# **Objectives**

The objective of this assignment was to programmatically determine the correlation coefficient of shuffling cards in four different ways on both 52 and 104 card decks. Given a deck of 52 cards in the first run, the cards must be shuffled in a way such that the first card is always 1 and the last card is always 52. The same applies to the deck of 104 cards in the first run, only the last card is always 104 instead of 52. With 52 cards the second run, the shuffling is changed so that the first card is 27, the second is 1, and the third is 26, and so on with the same applying to the 104-card deck only the cards being 53, 1, 54, and so on. Each shuffle must be repeated 15 time on each deck, and the correlation coefficient calculated for each. Finally, the correlation coefficient of each shuffle must be plotted against the number of shuffles.

Furthermore, for each run, two additional shuffling methods are employed. In the first alternative shuffling method, cards are shuffled by randomly selecting the next card from either the top half (cards 1-26) or the bottom half (cards 27-52) of the deck, with equal probability. When one half of the deck is depleted, cards are drawn solely from the remaining half. In the second alternative shuffling method, cards are shuffled by selecting the next card at random from the first half of the deck based on the probability calculated as the number of cards in the first half deck divided by the total number of cards left. Similarly, the next card is chosen randomly from the second half of the deck based on the probability calculated as the number of cards in the second half deck divided by the total number of cards left.

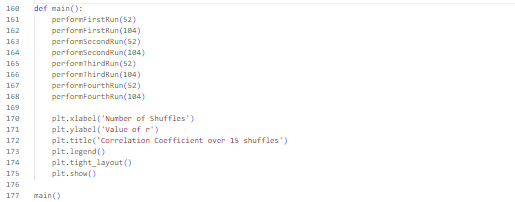
# **Program Design**

To accomplish the required functionality, six functions were implemented:

1. **shuffle1() –** This function is responsible for shuffling the decks of cards in the first run. It first calculates the half-length of the deck. It then iterates over a new empty array to length n, adding a card from the first half of the deck, then a card from the second until all the cards have been added.
2. **shuffle2() –** This function shuffles the decks of cards in the second run. It functions similarly to shuffle1(), except instead of taking the cards from the first half of the deck it takes the cards from the second half of the deck first.
3. **shuffle3()** - This function simulates shuffling the deck in the third run. Initially, it determines the length of the deck. Then, it initializes an empty list for the shuffled deck. Next, it divides the deck into two halves. The function iterates as long as both halves are not empty. During each iteration, it randomly selects a half to draw a card from, giving equal probability to both halves. After each draw, it appends the drawn card to the shuffled deck. Once one of the halves becomes empty, it extends the shuffled deck with the remaining cards from the non-empty half.
4. **shuffle4()** - This function represents the shuffling process in the fourth run. It starts by computing the length of the deck. Then, it initializes an empty list to hold the shuffled deck. After splitting the deck into two halves, the function enters a loop where it continues until both halves are empty. During each iteration, it calculates the probability of selecting a card from the first half based on the number of remaining cards in each half. It randomly decides whether to draw a card from the first half according to this probability. After each draw, it appends the selected card to the shuffled deck. Once one of the halves becomes empty, it extends the shuffled deck with the remaining cards from the non-empty half.
5. **calculateR() –** This function is a Python implementation of the Pearson’s correlation coefficient.The function first calculates *sumi*, or the sum of the numbers one to *n*. Then it calculates *sumsq*, or the sum of the squares of the numbers one to *n*. The *sqsum* calculated in the next line is a square of *sumi.* The next line calculates *sum\_product,* or the sum of the products of the positions of each card in the shuffled deck and its value. It iterates over each card using *enumerate(deck)* to get both the index and the value of the card, then multiplies the index by the value to and sums each of these. Lastlty, *r* or the correlation coefficient is calculated by .
6. **performFirstRun() –** This function represents the first run, and is responsible for iteratively calling the shuffle1() and calculateR() functions 15 times and adding the shuffled deck with its r value to a list and shuffle number. In addition to this, it also plots the r values vs. shuffle number with some formatting to make the graph easy to interpret.
7. **performSecondRun() –** Functionally similar to performFirstRun(), except it calls shuffle2() instead of shuffle one and contains some different formatting for the plot.
8. **performThirdRun() –** Functionally similar to performFirstRun(), except it calls shuffle3() instead of shuffle one and contains some different formatting for the plot.
9. **performFourthRun() –** Functionally similar to performFirstRun(), except it calls shuffle4() instead of shuffle one and contains some different formatting for the plot.
10. **main() –** This is the driver code that calls performFirstRun(), performSecondRun(), performThirdRun(), and performFourthRun() on both 52 and 104 cards, as well as some additional plot formatting and the final displaying for the plot itself.

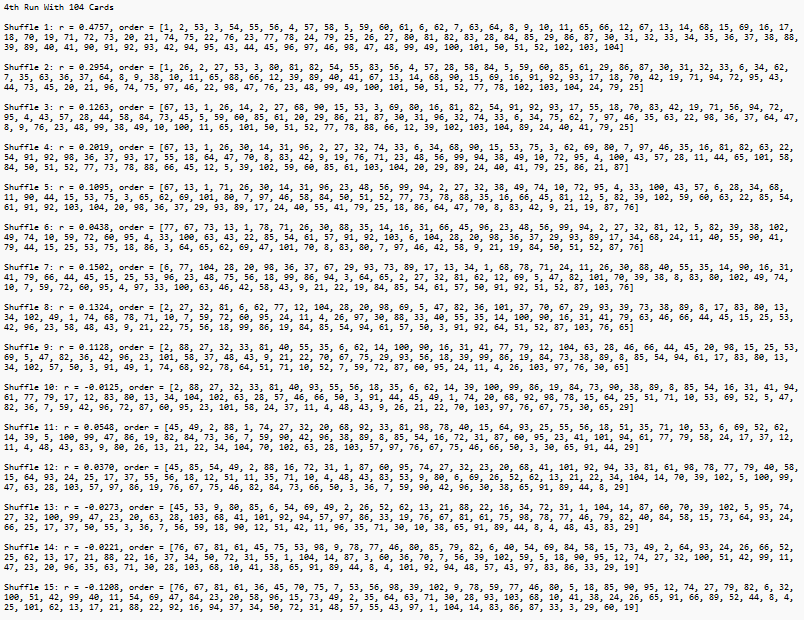
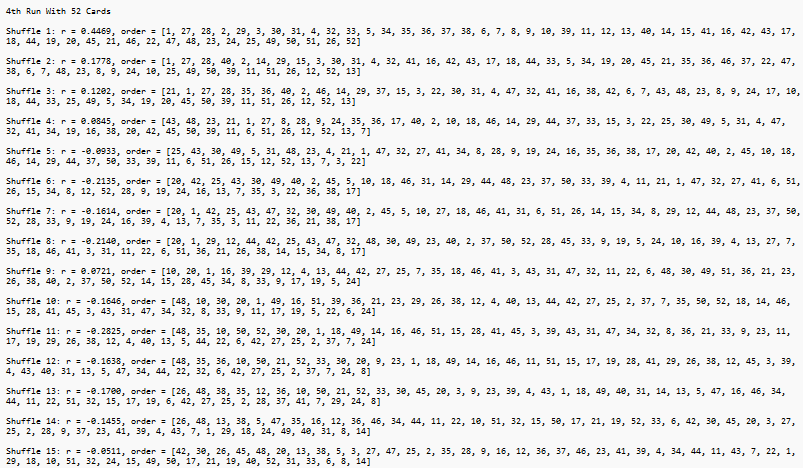
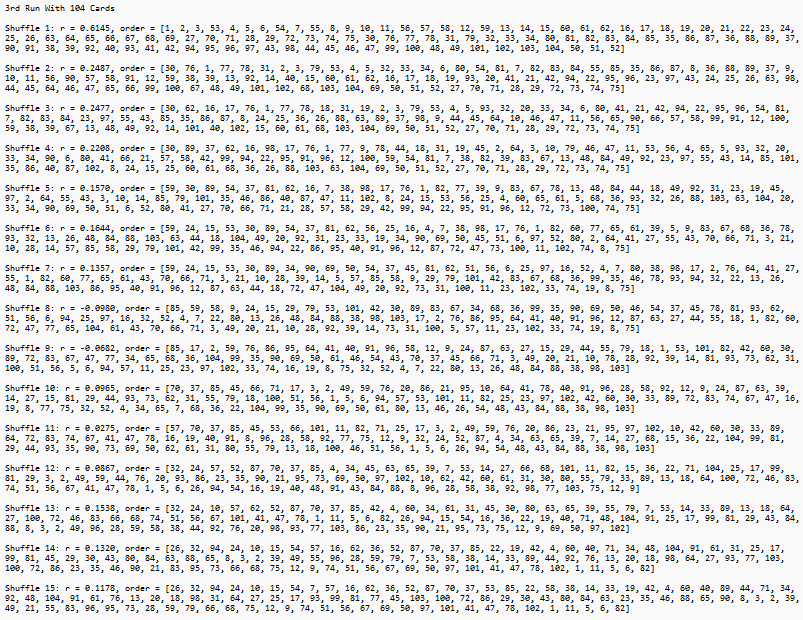
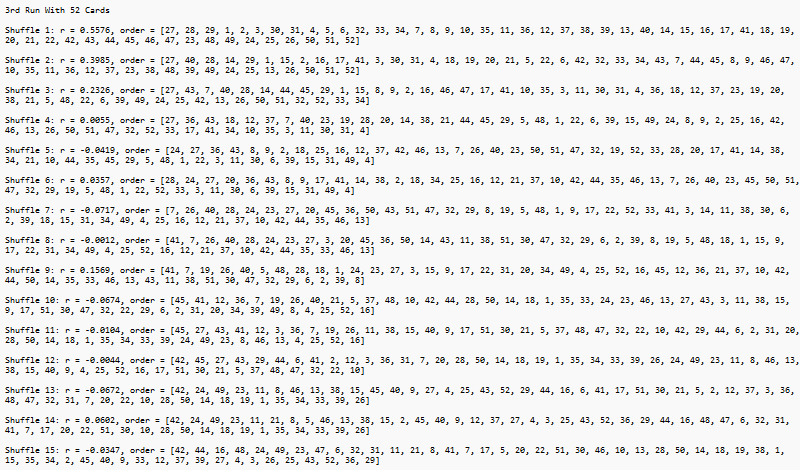
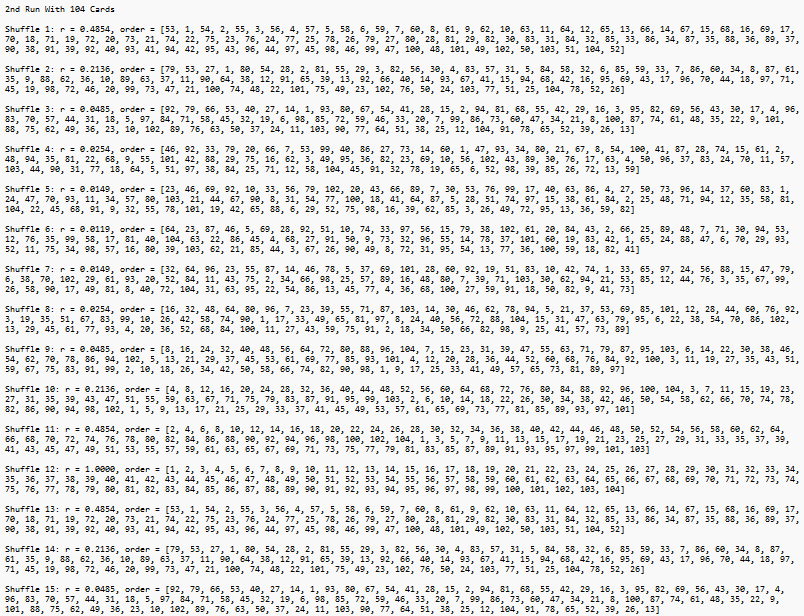
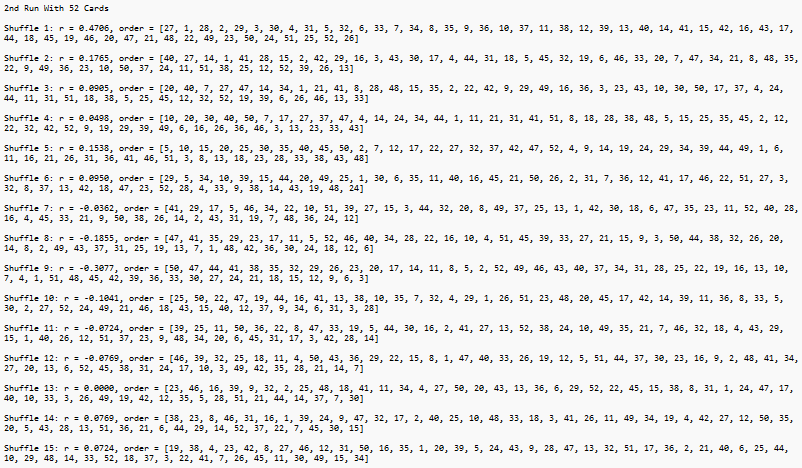
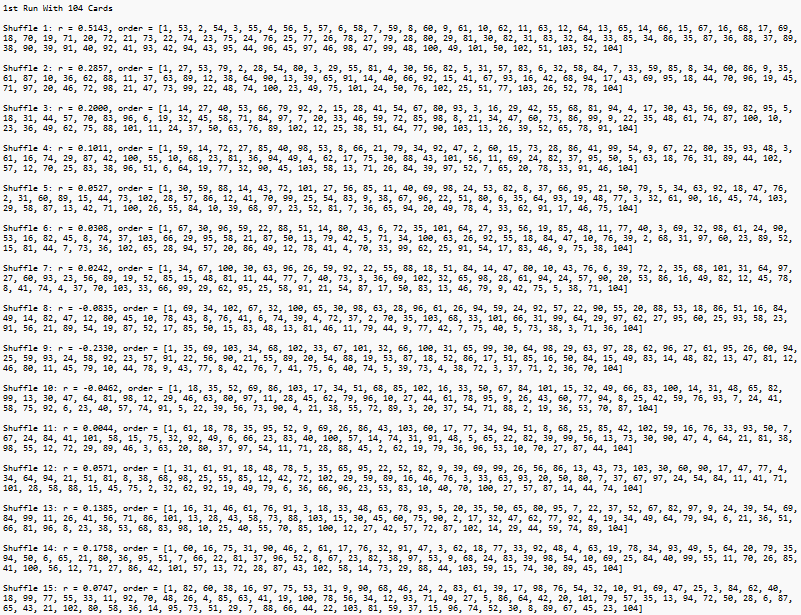
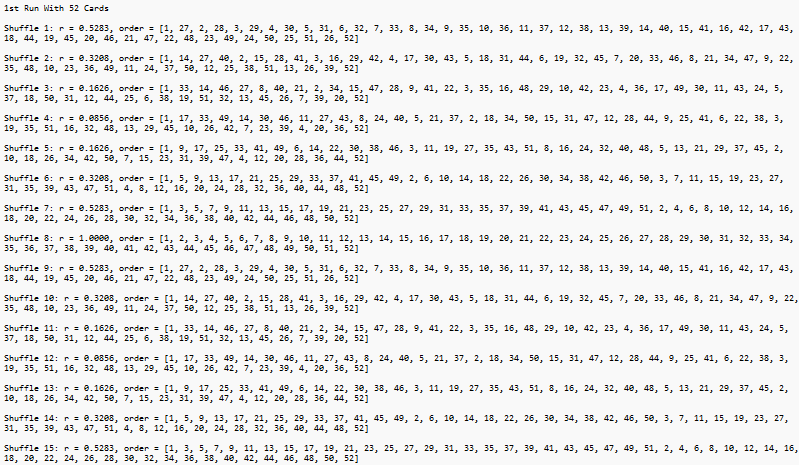
Code screenshots: **A screenshot of a computer program

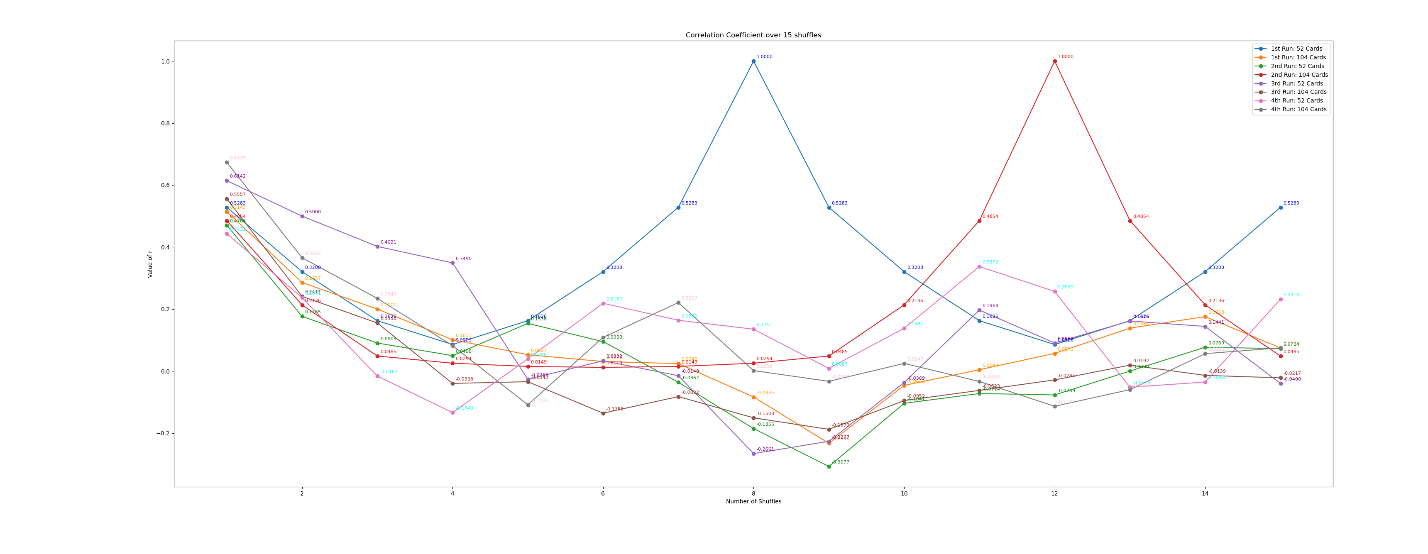
Description automatically generated**

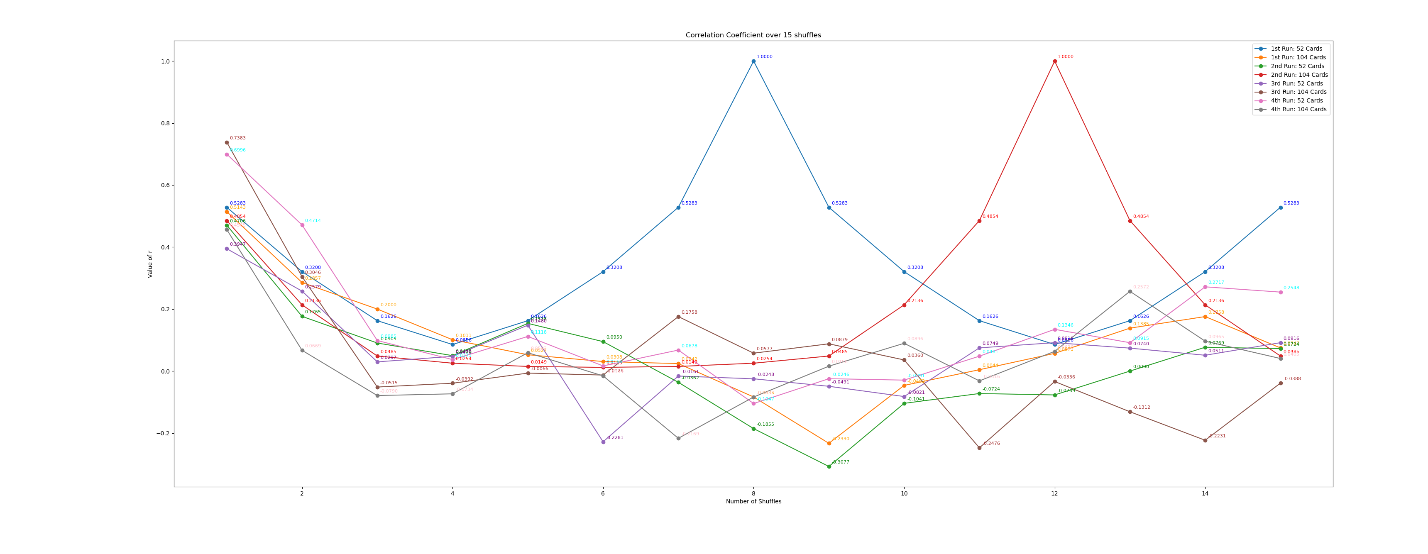
****

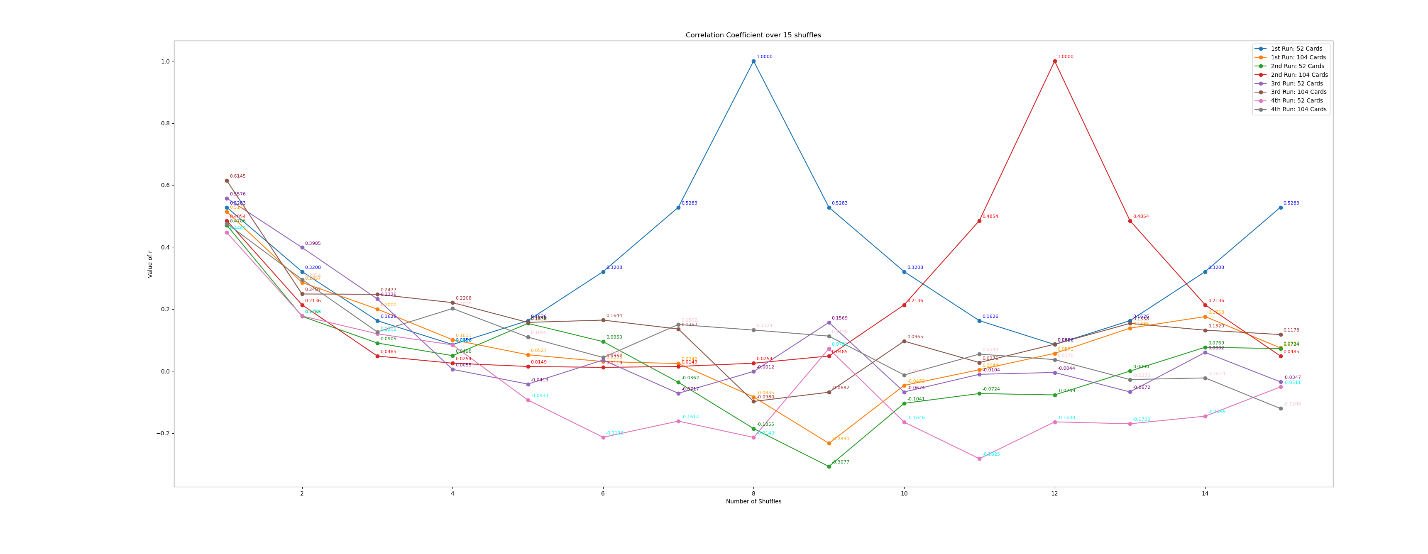
# **Results**

In this version of the code the output is redirected to a text file to record each deck and its corresponding R value. You can find the screenshots of the text file for the first execution shown below, as well as its corresponding in addition to two other graphs:









# **Questions / Analysis**

1. In contrast to the original 2 shuffling methods, shuffle3() and shuffle4() seem to order the cards in a much more random fashion. The R value for the decks of cards seems to vary between executions of the program, meaning that the order of the cards varies unlike shuffle1() and shuffle2(), which hold the same R values for a certain shuffle. This is likely because shuffle3() and shuffle4() rely on probability when picking between the first or second half of the deck, while the first two shuffle methods choose the first half, then the second half repeatedly. For example, in one execution of the program for shuffle 8 the R value of the third shuffle method with 52 cards is equal to -0.2661, but in another that same R value is equal to -0.0248. This means that there is no determinable shuffle for all executions of the program at which R is at a minimum, instead the randomness of the order of the cards is also random.
2. Due to the nature of the shuffling in shuffle3() and shuffle4(), the cards do not deterministically return to their original order after a set number of shuffles. Instead, the cards returning to their original order falls completely on probability. For the third shuffle method, the probability of returning to the original order in one shuffle is not straightforward to calculate without making some assumptions about the randomness of the shuffling process. However, with each shuffle, the deck's order becomes increasingly random, making it less likely to return to the original order with subsequent shuffles. Like the third method, the fourth shuffling method also randomly selects cards from either half of the deck, but the probability of selecting from each half is proportional to the size of the half. Again, without specific assumptions about the randomness of the process, calculating the exact probability of returning to the original order is complex. However, like the third method, the likelihood of returning to the original order decreases with each shuffle.
3. For a deck of 52 cards, the number of possible permutations of the deck is 52!, or 8.0658 x 1067. For a deck of 104 cards, the number of possible permutations of the deck are 104!, or 1.0299028 x 10166. This makes it highly unlikely that the deck would return to its original order after just 15 shuffles using the third or fourth shuffling method.
4. It would likely take an astronomical number of shuffles for the cards to reach their original order.

# **Sources**

<https://www.w3schools.com/python/matplotlib_plotting.asp>

<https://rosettacode.org/wiki/Perfect_shuffle>

<https://stackoverflow.com/questions/43146319/annotate-a-plot-using-matplotlib-showing-values-in-the-plot>

<https://www.geeksforgeeks.org/matplotlib-pyplot-annotate-in-python/>

<https://www.socscistatistics.com/tests/pearson/default2.aspx>