lab14sol

December 8, 2014

1 Virtual Lab 14 Solution: Random Variables and Distributions

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

Due Date: Monday, December 8th, 2014 at 12pm **Name**:
Login: cs70-
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 string
import itertools
from scipy.stats import bernoulli, poisson, binom, geom, expon, norm
```

Introduction

In this week's lab, we will explore common discrete random variables and their corresponding distributions.

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

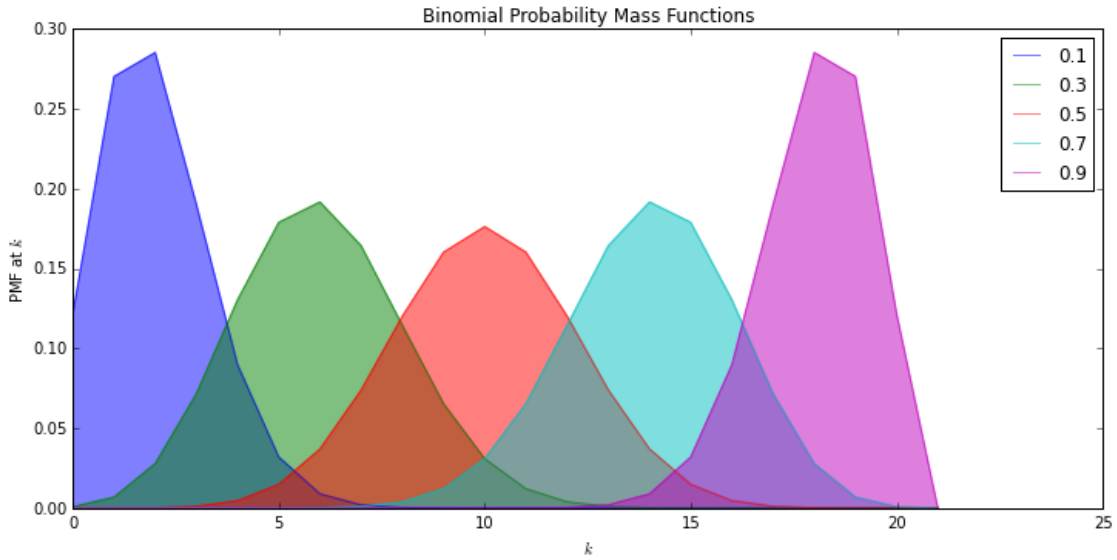
Part (a): Binomial Random Variables

Plot the PMFs of binomial random variables with $N = 20$, and with success probabilities $p = 0.1, 0.3, 0.5, 0.7$, and 0.9, respectively. You should have 5 different plots in one figure.

What do you observe as the success probabilities increase?

In [4]:
```
def partA(N=20, pranges=[0.1, 0.3, 0.5, 0.7, 0.9]):
    """
    Part (a) code
    """

    plt.figure(figsize=(10, 5))
    k = np.arange(22)

    for i, p in enumerate(pranges):
        rv = binom(N, p)

        # YOUR CODE HERE
        # Try setting `color=colors[i]` to make it easier to differentiate the plots
        ################
        plt.plot(k, rv.pmf(k), label=p, color=colors[i], alpha=0.5)
        ################

        plt.fill_between(k, rv.pmf(k), color=colors[i], alpha=0.5)

    plt.legend()
    plt.title("Binomial Probability Mass Functions")
    plt.tight_layout()
    plt.ylabel("PMF at $k$")
    plt.xlabel("$k$")
    plt.show()
```

In [5]:
```
partA()
```



YOUR COMMENTS HERE:

Part (b): Binomial CMFs

In probability theory and statistics, the cumulative mass function (CMF) describes the probability that a real-valued discrete random variable X with a given probability distribution will be found to have a value less than or equal to x. Mathematically, we define the CMF $F_X(x)$ as follows

$$F_X(x) = P(X \leq x)$$

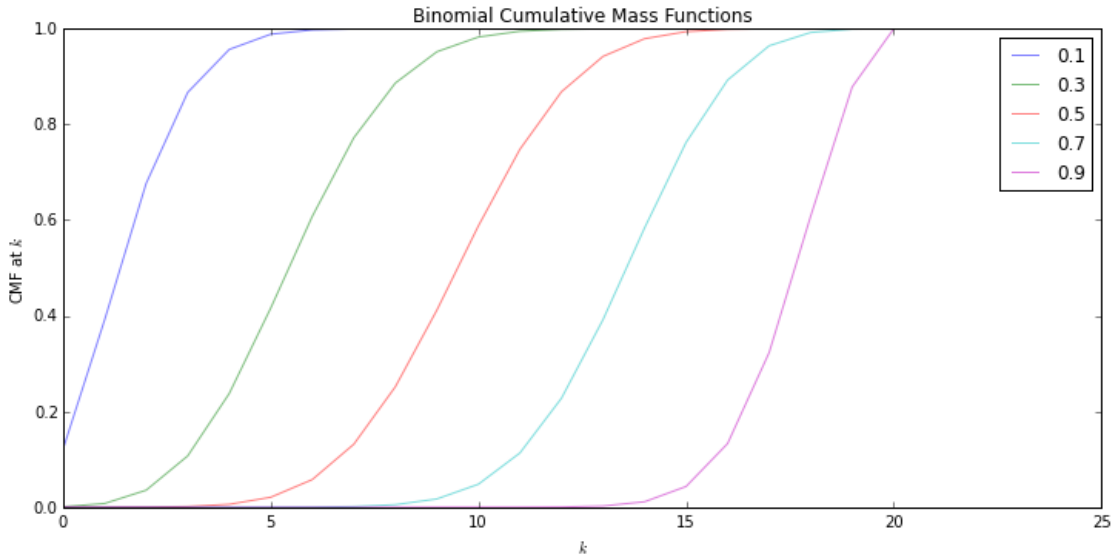
where the right-hand side represents the probability that the random variable X takes on a value less than or equal to x.

Plot the CMFs of the five binomial random variables from part (a). These should all be increasing curves. At what value does each CMF plot converge to and stay at 1 (i.e. at what value can you be almost 100% certain that the number of heads (or successful trials) is less than such value)?

```
In [6]: def partB(N=20, pranges=[0.1, 0.3, 0.5, 0.7, 0.9]):
            """
            Part (d) code
            """

            plt.figure(figsize=(10, 5))
            k = np.arange(N+5)

            for i, p in enumerate(pranges):
                rv = binom(N, p)

                # YOUR CODE HERE
                # You should use rv.cdf, it's the same as rv.cmf, but the latter doesn't exist in the l
                ################
                plt.plot(k, rv.cdf(k), label=p, color=colors[i], alpha=0.5)
                ################

            plt.legend()
            plt.title("Binomial Cumulative Mass Functions")
            plt.tight_layout()
```

```
            plt.ylabel("CMF at $k$")
            plt.xlabel("$k$")
            plt.show()
```

```
In [7]: partB()
```



YOUR COMMENTS HERE:

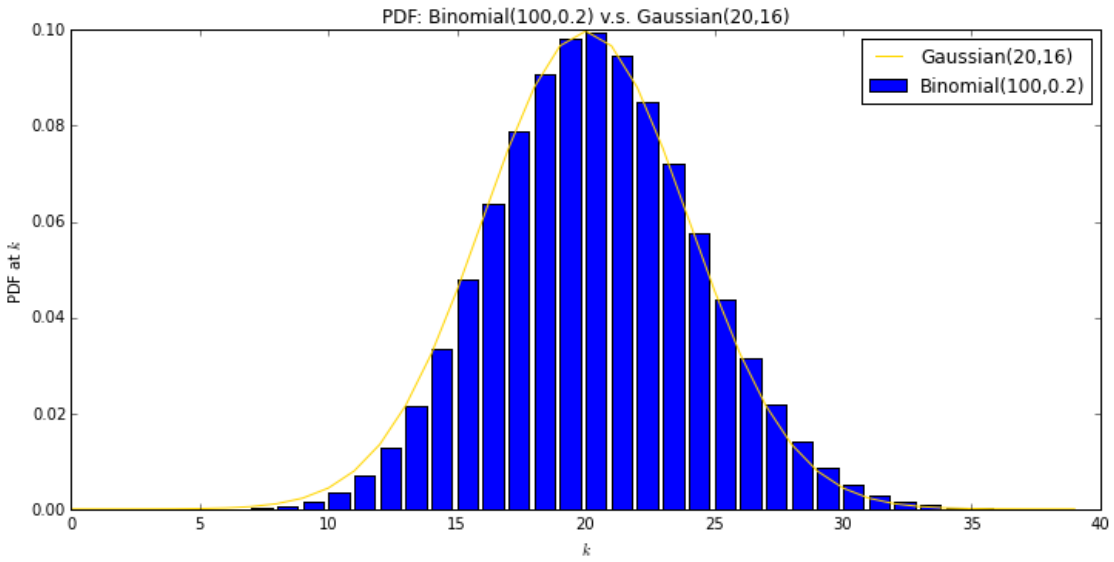
Part (c): Binomial v.s. Gaussian

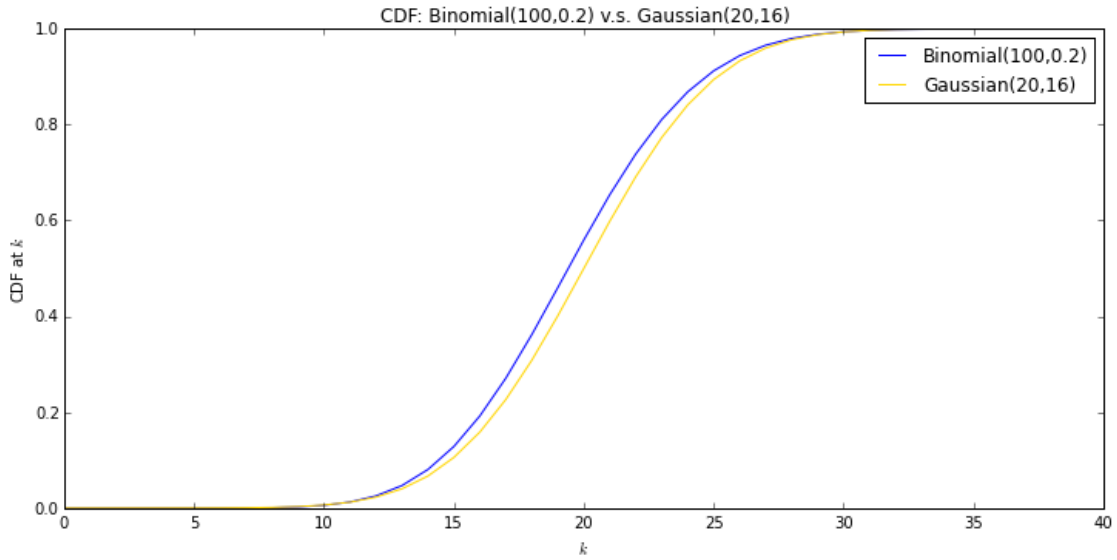
Plot a binomial distribution with parameters ($N = 100, p = 0.2$) in a bar chart. Then, overlay your plot with a normal distribution with parameters ($\mu = 20, \sigma^2 = 16$). What do you observe? Derive an approximation between the two distributions using a concept you learned in this week's lecture.

```
In [8]: def partC(N=100, p=0.2, mu=20, sigma=4):
            """
            Part (c) code
            """

            k = np.arange(0, 40)
            var = sigma ** 2

            # Plots and compares the PDFs
            plt.figure(figsize=(10, 5))

            # Plots binomial distribution pmf in a bar plot
            # YOUR CODE HERE
            #################
            plt.bar(k, binom(N, p).pmf(k), label="Binomial(%d,%.1f)" % (N, p), color="blue")
            #################

            # Overlays the pdf normal distribution
            # YOUR CODE HERE
            #################
            plt.plot(k, norm(mu, sigma).pdf(k), label="Gaussian(%d,%d)" % (mu, var), color="gold")
            #################
```

```
    plt.legend()
    plt.title("PDF: Binomial(%d,%.1f) v.s. Gaussian(%d,%d)" % (N, p, mu, var))
    plt.tight_layout()
    plt.ylabel("PDF at $k$")
    plt.xlabel("$k$")
    plt.show()

    #----------------------------------------#

    # Plots and compares the CDFs
    plt.figure(figsize=(10, 5))

    # Plots binomial distribution cmf
    # YOUR CODE HERE
    #################
    plt.plot(k, binom(N, p).cdf(k), label="Binomial(%d,%.1f)" % (N, p), color="blue")
    #################

    # Overlays the cdf normal distribution
    # YOUR CODE HERE
    #################
    plt.plot(k, norm(mu, sigma).cdf(k), label="Gaussian(%d,%d)" % (mu, var), color="gold")
    #################

    plt.legend()
    plt.title("CDF: Binomial(%d,%.1f) v.s. Gaussian(%d,%d)" % (N, p, mu, var))
    plt.tight_layout()
    plt.ylabel("CDF at $k$")
    plt.xlabel("$k$")
    plt.show()
```

In [9]:
```
partC()
```





YOUR COMMENTS HERE:

YOUR DERIVATION HERE:

(If you don't LaTeX or don't want to learn it, just derive an approximation between the two distributions on paper&pencil as part of your written homework, but please clearly tell the readers where they can find your work.)

Part (d): Geometric Random Variables

Given an experiment with two possible outcomes, S and F, with probability p for S and $1-p$ for F, the geometric random variable is the number of experiments run until the first outcome of S, i.e. the number of trials required to reach the first success. The number includes the last experiment with the S outcome. This type of R.V. was seen in the coupon collector's problem, where the successful event was collecting a coupon that you didn't have.

Its PMF and CMF are given by:

$$f(k;p) = p\,(1-p)^{k-1}$$
$$F(k;p) = 1-(1-p)^{k}$$

, respectively.

Related Values:

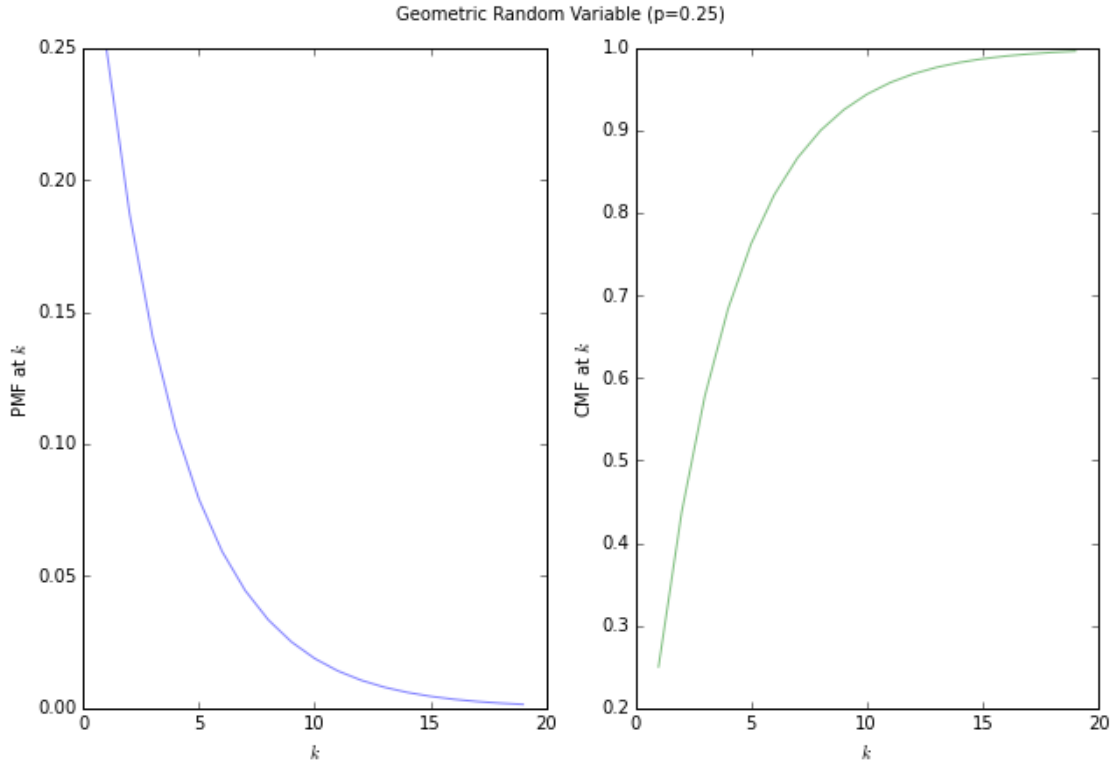
$$E\,[X] = \frac{1}{p}$$
$$Var\,(X) = \frac{1-p}{p^2}$$

Plot the PMF and CMF of a geometric random variable with parameter $p = 0.25$. What do you observe about the two curves?

```
In [10]: def partD(p=0.25):
             """
             Part (d) code
             """

             a = np.arange(1, 20)
             fig, axes = plt.subplots(1, 2)
```

```
    for i in range(len(axes)):
        ax = axes[i]
        ax.set_xlabel("$k$")

        if i == 0:
            # YOUR CODE HERE
            # Plots the pmf
            #################
            ax.plot(a, geom.pmf(a, p), color=colors[i], alpha=0.5)
            ax.set_ylabel("PMF at $k$")
            #################
        else:
            # YOUR CODE HERE
            # Plots the cmf (hint: use geom.cdf, there's no geom.cmf, but they are the same th
            #################
            ax.plot(a, geom.cdf(a, p), color=colors[i], alpha=0.5)
            ax.set_ylabel("CMF at $k$")
            #################

    fig.set_figwidth(9)
    fig.set_figheight(6)
    fig.tight_layout()
    fig.suptitle("Geometric Random Variable (p=%.2f)" % p, y=1.02)
    plt.show()
```

In [11]: `partD()`



YOUR COMMENTS HERE:

Part (e): Lottery Tickets

Suppose that every day, Alice buys a lottery ticket, and will only stop buying lottery tickets when she wins. Let p be the probability that on any given day, Alice wins the lottery. Let X represent the totaly number of lottery tickets Alice buys.

Simulate $m = 10,000$ trials of Alice buying lottery tickets until she gets a winner. Use $p = 0.2$, and plot a histogram with the number of lottery tickets on the x-axis and the fraction of trials with each of these outcomes on the y-axis.

Overlay $f(x) = \mathbb{P}(X = x) = (1-p)^{x-1}p$, which is the probability of getting $(x-1)$ non-winning tickets, then a winning ticket. What is the average number of lottery tickets Alice has to buy?

Hint: Implement the functions `lottery_trial`, which computes the number of tickets Alice has to buy to win the lottery, and `lottery_pmf`, which computes the probability of buying x lottery tickets (i.e. $f(x)$).

```
In [12]: def biased_coin(p):
             """
             Creates a biased coin with Pr(Head) = p
             Returns True if heads and false otherwise
             """

             return random.random() <= p
```

```
In [13]: def lottery_trial(p):
             """
             Computes the number of tickets Alice has to buy to win the lottery.
             Hint: use `biased_coin` and recursion.

             YOUR CODE HERE
             """

             if biased_coin(p):
                 return 1
             else:
                 return 1 + lottery_trial(p)
```

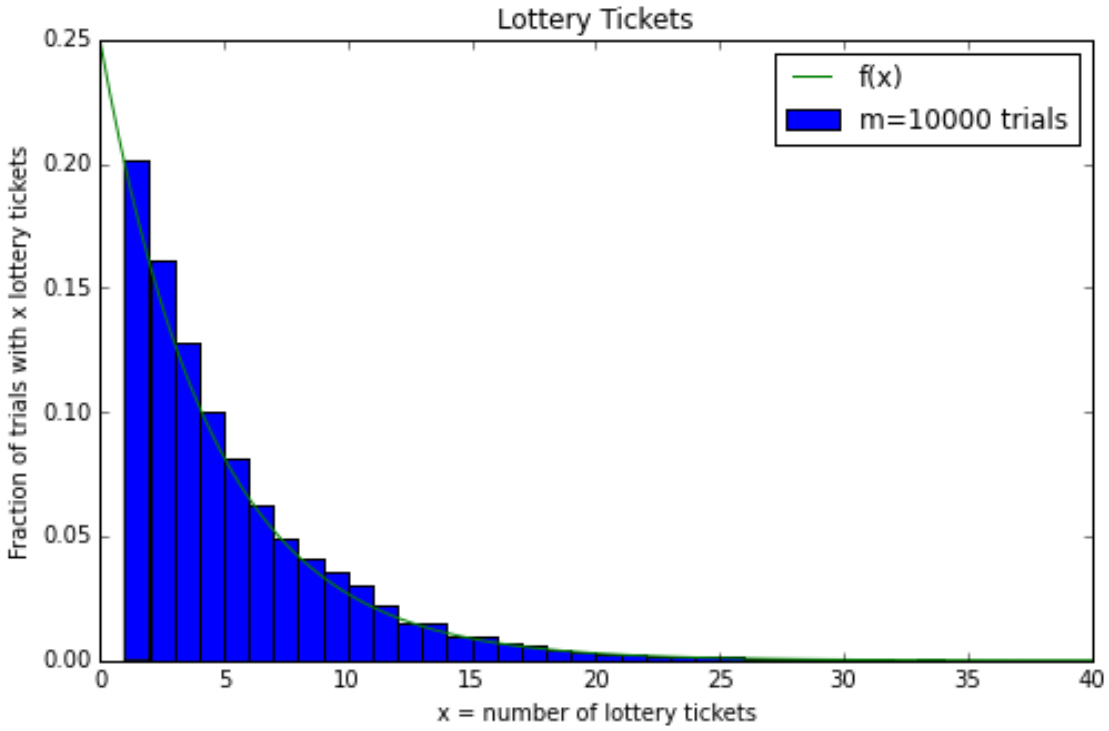
```
In [14]: def lottery_pmf(x, p):
             """
             Computes the probability of buying x lottery tickets (i.e. f(x)).
             Parameters: x, the number of lottery tickets and p, the probability of winning on a given
             Hint: use `pow`

             YOUR CODE HERE
             """

             return pow(1-p, x-1) * p
```

```
In [15]: def partE(p=0.2, m=10000):
             """
             Part (e) code
             """

             plt.figure(figsize=(8, 5))
             bin_width = 1

             # Plots a histogram with the number of lottery tickets on the x-axis
             # and the fraction of trials with each of these outcomes on the y-axis.
```

```
    # You can assume a bin_width of 1, and make sure to set normed=True
    # YOUR CODE HERE
    ################
    trial_list = []
    for trial in range(m):
        trial_list.append(lottery_trial(p))
    plt.hist(trial_list, bins=np.arange(min(trial_list), max(trial_list)+bin_width*2, bin_width
             normed=True, label="m=" + str(m) + " trials")
    ################

    x_values = np.linspace(0, max(trial_list)+bin_width*2, 100)
    # Overlay f(x)
    # YOUR CODE HERE
    ################
    plt.plot(x_values, [lottery_pmf(x, p) for x in x_values], label="f(x)")
    ################

    # Prints the average number of lottery tickets
    # YOUR CODE HERE
    ################
    print("Average number of tickets: " + str(sum(trial_list) / len(trial_list)))
    ################

    # set up plot
    plt.xlim(0, 40)
    plt.xlabel("x = number of lottery tickets")
    plt.ylabel("Fraction of trials with x lottery tickets")
    plt.title("Lottery Tickets")
    plt.legend()
    plt.show()
```

In [16]:
```
partE()
```

```
Average number of tickets: 5.0581
```



YOUR COMMENTS HERE:

Part (f): CMF of Lottery Tickets

The cumulative mass function (cmf) for X, i.e. the number of tickets Alice has to buy until she wins the lottery for the first time, is given as follows: $F(x) = \mathbb{P}(X \leq x)$. Use your histogram from the previous part to plot an "empirical CMF" (i.e. for each number of tickets x, plot the fraction of trials where Alice bought x or fewer tickets). Then overlay $F(x)$.

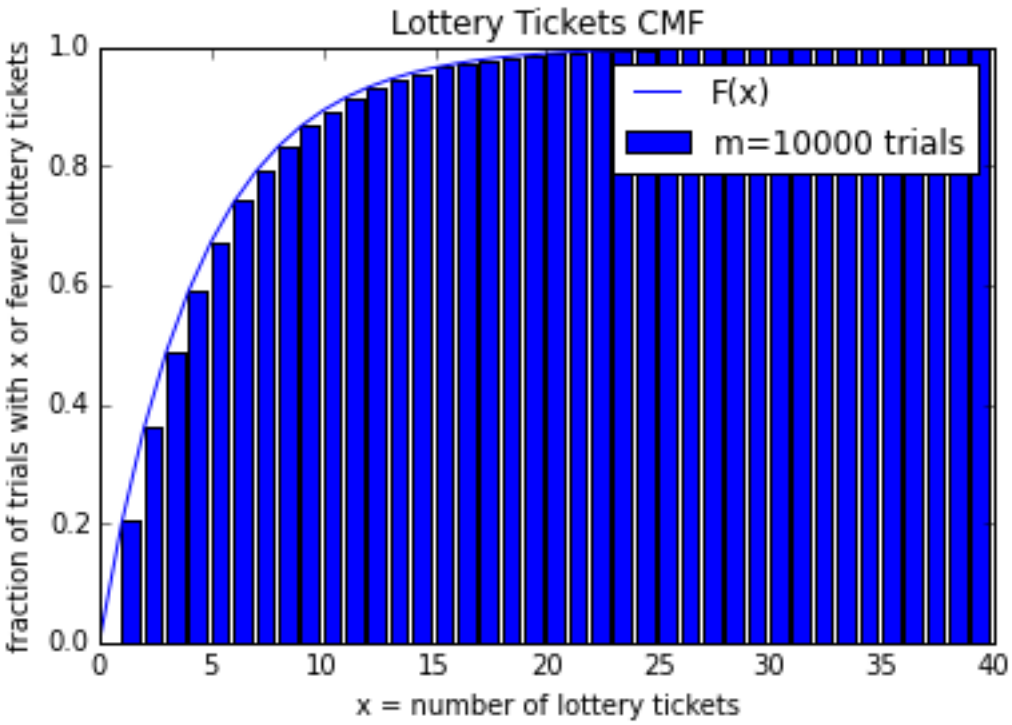
Hint: Implement the function `lottery_cmf`, which computes the probability of buying x or fewer lottery tickets.

```
In [17]: def lottery_cmf(x, p):
             """
             Computes the probability of buying x or fewer lottery tickets (i.e. F(x))
             Parameters: x, the number of lottery tickets and p, the probability of winning on a given

             YOUR CODE HERE
             """

             return 1 - pow(1-p, x)
```

```
In [18]: def partF(p=0.2, m=10000):
             """
             Part (f) code
             """

             trial_list = []
             for trial in range(m):
                 trial_list.append(lottery_trial(p))
             x_values = range(0, max(trial_list)+1)
```

```
    # Plots empirical cmf in a bar chart
    # YOUR CODE HERE
    ################
    empirical_cmf = []
    for i in x_values:
        empirical_cmf.append([x <= i for x in trial_list].count(True) / m)
    plt.bar(x_values, empirical_cmf, label="m=" + str(m) + " trials")
    ################

    # Overlays F(x)
    # YOUR CODE HERE
    ################
    plt.plot(x_values, [lottery_cmf(x, p) for x in x_values], label="F(x)")
    ################

    # Sets up plot
    plt.xlabel("x = number of lottery tickets")
    plt.ylabel("fraction of trials with x or fewer lottery tickets")
    plt.title("Lottery Tickets CMF")
    plt.legend()
    plt.show()
```

In [19]:
```
partF()
```



Part (g): Chebyshev's Inequality

In part (c) of VL 11, we went directly to one of the most powerful bounds we have, the Chernoff bound. Let's now look at a much simpler bound – the Chebyshev's inequality. For coin tosses, this inequality says

$$P(|S_k - kp| \geq \epsilon k) \leq \frac{p(1-p)}{k\epsilon^2}$$

, where S_k is the number of heads observed in k tosses of a p-biased coin.

Notice here that Chebyshev's inequality looks at two-sided deviations. We count both when S_k is much bigger than kp and when it is much smaller than kp. This is a big difference from the Chernoff's bound.

For $\epsilon = 0.1, 0.2, 0.3$, plot and compare the actual frequency of such deviations to what Chebyshev's inequality estimates for $m = 1000$ trials of $k = [10, 200]$ tosses of a $p = 0.7$-biased coin. What do you observe?

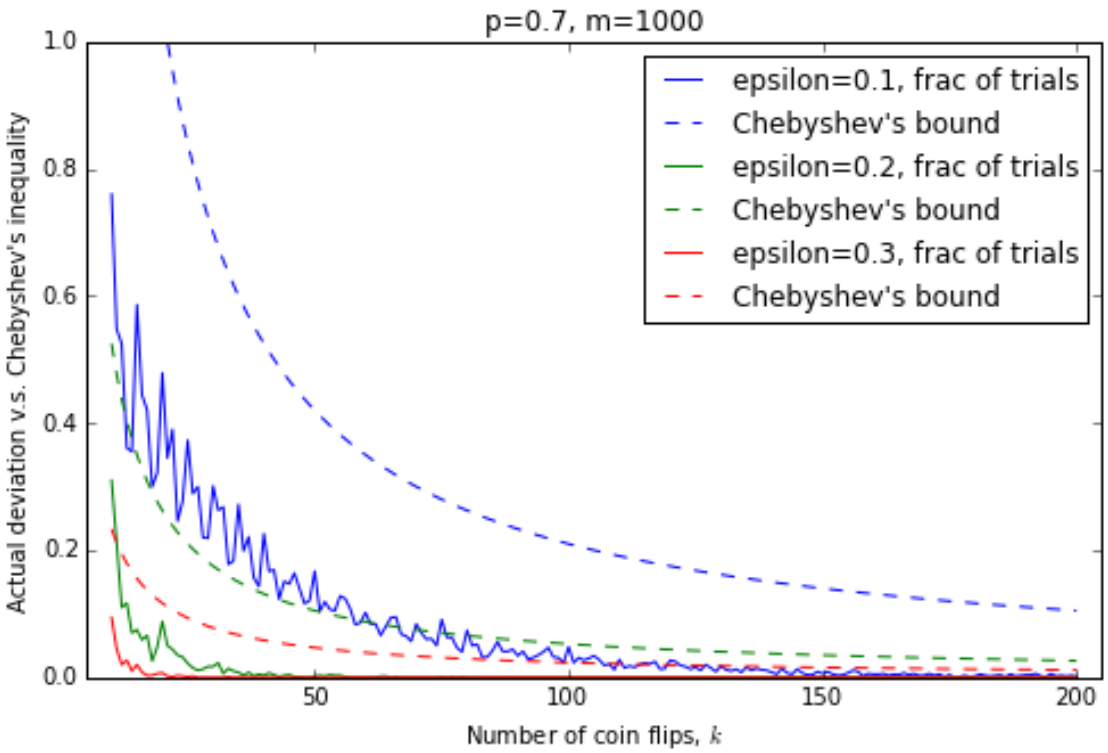
```
In [20]: def run_trial(p, k):
             """
             Runs a trial of k tosses of a biased coin (w.p. p of heads)
             and returns number of heads.
             """

             return sum([biased_coin(p) for _ in xrange(k)])
```

```
In [21]: def run_many_trials(p, k, m):
             """
             Runs m trials of k tosses of a biased coin (w.p. p of heads)
             and returns a list of the numbers of heads.
             """

             return [run_trial(p, k) for _ in xrange(m)]
```

```
In [22]: def partG(p=0.7, krange=xrange(10,201), erange=[0.1,0.2,0.3], m=1000):
             """
             Part (g) code
             """

             plt.figure(figsize=(8, 5))
             color_idx = 0

             all_trials = []
             for k in krange:
                 trials_k = run_many_trials(p, k, m)
                 all_trials.append(trials_k)

             for e in erange:

                 # Plots the fraction of trials where |S_k - kp| >= e*k
                 # Gives each line a different color to distinguish one from another, using `colors` (d
                 # YOUR CODE HERE
                 #################
                 frac_lst = []
                 for i in xrange(len(krange)):
                     k = krange[i]
                     trials_k = all_trials[i]
                     frac_close = [abs(Sk - k*p) >= e*k for Sk in trials_k].count(True) / m
                     frac_lst.append(frac_close)
                 plt.plot(krange, frac_lst, colors[color_idx],
                          label='epsilon=' + str(e) + ', frac of trials')
```

```
            ################

            # Overlays Chebyshev's bound
            # YOUR CODE HERE
            ################
            plt.plot(krange, [p*(1-p)/(k*e*e) for k in krange],
                     colors[color_idx] + '--', label='Chebyshev\'s bound')
            ################

            color_idx += 1

        # Sets up plot
        plt.legend()
        plt.axis([krange[0]-5, krange[-1]+5, 0, 1])
        plt.xlabel('Number of coin flips, $k$')
        plt.ylabel("Actual deviation v.s. Chebyshev's inequality")
        plt.title('p=' + str(p) + ', m=' + str(m))
        plt.show()
```

In [23]: partG()



YOUR COMMENTS HERE:

Part (h): Natural Language Processing

Natural Language Processing (NLP) is a field of computer science, artificial intelligence, and linguistics concerned with the interactions between computers and human (natural) languages. A solid understanding of probability is absolutely essential in doing research and solving NLP problems.

In this question, we will solve a simple NLP problem using conditional probability and the chain rule. According to Wikipedia, "a bigram is every sequence of two adjacent elements in a string of tokens, which

are typically letters, syllables, or words; they are n-grams for $n = 2$". For example, we have the following document:

Peter Piper picked a peck of pickled pepper.
Where's the pickled pepper that Peter Piper pickled?

Given a word, e.g. "Peter", we want to find the most likely word that will follow it. For example, the conditional probability of "Piper" given "Peter" is:

$$P(Piper|Peter) = \frac{|Peter, Piper|}{|Peter|} = \frac{2}{2} = 1$$

(the $|S|$ notation shows the cardinality, or size of the set S. In this context, it is the frequency of the words (or pair of consecutive words) in the original document.)

However, the conditional probability of "Piper" given "a" is:

$$P(Piper|a) = \frac{|a, Piper|}{|a|} = \frac{0}{1} = 0$$

Using conditional probabilities thus captures the fact that the likelihood of "Piper" varies by preceding context: it is more likely after "Peter" than after "a". In a bigram model, the context is the immediately preceding word, which is often known as the Markov property:

$$P(w_1 w_2 \ldots w_i) = P(w_1) \times P(w_2|w_1) \times \ldots \times P(w_i|w_{i-1})$$

Given the formula above, your task is to estimate the probability that a phrase appears in a document. We will give you a simplified corpus where every word has been lower-cased and punctuation are all removed, based on an 1865 novel written by Lewis Carroll, *Alice's Adventures in Wonderland* (http://www.gutenberg.org/ebooks/11).

```
In [3]: sample_doc = "Peter Piper picked a peck of pickled pepper. Where's the pickled pepper that Pete
        print sample_doc
```

```
Peter Piper picked a peck of pickled pepper. Where's the pickled pepper that Peter Piper pickled?
```

```
In [4]: def clean(doc):
            """
            Cleans up a text document by lowercasing, stripping newlines and whitespaces,
            and removing all punctuations

            Note: In practice, there are many better ways to do this. If you're interested in NLP,
            take a look at the powerful Natural Language Toolkit (NLTK) library.
            """

            return doc.lower().replace("\n", " ").strip().translate(string.maketrans("",""), string.pun
```

```
In [5]: cleaned_doc = clean(sample_doc)
        cleaned_doc
```

```
Out[5]: 'peter piper picked a peck of pickled pepper wheres the pickled pepper that peter piper pickled
```

In the above code block, we aggresively strip out all punctuations. In practice, a better approach is usually used, so that apostrophe like "where's" either does not get stripped out, or is converted into "where is".

Let us now compute the probabilities given the example above.

```
In [7]: p_piper_given_peter = cleaned_doc.count("peter piper") / cleaned_doc.count("peter")
        p_piper_given_peter
```

```
Out[7]: 1.0
```

```
In [8]: p_piper_given_a = cleaned_doc.count("a piper") / cleaned_doc.count("a")
        p_piper_given_a
```

```
Out[8]: 0.0
```

Let's first download our corpus based on Alice in Wonderland by Lewis Carroll http://www.gutenberg.org/cache/epub/11/pg11.txt. Run the commands below.

```
In [29]: !wget http://www-inst.eecs.berkeley.edu/~cs70/fa14/hw/alice.txt
```

```
--2014-12-05 22:45:54--  http://www-inst.eecs.berkeley.edu/~cs70/fa14/hw/alice.txt
Resolving www-inst.eecs.berkeley.edu (www-inst.eecs.berkeley.edu)... 128.32.42.199
Connecting to www-inst.eecs.berkeley.edu (www-inst.eecs.berkeley.edu)|128.32.42.199|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 144320 (141K) [text/plain]
Saving to: 'alice.txt.2'

100%[======================================>] 144,320      667KB/s   in 0.2s

2014-12-05 22:45:55 (667 KB/s) - 'alice.txt.2' saved [144320/144320]
```

```
In [9]: with open("alice.txt", "r") as f:
            corpus = clean(f.read())

        print corpus[:500]
```

```
chapter i down the rabbithole  alice was beginning to get very tired of sitting by her sister on the ba
```

Not the best way to clean up a document, but for this question's purpose, this is alright. :-)

```
In [10]: def calc_prob_preceding(w1, w2, doc):
             """
             Calculates the conditional probability that w2 appears given that w1 appears right before

             YOUR CODE HERE
             """

             return doc.count(w1 + " " + w2) / doc.count(w1)
```

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

```
In [11]: # this should be around 0.01, since there are many words after "her"
         p1 = calc_prob_preceding("her", "sister", corpus)
         print p1
         round(p1, 3) == 0.010
```

```
0.0104947526237
```

```
Out[11]: True
```

```
In [12]: # this should be very high (above 0.88), since March Hare appears quite often in the document
         # occasionally, there are other words after "march", like "march off" on line 1985.
         p2 = calc_prob_preceding("march", "hare", corpus)
         print p2
         round(p2, 3) == 0.886
```

```
0.885714285714
```

```
Out[12]: True
```

```
In [14]: def estimate_phrase_prob(phrase, doc):
             """
             Calculates the probability that phrase appears in doc

             YOUR CODE HERE
             """

             words = phrase.split()
             words_in_doc = doc.split()
             assert len(words) >= 2, "Phrase's length is less than 2"
             prob = doc.count(words[0]) / len(words_in_doc)  # first term in our formula
             for i in range(len(words) - 1):
                 prob *= calc_prob_preceding(words[i], words[i+1], corpus)
             return prob
```

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

```
In [15]: # This phrase appears once in the entire document,
         # so you can probably guess its probability is going to be very small, or almost zero
         p3 = estimate_phrase_prob(clean("You shan't be beheaded!"), corpus)
         print p3
         round(p3, 4) == 0
```

```
7.23535390721e-08
```

```
Out[15]: True
```

```
In [21]: # On the other hand, this phrase is a lot more popular (6 matches),
         # but the probability is still quite small, since our corpus is huge
         p4 = estimate_phrase_prob(clean("said the March Hare"), corpus)
         print p4
         round(p4, 5) == 0.00009
```

```
9.22584068128e-05
```

```
Out[21]: True
```

Unfortunately, the solution above is not quite correct. To see why, try to run the code below.

```
In [22]: "cathy's cat is cute".count("cat")
```

```
Out[22]: 2
```

And we technically only want 1 as the answer. To get rid of this overcounting, we can either split the string and then use the `count` method on list, or use regular expression (the solution below).

```
In [23]: import re

         count_occurrences = lambda word, doc: sum(1 for _ in re.finditer(r'\b%s\b' % re.escape(word), d
```

```
In [24]: def calc_prob_preceding2(w1, w2, doc):
             """
             Calculates the conditional probability that w2 appears given that w1 appears right before
             """

             return count_occurrences(w1 + " " + w2, doc) / count_occurrences(w1, doc)
```

```
In [28]: # this should be around 0.01, since there are many words after "her"
         p1 = calc_prob_preceding2("her", "sister", corpus)
         print p1
         round(p1, 3) == 0.028
```

```
0.0282258064516
```

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

```
In [30]: # this should be very high (above 0.88), since March Hare appears quite often in the document
         # occasionally, there are other words after "march", like "march off" on line 1985.
         p2 = calc_prob_preceding2("march", "hare", corpus)
         print p2
         round(p2, 3) == 0.939
```

```
0.939393939394
```

```
Out[30]: True
```

```
In [36]: def estimate_phrase_prob2(phrase, doc):
             """
             Calculates the probability that phrase appears in doc

             YOUR CODE HERE
             """

             words = phrase.split()
             words_in_doc = doc.split()
             assert len(words) >= 2, "Phrase's length is less than 2"
             prob = count_occurrences(words[0], doc) / len(words_in_doc)  # first term in our formula
             for i in range(len(words) - 1):
                 prob *= calc_prob_preceding2(words[i], words[i+1], corpus)
             return prob
```

```
In [37]: # This phrase appears once in the entire document,
         # so you can probably guess its probability is going to be very small, or almost zero
         p3 = estimate_phrase_prob2(clean("You shan't be beheaded!"), corpus)
         print p3
         round(p3, 4) == 0
```

```
2.59679825163e-07
```

```
Out[37]: True
```

```
In [39]: # On the other hand, this phrase is a lot more popular (6 matches),
         # but the probability is still quite small, since our corpus is huge
         p4 = estimate_phrase_prob2(clean("said the March Hare"), corpus)
         print p4
         round(p4, 5) == 0.00014
```

```
0.000137083529067
```

```
Out[39]: True
```

Hopefully, this gives you a slight taste of the powerful bigram model and Natural Language Processing, as well as how applicable probability is to almost every field in modern Computer Science. To find the most likely word to follow another word, we can use a concept known as Maximum Likelihood Estimate (MLE),

but it is out of the scope of this class. If you're interested in this material, please take EECS 126+189, STAT 135, and/or INFO 256. Once you have mastered the basic, definitely consider doing research and/or going to grad school/industry in NLP. It's a young and very promising/exciting field.

Feel free to ping Khoa if you want to chat more about this kind of stuff.

Part (i): Deviation Probability

This is Question 9, part (j). Fill in the two code blocks below marked with "YOUR CODE HERE".

```
In [37]: def getXi():
             """
             Our random variable X_i. 0, 1 or 2 independently with probability 1/3
             """

             return random.randint(0, 2)
```

```
In [38]: def getX(n):
             """
             Sums up n independent trials of X_i
             """

             return sum([getXi() for _ in xrange(n)])
```

```
In [39]: def get_many_X(n, k):
             """
             Runs getX() many times, return a list of results
             """

             return [getX(n) for _ in xrange(k)]
```

```
In [40]: def get_probability_X_large(n, k):
             """
             Checks how many of the X's from our trials were large.
             Returns the fraction that were. An estimate of the prob
             that X is 1.5 times larger than its mean.
             """

             x_trials = get_many_X(n, k)
             a = 1.5
             return (sum([float(i >= a*n) for i in x_trials]) / k)
```

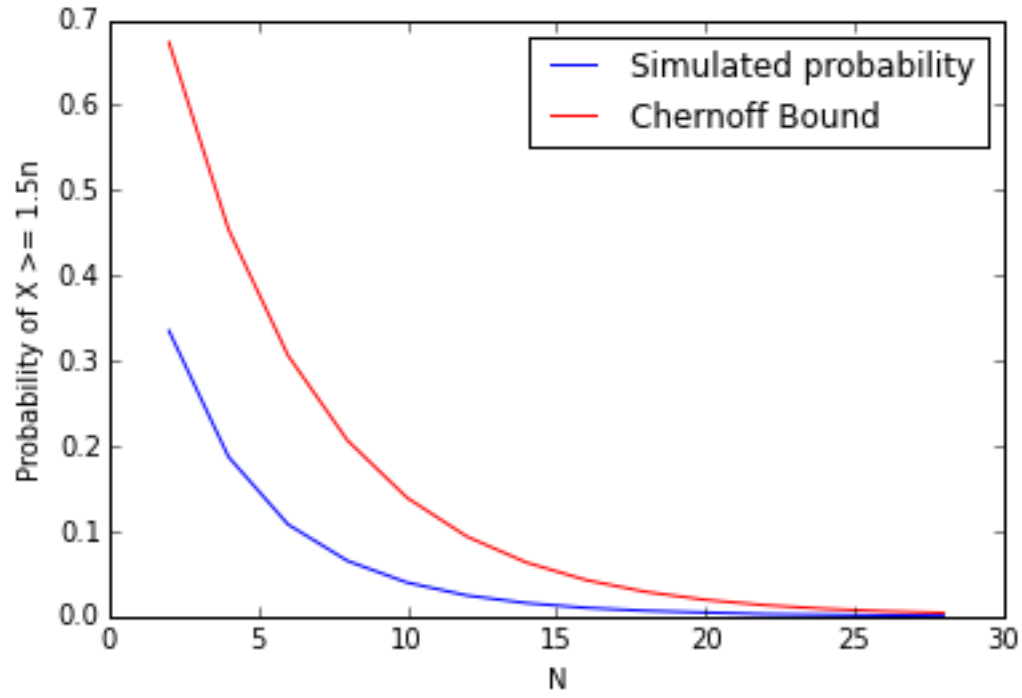
```
In [41]: def M(s):
             """
             The function M(s)= E[e^{sX_i}].
             Useful functions: math.exp

             YOUR CODE HERE
             """

             return 1/3*(1 + math.exp(s) + math.exp(2*s))
```

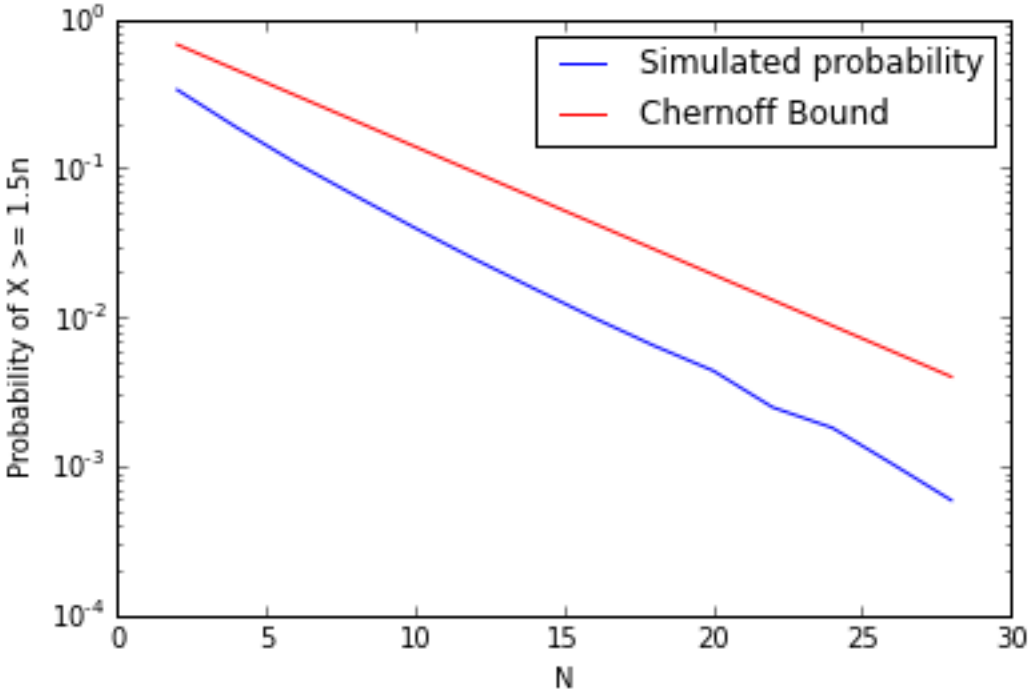
```
In [42]: def chernoff_bound(n):
             """
             A function that gives the value of the chernoff bound for a specific
             value of N, which is the number of samples that you are summing.

             You should use the value of s that maximizes sa-ln(M(s)), which you
```

```
        had to derive for a previous part.

        Useful functions: math.log, math.sqrt, math.exp, M(s)

        YOUR CODE HERE
        """

        a = 1.5
        s = math.log((-1*(1-a) + math.sqrt((1-a)**2 + 4*(2-a)*a)) / (2*(2-a)))
        return math.exp(-1*n*(s*a - math.log(M(s))))
```

In [43]:
```
def partI():
    """
    Code for part (i)
    """

    NUMREPS = 10000
    new_X_values = [2*i for i in xrange(1, 15)]

    # The probability of large X from simulation, for different values of N
    simulated_probs = [get_probability_X_large(i, NUMREPS) for i in new_X_values]

    # The prediction of the Chernoff bound, for different values of N
    chernoff_probs = [chernoff_bound(i) for i in new_X_values]

    # A plot on a linear scale
    plt.plot(new_X_values, simulated_probs, color='b', label='Simulated probability')
    plt.plot(new_X_values, chernoff_probs, color='r', label='Chernoff Bound')
    plt.xlabel("N")
    plt.ylabel("Probability of X >= 1.5n")
    plt.legend()
    plt.show()

    # A plot with the y-axis scaled logarithmically
    plt.figure()
    plt.semilogy(new_X_values, simulated_probs, color='b', label='Simulated probability')
    plt.semilogy(new_X_values, chernoff_probs, color='r', label='Chernoff Bound')
    plt.xlabel("N")
    plt.ylabel("Probability of X >= 1.5n")
    plt.legend()
    plt.show()
```

In [44]:
```
partI()
```





Extra: Central Limit Theorem

(This was originally a question, but we've decided to convert it to a tutorial. Please read everything below to fully appreciate the Central Limit Theorem.)

Take the mean of n random samples from ANY arbitrary distribution with a well defined standard deviation σ and mean μ. As n gets bigger the distribution of the sample mean will always converge to a Gaussian (normal) distribution with mean μ and standard deviation $\sigma/\sqrt{n}$.

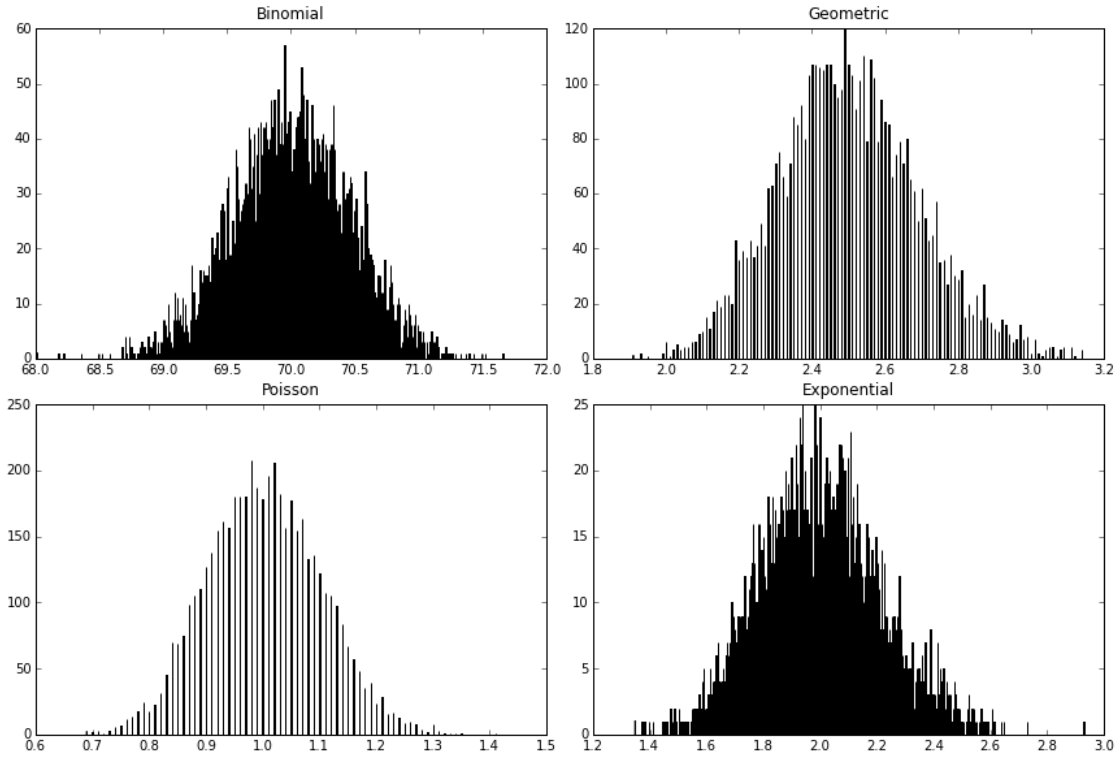
Colloquially speaking, the theorem states that the average (or sum) of a set of random measurements will tend to a bell-shaped curve no matter the shape of the original measurement distribution. This explains the ubiquity of the Gaussian distribution, which arises so commonly in science and statistics, and why it is generally an excellent approximation for the mean of a collection of data.

Below, we will demonstrate the Central Limit Theorem by sampling from 4 different distributions: binomial, geometric, poisson, and exponential. We perform 5000 trials, and for each trial, let's generate 100 samples from each distribution and compute the mean value. We then plot the distribution of the sample mean for each of the four distributions. Let's see what we have!

```
In [45]: def clt(num_samples=100, num_trials=5000):
             """
             CLT Tutorial code
             """

             # Initializes parameters for the distributions
             # Feel free to change and play around with them
             n, mu = 100, 0.7      # for binomial
             p = 0.4               # for geometric
             lam = 1.0             # for poisson
             scale = 2.0           # for exponential

             # Empty lists to hold the mean values of each trial
             samples_binomial = []
             samples_geometric = []
             samples_poisson = []
             samples_exponential = []

             for _ in range(num_trials):
                 # Appends the mean of num_samples random samples from each distribution into its corre
                 # Binomial Samples
                 binom_sample = binom.rvs(n, mu, size=num_samples)
                 samples_binomial.append(binom_sample.mean())

                 # Geometric Samples
                 geom_sample = geom.rvs(p=p, size=num_samples)
                 samples_geometric.append(geom_sample.mean())

                 # Poisson Samples
                 pois_sample = poisson.rvs(mu=lam, size=num_samples)
                 samples_poisson.append(pois_sample.mean())

                 # Exponential Samples
                 exp_sample = expon.rvs(scale=scale, size=num_samples)
                 samples_exponential.append(exp_sample.mean())

             # Plot the subplots in a 2x2 plot
             fig, axes = plt.subplots(nrows=2, ncols=2)
             fig.set_figwidth(12)
             fig.set_figheight(8)
             fig.tight_layout(pad=0.8, w_pad=1.0, h_pad=1.8)
```

```
    # Binomial
    axes[0,0].set_title('Binomial', y=1.01)
    axes[0,0].hist(samples_binomial, bins=1000)

    # Geometric
    axes[0,1].set_title('Geometric', y=1.01)
    axes[0,1].hist(samples_geometric, bins=1000)

    # Poisson
    axes[1,0].set_title('Poisson', y=1.01)
    axes[1,0].hist(samples_poisson, bins=1000)

    # Exponential
    axes[1,1].set_title('Exponential', y=1.01)
    axes[1,1].hist(samples_exponential, bins=1000)

    plt.show()
```

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



Congratulations! You are done with Virtual Lab 14.

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.