lab11sol

November 14, 2014

1 Virtual Lab 11 Solution: Biased Coins, Birthday Paradox, and Stirling's Approximation

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

Due Date: Monday, November 17th, 2014 at 12pm **Name**: EECS 70 TA
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.

Please note that there is no credit for doing part (k) of the VL. It is supposed to help you with part (j) of Question 8. Again, part (k) of the VL is **NOT** extra credit.

```
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
        import scipy.integrate
        import scipy.special
```

Introduction

Up until this point, everything that you have done in the last three virtual labs is something that you could've naturally discovered yourself as something worth trying. The data is speaking directly to the experimentalist in you. However, discovering an actual formula for the shape of this "cliff-face" is something that actually requires a theoretical investigation that is related to counting, Fourier Transforms, and Power Series. Guessing its exact shape is not something that comes very naturally on experimentalist intuition alone.

In this week's lab, we will simply provide you with the right curve and continue from last week's lab on biased coins. Unless specified otherwise, you can assume the same configurations from last week's lab. In other words, the coin is biased with $P(\text{head}) = 0.7$, the number of tosses are $(k = 10, 100, 1000, 4000)$, respectively, and the number of trials is $m = 1000$. Make sure you review the lab solution from Homework 10 before moving on.

In addition, we will also look at the Birthday Paradox and Stirling's Approximation. Please come back to the last three parts of the lab when you are working on Question 9.

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.

Below, you will find sample implementations of some functions we implemented in last week's lab.

```
In [3]: def biased_coin(p):
            """
            Creates a biased coin with p(Head) = p
            Returns True if heads and False otherwise.
            """

            assert p >= 0 and p <= 1, "Wrong biased coin probability"
            return random.random() <= p

In [4]: 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 [5]: 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 [6]: 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): Normal cdf v.s. Empirical cdf

Plot $\int_{-\infty}^{d} \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} dx$ overlaid with the normalized cliff-face shapes you had plotted in last week's lab. (This integral is related to something called the Error Function.) What do you observe?

This is the heart of the Central Limit Theorem as applied to coin tosses.

Hint: Implement the function `normal`, which takes a real number x and returns $\frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$. Then, implement the function `integrate_normal(d)`, which integrates the above function from $-\infty$ to d. In Python, you can use `scipy.integrate.quad`.

```
In [7]: def normal(x):
            """
            Normal distribution

            YOUR CODE HERE
            """

            return 1/math.sqrt(2*math.pi) * math.exp(-x*x/2)
```

Here's an example of how to calculate an integral using `scipy.integrate.quad`. We will calculate $\int_0^4 x^2 dx$.

For more details, check out the documentation, or simply execute `help(scipy.integrate.quad)`.

```
In [8]: f = lambda x: x**2
        print scipy.integrate.quad(f, 0, 4)[0]

        # Analytic result (which you learned in Math 1A)
        print 4**3 / 3
```

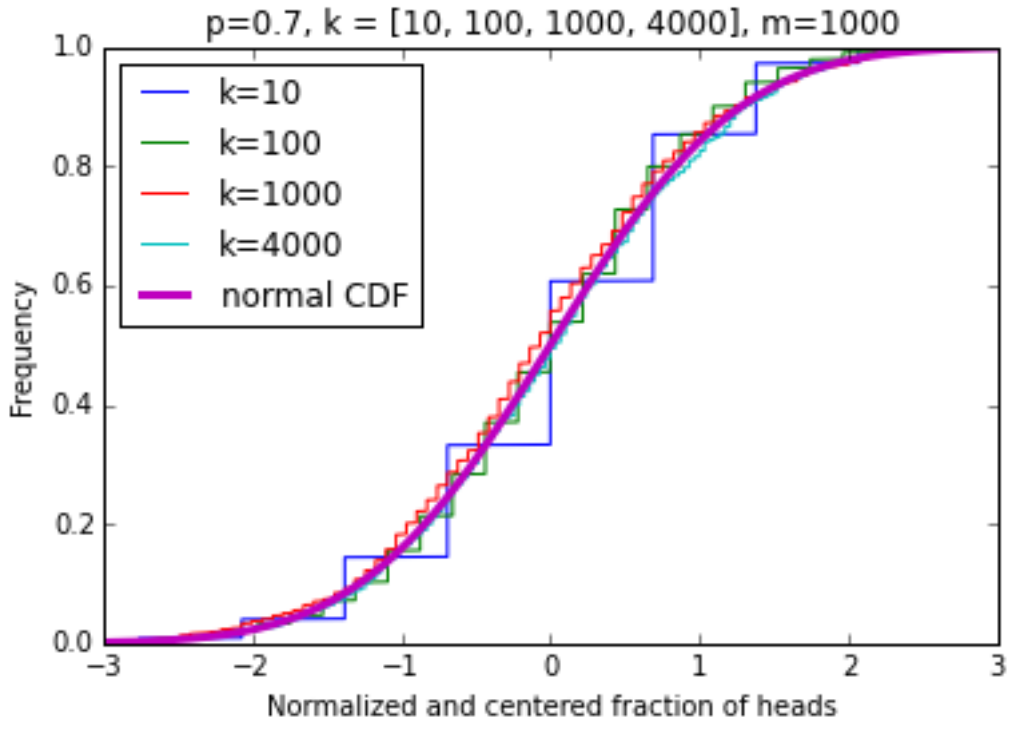
```
21.3333333333
21.3333333333
```

```
In [9]: def integrate_normal(d):
            """
            Integrates normal distribution

            Hint: negative infinity in Python is -float('inf')

            YOUR CODE HERE
            """

            return scipy.integrate.quad(lambda x: normal(x), -float('inf'), d)[0]
```

```
In [10]: def partA(p=0.7, krange=[10,100,1000,4000], m=1000):
             """
             Part (a) code
             """

             # Plots empirical cdf -- this is from HW10's part (k)
```

```
    std = math.sqrt(p*(1-p))
    for k in krange:
        results = {}
        results[k] = [(Sk-k*p)/(math.sqrt(k)*std) for Sk in run_many_trials(p, k, m)]
        results[k].sort()
        plt.plot(results[k], [float(y)/m for y in xrange(1, m+1)], label='k=' + str(k))

    # We want to cover the range [-min_max*num_points, min_max*num_points)
    num_points = 1000
    min_max = 3

    # Overlays normal cdf
    # YOUR CODE HERE
    x_values = [float(x)/num_points for x in xrange(-min_max*num_points, min_max*num_points)]
    plt.plot(x_values, [integrate_normal(x) for x in x_values], lw=3, label='normal CDF')
    # END YOUR CODE HERE

    # Sets up plot
    plt.legend(loc=2)
    plt.xlabel('Normalized and centered fraction of heads')
    plt.ylabel('Frequency')
    plt.xlim(-min_max, min_max)
    plt.title('p=' + str(p) + ', k = ' + str(krange) + ', m=' + str(m))
    plt.show()
```

In [11]:
```
# This plot should take a while (20 seconds or more) to generate. Be patient.

partA()
```



YOUR COMMENTS HERE:

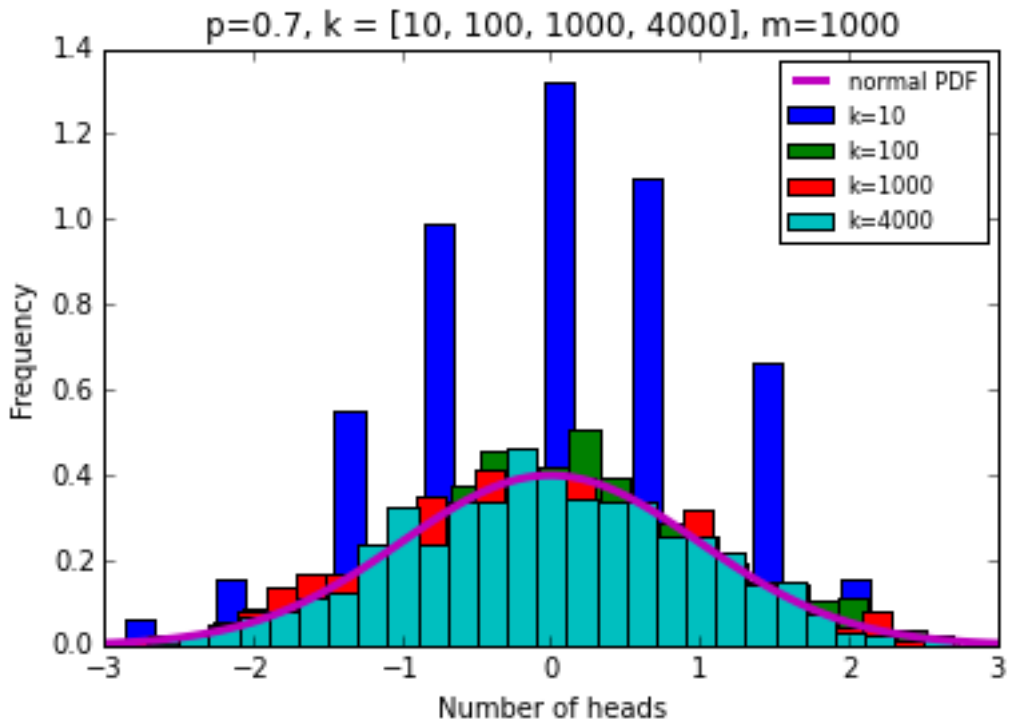
Part (b): Overlay

Now, since you had realized in the previous lab that the cliff-faces and the histograms have some natural relationship with each other, see if you can figure out a way to naturally overlay a smooth plot of $\frac{1}{\sqrt{2\pi}}e^{-\frac{x^2}{2}}$ to the normalized histograms. What does this mean?

Hint: There's a parameter for `plt.hist()` you learned in HW7 that you can use to normalize the histogram.

In [18]:
```
def partB(p=0.7, krange=[10,100,1000,4000], m=1000):
    """
    Part (b) code
    """

    bin_width = 0.2

    # Plots normalized histograms
    # YOUR CODE HERE
    std = math.sqrt(p*(1-p))
    for k in krange:
        results = [(Sk-k*p)/(math.sqrt(k)*std) for Sk in run_many_trials(p, k, m)]
        plt.hist(results, bins=np.arange(min(results), max(results)+bin_width*2, bin_width),
                 align='mid', normed=True, label='k=' + str(k))
    # END YOUR CODE HERE

    # Overlays normal distribution
    # YOUR CODE HERE
    x_values = np.linspace(-3,3,100)
    plt.plot(x_values, [normal(x) for x in x_values], lw=3, label='normal PDF')
    # END YOUR CODE HERE

    # Sets up plot
    plt.legend(fontsize='small')
    plt.xlabel('Number of heads')
    plt.ylabel('Frequency')
    min_max = 3
    plt.xlim(-min_max, min_max)
    plt.title('p=' + str(p) + ', k = ' + str(krange) + ', m=' + str(m))
    plt.show()
```

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



YOUR COMMENTS HERE:

Part (c): Kullback-Liebler Divergence

Another interesting pattern that you had seen in the previous Virtual Labs was the exponential drop in the frequencies of certain rare events. For an exponential drop, the most interesting thing is to understand the rate of the exponential — or the relevant slope on the Log-Linear plot.

For a coin with probability p of being heads, we are interested in the frequency by which tossing k such coins results in more than ak heads (where a is a number larger than p). We are interested in $p = 0.3, 0.7$ and $a = p + 0.05, p + 0.1$. Take $m = 1000$ and plot the natural log of the frequencies these deviations against k (ranging from 10 to 200). Approximately extract the slopes for all four of these.

Compare them in a table against the predictions of the following formula (which we will derive later in the course)

$$D(a||p) = a \ln \frac{a}{p} + (1-a) \ln \frac{1-a}{1-p}.$$

This expression is called the Kullback-Leibler divergence and is also called the relative entropy.

Finally, add $e^{-D(a||p)k}$ to the plots (there should be 4 of these) you have made as straight lines for immediate visual comparison. This straight line is called a "Chernoff Bound" on the probability in question.

What do you observe?

Hint: First, implement the function `KL`, which computes $D(a||p)$ using the given formula. To fit a line in Python, you can use `np.polyfit`. There will probably be some 0 values, which will mess up this fitting, so you can replace the zeros with 10^{-3}.

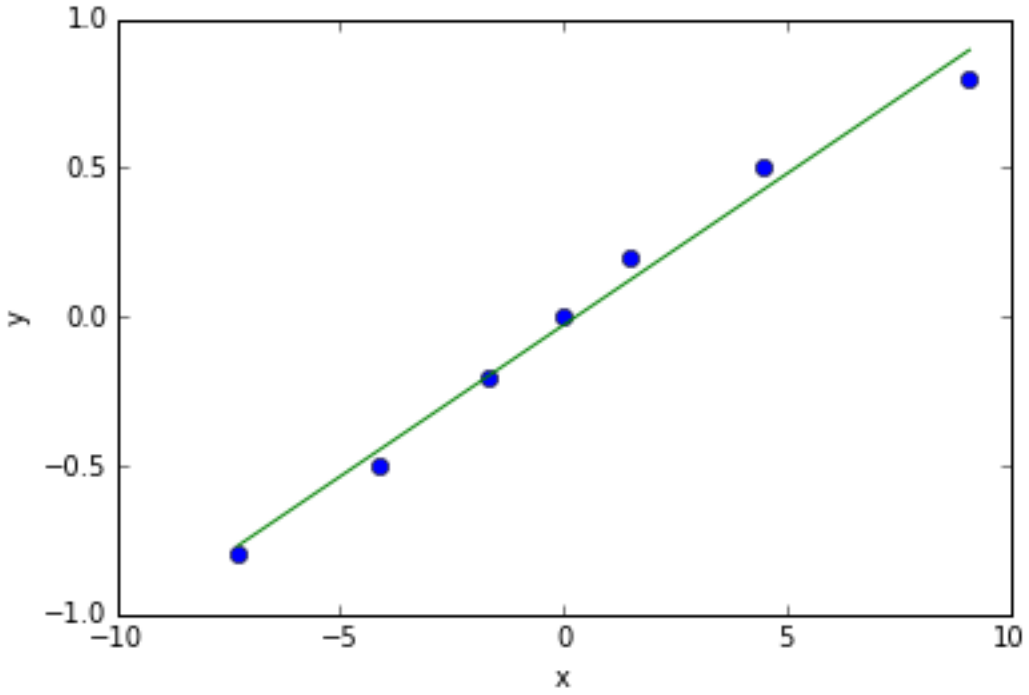
This is a very challenging question. If you can't figure it out, continue on with the rest of the lab and come back to this one later.

```
In [14]: def KL(a, p):
             """
             Computes KL divergence

             YOUR CODE HERE
```

```
        """

        return a*math.log(a/p) + (1-a)*math.log((1-a)/(1-p))
```

Here's an example of how to perform a linear fit in Python. This is releated to a concept known as "linear least square", which we will study later in the course. Interested students are encouraged to take Stat 135, 151A, and/or CS 189 to see the powerful mathematics behind least square analysis.

```
In [15]: x = [-7.30000, -4.10000, -1.70000, -0.02564,
              1.50000, 4.50000, 9.10000]
         y = [-0.80000, -0.50000, -0.20000, 0.00000,
              0.20000, 0.50000, 0.80000]

         coefficients = np.polyfit(x, y, 1)  # notice that np.polyfit returns coefficients
         polynomial = np.poly1d(coefficients)
         ys = polynomial(x)

         plt.plot(x, y, 'o')
         plt.plot(x, ys)
         plt.ylabel('y')
         plt.xlabel('x')
         plt.xlim(-10,10)
         plt.ylim(-1,1)
         plt.show()
```

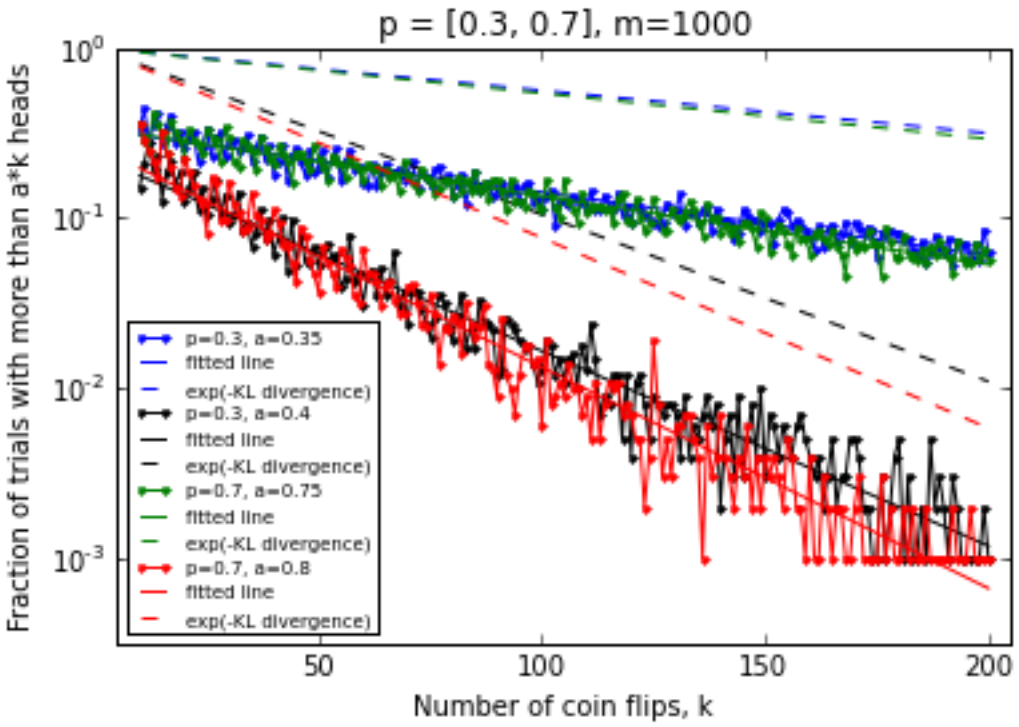



```
In [16]: def partC(prange=[0.3, 0.7], krange=xrange(10, 201), m=1000):
             """
             Part (c) code
```

```
    """

    log_thresh = 1e-3
    erange = [0.05, 0.1]
    COLORS = ['b', 'k', 'g', 'r']
    color_idx = 0

    for p in prange:
        all_trials = []
        for k in krange:
            trials_k = run_many_trials(p, k, m)
            all_trials.append(trials_k)
        arange = [p+e for e in erange]
        for a in arange:

            # Plots fraction of heads above ak heads
            # YOUR CODE HERE
            frac_lst = []
            for i in range(len(krange)):
                k = krange[i]
                frac_above = [Sk > a*k for Sk in all_trials[i]].count(True) / m
                frac_lst.append(max(frac_above, log_thresh))
            lbl = 'p=' + str(p) + ', a=' + str(a)
            plt.plot(krange, frac_lst, COLORS[color_idx] + '.-', label=lbl)
            # END YOUR CODE HERE

            # Fits a line to each set of points
            # YOUR CODE HERE
            ls_line = np.polyfit(krange, [math.log(f) for f in frac_lst], 1)
            lbl = 'fitted line'
            plt.plot(krange, [math.exp(ls_line[0]*k + ls_line[1]) for k in krange],
                     COLORS[color_idx], label=lbl)
            # END YOUR CODE HERE

            # Overlays KL divergence
            # YOUR CODE HERE
            lbl = 'exp(-KL divergence)'
            plt.plot(krange, [math.exp(-KL(a,p)*k) for k in krange],
                     COLORS[color_idx] + '--', label=lbl)
            # END YOUR CODE HERE

            color_idx += 1

    # Sets up plot
    plt.xlabel('Number of coin flips, k')
    plt.ylabel('Fraction of trials with more than a*k heads')
    plt.title('p = ' + str(prange) + ', m=' + str(m))
    plt.yscale('symlog', linthreshy=log_thresh*math.sqrt(1e-1))
    plt.axis([krange[0]-5, krange[-1]+5, log_thresh*math.sqrt(1e-1),1])
    plt.legend(loc=3, fontsize='x-small')
    plt.show()
```

In [17]:
```
# This plot should take a while (30 seconds or more) to generate. Be patient.
# If you want to know what your plot should look like, turn to page 9 of Note 19
```

```
partC()
```



Fitted slopes v.s. Kullback-Liebler divergence table. Put your answers below. (We use $D(a, p)$ instead of $D(a||p)$ in the table below.)

p	a	fitted slope	D(a,p)

YOUR COMMENTS HERE:

Part (d): Same Birthday

During your first week of Charm School (CS), you want to find fellow CS students who have the same birthday. Let's switch gears to an interesting problem studied in Lecture Note 12: the Birthday Paradox. This interesting phenomenon concerns the probability of two people in a group of m people having the same birthdays. This probability is given by

$$P(A^c) = 1 - \frac{365 \times 364 \times \ldots \times (365 - m + 1)}{365^m} = 1 - \frac{365!}{(365 - m)!365^m}$$

where

$$P(A) = \frac{365!}{(365 - m)!365^m} = \left(1 - \frac{1}{365}\right) \times \left(1 - \frac{2}{365}\right) \times \ldots \times \left(1 - \frac{m - 1}{365}\right)$$

and $P(A)$ is the probability that no two people have the same birthdays.

For $m = 10, 23, 50, 60$, randomly generate birthdays by uniformly picking m numbers between 1 and 365. Do this 1000 times for each value of m. Record how many trials have at least 2 same birthdays. Plot this fraction vs. m using a bar chart. What do you observe?

Hint: First, implement the function `has_duplicate`, which returns True if a list contains any repeated element and False otherwise. Then, implement the function `gen_birthday(m)`, which generates random birthday for m people.

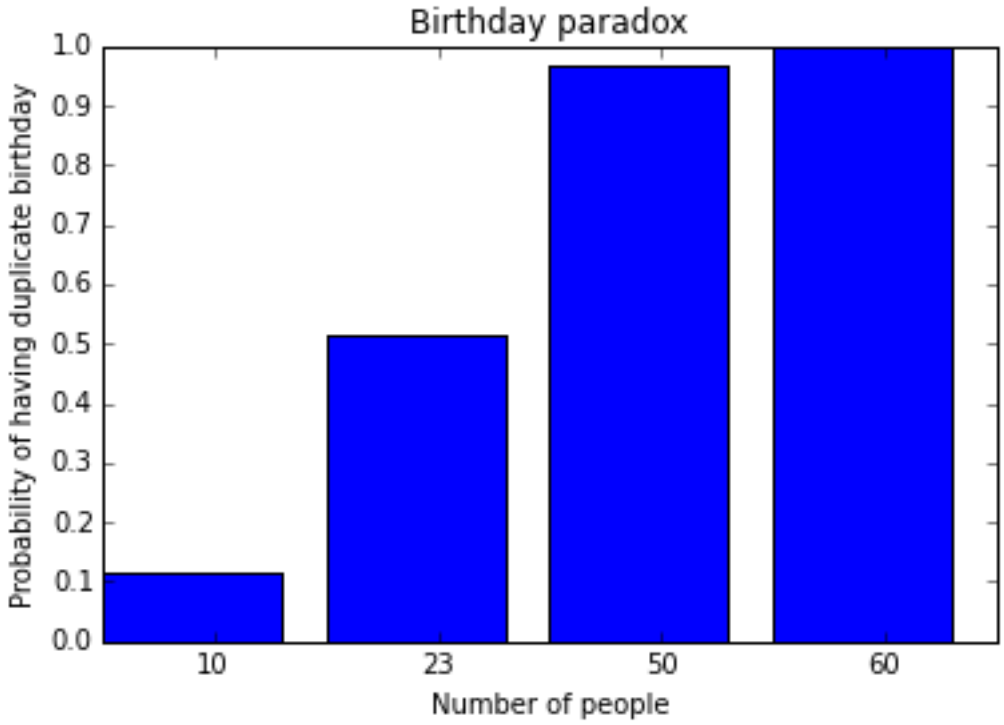
```
In [18]: def has_duplicate(lst):
             """
             Returns True if lst contains any duplicate element, False otherwise

             YOUR CODE HERE
             """

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

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

```
In [19]: has_duplicate([3, 4, 5, 6, 3]) == True  # 3 is repeated

Out[19]: True

In [20]: has_duplicate([3, 4, 5, 6, 7]) == False

Out[20]: True

In [21]: def gen_birthday(m):
             """
             Generates random birthday for m people.

             Returns a list of length n, where each element is an
             integer in the range [1, 365].

             Don't worry about leap year for this problem.

             YOUR CODE HERE
             """

             return [random.randint(1, 365) for _ in range(m)]

In [22]: def partD(num_people_lst=[10,23,50,60], num_trials=1000):
             """
             Code for part (d)
             """

             # Retrieves the probability of having at least two people
             # with the same birthday for each value of n in num_people_lst
             # YOUR CODE HERE
             prob_duplicate_lst = []
             for m in num_people_lst:
                 counter = 0
                 for _ in range(num_trials):
                     birthdays = gen_birthday(m)
                     if has_duplicate(birthdays):
                         counter += 1
```

```
            prob_duplicate_lst.append(counter / num_trials)
    # END YOUR CODE HERE

    # Plots the result
    # YOUR CODE HERE
    ticks = range(len(num_people_lst)) # for plotting
    plt.bar(ticks, prob_duplicate_lst)
    plt.xticks([tck + 0.5 for tck in ticks], num_people_lst)
    # END YOUR CODE HERE

    # Other configurations. No need to change anything below.
    plt.xlabel('Number of people')
    plt.yticks(np.arange(11)/10)
    plt.ylabel('Probability of having duplicate birthday')
    plt.title('Birthday paradox')
    plt.show()
```

```
In [23]: partD()
```



YOUR COMMENTS HERE:

Part (e): Birthday Paradox

We will now calculate the probability of having two people with the same birthday empirically, and plot the result against the expected probability, which is derived in Note 12.

Implement the function `birthday_formula(n)`, which calculates the probability of at least two people having the same birthday among m people.

Plot the empirical result (you can assume 1000 trials) v.s. the analytical result `birthday_formula(m)`), for $m = [1, 100]$ people. What do you observe? What happens at $m = 23$ and $m = 60$? Is this consonant with what we previously knew about the Birthday Paradox?

```
In [24]: def birthday_formula(m):
             """
             Calculates the probability of at least two people having the same birthday
             among m people. Use the formula at the top of page 4 of Note 12.

             Please only use math.factorial(), ** (raise to a power), and elementary arithmetic
             operations for this problem.

             YOUR CODE HERE
             """

             return (1 - math.factorial(365) / (365**m * math.factorial(365-m)))
```

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

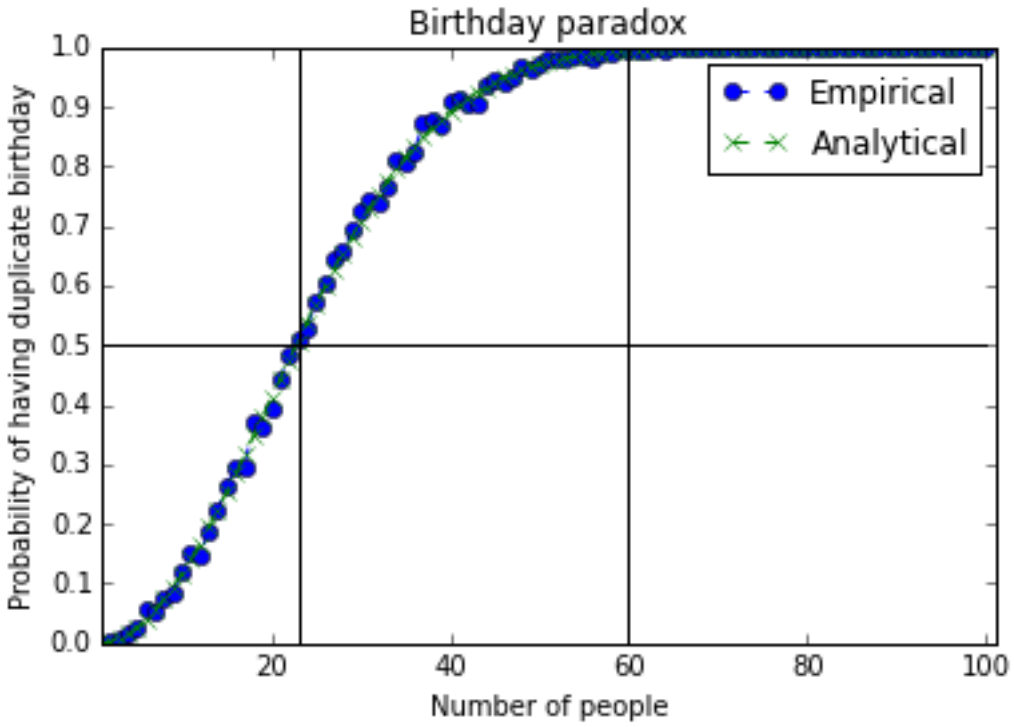
```
In [25]: birthday_formula(23) > 0.5

Out[25]: True

In [26]: birthday_formula(60) > 0.99

Out[26]: True
```

Now it's time to plot the above result!

```
In [27]: def partE(max_num_people=100, num_trials=1000):
             """
             Code for part (e)
             """

             num_people_lst = xrange(1, max_num_people+1)

             # Calculates the empirical and analytical probabilities
             # YOUR CODE HERE
             prob_duplicate_lst = []
             birthday_formula_lst = []

             for m in num_people_lst:
                 counter = 0
                 for _ in range(num_trials):
                     birthdays = gen_birthday(m)
                     if has_duplicate(birthdays):
                         counter += 1

                 # Appends the values
                 prob_duplicate_lst.append(counter / num_trials)
                 birthday_formula_lst.append(birthday_formula(m))
             # END YOUR CODE HERE

             # Plots the analytical vs empirical result
             # YOUR CODE HERE
             plt.plot(num_people_lst, prob_duplicate_lst, 'o--', label="Empirical")
             plt.plot(num_people_lst, birthday_formula_lst, 'x--', label="Analytical")
             # END YOUR CODE HERE

             # Plots the two vertical lines at x = 23 and x = 60, and a horizontal line at y = 0.5
```

```
    plt.plot([23 for _ in range(1, max_num_people+1)], np.linspace(0, 1, max_num_people), colo
    plt.plot([60 for _ in range(1, max_num_people+1)], np.linspace(0, 1, max_num_people), colo
    plt.plot(num_people_lst, [0.5 for _ in range(max_num_people)], color='black')

    # Other configurations. No need to change anything.
    plt.xlim(1, max_num_people+1)
    plt.ylim(0, 1)
    plt.yticks(np.arange(11)/10)
    plt.title("Birthday paradox")
    plt.ylabel("Probability of having duplicate birthday")
    plt.xlabel("Number of people")
    plt.legend(loc=1)
    plt.show()
```

In [28]:
```
partE()
```



YOUR COMMENTS HERE:

Part (f): Stirling's Approximation to the Birthday Paradox

Now approximate $P(A)$ using Stirling's approximation for $n!$ and plot the approximated $P(A^c) = 1 - P(A)$ as a function of m. Stirling's approximation is given by

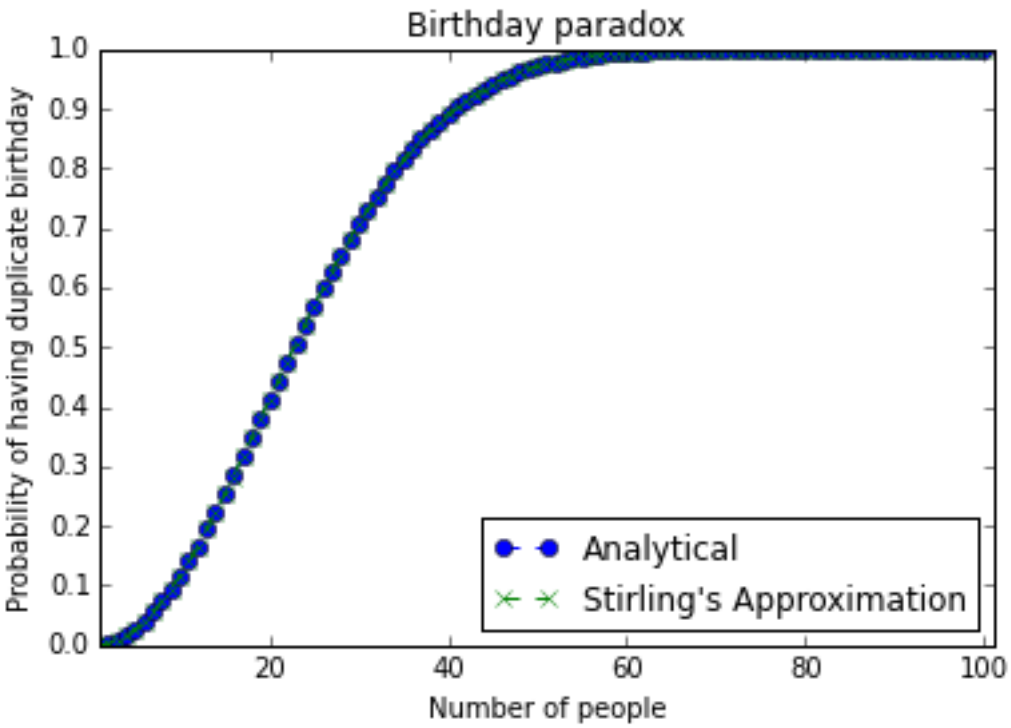
$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n.$$

Plot the analytical result from the previous part and the approximated result in the same figure. What do you observe?

Hint: Implement the function `birthday_stirling`, which computes the probability that no two people have the same birthday given that there are m birthdays using Stirling's approximation.

That said, don't use Stirling's approximation directly! Simplify your expression after using the approximation as much as possible before you implement the `birthday_stirling` function, or the large values will blow up your computer.

```
In [29]: def birthday_stirling(m):
             """
             Probability that no two people have the same birthday
             given that there are m birthdays.

             Uses Stirling's approximation and simplifies P(A) first as much as possible

             YOUR CODE HERE
             """

             return math.sqrt(365.0/(365.0-m))*math.pow(1.0-m/365.0, m-365)*math.pow(math.e,-m)

In [30]: def partF(max_num_people=100):
             """
             Code for part (f)
             """

             # Calculates the analytical and Stirling approximated probabilities
             # YOUR CODE HERE
             birthday_formula_lst = []
             birthday_stirling_lst = []
             num_people_lst = xrange(1, max_num_people+1)

             for m in num_people_lst:
                 birthday_formula_lst.append(birthday_formula(m))
                 birthday_stirling_lst.append(1-birthday_stirling(m))
             # END YOUR CODE HERE

             # Plots the two curves
             # YOUR CODE HERE
             plt.plot(num_people_lst, birthday_formula_lst, 'o--', label="Analytical")
             plt.plot(num_people_lst, birthday_stirling_lst, 'x--', label="Stirling's Approximation")
             # END YOUR CODE HERE

             # Other configurations. No need to change anything.
             plt.xlim(1, max_num_people+1)
             plt.ylim(0, 1)
             plt.yticks(np.arange(11)/10)
             plt.title("Birthday paradox")
             plt.ylabel("Probability of having duplicate birthday")
             plt.xlabel("Number of people")
             plt.legend(loc=4)
             plt.show()

In [31]: partF()
```



YOUR COMMENTS HERE:

Part (g): Generate Permutations

Lastly, let's come back to the problem of counting the number of ways to throw m balls into n bins. Suppose the number of balls in each bin is a nonnegative integer, implement the function `permutation(m,n)`, which generates all possible permutations of throwing m balls into n bins. For example, `permutation(2,3)` should return `[[0,0,2],[0,1,1],[0,2,0],[1,0,1],[1,1,0],[2,0,0]]`.

How would you change your implementation if we now require each bin to contain a positive number of balls?

Hint: Use recursion. You should have three base cases.

```
In [1]: def permutation(m, n):
            """
            Generates all permutations for throwing m balls into n bins,
            where each bin contains a nonnegative number of balls.

            YOUR CODE HERE
            """

            assert m >= 0 and n >= 0, "Incorrect parameters"

            if m == 0:    # 0 balls, all bins are empty
                return [[0]*n]
            elif n == 0:  # 0 bin, doesn't matter how many balls you have
                return []
            elif n == 1:  # 1 bin, then that bin has all the m balls
                return [[m]]

            # We either put 0 ball in the first bin (essentially remove that bin)
```

```
            # or put 1 ball in the first bin, and count the number of ways to
            # throw m-1 balls into n bins
            return [[0]+val for val in permutation(m, n-1)] + \
                   [[val[0]+1] + val[1:] for val in permutation(m-1, n)]
```

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

```
In [2]: sorted(permutation(3, 0)) == []  # zero bin...
```

```
Out[2]: True
```

```
In [3]: sorted(permutation(0, 3)) == [[0, 0, 0]]  # all bins are empty
```

```
Out[3]: True
```

```
In [4]: sorted(permutation(3, 1)) == [[3]]  # 1 bin has all balls
```

```
Out[4]: True
```

```
In [5]: sorted(permutation(2, 3)) == [[0, 0, 2], [0, 1, 1], [0, 2, 0], [1, 0, 1], [1, 1, 0], [2, 0, 0]]
```

```
Out[5]: True
```

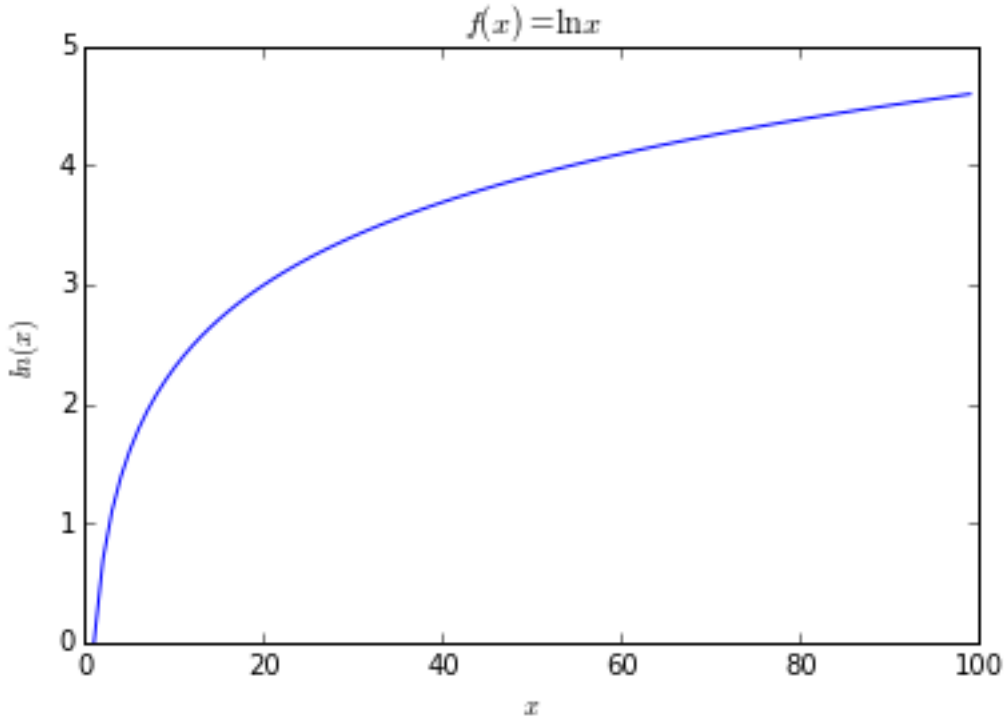
```
In [10]: len(permutation(20, 6)) == choose(25, 5)  # this is the first part of Discussion 10M Q3
```

```
Out[10]: True
```

YOUR COMMENTS HERE:
Part (h): Plot $ln(x)$
Plot the function $f(x) = \ln x$.

```
In [34]: def partH():
             """
             Part (h) code

             YOUR CODE HERE
             """

             plt.plot(xrange(1, 100), [math.log(i) for i in xrange(1, 100)])
             plt.xlabel("$x$")
             plt.ylabel("$ln(x)$")
             plt.title("$f(x)=\ln x$")
             plt.show()
```

```
In [35]: partH()
```



Part (i): Plot Stirling's Approximation

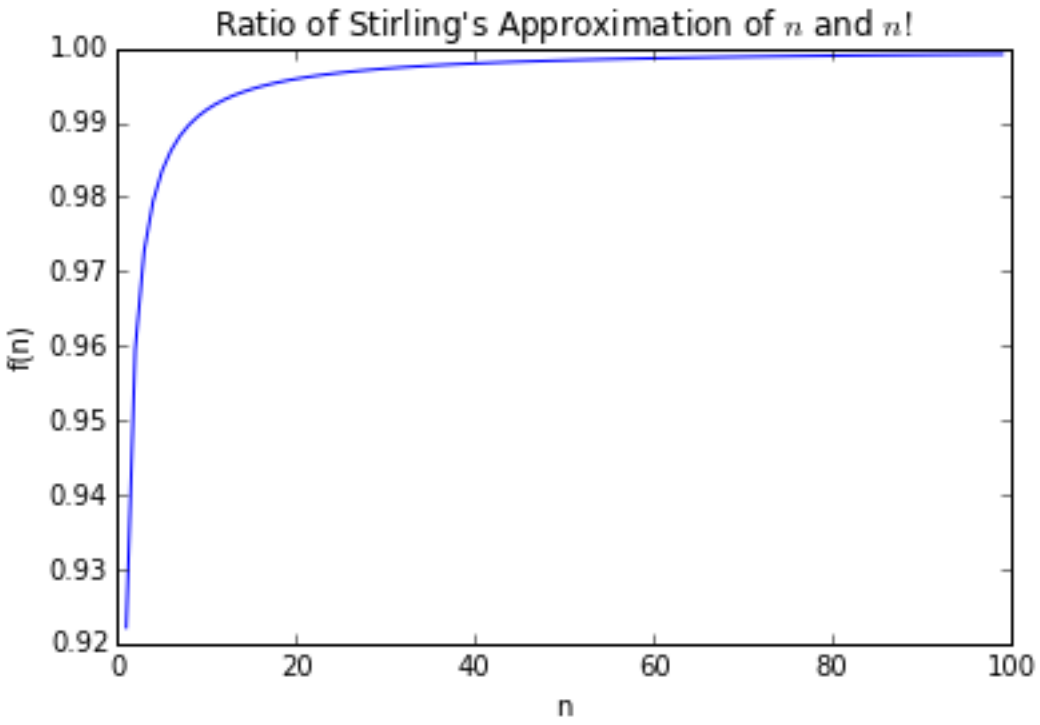
The Stirling's approximation is usually written as $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ or a simpler version $n! \approx \left(\frac{n}{e}\right)^n$. Plot the function $f(n) = \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{n!}$. What do you observe?

```
In [41]: def stirling(n):
             """
             Returns Stirling's Approximation to n!

             You probably will find math.sqrt(x), math.pi, math.e, and math.pow(x, p) useful

             YOUR CODE HERE
             """

             return math.sqrt(2*math.pi * n) * math.pow((n/math.e), n)
```

```
In [50]: def partI():
             """
             Part (i) code

             YOUR CODE HERE
             """

             def f(n):
                return stirling(n) / math.factorial(n)

             plt.plot(xrange(1, 100), [f(i) for i in xrange(1, 100)])
             plt.xlabel("n")
             plt.ylabel("f(n)")
```

```
        plt.title("Ratio of Stirling's Approximation of $n$ and $n!$")
        plt.show()
```

```
In [51]: partI()
```



YOUR COMMENTS HERE:

Part (j): Plot $ln(\binom{n}{k})$

Now, suppose $m_1 = \frac{k_1}{n} = 0.25$, $m_2 = \frac{k_2}{n} = 0.5$, and $m_3 = \frac{k_3}{n} = 0.75$, plot $\ln(\binom{n}{k_1})$, $\ln(\binom{n}{k_2})$, and $\ln(\binom{n}{k_3})$ as functions of n on a plot with linear-scaled axes. What do you observe?

Hint: Define $j_i(n)$ such that $j_i(n) = ln(\binom{n}{k_i})$. Implement the three functions below, which calculate each $ln(\binom{n}{k_i})$.

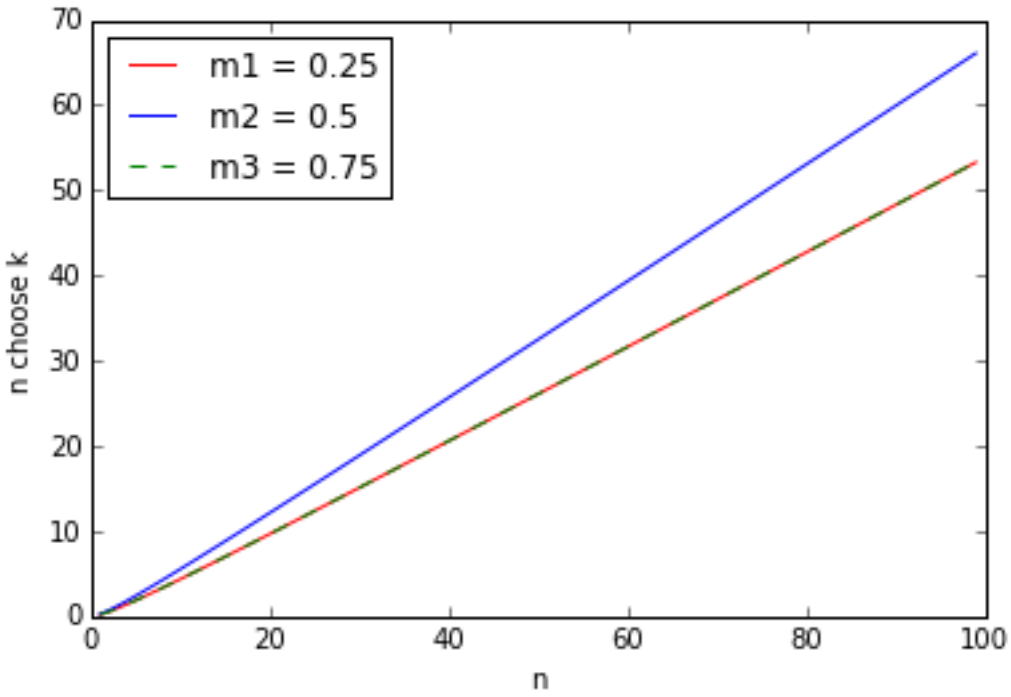
Note: `math.log(x)` and `scipy.special.binom(n, k)` could be useful here. Use `scipy.special.binom(n, k)` instead of our home-made `choose(n, k)`.

```
In [63]: def j_1(n):
             """
             YOUR CODE HERE
             """

             return math.log(scipy.special.binom(n, 0.25*n))

         def j_2(n):
             """
             YOUR CODE HERE
             """

             return math.log(scipy.special.binom(n, 0.5*n))
```

```
        def j_3(n):
            """
            YOUR CODE HERE
            """

            return math.log(scipy.special.binom(n, 0.75*n))
```

In [64]:
```
def partJ():
    """
    Part (j) code

    YOUR CODE HERE
    """

    N = 100
    plt.plot(range(1, N), [j_1(i) for i in range(1, N)], color='red', label='m1 = 0.25')
    plt.plot(range(1, N), [j_2(i) for i in range(1, N)], color='blue', label='m2 = 0.5')
    plt.plot(range(1, N), [j_3(i) for i in range(1, N)], color='green', label='m3 = 0.75', lin
    plt.xlabel("n")
    plt.ylabel("n choose k")
    plt.legend(loc=2)
    plt.show()
```

In [65]:
```
partJ()
```



YOUR COMMENTS HERE:

Part (l): (Optional) Approximate Slope

Use a point at $n = 70$ and a point at $n = 90$ to approximate the slopes of your lines. Use these slopes for the rest of Question 8's part (j); putting this in this file was just to guide you on how you are expected to get them.

There is no extra credit for doing this question. It is supposed to help you with part (j) of Question 8.

In [66]:
```
# Slope of (n choose k_1)
# YOUR CODE HERE
slope1 = (j_1(90) - j_1(70))/(90-70)

# Slope of (n choose k_2)
# YOUR CODE HERE
slope2 = (j_2(90) - j_2(70))/(90-70)

# Slope of (n choose k_3)
# YOUR CODE HERE
slope3 = (j_3(90) - j_3(70))/(90-70)
```

In [67]:
```
print("Slope 1: {0}, Slope 2: {1}, Slope 3: {2}".format(slope1, slope2, slope3))
```

```
Slope 1: 0.556109589137, Slope 2: 0.686903999178, Slope 3: 0.556109589137
```

Congratulations! You are done with Virtual Lab 11.

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.