The Labouchere System: A Numerical Approach

Math 231 - Final Paper

Alejandro Dobles

June 9, 2018

1 Abstract

Every gambler has at some point or another come across betting systems. Some gamblers swear by them, others despise them (usually because of some previous unfavorable experience with them). Regardless of the sentiment of the gambler community to betting systems, they prove to be very interesting mathematical objects. In this paper we shall explore the mathematical properties of one betting system in particular: the Labouchere system. We will tackle problems concerning the Labouchere system with a simulation based approach, as well as state and prove certain propositions analytically.

2 Introduction

For our purposes, it will suffice to consider betting systems for games with even-money payoffs. With this in mind, a betting system can be defined as follows:

Definition 1. (*Betting System*)

Consider a discrete time stochastic process of i.i.d. random variables $\{X_n\}_{n\geq 1}$ with $P(X_1 = 1) = p$ and $P(X_1 = -1) = q$. A *betting system* is a sequence of functions $b_n(X_1, \ldots, X_{n-1}) \geq 0$ which specifies how much a gambler should bet after the first $n-1$ coups, with with b_1 a constant. [1]

From this definition we can generate the sequence of bets $\{B_n\}_{n\geq 1}$ where $B_n = b_n(X_1, \ldots, X_n)$ and $B_1 = b_1$. When dealing with betting systems it is also desirable to keep track of the gambler's bankroll at each coup. This quantity is typically called a gambler's *fortune* and, after the n^{th} coup, is given by $F_n = F_0 + \sum_{i=1}^{n} B_j X_j$, where F_0 is the initial bankroll of the gambler. [1]

Now, we introduce the Labouchere betting system.

Definition 2. (*Labouchere Betting System*)

In the Labouchere betting system the gambler starts out with a list of numbers $\boldsymbol{l}^0 = (l_1, \ldots, l_k)$ which gets modified as the game unfolds. Let $\boldsymbol{l}^n$ be the list after the n^{th} coup and let $L_n = |\boldsymbol{l}^n|$ be the length of said list after the n^{th} coup. Then, the gambler bets as follows:

$$b_n(\boldsymbol{l}^{n-1}) = \begin{cases} 0 & \text{if } L_{n-1} = 0 \\ l_1 & \text{if } L_{n-1} = 1 \\ l_1 + l_{L_{n-1}} & \text{if } L_{n-1} \geq 2 \end{cases}$$

In words, the gambler bets the first plus the last terms of the list if the list has more than one element; if it has only 1 element she bets that amount; and if it is empty she stops playing. It is important to note that $\boldsymbol{l}^{n-1}$ is determined by $(X_1, \ldots, X_{n-1})$ so that b_n is really a function of $(X_1, \ldots, X_{n-1})$ in accordance with the definition of a betting system above. Right after a coup, the list is updated as follows: $\boldsymbol{l}^n = \boldsymbol{l}^{n-1}[X_n]$, where the square bracket notation is outlined below.

$$\boldsymbol{l}^n[1] = \begin{cases} \varnothing & \text{if } L_n \in \{0, 1, 2\} \\ (l_2, \ldots, l_{L_n - 1}) & \text{if } L_n \geq 3 \end{cases}$$

$$\boldsymbol{l}^n[-1] = \begin{cases} \varnothing & \text{if } L_n \in \{0, 1, 2\} \\ (l_1, \ldots, l_{L_n}, B_n) & \text{if } L_n \geq 3 \end{cases}$$

In words, after a win either the first and last terms are removed from the list, or the sole element of the list is removed; after a loss, the amount just bet is appended to the list.

3 Properties of the Labouchere System

Having explained how the Labouchere system works, we can now begin to explore its mathematical properties.

We can start by asking what does the fortune of a player look like when playing with the Labouchere system. Recall that the fortune of a player at the n^{th} coup is given by $F_n = F_0 + \sum_{j=1}^{n} B_j X_j$. This means that $F_n - F_{n-1} = B_n X_n$. Now let S_n be the sum of the terms in the list after the n^{th} coup, that is $S_n = \sum_i^n \boldsymbol{l}_i^n$. Thus $S_n - S_{n-1} = -B_n X_n$. This is because after a win, S_n decreases by the amount bet at the n^{th} coup; after a loss, S_n increases by the amount bet at the n^{th} coup. Therefore, $F_n - F_{n-1} = S_{n-1} - S_n$, which means that $F_n + S_n = F_{n-1} - S_{n-1}$. Inductively, we conclude that $F_n = F_0 + S_0 - S_n$ for all n. So, if the list empties at the n^{th} coup, thus ending the betting, we are guaranteed that $F_n = F_0 + S_0$. That is, if the betting stops the player is guaranteed to walk away with a profit equal to the sum of the terms of her initial list!

But this certainty isn't any good if probability of the list emptying is small. So, a natural question to ask is does the list empty with high probability? Let's pose this question in mathematical terms. Let $N = \min\{n : L_n = 0\}$, that is, N represents the first time our list becomes empty. We want to know the value of $P(N < \infty)$.

3.1 Distribution of playing times

Before we begin let's re-frame the problem in a way that will make it easier to think about. Notice that whenever the player wins a coup, the value of L_n goes down by 2. Conversely, when the player loses a coup the value of L_n increases by 1. The former happens with probability p and the latter with probability $q = 1 - p$. Thus we can think of L_n as an asymmetric random walk starting from L_0, where L_0 is the length of the initial list. Thinking of L_n like this we can write $L_n = L_0 + \sum_{i=1}^{n} X_i$, where the X_i are i.i.d with distribution given by $P(X = -2) = p$, $P(X = 1) = q$. It turns out we can simplify things a bit if we re-frame the problem a little further and consider an asymmetric random walk starting from 0, $Y_n = L_n - L_0$, and notice that $N = \min\{n : Y_n \leq -L_0\}$. Before we proceed, we need to lay some groundwork that will allow us to come up with a value for $P(N < \infty)$.

Lemma 1. Let $\{X_n\}$ be a Markov Chain on state space S. Let $A \subset S$ be such that A^C is finite. Let $M_A = \min\{n : X_n \in A\}$. Suppose that $P(M_A < \infty \mid X_0 = i) > 0$ for $i \in A^C$. Then, there exists a positive integer B and $\varepsilon \in (0, 1)$ such that:

$$P(M_A > nB \mid X_0 = i) < (1 - \varepsilon)^n, \quad n \geq 1$$

for every $i \in S$.

Proof. By assumption, we know that $P(M_A < \infty \mid X_0 = i) > 0$ for any $i \in A^C$, thus we can pick $0 < \varepsilon < \min_{i \in A^C}\{P(M_A < \infty \mid X_0 = i)\}$. This implies that there exists a positive integer B such that $P(M_A \leq B \mid X_0 = i) > \varepsilon$ for all $i \in A^C$. Using the Markov property we establish that:

$$P(M_A > 2B \mid X_0 = i) = P(X_1 \notin A, \ldots, X_2B \notin A \mid X_0 = i)$$

$$= \sum_{j \in A^C} P(X_1 \notin A, \ldots, X_{B-1} \notin A, X_B = j, X_{B+1} \notin A, \ldots, X_2B \notin A \mid X_0 = i)$$

$$= \sum_{j \in A^C} P(M_A > B, X_B = j \mid X_0 = i) P(M_A > B \mid X_0 = j)$$

$$\leq P(M_A > B \mid X_0 = i) \max_{j \in A^C} P(M_A > B \mid X_0 = j) < (1 - \varepsilon)^2$$

This is true for all $i \in A^C$. [1] □

Now, we prove the following theorem.

Theorem 1. Let $\{X_n\}$ be a Markov Chain on state space S, with transition matrix $\mathbf{P}$. Let A and B be two non-empty, disjoint subsets of S, such that $(A \cup B)^C$ is finite. Let $M_A = \min\{n : X_n \in A\}$ and $M_B = \min\{n : X_n \in A\}$. Assume that $P(M_A \wedge M_B < \infty \mid X_0 = i) > 0$ where $i \in (A \cup B)^C$.

Now, let $g : S \to [0, \infty)$ be such that $g = 1$ on A, $g = 0$ on B, and $g(i) = \sum_{j \in S} g(i)\mathbf{P}_{ij}$ for $i \in (A \cup B)^C$.

Then,

$$g(i) = P(M_A < M_B \mid X_0 = i)$$

Proof. Let $M = M_A \wedge M_B = \min\{n : X_n \in A \cup B\}$. Now assume $X_0 = i \in (A \cup B)^C$. Then $g(X_{n \wedge M})$ is a martingale adapted to $\{X_n\}$. This is true because

$$\mathbf{E}[g(X_{n+1}) \mid X_0 = i, X_1, \ldots, X_n] = \sum_{j \in S} g(j) P(X_{n+1} = j \mid X_0 = i, X_1, \ldots, X_n)$$

$$= \sum_{j \in S} g(j) \mathbf{P}_{X_n j} = g(X_n)$$

Note that the last equality holds only when $n < N$, hence why we stop X_n at N. Now, we apply lemma 1 on $(A \cup B)$, to get that $P(M < \infty \mid X_0 = i) = 1$ for every $i \in (A \cup B)^C$. In addition notice that g is bounded, since $(A \cup B)^C$ is finite. This allows us to apply the optional stopping theorem to get that:

$$g(i) = \mathbf{E}[g(X_0) \mid X_0 = i] = \mathbf{E}[g(X_M)] = 0 * P(X_M \in B \mid X_0 = i) + 1 * P(X_M \in A \mid X_0 = i)$$

$$g(i) = P(X_M \in A \mid X_0 = i) = P(M_A < M_B \mid X_0 = i)$$

[1]

□

Equipped with these results, let's try to solve a gambler's ruin problem, stopping our process $\{Y_n\}$ at $N_W = \min\{n : Y_n \leq L_0 \text{ or } Y_n \geq W\}$ for some $W > 0$. Suppose that there exists a function $u(i) : \{-L_0 - 1, \ldots, W\} \to \mathbb{R}$ such that:

$$u(i) = pu(i-2) + qu(i+1)$$

With boundary conditions $u(-L_0 - 1) = u(-L_0) = 0$, and $u(W) = 1$. Then we claim the following:

Claim 1. $u(0) = P(Y_{N_W} \geq W)$

Proof. The claim follows from theorem 1. This is because Y_{N_W} can be seen as a Markov chain with state space $\{-L_0 - 1, \ldots, W\}$ and absorbing states $\{L_0 - 1, -L_0, W\}$. In addition, the function $u(\cdot)$ meets the criteria of the function $g(\cdot)$ in theorem 1. Thus, theorem 1 tell us that $u(0) = P(M_A < M_B \mid Y_0 = 0)$ where $A = \{W\}$ $B = \{-L_0 - 1, L_0\}$, and M_A and M_B are stopping times as defined in the statement of theorem 1. Finally, this means $u(0) = P(M_A < M_B \mid Y_0 = 0) = P(Y_{N_W} \in A) = P(Y_{N_W} \geq W)$ □

As it turns out, letting $u(i)$ be of the form $u(i) = a_0 \lambda_0^i + a_1 \lambda_1^i + a_2 \lambda_2^i$, where the λ_k are the roots of the polynomial $P(\lambda) = q\lambda^3 - \lambda^2 + p$, ensures that $u(i) = pu(i-2) + qu(i+1)$.[1]

The polynomial $P(\lambda)$ has roots $\lambda_0 = 1, \lambda_1 = \frac{-p - \sqrt{p(4-3p)}}{2(p-1)}, \lambda_2 = \frac{-p + \sqrt{p(4-3p)}}{2(p-1)}$. Imposing the boundary conditions on $u(\cdot)$ we can solve for a_0, a_1, a_2. This, in turn, will

[1] We skip over a few steps here for space and time considerations. Since the main focus of this paper is numerical simulation we'll not get too involved in the algebra required to show these facts. For more information refer to Ethier(2010, p.254-255)

give us $P(Y_{N_W} \leq -L_0) = 1 - P(Y_{N_W}) \geq W) = 1 - u(0) = a_0 + a_1 + a_2$. Solving we get that:

$$P(Y_{N_W} \leq -L_0) = \frac{(\lambda_1^W - 1)\lambda_2^{W+L_0}(\lambda_2 - 1) - \lambda_1^{W+L_0}(\lambda_1 - 1)(\lambda_2^W - 1)}{(\lambda_1^{W+L_0} - 1)(\lambda_2^{W+L_0+1} - 1)(\lambda_2^{W+L_0} - 1)}$$

While the expression above may be a little difficult to parse, with a little extra work we can show that $\lim_{W\to\infty} P(Y_{N_W} \leq -L_0) = 1 \iff p \geq \frac{1}{3}$). As an example, in the case where $p = \frac{1}{2}$ we get that $\lambda_1 = \frac{1+\sqrt{5}}{2}$, and $\lambda_2 = \frac{1-\sqrt{5}}{2}$.[2] In this case $\lim_{W\to\infty} P(Y_{N_W} \leq -L_0) = 1$. Notice that this is equivalent to the statement that $P(N < \infty) = 1$.[3]

In fact, as Ethier shows in *Absorption Time Distribution for an Asymmetric Random Walk*[2], we can go further and derive a closed form expression for the distribution of N. In the case where we start with a list of length 1, for instance, we have that:

$$P(N = 3m + 1) = \frac{1}{2m+1}\binom{3m}{m}p^{m+1}q^{2m}$$

On the next page, figure 1 shows several graphs depicting the evolution of the length of a Labouchere list in 200 games, for three different win probabilities p. The list is assumed to have initial length 5. The simulations were capped at 1000 coups. As can be seen from the graphs most lists empty out reasonably quickly, even for games with low win probability, with only four simulated games reaching the cap of 1000 coups. An interesting thing to notice is that the first two graphs seem to have less simulated paths than the third. But the three graphs actually show the same amount of simulations each. The first two simply have many overlapping paths. This is because the higher values of p concentrate more of the probability on small values of N, larger values becoming increasingly improbable. This can be seen in the histograms of figure 6 (explained in the next paragraph). Furthermore, because in high p scenarios N is smaller, there are combinatorially fewer paths that attain these restricted values of N, and hence there's more overlap in these simulated paths.

In page 7, figure 2 shows histograms of simulated values of N for different winning probabilities p. A total of 10,000 games were simulated for each win probability, all starting with a list of length 5, and capped at 10,000 coups. This time no simulations produced a list that exceeded the cap. As can be seen from the histograms, as the win probability decreases the mean absorption time increases (predictably), as well as the standard deviation, and considerably so.

To see how these figures were created, and how the simulations were carried out, refer to appendix A, in particular to the methods *gen_ l, plot_ sp,* and *make_ hist.*

[2] Golden ratio!

[3] Again, this part could be more detailed here, but I've chosen to leave the details aside to move one with other characteristics of the Labouchere system, instead of spending several pages only on proving that the absorption time is finite. The truth is, initially I thought the proof was going to be simpler. Alas, it wasn't. But for more information, reference Ethier(2010, p.255, 283)

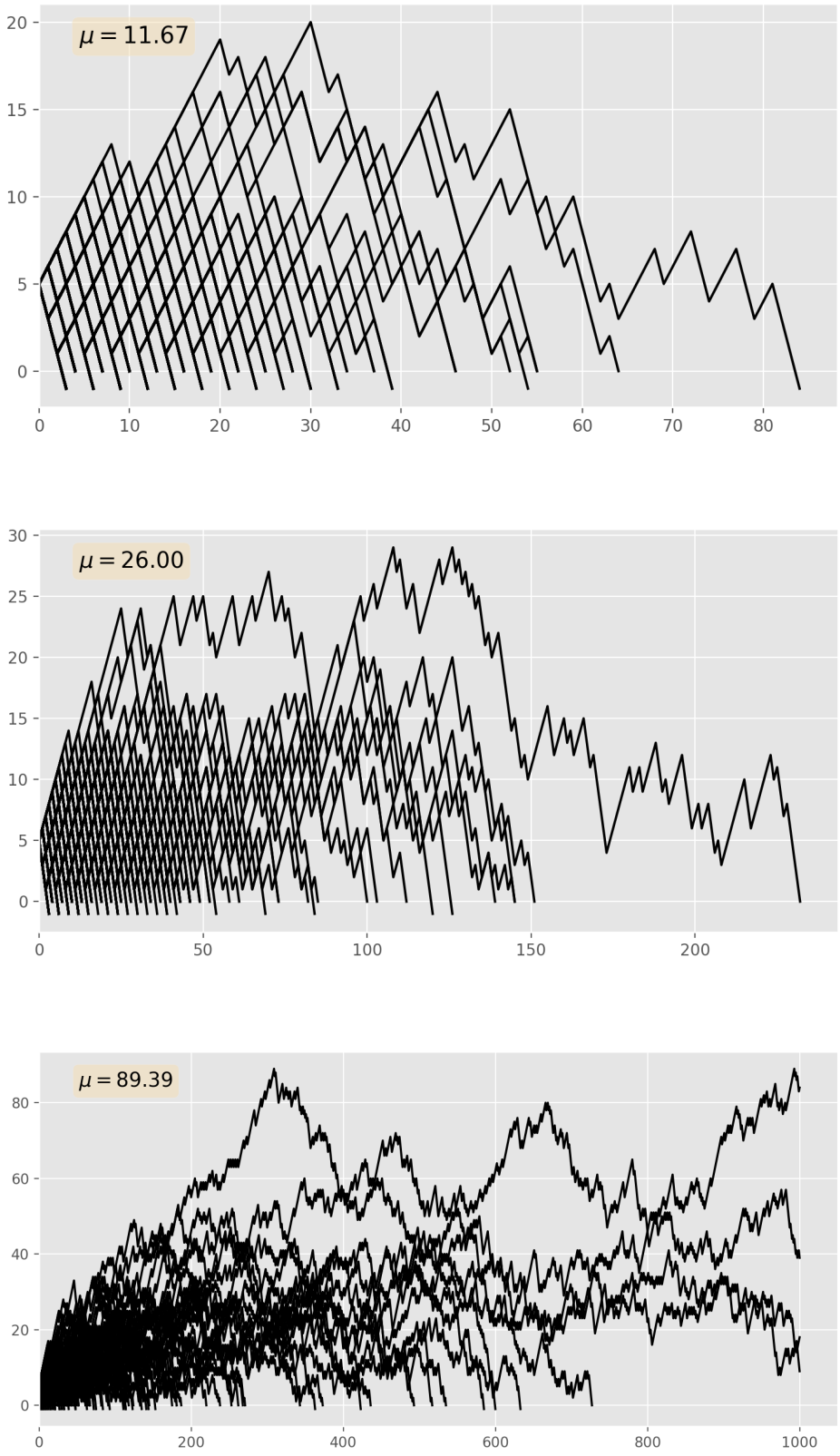

Figure 1: Length of a Labouchere list over time. From top to bottom $p = 0.5$, $p = 0.4$, $p = 0.35$. Number of simulations = 200.

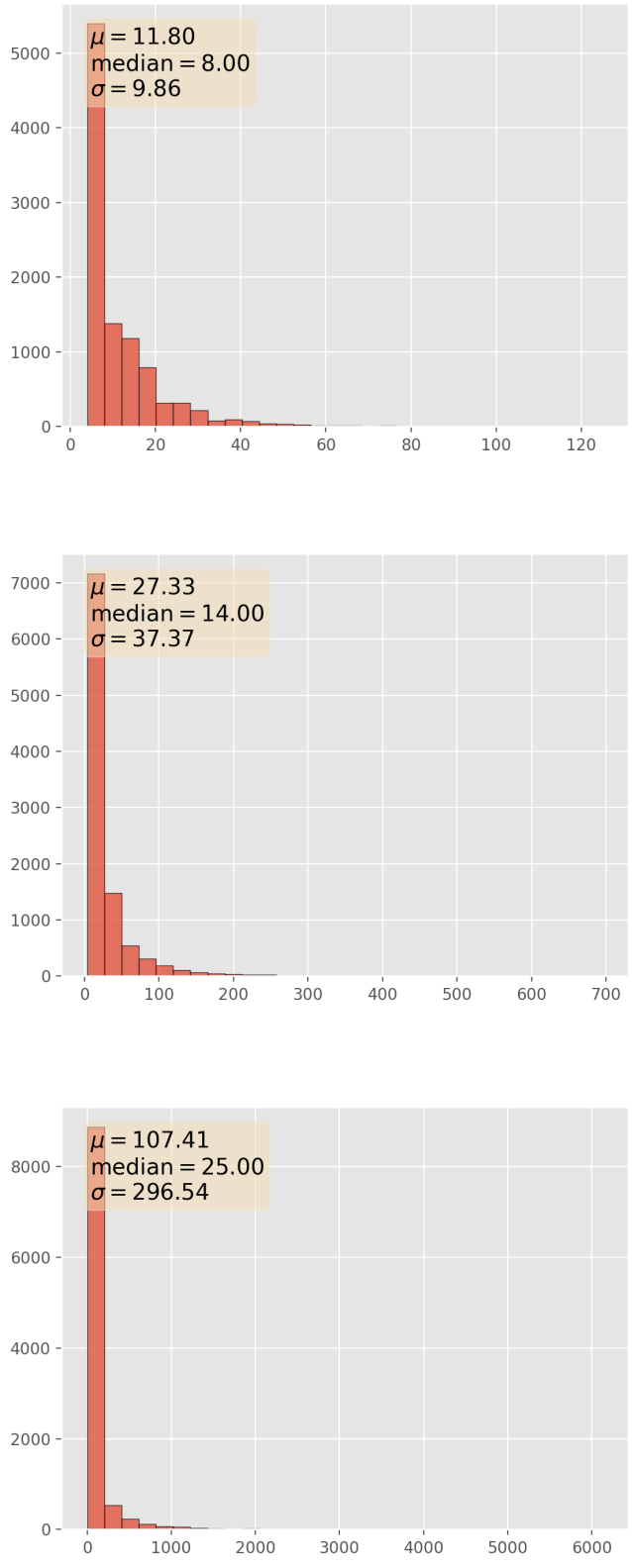

Figure 2: Distribution of absorption time N. From top to bottom $p = 0.5$, $p = 0.4$, $p = 0.35$. Number of simulations = 10,000.

3.2 Other interesting quantities

In addition to the absorption time of the length of the list in a Labouchere game, there are other values pertaining to the system that are worth studying. In this section we assume that a house betting limit of M units is imposed on the games, and explore three quantities, namely the probability of having to abort the system due to an over-the-limit bet; the expected number of coups played, given that the system is aborted when an over-the-limit bet is required; and the expected total amount bet, again with the constraint that no bets over M are allowed.

Let $Q(\boldsymbol{l}^0)$ be the probability that, starting with list $\boldsymbol{l}^0$ the list becomes empty before it reaches a state which demands an over-the-limit bet, that is, one where $b(\boldsymbol{l}^n) > M$. We can write a recursive expression for $Q(\cdot)$ as follows:

$$Q(\boldsymbol{l}) = pQ(\boldsymbol{l}[1]) + qQ(\boldsymbol{l}[-1])$$

The base cases of $Q(\cdot)$ being $Q(\varnothing) = 1$ and $Q(\boldsymbol{l}) = 0$ if $b(\boldsymbol{l}) > M$.

Along the same lines we can come up for a recursive expression for the following quantities: expected cumulative profit, expected number of coups, and expected total amount bet. Letting:

$$R(\boldsymbol{l}) = A(\boldsymbol{l}) + pR(\boldsymbol{l}[1]) + qR(\boldsymbol{l}[-1])$$

We get that $R(\boldsymbol{l}^0)$ is:

- The expected number of coups when $A(\boldsymbol{l}) = 1$
- The expected total amount bet when $A(\boldsymbol{l}) = b(\boldsymbol{l})$
- The expected cumulative profit when $A(\boldsymbol{l}) = b(\boldsymbol{l})(2p - 1)$

The bases cases of $R(\cdot)$ being $R(\varnothing) = R(\boldsymbol{l}) = 0$ if $b(\boldsymbol{l}) > M$.[1]

If one tries to evaluate $Q(\cdot)$ or $R(\cdot)$ by doing out the algebra, the problem quickly becomes onerous. Even for small values of M and nice values of p like $\frac{1}{2}$, it requires solving systems of several equations, and expanding recursive expressions countless times. For this reason, it is better to approach this problem with computational methods. In *The Doctrine of Chances*, Ethier calculates these quantities for a number of betting systems, but excludes the Labouchere system "because it is not amenable to exact calculations."[1] Here, we have come up with a recursive algorithm which is able to evaluate these values in a reasonable amount of time for M up to ~ 100.

The algorithm's exact implementation can be seen under the $\_Q$ and $\_R$ methods in Appendix A. Broadly, speaking it works by visualizing the problem as a binary tree and solving equations recursively by keeping track of previously seen but not yet computed nodes and their coefficients. Whenever the recursion ascends to a node

whose value appears in the tracked unknowns, it is solved like a standard linear equation, dividing through the coefficients of other unknowns.[4]

The algorithm was checked by comparing its results with the analytic formulas for the case $M = 5$ derived by Ethier[5]. They match exactly. Additionally, the algorithm was checked by using it to calculate values for the Fibonacci system. The values match exactly for multiple values of M with the values calculated by Ethier[6]. Therefore, it is with high confidence that the values for the Labouchere system in tables 1 through 4 are reported. They assume win probability $p = 244/495$ and initial list $\boldsymbol{l}^0 = (1)$. The values for the other systems were calculated by Ethier, under the same win probability.

Betting system	**M = 5**	**M = 25**	**M =100**
Martingale	6.66998 : 1	28.8303 : 1	115.017 : 1
Fibonacci	10.2918 : 1	47.0865 : 1	193.762 : 1
Oscar	13.3971 : 1	106.751 : 1	628.798 : 1
d'Alembert	8.32569 : 1	34.8477 : 1	65.5349 : 1
Blundell	76.6370 : 1	250.118 : 1	264.312 : 1
Labouchere	12.0723 : 1	64.6638 : 1	253.620 : 1

Table 1: Odds against having to abort the system

Betting system	**M = 5**	**M = 25**	**M =100**
Martingale	1.76419	1.96068	2.01120
Fibonacci	2.36811	3.00569	3.24728
Oscar	4.67097	6.20476	6.88647
d'Alembert	3.05655	14.4587	71.4244
Blundell	28.5345	79.5223	278.035
Labouchere	2.50542	3.15965	3.31142

Table 2: Expected number of coups played

Betting system	**M = 5**	**M = 25**	**M =100**
Martingale	3.04262	5.14343	7.304067
Fibonacci	4.43481	8.69619	13.5203
Oscar	7.47751	21.2394	50.9640
d'Alembert	7.19848	137.118	2,877.57
Blundell	66.8215	844.759	13,389.4
Labouchere	5.11555	11.5623	18.1052

Table 3: Expected total amount bet

[4]I will refrain from explaining the algorithm fully for want of time, and also because I feel like a proper explanation requires a diagram. Moreover, reading the code is probably sufficient for understanding how it works. If anyone is really interested please contact me and I'll be happy to talk about it in person.

[5]See *The Doctrine of Chances*, p.285 - 286

[6]See *The Doctrine of Chances*, p.297

$\mathbf{M = 5}$	$\mathbf{M = 25}$	$\mathbf{M = 100}$
-0.0723411	-0.163507	-0.256033

Table 4: Expected cumulative profit for the Labouchere system. Other systems are excluded here because Ethier didn't calculate cumulative profit for these systems.

Interestingly, the computation time grows really quickly as we increase M. For example, when calculating $Q((1))$ with $M = 5$ the computation takes 96 microseconds, with $M = 25$ it takes 9 millisecond, and when $M = 100$ it takes roughly 12 minutes. Nevertheless, we could make certain adjustments to the code in order to make calculations for $M \sim 500$ feasible. For starters, we could use Cython, a python package, to compile python code as C code. This makes it run much faster with speedups of up to 100x. Moreover, we could optimize the b and new_l methods, and use memoization on them in order to further increase the speed. Finally, if all else fails, we could approximate values for high M (at least values of $Q(\cdot)$) by capping the recursive depth at some level n. Due to the compounding of the p and q coefficients in the recursive expressions for Q and R, values at deep recursive levels become very small. In fact they are bounded above by $\max(p, q)^n$ where n is the recursive depth. Since the value of $Q(\cdot)$ is bounded by 1, the contribution of terms at these recursive depths quickly decays to 0. Thus capping the recursive depth of the computation and assuming that $Q(\cdot) = 0$ or 1 at this depth can give lower and upper bounds on the actual value of $Q(\cdot)$ being computed.

4 Conclusion: an open problem

As we have seen throughout this paper, there is plenty of mathematical scaffolding underpinning betting systems like the Labouchere system. Moreover, some of it offers fertile ground for exploration via numerical methods. To conclude, we apply numerical methods once more in the hopes of making an educated guess about an open problem in the mathematics of gambling. Specifically, what is the expectation of the maximum bet size in the Labouchere system, $\mathbf{E}[B^*]$?

I conjecture that $\mathbf{E}[B^*] = \infty$ for $\frac{1}{3} \leq p \leq \frac{1}{2}$.

Using the *simulate_max_bet* method, outlined in Appendix A, we can simulate many runs of the Labouchere system, record the values of B^* and calculate the sample mean, median, standard deviation and maximum. The following table outlines the results of said method for different values of p, all with initial list (1). Each value was generated by running 10,000,000 simulations.

	p = 0.5	**p = 0.4**	**p = 0.35**
Sample mean	5.635	1.45×10^{16}	6×10^{98}
Sample median	2	2	2
Sample std. dev.	654.26	4.5×10^{19}	1.9×10^{102}
Sample max	1,177,820	1.4×10^{23}	6×10^{106}

Table 5: Sample statistics for maximum bet size under varying win probabilities

While table 5 is useful to gauge the effect of the winning probability on the mean of the distribution of maximum bets, it is perhaps not as convincing as figure 3 in showing why $\mathbf{E}[B^*]$ is likely infinite. Figure 3 shows a log plot of sample mean of maximum bets versus number of simulations for a Labouchere game with winning probability 0.4 and initial list (1). As we see, there seems to be no convergence, on the contrary, the sample mean is quite steadily diverging towards larger and larger values.

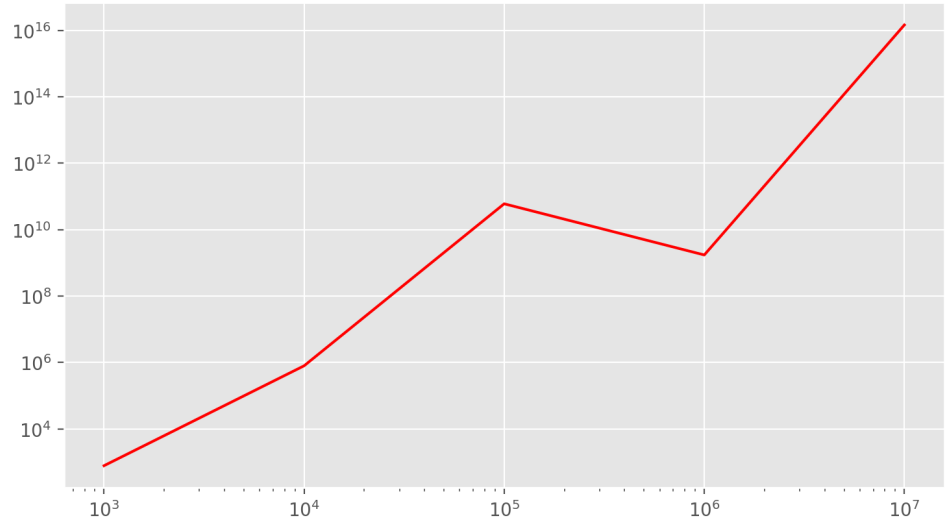

Figure 3: Log plot of sample mean B^* vs. number of simulations, for Labouchere games with winning probability 0.4 and initial list (1)

References

[1] Ethier, S. N. (2010). *The Doctrine of Chances: Probabilistic Aspects of Gambling.* New York: Springer.

[2] Ethier, S. N. *Absorption Time Distribution for an Asymmetric Random Walk.* Markov Processes and Related Topics: A Festschrift for Thomas G. Kurtz, 31–40, Institute of Mathematical Statistics, Beachwood, Ohio, USA, 2008.

5 Appendix A: Code

See next page.

```python
import numpy.random as random
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as stats
import time
import matplotlib

p = 0.4
q = 1 - p
a_config = 'profit' # 'profit': A(l) = b(l)(2p-1) -- 'total bet': A(l) = b(l) -- 'coups': A(l) = 1
M = 5


def b(l):
    """
    Returns the size of the next Labouchere bet given a current list l.

    :param l: The current list

    :return: The size of the next wager.
    """
    if not l:
        return 0
    elif len(l) == 1:
        return l[0]
    else:
        return l[0] + l[-1]


def b_fib(l):
    """
    Returns the size of the next Fibonacci bet given a current list l.

    :param l: The current list

    :return: The size of the next wager.
    """
    if not l:
        return 0
    elif len(l) == 1:
        return l[0]
    else:
        return l[-2] + l[-1]


def new_l(l, X, inplace=False):
    """
    Updates the list l according to the Labouchere betting system.

    :param l: current list
    :param X: outcome of last coup
    :param inplace: If true list is modified in-place, otherwise a new list is created and returned

    :return:  updated list or None if inplace=True
    """
    if not l:
        return l
    if inplace:
        if X == 1:
            l.pop()
            if l:
                del(l[0])
        else:
            l.append(b(l))
        return
    if X == 1:
        return l[1:-1]
    else:
        return l + [b(l)]


def new_l_fib(l, X, inplace=False):
    """
    Updates the list l according to the Fibonacci betting system.

    :param l: current list
    :param X: outcome of last coup
    :param inplace: If true list is modified in-place, otherwise a new list is created and returned
```

```
    :return:  updated list or None if inplace=True
    """
    if not l:
        return l
    if inplace:
        if X == 1:
            l.pop()
            if l:
                del(l[0])
        else:
            l.append(b_fib(l))
        return
    if X == 1:
        return l[:-2]
    else:
        return l + [b_fib(l)]


def A(l):
    """
    A function that depends on the constant a_config. To be used in computing R(l).
    When a_config = 'profit', R(l) will give the expected cumulative profit if gambler starts with list l.
    When a_config = 'total bet', R(l) will give the expected total amount bet if gambler starts with list l.
    When a_config = 'coups', R(l) will give the expected number of coups if gambler starts with list l.

    :param l: the current list

    :return: see above.
    """
    if a_config == 'profit':
        return b(l)*(2*p - 1)
    elif a_config == 'total bet':
        return b(l)
    return 1


def Q(l):
    """
    Wrapper function for recursive function _Q.
    Computes the probability that a Labouchere list becomes empty before a betting limit M is reached.
    :param l: starting list


    :return: the probability the the list becomes empty before a betting limit M is reached.
    """
    return _Q(l, set(), {})[0]

def _Q(l, seen, d):
    """
    Utility recursive function for computing the probability that a Labouchere list becomes empty before a betting limit
    M is reached.

    :param l: Current list.
    :param seen: A set keeping track of previously seen values of l.
    :param d: A memoization dictionary to keep track of previously computed values of Q(l).

    :return: A tuple of
    """
    if not l:
        return 1, {}
    if b(l) > M:
        return 0, {}
    key = str(l)
    if key in d:
        return d[key], {}
    if key in seen:
        return 0, {key : 1}
    seen.add(key)
    left_total, left_unks = _Q(new_l(l, -1), seen, d)
    right_total, right_unks = _Q(new_l(l, 1), seen, d)
    total = q*left_total + p*right_total
    c = q*left_unks.pop(key, 0) + p*right_unks.pop(key, 0)
    unks = {}
    for u in left_unks.keys() | right_unks.keys():
        unks[u] = (q*left_unks.get(u, 0) + p*right_unks.get(u, 0)) / (1 - c)
    total /= (1 - c)
    if not unks:
        d[key] = total
    return total, unks
```

```
def R(l):
    """
    Wrapper function for recursive function _R.
    Computes relevant values related to the Labouchere betting system. See return for a more detailed explanation.

    :param l: Initial list.

    :return: When a_config = 'profit', R(l) will give the expected cumulative profit if gambler starts with list l.
    When a_config = 'coups', R(l) will give the expected number of coups if gambler starts with list l.
    When a_config = 'total bet', R(l) will give the expected total amount bet if gambler starts with list l.
    """
    return _R(l, set(), {})[0]


def _R(l, seen, d):
    """
    Utitlity recursive function to compute relevant values related to the Labouchere system. See docstring for R().

    :param l: Current list
    :param seen: A set that keeps track of previously seen values of l
    :param d: A memoization dictionary to keep track of previously computed values of R(l)

    :return: A tuple of (total, unks). Total is the value of R(l) minus any unknowns seen during its computation.
    Unks is a dictionary mapping these unknown values of R(.) to their coefficients.
    """
    if not l or b(l) > M:
        return 0, {}
    key = str(l)
    if key in d:
        return d[key], {}
    if key in seen:
        return 0, {key : 1}
    seen.add(key)
    left_total, left_unks = _R(new_l(l, -1), seen, d)
    right_total, right_unks = _R(new_l(l, 1), seen, d)
    total = A(l) + q*left_total + p*right_total
    c = q*left_unks.pop(key, 0) + p*right_unks.pop(key, 0)
    unks = {}
    for u in left_unks.keys() | right_unks.keys():
        unks[u] = (q*left_unks.get(u, 0) + p*right_unks.get(u, 0)) / (1 - c)
    total /= (1 - c)
    if not unks:
        d[key] = total
    return total, unks


def simulate(m, l, f0):
    """
    Simulates a run of the Labouchere system at an even-money game, starting with list l and initial fortune f0.
    The simulation stops after n coups, or after the list is empty, or after the maximum bet M is exceeded.

    :param m: Specifies the maximum number of coups to play. If set to -1 then the simulation will be carried out
    indefinitely until either l becomes empty or the maximum bet M is exceeded
    :param l: specifies the initial list
    :param f0: specifies the initial fortune

    :return: A tuple of ({B_n}, {L_n}, {F_n}, l, n) where the first is a list of bets made, the second is a list with
    the length of the Labouchere list after each coup, the third is a list of the fortune of the gambler after each coup,
    the fourth is the final list, ie the one at termination, and n is the number of coups played.
    """
    B = []
    L = []
    F = [f0]
    n = 0
    while n < m or m < 0:
        bet = b(l)
        if bet > M:
            print('Maximum bet size exceeded.')
            return (B, L, F, l, n)
        if bet == 0:
            print('List emptied.')
            return (B, L, F, l, n)
        B.append(bet)
        if random.rand() < p:
            X = 1
        else: X = -1
        new_l(l, X, True)
```

```
        L.append(len(l))
        F.append(F[-1] + bet*X)
        n += 1
    print('Simulation completed.')
    return (B, L, F, l, n)

def gen_l(n, l0, absorbption):
    """
    Simulates an asymmetric random walk starting at l0 moving up 1 with prob q, and down 2 with prob p.

    :param n: number of steps in the stochastic process to simulate
    :param l0: initial value
    :param absorbption: stops generating if absorbing state hit
    :return:
    """
    l = [l0]
    l_i = l0
    for _ in range(n):
        if l_i <= absorbption:
            break
        if np.random.rand() < p:
            l_i = l_i - 2
            l.append(l_i)
        else:
            l_i = l_i + 1
            l.append(l_i)
    return l

def plot_sp(l):
    """
    Plots stochastic processes in time.

    :param l: A list of lists containing the values of a stochastic process
    :return: None
    """
    matplotlib.style.use('ggplot')
    fig, ax = plt.subplots()
    #cycle = plt.rcParams['axes.prop_cycle'].by_key()['color']
    for i in range(len(l)):
        ax.plot(l[i], color='k')#cycle[i % len(cycle)])
    _, x2, y1, y2 = plt.axis()
    plt.axis((0, x2, y1, y2))
    mean = np.mean([len(x) for x in l])
    props = dict(boxstyle='round', facecolor='wheat', alpha=0.5)
    txt = '$\mu=%.2f$' % (mean)
    ax.text(0.05, 0.95, txt, transform=ax.transAxes, fontsize=14,
            verticalalignment='top', bbox=props)
    plt.show()

def make_hist(l):
    matplotlib.style.use('ggplot')
    fig, ax = plt.subplots()
    ax.hist(l, bins=30, alpha=0.75, ec='black', histtype='bar')
    mean = np.mean(l)
    median = np.median(l)
    std = np.std(l)
    props = dict(boxstyle='round', facecolor='wheat', alpha=0.5)
    txt = '$\mu=%.2f$\n$\mathrm{median}=%.2f$\n$\sigma=%.2f$' % (mean, median, std)
    ax.text(0.05, 0.95, txt, transform=ax.transAxes, fontsize=14,
            verticalalignment='top', bbox=props)
    plt.show()


def simulate_max_bet(l, n):
    """
    Simulates n Labouchere games played with no betting limit, infinite credit, and initial list l.

    :param l: The initial list.

    :return: An n-element list where the ith element is the maximum bet size of the ith game simulated.
    """
    B_stars = []
    for i in range(1, n + 1):
        if i % 100000 == 0:
            print('{}th simulation completed.'.format(i))
        B_star = 1
        curr_l = l.copy()
        while curr_l:
            bet = b(curr_l)
            if bet > B_star:
                B_star = bet
```

```
            if random.rand() < p:
                X = 1
            else: X = -1
            new_l(curr_l, X, True)
        B_stars.append(B_star)
    print('Simulation completed.')
    return B_stars


def simulate_stopped_max_bets(l, n, st):
    """
    Simulates n runs of the stochastic process {B*_i} each element of which is generated as in simulate_max_bet, but
    this time without stopping until the criteria for a stopping time st is met.

    :param l: The initial list.
    :param n: The number of times the stochastic process {B*_i} will be generated till stopping.
    :param st: A function with boolean return value. Signature bool st(list B_stars).

    :return: -stopped_b_stars: a list of the B* at the stopped time.
    -stopping_times: the stopping time at which the ith run was stopped.
    """
    stopped_b_stars = []
    stopping_times = []
    for i in range(1, n+1):
        B_stars = []
        stopped = False
        if i % 1000 == 0:
            print('{}th simulation completed.'.format(i))
        while not stopped:
            B_star = 1
            curr_l = l.copy()
            while curr_l:
                bet = b(curr_l)
                if bet > B_star:
                    B_star = bet
                if random.rand() < p:
                    X = 1
                else:
                    X = -1
                new_l(curr_l, X, True)
            B_stars.append(B_star)
            stopped = st(B_stars)
        stopped_b_stars.append(B_star)
        stopping_times.append(len(B_stars))
    print('Simulation completed.')
    return stopped_b_stars, stopping_times


def run_simul_1():
    B, L, F, l, n = simulate(500, [1, 2, 3, 4], 50)
    print('Coups played = {}'.format(n))
    print('Final profit = {}'.format(F[-1] - F[0]))
    print('Final list: {}'.format(l))
    make_hist(B)
    make_hist(L)
    make_hist(F)


def run_simul_2(num_sims):
    B_stars = simulate_max_bet([1], num_sims)
    print('Maximum B* = {}'.format(max(B_stars)))
    print('Mean B* = {}'.format(np.mean(B_stars)))
    print('Median B* = {}'.format(np.median(B_stars)))
    print('Std dev B* = {}'.format(np.std(B_stars)))


C = 10000

def st(l):
    return l[-1] == 1 or l[-1] >= C


def run_simul_3(num_sims):
    stopped_b_stars, stopping_times = simulate_stopped_max_bets([1], num_sims, st)
    print('Maximum Stopped B* = {}'.format(max(stopped_b_stars)))
    print('Mean Stopped B* = {}'.format(np.mean(stopped_b_stars)))
    print('Proportion of B*=1: {}'.format(stopped_b_stars.count(1)/len(stopped_b_stars)))
    make_hist(stopped_b_stars)
    probs = [x/stopping_times.count(x) for x in set(stopping_times)]
    geom = [x/stats.geom.pmf(x, p) for x in range(min(stopping_times), max(stopping_times) + 1)]
    kl = stats.entropy(probs, geom)
```

```
    print('KL divergence between stopping times and geometric dist. = {}'.format(kl))
    make_hist(stopping_times)

def run_simul_4(num_sims, length, l0):
    l = [gen_l(length, l0, 0) for _ in range(num_sims)]
    plot_sp(l)
    make_hist([len(x) for x in l])


def test_Q_and_R():
    start_time = time.time()
    print(Q([1]))
    end_time = time.time()
    print("--- Completed in {:0.6f} seconds ---".format(end_time - start_time))
    start_time = time.time()
    print(R([1]))
    end_time = time.time()
    print("--- Completed in {:0.3f} seconds ---".format(end_time - start_time))


if __name__ == '__main__':
   run_simul_2(1000000)
```