Gambler's Ruin

May 11, 2020

A gambler G starts with two chips at a table game in a casino, pledging to quit once 8 more chips are won. G can either win a chip or lose a chip in each game, with probabilities p and $1-p$, respectively (independent of past game history). What is the probability that G accumulates a total of 10 chips playing the game repeatedly, without being ruined while trying? The ruining state is one where G has no chips and can no longer play. What is the probability that G is ruined while trying to get to 10 chips?

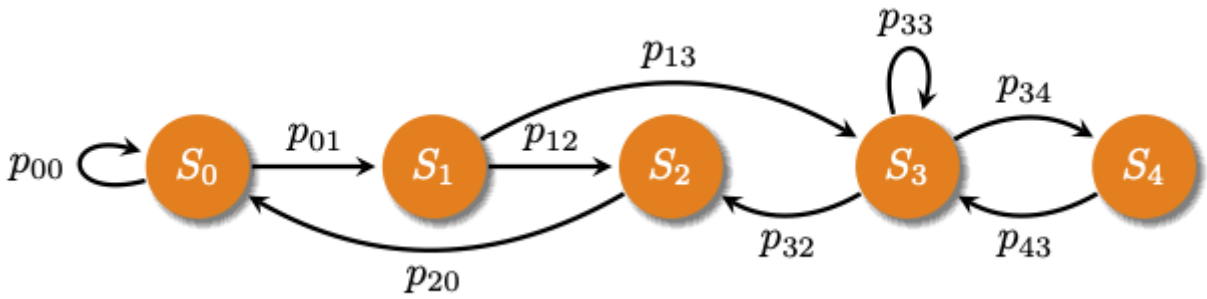
The goal of this activity is to introduce you to the rich subject of Markov chains, and in the process also get you acquainted with graphs, random walks, and stochastic matrices, with the gambler G as an entry point example. Since a probability course is not a prerequisite for this course, I will try to present the results colloquially and without proofs, with my advance apologies to the probabilists among you. The two theorems you will find stated below are proved using probabilistic tools (see, for example, [S] (https://www.springer.com/gp/book/9783540893318)) and is material one might usually find in a statistics program. In the next activity, I will connect this to material from other fields.

Markov chains

A Markov chain is an abstraction used to model systems that transition from a current state to the next state according to some given probability. It has proven itself to be a powerful construct in statistics due to its wide applicability. Specifically, given

- a set of *states* $\mathbb{S} = \{S_0, S_1, \ldots\}$,
- and a set of numbers $0 \leq p_{ij} \leq 1$,

a *Markov chain* is a sequence whose elements are taken from $\mathbb{S}$ in such a way that probability to go from state S_i to S_j is p_{ij}. The number p_{ij} is called the *transition probability.* The states and transition probabilities are often represented in diagrams like this:



The assumptions when considering a Markov chain are that the system is required to move from state to state (the next state can be the same as the current state), and that the next state is determined only by the current state's transition probabilities (not by prior states). The latter is called the *memorylessness property* or the Markov property. The former implies that the probability that the system will transition from the current state is one, so that for any i,

$$\sum_j p_{ij} = 1.$$

Note that the sum above may be finite or infinite: if the set of states is a finite set, say $\mathbb{S} = \{S_0, S_1, \ldots, S_N\}$, then the above sum runs from $j = 0, 1,$ through N; otherwise the sum should be treated as an infinite sum. Irrespective of the finiteness of the set of states $\mathbb{S}$, the Markov chain itself is thought of as an infinite sequence.

(*Optional note:* Here is a formal definition using the conditional probability notation, $\texttt{Pr}(A|B)$. A stochastic sequence X_n taking values from a set of states $\mathbb{S} = \{S_0, S_1, \ldots\}$ is called a Markov chain if for any subset of states $S_i, S_j, S_{k_0}, S_{k_1}, \ldots, S_{k_{n-1}} \in \mathbb{S}$,

$$\begin{aligned}&\texttt{Pr}(X_{n+1} = S_j | X_n = S_i)\\ &\quad = \texttt{Pr}(X_{n+1} = S_j | X_n = S_i, X_{n-1} = S_{k_{n-1}}, X_{n-2} = S_{k_{n-2}}, \ldots, X_0 = S_{k_0})\\ &\quad = p_{ij}.\end{aligned}$$

Throughout, we only consider what are known as *time-homogeneous* Markov chains, where the probabilities p_{ij} are independent of the "time" step n.)

To connect this to the concept of random walks, let us first introduce graphs.

Graphs

In mathematics, we often use the word graph in a sense completely different from the graph or plot of a function.

A *graph* (V, E) is a set V of n *vertices*, together with a set E of m *edges* between (some) vertices. Although vertices are often pictorially represented as points, technically they can be whatever things lumpable into a set V, e.g.,

- people, labels, companies, power stations, cities, etc.

Edges are often pictorially represented as line segments (or curves) connecting two points representing two vertices, but technically, they are just a "*choice of two vertices*" (not necessarily distinct) from V, e.g., corresponding to the above-listed vertex examples, an edge can represent

- friendship, similarities, spinoffs, wires, roads, etc.

When the above-mentioned "choice of two vertices" is changed to an ordered tuple, then the ordering of the two vertex choices that form an edge is significant, i.e., the edge has a direction. Thus a directed edge from vertex v_i to vertex v_j is the tuple (v_i, v_j). If all edges in E are directed, the graph is called a *directed graph* or a *digraph*. If a non-negative number, a *weight*, is associated to each edge of a digraph, then we call the graph a *weighted digraph*.

Python does not come with a graph data structure built in. Before you begin to think this somehow runs counter to the "batteries-included" philosophy of python, let me interrupt. Python's built-in dictionary data structure encapsulates a graph quite cleanly. Here is an example of a graph with vertices `a, b, c, d`:

In [1]:

```
gd = {'a': ['b', 'd'],           # a -> b,  a -> d
      'b': ['c', 'd', 'a'] }     # b -> c,  b -> d, b -> a
```

You can use a `dict` of `dict`s to incorporate more edge properties, such as assign names/labels, or more importantly, weights to obtain a weighted digraph.

In [2]:

```
gd = {'a': {'b': {'weight': 0.1},
            'd': {'weight': 0.8}},
      'b': {'d': {'weight': 0.5},
            'c': {'weight': 0.5}}
     }
```

Although we now have a graph data structure using the python dictionary, for this to be useful, we would have to implement various graph algorithms on it. Thankfully, there are many python packages that implement graph algorithms. Let's pick one package called NetworkX (http://networkx.github.io) as an example. Please install it before executing the next code cell. NetworkX allows us to send in the above dictionary to its digraph constructor.

In [3]:

```
import networkx as nx

g = nx.DiGraph(gd)    # dictionary to graph
```

Now `g` is a `DiGraph` object with many methods. To see all edges connected to vertex `a`, a dictionary-type access is provided. We can use it to double-check that the object is made as intended.

In [4]:

```
g['a']
```

Out[4]:

```
AtlasView({'b': {'weight': 0.1}, 'd': {'weight': 0.8}})
```

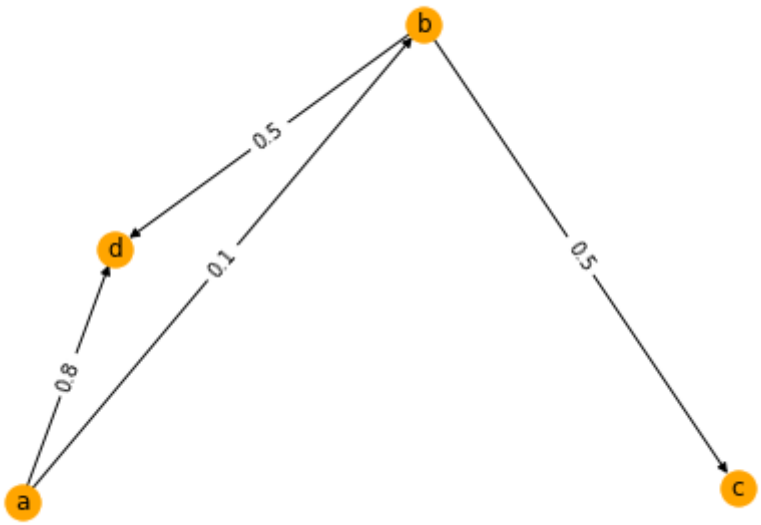
You can plot this graph using NetworkX's facilities (which uses matplotlib under the hood).

In [5]:

```
import matplotlib.pyplot as plt
%matplotlib inline

def plot_gph(g):
    pos = nx.spectral_layout(g)
    nx.draw(g, pos, with_labels=True, node_color='orange')
    labels = nx.get_edge_attributes(g, 'weight')
    nx.draw_networkx_edge_labels(g, pos, edge_labels=labels);

plot_gph(g)
```



Random walks

Consider a weighted digraph (V, E) where the weight associated to a directed edge $e = (v_i, v_j)$ is a number $0 < p_{ij} \leq 1$. Let us extend these numbers to every pair of vertices v_i and v_j such that $p_{ij} = 0$ if (v_i, v_j) is not an edge of the graph. Let us restrict ourselves to the scenario where

$$\sum_j p_{ij} = 1$$

for any i.

A *random walk* on such a directed graph is a sequence of vertices in V generated stochastically in the following sense. Suppose the nth element of the sequence is the ith vertex v_i in V. Then one of its outgoing edges (v_i, v_j) is selected with probability p_{ij}, and the $(n+1)$th element of the random walk sequence is set to v_j. This process is repeated to get the full random walk, once a starting vertex is selected.

Conceptual equivalences

There are three equivalent ways of viewing what is essentially the same concept:

- a probabilistic transition of states,
- a vertex-to-vertex probabilistic movement in digraphs, or
- a non-negative matrix of unit row sums.

Given a random walk on a weighted digraph, the sequence it generates is a Markov chain. Indeed, the digraph's edge weights give the transition probabilities. The graph vertices form the Markov chain states. Conversely, given a Markov chain, there is a corresponding random walk. We first generate a digraph using the Markov chain states as the graph vertices. Positive transition probabilities indicate which directed edges should exist in the graph and what their edge weight should be. The sequence of states of the Markov chain is now identifiable as the sequence of vertices generated by a random walk on this digraph. This equivalence is betrayed even by our very first figure above, where we illustrated a Markov chain using a graph.

To understand why the third concept is equivalent, it is sufficient to note that all information to specify either a Markov chain, or a random walk is encapsulated in a single mathematical object, namely the matrix P whose (i,j)th entry is p_{ij}. This matrix of probabilities is called a *transition matrix* (sometimes also called a *stochastic matrix*) and it can be associated either to a Markov chain or a random walk provided its rows sum to one.

Here is an example of a transition matrix.

In [6]:

```
import numpy as np
np.set_printoptions(suppress=True)

#              S0   S1   S2   S3
P = np.array([[0,   0.0, 0.5, 0.5],   # S0
              [1.0, 0.0, 0.0, 0.0],   # S1
              [0.0, 0.0, 0.0, 1.0],   # S2
              [0,   1.0, 0.0, 0.0]])  # S3

# Here S0, S1, S2, S3  are conceptual labels either
# for the Markov chain states or the digraph vertices
```

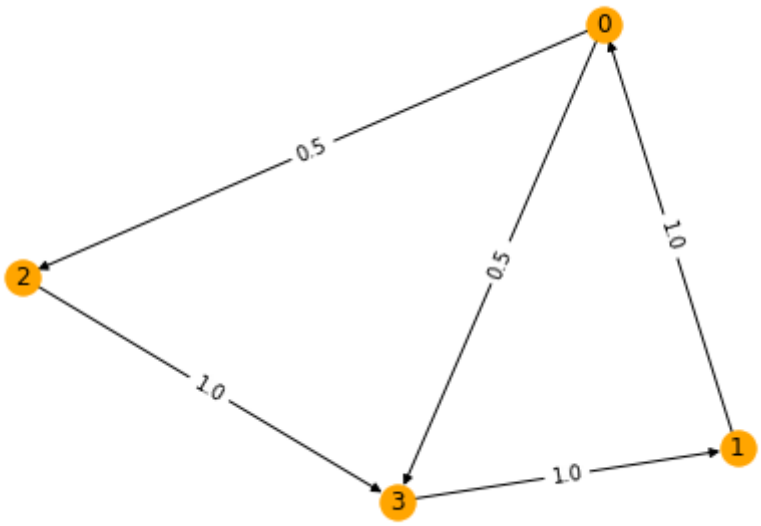
Matrix to digraph

The above-mentioned conceptual equivalences are often tacitly used in graph programs. For instance, in NetworkX, one can easily make a graph out of the above transition matrix `P` as follows.

In [7]:

```
gP = nx.from_numpy_array(P, create_using=nx.DiGraph)

plot_gph(gP)
```

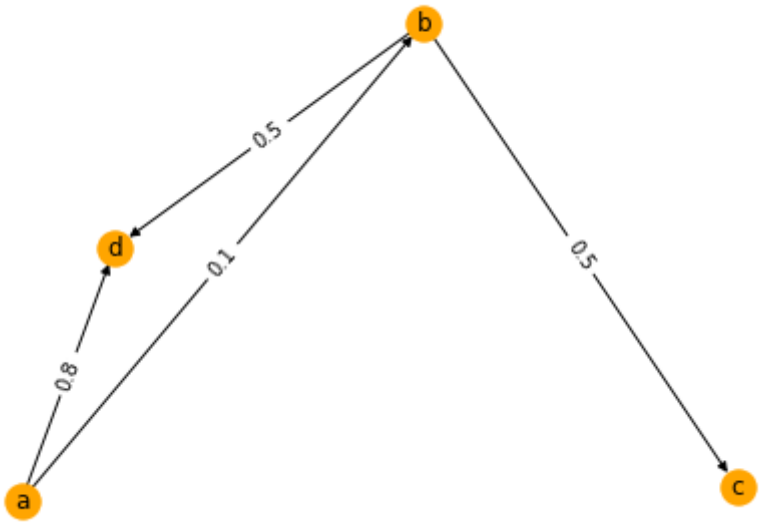


Digraph to matrix

We can, of course, also go the other way. For example, consider the small graph g we made "by hand" previously:

In [8]:

```
plot_gph(g)
```



In [9]:

```
g.nodes   # note the ordering of vertices
```

Out[9]:

```
NodeView(('a', 'b', 'd', 'c'))
```

This NetworkX object `g` can produce a matrix of the graph's edge weights. It is typical to make this as a `scipy.sparse` matrix since one anticipates many zero entries (corresponding to edges that do not exist in a realistic large graph). The result `Pg` below is a sparse matrix, which we convert to a dense matrix, just for printing.

In [10]:

```
Pg = nx.convert_matrix.to_scipy_sparse_matrix(g)
Pg.todense()
```

Out[10]:

```
matrix([[0. , 0.1, 0.8, 0. ],
        [0. , 0. , 0.5, 0.5],
        [0. , 0. , 0. , 0. ],
        [0. , 0. , 0. , 0. ]])
```

Note how the matrix entries and edge weights in the figure are in correspondence. The matrix is generally called the *adjacency matrix* of a weighted graph. (Note that many textbooks will define adjacency matrix entries with 1 in place of the nonzeros above to accommodate graphs without weights.) For digraphs in a random walk discussion, we shall refer to this adjacency matrix as the *transition matrix,* as previously noted.

The example of the gambler

Let us return to the gambler G with whom we made the acquaintance in the beginning of this activity. We formulate a Markov chain for G as follows.

Let S_i be the state of play where G has i chips. In the next step of the game, G can win the game and go to S_{i+1} with probability p, or lose and go to state S_{i-1} with probability $q = 1 - p$. The only possible states for G to be in are $S_0, S_1, \ldots, S_{10}$. The directed graph on which G is the random walker is as follows.

Here we have also indicated two additional pieces of information: G has pledged to quit upon reaching state S_{10}, so once the Markov chain reaches S_{10} it will not go to any other state forever. Furthermore, if G's Markov chain reaches the ruining state of S_0, then G can't play any more, so the Markov chain cannot go to any other state forever.

Let us look at the corresponding transition matrix, say when $p = 0.4$.

In [11]:

```
def PforG(p=0.4, N=10):
    q = 1 - p
    P = np.diag(q*np.ones(N), k=-1) + np.diag(p*np.ones(N), k=1)
    P[0, :] = 0
    P[0, 0] = 1
    P[N, :] = 0
    P[N, N] = 1
    return P

PforG(p=0.4)
```

Out[11]:

```
array([[1. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ],
       [0.6, 0. , 0.4, 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ],
       [0. , 0.6, 0. , 0.4, 0. , 0. , 0. , 0. , 0. , 0. , 0. ],
       [0. , 0. , 0.6, 0. , 0.4, 0. , 0. , 0. , 0. , 0. , 0. ],
       [0. , 0. , 0. , 0.6, 0. , 0.4, 0. , 0. , 0. , 0. , 0. ],
       [0. , 0. , 0. , 0. , 0.6, 0. , 0.4, 0. , 0. , 0. , 0. ],
       [0. , 0. , 0. , 0. , 0. , 0.6, 0. , 0.4, 0. , 0. , 0. ],
       [0. , 0. , 0. , 0. , 0. , 0. , 0.6, 0. , 0.4, 0. , 0. ],
       [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.6, 0. , 0.4, 0. ],
       [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.6, 0. , 0.4],
       [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 1. ]])
```

In the scenario described in the beginning, G starts with 2 chips and wants to reach a total of 10 chips. This translates to the following question: what is the probability that the above random walk enters the state S_{10}? It turns out that such questions are so often asked off Markov chains that the answer is a basic theoretical result on Markov chains (see the next theorem below).

Getting to a state

Let A and B be a partitioning of indices such that the subset of states

$$\mathbb{S}_A = \{S_i : i \in A\}, \qquad \mathbb{S}_B = \{S_i : i \in B\}$$

form a disjoint partitioning of the set $\mathbb{S}$ of all the states of a Markov chain. The probability that the Markov chain attains a state in $\mathbb{S}_A$ in some finite number of steps (i.e., if it ever hits $\mathbb{S}_A$) starting from a state S_i is called a *hitting probability* of $\mathbb{S}_A$ and is denoted by h_i. Of course, if $i \in A$, then $h_i = 1$. What is not obvious is the value of h_i when the chain starts from $i \in B$. Classic probabilistic arguments prove that the h_i values satisfy the system in the next theorem.

Theorem 1. The hitting probability of a subset A of states in a Markov chain is the minimal non-negative solution of the system of equations

$$h_i = \sum_j p_{ij} h_j$$

for all i with $i \in B$, and $h_i = 1$ if $i \in A$. Here minimality means that if x is another solution, then $x_i \geq h_i$ for all i.

The reasoning that leads to the system of equations in the theorem is as follows: if the Markov chain starts from state S_i, then in the next step, it can be in any of the states S_j with probability p_{ij}, from where the hitting probability is h_j, so the hitting probability h_i from S_i must be the sum of all $p_{ij} \times h_j$ over all states S_j. This idea can be formalized into a proof of Theorem 1. Let me highlight a few more things to note about Theorem 1:

- *First trivial solution:* From the definition of Markov chain, recall that
$$\sum_j p_{ij} = 1.$$
Hence an obvious solution to the system of equations in Theorem 1 is
$$h_i = 1$$
for all i. However, this solution need not be the *minimal* one mentioned in the theorem.

- *Second trivial solution:* One case where the minimal nonnegative solution is obvious is when A is such that $p_{ij} = 0$ for all $i \in B$ and $j \in A$, i.e., when it is not possible to go from B to A. Then
$$h_i = \begin{cases} 1, & i \in A, \\ 0, & i \in B, \end{cases}$$
obviously satisfies the system of equations in Theorem 1. Since the h_i values for $i \in B$ cannot be reduced any further, this is the minimal solution.

- Collecting h_i into a vector h, the system of equations in Theorem 1 can *almost* be written as the eigenvalue equation for eigenvalue 1,
$$h = Ph,$$
but *not* quite, because the equation need not hold for indices $i \in A$. Indeed, as stated in the theorem, h_i must equal 1 if $i \in A$.

Application to the gambler G

Setting $A = \{10\}$ and $B = \{0, 1, \ldots, 9\}$ in the above general theory, we see that G wins with probability h_2. Let us try to apply Theorem 1 to calculate h_2.

The approach I present here is not standard, but has the advantage that it uses eigenvector calculations that you are familiar with from your prerequisites. Even though the eigenvector remark I made at the end of the previous section is pessimistic, looking at G's transition matrix, we find that the condition $h_i = 1$ for $i \in A$ can be lumped together with the remaining equations. Indeed, because the last row of P contains only one nonzero entry (of 1),

$$h_{10} = \sum_j p_{10,j} h_j$$

holds. Therefore in G's case, h is a solution of the system in Theorem 1 if and only if h is a non-negative eigenvector of P corresponding to eigenvalue 1 and scaled to satisfy $h_{10} = 1$. (Be warned again that this may or may not happen for other Markov chains: see exercises.) What gives us further hope in the example of G is that we have a key piece of additional information:

$$h_0 = 0,$$

i.e., if G starts with no chips, then G cannot play, so G will stay in state S_0 forever. We might guess that this condition will help us filter out the irrelevant first trivial solution with $h_i = 1$ for all i.

Let me make one more remark before we start computing. The system of equations of Theorem 1 in the case of G reduces to

$$h_i = p h_{i+1} + (1-p) h_{i-1}$$

for $1 \leq i \leq 9$ together with $h_0 = 0$ and $h_{10} = 1$. You can make intuitive sense of this outside the general framework of the theorem. If G starts with i chips ($1 \leq i \leq 9$) so that the probability of hitting A is h_i, then in the next step there are two cases: (a) G has $i+1$ chips with probability p, or (b) G has $i-1$ chips with probability $1-p$. The probability of hitting A in case (a) is $p \times h_{i+1}$, and the probability of hitting A in case (b) is $q \times h_{i-1}$. Hence h_i must be equal to the sum of these two, thus explaining the theorem's equation $h_i = p h_{i+1} + (1-p) h_{i-1}$.

Let us now compute h_i using the knowledge that in G's case, h is a non-negative eigenvector of P corresponding to eigenvalue 1, scaled to satisfy $h_{10} = 1$.

In [12]:

```
from numpy.linalg import eig, inv, det

P = PforG(p=0.4)
ew, ev = eig(P)
ew
```

Out[12]:

```
array([-0.93184127, -0.79267153, -0.57590958, -0.30277358, -0.        ,
        0.93184127,  0.79267153,  0.30277358,  0.57590958,  1.        ,
        1.        ])
```

The computed set of eigenvalues of P include 1 *twice.* Since the diagonalization (the factorization produced by `eig`) was successful, we know that the eigenspace of P corresponding to eigenvalue 1 is two-dimensional. *If* there are vectors h in this eigenspace satisfying

$$h_0 = 0, \quad h_{10} = 1,$$

then such vectors clearly solve the system in Theorem 1. We can algorithmically look for eigenvectors satisfying these two conditions in a two-dimensional space: it's a system of two equations and two unknowns, made below in the form

$$Mc = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

In [13]:

```
H = ev[:, abs(ew - 1) < 1e-15]     # Eigenvectors of eigenvalue 1
M = np.array([H[0, :], H[-1, :]])  # Matrix of the two conditions
det(M)
```

Out[13]:

```
0.2825542003687932
```

The nonzero determinant implies that M is invertible. This means that there is a unique solution c and hence a *unique* vector in the eigenspace satisfying both the conditions, which can be immediately computed as follows.

In [14]:

```
def Gchances(p=0.4, N=10):
    P = PforG(p, N)
    ew, ev = eig(P)
    H = ev[:, abs(ew - 1) < 1e-15]
    M = np.array([H[0, :], H[-1, :]])
    c = inv(M) @ np.array([0, 1])
    return H @ c
```

In [15]:

```
h = Gchances(p=0.4)
h
```

Out[15]:

```
array([0.        , 0.00882378, 0.02205946, 0.04191297, 0.07169324,
       0.11636364, 0.18336924, 0.28387764, 0.43464024, 0.66078414,
       1.        ])
```

The significance of the above-mentioned uniqueness is that we no longer have to check if this `h` is the *minimal non-negative* solution of Theorem 1, since we have no more degrees of freedom to further reduce the above non-negative components.

We have just solved G's problem posed in the beginning.

The answer `h`, printed in the output above, tells us that *the probability of G accumulating 10 chips starting from 2 chips when $p = 0.4$ is* `h[2]`, whose value is approximately 0.022.

This is a lousy probability! Have we made a mistake? We began by assuming that the casino gives G *almost* a fair chance at winning each game, at a probability of $p = 0.4$, which is pretty close to the exactly fair chance of $p = 0.5$ (which we suspect no casino would give). Yet, the chance of G getting out with 10 chips is much less than p, per our computation. In fact, looking at which printed out entries of `h` that are above 0.5, we find that G has more than a 50% chance of making 10 chips only if G starts with 9 chips!

Cross checking

The answer we got above is correct, even if not intuitive. In fact, this is a manifestation of the phenomena that goes by the name of **Gambler's Ruin.** How can we double-check the above answer? One way to double-check the answer is the analytical technique described in the optional exercise below.

Optional exercise: Solve the equations of Theorem 1 in closed form to conclude that the probability of G making 10 chips, starting from i chips, is

$$h_i = \frac{1 - (q/p)^i}{1 - (q/p)^{10}}$$

whenever $p \neq q$.

I'll omit more details on this analytical way for verification, since this course is aimed at learning computational thinking. Instead, let's consider another computational way.

To let the computer cross check the answer, we design a completely different algorithm whose output "should" approximate the correct result. Namely, we simulate many many gambles, get the statistics of the outcomes, and then cross check the frequency of G's winnings. (That this "should" give the right probability is connected to the law of large numbers.)

Here is a simple way to implement many gambles (each gamble is a sequence of games until G reaches either S_0 or S_{10}) using the built-in random module that comes with python (and using the uniform distribution in $[0, 1]$).

In [16]:

```
from random import uniform

def gamble(init=2, p=0.4, win=10, n=10000):

    """Let G gamble "n" times, starting with "init" chips."""

    wl = np.zeros(n)   # mark win or lose here for each gamble i
    for i in range(n):
        chips = init
        while chips:
            if uniform(0, 1) > p:  # losing game
                chips -= 1
            else:                  # winning game
                chips += 1
            if chips == win:       # reached wanted winnings
                wl[i] = 1
                break
    return wl
```

In [17]:

```
n = 500000
wl = gamble(n=n)
print('Proportion of winning gambles:', np.count_nonzero(wl) / n)
```

```
Proportion of winning gambles: 0.022318
```

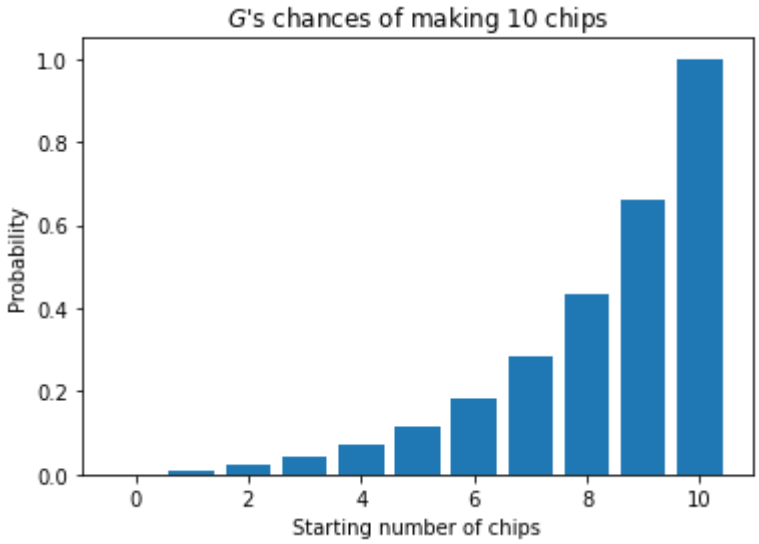
The number produced as the output is pretty close to the previously obtained `h[2]`. Indeed, it gets closer to `h[2]` with increasing number of gambles. Now that we have built more confidence in our answer computed using the eigensolver, let us proceed to examine all the components of `h` more closely.

Gambler's Ruin

Visualizing `h` in a bar plot, we find that G's computed chances of winning seem to decrease exponentially as the starting chip count decreases.

In [18]:

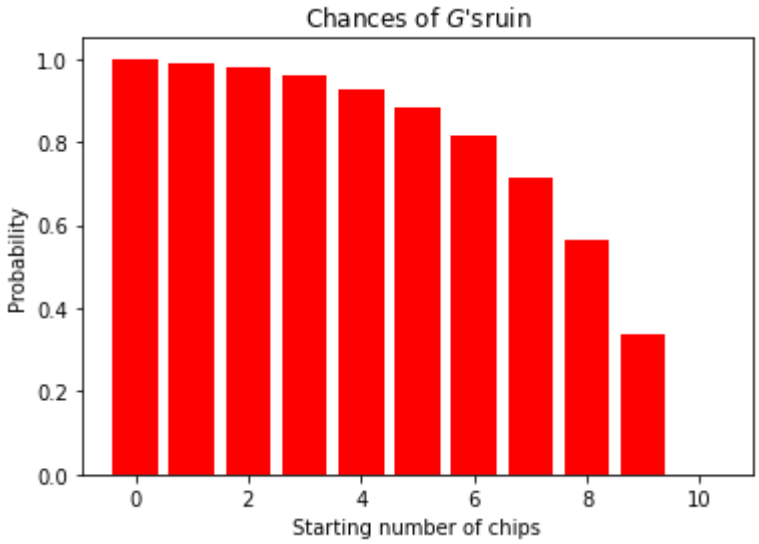
```
plt.bar(range(len(h)), h)
plt.title('$G$\'s chances of making 10 chips');
plt.xlabel('Starting number of chips'); plt.ylabel('Probability');
```



Since G either quits winning or gets ruined with 0 chips (not both), the probability of G's ruin is $1 - h_i$.

In [19]:

```
plt.bar(range(len(h)), 1-h, color='red')
plt.title('Chances of $G$\'sruin');
plt.xlabel('Starting number of chips'); plt.ylabel('Probability');
```

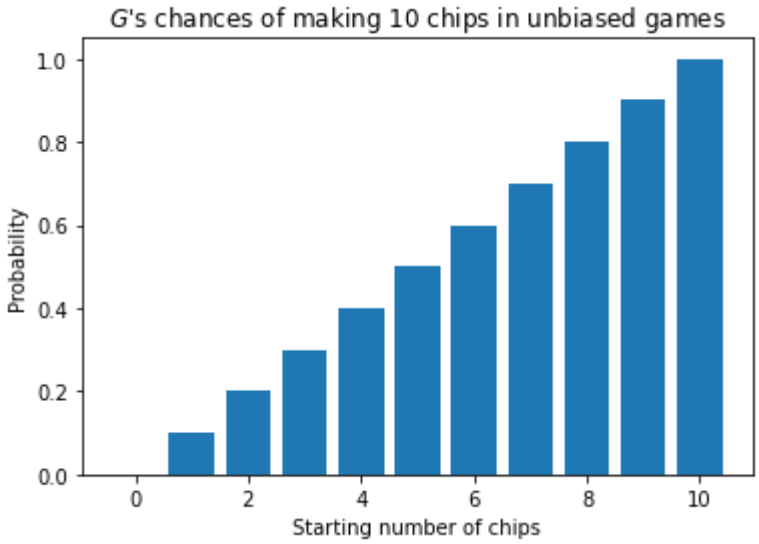


This exemplifies the concept of *Gambler's Ruin*: in a biased game (where $p < 1/2$), the probability of G's ruin could be much higher than the "intuitive" $1-p$ for most starting values.

Note that if all games are unbiased with $p = 1/2$, then we get the following linear plot, which perhaps jives with the intuition more than the unbiased case.

In [20]:

```
plt.bar(range(len(h)), Gchances(p=0.5, N=10))
plt.title('$G$\'s chances of making 10 chips in unbiased games');
plt.xlabel('Starting number of chips'); plt.ylabel('Probability');
```



Absorbing Markov chains

To round out our discussion of hitting probabilities, I should tell you that there is another, easier, way to algorithmically compute h_i in some circumstances.

In the example of G's Markov chain, we were able to extract the minimal non-negative solution of Theorem 1 uniquely from an eigenspace. Unique representations of solutions make for nice algorithmic prescriptions. There is a class of Markov chains for which we can always find certain hitting probabilities through a unique representation, and we don't even have to compute an eigenspace for it: we just need to solve a linear system. This is the subject of the next theorem (Theorem 2 below) also proved using basic probability methods.

A state S_i of a Markov chain is called an *absorbing state* if $p_{ii} = 1$. Clearly, once the chain reaches an absorbing state, it cannot transition to any other state forever.

An *absorbing Markov chain* is a Markov chain which has at least one absorbing state and has paths made of directed edges from any state to an absorbing state.

Partition the states in an absorbing Markov chain using index sets A and B, like before, except that now A denotes the indices for all the absorbing states (and B indicates the remaining states). Then the following partitioning of the transition matrix is both easy to conceptualize and easy to implement using numpy's slicing operations:

$$P = \begin{bmatrix} P_{AA} & P_{AB} \\ P_{BA} & P_{BB} \end{bmatrix}$$

Note that P_{AA} is an identity matrix and P_{AB} is the zero matrix, because $p_{ii} = 1$ for all $i \in A$.

Example: The gambler G has two absorbing states S_0 and S_{10}, and G's Markov chain is an absorbing Markov chain. Setting $A = \{0, 10\}$ and $B = \{1, \ldots, 9\}$, the blocks of the above partitioning for this case are as follows:

In [21]:

```
A = [0, 10]
B = range(1, 10)
P = PforG()
PAA = P[np.ix_(A, A)]
PBA = P[np.ix_(B, A)]
PBB = P[np.ix_(B, B)]
```

In [22]:

```
PBA
```

Out[22]:

```
array([[0.6, 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0.4]])
```

In [23]:

```
PBB
```

Out[23]:

```
array([[0. , 0.4, 0. , 0. , 0. , 0. , 0. , 0. , 0. ],
       [0.6, 0. , 0.4, 0. , 0. , 0. , 0. , 0. , 0. ],
       [0. , 0.6, 0. , 0.4, 0. , 0. , 0. , 0. , 0. ],
       [0. , 0. , 0.6, 0. , 0.4, 0. , 0. , 0. , 0. ],
       [0. , 0. , 0. , 0.6, 0. , 0.4, 0. , 0. , 0. ],
       [0. , 0. , 0. , 0. , 0.6, 0. , 0.4, 0. , 0. ],
       [0. , 0. , 0. , 0. , 0. , 0.6, 0. , 0.4, 0. ],
       [0. , 0. , 0. , 0. , 0. , 0. , 0.6, 0. , 0.4],
       [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.6, 0. ]])
```

In [24]:

```
PAA
```

Out[24]:

```
array([[1., 0.],
       [0., 1.]])
```

As already noted above, P_{AA} should always be the identity matrix in an absorbing Markov chain per our definition.

Now we are ready to state the second result on hitting probabilities, this time with an easier algorithmic prescription for computing them.

Theorem 2. In any finite absorbing Markov chain, the matrix $I - P_{BB}$ is invertible (where I denotes the identity matrix of the same size as P_{BB}). Moreover, letting H denote the matrix whose (i, j)th entry equals the probability that the chain hits an absorbing state S_j, $j \in A$, starting from another state S_i, $i \in B$, in any number of steps, we may compute H by

$$H = (I - P_{BB})^{-1} P_{BA}.$$

Example: In one line of code, we can apply Theorem 2 to gambler G using the matrices made just before the theorem:

In [25]:

```
np.linalg.inv(np.eye(len(B)) - PBB) @ PBA
```

Out[25]:

```
array([[0.99117622, 0.00882378],
       [0.97794054, 0.02205946],
       [0.95808703, 0.04191297],
       [0.92830676, 0.07169324],
       [0.88363636, 0.11636364],
       [0.81663076, 0.18336924],
       [0.71612236, 0.28387764],
       [0.56535976, 0.43464024],
       [0.33921586, 0.66078414]])
```

Since the *second* entry of A represents the winning state S_{10}, the *second* column above gives the probability of hitting S_{10} from various starting states. Note that that second column is the same as the previously computed `h`. The first column in the output above gives the probability of G's ruin for various starting states.

Greedy gambler

Let us conclude with another manifestation of the *Gambler's Ruin* concept that emerges when you ask the following question. What happens if G gets greedy and reneges on the pledge to quit upon reaching 10 chips? In other words, G continues to play infinitely many games unless ruined in between. What is the probability of G's ruin?

This is the same as considering $N = \infty$ case in our previous setting. This case results in an infinite set of states. Theorem 1 applies both to finite and infinite set of states, but we can only simulate Markov chains with finite number of states. Nonetheless, we can certainly apply Theorem 2 to compute the hitting probabilities for larger and larger N and get a feel for what might happen when $N = \infty$.

But first, we have to make our code better to go to large N. We use scipy's `sparse` facilities to remake the matrices and improve efficiency.

In [26]:

```
from scipy.sparse import diags, eye
from scipy.sparse.linalg import spsolve
```

In [27]:

```
def sparseGmats(p=0.4, N=10000):

    """ Return I - PBB and PBA as sparse matrices """

    q = 1 - p
    # Note that the first and last row of P are not accurate
    # in this construction, but we're going to trim them away:
    P = diags(q*np.ones(N), offsets=-1, shape=(N+1, N+1)) \
    +   diags(p*np.ones(N), offsets=1,  shape=(N+1, N+1))

    A = [0, N]
    B = range(1, N)
    I_PBB = (eye(N-1) - P[np.ix_(B, B)]).tocsc()
    PBA = P[np.ix_(B, A)].tocsc()

    return I_PBB, PBA

def ruinG(p=0.4, N=10000):

    """ Given that the winning probability of each game is "p",
    compute the probability of G's ruin for each starting state """

    I_PBB, PBA = sparseGmats(p, N)
    return spsolve(I_PBB, PBA[:, 0])
```

In [28]:

```
ruinG(N=10)
```

Out[28]:

```
array([0.99117622, 0.97794054, 0.95808703, 0.92830676, 0.88363636,
       0.81663076, 0.71612236, 0.56535976, 0.33921586])
```

After verifying that for the $N = 10$ case, we obtained the same result, we proceed to examine the higher values of N. One quick way to visualize the resulting `h`-values are as plots over starting states.

In [29]:

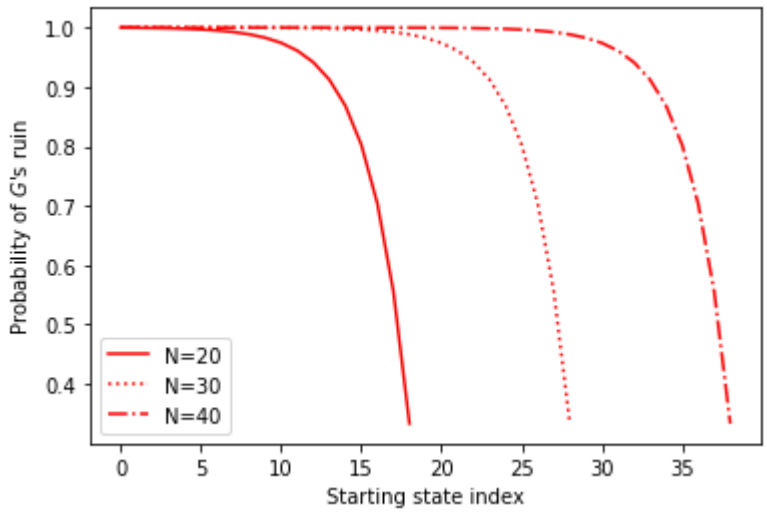
```
fig = plt.figure()
ax = plt.gca()

hs = ruinG(N=20)
ax.plot(hs[:21], 'r-', label='N=20')

hs = ruinG(N=30)
ax.plot(hs, 'r:', label='N=30')

hs = ruinG(N=40)
ax.plot(hs, 'r-.', label='N=40')

ax.set_ylabel('Probability of $G$\'s ruin')
ax.set_xlabel('Starting state index')
ax.legend();
```



Clearly, as N increases, we see a larger range of starting indices for which G hardly stands a chance of escaping ruin.

For a specific case, suppose G is unable to start with more than 20 chips, but is willing to play N games, for larger and larger N. Then we compute G's *least ruin probability*, the lowest probability of G's ruin among all possible starting values, namely

$$\min_{i=0,\ldots,20} h_i.$$

Let us examine how this minimal value changes with N.

In [30]:

```
def least_ruin_prob(p=0.4, N0=20, dbl=11):

    """ Compute least ruin probability starting with N="N0" and
    recompute "dbl" times, doubling N each time. """

    for i in range(dbl):
        print('N = %5d,  least ruin probability = %5.4f'
              %(N0*2**i,  min(ruinG(p=p, N=N0*2**i)[:21])))
```

In [31]:

```
least_ruin_prob(p=0.4, dbl=7)
```

```
N =    20,  least ruin probability = 0.3334
N =    40,  least ruin probability = 0.9995
N =    80,  least ruin probability = 1.0000
N =   160,  least ruin probability = 1.0000
N =   320,  least ruin probability = 1.0000
N =   640,  least ruin probability = 1.0000
N =  1280,  least ruin probability = 1.0000
```

Clearly, G is *ruined with certainty,* i.e., with probability 1, as $N \to \infty$.

What if the games are fair? The above results were with $p = 0.4$. Rerunning the code with the *fair* chance $p = 0.5$, we again observe convergence, albeit slower, to the inevitable.

In [32]:

```
least_ruin_prob(p=0.5, dbl=11)
```

```
N =    20,  least ruin probability = 0.0500
N =    40,  least ruin probability = 0.4750
N =    80,  least ruin probability = 0.7375
N =   160,  least ruin probability = 0.8687
N =   320,  least ruin probability = 0.9344
N =   640,  least ruin probability = 0.9672
N =  1280,  least ruin probability = 0.9836
N =  2560,  least ruin probability = 0.9918
N =  5120,  least ruin probability = 0.9959
N = 10240,  least ruin probability = 0.9979
N = 20480,  least ruin probability = 0.9990
```

What is illustrated in this output is often identified as another, perhaps stronger, manifestation of the *Gambler's Ruin* concept: *even when the games are fair,* G is certain to be ruined if G continues to play forever.

Author: Jay Gopalakrishnan (http://web.pdx.edu/~gjay/)

License: ©2020. CC-BY-SA (https://creativecommons.org/licenses/by-sa/4.0/legalcode)

≪Table of Contents (./TOC.html)