Mathematics: Analysis and Approaches HL

Investigating Optimal Strategies in the Dice Game 'Snake Eyes'

Number of pages: 20 pages

Introduction

Dice games are a seemingly simple concept, however complex and unexpected behaviour can arise from them. When playing dice games, many people often rely on strategies which are based on emotions such as their 'feeling of luck' or other strategies that have not been adequately thought through. During our lessons, we were introduced to a few dice games to demonstrate to us how probabilities work with dice.

One of these dice games known is as 'Snake Eyes', which involves two dice and can be played by two or more players. During a turn, a player rolls the pair of dice, and can keep rolling the dice while adding up their sums. They can do so until they wish to stop. However, if they roll a '1' on one of the dice, they lose the turn and this does not get added to their total score, and if they roll two '1's (colloquially known as *Snake Eyes*), their total score gets reset to zero. The first player to reach or exceed the score of 100 wins the game.

While playing the game myself, I predominantly thought of two strategies, one in which you aim for a specific number of rolls in a particular turn, and another one, in which you aim for a specific score in a turn before stopping. Both these strategies can be effective, however this would require finding an optimal number of target rolls or stopping score, that minimises the number of turns to reach the goal of 100.

Thus, the aim of this exploration is to find this optimal target score or number of rolls. To do so, I will use probability theory to model the stochastic process of the game, and analytically find an optimum solution. I will then be able to apply my knowledge of computer programming to create a simulation and test the optimal strategy.

The rules of 'Snake Eyes'

The game can be played by two or more players, where the winner reaches a total score of 100 points. The player can roll as many times as they wish in a single turn, with the turn's total score being the sum of dice of each roll within the turn. Yet, there are some limitations, which are as follows:

1. The roll's sum will only be counted if neither of the dice rolls a '1'
2. The roll's sum will not be counted to the total if during the turn any dice rolls a '1'
3. If both dice roll a '1' during a turn, the players total score gets reset to zero

Investigating the optimal roll strategy

During a single roll, two dice are rolled together forming different combinations of numbers. These combinations (represented in figure 1a), can then be used to calculate the outcome or score of that single turn. This is simply the sum of the two dice (figure 1b), however it is important to notice that any combination containing a '1' results in a 0 according to the rules of the game.

Figure 1: Die roll combinations and corresponding turn scores

1,1	2,1	3,1	4,1	5,1	6,1
1,2	2,2	3,2	4,2	5,2	6,2
1,3	2,3	3,3	4,3	5,3	6,3
1,4	2,4	3,4	4,4	5,4	6,4
1,5	2,5	3,5	4,5	5,5	6,5
1,6	2,6	3,6	4,6	5,6	6,6

(a) die combinations

Turn score →

0	0	0	0	0	0
0	4	5	6	7	8
0	5	6	7	8	9
0	6	7	8	9	10
0	7	8	9	10	11
0	8	9	10	11	12

(b) turn scores

From figure 1b, it can be observed that 11 out of all combinations result in a zero, 1 combination results in a score of 4, 2 in a 5 and so on. Using figure 1b, a discrete probability distribution can be written down for the score of the first roll of the game which will be denoted as X_1.

Table 1: Discrete probability distribution of scores ruing the first roll ($r = 1$)

$\boldsymbol{x}$	0	1	2	3	4	5	6	7	8	9	10	11	12
$\boldsymbol{P(X_1 = x)}$	$\frac{11}{36}$	$\frac{0}{36}$	$\frac{0}{36}$	$\frac{0}{36}$	$\frac{1}{36}$	$\frac{2}{36}$	$\frac{3}{36}$	$\frac{4}{36}$	$\frac{5}{36}$	$\frac{4}{36}$	$\frac{3}{36}$	$\frac{2}{36}$	$\frac{1}{36}$

Notation: X_r will denote the random variable of the game score on the r^{th} roll ($r \in \mathbb{Z}^+$, $r \geq 1$).

The expected value of the first roll ($r = 1$) can then be derived as follows:

$$E(X_1) = \sum_{x=0}^{12} x \times P(X_1 = x) = 0\left(\frac{11}{36}\right) + 1\left(\frac{0}{36}\right) + \cdots + 12\left(\frac{1}{36}\right) = \frac{50}{9} \approx 5.5556 \ (5 \ s.f.)$$

Next we can generalise the expected value for r rolls. After r rolls, we should expect to have a score that is r times the expected value of the first roll $E(X_r) = r \times E(X_1)$, however this would not be correct, as if a '1' is rolled on any turn, the X goes to zero. To account for this, we can multiply the expected value by the probability that we have *not* rolled a '1' (disqualifying the turn) r times.

$$\text{Probability } r^{\text{th}} \text{ roll counts} = (\underbrace{P(\overline{X_1 = 0})}_{not \text{ zero}})^{r-1} = \left(1 - \frac{11}{36}\right)^{r-1}$$

Taking this probability into consideration, an expression can be written for the expected value of the turn score at r rolls as follows:

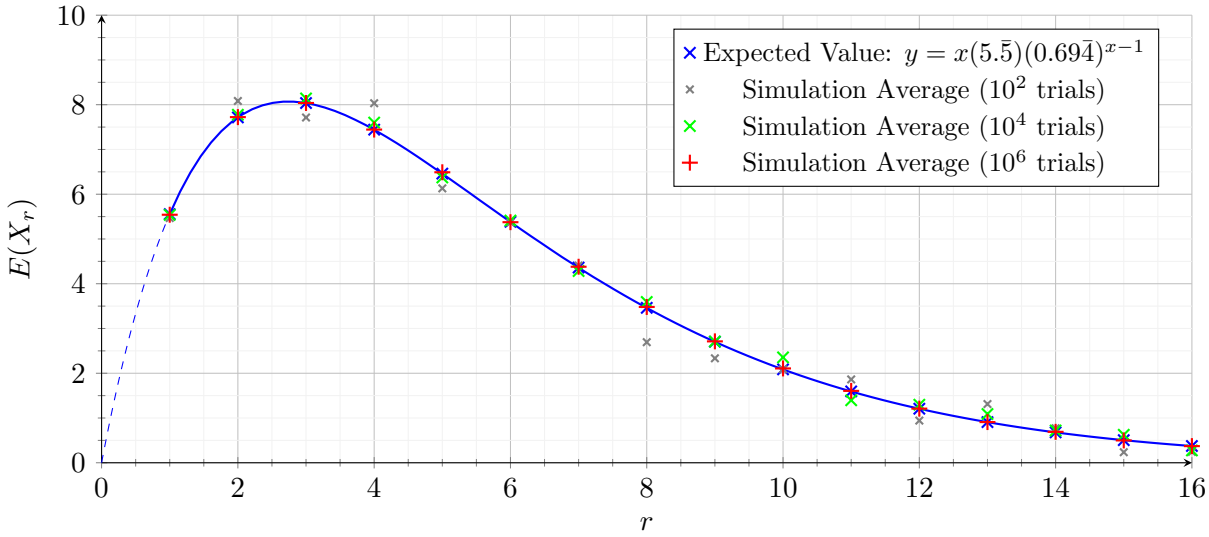
$$E(X_r) = E(X_1) \cdot r \left(1 - \frac{11}{36}\right)^{r-1} = \frac{50}{9} \cdot r \cdot \left(1 - \frac{11}{36}\right)^{r-1} \tag{1}$$

This can then be plotted to observe which value of r yields the highest expected value. The prediction can also be compared to my Python simulation (refer to appendix).

Table 2: Expected value calculation & simulation

r	$E(X_r)$	Simulation (10^3 trials)	Simulation (10^4 trials)	Simulation (10^6 trials)
1	5.5556	5.4680	5.5378	5.5417
2	7.7160	8.0830	7.7754	7.7228
3	8.0376	7.7140	8.1402	8.0386
4	7.4422	8.0330	7.6007	7.4444
5	6.4602	6.1310	6.3738	6.4905
6	5.3835	5.3700	5.4116	5.3750
7	4.3616	4.4060	4.2838	4.3797
8	3.4616	2.6940	3.5926	3.4812
...	...	...	...	...
24	0.0304	0.0000	0.0195	0.0304
25	0.0220	0.0000	0.0202	0.0219

Expected value of turn vs. target number rolls r



As can be seen in the plot, the optimal value of r is around 2-3. The maxima can be found by equating the derivative to zero:

$$\frac{d}{dr}E(X_r) = 0 \Rightarrow \text{using G.D.C } r = 2.7424 \ (5 \ s.f.)$$

r must be an integer ($r \in \mathbb{Z}^+$), so the closest value of r that would yield the highest $E(X_r)$ would be $r = 3$. It can be concluded that in a given turn, the player should aim for 3 consecutive rolls each turn of the game to maximise the number of points gained in the given turn. This strategy of maximising the expected score of a turn was described in Roters 1998, but for a simpler dice game similar to 'Snake Eyes' colloquially known as 'Pig' and only involves one die.

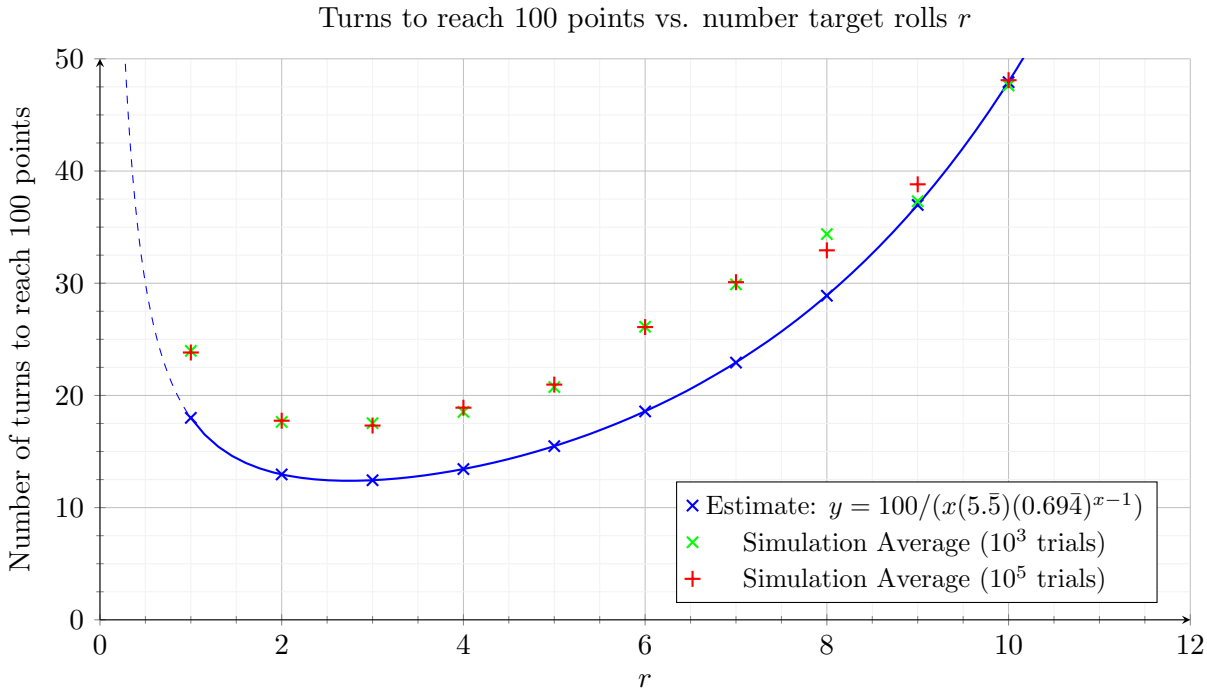
However, it is also important to consider the probability of losing all points by rolling *snake eyes*. By rolling more times, there is an increased probability of this occurring, which was not considered in this simplified model, which only considers an independent turn in the game. Likewise, the dice game in Roters 1998 does not have such a rule, and is thus not considered in the strategy. This raises the question, is it better to roll *less* than 3 times? This could be true, as the probability of losing all points increases with more rolls, and over an entire game this would have a negative effect. To determine this, I applied my computing knowledge and created a simulation in Python (refer to appendix).

Essentially, the simulation ran millions of games where, during a turn, the dice was rolled for a targeted number of rolls. The average number of turns to reach the goal of 100 total points was then calculated. In a turn, if a the goal is reached before all target rolls are used, then the game will stop as it is not necessary to continue. An estimate for this value could be found by dividing the goal of 100 points by the expected value of a given target roll strategy:

$$\text{Estimated turns to win} = \frac{100}{E(X_1) \cdot r \left(1 - \frac{11}{36}\right)^{r-1}}$$

Table 3: Expected and average turns to win

r	**Estimated Turns**	**Simulation Average Turns (10^3 trials)**	**Simulation Average Turns (10^5 trials)**
1	18.0000	23.9710	23.8256
2	12.9600	17.6370	17.7469
3	12.4416	17.5290	17.3134
4	13.4369	18.5290	18.9076
5	15.4793	20.7450	20.9664
6	18.5752	26.1180	26.0924
7	22.9271	29.8920	30.0941
8	28.8882	34.3780	32.9309
9	36.9769	37.3070	38.8154
10	47.9220	47.6250	48.0947
...	...	...	...
15	197.8136	102.3320	106.9874



Looking at the plot above, it can be observed that the values for the average number of rolls to win is higher in the simulation. This is due to the fact that the possibility of losing all points when rolling *snake eyes* was accounted for. Despite this, the value of $r = 3$ is still shown to be the most effective, as playing by this strategy of 3 rolls achieves a win in only 17.32 turns.

When simulating, I calculated averages at different numbers of trials, and it can be observed that the simulation quickly converges to an average value, even at low trial counts such as 10^3. While developing my program in Python, I was restricted by the relatively low performance of the Python interpreter. To overcome this challenge, I employed *Just-In-Time* compilation using the `numba` library, which utilises the LLVM compiler to optimise the Python script. With this I was able to make a high performance simulation that ran in parallel, enabling me to run simulations with millions of trials within minutes rather than hours.

It is also interesting to see that there is not a significant difference between targeting $r = 1$ and $r = 6$. The reason for this may be that the risk of losing all points is cancelled out to some extent by the higher expected value, however the average number of rolls to win for $r = 6$ is only minimally higher than that of $r = 1$. This could indicate that the impact of rolling *snake eyes* may not have been as significant as I had initially anticipated.

An alternative method to finding expected values

While researching, I came across Nica 2021, an article and its accompanying program to model the dice game 'Pig', which is similar to 'Snake Eyes', just it only involves a single die. However it is slightly simpler to model as it does not have a rule in which the player loses all their points. Nica uses a discrete-time Markov chain to model the process of rolling dice during a single turn. This article provided many useful ideas which I have applied to 'Snake Eyes' with some slight modifications required to accommodate the rules of 'Snake Eyes'.

In Nica 2021, a 'sequence of random variables' is used to denote the score of the player in a single turn of 'Pig', and the same can be done for this game.

Let $X_1, X_2, X_3, X_4, \ldots, X_r$ be a sequence of random variables that denoting total score at roll r.

These random variables observe the *Markov property*, which means that a current state is only dependant on the state preceding it, and can be determined from it.

$$\underbrace{P(X_r = x_r \mid X_{r-1} = x_{r-1}, X_{r-2} = x_{r-2}, \ldots, X_0 = x_0)}_{\text{current state's probability given all previous states}} \implies \underbrace{P(X_r = x_r \mid X_{r-1} = x_{r-1})}_{\substack{\text{current state's probability} \\ \text{given only the previous state}}}$$

With this established, we can find the probabilities of values of X_r given only the previous state, starting at X_1 (previously derived). By modifying some basic rules from Nica 2021, these probabilities can be iterated to find the desired discrete probability distribution of scores at a given number of target rolls.

Example 1: The probability of rolling a zero on the 2^{nd} turn is the probability of rolling any combination of the two dice containing a '1' and having after not doing so for the previous turn. For example, rolling a '1' after rolling any score on the first turn (4 to 12).

Example 2: The probability of the the total score being equal to 8 points at 2 rolls $P(X_2 = 8)$ is dependant on the previous state on the first roll. In this case, it is only possible to arrive at 8 by rolling two 4s (the minimum number of points available in a roll). Thus, the probability of rolling 8 on the 2^{nd} turn is the probability of rolling two 4s.

Rule 1: Probability of losing turn

There are two ways in which the score turn's score can be set to zero:

1. Rolling a zero on the current turn, having not rolled zero at any previous roll
2. Having rolled a zero on the previous roll

The probability of losing a turn when targeting r rolls, can then be found by propagating the probability from the previous roll.

Figure 2 on the next page, illustrates that each state in the previous roll can lead to a turn score of zero, if a zero causing roll (involving rolling a one) is rolled.

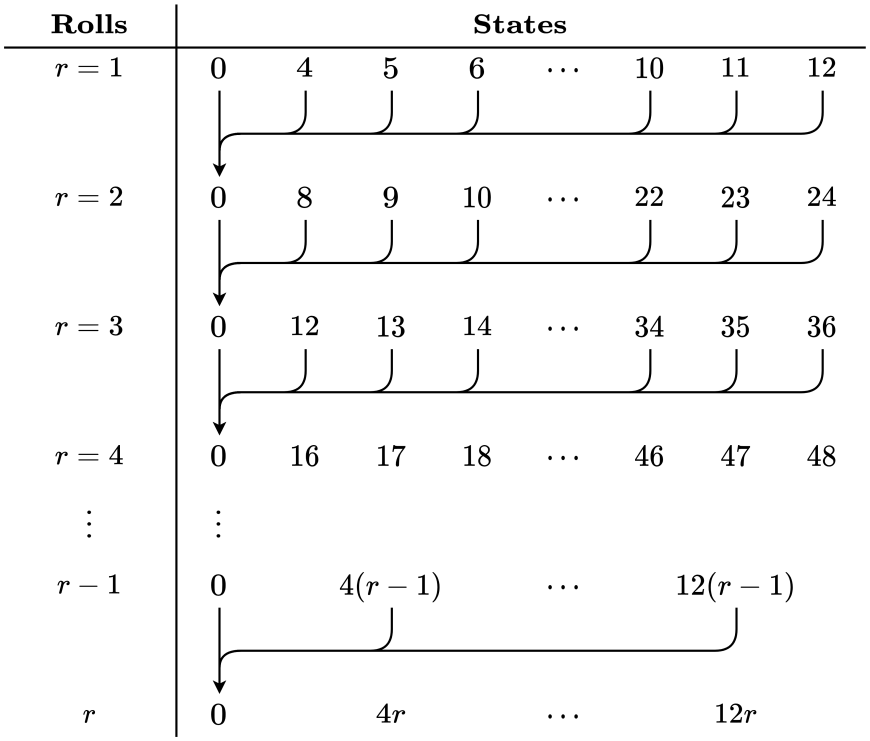


Figure 2: Propagation of probabilities for $X_r = 0$

Looking at figure 2, the probabilities of $X_r = 0$ can be found for each value of r, and can be generalised as will be demonstrated on the next page.

Notation: Here $p(x)$ will represent the probability of rolling a score x in a single dice roll. This is effectively the same thing as the probability distribution found in table 1. For example $p(0) = \frac{11}{36}$, and $p(4) = \frac{1}{36}$ and so on.

$$r = 1: \qquad P(X_1 = 0) = p(0) = \frac{11}{36}$$

$$\begin{aligned} r = 2: \qquad P(X_2 = 0) &= P(X_1 = 0) + p(0) \cdot P(X_1 = 4) + \cdots + p(0) \cdot P(X_1 = 12) \\ &= P(X_1 = 0) + p(0) \cdot (P(X_1 = 4) + \cdots + P(X_1 = 12)) \\ &= P(X_1 = 0) + p(0) \cdot \sum_{i=4}^{12} P(X_1 = i) \end{aligned}$$

$$\begin{aligned} r = 3: \qquad P(X_3 = 0) &= P(X_2 = 0) + p(0) \cdot P(X_1 = 8) + \cdots + p(0) \cdot P(X_2 = 24) \\ &= P(X_2 = 0) + p(0) \cdot \sum_{i=8}^{24} P(X_2 = i) \end{aligned}$$

$$\begin{aligned} r = 4: \qquad P(X_4 = 0) &= P(X_3 = 0) + p(0) \cdot P(X_3 = 12) + \cdots + p(0) \cdot P(X_3 = 36) \\ &= P(X_3 = 0) + p(0) \cdot \sum_{i=12}^{36} P(X_3 = i) \end{aligned}$$

$$r - 1: \quad P(X_{r-1} = 0) = P(X_{r-1} = 0) + p(0) \cdot \sum_{i=4(r-2)}^{12(r-2)} P(X_{r-2} = i)$$

$$r: \qquad P(X_r = 0) = P(X_{r-1} = 0) + p(0) \cdot \sum_{i=4(r-1)}^{12(r-1)} P(X_{r-1} = i)$$

The generalisation for $P(X_r = x)$ where $x = 0$ is then:

$$r^{\text{th}} \text{ roll: } P(X_r = 0) = \underbrace{P(X_{r-1} = 0)}_{\text{at zero previously}} + \underbrace{p(0) \cdot \sum_{i=4(r-1)}^{12(r-1)} P(X_{r-1} = i)}_{\text{roll zero from all possible previous states}}$$

Note that the range of values for X_r are $4r \leq X_r \leq 12r$ since the minimum possible roll value is 4 and the maximum is 12. So hence $4(r-1) \leq X_{r-1} \leq 12(r-1)$. Any value outside of this range will result in a probability of zero as they are impossible to reach by rolling the dice r times.

It is also possible to model the probability of a turn r being a zero, by using a geometric distribution. Considering the act of not rolling a zero a *success*.

$$P(X_r = 0) = \overline{(p_{\text{success}})^r}$$

The probability of having a zero score at roll r is the complement of *success* occurring r times, implying that at a particular point, success did not occur, which would mean that the whole turn in the game's total will be 0. Now the values can be substituted. Since p_{success} is effectively the complement of $p(0)$, and $p(0) = \frac{11}{36}$, the equation can be rewritten as:

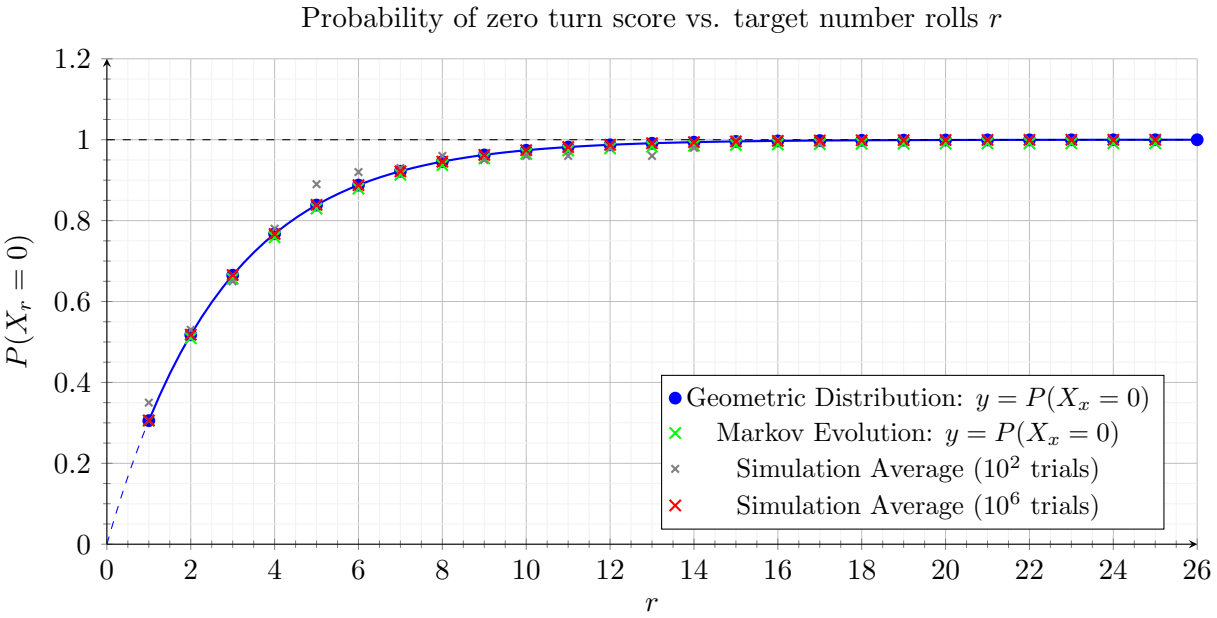
$$P(X_r = 0) = 1 - (1 - p(0))^r = 1 - \left(1 - \frac{11}{36}\right)^r \implies P(X_r = 0) = 1 - \left(\frac{25}{36}\right)^r$$

To confirm that each of these methods is correct, I made another simulation in Python. Results from each method can be plotted as shown on the next page. It is evident that each method's results are identical, showing they are all valid. It is also interesting to see that the probability rather quickly approaches the asymptotic value of 1, arriving there at around $r = 14$.

Table 4: Probability of losing a turn at r rolls

r	Markov $P(X_r = 0)$	Geometric $P(X_r = 0)$	Simulation (10^2 trials)	Simulation (10^4 trials)	Simulation (10^6 trials)
1	0.3056	0.3056	0.3500	0.3063	0.3060
2	0.5093	0.5177	0.5300	0.5182	0.5180
3	0.6564	0.6651	0.6500	0.6700	0.6655
4	0.7587	0.7674	0.7800	0.7717	0.7670
5	0.8298	0.8385	0.8900	0.8434	0.8390
6	0.8791	0.8878	0.9200	0.8881	0.8877
7	0.9134	0.9221	0.9300	0.9186	0.9218
8	0.9372	0.9459	0.9600	0.9477	0.9458
9	0.9537	0.9624	0.9500	0.9634	0.9624
10	0.9652	0.9739	0.9600	0.9768	0.9740
11	0.9732	0.9819	0.9600	0.9806	0.9818
12	0.9787	0.9874	0.9800	0.9866	0.9876
13	0.9825	0.9913	0.9600	0.9915	0.9913
14	0.9852	0.9939	0.9800	0.9935	0.9939
15	0.9871	0.9958	1.0000	0.9953	0.9958
...	...	...	...	...	...
25	0.9912	0.9999	1.0000	1.0000	0.9999

Note: The minor difference between the Markov and Geometric probabilities is negligible, as it is due to floating-point precision rounding when using a Python script.



Rule 2: Probability of X_r being any value x

The probability of rolling any value x on roll r is every possible combination of the current roll, that when summed with the previous total X_{r-1} equals x.

$$
\begin{aligned}
X_2 = 8 \quad &\longleftarrow \quad X_1 = 4 \ \ \& \ \ \text{roll } 4 \\
X_2 = 9 \quad &\longleftarrow \quad X_1 = 4 \ \ \& \ \ \text{roll } 5 \ \textbf{or} \ X_1 = 4 \ \ \& \ \ \text{roll } 4 \\
X_2 = 10 \quad &\longleftarrow \quad X_1 = 4 \ \ \& \ \ \text{roll } 6 \ \textbf{or} \ X_1 = 5 \ \ \& \ \ \text{roll } 5 \ \textbf{or} \ X_1 = 6 \ \ \& \ \ \text{roll } 4 \\
&\vdots \\
X_2 = 22 \quad &\longleftarrow \quad X_1 = 10 \ \ \& \ \ \text{roll } 12 \ \textbf{or} \ X_1 = 11 \ \ \& \ \ \text{roll } 11 \ \textbf{or} \ X_1 = 12 \ \ \& \ \ \text{roll } 10 \\
X_2 = 23 \quad &\longleftarrow \quad X_1 = 11 \ \ \& \ \ \text{roll } 12 \ \textbf{or} \ X_1 = 11 \ \ \& \ \ \text{roll } 12 \\
X_2 = 24 \quad &\longleftarrow \quad X_1 = 12 \ \ \& \ \ \text{roll } 12
\end{aligned}
$$

Using this idea, the case for the probability of finding x at r rolls can be expressed as such:

$$P(X_r = x) = \sum_{i=\max(x-12,4)}^{x-4} P(X_{r-1} = i) \cdot p(x - i)$$

The $\max(x-12, 4)$ ensures that the minimum possible score value for the previous roll is not negative, and is instead 4 (the minimum) which was taken from the implementation in Nica 2021, and was adapted to the 'Snake Eyes' problem.

Combining these rules together, a piecewise function can be made finding the probability of a certain score at a particular roll r:

$$P(X_r = x) = \begin{cases} P(X_{r-1} = 0) + p(0) \cdot \sum_{i=4(t-1)}^{12(t-1)} P(X_{r-1} = i) & \text{if } x = 0 \\ \sum_{i=\max(x-12,4)}^{x-4} P(X_{r-1} = i) \cdot p(x-i) & \text{otherwise} \end{cases} \tag{2}$$

Using this function, another function can be created to find the expected value of a turn's score given its target number of rolls r. Firstly, it must be noted that the range of possible values after rolling r times is between $4r$ and $12r$.

$$4r \leq X_r \leq 12r$$

The lower bound of $4r$ would be reached if we rolled r consecutive '4s', the minimum possible score during a roll, and vice versa for the upper bound, with consecutive '12s' being rolled.

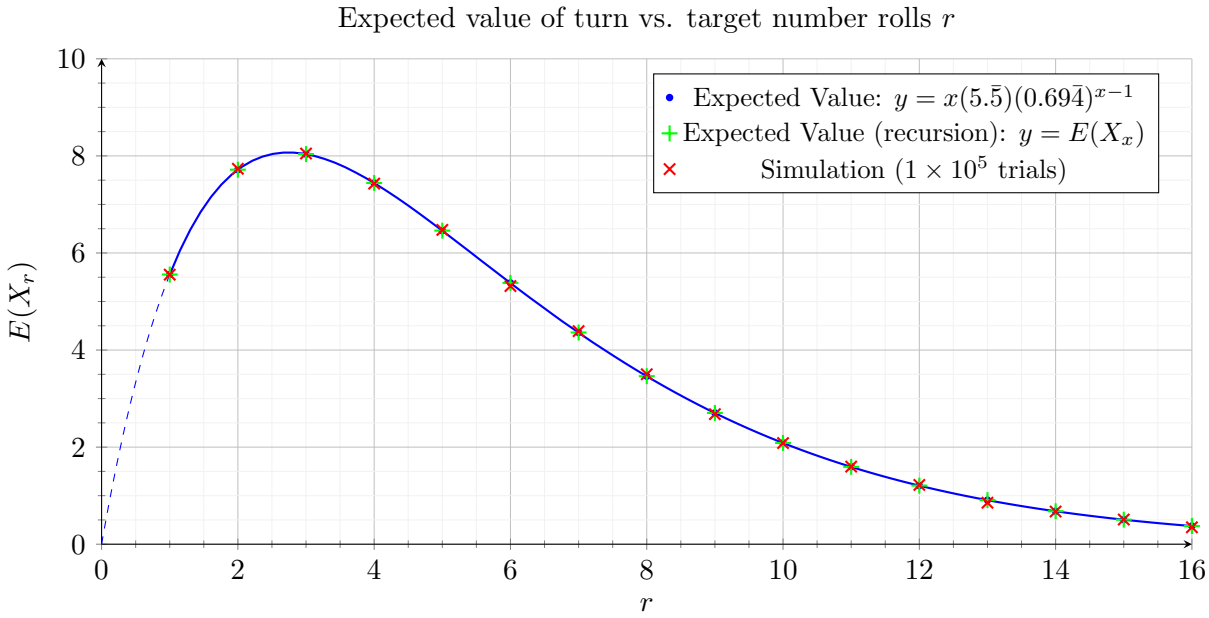
When calculating the expected value, the probability of rolling zero can also be ignored, since the term cancels out due to multiplication by zero. The function for expected value is as follows:

$$E(X_r) = \cancel{0 \cdot P(X_r = 0)} + \sum_{i=4r}^{12r} i \cdot P(X_r = i) \implies E(X_r) = \sum_{i=4r}^{12r} i \cdot P(X_r = i)$$

I created a Python script that calculates the expected values. This can be done very elegantly in Python, as a recursive function can be implemented. However, while implementing this in code I encountered a problem where the program began to perform very poorly at high values of r, and this was due to the recursion causing the function to be called too many times. To fix this I had to research *memoisation* which is a technique to optimise the way a recursive function works by caching its results. This significantly improved the performance of my program.

```
@lru_cache(maxsize=10000)
def P(r:int, x:int) -> float:
    if r == 1:
        if x > 12: return 0
        else: return p[x]
    else:
        if x == 0: return P(r-1,0) + p[0]*sum([P(r-1,i) for i in range(4*(r-1), 12*(r-1))])
        else: return sum([P(r-1,i)*p[x-i] for i in range(max(x-12,4),x-4+1)])

def E(r: int) -> float:
    return sum([i*P(r,i) for i in range(4*r, 12*r +1)])
```



As can be seen in the plot below, the new method (2) of calculating expected values (in green) is identical to the previous method (1), and also matches perfectly with the results of the simulation. From this it can be concluded that this new method is equivalent to the previous method of calculating the expected value given the target rolls r.

Investigating the target score strategy

The other strategy that I thought of for 'Snake Eyes' was the *target roll* strategy. The objective would be to find the optimal stopping score for a turn. A higher target score involves higher risk of losing the turn, however the reward associated with this risk is higher. In this sense, the player must choose the optimal risk:reward ratio, or how 'greedy' they want to be.

Using the same method of Markov chain evolution as was done in Nica 2021, the expected value of a turn can be found, given the number of target rolls with some slight modifications. The objective is to find the probability of scores at or above the target score which will be referred to as s.

Rule 3: Probability of scores above s

The following case exists for when $x \geq s$:

$$P(X_r = x) = \underbrace{P(X_{r-1} = x)}_{\text{previously at } x} + \underbrace{\sum_{i=\max(x-12,4)}^{\min(x-4,s-1)} P(X_{r-1} = i) \cdot p(x-i)}_{\text{combinations of past and current rolls to reach/exceed } s}$$

The probability of being at x at roll r when x will be s or greater than s is that that it was already at x previously, or it has reached x from a state below s.

Now with this third rule, the function $P(X_r = x)$ can be modified to accommodate the target roll strategy:

$$P(X_r = x) = \begin{cases} P(X_{r-1} = 0) + p(0) \cdot \sum_{i=4(t-1)}^{12(t-1)} P(X_{r-1} = i) & \text{if } x = 0 \\ \sum_{i=\max(x-12,4)}^{x-4} P(X_{r-1} = i) \cdot p(x-i) & \text{if } 0 < x < s \\ P(X_{r-1} = x) + \sum_{i=\max(x-12,4)}^{\min(x-4,s-1)} P(X_{r-1} = i) \cdot p(x-i) & \text{if } s \leq x < s+12 \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

Notice that the third rule applies to only up to $s + 12$. This is because it is unnecessary to achieve a higher score than s by 12 points, as this would mean rolling another time, when already at s.

Next, the expected value of the roll score can be found as follows:

$$E(X_u) = \sum_{i=s}^{s+12} i \cdot P(X_u = i) \qquad u = \left\lceil \frac{s}{4} \right\rceil \text{ (upper bound of rolls)}$$

Notice that the roll r is the ceiling of the target score divided by 4. This is the minimum number of rolls required to meet or exceed the target score. The possible values of X are also between s and $s + 12$, as only these values are needed to calculate the expected value. Again these functions can be implemented in Python as follows:

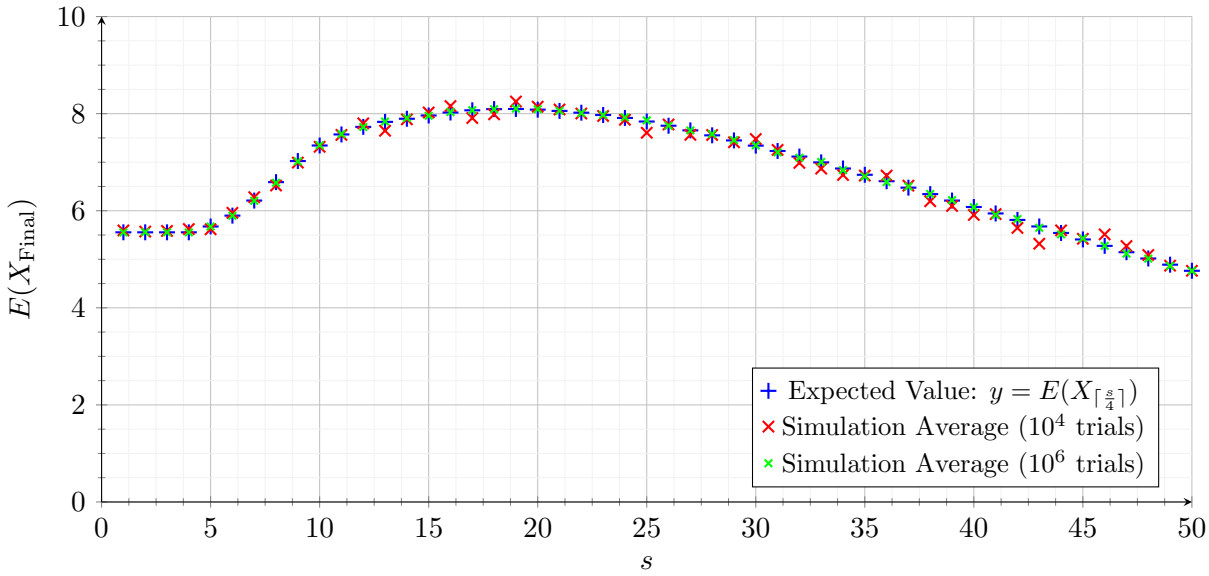
```
@lru_cache(maxsize=10000)
def Ps(r:int, x:int, s:int) -> float:
    if r == 1:
        if x > 12: return 0
        else: return p[x]
    else:
        if x == 0:
            return Ps(r-1,0,s) + p[0]*sum([Ps(r-1,i,s) for i in range(4*(r-1), 12*(r-1))])
        if x < s:
            return sum([Ps(r-1,i,s)*p[x-i] for i in range(max(x-12,4),x-4+1)])
        else:
            return Ps(r-1,x,s) + sum(
                [p[x-i]*Ps(r-1,i,s) for i in range(max(x-12,4), min(x-4, s-1)+1)])

def Es(s: int) -> float:
    r = np.ceil(s/4)
    return sum([i*Ps(r,i,s) for i in range(s, s+12+1)])
```

Table 5: Expected values and target score s

s	Expected Score	Simulation (10^2 trials)	Simulation (10^4 trials)	Simulation (10^6 trials)
1	5.5556	5.3900	5.5945	5.5550
...	...	...	...	...
17	8.0694	6.9600	7.9102	8.0724
18	8.0936	9.7400	7.9866	8.0917
19	8.0970	7.3700	8.2503	8.1042
20	8.0834	8.1800	8.1439	8.0914
21	8.0569	7.4100	8.0864	8.0584
22	8.0203	9.0500	8.0030	8.0125
23	7.9724	8.6200	7.9523	7.9738
...	...	...	...	...
50	4.7629	6.3800	4.7629	4.7619

Expected value of turn vs. target score s



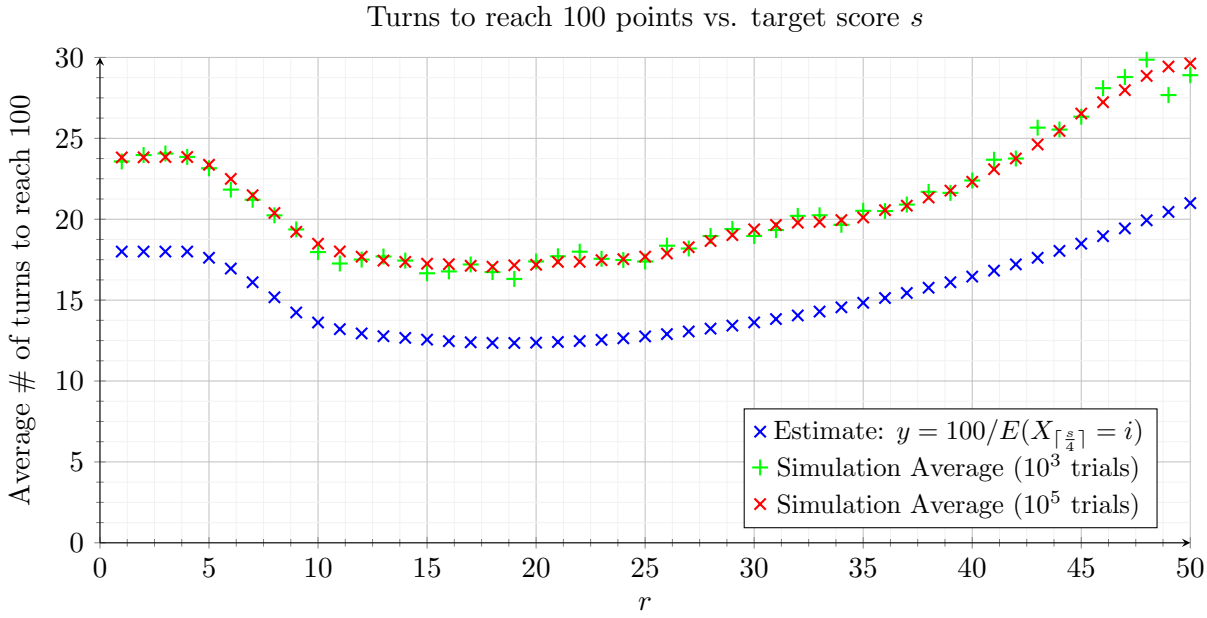
Looking at the plotted expected values, it is visible that there is a clear peak at a target score of 19. It is also good to see that the calculations are correct as they match the simulation averages.

Again, the value of $s = 19$ optimises the score during a single turn. Similar to the target roll strategy, the increased target score, causes an increased number of rolls as a result. This in turn causes the probability of rolling *snake eyes* increase, which, over the long run could make the strategy of aiming for 19 points per turn less suitable and maybe a higher or lower value may be optimal. To determine this, I made another simulation that found the average number of turns to win a game (reach 100 points), so that each value of s can be compared.

Table 6: Turns to win game and target score s

s	Estimated Turns	Simulation (10e3 trials)	Simulation (10e5 trials)
1	18.0000	23.5730	23.8183
...	...	...	...
15	12.5586	16.6600	17.2512
16	12.4628	16.7740	17.2204
17	12.3925	17.2030	17.1211
18	12.3555	16.7320	17.0628
19	12.3503	16.3070	17.1513
20	12.3711	17.3900	17.1850
21	12.4117	17.7110	17.3651
22	12.4684	17.9940	17.3675
...	...	...	...
50	20.9954	28.9030	29.6294

Looking at the plot, it can be seen that 19 is still the optimal target score, reaching the goal of 100 points in an average of 17.0628 turns whereas the target roll strategy yielded a minimum average turns of 17.3133. There is also a noticeable difference between the simulation result and the estimate, which is due to the rolling of *snake eyes* being factored in.



A reason why the target score strategy marginally outperforms the target roll strategy could be that the player is able to stop rolling in a turn once they reach a certain point, since if they are *lucky* and reach a high turn score in the first couple rolls, they can end the turn early, reducing the likelihood of rolling *snake eyes* and losing all their points. On the other hand, in the target roll strategy, this number is fixed, and so the likelihood of snake eyes cannot be reduced as it can be with the target score strategy.

Testing the theory on biased dice

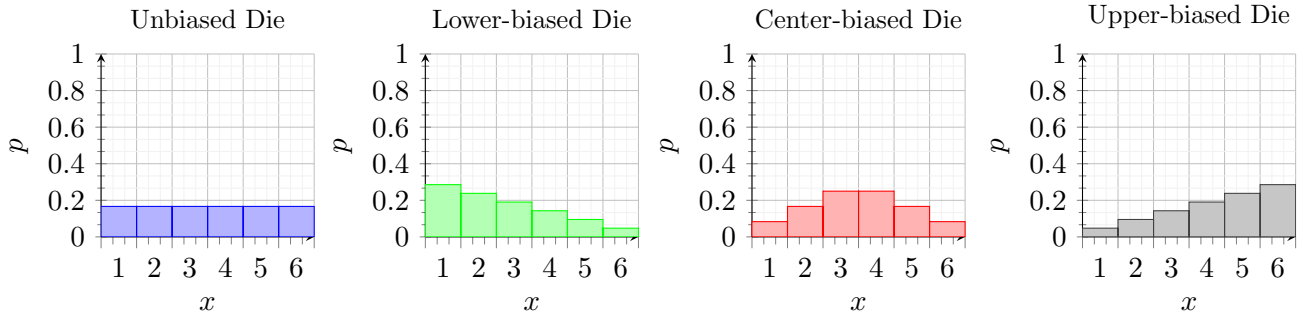
Up until this point, the types of dice investigated were unbiased, meaning that each number on each die is equally probable and is unweighted. However, it is possible to extend this model to games of 'Snake Eyes' involving biased or weighted dice. This is possible by simply recalculating the discrete probability distribution for a single dice roll as was done in table 1.

To do so, I have devised 3 types of biased dice which are as follows in table 7:

Table 7: Probability distributions of biased dice

Die Type	1	2	3	4	5	6
Unbiased	$\frac{1}{21}$	$\frac{1}{21}$	$\frac{1}{21}$	$\frac{1}{21}$	$\frac{1}{21}$	$\frac{1}{21}$
Lower-biased	$\frac{1}{21}$	$\frac{2}{21}$	$\frac{3}{21}$	$\frac{4}{21}$	$\frac{5}{21}$	$\frac{6}{21}$

Die Type	1	2	3	4	5	6
Center-biased	$\frac{1}{12}$	$\frac{2}{12}$	$\frac{3}{12}$	$\frac{3}{12}$	$\frac{2}{12}$	$\frac{1}{12}$
Upper-biased	$\frac{6}{21}$	$\frac{5}{21}$	$\frac{4}{21}$	$\frac{3}{21}$	$\frac{2}{21}$	$\frac{1}{21}$

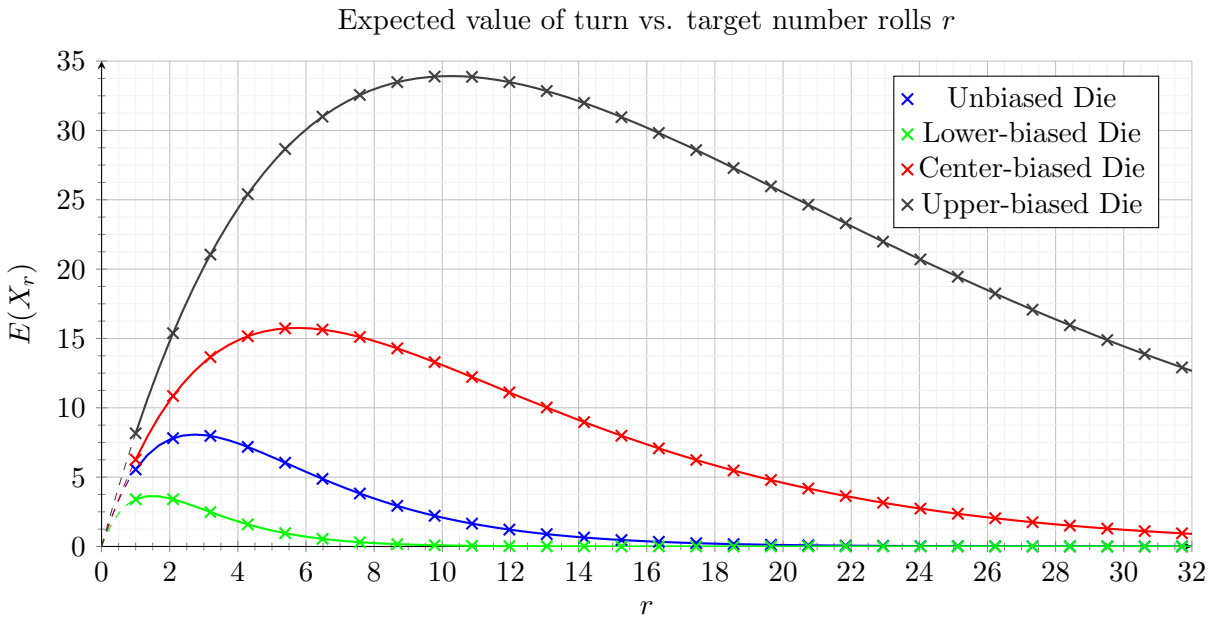


Using the equation (1) or (2), the expected values given target rolls r, as well as the simulation to compare and verify the results.

Table 8: Expected score values with target rolls r for biased dice

r	**Unbiased Die**	**Sim. Average**	**Lower-biased Die**	**Sim. Average**	**Center-biased Die**	**Sim. Average**	**Upper-biased Die**	**Sim. Average**
1	5.5556	5.5610	3.4014	3.3989	6.2639	6.2622	8.1633	8.1600
2	7.7161	7.7237	3.4708	3.4702	10.5268	10.5396	14.8086	14.8171
3	8.0376	8.0575	2.6562	2.6593	13.2682	13.2778	20.1478	20.1615
4	7.4422	7.4372	1.8069	1.8137	14.8653	14.8553	24.3662	24.3906
5	6.4602	6.4531	1.1524	1.1703	15.6137	15.6168	27.6261	27.6018
6	5.3835	5.3910	0.7055	0.7087	15.7438	15.7306	30.0692	30.0271
7	4.3616	4.3320	0.4200	0.4171	15.4340	15.4142	31.8193	31.8046
8	3.4616	3.4454	0.2449	0.2364	14.8216	14.7871	32.9840	33.0282
...	...	...	...	...	...	...	...	...
23	0.0419	0.0433	0.0000	0.0000	3.1324	3.1538	21.9413	21.9274
24	0.0304	0.0331	0.0000	0.0000	2.7465	2.7361	20.7667	20.6633
25	0.0220	0.0231	0.0000	0.0000	2.4040	2.4187	19.6208	19.6287

The data gathered in table 8 can then be plotted to compare the optimal number of target rolls depending on various biased dice. The following plot shows this below:



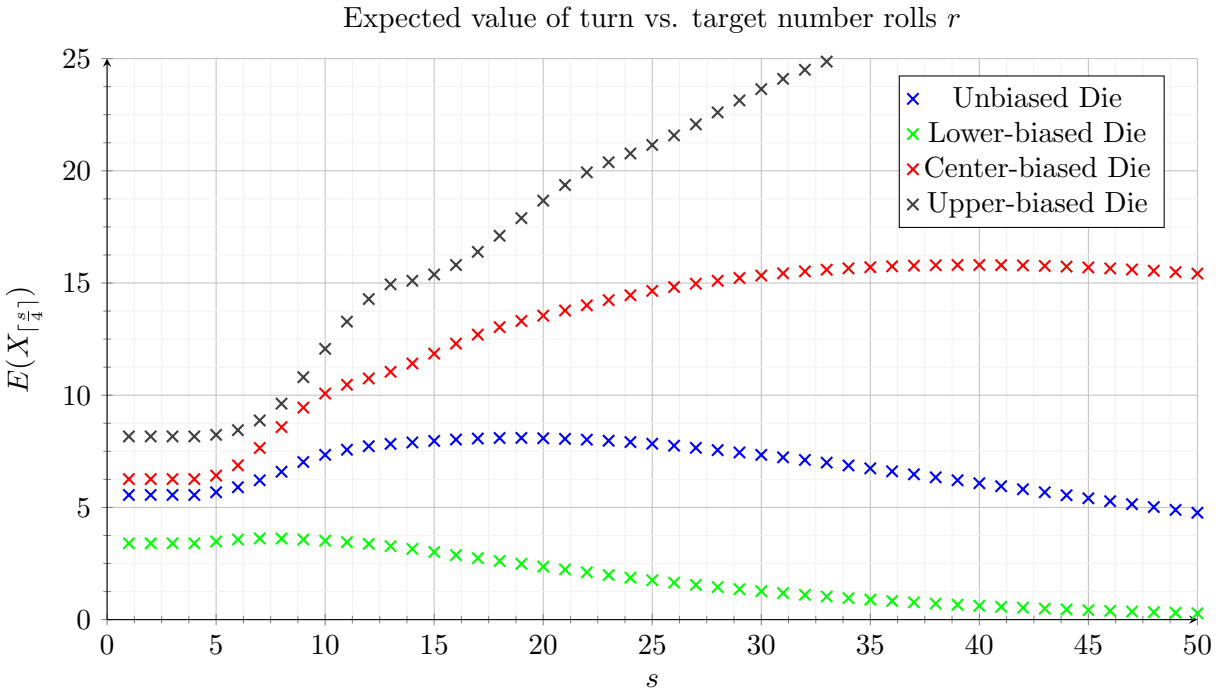
It can be seen that dice that are weighted towards higher values have much higher expected values, but also at higher numbers of target rolls. This is to be expected as the probability of rolling a 1 is lower, meaning that there is a lower risk of losing the turn or rolling snake eyes. Specifically, the optimal score for the lower-biased dice is $r = 2$, unbiased dice is $r = 3$, center-biased die is $r = 6$, and upper-biased die is $r = 10$.

A similar investigation can be conducted with the target score strategy. Below table 9 contains the results of the simulation with different target scores s and corresponding simulations which match the prediction made with the function from (2).

Table 9: Expected score values with target score r for biased dice

s	**Unbiased Die**	**Sim. Average**	**Lower-biased Die**	**Sim. Average**	**Center-biased Die**	**Sim. Average**	**Upper-biased Die**	**Sim. Average**
1	5.5556	5.5550	3.4014	3.3985	6.2639	6.2647	8.1633	8.1587
2	5.5556	5.5601	3.4014	3.3972	6.2639	6.2631	8.1633	8.1629
3	5.5556	5.5565	3.4014	3.4000	6.2639	6.2630	8.1633	8.1601
4	5.5556	5.5534	3.4014	3.4015	6.2639	6.2642	8.1633	8.1603
5	5.6759	5.6744	3.4831	3.4850	6.4201	6.4190	8.2339	8.2342
...	...	...	...	...	...	...	...	...
47	5.1457	5.1156	0.3602	0.3583	15.6031	15.6536	29.3782	29.3134
48	5.0166	5.0133	0.3330	0.3403	15.5475	15.5367	29.6402	29.6264
49	4.8890	4.8619	0.3076	0.3093	15.4862	15.5210	29.8917	29.8616
50	4.7629	4.7619	0.2841	0.2812	15.4197	15.4358	30.1301	30.0931

The data from table 9 can then be plotted as follows:



As seen in the plot, the dice biased upwards have much higher expected values and occur at higher target scores, which was also the case with target rolls. The lower-biased die peaks at $s = 8$, the unbiased at $s = 19$, the center-biased at $s = 41$, and the upper-biased (not shown on plot) peaks at $s = 88$.

Conclusion, Evaluation & Reflection

It can be concluded that the optimal number of target rolls is 3, and the optimal target score is 19 for games involving unbiased dice. Between these two, the better strategy is the target score strategy, as it wins the game in 17.04 turns on average, while the target roll strategy is slightly higher at 17.32 turns.

The strategies investigated in this paper were rather simple as they only consider an isolated roll in the game and how a player should approach it optimally, which is also investigated in Roters 1998 for 'Pig'. More advanced strategies can exist, especially once the two or more player aspect is considered. For example, if a player is behind their opponent, and the opponent is close to winning, it would make sense for the player to be more 'greedy', and choose a higher target score to aim for in the next turn. When considering the current state of the game including other players, the optimal strategy becomes more difficult to find, however the findings here can serve as a starting point. Research exists surrounding the dice game 'Pig', suggesting strategies on how to minimise the number of turns to win the game as was done in Haigh and Roters 2000, and also considering the other players' state(s) as in Crocce and Mordecki 2009.

Despite this, the strategies investigated in this paper are still effective against new players of the game, as they are unfamiliar with when to stop, mainly relying on emotions. The use of this simple strategy grants a player a slight competitive edge.

Throughout the process of researching for this paper, I was able to apply my knowledge of programming and computer science, and learned about techniques such as *memoisation* and *Just-In-Time* compilation to overcome challenges I faced.

Future Work

As stated previously, the strategies investigated only consider the game on a single roll. After researching dice game optimisation, I found that optimal strategies for dice games are often found using machine learning, specifically reinforcement learning. Without getting into to much detail, this involves the Markov Decision Process, and the Bellman equation. This essentially finds an optimal *policy* which dictates the most optimal next move given the current state (Ye 2022). It would be interesting to see how much of an improvement such a method could have. Again, research has been conducted on simpler dice games such as the aforementioned 'Pig', however it could be possible to extend this to more complex games such as 'Snake Eyes'.

In this paper I also investigated biased dice but additional variations of 'Snake Eyes' could be investigated, such as versions involving more than two dice, or dice with more than 6 sides. The effect of changing the goal from something other than 100 may also have implications on how to optimally approach the game.

References

Crocce, Fabian and Ernesto Mordecki (2009). *Optimal minimax strategy in a dice game*. DOI: 10.48550/ARXIV.0912.5518. URL: https://arxiv.org/abs/0912.5518.

Haigh, John and Markus Roters (2000). "Optimal Strategy in a Dice Game". In: *Journal of Applied Probability* 37.4, pp. 1110–1116. ISSN: 00219002. URL: http://www.jstor.org/stable/3215500.

Nica, Mihai (Apr. 2021). *Pig_Simulator.ipynb*. URL: https://colab.research.google.com/drive/1Mdk26YQYhUsfDSFNYzAL-qA9g_QmFooa?usp=sharing#scrollTo=4VQQk2fTe_sF.

Roters, Markus (1998). "Optimal Stopping in a Dice Game". In: *Journal of Applied Probability* 35.1, pp. 229–235. ISSN: 00219002. URL: http://www.jstor.org/stable/3215561.

Ye, Andre (Mar. 2022). *A crash course in Markov decision processes, the bellman equation, and dynamic programming*. URL: https://medium.com/mlearning-ai/a-crash-course-in-markov-decision-processes-the-bellman-equation-and-dynamic-programming-e80182207e85.

Appendices

Full Tables

Table 2: Expected value calculation & simulation

r	$E(X_r)$	Simulation (10^3 trials)	Simulation (10^4 trials)	Simulation (10^6 trials)
1	5.5556	5.4680	5.5378	5.5417
2	7.7160	8.0830	7.7754	7.7228
3	8.0376	7.7140	8.1402	8.0386
4	7.4422	8.0330	7.6007	7.4444
5	6.4602	6.1310	6.3738	6.4905
6	5.3835	5.3700	5.4116	5.3750
7	4.3616	4.4060	4.2838	4.3797
8	3.4616	2.6940	3.5926	3.4812
9	2.7044	2.3340	2.7092	2.7125
10	2.0867	2.0900	2.3547	2.1075
11	1.5940	1.8620	1.3959	1.6059
12	1.2076	0.9430	1.2968	1.2094
13	0.9085	1.3120	1.0892	0.9089
14	0.6794	0.7700	0.7211	0.6912
15	0.5055	0.2360	0.6253	0.4992
16	0.3745	0.2430	0.2757	0.3720
17	0.2763	0.5090	0.3030	0.2779
18	0.2032	0.0000	0.1599	0.2037
19	0.1489	0.6120	0.1557	0.1495
20	0.1089	0.4650	0.1419	0.1147
21	0.0794	0.0000	0.0513	0.0786
22	0.0577	0.0000	0.0692	0.0598
23	0.0419	0.0000	0.0370	0.0404
24	0.0304	0.0000	0.0195	0.0304
25	0.0220	0.0000	0.0202	0.0219

Table 3: Expected and average turns to win

r	Estimated Turns	Simulation Average Turns (10^3 trials)	Simulation Average Turns (10^5 trials)
1	18.0000	23.9710	23.8256
2	12.9600	17.6370	17.7469
3	12.4416	17.5290	17.3134
4	13.4369	18.5290	18.9076
5	15.4793	20.7450	20.9664
6	18.5752	26.1180	26.0924
7	22.9271	29.8920	30.0941
8	28.8882	34.3780	32.9309
9	36.9769	37.3070	38.8154
10	47.9220	47.6250	48.0947
11	62.7343	61.9710	62.2256
12	82.8092	87.2600	83.0218
13	110.0726	97.2430	100.0597
14	147.1827	99.2760	106.1584
15	197.8136	102.3320	106.9874

Table 4: Probability of losing a turn at r rolls

r	Markov $P(X_r = 0)$	Geometric $P(X_r = 0)$	Simulation (10^2 trials)	Simulation (10^4 trials)	Simulation (10^6 trials)
1	0.3056	0.3056	0.3500	0.3063	0.3060
2	0.5093	0.5177	0.5300	0.5182	0.5180
3	0.6564	0.6651	0.6500	0.6700	0.6655
4	0.7587	0.7674	0.7800	0.7717	0.7670
5	0.8298	0.8385	0.8900	0.8434	0.8390
6	0.8791	0.8878	0.9200	0.8881	0.8877
7	0.9134	0.9221	0.9300	0.9186	0.9218
8	0.9372	0.9459	0.9600	0.9477	0.9458
9	0.9537	0.9624	0.9500	0.9634	0.9624
10	0.9652	0.9739	0.9600	0.9768	0.9740
11	0.9732	0.9819	0.9600	0.9806	0.9818
12	0.9787	0.9874	0.9800	0.9866	0.9876
13	0.9825	0.9913	0.9600	0.9915	0.9913
14	0.9852	0.9939	0.9800	0.9935	0.9939
15	0.9871	0.9958	1.0000	0.9953	0.9958
16	0.9883	0.9971	1.0000	0.9984	0.9971
17	0.9892	0.9980	0.9900	0.9981	0.9980
18	0.9899	0.9986	1.0000	0.9983	0.9986
19	0.9903	0.9990	1.0000	0.9985	0.9990
20	0.9906	0.9993	1.0000	0.9995	0.9993
21	0.9908	0.9995	1.0000	0.9998	0.9995
22	0.9909	0.9997	1.0000	0.9998	0.9997
23	0.9910	0.9998	1.0000	0.9999	0.9998
24	0.9911	0.9998	1.0000	0.9998	0.9998
25	0.9912	0.9999	1.0000	1.0000	0.9999

Table 5: Expected values and target score s

s	Expected Score	Simulation (10^2 trials)	Simulation (10^4 trials)	Simulation (10^6 trials)
1	5.5556	5.3900	5.5945	5.5550
2	5.5556	5.5800	5.5710	5.5601
3	5.5556	5.9900	5.5844	5.5565
4	5.5556	5.0300	5.6235	5.5534
5	5.6759	5.6600	5.6183	5.6744
6	5.8997	5.7800	5.9575	5.8969
7	6.2099	6.2600	6.2798	6.2057
8	6.5895	6.4000	6.5199	6.5774
9	7.0240	6.9000	6.9919	7.0172
10	7.3444	7.3900	7.3176	7.3496
11	7.5720	6.4100	7.5625	7.5806
12	7.7278	7.1200	7.8010	7.7259
13	7.8313	8.1800	7.6484	7.8384
14	7.8950	9.2500	7.8838	7.9101
15	7.9627	6.2800	8.0251	7.9578
16	8.0238	6.6300	8.1572	8.0332
17	8.0694	6.9600	7.9102	8.0724
18	8.0936	9.7400	7.9866	8.0917
19	8.0970	7.3700	8.2503	8.1042
20	8.0834	8.1800	8.1439	8.0914
21	8.0569	7.4100	8.0864	8.0584
22	8.0203	9.0500	8.0030	8.0125
23	7.9724	8.6200	7.9523	7.9738
24	7.9119	6.6800	7.8740	7.9233
25	7.8380	5.5700	7.6071	7.8610
26	7.7518	8.9500	7.7774	7.7522
27	7.6558	3.5400	7.5644	7.6544
28	7.5539	5.9900	7.5610	7.5817
29	7.4489	8.3800	7.4108	7.4553
30	7.3416	7.7900	7.4775	7.3469
31	7.2312	7.3300	7.2535	7.2025
32	7.1164	7.5200	6.9861	7.0916
33	6.9962	7.2000	6.8685	7.0064
34	6.8707	7.1300	6.7370	6.8364
35	6.7410	8.2300	6.7247	6.7071
36	6.6088	7.2200	6.7246	6.5905
37	6.4757	6.0000	6.5170	6.4993
38	6.3427	7.7300	6.1962	6.3456
39	6.2101	5.1300	6.0985	6.2192
40	6.0775	5.5000	5.9137	6.0725
41	5.9444	6.1700	5.9315	5.9131
42	5.8107	5.5500	5.6465	5.8295
43	5.6765	5.5700	5.3207	5.6493
44	5.5422	3.3100	5.5954	5.5134
45	5.4087	5.2000	5.4222	5.4314
46	5.2764	4.4000	5.5131	5.2764
47	5.1457	7.3800	5.2730	5.1156
48	5.0166	4.6000	5.0883	5.0133
49	4.8890	7.4000	4.8690	4.8619
50	4.7629	6.3800	4.7629	4.7619

Table 6: Turns to win game and target score s

s	Estimated Turns	Simulation (10e3 trials)	Simulation (10e5 trials)
1	18.0000	23.5730	23.8183
2	18.0000	23.9670	23.8274
3	18.0000	24.0610	23.8457
4	18.0000	23.8530	23.8446
5	17.6183	23.1520	23.3645
6	16.9500	21.8290	22.4920
7	16.1034	21.1990	21.4852
8	15.1756	20.2400	20.3858
9	14.2369	19.3700	19.2222
10	13.6158	17.9680	18.4860
11	13.2065	17.2630	18.0108
12	12.9403	17.5070	17.6859
13	12.7692	17.6960	17.4451
14	12.6662	17.4440	17.3648
15	12.5586	16.6600	17.2512
16	12.4628	16.7740	17.2204
17	12.3925	17.2030	17.1211
18	12.3555	16.7320	17.0628
19	12.3503	16.3070	17.1513
20	12.3711	17.3900	17.1850
21	12.4117	17.7110	17.3651
22	12.4684	17.9940	17.3675
23	12.5432	17.5550	17.4693
24	12.6391	17.4780	17.5294
25	12.7583	17.3870	17.6879
26	12.9003	18.3630	17.8837
27	13.0621	18.1890	18.2705
28	13.2381	18.9600	18.6534
29	13.4248	19.3950	19.0177
30	13.6210	18.9630	19.3752
31	13.8290	19.3300	19.6473
32	14.0521	20.2070	19.7861
33	14.2936	20.2440	19.8327
34	14.5546	19.6570	19.9470
35	14.8347	20.5190	20.1071
36	15.1314	20.5050	20.5598
37	15.4423	20.9040	20.8369
38	15.7660	21.6920	21.3418
39	16.1028	21.6270	21.7681
40	16.4541	22.3930	22.3150
41	16.8224	23.6770	23.0895
42	17.2096	23.7510	23.7551
43	17.6166	25.6600	24.6204
44	18.0432	25.5410	25.4599
45	18.4887	26.3350	26.5315
46	18.9522	28.1000	27.2338
47	19.4337	28.7880	27.9802
48	19.9338	29.8610	28.8576
49	20.4539	27.6750	29.4369
50	20.9954	28.9030	29.6294

Table 8: Expected score values with target rolls r for biased dice

r	Unbiased Die	Sim. Average	Lower-biased Die	Sim. Average	Center-biased Die	Sim. Average	Upper-biased Die	Sim. Average
1	5.5556	5.5610	3.4014	3.3989	6.2639	6.2622	8.1633	8.1600
2	7.7161	7.7237	3.4708	3.4702	10.5268	10.5396	14.8086	14.8171
3	8.0376	8.0575	2.6562	2.6593	13.2682	13.2778	20.1478	20.1615
4	7.4422	7.4372	1.8069	1.8137	14.8653	14.8553	24.3662	24.3906
5	6.4602	6.4531	1.1524	1.1703	15.6137	15.6168	27.6261	27.6018
6	5.3835	5.3910	0.7055	0.7087	15.7438	15.7306	30.0692	30.0271
7	4.3616	4.3320	0.4200	0.4171	15.4340	15.4142	31.8193	31.8046
8	3.4616	3.4454	0.2449	0.2364	14.8216	14.7871	32.9840	33.0282
9	2.7044	2.7031	0.1406	0.1400	14.0110	14.0290	33.6572	33.6421
10	2.0867	2.0873	0.0797	0.0832	13.0813	13.0549	33.9201	34.0132
11	1.5940	1.5987	0.0447	0.0433	12.0911	12.1071	33.8431	33.8605
12	1.2076	1.2056	0.0249	0.0218	11.0835	11.0913	33.4873	33.4420
13	0.9085	0.9135	0.0138	0.0147	10.0893	10.0824	32.9052	32.9792
14	0.6794	0.6835	0.0076	0.0074	9.1300	9.1333	32.1418	32.0963
15	0.5055	0.5032	0.0041	0.0053	8.2197	8.2643	31.2360	31.3272
16	0.3745	0.3742	0.0022	0.0016	7.3673	7.3601	30.2207	30.1746
17	0.2763	0.2796	0.0012	0.0011	6.5775	6.5672	29.1243	29.1041
18	0.2032	0.2064	0.0007	0.0006	5.8520	5.8769	27.9705	28.1244
19	0.1489	0.1463	0.0004	0.0000	5.1905	5.1971	26.7795	26.7563
20	0.1089	0.1088	0.0002	0.0000	4.5910	4.5906	25.5682	25.5022
21	0.0794	0.0780	0.0001	0.0004	4.0506	4.0538	24.3507	24.3520
22	0.0577	0.0603	0.0001	0.0000	3.5657	3.5606	23.1385	23.1497
23	0.0419	0.0433	0.0000	0.0000	3.1324	3.1538	21.9413	21.9274
24	0.0304	0.0331	0.0000	0.0000	2.7465	2.7361	20.7667	20.6633
25	0.0220	0.0231	0.0000	0.0000	2.4040	2.4187	19.6208	19.6287

Table 9: Expected score values with target score r for biased dice

s	Unbiased Die	Sim. Average	Lower-biased Die	Sim. Average	Center-biased Die	Sim. Average	Upper-biased Die	Sim. Average
1	5.5556	5.5550	3.4014	3.3985	6.2639	6.2647	8.1633	8.1587
2	5.5556	5.5601	3.4014	3.3972	6.2639	6.2631	8.1633	8.1629
3	5.5556	5.5565	3.4014	3.4000	6.2639	6.2630	8.1633	8.1601
4	5.5556	5.5534	3.4014	3.4015	6.2639	6.2642	8.1633	8.1603
5	5.6759	5.6744	3.4831	3.4850	6.4201	6.4190	8.2339	8.2342
6	5.8997	5.8969	3.5695	3.5677	6.8756	6.8753	8.4434	8.4449
7	6.2099	6.2057	3.6178	3.6235	7.6493	7.6491	8.8746	8.8779
8	6.5895	6.5774	3.6150	3.6154	8.5784	8.5797	9.6241	9.6251
9	7.0240	7.0172	3.5723	3.5737	9.4479	9.4525	10.8024	10.8023
10	7.3444	7.3496	3.5163	3.5242	10.0735	10.0828	12.0687	12.0687
11	7.5720	7.5806	3.4524	3.4626	10.4678	10.4625	13.2788	13.2742
12	7.7278	7.7259	3.3743	3.3817	10.7478	10.7407	14.2853	14.3042
13	7.8313	7.8384	3.2742	3.2742	11.0433	11.0435	14.9418	14.9493
14	7.8950	7.9101	3.1519	3.1505	11.4148	11.4188	15.1024	15.1102
15	7.9627	7.9578	3.0147	3.0248	11.8532	11.8539	15.3794	15.3772
16	8.0238	8.0332	2.8751	2.8823	12.2992	12.3015	15.8046	15.8069
17	8.0694	8.0724	2.7409	2.7369	12.7007	12.7009	16.3896	16.3814
18	8.0936	8.0917	2.6132	2.6004	13.0340	13.0297	17.1050	17.1112
19	8.0970	8.1042	2.4883	2.4784	13.3070	13.3018	17.8914	17.8780
20	8.0834	8.0914	2.3629	2.3553	13.5457	13.5249	18.6708	18.6836
21	8.0569	8.0584	2.2361	2.2349	13.7751	13.7794	19.3664	19.3620
22	8.0203	8.0125	2.1096	2.1078	14.0066	13.9856	19.9308	19.9309
23	7.9724	7.9738	1.9862	1.9790	14.2368	14.2428	20.3831	20.3883
24	7.9119	7.9233	1.8680	1.8621	14.4547	14.4578	20.7701	20.7984
25	7.8380	7.8610	1.7560	1.7531	14.6507	14.6576	21.1509	21.1455
26	7.7518	7.7522	1.6495	1.6432	14.8216	14.8411	21.5769	21.5902
27	7.6558	7.6544	1.5477	1.5448	14.9702	14.9511	22.0703	22.0500
28	7.5539	7.5817	1.4501	1.4598	15.1022	15.1024	22.6018	22.5989
29	7.4489	7.4553	1.3566	1.3445	15.2222	15.2081	23.1361	23.1216
30	7.3416	7.3469	1.2675	1.2785	15.3322	15.3287	23.6412	23.6325
31	7.2312	7.2025	1.1832	1.1769	15.4315	15.4434	24.0977	24.0949
32	7.1164	7.0916	1.1038	1.1117	15.5187	15.4949	24.5026	24.5106
33	6.9962	7.0064	1.0290	1.0372	15.5930	15.5818	24.8681	24.8550
34	6.8707	6.8364	0.9587	0.9568	15.6544	15.6769	25.2151	25.2320
35	6.7410	6.7071	0.8924	0.8859	15.7041	15.7002	25.5638	25.5374
36	6.6088	6.5905	0.8301	0.8221	15.7432	15.7450	25.9252	25.9310
37	6.4757	6.4993	0.7715	0.7667	15.7724	15.7540	26.2989	26.3180
38	6.3427	6.3456	0.7166	0.7120	15.7923	15.8097	26.6758	26.6900
39	6.2101	6.2192	0.6652	0.6630	15.8031	15.7944	27.0427	27.0575
40	6.0775	6.0725	0.6172	0.6239	15.8050	15.7903	27.3889	27.3432
41	5.9444	5.9131	0.5723	0.5737	15.7984	15.8218	27.7104	27.7198
42	5.8107	5.8295	0.5305	0.5272	15.7838	15.8250	28.0093	28.0169
43	5.6765	5.6493	0.4914	0.4793	15.7616	15.7692	28.2924	28.2659
44	5.5422	5.5134	0.4550	0.4523	15.7322	15.7306	28.5670	28.5473
45	5.4087	5.4314	0.4211	0.4220	15.6958	15.7203	28.8388	28.8398
46	5.2764	5.2764	0.3896	0.3929	15.6527	15.6541	29.1098	29.1362
47	5.1457	5.1156	0.3602	0.3583	15.6031	15.6536	29.3782	29.3134
48	5.0166	5.0133	0.3330	0.3403	15.5475	15.5367	29.6402	29.6264
49	4.8890	4.8619	0.3076	0.3093	15.4862	15.5210	29.8917	29.8616
50	4.7629	4.7619	0.2841	0.2812	15.4197	15.4358	30.1301	30.0931

Python Simulation

tables.py

```
import pandas as pd
import numpy as np
import average, probability, sampling
import recursion
from tqdm.auto import tqdm, trange

def save_df(data: dict, name: str):
    df = pd.DataFrame(data)
    df.to_csv(f"{name}.csv", index=False)
    return df

def table_1() -> pd.DataFrame:
    print("Generating Table 1")
    data = {"x": [x for x in range(12 + 1)], "P(X1=x)": sampling.p}
    df = save_df(data, "Table_1")
    print(df)
    return df

def table_2(max_r: int = 25, sim_mag=[3, 4, 6]) -> pd.DataFrame:
    print("Generating Table 2")
    data = {
        "r": [r for r in range(1, max_r + 1)],
        "E(Xr)": [
            recursion.E(1) * r * (1 - sampling.p[0]) ** (r - 1)
            for r in range(1, max_r + 1)
        ],
    }

    for mag in tqdm(sim_mag, desc="Table 2"):
        sim = [average.avg_roll_tr(r, 10**mag) for r in range(1, max_r + 1)]
        data[f"Simulation (10e{mag})"] = sim

    df = save_df(data, "Table_2")
    print(df)
    return df

def table_3(max_r: int = 15) -> pd.DataFrame:
    print("Generating Table 3")
    data = {
        "r": [r for r in range(1, max_r + 1)],
        "Estimated Turns": [100 / recursion.E(r) for r in range(1, max_r + 1)],
        "Sim Avg (10e3)": [
            average.avg_turns_tr(tr, early_stop=True, trials=10**3)
            for tr in trange(1, max_r + 1)
        ],
        "Sim Avg (10e5)": [
            average.avg_turns_tr(tr, early_stop=True, trials=10**5)
            for tr in trange(1, max_r + 1)
        ],
    }

    df = save_df(data, "Table_3")
    print(df)
    return df

def table_4(max_r: int = 25, sim_mag=[2, 4, 6]) -> pd.DataFrame:
    print("Generating Table 4")
    data = {
        "r": [r for r in range(1, max_r + 1)],
```

```
        "Markov": [recursion.P(r, 0) for r in range(1, max_r + 1)],
        "Geometric": [1 - (1 - sampling.p[0]) ** r for r in range(1, max_r + 1)],
    }

    for mag in tqdm(sim_mag, desc="Table 4"):
        sim = [
            probability.p_losing_turn(r, trials=10**mag) for r in range(1, max_r + 1)
        ]
        data[f"Sim (10e{mag})"] = sim

    df = save_df(data, "Table_4")
    print(df)
    return df

def table_5(max_s: int = 50, sim_mag=[2, 4, 6]) -> pd.DataFrame:
    print("Generating Table 5")
    data = {
        "s": [s for s in range(1, max_s + 1)],
        "Expected Score": [recursion.Es(s) for s in range(1, max_s + 1)],
    }

    for mag in tqdm(sim_mag, desc="Table 5"):
        sim = [average.avg_roll_ts(ts, trials=10**mag) for ts in range(1, max_s + 1)]
        data[f"Sim Avg (10e{mag})"] = sim

    df = save_df(data, "Table_5")
    print(df)
    return df

def table_6(max_s: int = 50) -> pd.DataFrame:
    print("Generating Table 6")
    data = {
        "s": [s for s in range(1, max_s + 1)],
        "Estimated Turns": [100 / (recursion.Es(s)) for s in trange(1, max_s + 1)],
        "Sim Avg (10e3)": [
            average.avg_turns_ts(ts, trials=10**3) for ts in trange(1, max_s + 1)
        ],
        "Sim Avg (10e5)": [
            average.avg_turns_ts(ts, trials=10**5) for ts in trange(1, max_s + 1)
        ],
    }

    df = save_df(data, "Table_6")
    print(df)
    return df

def table_7():
    data = {
        "x": [x for x in range(1, 6 + 1)],
        "Unbiased Die": [1 / 6] * 6,
        "Lower-biased Die": [i / 21 for i in reversed(range(1, 6 + 1))],
        "Upper-biased Die": [i / 21 for i in range(1, 6 + 1)],
        "Center-biased Die": [i / 12 for i in (1, 2, 3, 3, 2, 1)],
    }

    df = save_df(data, "Table_7")
    print(df)
    return df

table_1()
table_2()
table_3()
```

```
table_4()

table_5()
table_6()
table_7()
```

recursion.py

```
import numpy as np
from functools import lru_cache
import sampling

p = sampling.p


@lru_cache(maxsize=10000)
def P(r: int, x: int) -> float:
    if r == 1:
        if x > 12: return 0
        else: return p[x]
    else:
        if x == 0:
            return P(r - 1, 0) + p[0] * sum(
                [P(r - 1, i) for i in range(4 * (r - 1), 12 * (r - 1))]
            )
        else:
            return sum(
                [p[x - i] * P(r - 1, i) for i in range(max(x - 12, 4), x - 4 + 1)]
            )

def E(r: int) -> float:
    return sum([i * P(r, i) for i in range(4 * r, 12 * r + 1)])

@lru_cache(maxsize=10000)
def Ps(r: int, x: int, s: int) -> float:
    if r == 1:
        if x > 12: return 0
        else: return p[x]
    else:
        if x == 0:
            return Ps(r - 1, 0, s) + p[0] * sum(
                [Ps(r - 1, i, s) for i in range(4 * (r - 1), 12 * (r - 1))]
            )
        if x < s:
            return sum(
                [Ps(r - 1, i, s) * p[x - i] for i in range(max(x - 12, 4), x - 4 + 1)]
            )
        else:
            return Ps(r - 1, x, s) + sum(
                [
                    p[x - i] * Ps(r - 1, i, s)
                    for i in range(max(x - 12, 4), min(x - 4, s - 1) + 1)
                ]
            )


def Es(s: int) -> float:
    r = np.ceil(s / 4)
    return sum([i * Ps(r, i, s) for i in range(s, s + 12 + 1)])
```

average.py

```
import numpy as np
from numba import njit, prange

import rolls, game

@njit(parallel=True)
def avg_roll_tr(target_rolls: int, trials: int = 100_000) -> float:
    running_total: int = 0
    for t in prange(trials):
        running_total += rolls.roll_turn_tr(target_rolls)
    return running_total / trials

@njit(parallel=True)
def avg_roll_ts(target_score: int, trials: int = 100_000) -> float:
    running_total: int = 0
    for t in prange(trials):
        running_total += rolls.roll_turn_ts(target_score)
    return running_total / trials


# Average number of rolls, target rolls
@njit(parallel=True)
def avg_turns_tr(target_rolls: int, early_stop: bool = True, trials: int = 100_000):
    running_total: int = 0
    for t in prange(trials):
        running_total += game.game_turns_tr(target_rolls, early_stop)
    return running_total / trials


@njit(parallel=True)
def avg_turns_ts(target_score: int, early_stop: bool = True, trials: int = 100_000):
    running_total: int = 0
    for t in prange(trials):
        running_total += game.game_turns_ts(target_score, early_stop)
    return running_total / trials
```

game.py

```
from numba import njit, prange
import numpy as np
import sampling

@njit
def game_turns_tr(target_rolls: int, early_stop: bool) -> int:
    turns: int = 0
    total: int = 0
    while total < 100:
        turn_total: int = 0
        for r in range(target_rolls):
            dice = sampling.roll_dice()
            if np.array_equal(dice, np.ones_like(dice)):
                turn_total = 0
                total = 0
                break
            elif np.any(dice[:] == 1):
                turn_total = 0
                break
            else:
```

```
                turn_total += dice.sum()
                if early_stop and turn_total >= 100 - total: break
        turns += 1
        total += turn_total
    return turns


@njit
def game_turns_ts(target_score: int, early_stop: bool) -> int:
    turns: int = 0
    total: int = 0
    while total < 100:
        turn_total: int = 0
        while turn_total < target_score:
            dice = sampling.roll_dice()
            if np.array_equal(dice, np.ones_like(dice)):
                turn_total = 0
                total = 0
                break
            elif np.any(dice[:] == 1):
                turn_total = 0
                break
            else:
                turn_total += dice.sum()
                if early_stop and turn_total >= 100 - total: break
        turns += 1
        total += turn_total
    return turns
```

rolls.py

```
import numpy as np
from numba import njit
import sampling

@njit
def roll_turn_tr(target_rolls: int) -> int:
    total: int = 0
    for r in range(target_rolls):
        dice = sampling.roll_dice()
        if np.any(dice[:] == 1): return 0
        total += dice.sum()
    return total

@njit
def roll_turn_ts(target_score: int) -> int:
    total: int = 0
    while total < target_score:
        dice = sampling.roll_dice()
        if np.any(dice[:] == 1): return 0
        total += dice.sum()
    return total
```

probability.py

```
import numpy as np
from numba import njit, prange
import rolls

@njit(parallel=True)
def p_losing_turn(target_rolls: int, trials: int = 100_000) -> float:
    total: int = 0
    for t in prange(trials):
        total += 0 if rolls.roll_turn_tr(target_rolls) > 0 else 1
    return total / trials
```

sampling.py

```
import numpy as np
from numba import njit

# Unbiased die
p_d1 = [1 / 6, 1 / 6, 1 / 6, 1 / 6, 1 / 6, 1 / 6]
p_d2 = [1 / 6, 1 / 6, 1 / 6, 1 / 6, 1 / 6, 1 / 6]

# Lower-biased die
#p_d1 = [i/21 for i in reversed(range(1,6+1))]
#p_d2 = [i/21 for i in reversed(range(1,6+1))]

# Center-biased die
#p_d1 = [i/12 for i in (1,2,3,3,2,1)]
#p_d2 = [i/12 for i in (1,2,3,3,2,1)]

# Upper-biased die
#p_d1 = [i / 21 for i in range(1, 6 + 1)]
#p_d2 = [i / 21 for i in range(1, 6 + 1)]


assert len(p_d1) == len(p_d2)

p_grid = np.array([[p_d1[i] * p_d2[j] for j in range(6)] for i in range(6)])
p = (
    [p_grid[0][0] + p_grid[:, 0][1:].sum() + p_grid[0, :][1:].sum()]
    + [0] * 3
    + [
        np.trace(np.flip(p_grid[1:, 1:], axis=1), offset=o)
        for o in reversed(range(-4, 4 + 1))
    ]
)

cd_d1 = np.cumsum([0] + p_d1) / np.cumsum([0] + p_d1)[-1]
cd_d2 = np.cumsum([0] + p_d2) / np.cumsum([0] + p_d2)[-1]

cd_d1 = np.cumsum([0] + p_d1) / np.cumsum([0] + p_d1)[-1]
cd_d2 = np.cumsum([0] + p_d2) / np.cumsum([0] + p_d2)[-1]

@njit
def roll_dice() -> np.array:
    u1, u2 = np.random.rand(1), np.random.rand(1)
    idx_1 = np.searchsorted(cd_d1, u1, side="right")
    idx_2 = np.searchsorted(cd_d1, u2, side="right")
    return np.concatenate((idx_1, idx_2))
```