Math 381 **HW 5: Markov chain** Xinming Zhang

This document is about applying the concept of Markov chain in our game in real life.

Considering the game like following:
We set an integer M as our goal score, and we start with a score of zero.
We will roll a six-sided fair die every turn.

If our score is greater than zero and our roll can decide our current score, then we subtract the roll from our current score. Otherwise, we add the roll to our score.
Note that our score will always be a nonnegative number, if the score becomes negative in the process, we will replace the score with zero.

If the score is equal than or greater than the goal score M, we win the game. Otherwise, if the score is a perfect square greater than 4 (9, 16, 25, etc), we lose.

We will roll the die every turn to change our score until we either win or lose.

One example of the game provided form the assignment is following:
The goal score M is 10, and we roll the die to change the score as the table below,

roll	score
2	2
4	6
2	4
5	9

From the table, the game ended and we lost after four rolls.
The score either decreased or increased, and we got a 9 which is a perfect square greater than 4, so we lost.

I will investigate the game using Markov Chains.
Before we create the matrix, we have to know the states we are going to define.
First, the score states are the first ones that come to me. There will be M scores we can get to before the game ends (0 to M-1). Then there will be M score states (normal states).
Beside the score, I also will set two absorbing states as winning states (win the game) and losing states (lose the game).

Then we can create our transition matrix.

I will create a M+2 by M+2 transition matrix, because there are M+2 states. First M states are the score states, then M+1 state will be the winning state, so M+2 state will be losing states.

Above all, the ijth entry (ith row and jth column) of the transition matrix will represent the probability of moving from ith state to jth state in each step for all 1≤ i ≤ M+2 and 1 ≤ j ≤ M+2. After all those, I can generate the transition matrix by sage coding as following:

```
import random
# set M as the goal score, and M is 10 in this case
M = 10;
# set temp as our current score
temp = 0;

# A function determine whether we lose the game getting a perfect square greater than 4
def lose(x):
    root = int(math.sqrt(x))
    return (root * root) == x and x >4

# initialize the transition matrix A a as M+2 by M+2 matrix with all zeros
A=zero_matrix(QQ,M+2);

# Go through every score we can get
for i in range(M):
    # every roll we can get
    for j in range(1,7):
        # Subtract the roll from the score when the score is greater than zero, and the roll
divides it
        if i>0 and i%j == 0:
            temp = i-j
        # Otherwise, add them together
        else:
            temp = i+j
        # if we win the game by getting a score greater than or equals to M, we win
        if temp >= M:
            # adding 1/6 to the winning state
            A[i,M] += 1/6
        # if we lose the game by getting a perfect square greater than 4
        elif lose(temp):
            # adding 1/6 to the losing state
            A[i,M+1] += 1/6
        # Otherwise, add 1/6 to the current score state
        else:
            A[i,temp] += 1/6
# generate the winning and losing state
A[M,M] = 1
A[M+1,M+1] = 1
# output the transition matrix
print(A)
```

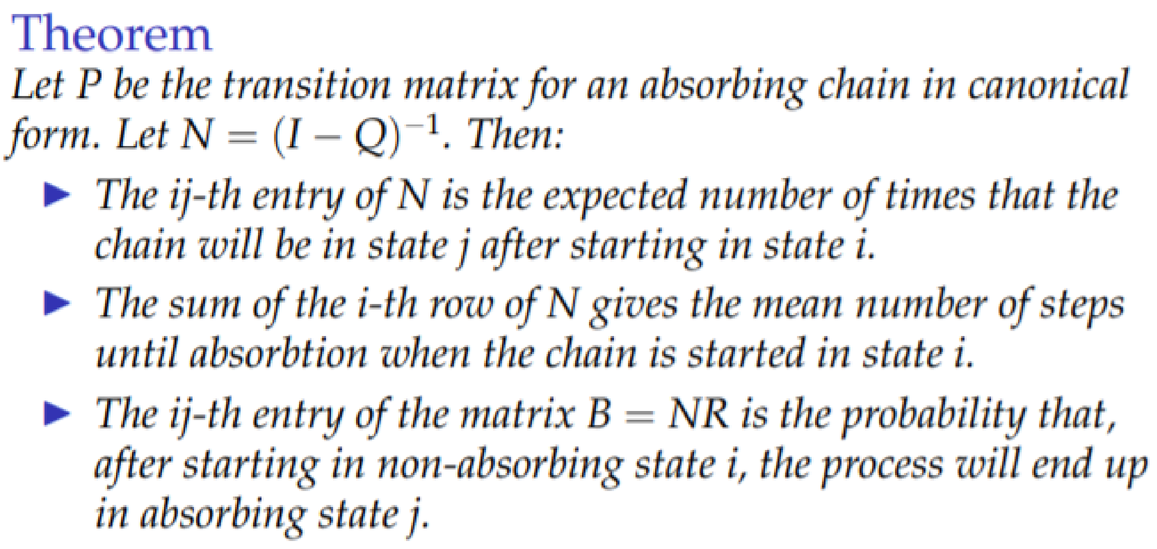
By running the code above on sage to get the transition matrix A as following:

```
[  0 1/6 1/6 1/6 1/6 1/6 1/6   0   0   0   0   0]
[1/6   0   0 1/6 1/6 1/6 1/6 1/6   0   0   0   0]
[1/6 1/6   0   0   0 1/6 1/6 1/6 1/6   0   0   0]
[1/6   0 1/6   0   0 1/6   0 1/6 1/6   0   0 1/6]
[1/6   0 1/6 1/6   0   0   0 1/6   0   0 1/6 1/6]
[1/6   0   0   0 1/6   0   0 1/6 1/6   0 1/6 1/6]
[1/6   0   0 1/6 1/6 1/6   0   0   0   0 1/3   0]
[  0   0   0   0   0   0 1/6   0   0   0 2/3 1/6]
[  0   0   0   0 1/6   0 1/6 1/6   0   0 1/2   0]
[  0   0   0   0   0   0 1/6   0 1/6   0 2/3   0]
[  0   0   0   0   0   0   0   0   0   0   1   0]
[  0   0   0   0   0   0   0   0   0   0   0   1]
```

This is the transition matrix for M = 10, it is an absorbing Markov chain, and it is the canonical form. The last two columns are the winning and losing state, and they are absorbing states. Hence, the first M by M matrix is the Q matrix, and the M by 2 matrix in the last two columns is matrix R. The picture explanation from class is shown below.

$$P = \left(\begin{array}{c|c} Q & R \\ \hline O & J \end{array}\right)$$

Canonical form



Theorem

Let P be the transition matrix for an absorbing chain in canonical form. Let $N = (I - Q)^{-1}$. Then:

- *The ij-th entry of N is the expected number of times that the chain will be in state j after starting in state i.*
- *The sum of the i-th row of N gives the mean number of steps until absorbtion when the chain is started in state i.*
- *The ij-th entry of the matrix $B = NR$ is the probability that, after starting in non-absorbing state i, the process will end up in absorbing state j.*

Theorem from class

Then, we use the Theorem from the class to get matrix N, and use the information below to get the expected value of the number of rolls made before the game ends and the winning probability (M=10). Put that in sage code will be as following:

```
# Matrix Q
Q = A[:M,:M]
#Matrix N
N = (matrix.identity(M)-Q).inverse()
# Calculating the expected number of rolls to end the game by summing the first row of N
matrix
sum(N[0][i] for i in range(10))*1.0
# Matrix R
R = A[:M,M:]
# Matrix B(winning state and losing probability)
B = N*R
# Winning probability of starting from a score of zero
B[0,0]*1.0
print(" ")
```

Output (Note that everything is under M = 10 case):

Matrix Q:

```
[   0 1/6 1/6 1/6 1/6 1/6 1/6   0   0   0]
[1/6   0   0 1/6 1/6 1/6 1/6 1/6   0   0]
[1/6 1/6   0   0   0 1/6 1/6 1/6 1/6   0]
[1/6   0 1/6   0   0 1/6   0 1/6 1/6   0]
[1/6   0 1/6 1/6   0   0   0 1/6   0   0]
[1/6   0   0   0 1/6   0   0 1/6 1/6   0]
[1/6   0   0 1/6 1/6 1/6   0   0   0   0]
[   0   0   0   0   0   0 1/6   0   0   0]
[   0   0   0   0 1/6   0 1/6 1/6   0   0]
[   0   0   0   0   0   0 1/6   0 1/6   0]
```

Matrix N (All in exact rational form):

```
[ 1157637/807644  492165/1615288   159429/403822  722763/1615288  788259/1615288
818133/1615288    92610/201911  637023/1615288   181551/807644               0]
[ 833985/1615288 3632841/3230576   186405/807644 1352823/3230576 1452927/3230576
1460409/3230576    85200/201911 1539627/3230576  296571/1615288               0]
[ 793665/1615288  900009/3230576   953181/807644  837687/3230576 1067871/3230576
1434777/3230576    91932/201911 1511211/3230576  507099/1615288               0]
[ 670575/1615288  388407/3230576   247323/807644 3793641/3230576  800961/3230576
1226727/3230576    52992/201911 1366773/3230576  500805/1615288               0]
[  324657/807644  194073/1615288   128781/403822  522135/1615288 1919295/1615288
375129/1615288    46266/201911  626859/1615288   117699/807644               0]
[ 554157/1615288  259797/3230576   112617/807644  512139/3230576 1082355/3230576
3722757/3230576    38898/201911 1134735/3230576  390447/1615288               0]
[  174555/403822    84195/807644    39015/201911   283149/807644   304101/807644
305667/807644   240372/201911   209553/807644    62073/403822               0]
[   58185/807644   28065/1615288    13005/403822   94383/1615288  101367/1615288
101889/1615288    40062/201911 1685139/1615288    20691/807644               0]
[   30498/201911     8136/201911    18318/201911    24642/201911    54768/201911
22674/201911    54450/201911    56898/201911   212850/201911               0]
[   78517/807644   38913/1615288    19111/403822  127239/1615288  174391/1615288
132121/1615288    49137/201911  145715/1615288   162591/807644               1]
```

Matrix R:
```
[   0    0]
[   0    0]
[   0    0]
[   0 1/6]
[1/6 1/6]
[1/6 1/6]
[1/3    0]
[2/3 1/6]
[1/2    0]
[2/3    0]
```

Matrix B (N*R):
```
[1120925/1615288  494363/1615288]
[2262945/3230576  967631/3230576]
[2421985/3230576  808591/3230576]
[2032559/3230576 1198017/3230576]
[1041385/1615288  573903/1615288]
[2155245/3230576 1075331/3230576]
[  623899/807644   183745/807644]
[1284825/1615288  330463/1615288]
[  175414/201911    26497/201911]
[4556131/4845864  289733/4845864]
```

The expected value of the number of rolls made before the game ends (M = 10):
```
4.65261612789793 (Decimal form)
7515315/1615288 (Rational form)
```

The probability of the winning the game (M = 10):
```
0.693947457047907 (Decimal form)
1120925/1615288 (Rational form)
```

From the results shown above, we know when we set the goal score to 10, the expected value of the number of rolls made before the game ends is about 4.65 (exact one shown in the output), but that is the ideal result we got from the transition matrix. We can try to prove that it should be right by doing simulations. We can simulate the game and average the number of rolls we took to the end of the game. Then compare those two numbers to check our answer.

Generate the code by simulating 10000 games 10 times, and averaging number of roll to get a list of 10 “real” results as following (**Simulation**):
```
# initialize the simulation result
result = []
# 10 times of 10000 games
for i in range(10):
    for j in range(10000):
        # initialize the score and times of rolls and the current score
        score = 0
        count = 0
        # while the game is not ended
        while(score < M and not lose(score)):
            # roll the die and record the time
            count += 1
            roll = random.randint(1,6)
            # the operations of the game for corresponding score
            if(score > 0 and score % roll == 0):
                score -= roll
            else:
                score += roll
        result.append(count)
    #output the average rolls needed to end the game for those 10 times 10000 games
    op = sum(result)/len(result) * 1.0;
    print(op)
```

After running the code, we can get the output as (Decimal form):

```
4.68330000000000
4.67175000000000
4.66833333333333
4.66775000000000
4.65806000000000
4.66446666666667
4.66271428571429
4.65836250000000
4.65688888888889
4.65071000000000
```

The range of those 10 results is from 4.65071 to 4.6833. Back to the expected value we got which is 4.65261612789793, the expected value is in the range of simulation results. From that, we know we can trust the expected value we got. We are confident to use that for the expected value of the number of rolls made before the game ends. (At least for M=10 case)

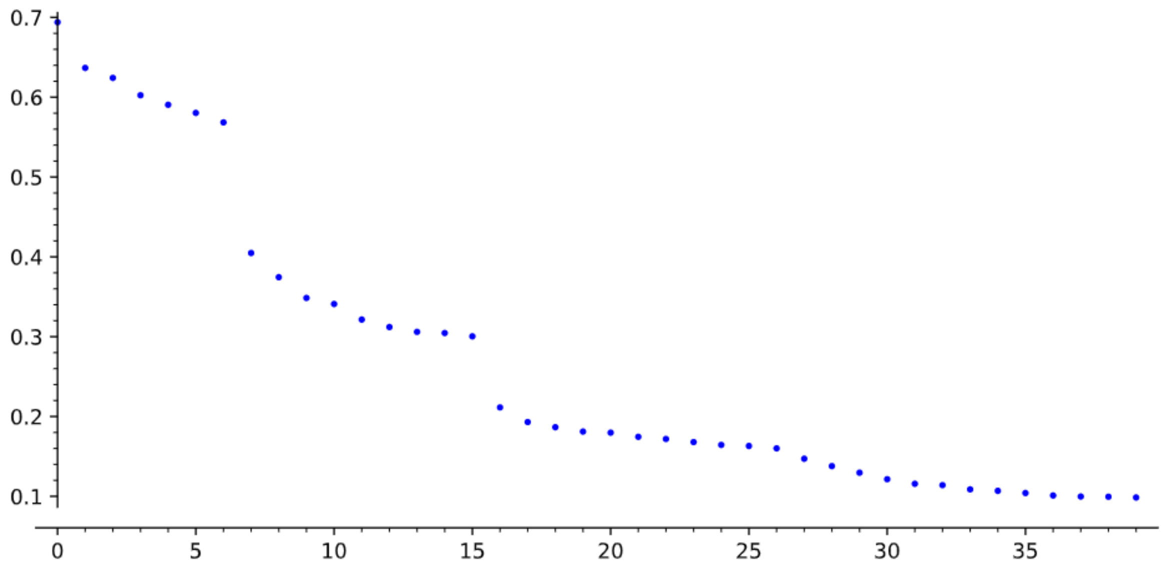
After we did all the analysis on the M = 10 case, we can move on to a more general problem. We can try to look at the probability of winning for various M.
We already know how to get the winning probability from using the transition matrix.
Let's say the winning probability is P, so we can calculate P for every M using the same method. The only difference will be the value of M.

When I was choosing the range of M I would like to run, I found out that when M is less than 10 the winning probability is always 1. That makes sense, when I look back to the rules of the game. There is only one perfect square less than , but according to the game, 4 can not be a losing state, so I chose the range starting from 10. Note that when M=9, when we get a score of 9, we win instead of losing.

Then, we can generate the code as following:

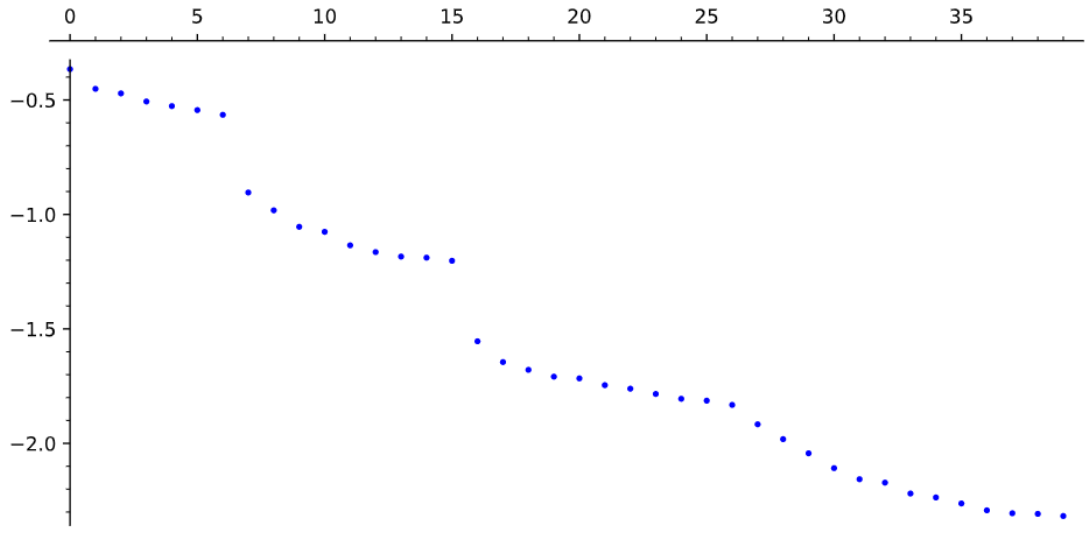
```
# Initialize the total winning probability as a list
total_win = []
log_win = []
# Running M a from 10 to 50
for M in range(10,50):
    # same process as the original transition matrix, by just changing the M
    A=zero_matrix(QQ,M+2);
    temp = 0
    for i in range(M):
        for j in range(1,7):
            if i>0 and i%j == 0:
                temp = i-j
            else:
                temp = i+j
            if temp >= M:
                A[i,M] += 1/6
            elif lose(temp):
                A[i,M+1] += 1/6
            else:
                A[i,temp] += 1/6
    A[M,M] = 1
    A[M+1,M+1] = 1
    Q = A[:M,:M]
    N = (matrix.identity(M)-Q).inverse()
    R = A[:M,M:]
    B = N*R
    win_r = B[0,0]*1.0
    total_win.append(win_r)
    log_win.append(log(win_r))
# Plot the winning probability
list_plot(total_win)
# list_plot(log_win)
```

Output:



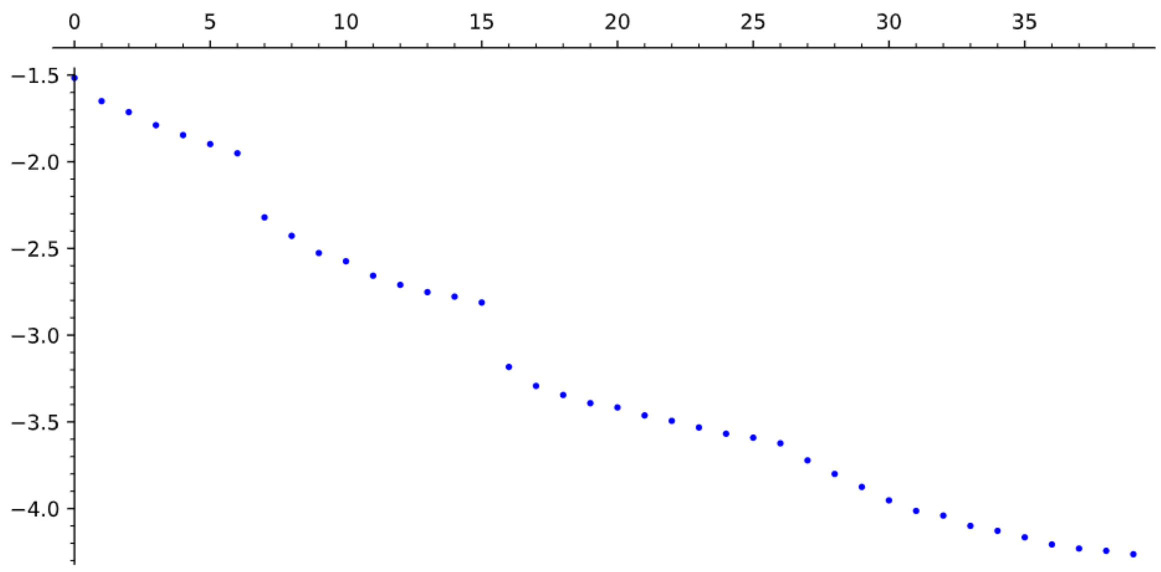
Plot of normal probability of winning for 10 ≤ M ≤ 50

The first thing that came up to me was how the gap formed in the graph. Then I found out that every gap is when M is equal to a perfect square plus 1. That makes sense, because the probability of losing increases instantly when we are adding one perfect square (losing state) to the possible numbers.
We can also see the graph is decreasing, but we don't really know what the function is for the graph. Then, I plotted the log of those probabilities to see the relationship.

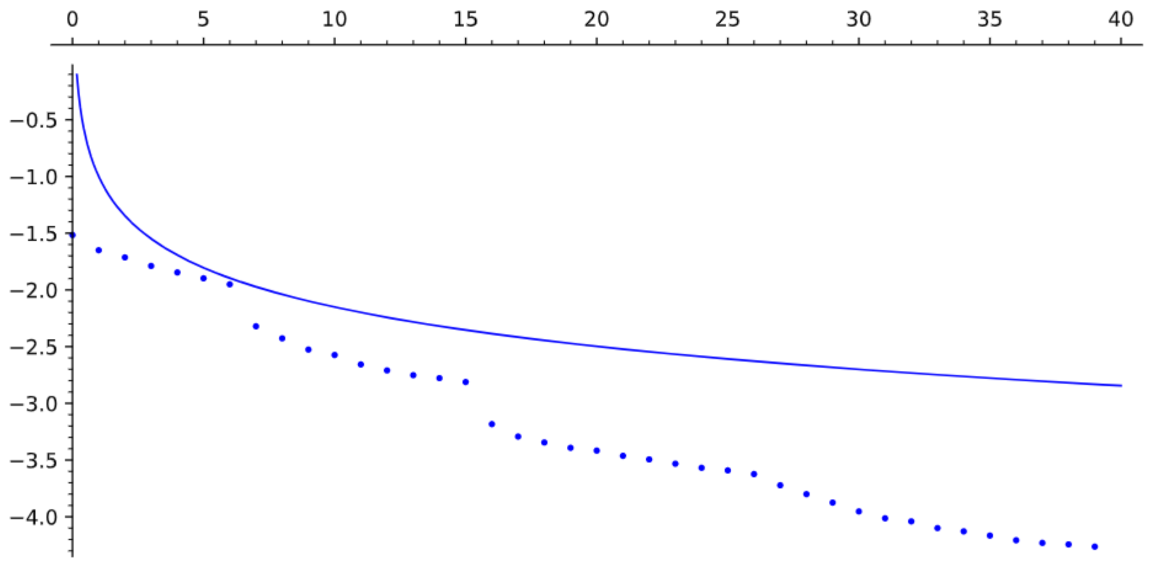


The graph has a more linear look, but the points are a bit "curved". That is why we cannot use a single function to express the graph. What we can do is modify it to make it look more linear.

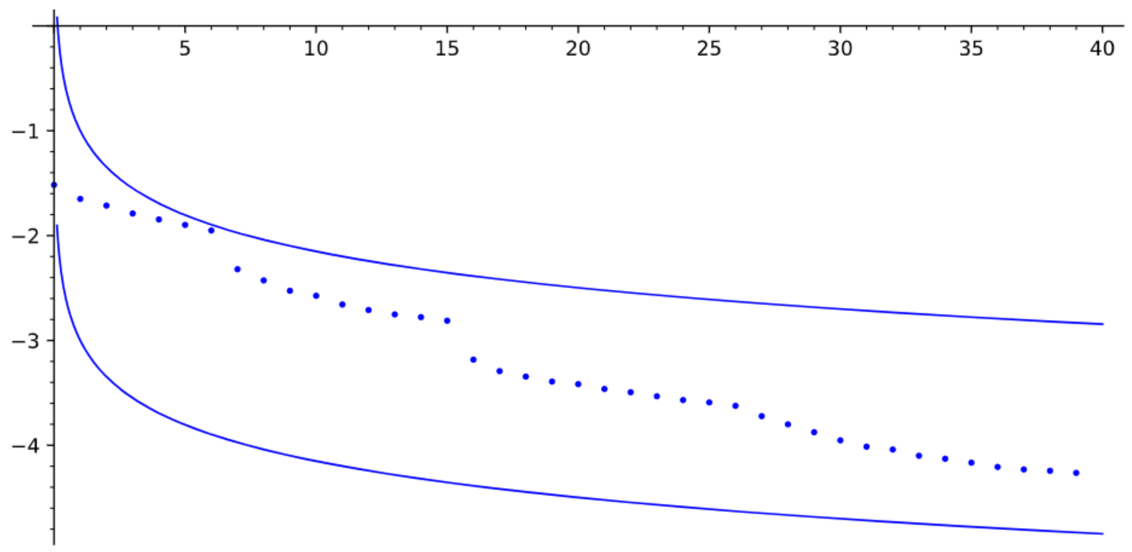
We know the graph is strongly related to the goal score M, and we also know the approximate number of perfect squares in M is sqrt(M). Then, I tried to graph log(P)/sqrt(M) for every P and M. (Note that P is the winning probability, and M is the goal score, and sqrt is square root) It will be shown as below:



After that, we know the graph looks like an exponential function over a square root. Then, I tried to draw a function of M to see the relationship. I set f(M) = -log(sqrt(M))-1, and I graph both of them on a single graph.



The curve seems pretty close, but it is not perfect. Then I choose to forget about finding a single function to conclude the relationship. I decided to find two functions which can make a bound to include all the points. I just changed the f(M) I got above by moving it down 2 units to get another function to make the boundary.



All the points on the graph are in the boundary, and the “curve” are similar. One thing is about the lower bound. I did not change the funcion that much, I just moved the starting point, so it might not be good anymore if we increase the value of M. I would like to look into that in the future.

Code for graph if needed:

```
log_win.append(log(win_r/sqrt(M)))
var('x')
g = Graphics()
g += list_plot(log_win)
g += plot(-log(sqrt(x))-1, (x, 0, 40))
g += plot(-log(sqrt(x))-3, (x, 0, 40))
g.show()
```

There are lots of interesting finding for the problem, and I would love to learn more about it.