coins = player_cycle(player_b_coins, pot_coins)
    #add to cycle
    if player_a_coins >=0 or player_b_coins >= 0:
        cycles += 1
    #if game ends do nothing with pot coins
    if player_a_coins < 0 or player_b_coins < 0:
        pot_coins = pot_coins - 1
    return cycles, player_a_coins, player_b_coins, pot_coins
```

One million games are simulated and the outputs of each run are stored in vectors (for brevity, initial definitions of these vectors is shown in the Appendix):

```
for i in range(0,1000000):
    cycles = 0
    player_a_coins = 4
    player_b_coins = 4
    pot_coins = 2
    while player_a_coins >= 0 and player_b_coins >= 0:
        old_player_a_coins = player_a_coins
        old_player_b_coins = player_b_coins
        cycles, player_a_coins, player_b_coins, pot_coins = game_cycle(cycles,
player_a_coins, player_b_coins, pot_coins)
        player_a_coins_each_round_vector.append(player_a_coins)
        player_b_coins_each_round_vector.append(player_b_coins)
        pot_coins_each_round_vector.append(pot_coins)
        if old_player_a_coins not in frequency_dictionary_player_a:
            frequency_dictionary_player_a[old_player_a_coins] = [player_a_coins]
        else:
            frequency_dictionary_player_a[old_player_a_coins].append(player_a_coins)
        if old_player_b_coins not in frequency_dictionary_player_b:
            frequency_dictionary_player_b[old_player_b_coins] = [player_b_coins]
        else:
            frequency_dictionary_player_b[old_player_b_coins].append(player_b_coins)
    cycle_vector.append(cycles)
    player_a_coins_vector.append(player_a_coins)
    player_b_coins_vector.append(player_b_coins)
    pot_coins_vector.append(pot_coins)
```

After 1 million runs were completed in Python (see Appendix for full code), observations were made about the cycle length outputs from each run using common statistical functions and packages.



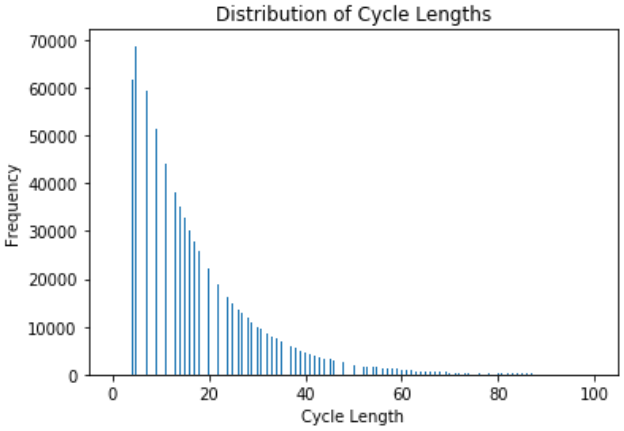
The chart to the left showcases the mean cycle length per sample size. Sample size in this case refers to the number of games simulated. By mean square convergence[3]:

$$\lim_{n\to\infty} E[(X_n - X)^2] = 0$$

In other words, as the number of games simulated increases, the mean cycle length will converge to the true mean. The chart indicates that 1 million simulated games is satisfactory to estimate the true cycle length since convergence can be observed as sample size increases.

Geometric Distribution

After 1 million simulations, the plot to the right illustrates that cycle lengths resemble a geometric distribution. A geometric distribution is a type of discrete probability distribution that represents the probability of the number of successive failures before a success is obtained in a Bernoulli trial. A Bernoulli trial is an experiment with only two outcomes: success and failure. In the case of this game, a player losing is considered a "success". Note that due to the nature of the game, the probability of success does change from trial to trial. The mean cycle length can be defined as:



$$E[X] = \mu = \frac{1 - \delta p}{p(1 - \delta)}$$

where p is the probability of success, δ is the dependency coefficient[4]

At any given cycle, the Pots & Coins game changes its probability of success due to the ever-changing amount of coins in each player's hand as the game progresses. Because a player can only lose if the player has 0 coins in a certain cycle and has to put a coin back in the pot, the probability of a player losing will be

$$P(Player\ losing) = \begin{cases} 0.5\ if\ player\ has\ 0\ coins \\ 0\ if\ player\ has\ greater\ than\ 0\ coins \end{cases}$$

Also note that the expected number of cycles it will take the player to get to 0 coins is dependent on the amount of coins the player has in their hands. The data frame below illustrates the mode number of cycles it will take both players to reach 0 coins based on the number of coins currently in their hands. The mode was the preferred metric to use here as the mean was too heavily influenced by outliers[5]. The mode was calculated off of a truncated list of 1 million cycle observations for both players as the full list was too computationally expensive (the term cycle here refers to all cycles within each game simulated for both players).

Coins in Hand	Mode Cycles Until 0 Coins in Hand: Player A	Mode Cycles Until 0 Coins in Hand: Player B
1	1	1
2	2	2
3	3	3
4	5	5
5	7	6
6	11	11
7	14	12
8	15	10
9	12	8
10	6	6

Player Experience

It is important to know the expected game experience for both players. Game balance is important so that the game does not give an automatic advantage or disadvantage to certain players[6]. Through simulation, probabilities of going from a certain number of coins to the next in a cycle were calculated to determine the differences between players. The matrices on the bottom showcase these probabilities.

	-1	0	1	2	3	4	5	6	7	8	9	10
0	50.08	25.74	7.40	6.05	3.70	2.69	1.62	0.93	0.65	0.49	0.37	0.28
1	NaN	50.01	26.28	7.91	6.00	3.69	2.57	1.20	0.85	0.67	0.55	0.26
2	NaN	NaN	50.08	26.61	8.20	5.98	3.99	1.92	1.26	1.20	0.52	0.25
3	NaN	NaN	NaN	49.99	26.74	8.22	7.09	3.19	2.91	1.05	0.52	0.28
4	NaN	NaN	NaN	NaN	49.98	24.76	11.85	8.99	2.37	1.11	0.63	0.32
5	NaN	NaN	NaN	NaN	NaN	49.97	29.32	10.64	5.76	2.40	1.24	0.67
6	NaN	NaN	NaN	NaN	NaN	NaN	50.00	32.20	10.57	4.23	1.98	1.02
7	NaN	NaN	NaN	NaN	NaN	NaN	NaN	50.07	33.55	11.13	3.17	2.06
8	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	50.01	36.37	10.40	3.22
9	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	49.96	41.12	8.92
10	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	50.01	49.99

	-1	0	1	2	3	4	5	6	7	8	9	10
0	49.61	26.38	7.33	6.05	3.76	2.63	1.57	0.96	0.69	0.47	0.33	0.24
1	NaN	49.61	26.87	7.88	6.06	3.64	2.54	1.23	0.87	0.57	0.44	0.31
2	NaN	NaN	49.70	27.19	8.32	5.95	3.87	2.02	1.11	0.88	0.67	0.28
3	NaN	NaN	NaN	49.63	27.61	8.74	6.69	3.11	1.79	1.58	0.56	0.27
4	NaN	NaN	NaN	NaN	49.56	28.45	10.79	4.87	4.34	1.11	0.59	0.30
5	NaN	NaN	NaN	NaN	NaN	49.11	31.04	10.07	5.31	2.44	1.30	0.73
6	NaN	NaN	NaN	NaN	NaN	NaN	48.33	33.55	10.33	4.47	2.19	1.14
7	NaN	NaN	NaN	NaN	NaN	NaN	NaN	47.67	36.70	10.67	3.03	1.93
8	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	44.93	42.76	9.41	2.90
9	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	42.17	50.35	7.48
10	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	24.90	75.10

The charts to the left display the probability (calculated via simulating 1 million games) of going from x coins to y coins in a cycle. The columns represent the number of coins a player starts with and the rows represent the number of coins a player ends with in a cycle. The top chart showcases the probability matrix for Player A whereas the bottom chart showcases the probability matrix for Player B.

While differences in probabilities between players are noticeable, it was determined to be insignificant. Note that in games where Player A or Player B wins, they are both estimated to have 3 coins in hand when the other player loses. Also, via simulation, Player A is estimated to lose 49.8% of the time and Player B is estimated to lose 50.2% of the time. The difference in chances to lose is miniscule enough that players won't have a strong preference between going first or second.

Simulation Results

After 1 million simulations of games, the mean number of cycles per game was 17.53 cycles. In other words, one of the players can expect to lose at the 17th or 18th cycle. The minimum number of cycles from the simulated runs was 5 cycles, which supports the elementary analysis. The maximum number of cycles from the simulated runs was 182 cycles. This means that it's possible that a game of Pots & Coins could last for quite a long time!

Output Analysis

Pots & Coins can be considered a finite-horizon simulation where the termination occurs when a player loses. To obtain a confidence interval for the expected mean number of cycles, the method of independent replications ("IR") can be used. IR estimates the variance of means by conducting *r* independent simulation runs, each with *m* observations.

The estimator for the expected value of sample means is:

$$Z_r^{--} = \frac{1}{r}\sum_{i=1}^{r} Z_i$$

The estimator for the sample variance of sample means is:

$$S_Z^2 = \frac{1}{r-1}\sum_{i=1}^{r} (Z_i - Z_r^{--})^2$$

A reasonable estimator for the variance of means is S_Z^2/r and the approximate IR 100(1-α)% two-sided confidence interval for the expected value of sample means is:

$$Z_r^{--} \pm t_{\alpha/2,r-1}\sqrt{S_Z^2/r}$$

To implement in Python, the same simulation (m=1000000) is run at 10 different seed values (r=10). The simulation code from earlier is run for each new seed value and the means are stored in a list. Those means are then evaluated using the IR estimators, which are implemented using this code:

```
def z_samp_var(list_of_means):
  n = len(list_of_means)
  z_mean = sum(list_of_means) / n
  deviations = [(x - z_mean) ** 2 for x in list_of_means]
  z_sample_var = sum(deviations) / (n-1)
  return z_mean, z_sample_var, n

z_samp_var(cycle_length_means)

### student-t value at 95% confidence and r-1 degrees of freedom ###
t = stats.t.ppf(q=1-0.025,df=(r-1))
print(cycle_length_means)

### half length of confidence interval ###
half_length = t*math.sqrt(z_samp_var(cycle_length_means)[1]/r)
```

The outputs are Z_r^{--} =17.54 and S_Z^2 = 5.123e-06. The 95% confidence interval for the expected value of the sample mean is [17.5343 , 17.5375].

Conclusions

From simulation, Pots & Coins proved to be a fair game that should attract many players. In a game between two players, it was determined that neither side held an advantage and, thus, had equal chances to win. It was also determined that Pots & Coins can be expected to last for an average of approximately 17.5 cycles. Because the estimated cycle length is reasonable, players would not be deterred from playing. Nevertheless, cycle lengths have a long tail, so two players who expect a relatively short game may be surprised to end up in that long tail and encounter a game that lasts for hundreds of cycles!

There are many methods that can be used to further iterate on this project. For example, a similar simulation using identical game rules could be performed in other simulation software systems such as ARENA. The results from an ARENA and Python simulations could be used to calculate the variance across systems and determine which simulation performs better. The two systems can be compared using two-sample confidence intervals for the difference in two normal means. Various adjustments can be made to the simulations to see if results are materially affected. More robust and complex analytical methods could also be used to cross-verify the simulation results. In general, analytical and simulation methods can be used together to make an analysis more sound.

Appendix

```
#import libaries
import numpy as np
import math
import matplotlib.pyplot as plt
import collections
import pandas as pd
from matplotlib.ticker import PercentFormatter
import random
import scipy.stats as stats
import statistics
#create seed value
np.random.seed(1)
#generates 6 sided dice toss
def dice_toss():
    return math.ceil(6*np.random.uniform())
#Simulate dice toss 1 million times and make sure it's a uniform distribution
dice_toss_output= []
for i in range(0, 1000000):
  dice toss output.append(dice toss())
plt.hist(dice toss output, bins=6)
plt.title("Histogram of Dice Tosses")
plt.xlabel('Dice Toss Result')
plt.ylabel('Frequency')
plt.show()

## Check that generated dice toss are uniform with the Chi-Square Goodness-of-Test for Uniformity
### find the number of generated dice values in each bucket 1 through 6 ###
(unique, counts) = np.unique(dice_toss_output, return_counts=True)
{x:y for x,y in zip(unique, counts)}
print("The unique generated values are",(unique, counts)[0])
print("The frequency for these values is",(unique, counts)[1])

### perform Chi-Square Goodness of Fit Test ###
expected = [1000000/6] * 6
observed = list(counts)

### Chi-Square test statistic
print(stats.chisquare(f_obs=observed, f_exp=expected))
print("The Chi-Square test statistic for a confidence level of 95% is",stats.chisquare(f_obs=observed,
f_exp=expected)[0])
print("Since",round(stats.chisquare(f_obs=observed, f_exp=expected)[0],2), "is less
than",round(stats.chi2.ppf(1-.05, df=len(list(unique))-1),2),"we accept the null hypothesis that the
PRNs are uniform.")

# Check that generated dice toss are uniform with Runs Test "Above and Below the Mean" for
Independence
def runsTest(n, n_mean):
    runs, n1, n2 = 0, 0, 0
    # Checking for start of new run
    for i in range(len(n)):
        # no. of runs
        if (n[i] >= n_mean and n[i-1] < n_mean) or (n[i] < n_mean and n[i-1] >= n_mean):
            runs += 1
        # no. of positive values
        if (n[i]) >= n_mean:
            n1 += 1
        # no. of negative values
        else:
            n2 += 1
    exp_runs = 2*n1*n2/len(n)+0.5
    std_dev = math.sqrt(2*n1*n2*(2*n1*n2-len(n))/(len(n)**2)/(len(n)-1))
    z_stat = (runs-exp_runs)/std_dev
    return runs, n1, n2, len(n), exp_runs, std_dev, z_stat, abs(z_stat)
print(runsTest(dice_toss_output, statistics.mean(dice_toss_output)))
print("The z-quantile for a confidence level of 95% is 1.96")
print("The test Z-statistic is", runsTest(dice_toss_output, statistics.mean(dice_toss_output))[6])
print("Since the absolute value of the test Z-statistic",round(runsTest(dice_toss_output,
statistics.mean(dice_toss_output))[7],2),"is less than 1.96, we accept the null hypothesis that the
PRNs are independent.")
```

```
#function for a player's turn
def player_cycle(player_coins, pot_coins):
    player_dice_toss = dice_toss()
    if player_dice_toss == 2:
        player_coins = player_coins + pot_coins
        pot_coins = 0
    elif player_dice_toss == 3:
        half_pot_coins = math.floor(pot_coins/2)
        player_coins = player_coins + half_pot_coins
        pot_coins = pot_coins - half_pot_coins
    elif player_dice_toss in [4,5,6]:
        player_coins = player_coins - 1
        pot_coins = pot_coins + 1
    return player_coins, pot_coins


def game cycle(cycles, player a coins, player b coins, pot coins):
    #player a goes first
    player_a_coins, pot_coins = player_cycle(player_a_coins, pot_coins)
    #player b goes second
    if player_a_coins >= 0:
        player_b_coins, pot_coins = player_cycle(player_b_coins, pot_coins)
    #add to cycle
    if player_a_coins >=0 or player_b_coins >= 0:
        cycles += 1
    #if game ends do nothing with pot coins
    if player_a_coins < 0 or player_b_coins < 0:
        pot_coins = pot_coins - 1
    return cycles, player a coins, player b coins, pot coins


#simulate 1 million games and store simulation metrics in vectors as needed
cycle_vector = []
player_a_coins_vector = []
player_b_coins_vector = []
pot_coins_vector = []
player_a_start_vector = []
frequency_dictionary_player_a = dict()
frequency_dictionary_player_b = dict()
player a coins each round vector = []
player_b_coins_each_round_vector = []
pot_coins_each_round_vector = []
for i in range(0,1000000):
    cycles = 0
    player_a_coins = 4
    player_b_coins = 4
    pot_coins = 2
    while player_a_coins >= 0 and player_b_coins >= 0:
        old_player_a_coins = player_a_coins
        old_player_b_coins = player_b_coins
        cycles, player_a_coins, player_b_coins, pot_coins = game_cycle(cycles, player_a_coins,
player_b_coins, pot_coins)
        player_a_coins_each_round_vector.append(player_a_coins)
        player_b_coins_each_round_vector.append(player_b_coins)
        pot_coins_each_round_vector.append(pot_coins)
        if old_player_a_coins not in frequency_dictionary_player_a:
            frequency_dictionary_player_a[old_player_a_coins] = [player_a_coins]
        else:
            frequency_dictionary_player_a[old_player_a_coins].append(player_a_coins)
        if old_player_b_coins not in frequency_dictionary_player_b:
            frequency_dictionary_player_b[old_player_b_coins] = [player_b_coins]
        else:
            frequency_dictionary_player_b[old_player_b_coins].append(player_b_coins)
    cycle_vector.append(cycles)
    player_a_coins_vector.append(player_a_coins)
    player_b_coins_vector.append(player_b_coins)
    pot_coins_vector.append(pot_coins)
```

```
#calculate mode cycles for player A to get from x coins to 0 coins
#need to truncate vector as the full vector is too computationally expensive
player_a_coins_each_round_vector_truncated = player_a_coins_each_round_vector[0:1000000]
zerolist = np.array([i for i, x in enumerate(player_a_coins_each_round_vector_truncated) if x == 0])
zero_frequency_player_a = dict()
for j in range(1,11):
  coinlist = np.array([i for i, x in enumerate(player_a_coins_each_round_vector_truncated) if x == j])
  for i in coinlist:
    if j not in zero_frequency_player_a:
      zero_frequency_player_a[j] = [zerolist[np.argmax(zerolist > i)]-i]
    else:
      if zerolist[np.argmax(zerolist > i)] > i:
        zero_frequency_player_a[j].append(zerolist[np.argmax(zerolist > i)]-i)

#create player A mode df
player_a_df = pd.DataFrame({k:max(set(v), key=v.count) for k,v in
zero_frequency_player_a.items()}.items(), columns=['Coins in Hand', 'Mode Cycles Until 0 Coins in
Hand: Player A'])

#calculate mode cycles for player A to get from x coins to 0 coins
#need to truncate vector as the full vector is too computationally expensive
player b coins each round vector truncated = player b coins each round vector[0:1000000]
zerolist = np.array([i for i, x in enumerate(player_b_coins_each_round_vector_truncated) if x == 0])
zero_frequency_player_b = dict()
for j in range(1,11):
  coinlist = np.array([i for i, x in enumerate(player_b_coins_each_round_vector_truncated) if x == j])
  for i in coinlist:
    if j not in zero_frequency_player_b:
      zero_frequency_player_b[j] = [zerolist[np.argmax(zerolist > i)]-i]
    else:
      if zerolist[np.argmax(zerolist > i)] > i:
        zero frequency player b[j].append(zerolist[np.argmax(zerolist > i)]-i)

#create player B mode df
player_a_df = pd.DataFrame({k:max(set(v), key=v.count) for k,v in
zero_frequency_player_a.items()}.items(), columns=['Coins in Hand', 'Mode Cycles Until 0 Coins in
Hand: Player A'])
player_b_df = pd.DataFrame({k:max(set(v), key=v.count) for k,v in
zero_frequency_player_b.items()}.items(), columns=['Coins in Hand', 'Mode Cycles Until 0 Coins in
Hand: Player B'])
#merge the 2 df together
pd.merge(player_a_df, player_b_df, how='inner', on = 'Coins in Hand').style.hide_index()

#create dictionary of frequencies to get from x coins to y coins
frequency_dictionary_player_a =
dict(collections.OrderedDict(sorted(frequency_dictionary_player_a.items())))
frequency_dictionary_player_b =
dict(collections.OrderedDict(sorted(frequency_dictionary_player_b.items())))
percent_player_a = dict()
percent_player_b = dict()
#populate dictionary
for i in frequency_dictionary_player_a.keys():
    percent_player_a[i] = dict(sorted(collections.Counter(frequency_dictionary_player_a[i]).items()))
    for j in percent_player_a[i]:
        percent_player_a[i][j] /= len(frequency_dictionary_player_a[i])
        percent_player_a[i][j] *= 100
for i in frequency_dictionary_player_b.keys():
    percent_player_b[i] = dict(sorted(collections.Counter(frequency_dictionary_player_b[i]).items()))
    for j in percent_player_b[i]:
        percent_player_b[i][j] /= len(frequency_dictionary_player_b[i])
        percent_player_b[i][j] *= 100
#output Player A matric where rows are starting coins in a cycle and columns are ending coins
percent_player_a_df = pd.DataFrame(percent_player_a.items())[1].apply(pd.Series).round(decimals = 2)
percent_player_a_df

#output Player B matric where rows are starting coins in a cycle and columns are ending coins
percent_player_b_df = pd.DataFrame(percent_player_b.items())[1].apply(pd.Series).round(decimals = 2)
percent_player_b_df
#Histogram for cycle lengths
plt.hist(cycle_vector, bins = 'auto', range = (0, 100))
plt.hist(cycle_vector, bins = 'auto', range = (0, 100))
plt.xlabel('Cycle Length')
plt.ylabel('Frequency')
plt.title("Distribution of Cycle Lengths")
plt.show()
```

```
#randomly select x games (sample sizes) and look to see if mean converges
sample = []
mean = []
for i in range(0,1000000, 10000):
    sample.append(i)
    mean.append(np.mean(random.sample(cycle_vector, i)))
data = {'sample_size': sample, 'mean_cycle': mean}

# Create DataFrame
df = pd.DataFrame(data)
df
#plot convergende
df.plot.line(x='sample_size', y='mean_cycle', title='Expected Cycle Length By Sample Size')

#Probability Player B loses
len([i for i in player_b_coins_vector if i == -1])/len(player_b_coins_vector)

#Probability Player A loses
len([i for i in player_a_coins_vector if i == -1])/len(player_a_coins_vector)

#get the expected value for cycles in a game and min/max from all runs
np.mean(cycle_vector), np.min(cycle_vector), np.max(cycle_vector)

#get the expected value for Player A's ending coins is Player A wins
np.mean(player a coins vector)

#get the expected value for Player B's ending coins is Player B wins
np.mean(player_b_coins_vector)
```

```
### Output Analysis: use IR to create confidence interval for the expected value of sample mean of
observations ###
r = 10
seed values = range(2,r+2)

### run simulation at different seeds ####
cycle_length_means = []

for s in seed_values:
  np.random.seed(s)
  for i in range(0,1000000):
    cycles = 0
    player_a_coins = 4
    player_b_coins = 4
    pot_coins = 2
    while player_a_coins >= 0 and player_b_coins >= 0:
        old_player_a_coins = player_a_coins
        old_player_b_coins = player_b_coins
        cycles, player_a_coins, player_b_coins, pot_coins = game_cycle(cycles, player_a_coins,
player_b_coins, pot_coins)
        player a coins each round vector.append(player a coins)
        player b coins each round vector.append(player b coins)
        pot_coins_each_round_vector.append(pot_coins)
        if old_player_a_coins not in frequency_dictionary_player_a:
            frequency_dictionary_player_a[old_player_a_coins] = [player_a_coins]
        else:
            frequency_dictionary_player_a[old_player_a_coins].append(player_a_coins)
        if old_player_b_coins not in frequency_dictionary_player_b:
            frequency_dictionary_player_b[old_player_b_coins] = [player_b_coins]
        else:
            frequency_dictionary_player_b[old_player_b_coins].append(player_b_coins)
    cycle_vector.append(cycles)
    player_a_coins_vector.append(player_a_coins)
    player b coins vector.append(player b coins)
    pot_coins_vector.append(pot_coins)
  cycle_length_means.append(np.mean(cycle_vector))

### define expected value of sample means and sample variance of sample means ###
def z_samp_var(list_of_means):
  n = len(list_of_means)
  z_mean = sum(list_of_means) / n
  deviations = [(x - z_mean) ** 2 for x in list_of_means]
  z_sample_var = sum(deviations) / (n-1)
  return z_mean, z_sample_var, n

z_samp_var(cycle_length_means)

### student-t value at 95% confidence and r-1 degrees of freedom ###
t = stats.t.ppf(q=1-0.025,df=(r-1))
print(cycle_length_means)

### half length of confidence interval ###
half_length = t*math.sqrt(z_samp_var(cycle_length_means)[1]/r)

print("The expected value of sample means is",z_samp_var(cycle_length_means)[0])
print("The sample variance of sample means is",z samp var(cycle length means)[1])
print("The 95% confidence interval for the expected value of the sample mean is
(",round(z_samp_var(cycle_length_means)[0]-
half_length,5),",",round(z_samp_var(cycle_length_means)[0]+half_length,5),")")
```

Standard Normal Table

Z	0.00	0.01	0.02	0.03	0.04	0.05	0.06	0.07	0.08	0.09
0.0	0.5000	0.5040	0.5080	0.5120	0.5160	0.5199	0.5239	0.5279	0.5319	0.5359
0.1	0.5398	0.5438	0.5478	0.5517	0.5557	0.5596	0.5636	0.5675	0.5714	0.5753
0.2	0.5793	0.5832	0.5871	0.5910	0.5948	0.5987	0.6026	0.6064	0.6103	0.6141
0.3	0.6179	0.6217	0.6255	0.6293	0.6331	0.6368	0.6406	0.6443	0.6480	0.6517
0.4	0.6554	0.6591	0.6628	0.6664	0.6700	0.6736	0.6772	0.6808	0.6844	0.6879
0.5	0.6915	0.6950	0.6985	0.7019	0.7054	0.7088	0.7123	0.7157	0.7190	0.7224
0.6	0.7257	0.7291	0.7324	0.7357	0.7389	0.7422	0.7454	0.7486	0.7517	0.7549
0.7	0.7580	0.7611	0.7642	0.7673	0.7704	0.7734	0.7764	0.7794	0.7823	0.7852
0.8	0.7881	0.7910	0.7939	0.7967	0.7995	0.8023	0.8051	0.8078	0.8106	0.8133
0.9	0.8159	0.8186	0.8212	0.8238	0.8264	0.8289	0.8315	0.8340	0.8365	0.8389
1.0	0.8413	0.8438	0.8461	0.8485	0.8508	0.8531	0.8554	0.8577	0.8599	0.8621
1.1	0.8643	0.8665	0.8686	0.8708	0.8729	0.8749	0.8770	0.8790	0.8810	0.8830
1.2	0.8849	0.8869	0.8888	0.8907	0.8925	0.8944	0.8962	0.8980	0.8997	0.9015
1.3	0.9032	0.9049	0.9066	0.9082	0.9099	0.9115	0.9031	0.9147	0.9162	0.9177
1.4	0.9192	0.9207	0.9222	0.9236	0.9251	0.9265	0.9279	0.9292	0.9306	0.9319
1.5	0.9332	0.9345	0.9357	0.9370	0.9382	0.9394	0.9406	0.9418	0.9429	0.9441
1.6	0.9452	0.9463	0.9474	0.9484	0.9495	0.9505	0.9515	0.9525	0.9535	0.9545
1.7	0.9554	0.9564	0.9573	0.9582	0.9591	0.9599	0.9608	0.9616	0.9625	0.9633
1.8	0.9641	0.9649	0.9656	0.9664	0.9671	0.9678	0.9686	0.9693	0.9699	0.9706
1.9	0.9713	0.9719	0.9726	0.9732	0.9738	0.9744	0.9750	0.9756	0.9761	0.9767
2.0	0.9772	0.9778	0.9783	0.9788	0.9793	0.9798	0.9803	0.9808	0.9812	0.9817
2.1	0.9821	0.9826	0.9830	0.9834	0.9838	0.9842	0.9846	0.9850	0.9854	0.9857
2.2	0.9861	0.9864	0.9868	0.9871	0.9875	0.9878	0.9881	0.9884	0.9887	0.9890
2.3	0.9893	0.9896	0.9898	0.9901	0.9904	0.9906	0.9909	0.9911	0.9913	0.9916
2.4	0.9918	0.9920	0.9922	0.9924	0.9927	0.9929	0.9931	0.9932	0.9934	0.9936
2.5	0.9938	0.9940	0.9941	0.9943	0.9945	0.9946	0.9948	0.9949	0.9951	0.9952
2.6	0.9953	0.9955	0.9956	0.9957	0.9958	0.9960	0.9961	0.9962	0.9963	0.9964
2.7	0.9965	0.9966	0.9967	0.9968	0.9969	0.9970	0.9971	0.9972	0.9973	0.9974
2.8	0.9974	0.9975	0.9976	0.9977	0.9977	0.9978	0.9979	0.9979	0.9980	0.9981
2.9	0.9981	0.9982	0.9982	0.9983	0.9984	0.9984	0.9985	0.9985	0.9986	0.9986

Percentage Points of the Chi-Square Distribution

Degrees of Freedom	Probability of a larger value of x^2								
	0.99	**0.95**	**0.90**	**0.75**	**0.50**	**0.25**	**0.10**	**0.05**	**0.01**
1	0.000	0.004	0.016	0.102	0.455	1.32	2.71	3.84	6.63
2	0.020	0.103	0.211	0.575	1.386	2.77	4.61	5.99	9.21
3	0.115	0.352	0.584	1.212	2.366	4.11	6.25	7.81	11.34
4	0.297	0.711	1.064	1.923	3.357	5.39	7.78	9.49	13.28
5	0.554	1.145	1.610	2.675	4.351	6.63	9.24	11.07	15.09
6	0.872	1.635	2.204	3.455	5.348	7.84	10.64	12.59	16.81
7	1.239	2.167	2.833	4.255	6.346	9.04	12.02	14.07	18.48
8	1.647	2.733	3.490	5.071	7.344	10.22	13.36	15.51	20.09
9	2.088	3.325	4.168	5.899	8.343	11.39	14.68	16.92	21.67
10	2.558	3.940	4.865	6.737	9.342	12.55	15.99	18.31	23.21
11	3.053	4.575	5.578	7.584	10.341	13.70	17.28	19.68	24.72
12	3.571	5.226	6.304	8.438	11.340	14.85	18.55	21.03	26.22
13	4.107	5.892	7.042	9.299	12.340	15.98	19.81	22.36	27.69
14	4.660	6.571	7.790	10.165	13.339	17.12	21.06	23.68	29.14
15	5.229	7.261	8.547	11.037	14.339	18.25	22.31	25.00	30.58
16	5.812	7.962	9.312	11.912	15.338	19.37	23.54	26.30	32.00
17	6.408	8.672	10.085	12.792	16.338	20.49	24.77	27.59	33.41
18	7.015	9.390	10.865	13.675	17.338	21.60	25.99	28.87	34.80
19	7.633	10.117	11.651	14.562	18.338	22.72	27.20	30.14	36.19
20	8.260	10.851	12.443	15.452	19.337	23.83	28.41	31.41	37.57
22	9.542	12.338	14.041	17.240	21.337	26.04	30.81	33.92	40.29
24	10.856	13.848	15.659	19.037	23.337	28.24	33.20	36.42	42.98
26	12.198	15.379	17.292	20.843	25.336	30.43	35.56	38.89	45.64
28	13.565	16.928	18.939	22.657	27.336	32.62	37.92	41.34	48.28
30	14.953	18.493	20.599	24.478	29.336	34.80	40.26	43.77	50.89
40	22.164	26.509	29.051	33.660	39.335	45.62	51.80	55.76	63.69
50	27.707	34.764	37.689	42.942	49.335	56.33	63.17	67.50	76.15
60	37.485	43.188	46.459	52.294	59.335	66.98	74.40	79.08	88.38

TABLE of CRITICAL VALUES for STUDENT'S *t* DISTRIBUTIONS

Column headings denote probabilities (α) ***above*** tabulated values.

d.f.	0.40	0.25	0.10	0.05	0.04	0.025	0.02	0.01	0.005	0.0025	0.001	0.0005
1	0.325	1.000	3.078	6.314	7.916	12.706	15.894	31.821	63.656	127.321	318.289	636.578
2	0.289	0.816	1.886	2.920	3.320	4.303	4.849	6.965	9.925	14.089	22.328	31.600
3	0.277	0.765	1.638	2.353	2.605	3.182	3.482	4.541	5.841	7.453	10.214	12.924
4	0.271	0.741	1.533	2.132	2.333	2.776	2.999	3.747	4.604	5.598	7.173	8.610
5	0.267	0.727	1.476	2.015	2.191	2.571	2.757	3.365	4.032	4.773	5.894	6.869
6	0.265	0.718	1.440	1.943	2.104	2.447	2.612	3.143	3.707	4.317	5.208	5.959
7	0.263	0.711	1.415	1.895	2.046	2.365	2.517	2.998	3.499	4.029	4.785	5.408
8	0.262	0.706	1.397	1.860	2.004	2.306	2.449	2.896	3.355	3.833	4.501	5.041
9	0.261	0.703	1.383	1.833	1.973	2.262	2.398	2.821	3.250	3.690	4.297	4.781
10	0.260	0.700	1.372	1.812	1.948	2.228	2.359	2.764	3.169	3.581	4.144	4.587
11	0.260	0.697	1.363	1.796	1.928	2.201	2.328	2.718	3.106	3.497	4.025	4.437
12	0.259	0.695	1.356	1.782	1.912	2.179	2.303	2.681	3.055	3.428	3.930	4.318
13	0.259	0.694	1.350	1.771	1.899	2.160	2.282	2.650	3.012	3.372	3.852	4.221
14	0.258	0.692	1.345	1.761	1.887	2.145	2.264	2.624	2.977	3.326	3.787	4.140
15	0.258	0.691	1.341	1.753	1.878	2.131	2.249	2.602	2.947	3.286	3.733	4.073
16	0.258	0.690	1.337	1.746	1.869	2.120	2.235	2.583	2.921	3.252	3.686	4.015
17	0.257	0.689	1.333	1.740	1.862	2.110	2.224	2.567	2.898	3.222	3.646	3.965
18	0.257	0.688	1.330	1.734	1.855	2.101	2.214	2.552	2.878	3.197	3.610	3.922
19	0.257	0.688	1.328	1.729	1.850	2.093	2.205	2.539	2.861	3.174	3.579	3.883
20	0.257	0.687	1.325	1.725	1.844	2.086	2.197	2.528	2.845	3.153	3.552	3.850
21	0.257	0.686	1.323	1.721	1.840	2.080	2.189	2.518	2.831	3.135	3.527	3.819
22	0.256	0.686	1.321	1.717	1.835	2.074	2.183	2.508	2.819	3.119	3.505	3.792
23	0.256	0.685	1.319	1.714	1.832	2.069	2.177	2.500	2.807	3.104	3.485	3.768
24	0.256	0.685	1.318	1.711	1.828	2.064	2.172	2.492	2.797	3.091	3.467	3.745
25	0.256	0.684	1.316	1.708	1.825	2.060	2.167	2.485	2.787	3.078	3.450	3.725
26	0.256	0.684	1.315	1.706	1.822	2.056	2.162	2.479	2.779	3.067	3.435	3.707
27	0.256	0.684	1.314	1.703	1.819	2.052	2.158	2.473	2.771	3.057	3.421	3.689
28	0.256	0.683	1.313	1.701	1.817	2.048	2.154	2.467	2.763	3.047	3.408	3.674
29	0.256	0.683	1.311	1.699	1.814	2.045	2.150	2.462	2.756	3.038	3.396	3.660
30	0.256	0.683	1.310	1.697	1.812	2.042	2.147	2.457	2.750	3.030	3.385	3.646
31	0.256	0.682	1.309	1.696	1.810	2.040	2.144	2.453	2.744	3.022	3.375	3.633
32	0.255	0.682	1.309	1.694	1.808	2.037	2.141	2.449	2.738	3.015	3.365	3.622
33	0.255	0.682	1.308	1.692	1.806	2.035	2.138	2.445	2.733	3.008	3.356	3.611
34	0.255	0.682	1.307	1.691	1.805	2.032	2.136	2.441	2.728	3.002	3.348	3.601
35	0.255	0.682	1.306	1.690	1.803	2.030	2.133	2.438	2.724	2.996	3.340	3.591
36	0.255	0.681	1.306	1.688	1.802	2.028	2.131	2.434	2.719	2.990	3.333	3.582
37	0.255	0.681	1.305	1.687	1.800	2.026	2.129	2.431	2.715	2.985	3.326	3.574
38	0.255	0.681	1.304	1.686	1.799	2.024	2.127	2.429	2.712	2.980	3.319	3.566
39	0.255	0.681	1.304	1.685	1.798	2.023	2.125	2.426	2.708	2.976	3.313	3.558
40	0.255	0.681	1.303	1.684	1.796	2.021	2.123	2.423	2.704	2.971	3.307	3.551
60	0.254	0.679	1.296	1.671	1.781	2.000	2.099	2.390	2.660	2.915	3.232	3.460
80	0.254	0.678	1.292	1.664	1.773	1.990	2.088	2.374	2.639	2.887	3.195	3.416
100	0.254	0.677	1.290	1.660	1.769	1.984	2.081	2.364	2.626	2.871	3.174	3.390
120	0.254	0.677	1.289	1.658	1.766	1.980	2.076	2.358	2.617	2.860	3.160	3.373
140	0.254	0.676	1.288	1.656	1.763	1.977	2.073	2.353	2.611	2.852	3.149	3.361
160	0.254	0.676	1.287	1.654	1.762	1.975	2.071	2.350	2.607	2.847	3.142	3.352
180	0.254	0.676	1.286	1.653	1.761	1.973	2.069	2.347	2.603	2.842	3.136	3.345
200	0.254	0.676	1.286	1.653	1.760	1.972	2.067	2.345	2.601	2.838	3.131	3.340
250	0.254	0.675	1.285	1.651	1.758	1.969	2.065	2.341	2.596	2.832	3.123	3.330
inf	0.253	0.674	1.282	1.645	1.751	1.960	2.054	2.326	2.576	2.807	3.090	3.290

Work Cited

1. *Game Length & Maximizing Time Value in game design: Games precipice*. BoardGameGeek. (n.d.). Retrieved April 24, 2022, from https://boardgamegeek.com/blogpost/30453/game-length-maximizing-time-value-game-design
2. Eng, D. (2020, June 29). *The player experience*. University XP. Retrieved April 24, 2022, from https://www.universityxp.com/blog/2019/9/10/the-player-experience
3. Asymptotic theory. (n.d.). Retrieved April 24, 2022, from https://www.statlect.com/asymptotic-theory/mean-square-convergence
4. Traylor, R. (n.d.). The Math Citadel. Retrieved April 24, 2022, from https://www.themathcitadel.com/a-generalized-geometric-distribution-from-vertically-dependent-bernoulli-random-variables/
5. *Measures of central tendency*. Mean, Mode and Median - Measures of Central Tendency - When to use with Different Types of Variable and Skewed Distributions | Laerd Statistics. (n.d.). Retrieved April 25, 2022, from https://statistics.laerd.com/statistical-guides/measures-central-tendency-mean-mode-median.php
6. Level 12.0: Game Balance: Game Design Concepts. (n.d.). Retrieved April 25, 2022, from https://learn.canvas.net/courses/3/pages/level-12-dot-0-game-balance