lab12sol

November 17, 2014

1 Virtual Lab 12 Solution: Hashing & Drunk Man

1.0.1 EECS 70: Discrete Mathematics and Probability Theory, Fall 2014

Due Date: Monday, November 24th, 2014 at 12pm **Name**: EECS 70
Login: cs70-ta
Instructions:

- **Please fill out your name and login above.**
- **Please leave your answers in the Markdown cells, marked with `"YOUR COMMENTS HERE"`.** If you don't see this cell at the end of a question, it simply means that question doesn't require an answer.
- **Complete this lab by filling in all of the required functions, marked with `"YOUR CODE HERE"`.**
- If you plan to use Python, make sure to go over **Tutorial 1: Introduction to Python and IPython** and **Tutorial 2: Plotting in Python with Matplotlib** before attempting the lab.
- Make sure you run every code cell one after another, i.e. don't skip any cell. A shortcut for doing this in the notebook is Shift+Enter. When you finish, choose 'Cell > Run All' to test your code one last time all at once.
- Most of the solution requires no more than a few lines each.
- Please do not hardcode the result or change any function without the `"YOUR CODE HERE"` mark.
- Questions? Bring them to our Office Hour and/or ask on Piazza.
- Good luck, and have fun!

1.1 Table of Contents

The number inside parentheses is the number of functions or code blocks you are required to fill out for each question. Always make sure to double check before you submit.


```
In [1]: %pylab inline
```

```
Populating the interactive namespace from numpy and matplotlib
```

```
In [2]: from __future__ import division  # so that you don't have to worry about float division
        import random
        import math
        import itertools
        from scipy.stats import bernoulli, binom

In [3]: colors = matplotlib.rcParams['axes.color_cycle']  # give different plots different colors
```

Introduction

The Birthday Paradox, which we explored in last week's lab, is a specific form of a more general concept known as hashing. In hashing, we are interested in minimizing the probability that two or more keys are hashed to the same table location. This also can be looked at as a balls and bins problem, where we are trying to minimize the probability of balls being thrown into the same bin. For the questions below, let's consider n bins and m balls (for the Birthday Paradox, $n = 365$).

For each part, students who want to can choose to completely rewrite the question. Basically, you can come up with your own formulation of how to do a series of experiments that result in the same discoveries. Then, write up the results nicely using plots as appropriate to show what you observed. You can also rewrite the entire lab to take a different path through as long as they convey the key insights aimed at in each part.

```
In [4]: def choose(n, k):
            """
            Computes n choose k
            """

            assert n >= 0 and k >= 0 and n >= k, "Incorrect parameters"
            return math.factorial(n)/(math.factorial(k)*math.factorial(n-k))
```

Part (a): Balls and Bins Collision

For $n = 100, 365, 500, 1000$, and for $m = 0.1n, 0.3n, \sqrt{n}$, simulate throwing m balls into n bins. For every (m, n) pair, do this for 1000 trials. For each value of n, plot the fraction of trials for which there is no collision vs. m. What do you observe as m gets larger?

Hint: You may find implementing the five functions below helpful for completing this lab. Their intended usage is given in the function's docstring.

```
In [5]: def throw_ball_into_bins(n):
            """
            Randomly throws a ball into n bins and returns
            the index of the bin from 1 to n.
            """

            index = int(math.ceil(random.random()*n))  # YOUR CODE HERE
            assert 1 <= index <= n, "Bin index is incorrect"
            return index

In [6]: def throw_many_balls_into_bins(m, n):
            """
            Randomly throws m balls into n bins and returns
            a list of indices

            YOUR CODE HERE
            """

            return [throw_ball_into_bins(n) for _ in xrange(m)]

In [7]: def check_collision(trial):
            """
```

```
    Checks if there is a collision in a trial (a list of indices)
    Returns True if there is a collision.
    This should be similar to the has_duplicate() function from last week

    YOUR CODE HERE
    """

    return len(trial) != len(set(trial))
```

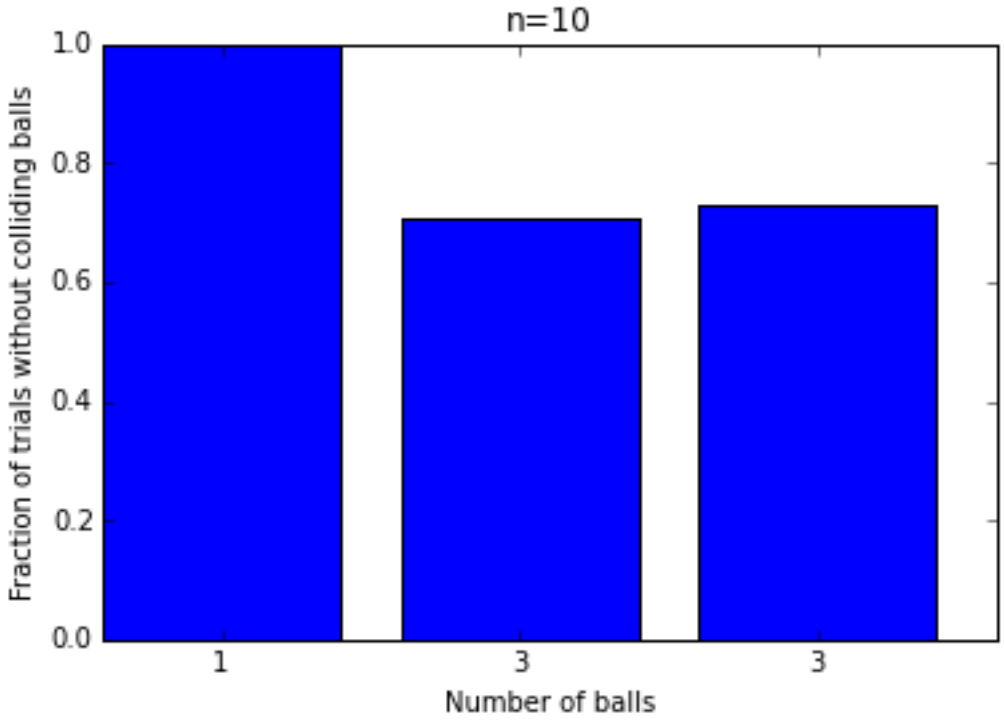
In [8]:
```
def run_many_ball_bin_trials (m, n, t):
    """
    Runs t trials of throwing m balls into n bins
    Returns a list of t trials

    YOUR CODE HERE
    """

    return [throw_many_balls_into_bins(m, n) for _ in xrange(t)]
```

In [9]:
```
def run_many_ball_bin_trials_check_collision (trials):
    """
    Returns a list of `check_collision` on each trial

    YOUR CODE HERE
    """

    return [check_collision(trial) for trial in trials]
```

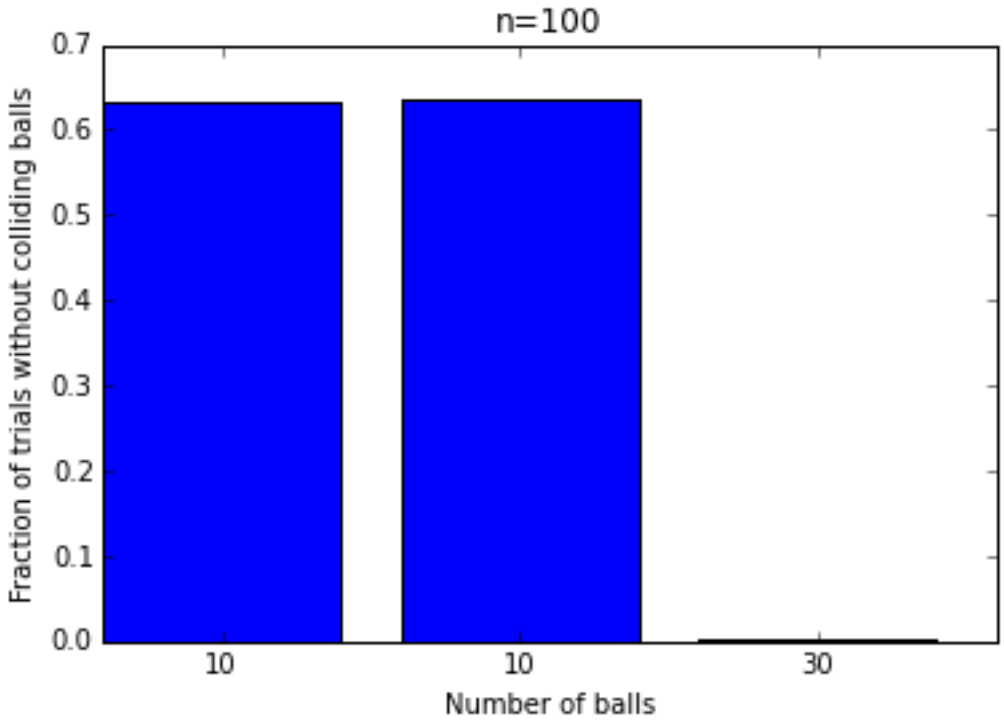
In [10]:
```
def partA(nrange=[10,100,365,1000], t=1000):
    """
    Part (a) code
    """

    for n in nrange:
        print 'n =', n
        mrange = sorted([int(0.1*n), int(0.3*n), int(math.sqrt(n))])
        ticks = range(len(mrange))  # for plotting

        ###################
        # YOUR CODE HERE
        trial_results = []
        for m in mrange:
            num_repeats = sum(run_many_ball_bin_trials_check_collision(run_many_ball_bin_trials
            repeat_frac = num_repeats / t
            print 'Fraction of trials without colliding balls when m = %i: %f' % (m, 1 - repeat
            trial_results.append(1 - repeat_frac)
        plt.bar(ticks, trial_results)
        ###################

        plt.xlabel('Number of balls')
        plt.xticks([tck + 0.4 for tck in ticks], mrange)  # Add 0.4 to make labels centered
        plt.ylabel('Fraction of trials without colliding balls')
        plt.title('n=%i' % n)
        plt.show()
```

In [11]:
```
partA()
```

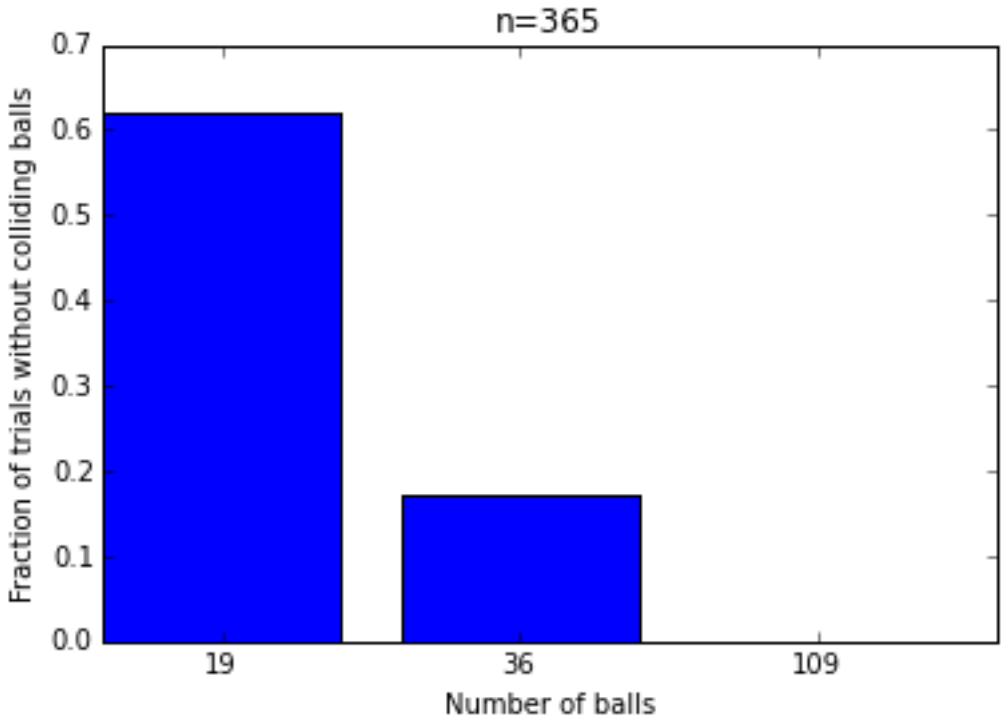
```
n = 10
Fraction of trials without colliding balls when m = 1: 1.000000
Fraction of trials without colliding balls when m = 3: 0.705000
Fraction of trials without colliding balls when m = 3: 0.731000
```



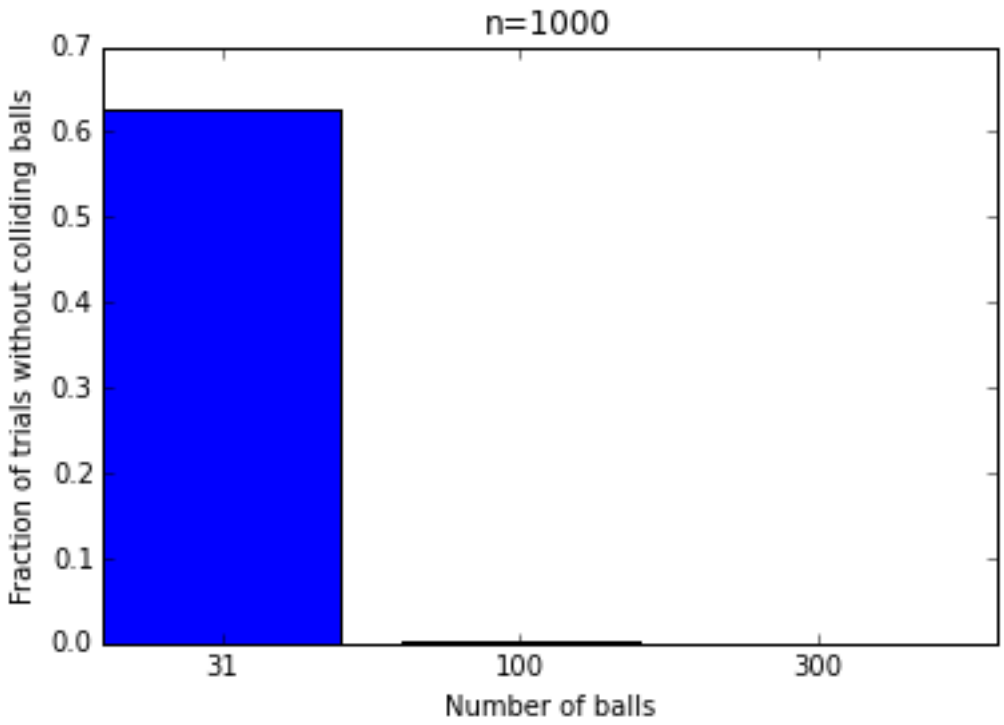

```
n = 100
Fraction of trials without colliding balls when m = 10: 0.632000
Fraction of trials without colliding balls when m = 10: 0.635000
Fraction of trials without colliding balls when m = 30: 0.004000
```




```
n = 365
Fraction of trials without colliding balls when m = 19: 0.619000
Fraction of trials without colliding balls when m = 36: 0.173000
Fraction of trials without colliding balls when m = 109: 0.000000
```




```
n = 1000
Fraction of trials without colliding balls when m = 31: 0.627000
Fraction of trials without colliding balls when m = 100: 0.004000
Fraction of trials without colliding balls when m = 300: 0.000000
```

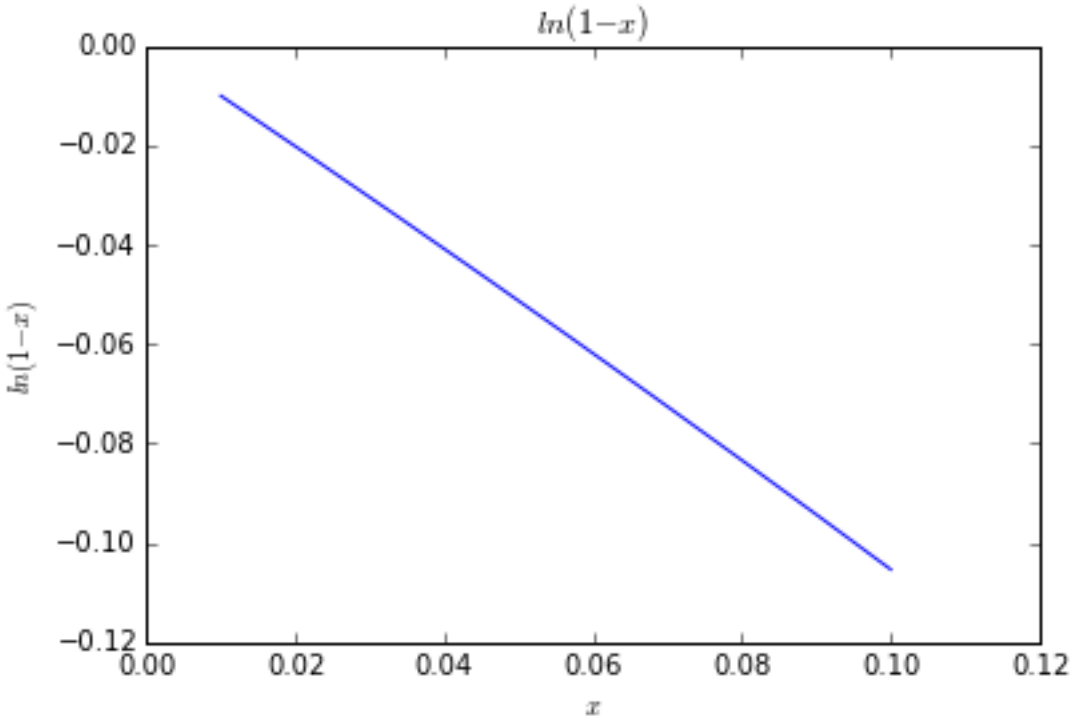


YOUR COMMENTS HERE:

Part (b): $ln(1-x)$

Let's take a detour and look at a very useful approximation for $ln(1-x)$. Plot $ln(1-x)$ for 1000 values of x from $0.01-0.1$. What is the approximate slope of this graph and why? (think about Taylor expansion.)

```
In [12]: def partB():
             """
             Part (b) code
             """

             xvals = np.linspace(0.01, 0.1, 1000)

             ###################
             # YOUR CODE HERE
             logvals = [math.log(1-x) for x in xvals]
             plt.plot(xvals, logvals)
             ###################

             plt.ylabel('$ln(1-x)$')
             plt.xlabel('$x$')
             plt.title('$ln(1-x)$')
             plt.show()

In [13]: partB()
```



YOUR COMMENTS HERE:

Part (c): Negative Log

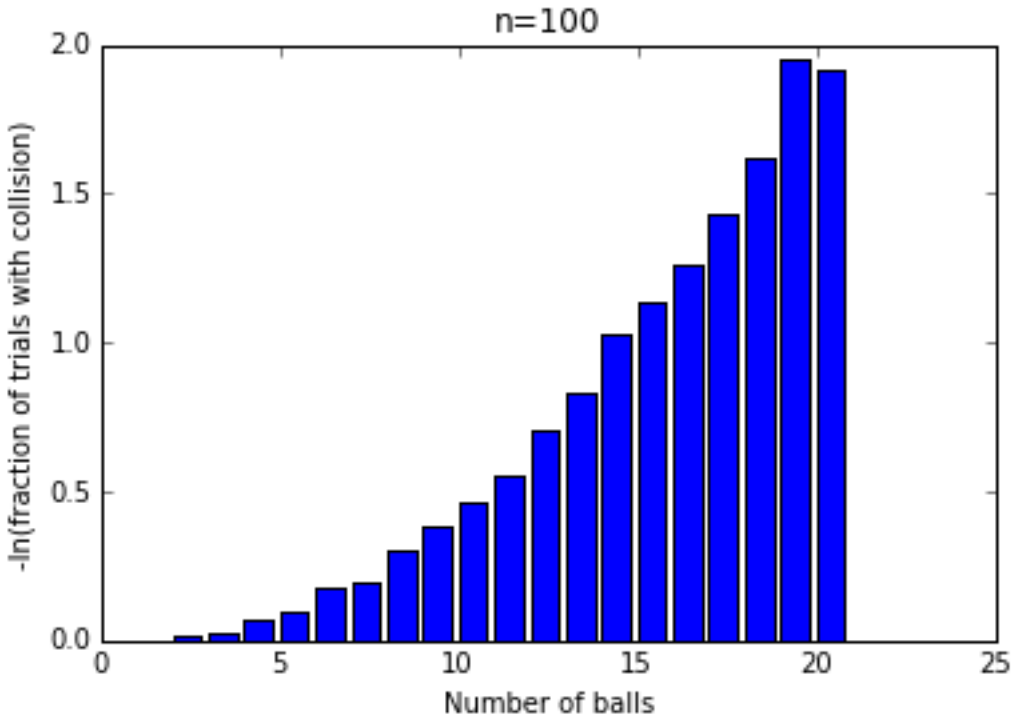
Repeat simulating throwing m balls in n bins for $n = 100, 365, 500, 1000$, and 20 values of m between $1 - 2\sqrt{n}$. Now for each n, plot the negative logarithm of the fraction of trials for which there were no collisions, for different values of m. Does this plot look linear in m? Quadratic? Exponential?

```
In [14]: def partC(nrange=[100,365,500,1000], t=1000):
             """
             Code for part (c)
             """

             for n in nrange:
                 print 'n =', n
                 mrange = [int(m) for m in np.linspace(1, 2*math.sqrt(n), 20)]

                 ###################
                 # YOUR CODE HERE
                 log_results = []
                 for m in mrange:
                     num_repeats = sum(run_many_ball_bin_trials_check_collision(run_many_ball_bin_trials
                     repeat_frac = 1 - num_repeats / t
                     #print 'Fraction of trials with no collisions when m = %i: %f' % (m, repeat_frac)
                     log_results.append(-math.log(repeat_frac))
                 plt.bar(mrange, log_results)
                 ###################

                 plt.xlabel('Number of balls')
                 plt.ylabel('-ln(fraction of trials with collision)')
```

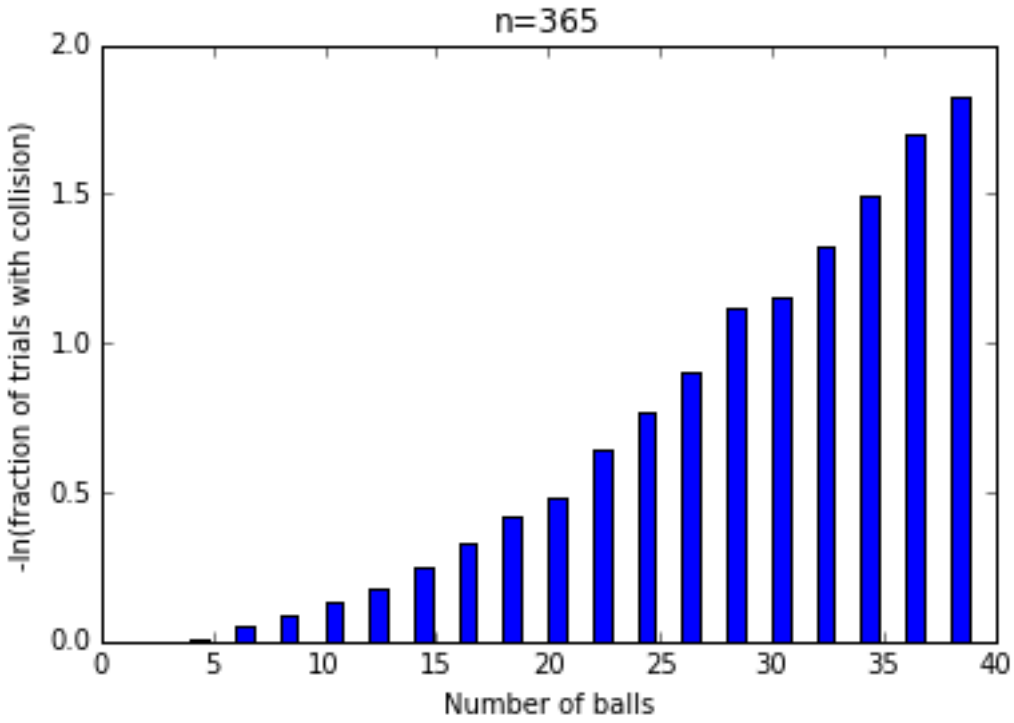
```
        plt.title('n=%i' % n)
        plt.show()
```

```
In [15]: partC()
```

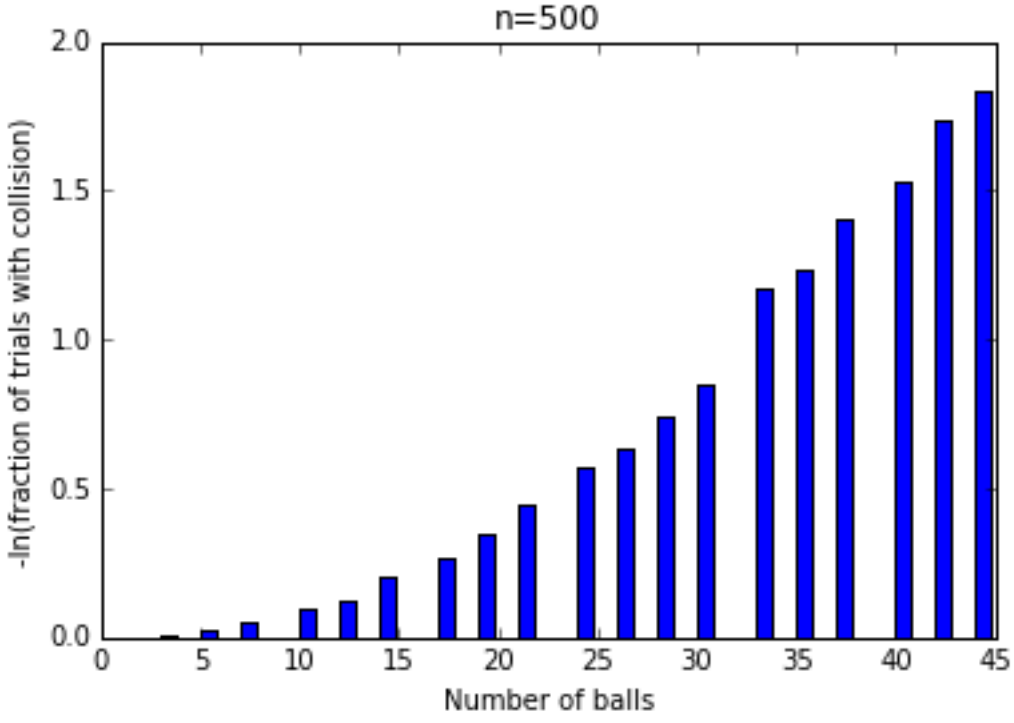
```
n = 100
```



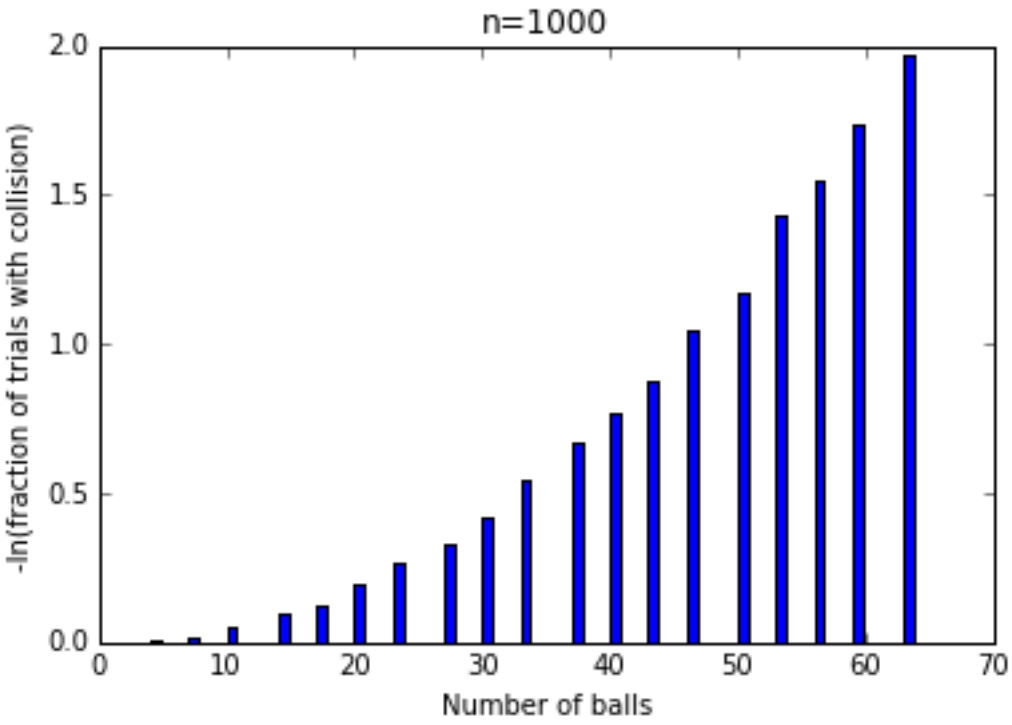

```
n = 365
```


n=365
-ln(fraction of trials with collision)
Number of balls
0.0
0.5
1.0
1.5
2.0
0
5
10
15
20
25
30
35
40

```
n = 500
```


n=500
-ln(fraction of trials with collision)
Number of balls
0.0
0.5
1.0
1.5
2.0
0
5
10
15
20
25
30
35
40
45

```
n = 1000
```



YOUR COMMENTS HERE:

Part (d): Overlay

Now overlay the plots in part (c) with $\frac{m^2}{2n}$. What do you notice?

From these graphs, and for each value of n, find the approximate m value such that $P(A) = 0.5$ for both the simulated data and $\frac{m^2}{2n}$. How much does this differ?

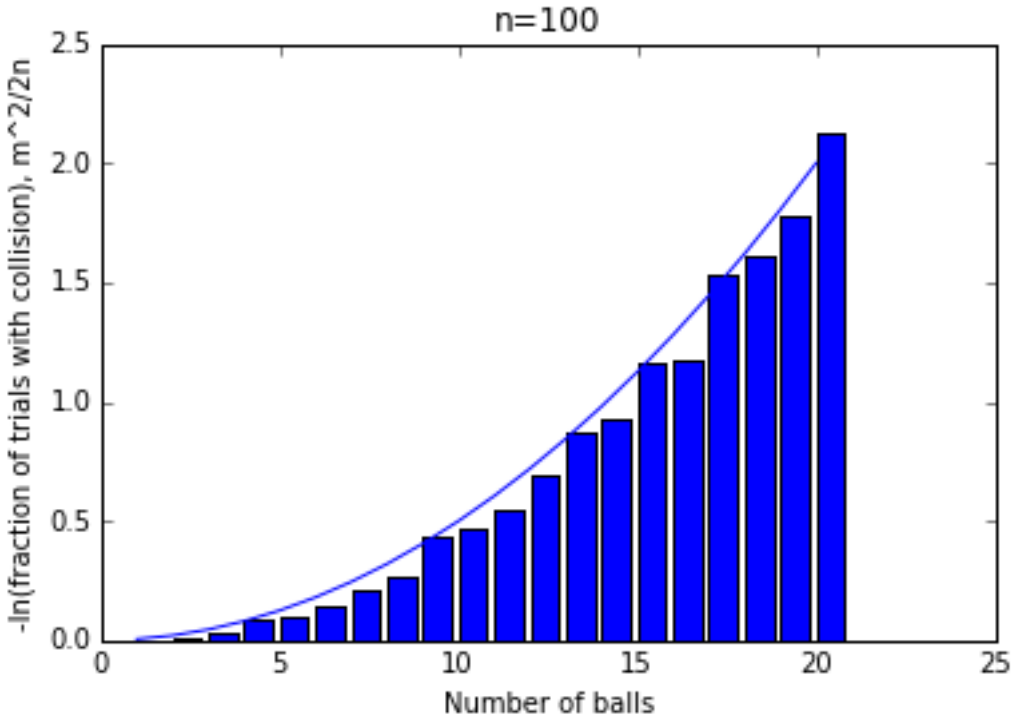
Note: Please find the m value using Python. You should not need to guess anything for this lab.

```
In [16]: def partD(nrange=[100,365,500,1000], t=1000):
             """
             Part (d) code
             """

             for n in nrange:
                 print 'n =', n
                 mrange = [int(m) for m in np.linspace(1, 2*math.sqrt(n), 20)]

                 ###################
                 # YOUR CODE HERE
                 log_results = []
                 m_over_2n_squared = []

                 for m in mrange:
                     num_repeats = sum(run_many_ball_bin_trials_check_collision(run_many_ball_bin_trials
                     repeat_frac = 1 - num_repeats/t
                     log_results.append(-math.log(repeat_frac))
                     m_over_2n_squared.append(m**2/(2*n))
```

```
    trials_diff = [abs(math.log(2) - v) for v in log_results]
    trials_m = mrange[trials_diff.index(min(trials_diff))]
    calc_diff = [abs(math.log(2) - v) for v in m_over_2n_squared]
    calc_m = mrange[calc_diff.index(min(calc_diff))]

    # Prints and plots the result
    print "\nTrial m for which P(A) = 0.5:", trials_m
    print "Calculated m for which P(A) = 0.5:", calc_m
    print "Absolute difference in m values for P(A) = 0.5:", abs(trials_m - calc_m)
    plt.bar(mrange, log_results)
    plt.plot(mrange, m_over_2n_squared)
    ###################

    plt.xlabel('Number of balls')
    plt.ylabel('-ln(fraction of trials with collision), m^2/2n')
    plt.title('n=%i' % n)
    plt.show()
```

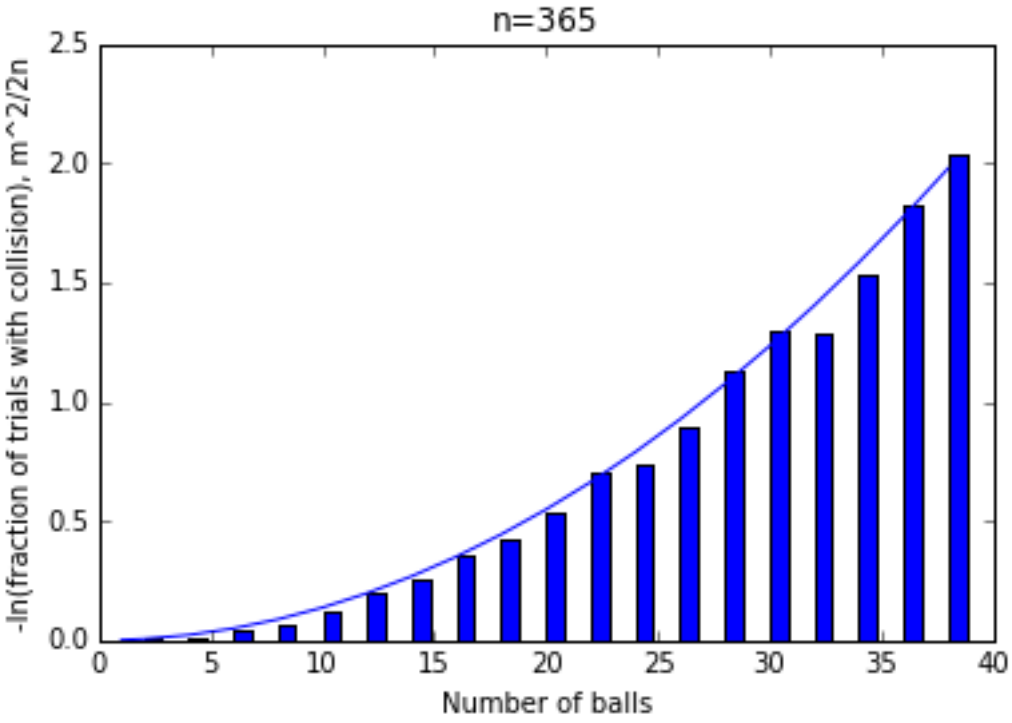
In [17]:
```
partD()
```

```
n = 100

Trial m for which P(A) = 0.5: 12
Calculated m for which P(A) = 0.5: 12
Absolute difference in m values for P(A) = 0.5: 0
```

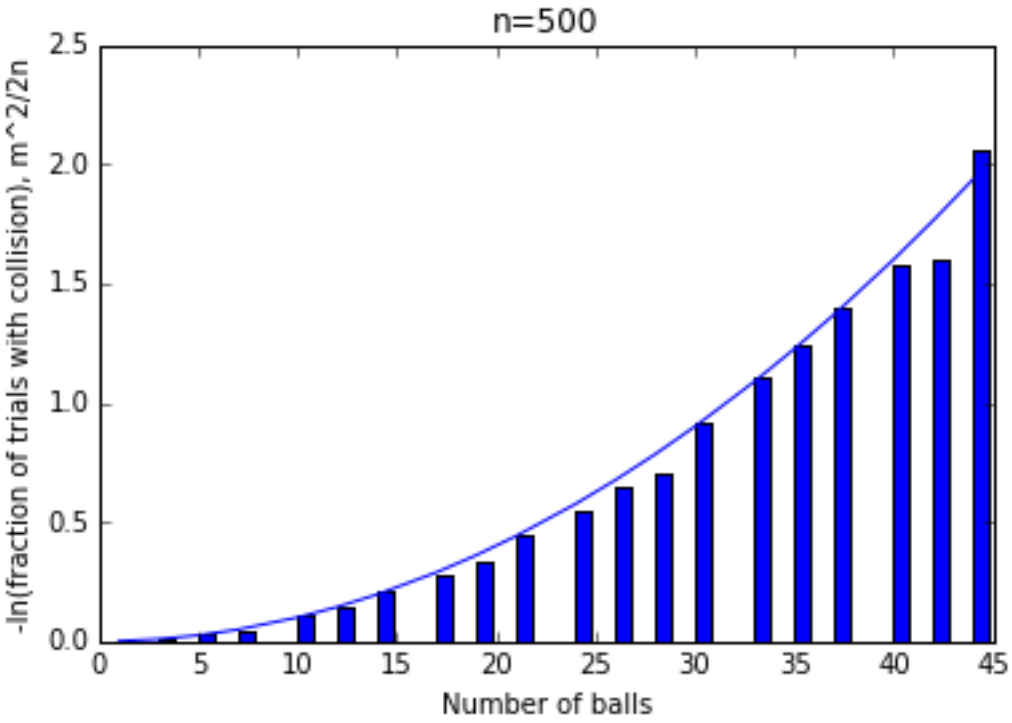



```
n = 365
```

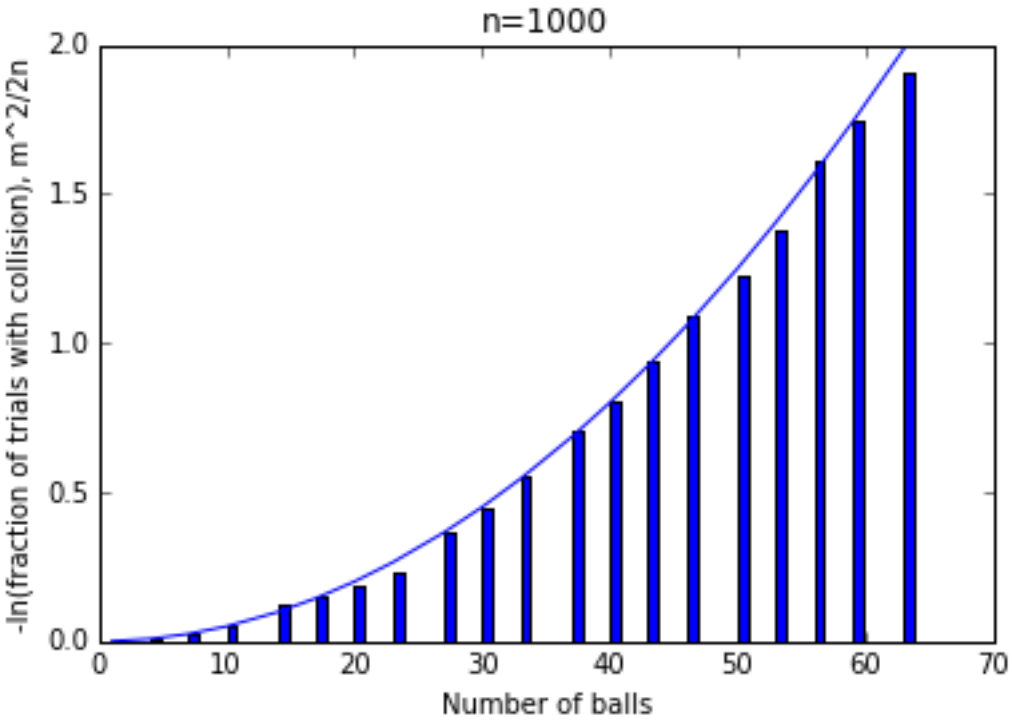
```
Trial m for which P(A) = 0.5: 22
Calculated m for which P(A) = 0.5: 22
Absolute difference in m values for P(A) = 0.5: 0
```




```
n = 500

Trial m for which P(A) = 0.5: 28
Calculated m for which P(A) = 0.5: 26
Absolute difference in m values for P(A) = 0.5: 2
```




```
n = 1000

Trial m for which P(A) = 0.5: 37
Calculated m for which P(A) = 0.5: 37
Absolute difference in m values for P(A) = 0.5: 0
```



YOUR COMMENTS HERE:

Part (e): Drunk Man

The drunk man is back, and he wants your help to plot 100 sample paths that he could take from time $t = 0$ to $t = 999$ (1000 timesteps). Assume the same probabilities as in Question 4, that is, the man moves forward with probability 0.5, backward with probability 0.3, and stays exactly where he is with probability 0.2. What do you observe about his paths? Where do you think he should end up at after 1000 timesteps, on average?

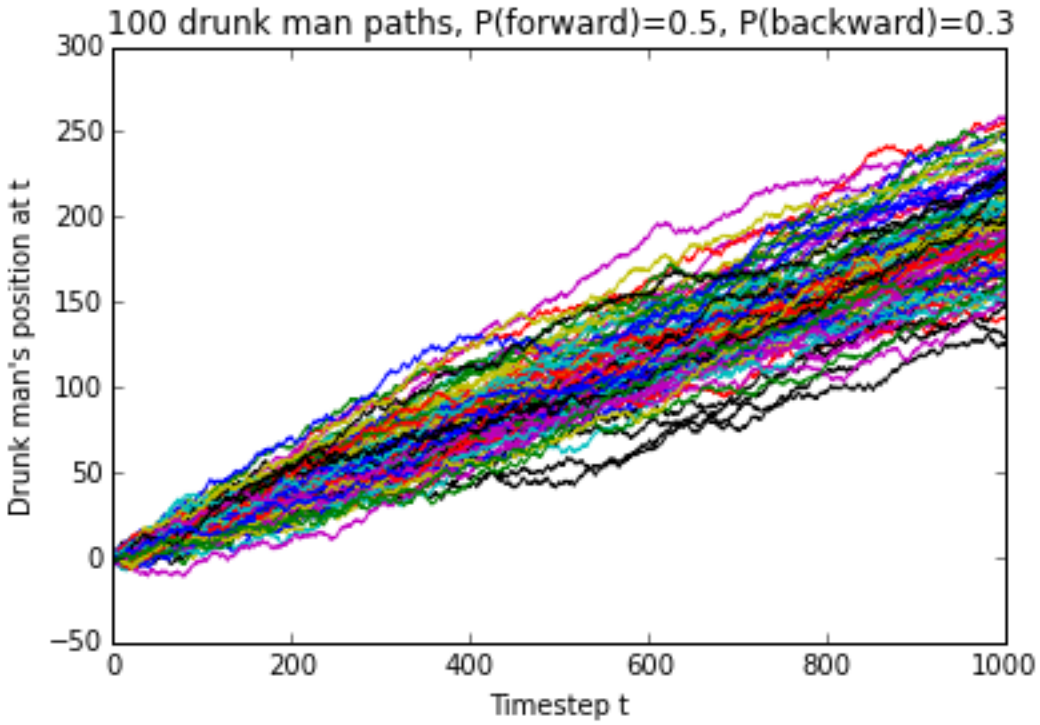
Hint: Implement the function `drunk_man`, which returns a list of elements that starts at 0, and every element thereafter is one more, one less, or equal to the previous one, with the correct probabilities for each possibility.

```
In [18]: def drunk_man(k=1000, forward_prob=0.5, backward_prob=0.3):
             """
             A random walk that moves forward with forward_prob probability,
             backward with backward_prob probability, and stay in the same
             place with 1 - forward_prob - backward_prob

             Returns a list of size k, where each element at the ith index
             is the location of the drunk man at time i.
             """

             path = [0]

             # YOUR CODE HERE
             ################
             for _ in range(1, k):
                 prob = random.random()
                 if prob < forward_prob:
```

```
            path.append(path[-1] + 1)
        elif forward_prob <= prob < forward_prob + backward_prob:
            path.append(path[-1] - 1)
        else:
            path.append(path[-1])
    ################

    assert len(path) == k, "Path length is incorrect"
    return path
```

In [19]:
```
def partE(k=1000, num_paths=100, forward_prob=0.5, backward_prob=0.3):
    """
    Code for part (e)

    YOUR CODE HERE
    """

    for _ in range(num_paths):
        plt.plot(drunk_man(k, forward_prob, backward_prob))
    plt.title(str(num_paths) + " drunk man paths, P(forward)=" + str(forward_prob) +
              ", P(backward)=" + str(backward_prob))
    plt.xlabel("Timestep t")
    plt.ylabel("Drunk man's position at t")
    plt.show()
```

In [20]:
```
partE()
```



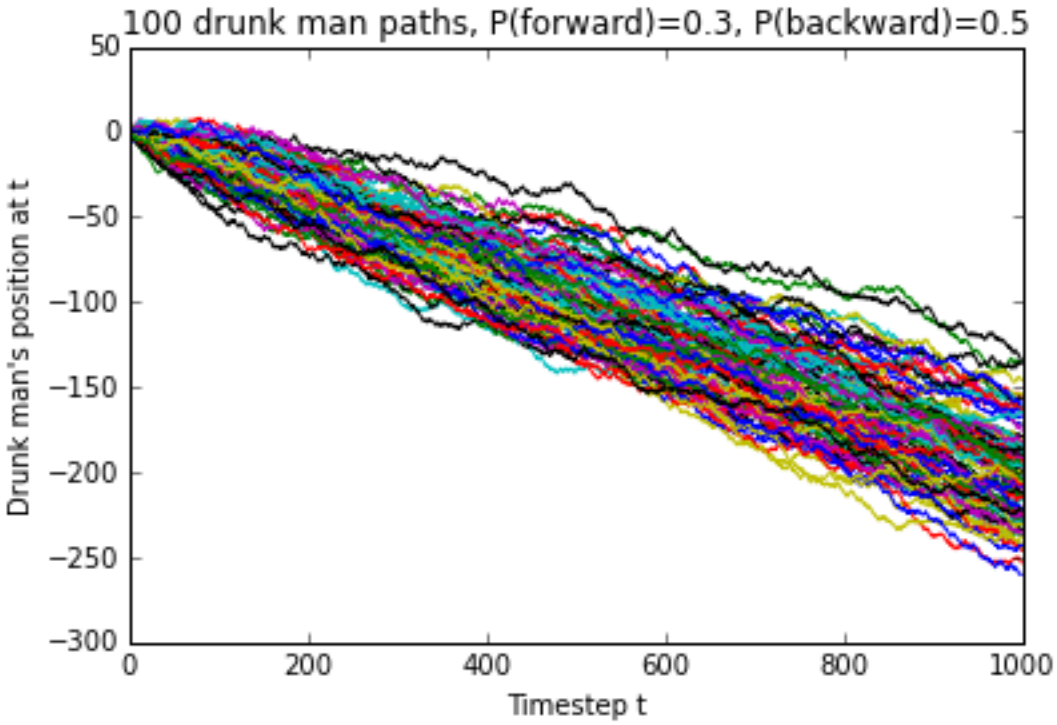
YOUR COMMENTS HERE:

Part (f): Drunk Man Needs Help Again

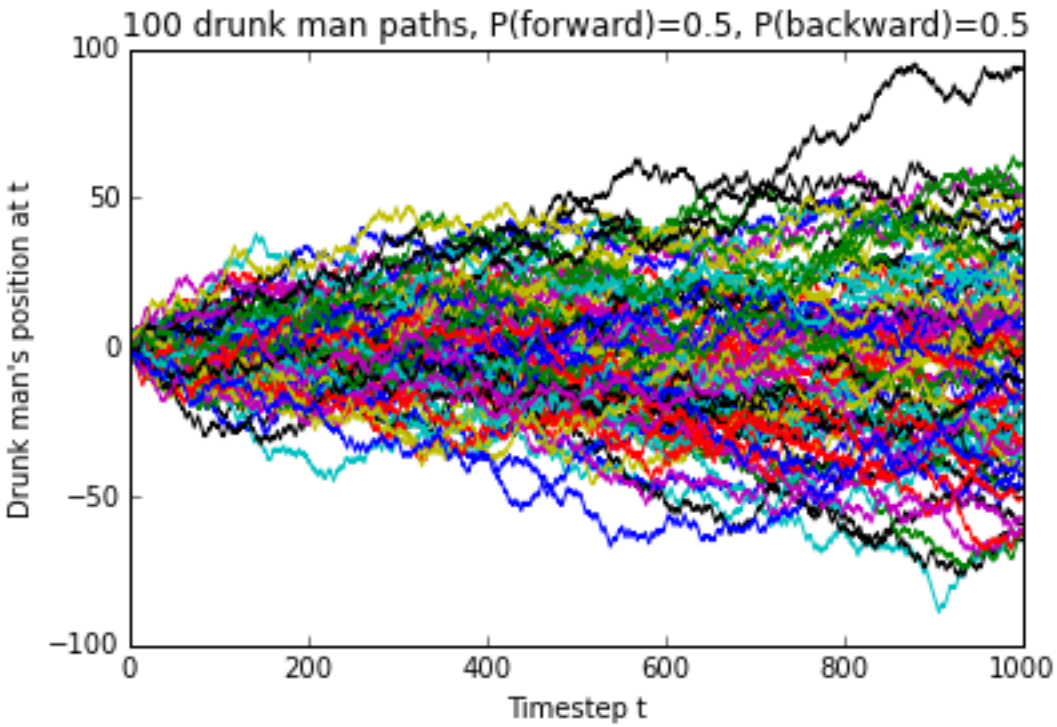
Redo the previous part, but this time, the man moves forward with probability 0.3, backward with probability 0.5, and stays exactly where he is with probability 0.2. What do you observe about his paths? Where do you think he should end up at after 1000 timesteps, on average?

Finally, assume the man moves forward and backward with probability 0.5 each, i.e. he never stays still. Plot his 100 sample paths. What do you observe? Does this graph remind you of something we have done in a previous virtual lab?

In [21]:
```
# YOUR CODE HERE (P(forward) = 0.3, P(backward) = 0.5)

partE(k=1000, num_paths=100, forward_prob=0.3, backward_prob=0.5)
```



In [22]:
```
# YOUR CODE HERE (P(forward) = 0.5, P(backward) = 0.5)

partE(k=1000, num_paths=100, forward_prob=0.5, backward_prob=0.5)
```



YOUR COMMENTS HERE:

Part (g): Drunk Man's Paths Count

One simple way of solving counting problems is to enumerate all possibilities, and then count the ones that we are looking for. In this question, let's check your answer to Question 5 from the last homework. Implement the function `count_paths(t)`, which generates all the paths the Drunk Man can take in t timesteps, then counts the number of paths in which he returns to 0 at time t and it is his first return. Remember that we no longer care about probabilities when counting paths.

Hint: One way to generate all possible paths is to use `itertools.product`. You might also want to implement the function `catalan(n)`, which computes the n^{th} Catalan number.

```
In [23]: def count_paths(t):
             """
             Count the number of path in which the Drunk Man returns to 0 at time t
             and it is his first return. We no longer care about probabilities here.

             YOUR CODE HERE
             """

             def is_first_return(path):
                 assert len(path) == t, "Path length is inconsistent with number of timesteps"
                 s = 0
                 for i in range(t):
                     if path[i] == "F":
                         s += 1
                     else:
                         s -= 1

                     # Returns to 0, but not at the last step
```

```
                if s == 0 and i != t-1:
                    return False
            return path.count("F") == path.count("B")

        paths = itertools.product("FB", repeat=t)
        return [is_first_return(path) for path in paths].count(True)
```

Test your implementation below. Both tests should print True if your implementation is correct.

```
In [24]: count_paths(4) == 2
```

```
Out[24]: True
```

```
In [25]: count_paths(10) == 28
```

```
Out[25]: True
```

Implement the function `catalan` below to compare your answer to the general answer from the last part of last week's homework.

```
In [26]: def catalan(n):
             """
             Computes the nth Catalan number

             YOUR CODE HERE
             """

             return choose(2*n, n) / (n+1)
```

Test your implementation below. Both tests should print True if your implementation is correct.

```
In [27]: catalan(0) == 1
```

```
Out[27]: True
```

```
In [28]: catalan(5) == 42
```

```
Out[28]: True
```

```
In [29]: # Let's check your answer to the last part of HW 11's Question 5
         # Don't try this for n > 10... or your computer will blow up.

         count_paths(2*8+2) == 2*catalan(8)
```

```
Out[29]: True
```

Part (h): Bernoulli Random Variables

The Bernoulli distribution is the probability distribution of a random variable which takes value 1 with success probability p and value 0 with failure probability $1 - p$. It can be used, for example, to represent a coin toss, where 1 is defined to mean "heads" and 0 is defined to mean "tails".

In other words:

$$f(k; p) = \begin{cases} p & \text{if } k = 1 \\ \\ 1 - p & \text{if } k = 0 \end{cases}$$

Related values:

$$E[X] = p$$

$$Var(X) = p(1-p)$$

Let's generate 100 Bernoulli random variables with success probability $p = 0.6$. Using SciPy and Python, you can do so as follows.

```
In [30]: bernoulli.rvs(0.6, size=100)

Out[30]: array([0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1,
                0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0,
                0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1,
                0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1,
                0, 1, 0, 0, 0, 0, 1, 1])
```

This is just a cleaner way of doing 100 flips of a 0.6-biased coin flip that we have done in previous virtual labs. Amazingly enough, you will see in this lab and the next that all the hard work you've done are all carried over. :-)

Let's now find out the mean, variance, and standard deviation of a Bernoulli random variable.

```
In [31]: round(bernoulli.mean(0.6), 1)

Out[31]: 0.6

In [32]: round(bernoulli.var(0.6), 2)

Out[32]: 0.24
```

We know analytically that the variance of a Bernoulli r.v. is $p(1-p)$. Let's see how close the above result is.

```
In [33]: 0.6*(1-0.6)

Out[33]: 0.24

In [34]: round(bernoulli.std(0.6), 2)

Out[34]: 0.49
```

Great! Hopefully the above functions give you a feel for the `scipy.stats` module, which will be useful in this question and the rest of the Virtual Labs.

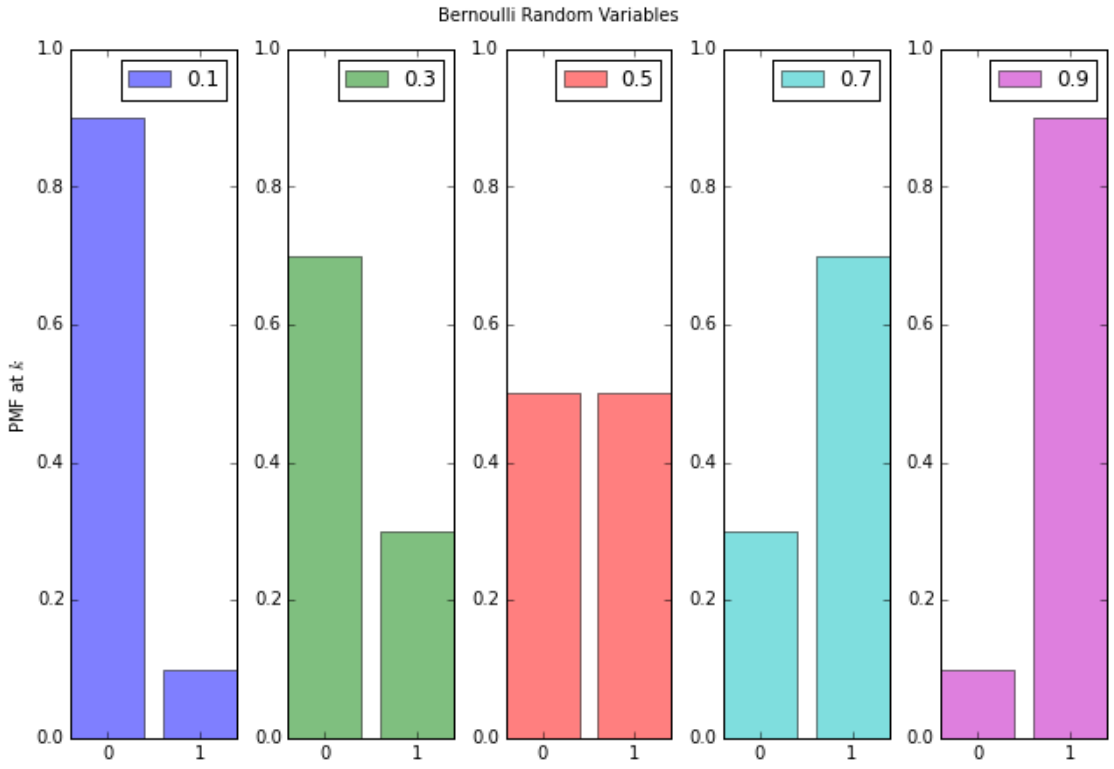
Plot the probability mass functions (pmf) for Bernoulli random variables with success probabilities of $0.1, 0.3, 0.5, 0.7$, and 0.9, respectively. Fill in the code below.

Hint: We're using axes instead of plt in this question, and also in some later questions. If you have gone over Tutorial 1B that was posted on Piazza, it's essentially the same as before. You will need to call `ax.bar` in the code below, similar to `plt.bar`. You will also find `bernoulli.pmf` useful.

```
In [35]: def partH(pranges=[0.1, 0.3, 0.5, 0.7, 0.9]):
             """
             Part (h) code
             """

             a = np.arange(2)
             fig, axes = plt.subplots(1, len(pranges))

             for i, p in enumerate(pranges):
```

```
        ax = axes[i]
        ax.set_xticks([x+0.4 for x in a])
        ax.set_xticklabels(a)
        ax.set_ylim(0, 1)

        # YOUR CODE HERE
        ################
        ax.bar(a, bernoulli.pmf(a, p), label=p, color=colors[i], alpha=0.5)
        ################

        ax.legend()
        if i == 0:
            ax.set_ylabel("PMF at $k$")

    fig.set_figwidth(9)
    fig.set_figheight(6)
    fig.tight_layout()
    fig.suptitle("Bernoulli Random Variables", y=1.02)
    plt.show()
```

In [36]: partH()



Part (i): Coin Tosses, Revisited

The binomial distribution gives us the probability of observing k successes, each with a probability p, out of N attempts.

$$f(k; N; p) = \binom{N}{k} p^k (1-p)^{N-k}$$

where

$$\binom{N}{k} = \frac{N!}{k!(N-k)!}$$

with $k = 0, 1, 2, \ldots, N$
Related values:

$$E[X] = Np$$

$$Var(X) = Np(1-p)$$

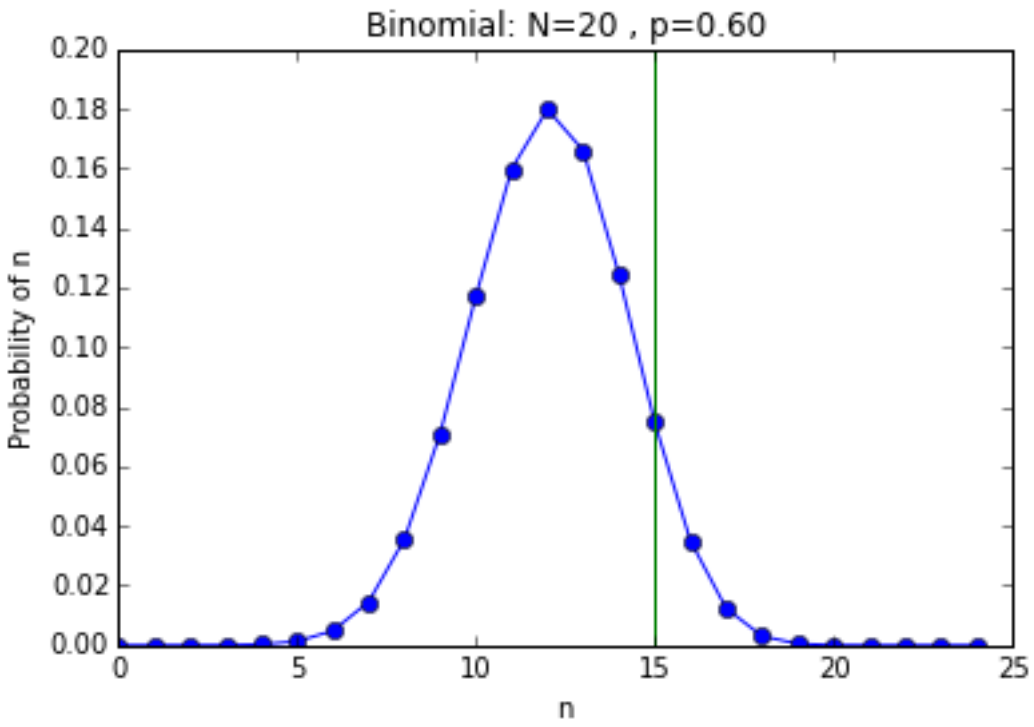
If I toss a 0.6-biased coin 20 times, what is the approximate probability of getting exactly 15 heads? Answer this question graphically by plotting the PMF of a binomial random variable with parameters $(20, 0.6)$. Do not carry out the exact calculation – we know you know how to do that already.

Looking at the plot, at which value of n is the probability of getting n heads maximized?

Hint: As before, you will find `binom.pmf` helpful.

```
In [47]: def partI(N=20, p=0.6):
             """
             Part (i) code
             """

             n = np.arange(0, N+5)

             # YOUR CODE HERE
             #################
             y = binom.pmf(n, N, p)
             plt.plot(n, y, 'o-')
             #################

             plt.plot([15 for _ in n], n)  # Plot the vertical line at x=15 to help you graphically ans
             plt.title("Binomial: N=%i , p=%.2f" % (N, p))
             plt.xlabel("n")
             plt.ylabel("Probability of n")
             plt.ylim(0, 0.2)
             plt.yticks(np.linspace(0, 0.2, 11))
             plt.show()
```

```
In [48]: partI()
```



YOUR COMMENTS HERE:

Part (j): Socialize

Let $n = 100$. Plot the probabilities in Question 9 parts (a) and (b) by varying m from 1 to n. You might want to implement the two functions below for the two strategies in parts (a) and (b).

```
In [39]: def strategy1(m, n=100):
             """
             Code for strategy 1, which is part (a) of Question 9.
             Returns the probability that you talk with everyone at most once.

             YOUR CODE HERE
             """

             return math.factorial(n) / (pow(n, m) * math.factorial(n-m))
```

```
In [40]: def strategy2(m, n=100):
             """
             Code for strategy 2, which is part (b) of Question 9.
             Returns the probability that you talk with everyone at most once.

             YOUR CODE HERE
             """

             if 1 <= m <= 2:
                 return 1
             result = 1
             for i in range(2, m):
                 result *= (1 - i*(i-1) / (n*(n-1)))
             return result
```

```
In [41]: def partJ(n=100):
             """
             Code for part (e) of Question 9
             """

             mrange = xrange(1, n+1)

             # YOUR CODE HERE
             #################
             plt.plot(mrange, [strategy1(m, n) for m in mrange], label="Strategy 1")
             plt.plot(mrange, [strategy2(m, n) for m in mrange], label="Strategy 2")
             #################

             plt.legend()
             plt.title("Probability that you talk with everyone at most once, for $n=100$ people", y=1.0
             plt.xlabel("$m$")
             plt.ylabel("$P_m$")
             plt.show()
```

```
In [42]: partJ()
```

Part (k): Expected Conversations

Plot the expected numbers of conversations in Question 9 parts (c) and (d) by varying n from 2 to 100.

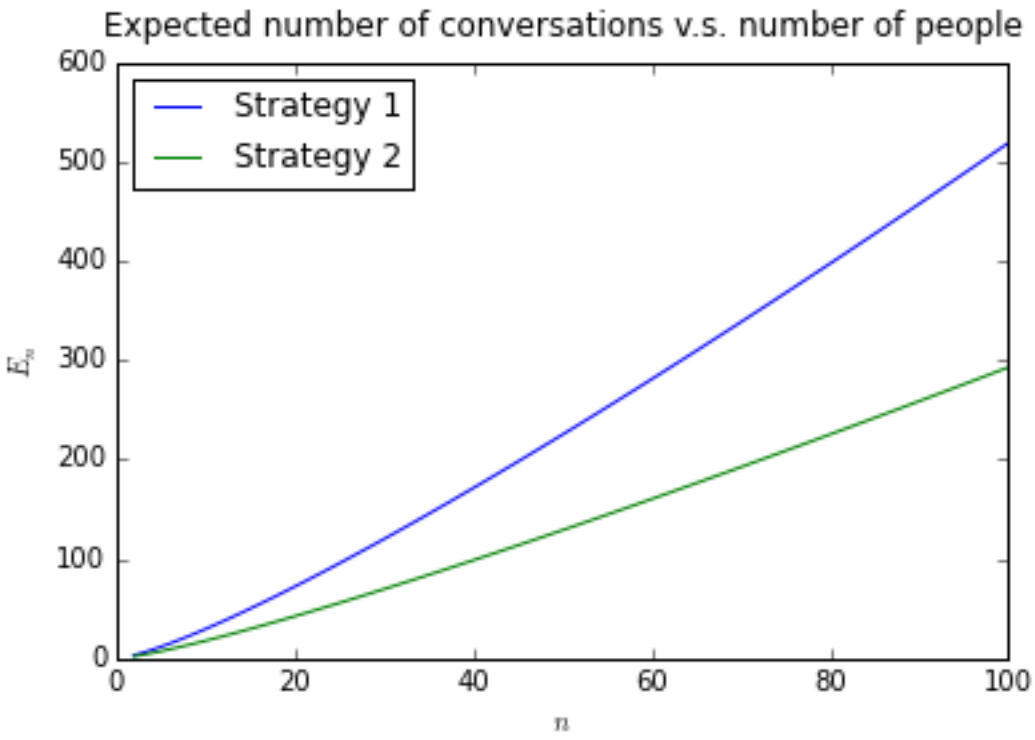
```
In [43]: def e_strategy1(n):
             """
             Code for strategy 1, which is part (c) of Question 9.
             Returns the expected number of conversation using the strategy in part (a).
```

```
    YOUR CODE HERE
    """

    return n * sum([1/i for i in xrange(1, n+1)])
```

In [44]:
```
def e_strategy2(n):
    """
    Code for strategy 2, which is part (c) of Question 9.
    Returns the expected number of conversation using the strategy in part (b).

    YOUR CODE HERE
    """

    if n == 2:
        return 2
    return 2 + choose(n, 2) *\
           sum([1 / (choose(n, 2) - choose(i, 2)) for i in xrange(2, n)])
```

In [45]:
```
def partK(max_num_people=100):
    """
    Code for part (f) of Question 9
    """

    nrange = xrange(2, max_num_people+1)

    # YOUR CODE HERE
    #################
    plt.plot(nrange, [e_strategy1(n) for n in nrange], label="Strategy 1")
    plt.plot(nrange, [e_strategy2(n) for n in nrange], label="Strategy 2")
    #################

    plt.legend(loc=2)
    plt.title("Expected number of conversations v.s. number of people", y=1.02)
    plt.xlabel("$n$")
    plt.ylabel("$E_n$")
    plt.show()
```

In [46]:
```
partK()
```



Congratulations! You are done with Virtual Lab 12.

Don't forget to convert this notebook to a pdf document, merge it with your written homework, and submit both the pdf and the code (as a zip file) on glookup.

Reminder: late submissions are NOT accepted. If you have any technical difficulty, resolve it early on or use the provided VM.