# ISYE 6644 Project: A Two Person Coin Toss Game

March 12, 2021
Kaitlyn Boyle

## Abstract:

In this project I created a simulation of two people playing a simple game – toss a die, and based on the outcome, either win or lose money. This game repeats until one player is unable to complete the transaction. The idea of simulating the game is to determine the expected value of the number of cycles the game is expected to last until a player goes out, and the distribution of the number of cycles over many simulations. I used Python to simulate the game and first step analysis of a Markov Chain to get the expected value. I found that from repeated trials, the expected number of cycles before a player lost the game was about 17.5 cycles, and the overall distribution of the number of cycles played per game appears to follow a geometric distribution. The first roll of each player is a weak predictor of how long the game will last.

## Background:

Rules: The rules of the game are simple: Player A and Player B each start with 4 coins, and the pot starts with 2 coins. Each player takes turns rolling a fair, 6-sided die. If the player rolls a 1, the player does nothing. If the player rolls a 2, the player takes all the coins in the pot. If the player rolls a 3, the player takes half of the coins in the pot, rounded down in the case of an odd number of coins in the pot. If the player rolls a 4, 5, or 6, the player must put one coin into the pot. This is repeated until one player can no longer perform a transaction, i.e., a player needs to put a coin in the pot, but the player does not have any coins left. This would then be the end of the game. A cycle is defined as Player A, then Player B rolling the die and completing their turn.

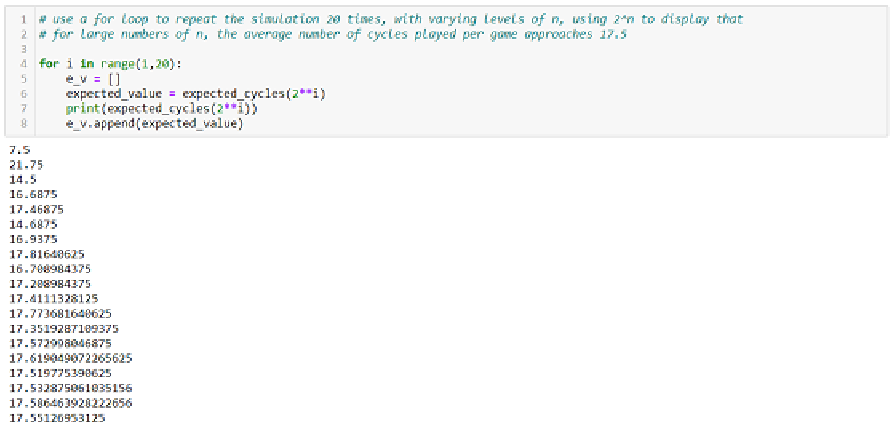
Assumptions: It is assumed that the die is fair, and that each roll of the die is independent and identically distributed. Each side of the die has a 1/6 chance of being rolled. Each player starts with the same number of coins at the beginning of each game, and each player has the same chance of winning or losing.

I followed first step analysis for a Markov Chain, which I learned about from a lecture from the University of Washington Stochastic Modeling course (Chen, 2018). This problem is similar to the Gambler's Ruin problem, which is a well-known concept that demonstrates that a Gambler will eventually lose all their money after repeated trials (Wikipedia, 2021). Reading about the Gambler's Ruin problem helped me understand the problem I needed to simulate, and a process to follow to find the expected value of the number of cycles until a player lost. The process is a Markov process in discrete time, and the expected value of the number of cycles necessary until one player loses can be determined by running the simulation thousands of times. The greater the number of simulations, the better the approximation of the expected value. The idea of the first step analysis approach to the Markov chain is that you can condition on the first step of the Markov chain to determine the number of cycles until the expected outcome happens. We are interested to see if the first roll of the die for each player can lead to a prediction of the number of cycles until one of the players goes out.

Simulation Setup: I used Python to create a function to simulate the game and do statistical analysis on the outcomes. Because the code is a bit long, I am omitting it here, but it can be found in the Appendix and the attached Jupyter Notebook. I used the pseudo-random number generator, random.randint() to simulate the die tosses. This returns a random integer in the specified range, in this case, an integer between 1 and 6. According to the Python random module documentation, the random.randint() function uses the Mersenne Twister algorithm to produce pseudo-random numbers (Python Docs, 2021). This is a very widely used and trusted pseudo-random number generator and passes rigorous statistical tests for randomness (Wikipedia, 2021). I defined a function, game(), which includes all the code necessary to produce the simulation. This function initializes the players and the pot, rolls the die, defines the rules of the game, and tracks the current balance of each player and the pot. The game() function is designed to track several areas while running, including: each player's current balance during the game, each player's final balance after the game is over, and the pot's current and ending balance; the winner and loser of each game; every roll of the die for each player; and the total number of cycles played until the game ended. I then used a for loop to run the game() function for 10,000 games, and recorded every game in a Pandas data frame, to use for statistical analysis after simulating thousands of games.
To get the expected value of the number of cycles required until a player lost the game, I created a second function called game_cycles(), which functions the same way as game() but only returns the number of cycles of each game. I did this so that I could run the simulation thousands or even millions of times without recording all the extra information, and just the expected number of cycles. I defined another function, expected_cycles(n), which runs the game_cycles() function n times, and records the average number of cycles completed for each run of the game_cycles() function. I then used a for loop to run expected_cycles(n) 20 times, to get different ranges of values for the expected number of cycles. We observed that for large numbers of n, the number of cycles approaches 17.5.

## Main Findings

I used the functions I created to simulate the game thousands of times. I first ran the simulation for 10,000 games and recorded each game in a Pandas data frame. I then repeated the simulation 20 times with a different number of iterations each time, ranging from 2 up to 1,048,576 times, by using a for loop to run the simulation $2^n$ times, where n iterates from 1 to 20. As n approaches 20, we see that the average number of cycles until a player loses the game approaches 17.5.



```
# use a for loop to repeat the simulation 20 times, with varying levels of n, using 2^n to display that
# for large numbers of n, the average number of cycles played per game approaches 17.5

for i in range(1,20):
    e_v = []
    expected_value = expected_cycles(2**i)
    print(expected_cycles(2**i))
    e_v.append(expected_value)
```

```
7.5
21.75
14.5
16.6875
17.46875
14.6875
16.9375
17.81640625
16.708984375
17.208984375
17.4111328125
17.773681640625
17.3519287109375
17.572998046875
17.619049072265625
17.519775390625
17.532875061035156
17.586463928222656
17.55126953125
```

*Figure 1: Average Number of Cycles Over Thousands of Repeated Trials*

The average number of cycles from 10,000 repeated games was 17.69, very close to our average output on a million trials. The minimum number of cycles was 5, and the maximum number of cycles was 118. Obviously, there were some outliers. I used a box plot to determine the range of values and where the outliers were.

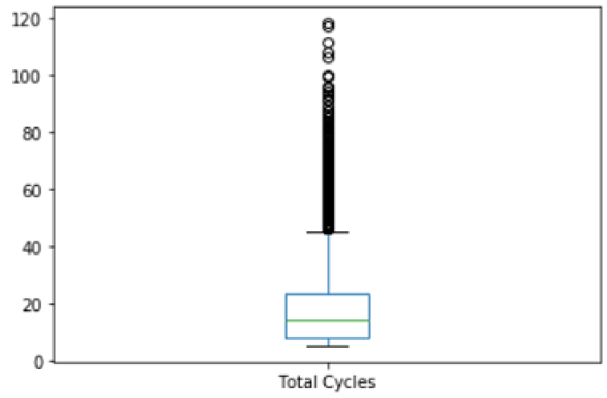


*Figure 2: Boxplot of Total Cycles*

I considered an outlier to be any number of cycles outside of the 99th percentile from the box plot. I took the top outliers corresponding to the games that were played for 90 trials or more and did some exploratory analysis. I found that for these outlier games, the distribution of die rolls was not uniform, as it appeared to be in the other games. In fact, in these games, Player A rolled a 2 (take all the coins from the pot) more than 20% of the time, a 3 (take half the coins from the pot) 17.1% of the time, and a 1 (do nothing) 17.2% of the time, while rolling a 4, 5, or 6 (put a coin in the pot) 15.5%, 16.5%, and 13.5% of the time, respectively. Player B rolled a 3 20.3% of the time, a 2 18.7% of the time, and a 1 15.5% of the time, and a 4, 5, or 6 less than 14.9%, 15%, and 15.6% of the time, respectively. Typically, a fair die toss has a uniform distribution, with each side of the die having a probability of 1/6, or 16.67%, of being rolled on each die toss., and the overall distribution of each player's die roll follows this uniform distribution.

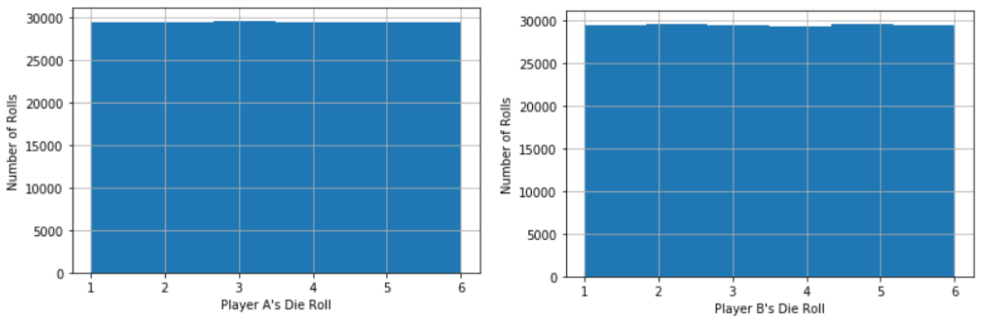


*Figure 3: Distribution of All Die Rolls for Players A and B*

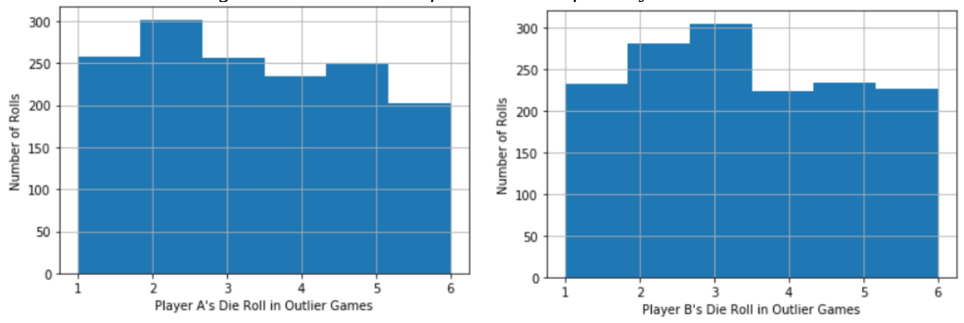


*Figure 4: Distribution of Player A and B's Die Rolls in Higher Outlier Games*

It appears that the distribution of the number of cycles follows a geometric distribution. A geometric distribution is discrete, but it could take any number of trials to see the expected result (Wikipedia, 2021). There are only two possible outcomes of each cycle, either the player stays in the game or loses the game. The geometric distribution is helpful in modeling a sequence of trials and the number of trials before seeing an expected outcome. Here, the geometric distribution is useful in modeling how many cycles in a game until a player loses. The assumptions of a geometric distribution are that it is modeling a sequence of independent trials, there are only two outcomes (a player goes out of the game or does not), and the probability of success for each trial is the same. The game here perhaps does not meet the last assumption, that the probability of success for each trial is the same, because if a player has a lot of coins already and they roll a 4, 5, or 6, they are not likely to go out of the game during that cycle.

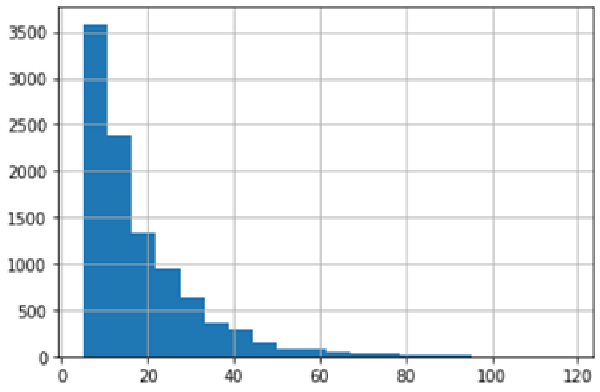


*Figure 5: Distribution of the number of cycles per game for 10,000 games*

Since we are interested in the first step analysis of the game, we can look at the first roll of each game and see how many cycles it then took for the game to end. To determine if the number of cycles can be determined by the first roll of each player, I looked at the overall distribution of first rolls, and compared this to the above average number of cycles and the below average number of cycles distributions. The graphs below show the overall distribution of each player's die rolls, and the distribution of die rolls of the outlier games.

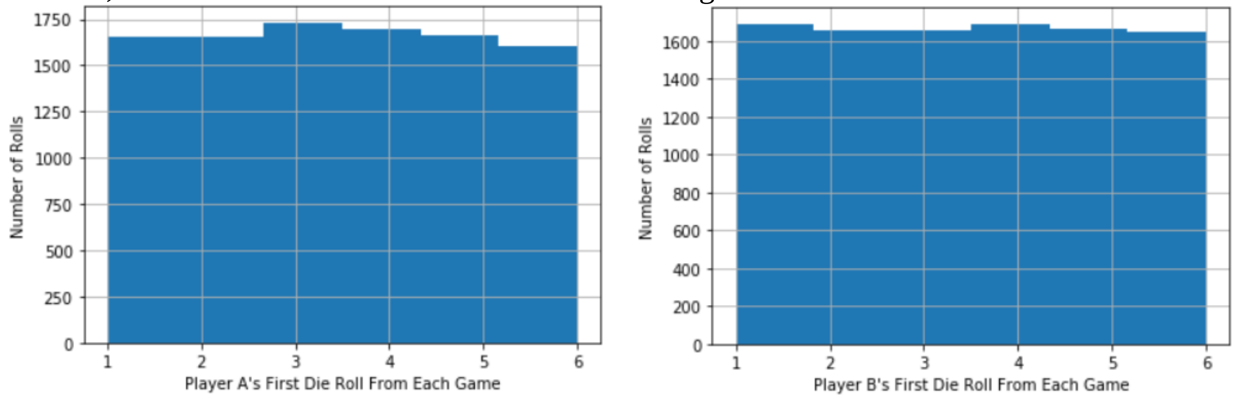


*Figure 6: Overall Distribution of Each Player's First Roll of the Die*

For games with between 15 and 19 cycles, the games started with a 1, 2, or 3 more often than a 4, 5, or 6. For game with 20 or more cycles, it was even more likely to start with a 1, 2, or 3. For games with less than 15 cycles per game, more than half of the games started with a 4, 5, or 6. This makes sense since after the first step, the players are playing with less money than they started with, and are therefore more likely to go out sooner. It looks like if a game starts with a 4, 5, or 6, we can say fairly confidently that the game will end before the average number of cycles, but if the game starts with a 1, 2, or 3, it is equally as likely that the game will last the average number of cycles as it is to last an above average number of cycles.

## Conclusions

It can be shown that running the simulation 10,000 times leads to finding the distribution and expected value of the number of cycles per game until the game ends. The expected value can be found to be approximately 17.5, and this is further proven by running the simulation 20 different times, with varying number of games. As the number of games becomes increasingly large, the average number of cycles played per game approaches the expected value of 17.5 The distribution of the number of cycles per game appears to follow a discrete Geometric distribution. It can be shown that the first roll of each game can be used as a weak predictor of the number of cycles per game. In the future, I would revisit the first roll of each game, and perhaps change the setup of the game so that the first roll is fixed, and see if we get a different expected value of number of cycles depending on the first roll.

## Appendix:

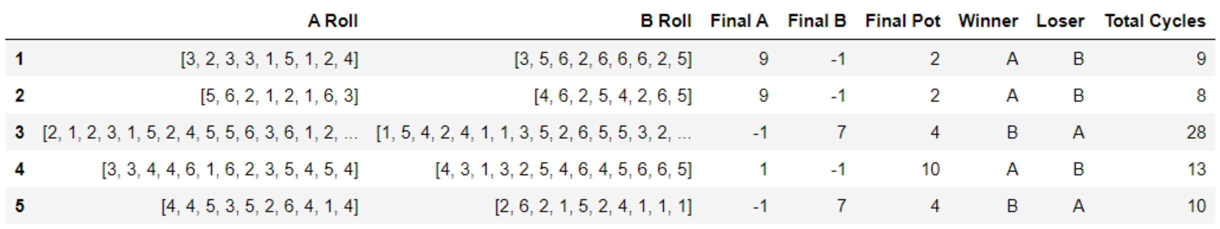
All code and statistical analysis can be found in the Jupyter Notebook: Simulation_mini_project.ipynb. Each cell block is commented out and explained with text so that you know what every cell block does.

### How to Run Code:

1. Download the zip folder titled "Coin_Game_Simulation.zip" and unzip the file in your local folder.

2. Open the Simulation_mini_project.ipynb file.
3. Make sure you have the required libraries installed.
4. Follow the Jupyter notebook code cells sequentially. Each code cell is commented out and has a text cell to describe what each cell is doing.

## Figures:



| | A Roll | B Roll | Final A | Final B | Final Pot | Winner | Loser | Total Cycles |
|---|---|---|---|---|---|---|---|---|
| **1** | [3, 2, 3, 3, 1, 5, 1, 2, 4] | [3, 5, 6, 2, 6, 6, 6, 2, 5] | 9 | -1 | 2 | A | B | 9 |
| **2** | [5, 6, 2, 1, 2, 1, 6, 3] | [4, 6, 2, 5, 4, 2, 6, 5] | 9 | -1 | 2 | A | B | 8 |
| **3** | [2, 1, 2, 3, 1, 5, 2, 4, 5, 5, 6, 3, 6, 1, 2, ... | [1, 5, 4, 2, 4, 1, 1, 3, 5, 2, 6, 5, 5, 3, 2, ... | -1 | 7 | 4 | B | A | 28 |
| **4** | [3, 3, 4, 4, 6, 1, 6, 2, 3, 5, 4, 5, 4] | [4, 3, 1, 3, 2, 5, 4, 6, 4, 5, 6, 6, 5] | 1 | -1 | 10 | A | B | 13 |
| **5** | [4, 4, 5, 3, 5, 2, 6, 4, 1, 4] | [2, 6, 2, 1, 5, 2, 4, 1, 1, 1] | -1 | 7 | 4 | B | A | 10 |

*Figure 7: A Few Rows of the Data Frame of Game Output*

## References

Chen, Y.-C. (2018, Autumn). *Lecture 3: Discrete-Time Markov Chain – Part I.* Retrieved from University of Washington: http://faculty.washington.edu/yenchic/18A_stat516/Lec3_DTMC_p1.pdf

Python Docs. (2021, March 12). *random — Generate pseudo-random numbers*. Retrieved from Python Documentation: https://docs.python.org/3/library/random.html#random.randint

Wikipedia. (2021, March 21). *Gambler's Ruin*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Gambler%27s_ruin

Wikipedia. (2021, March 12). *Geometric Distribution*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Geometric_distribution

Wikipedia. (2021, February 17). *Marsenne Twister*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Mersenne_Twister