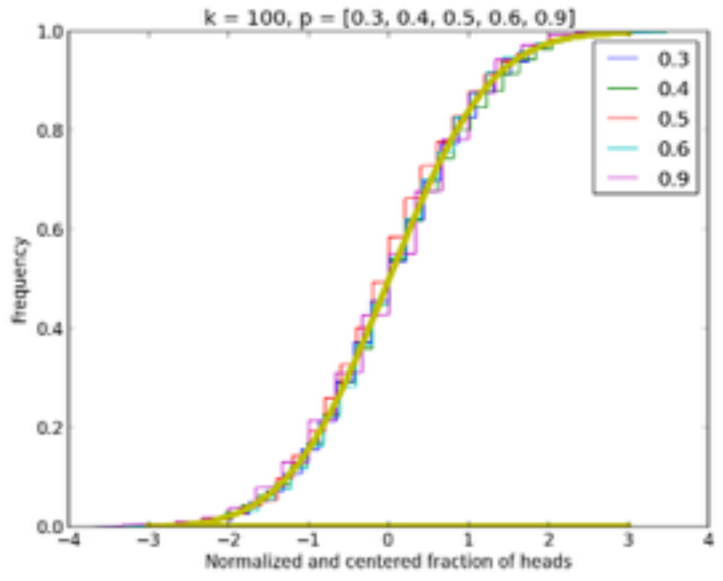
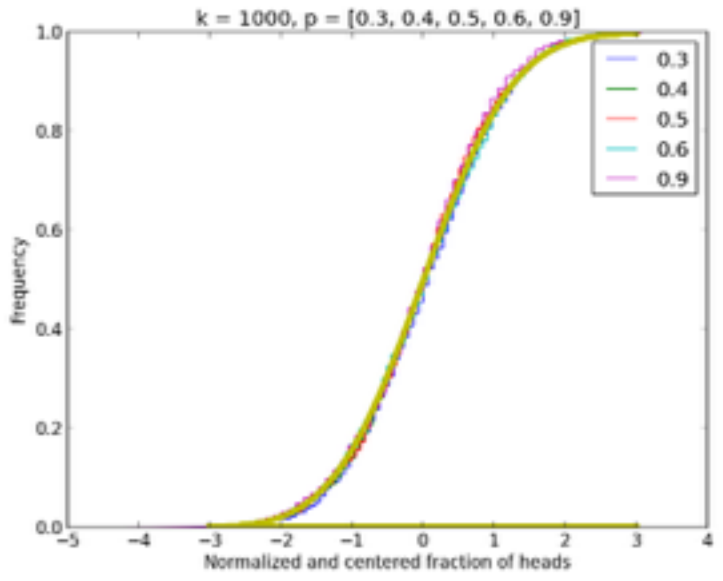
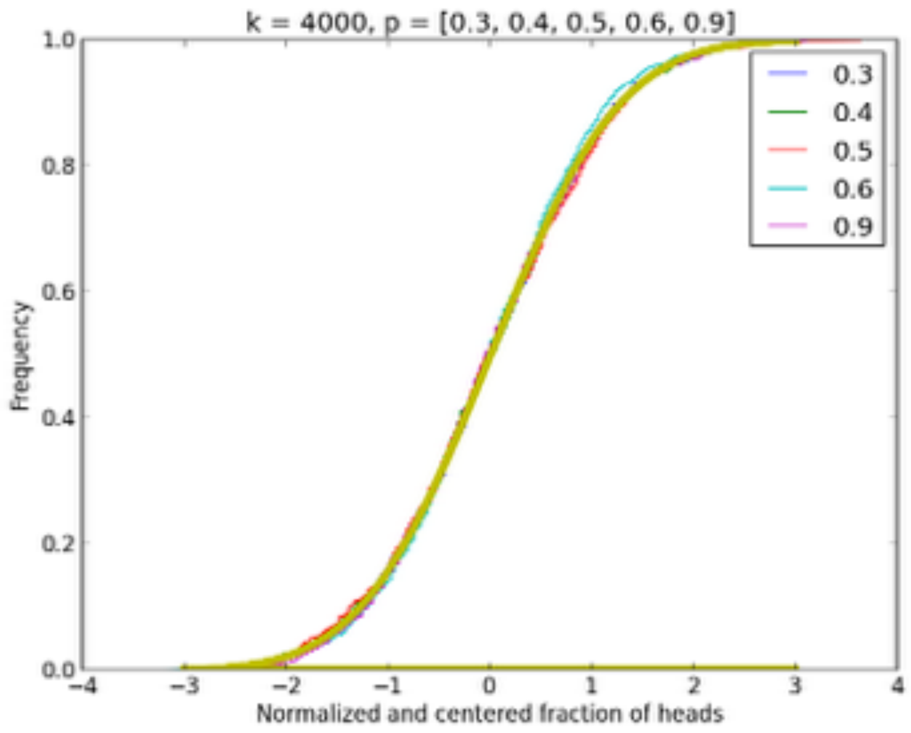
HW Problem 1 Writeup
Homework partners: Sam Drake, Serena Chan
Grader Instructions: Each part a-f below contains the code to run each part in python 2.7 with the partx() functions in the problem1.py file provide in the .zip file. Note, the python file contains some functions that are used in each of the parts below, but are only in the python file, so make sure you run these functions after starting with the command python2.7 -i problem1.py, otherwise these functions do not work on their own. Each part also contains all the relevant plots and comments; the plots will also be in the zip file.

Helper Functions:
```
import random
import numpy as np
import scipy.stats as stats
from scipy import integrate
import math
import matplotlib as mpl, matplotlib.pyplot as plot

def biasedCoin (p):
        # Creates a biased coin with p(Head) = p
        # Returns true if heads and false otherwise

        # Should check this!
        # assert p >= 0 and p <= 1
        return random.random() <= p

def runTrial (p, k):
        # Runs a trial of k tosses of a biased coin (w.p. p of heads)
        # and returns number of heads
        return sum([biasedCoin(p) for _ in xrange(k)])

def runManyTrials (p, k, m):
        # Runs m trials of k tosses of a biased coin (w.p. p of heads)
        # and returns all the numbers of heads
        return [runTrial(p, k) for _ in xrange(m)]

def calculateQuartileGap(results):
        # Calculates the quartile
        results.sort()
        n = len(results)
        q1 = int(round(0.25*n))
        q3 = int(round(0.75*n))
        return results[q3]-results[q1]

def linspace(a,b,n):
        # Returns n numbers evenly spaced between a and b, inclusive
        return [(a+(b-a)*i*1.0/(n-1)) for i in xrange(n)]
```

a) CODE:
```
def parta(pranges=[0.3,0.4,0.5,0.6,0.9], kranges = [100,1000,4000], m=1000,
show_indiv=False):
        # Q2 part (k)
        print('Question 2 part (k):')
        for k in kranges:
                plot.clf()
                results = {}
                print ('Number of trials k = %i'%k)
                for p in pranges:
                        print ('Probability of head p = %.1f'%p)
                        std = math.sqrt(p*(1-p))
                        results[p] = [(Sk -k*p)/(math.sqrt(k)*std) for Sk in runManyTrials(p, k, m)]
                        results[p].sort()
                        plot.plot(results[p],linspace(0,1,m),label=str(p))
                        x_values = np.arange(-3.0,3.1,.1)
                        y_values = list()
                        for i in x_values:
                                y_values.append(integral(i))
                        plot.plot(x_values,y_values,linewidth=4, color='y')
                plot.legend()
                plot.ylabel('Frequency')
                plot.xlabel('Normalized and centered fraction of heads')
                plot.title('k = %i, p = %s'%(k,str(pranges)))
                plot.show()

def integrand(x):
        return ((1/math.sqrt(2*math.pi))*math.exp(-1*(math.pow(x,2)/2)))

def integral(d):
        return integrate.quad(lambda x: integrand(x),-np.inf,d)
```





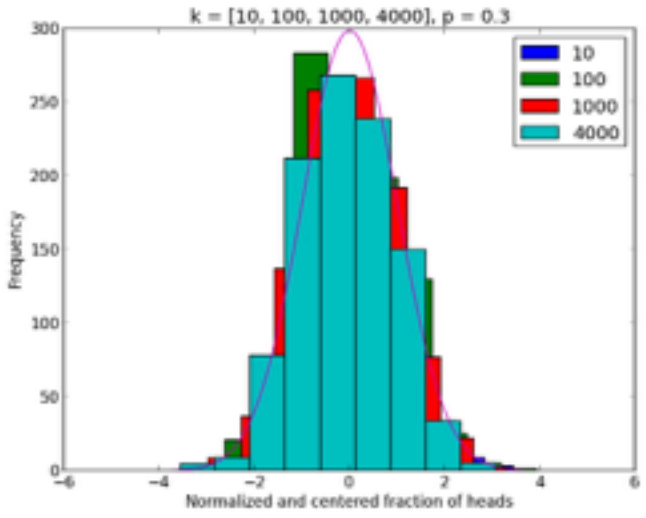


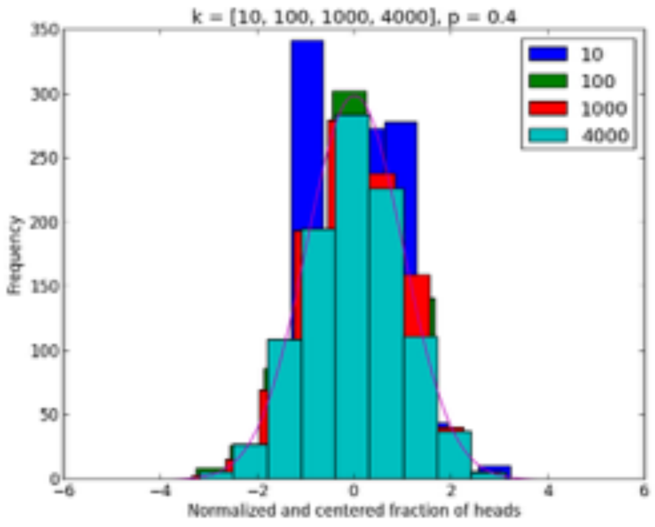
For this part, the function I plot was the integral as stated in the problem question, which i plotted against continuous values of d, since the x is just a dummy variable of integration and in general the integral is one value when evaluated for a certain d— so it made sense that I was plotting the function against d. The curve lines up with the S curves for all the k's.

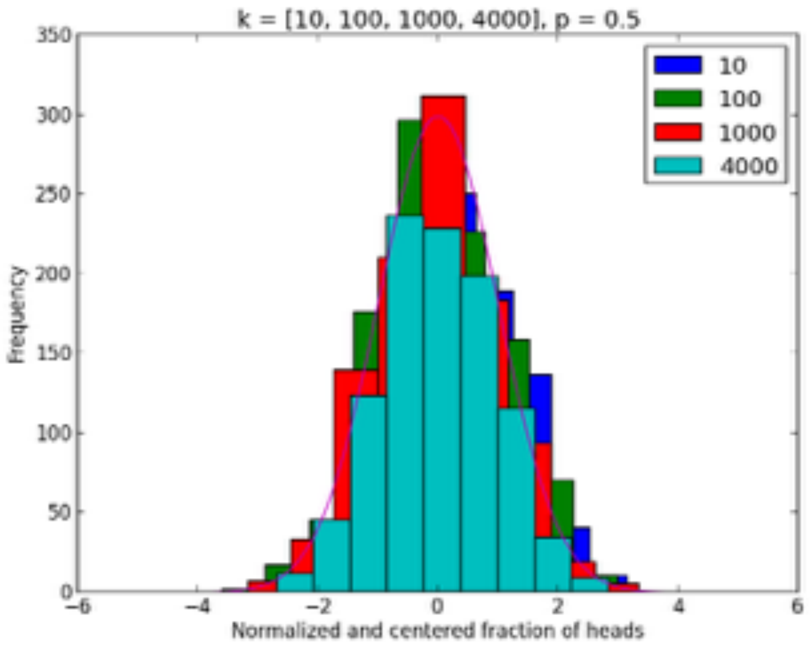
b) This part doesn't have codes or graphs
The total number of heads S is just the sum of all X_i from i=0 to to i = k. So S = X_1 + X_2 + ... + X_k. When you are at the jth toss, the number of heads so far is just the sum from i=0 to i=j. Of course, since S is a random number, the number of coin tosses S will have variations like what we have seen in previous homeworks.

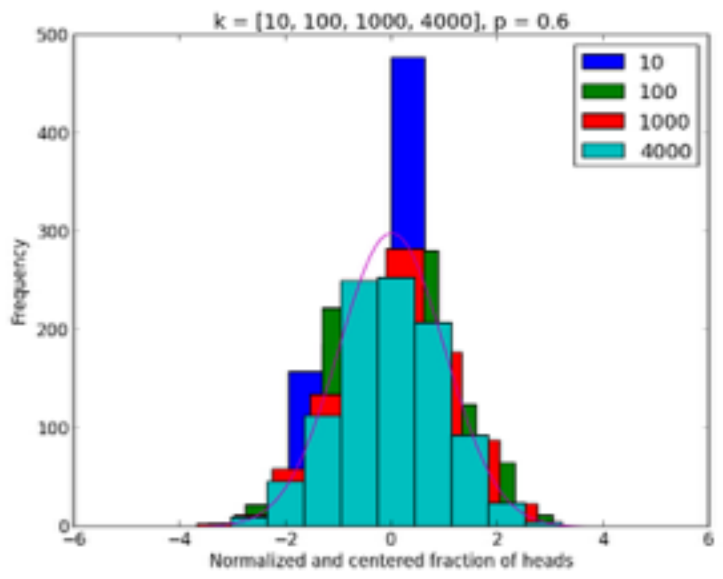
c)
```
def scale_integrand(x):
    return 750*integrand(x)

def partc(pranges=[0.3,0.4,0.5,0.6,0.7,0.9], kranges = [10,100,1000,4000], m=1000,
show_indiv=False):
    # Q2 part (j)
    print('Question 2 part (j):')
    for p in pranges:
        print ('Probability of head p = %.1f'%p)
        std = math.sqrt(p*(1-p))
        results = {}
        for k in kranges:
            results[k] = [(Sk -k*p)/(math.sqrt(k)*std) for Sk in runManyTrials(p, k, m)]
            if show_indiv:
                # bins = 9 so as to not have gaps in the display
                plot.hist(results[k],bins=9,label=str(k))
```

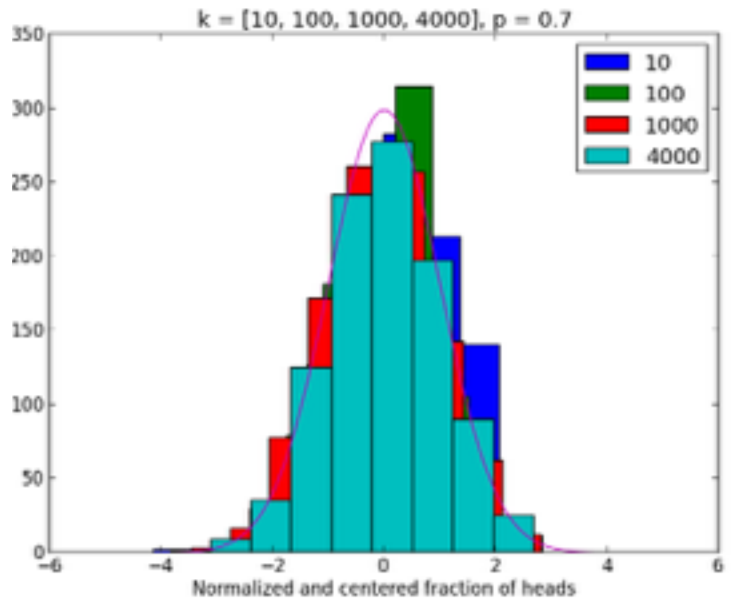
```
                    plot.show()
plot.clf()
x_values = np.arange(-5,5,.1)
y_values = [scale_integrand(i) for i in x_values]
for k in kranges:
          # bins = 9 so as to not have gaps in the display
          plot.hist(results[k],bins=9,label=str(k),histtype='barstacked')
plot.legend()
plot.ylabel('Frequency')
plot.xlabel('Normalized and centered fraction of heads')
plot.title('k = %s, p = %.1f'%(str(kranges),p))
plot.plot(x_values,y_values)
plot.show()
```

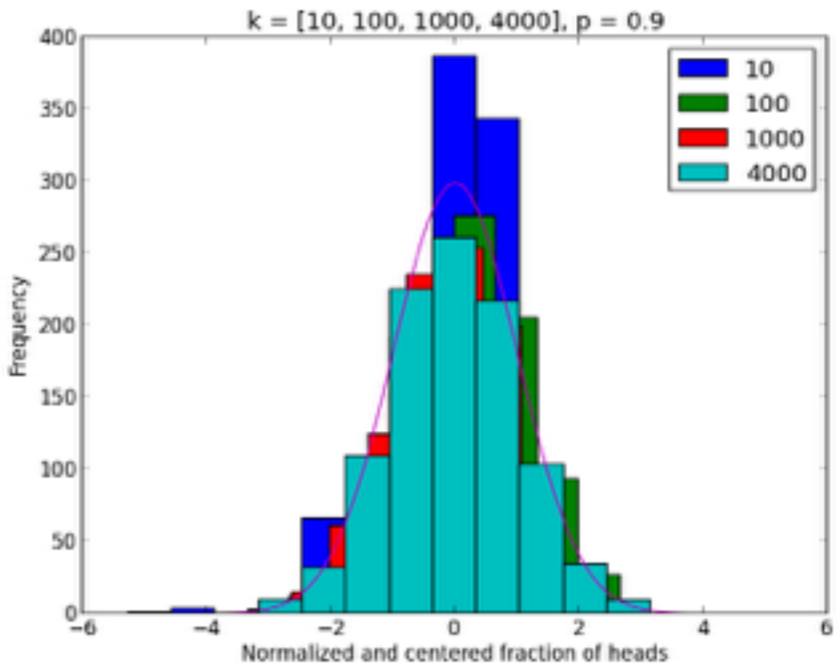










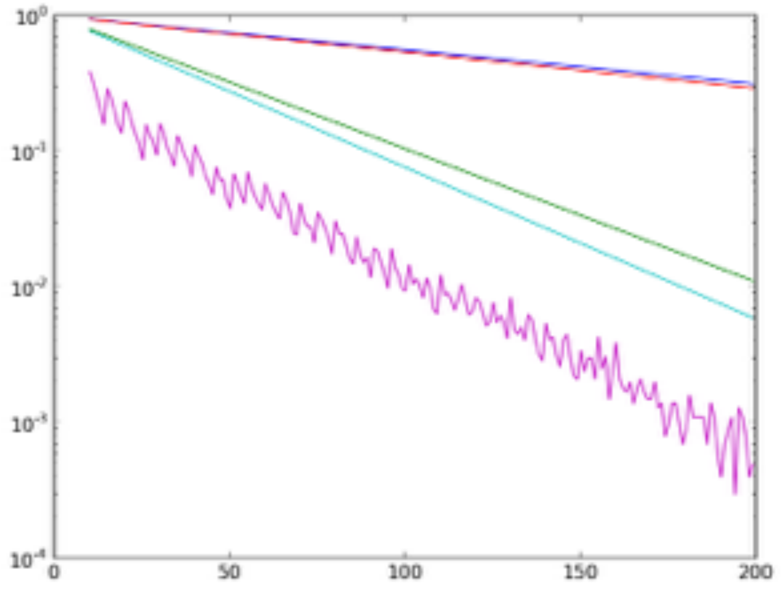


In order to gaussian function to lay over my histograms, I had to scale it by roughly 750, since the histogram y values are counts out of 1000 trials, whereas the gaussian itself has values between 0 and 1/sqrt(2*pi). This means that there is a way that we can represent our histograms since they seem to fit under the scaled gaussian function. That is, it looks like the area under the gaussian roughly corresponds to the “area” taken up by the histogram values. In essense, it means we can take an integral of the gaussian function as another representation of the histograms. This is exactly what we did in part a, and it seems that the integral of the gaussian produces the correct S curve shape that we got when we transformed the histograms into S curves.

d)

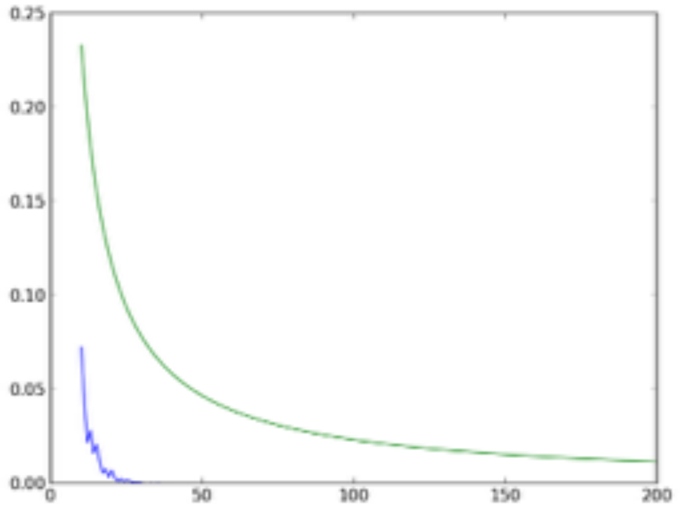
```
def int_a(a,p):
        return a*math.log((a/p),math.e)+(1-a)*math.log(((1-a)/(1-p)),math.e)
```

```
def func(a,p,k):
        return math.pow(math.e,(-1*int_a(a,p)*k))

def partd():
        r = np.arange(10,200,1)

        p_vals = [.3,.7]
        a_vals = [.05,.1]
        m = 10000
        for p in p_vals:
                for a_shift in a_vals:
                        a = p + a_shift
                        arr = [func(a,p,k) for k in r]
                        plot.semilogy(r,arr)
                        deltay = math.log(arr[-1])-math.log(arr[0])
                        slope = deltay/190
                        print("slope: " + str(slope))
        results = []
        for k in range(10,200):
                total = 0
                for n in runManyTrials(p,k,m):
                        if n > a*k:
                                total += 1
                results.append(total/float(m))
        plot.semilogy(range(10,200),results)
        plot.show()
```



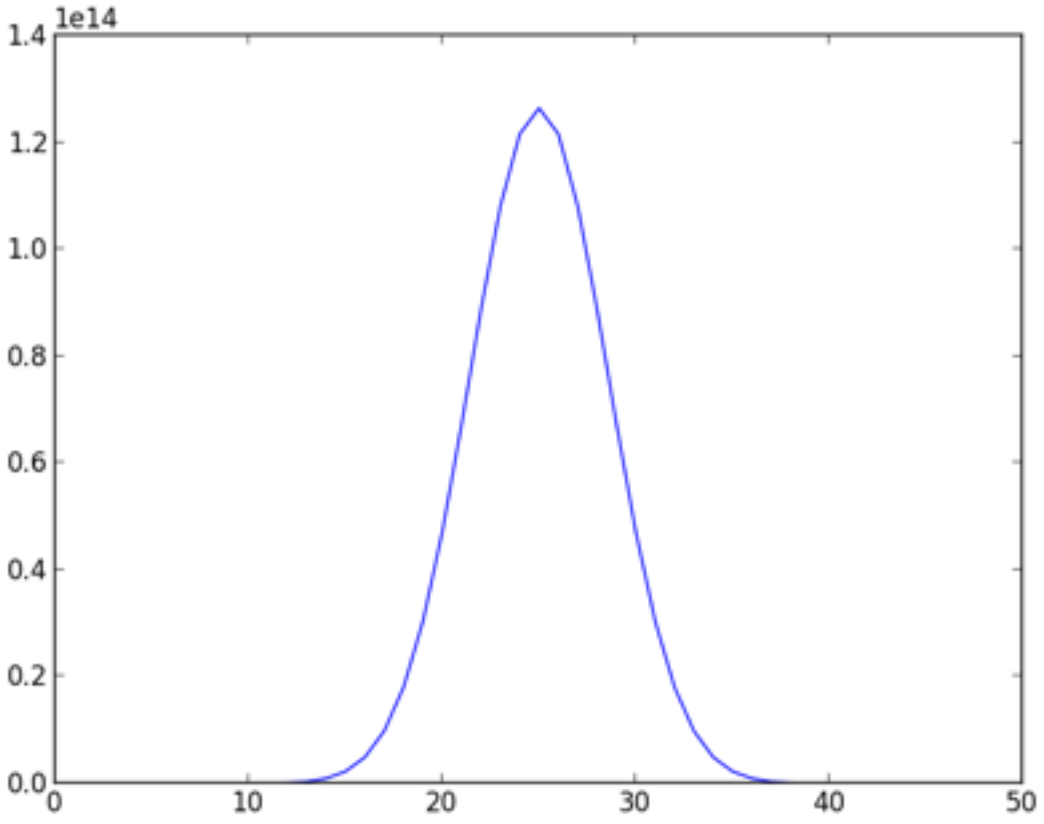
In a log-linear graph, we can get the exponential drops to look like straight lines. It seems that the Chernoff bound (the Kullback-Liebler divergence against k) sets a lower limit to all the graphs. It suggests an inequality—all 4 of the plots are greater than the Chernoff bounds, and it looks like for all values of k along the x axis. In other words, the Chernoff bound gives us an asymptotic limit for the behavior of all combinations of p and a—no combination could ever be under this bound, as suggested by the first cases that we tried.
For p=.3 and a=.35, the KL divergence gives a value of .00578, which is roughly the negative of the slope of the first plot, which is approximate -.5/200=-.0025 on the log-linear scale.
For p=.3 and a=.4, the KL divergence gives a value of .0225; the negative of the slope of the second plot is approximatel -.5/200=-.0025 on the log-linear scale (same as the previous plot). This value doesn't match up quite as well, and is strangely off by a factor of 10.
For p=.7and a=.75, the KL divergence gives a value of .061, which is roughly the negative of the slope of the first plot, which is approximate -2/200=-.01 on the log-linear scale. They match up okay.
For p=.7 and a=.8, the KL divergence gives a value of .0257, which is in the same order of magnitude as the negative of the slope of the fourth plot, which is -.01 (same as the previous plot).

e)
```
def funct (eps,p,k):
    return p*(1-p)/(k*math.pow(eps,2))

def parte():
    x_values = np.arange(10,201)

    m = 10000
    p = .3
    for eps in [.1,.2,.3]:
        results = []
        for k in x_values:
            arr = [math.fabs(Sk -k*p) for Sk in runManyTrials(p, k, m)]
            total = 0
            for elem in arr:
                if elem >= eps*k:
                    total += 1
            results.append(total / float(m))
    plot.plot(x_values,results)
    plot.plot(x_values,[funct(eps,p,k) for k in x_values])
    plot.show()
```



As the graph shows, we can get plot to be contained by the Chebyshev's equality, which is clearly greater than the plot at all points, so the inequality is valid for our coins. Its hard to plot of them on the same graph because they have different exponential behaviors, but in general it is clear that the inequality completely bounds every value in the actual frequencies.

f)



```
def choose(n,k):
	return math.factorial(n)/(math.factorial(k)*math.factorial(n-k))

def partf():
	x_values = np.arange(0,50.1,1)

	y_values = [choose(50,k) for k in x_values]

	plot.plot(x_values,y_values)
	plot.show()
```

The combination function (unordered picking, no replacements) is not always growing. It has a peak around the half way point of 50, and is fairly symmetric. It looks very much like a bell curve distribution, or the gaussian function that we plotted earlier, so I suspect that combinations are very related to the coin toss lab that we have been doing.