# lab10sol

November 10, 2014

# 1 Virtual Lab 10 Solution: Biased Coin

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

**Due Date: Monday, November 10th, 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
```

## Introduction

In this week's lab, we will continue to explore the ideas behind coin tosses. Denote $p$ as the probability of tossing a head, and $1-p$ as the probability of tossing a tail. In the last two labs, $p$ was 0.5, since we were exploring fair coin tosses. Now assume $p$ can be any number such that $0 \leq p \leq 1$. As before, let $k$ denote the total number of coin tosses.

At an abstract level, most of what this lab is about is doing the previous labs again, except that now the coin is biased. One new concept is introduced: we now need to understand how the bias of the coin affects the shapes of curves that emerge.

This lab might initially look long and daunting, but we are essentially redoing the previous two labs again. Do not worry about the length.

## Part (a): Biased Coin Warm-up

Consider $p = 0.7, \bar{p} = 1 - p = 0.3$. If you plotted a histogram for the number of heads and tails for any number of coin cosses $k$, what would you expect to see that is different from the fair coin toss?

YOUR ANSWER HERE:

## Part (b): Biased Coin Equation

If I tossed 100 coins, approximately how many heads should I get? Building on this, can you come up with an equation using $k, p, \bar{p}$ that approximately describes how many heads and tails you expect to see for any case?

YOUR ANSWER HERE:

## Part (c): Biased Coin's Heads

Use a random number generator to sample a sequence of coin tosses with $p = 0.7, \bar{p} = 0.3$. Plot a bar chart for each $k = 10, 100, 1000, 4000$. Is this what you expected from parts (a) and (b)? How well does your equation match for $k = 10$? For $k = 4000$?

*Hint*: Implement the functions `biased_coin`, which simulates a biased coin flip with $Pr(\text{Head}) = p$, and `run_trial`, which returns the number of heads in $k$ tosses of a biased coin with probability $p$ of getting heads. They should be very similar to the ones you implemented previously for the fair coin.

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

    YOUR CODE HERE
    """

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

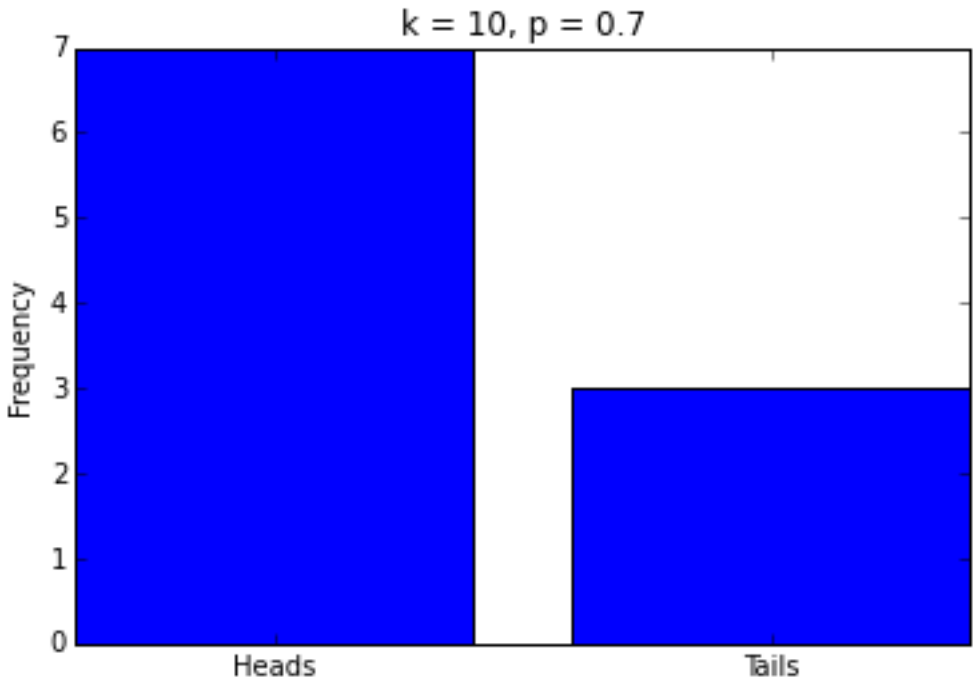
In [4]:
```
def run_trial(p, k):
    """
    Runs a trial of k tosses of a biased coin (w.p. p of heads)
    and returns number of heads.

    YOUR CODE HERE
    """

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

In [5]:
```
def partC(p=0.7, kranges=[10,100,1000,4000]):
    """
    Part (c) code
```

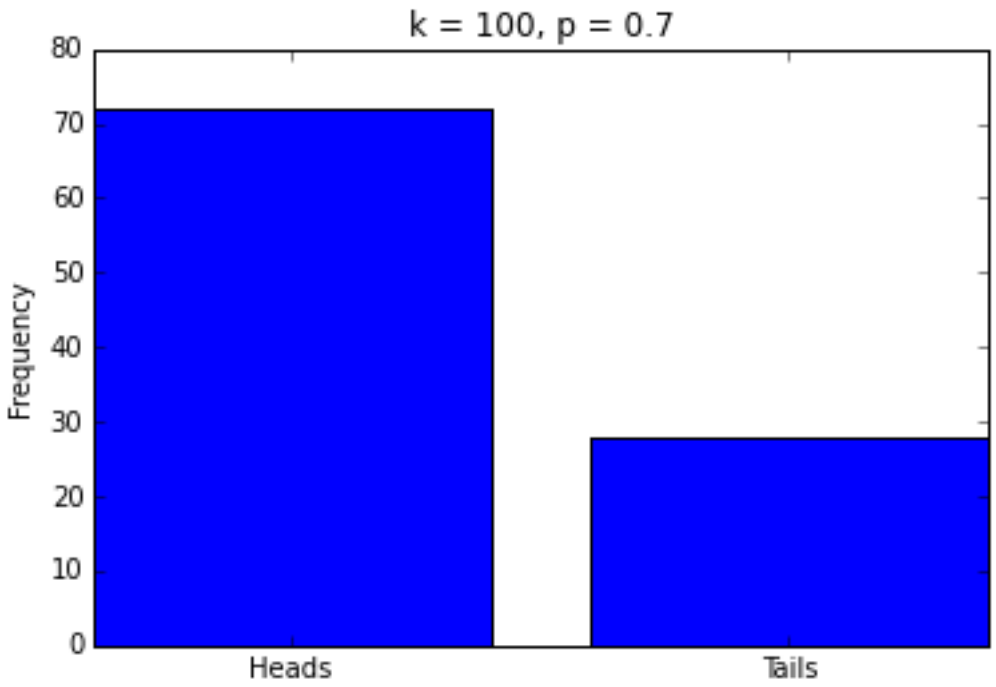
```
    """

    print 'Question 2 part (c):'
    print 'Probability of head: %f'%p
    for k in kranges:
        heads = run_trial(p, k)
        print 'Number of tosses: %i'%k
        print '\tHeads: %i, Tails: %i, H/(H+T): %f'%(heads, k-heads, heads*1.0/k)

        # YOUR CODE HERE
        plt.bar([0, 1], [heads, k-heads])
        plt.xticks([0.4, 1.4],['Heads', 'Tails'])
        plt.ylabel('Frequency')
        plt.title('k = %i, p = %.1f'%(k,p))
        plt.show()
```

In [6]:
```
partC()
```

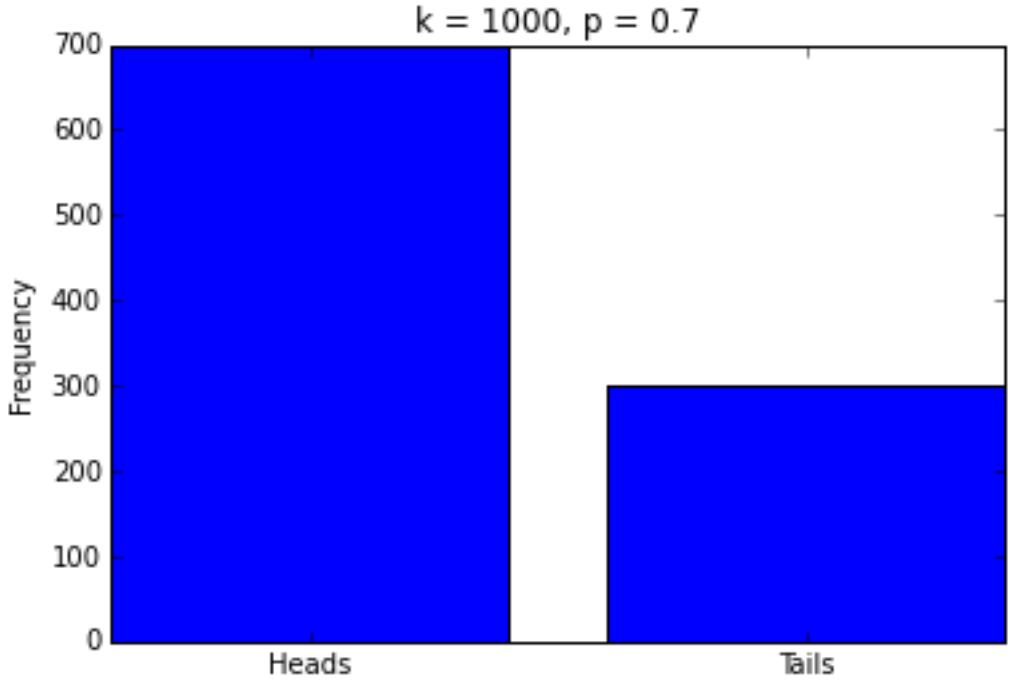
```
Question 2 part (c):
Probability of head: 0.700000
Number of tosses: 10
        Heads: 7, Tails: 3, H/(H+T): 0.700000
```



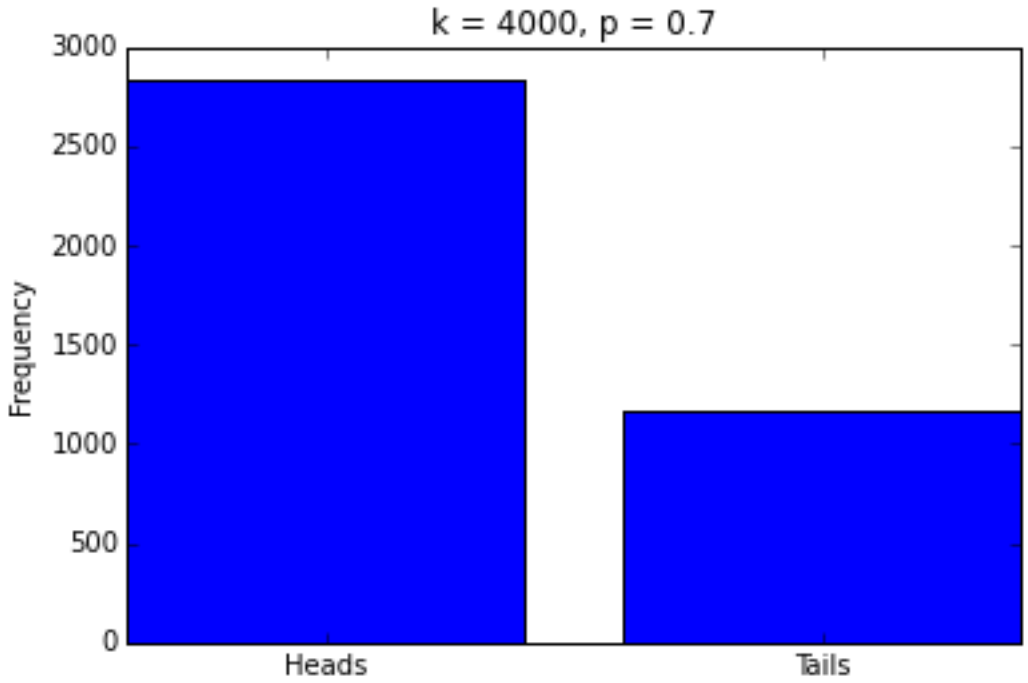

```
Number of tosses: 100
        Heads: 72, Tails: 28, H/(H+T): 0.720000
```




```
Number of tosses: 1000
        Heads: 699, Tails: 301, H/(H+T): 0.699000
```



```
Number of tosses: 4000
        Heads: 2838, Tails: 1162, H/(H+T): 0.709500
```




YOUR COMMENTS HERE:

## Part (d): Number of Heads in Many Trials

From the previous parts, you've counted the number of heads and tails for only one trial. Now fix $k$ to be 1000. Let $S_k$ be the total number of heads in a trial with $k$ total coin cosses, and let $m$ be the total number of trials.

Plot a histogram of $S_k$ for $m = 1000$ with bin size of 1. Do this again for $k = 10, 100, 4000$.

*Hint*: Implement the function `run_many_trials`, which returns a list of the number of heads in each trial.

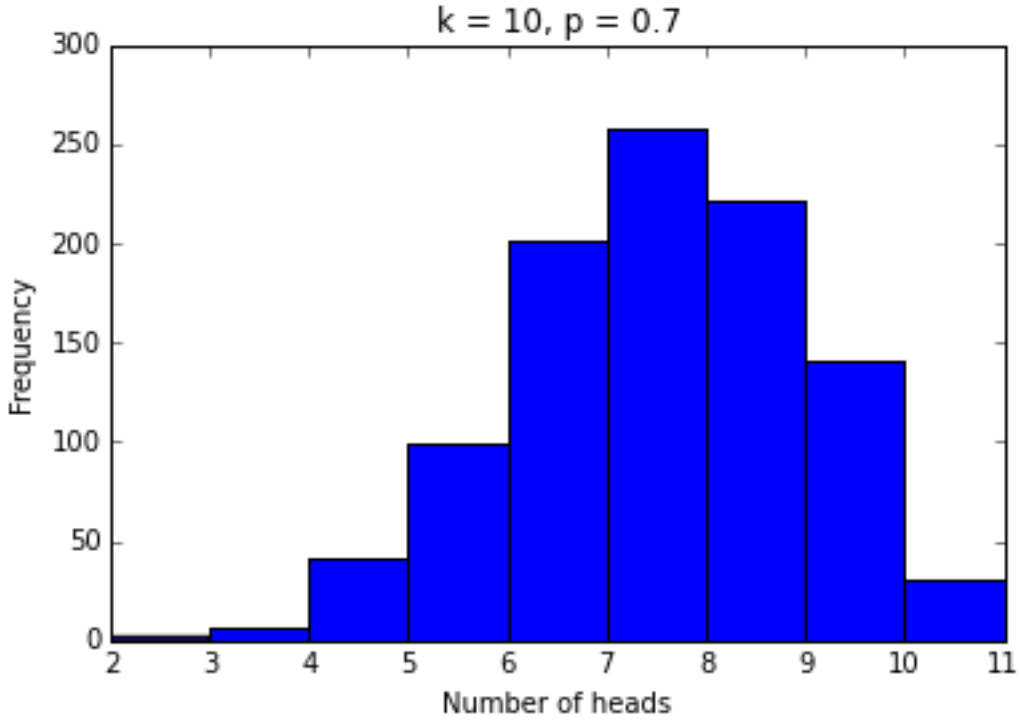
```
In [7]: def run_many_trials(p, k, m):
            """
            Runs m trials of k tosses of a biased coin (w.p. p of heads)
            and returns a list of the numbers of heads.

            YOUR CODE HERE
            """

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

```
In [8]: def partD(p=0.7, kranges=[10,100,1000,4000], m=1000):
            """
            Code for part (d)
            """

            print 'Question 2 part (d):'
            print 'Probability of head: %f'%p
            bin_width = 1
```

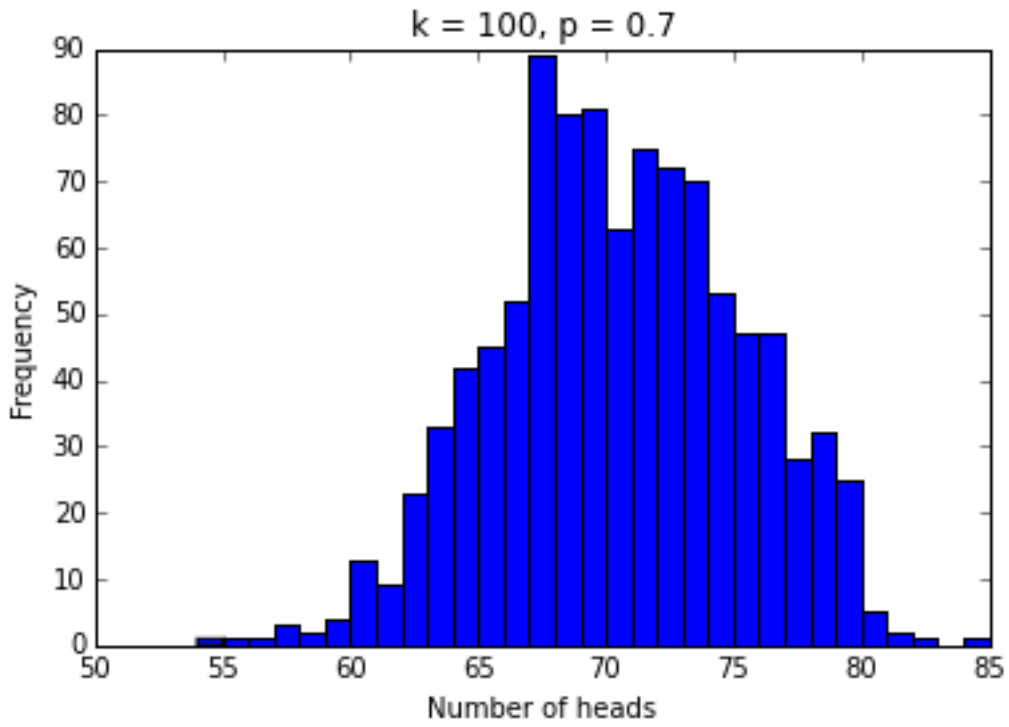
```
    for k in kranges:
        results = run_many_trials(p, k, m)
        print 'Number of tosses: %i'%k
        print 'Number of trials: %i'%m
        # YOUR CODE HERE
        plt.hist(results, bins=xrange(min(results), max(results)+bin_width*2, bin_width), align=
        plt.ylabel('Frequency')
        plt.xlabel('Number of heads')
        plt.title('k = %i, p = %.1f' % (k, p))
        plt.show()
```

In [9]: partD()

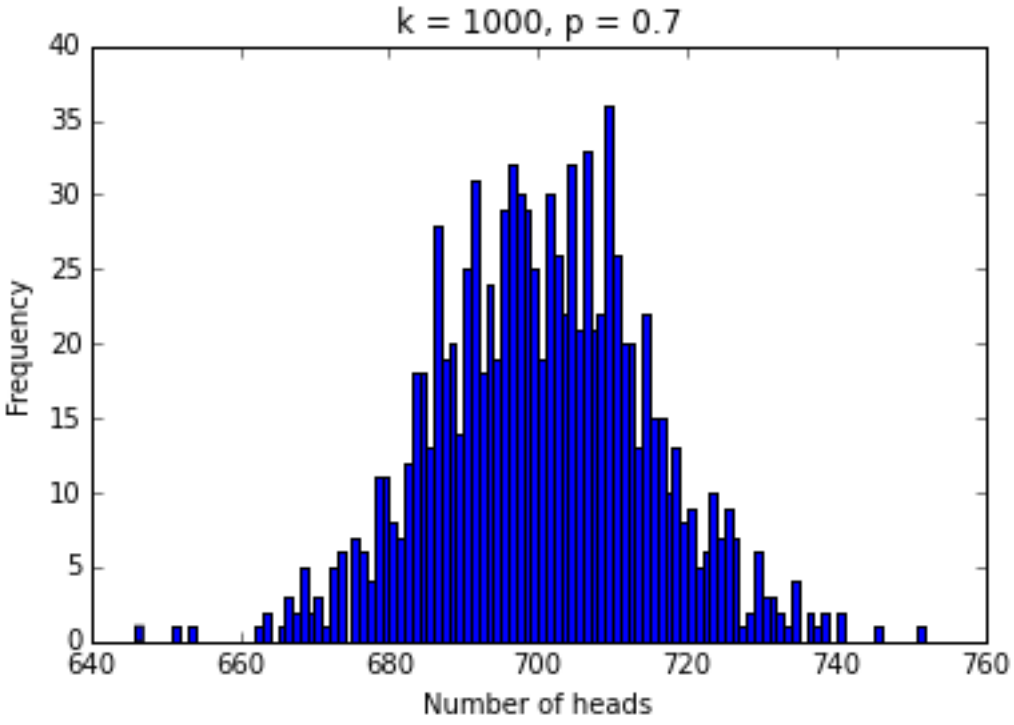
```
Question 2 part (d):
Probability of head: 0.700000
Number of tosses: 10
Number of trials: 1000
```



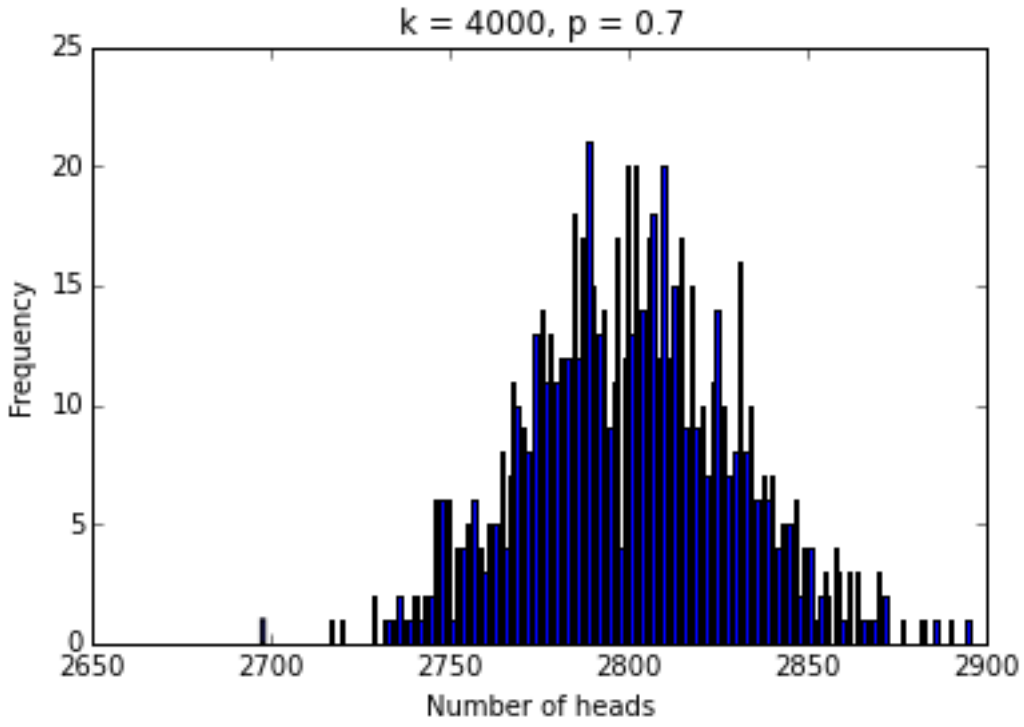

```
Number of tosses: 100
Number of trials: 1000
```




```
Number of tosses: 1000
Number of trials: 1000
```



```
Number of tosses: 4000
Number of trials: 1000
```




## Part (e): S_k / k

Repeat the previous part, but instead plot the histograms of $\frac{S_k}{k}$ with bin size of 0.01. Make sure to plot the histograms in the same figure (so only one plot) for different values of $k$. How are the histograms different as $k$ increases? Comment on what you are observing.

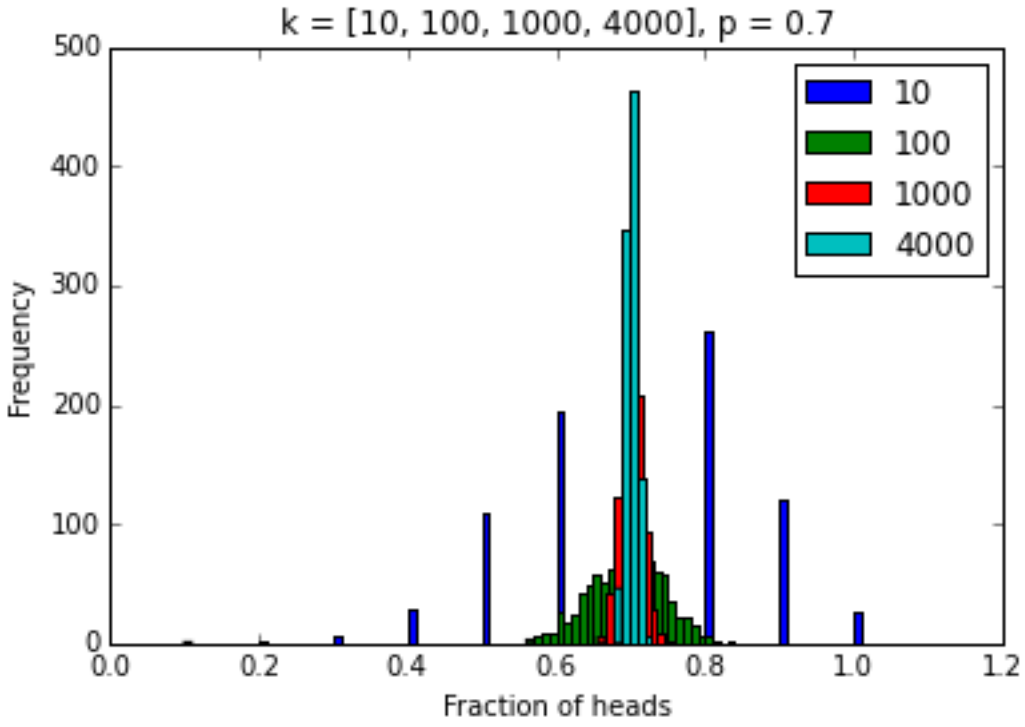
*Hint*: You can use the argument `histtype='barstacked'` when calling `plt.hist` to stack the bars on top of each other.

```
In [10]: def partE(p=0.7, kranges=[10,100,1000,4000], m=1000):
             """
             Code for part (e)
             """

             print 'Question 2 part (e):'
             print 'Probability of head: %f'%p
             bin_width = 0.01

             # YOUR CODE HERE
             results = {}
             for k in kranges:
                 results[k] = [Sk*1.0/k for Sk in run_many_trials(p, k, m)]
                 plt.hist(results[k], bins=np.arange(min(results[k]), max(results[k])+bin_width*2, bin_
                          label=str(k), histtype='barstacked')
```

```
            # END YOUR CODE HERE
            plt.legend()
            plt.ylabel('Frequency')
            plt.xlabel('Fraction of heads')
            plt.title('k = %s, p = %.1f' % (str(kranges), p))
            plt.show()
```

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

```
Question 2 part (e):
Probability of head: 0.700000
```




YOUR COMMENTS HERE:

## Part (f): Normalization Redux

In last week's lab, you discovered a very interesting normalization by $\sqrt{k}$ that seemed to make certain curves fall right on top of each other. Let's see if that still works.

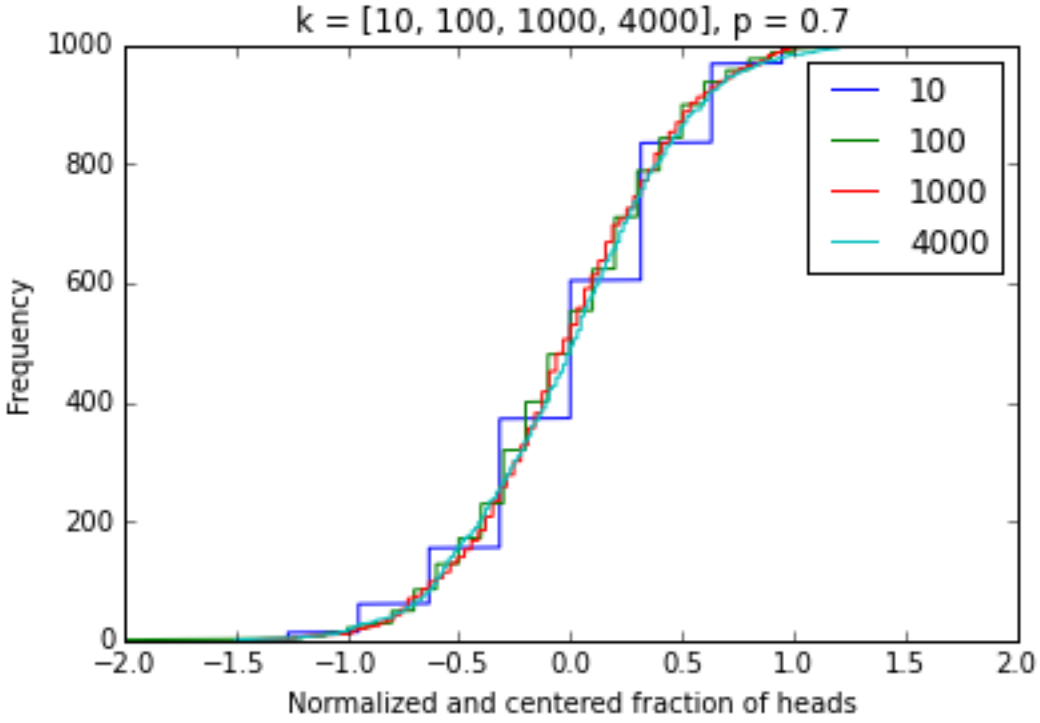
Redo the plot you did in part (g) of last week's lab, except for our biased coin and the $k$ values you have explored above. Center the horizontal axis of the plot to be around 0.

```
In [12]: def partF(p=0.7, kranges=[10,100,1000,4000], m=1000):
            """
            Code for part (f)
            """

            print 'Question 2 part (f):'
            print 'Probability of head: %f'%p
            # YOUR CODE HERE
```

```
    results = {}
    for k in kranges:
        results[k] = [(Sk*1.0/k-p)*math.sqrt(k) for Sk in run_many_trials(p, k, m)]
        results[k].sort()
        plt.plot(results[k], xrange(1,m+1), label=str(k))

    # END YOUR CODE HERE
    plt.legend()
    plt.ylabel('Frequency')
    plt.xlabel('Normalized and centered fraction of heads')
    plt.xlim(-2.0,2.0)
    plt.title('k = %s, p = %.1f' % (str(kranges), p))
    plt.show()
```

```
In [13]: partF()
```

```
Question 2 part (f):
Probability of head: 0.700000
```




## Part (g): Histogram Normalization

For kicks, let's try this same normalization for histograms. Plot histograms of $\frac{S_k - kp}{\sqrt{k}}$ for $m = 1000$ and $k = 10, 100, 1000, 4000$. Try 4 different bin sizes, $0.01, 0.05, 0.1, 0.5$. You should have 4 different plots, one for each bin size.

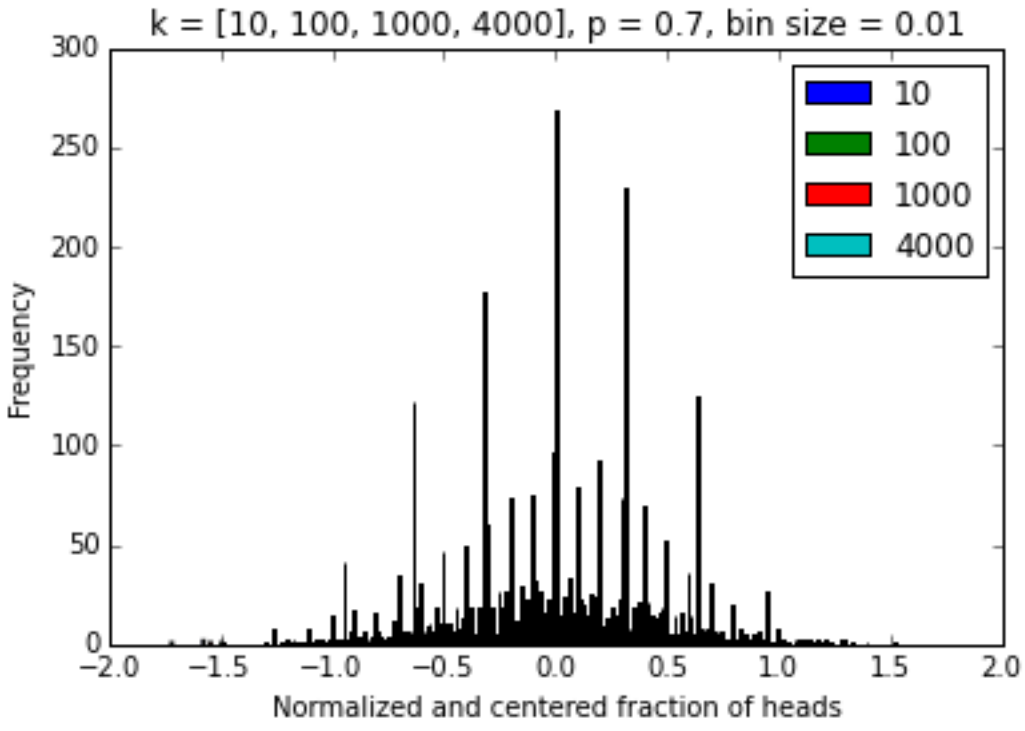
Comment on what you think the relationship is between these histograms and the "cliff-face" curves in the previous part. Think about what they mean.

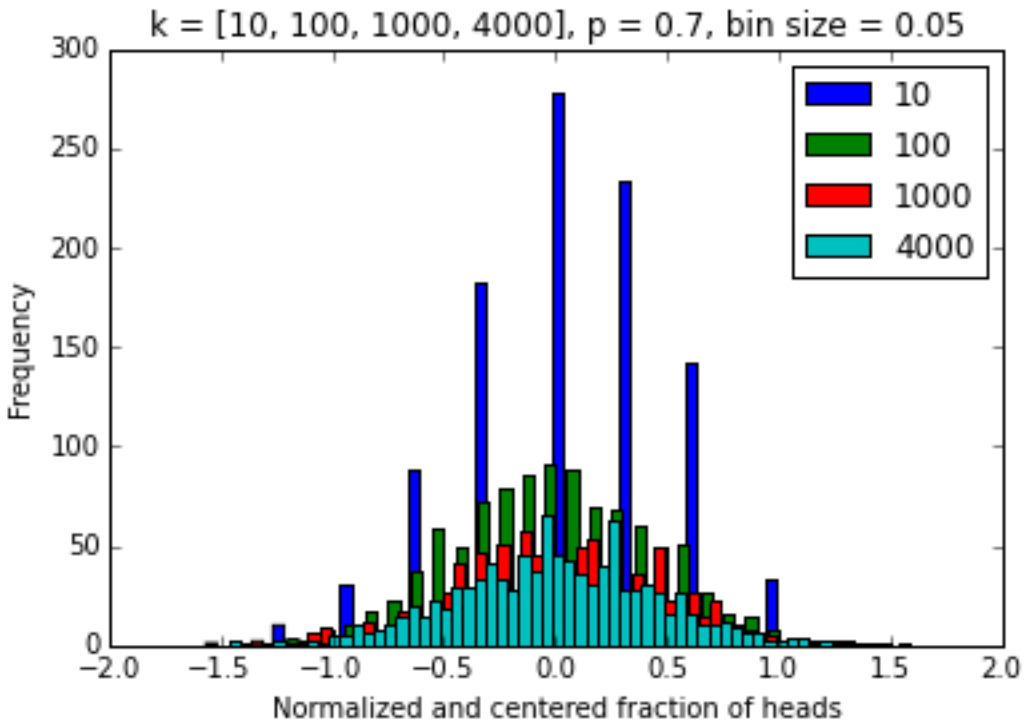
```
In [14]: def partG(p=0.7, kranges=[10,100,1000,4000], m=1000, bin_widths=[0.01,0.05,0.1,0.5]):
             """
```

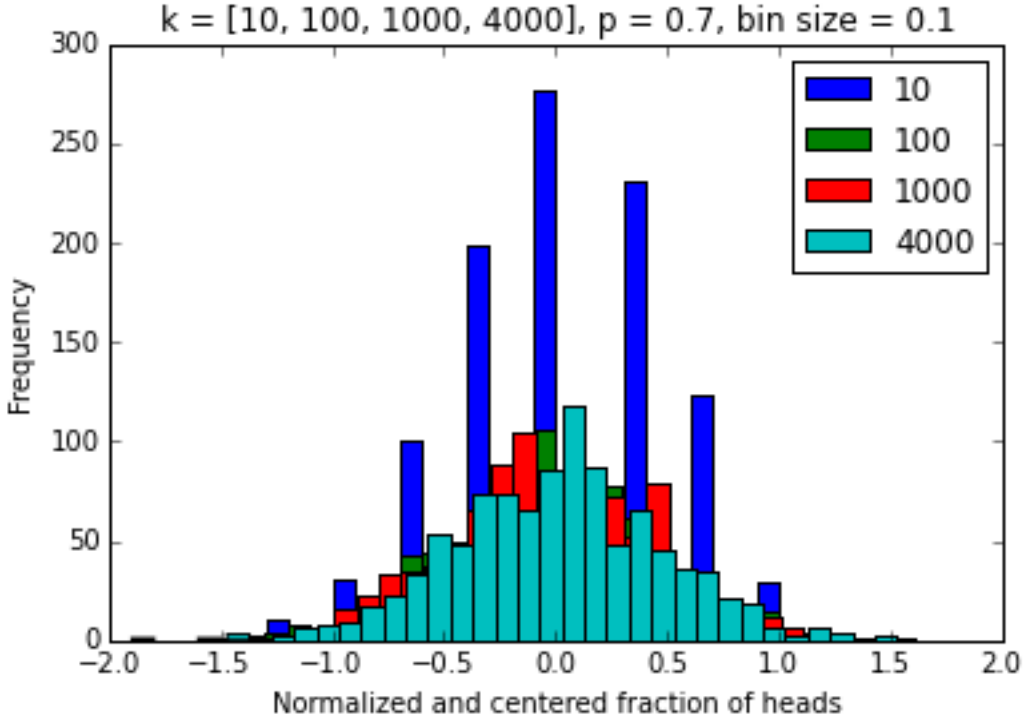
```
    Code for part (g)
    """

    print 'Question 2 part (g):'
    print 'Probability of head: %f'%p
    results = {}
    for bin_width in bin_widths:
        plt.figure()

        # YOUR CODE HERE
        for k in kranges:
            results[k] = [(Sk -k*p)/math.sqrt(k) for Sk in run_many_trials(p, k, m)]
            plt.hist(results[k], bins=np.arange(min(results[k]), max(results[k])+bin_width*2,
                     align='mid', label=str(k), histtype='barstacked')
        # END YOUR CODE HERE

        plt.legend()
        plt.ylabel('Frequency')
        plt.xlabel('Normalized and centered fraction of heads')
        plt.xlim(-2.0,2.0)
        plt.title('k = %s, p = %.1f, bin size = %s' % (str(kranges), p, bin_width))
        plt.show()
```

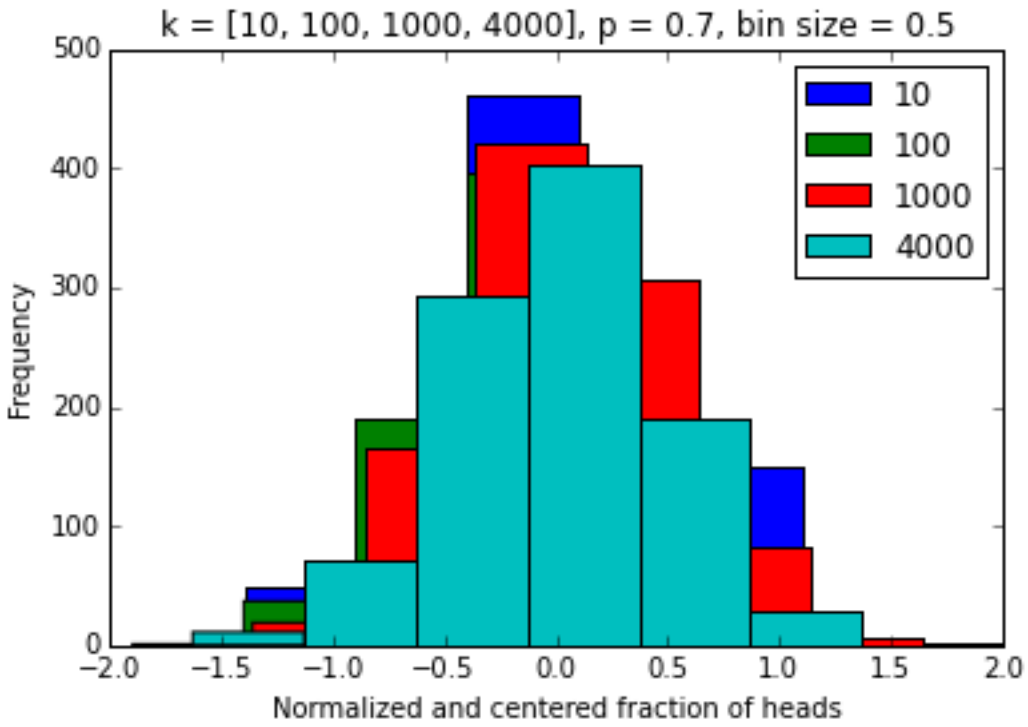
In [15]:
```
partG()
```

```
Question 2 part (g):
Probability of head: 0.700000
```




k = [10, 100, 1000, 4000], p = 0.7, bin size = 0.05
Frequency
Normalized and centered fraction of heads
10
100
1000
4000


k = [10, 100, 1000, 4000], p = 0.7, bin size = 0.1
Frequency
Normalized and centered fraction of heads
10
100
1000
4000




YOUR COMMENTS HERE:

## Part (h): Varying the Bias

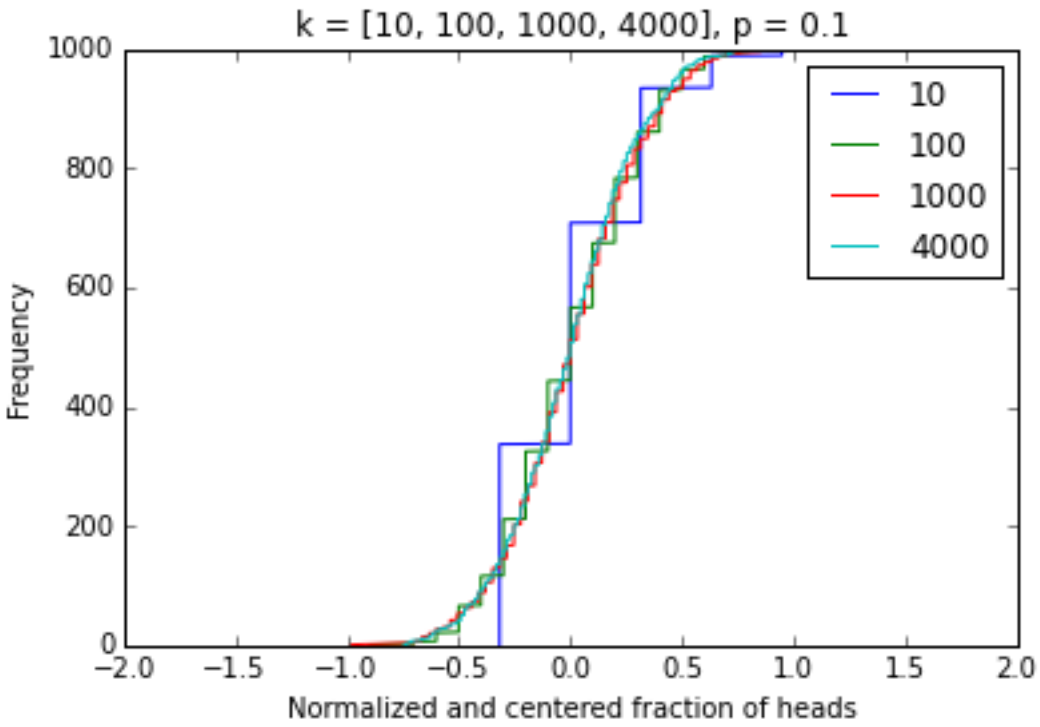
Now, let's see what the effect is of varying the bias of the coin. Repeat the previous two parts for $p = 0.1, 0.3, 0.5, 0.9$. You can assume the bin size is 0.1. You should have 4 pairs of plots.

Make sure that all of these are plotted on the same scale as the previous parts. What do you observe? Which $p$ seem more variable even after this $\sqrt{k}$ normalization? Less variable?

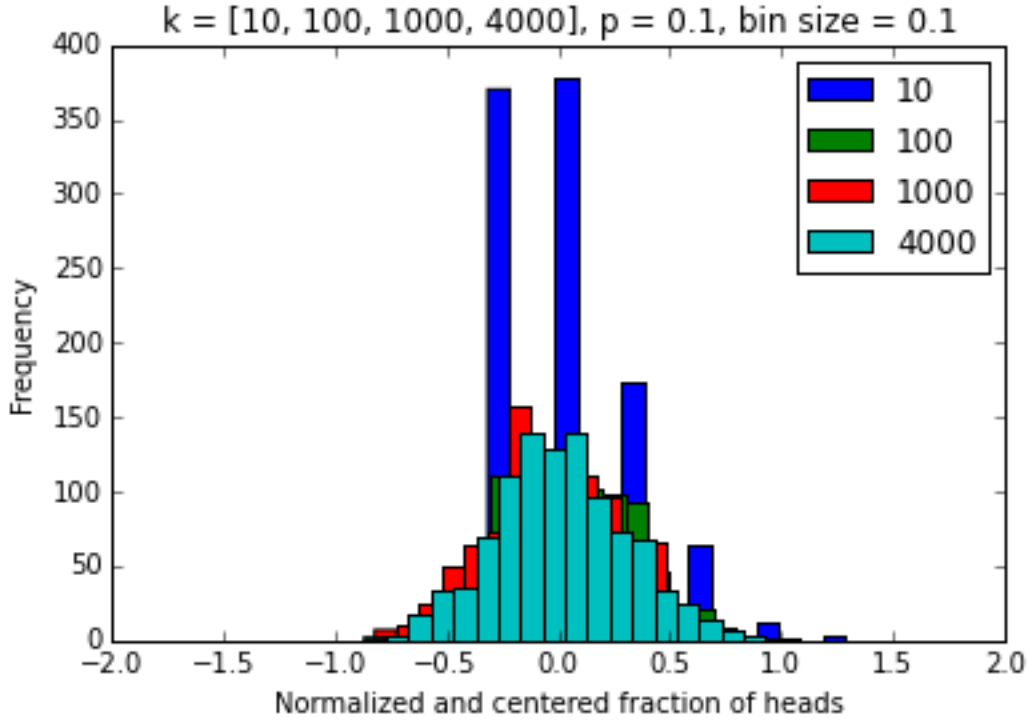
```
In [16]: def partH(pranges=[0.1,0.3,0.5,0.9], kranges=[10,100,1000,4000], m=1000):
             """
             Code for part (g)
             """

             print 'Question 2 part (h): (redo (f) and (g) for different p)'

             # YOUR CODE HERE
             # Make sure to reuse your code wisely...
             for p in pranges:
                 partF(p, kranges, m)
                 partG(p, kranges, m, bin_widths=[0.1])
```

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

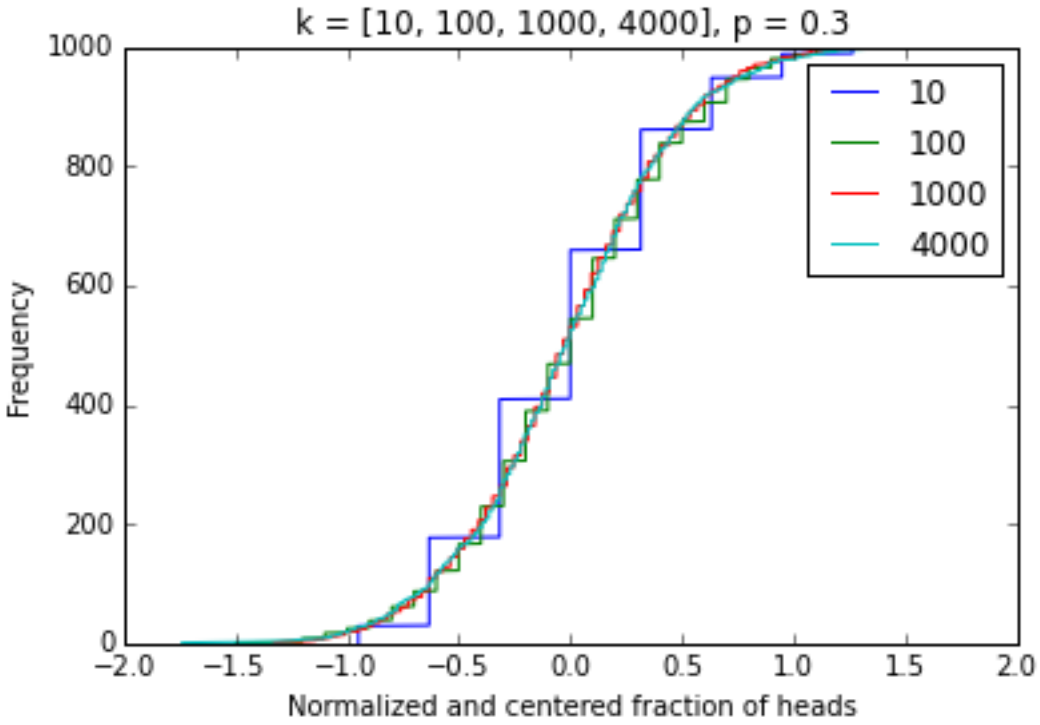
```
Question 2 part (h): (redo (f) and (g) for different p)
Question 2 part (f):
Probability of head: 0.100000
```



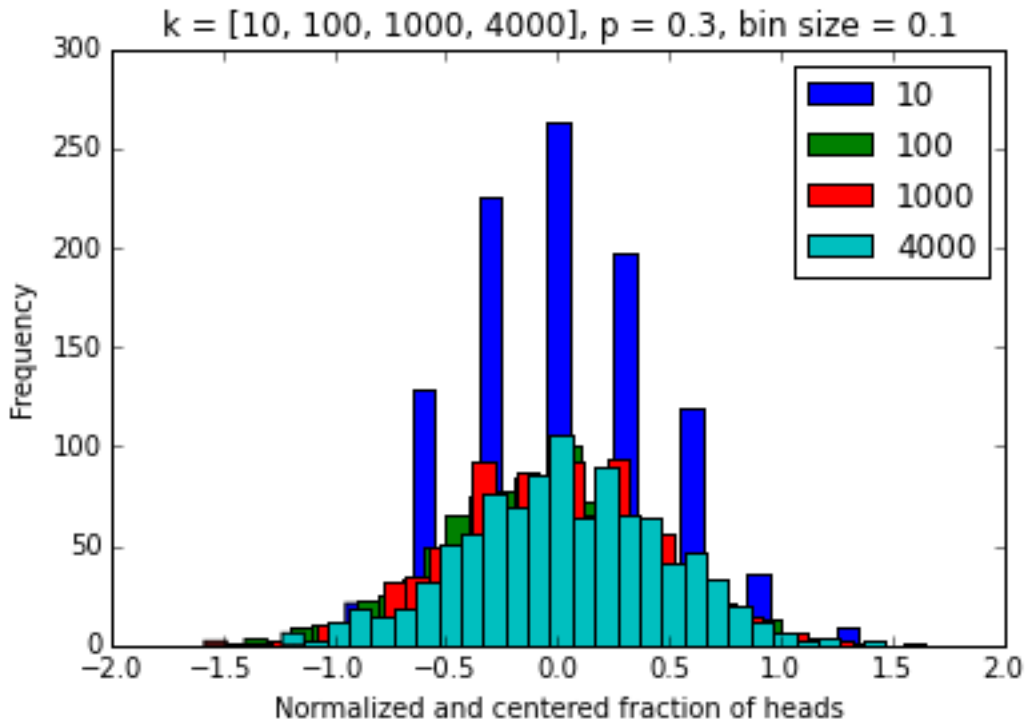

```
Question 2 part (g):
Probability of head: 0.100000
```



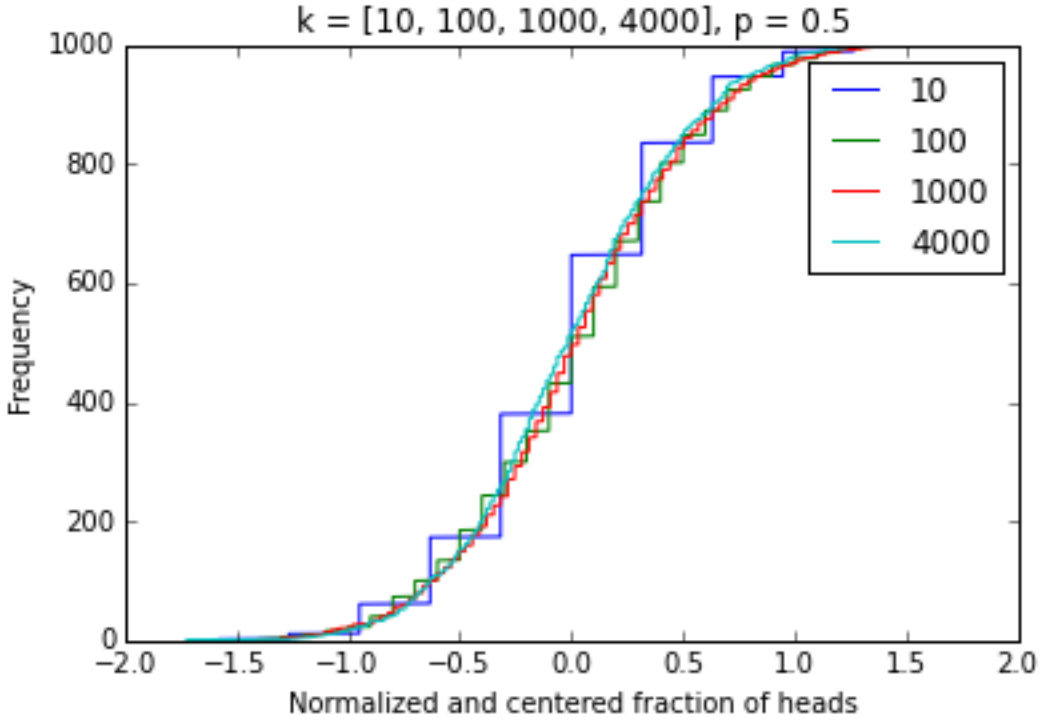
```
Question 2 part (f):
Probability of head: 0.300000
```



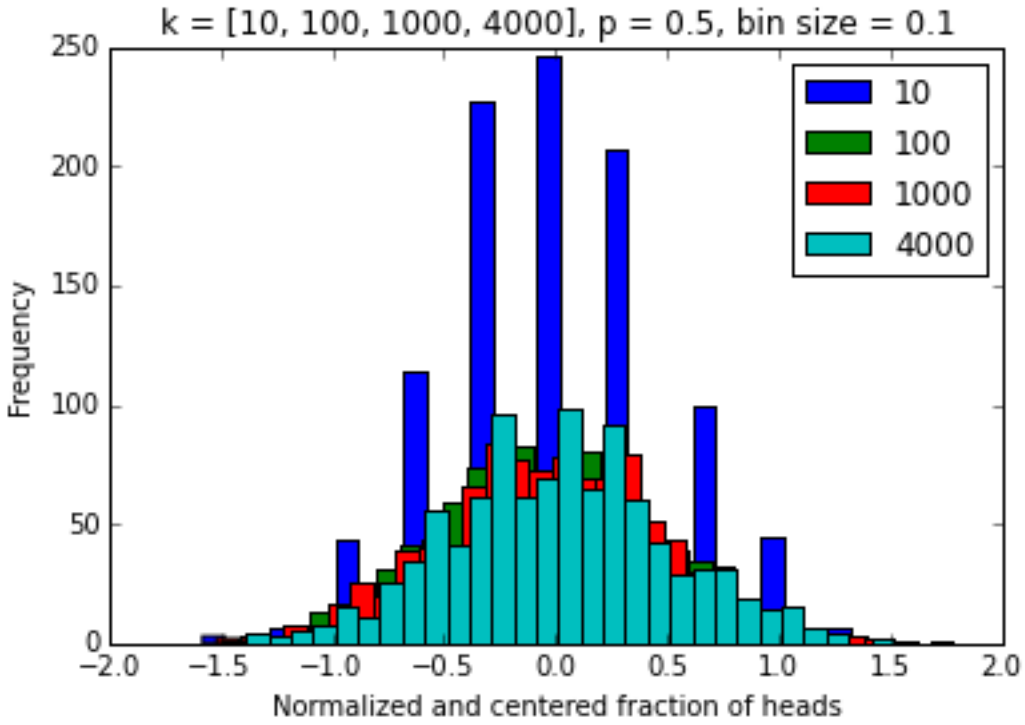

```
Question 2 part (g):
Probability of head: 0.300000
```



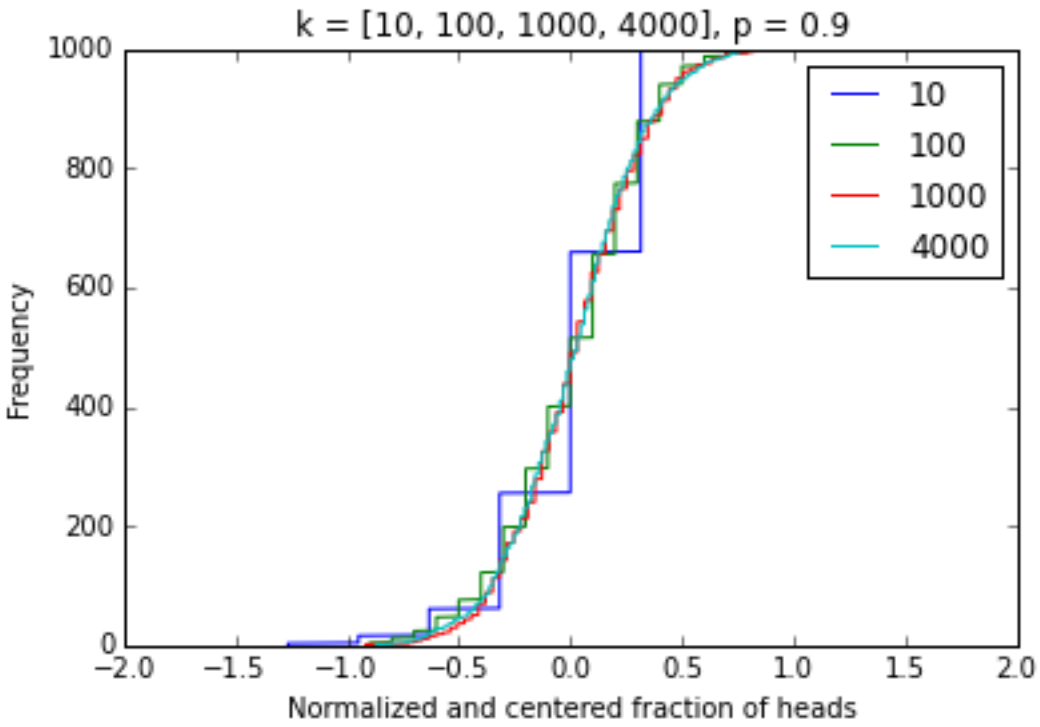

```
Question 2 part (f):
Probability of head: 0.500000
```



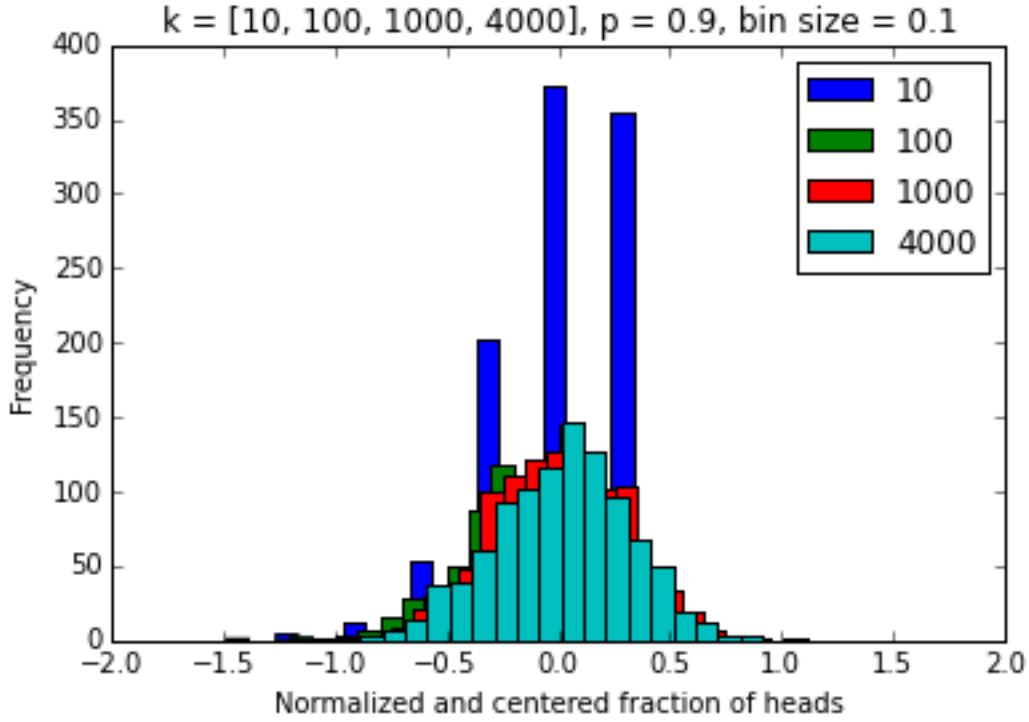
```
Question 2 part (g):
Probability of head: 0.500000
```




```
Question 2 part (f):
Probability of head: 0.900000
```




```
Question 2 part (g):
Probability of head: 0.900000
```



YOUR COMMENTS HERE:

## Part (i): Log Gap

Based on what you observe in the previous pattern, you decide to try and hunt out the actual dependence on the shape by looking at appropriate plots. As you did in part (f) of last week's lab, plot the Log of the gap between the 0.75 and 0.25 quartiles against $\ln p + \ln(1-p)$ in a scatter plot for $k = 4000$. (For this, you could also just plot the gap against $p(1-p)$ on a Log-Log plot.) What do you observe?

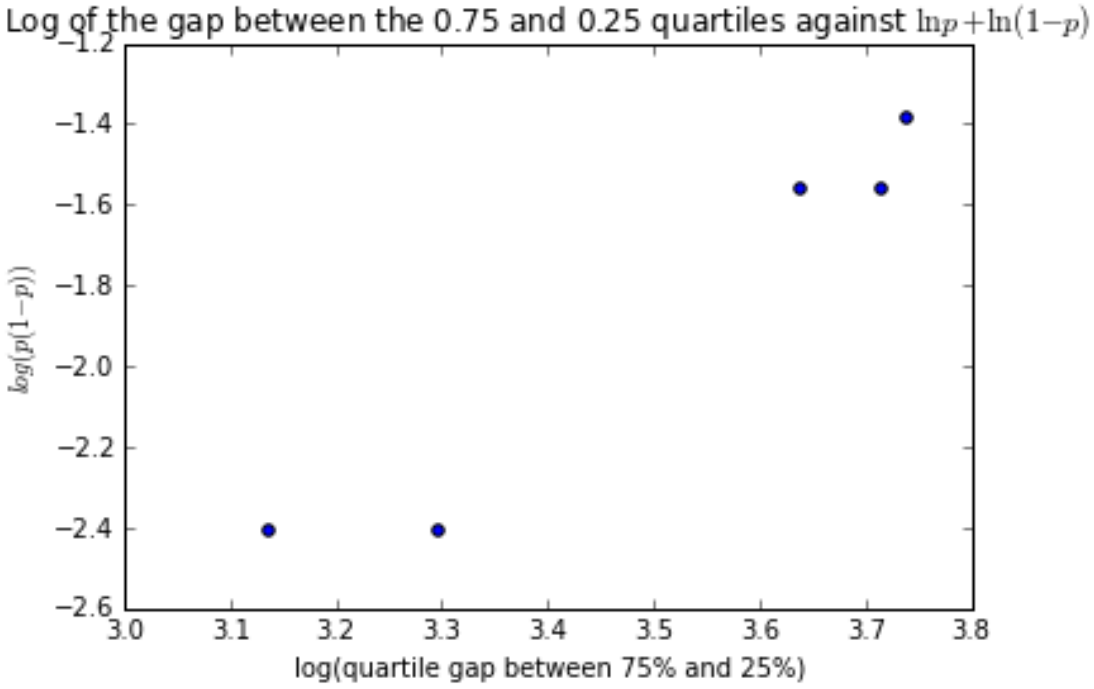
*Hint*: Implement the function `calculate_quartile_gap`, which takes a list containing the number of heads (the return value of `run_many_trials`) and calculates the gap between the first and third quartiles. Also, you can plot a scatter plot with `plt.scatter()`.

```
In [18]: def calculate_quartile_gap(results):
             """
             Calculates the gap between the first and third quartiles

             YOUR CODE HERE
             """

             results.sort()
             n = len(results)
             q1 = int(round(0.25*n))
             q3 = int(round(0.75*n))
             return results[q3]-results[q1]
```

```
In [34]: def partI(pranges=[0.1,0.3,0.5,0.7,0.9], k=4000, m=1000):
             """
             Code for part (i)
             """

             print 'Question 2 part (i)'

             # YOUR CODE HERE
             points = []
             for p in pranges:
                 print ('Calculating for p = %.1f'%p)
                 qgap = calculate_quartile_gap(run_many_trials(p, k, m))
                 points.append((math.log(qgap), math.log(p*(1-p))))
             plt.scatter([pt[0] for pt in points], [pt[1] for pt in points])
             plt.ylabel('$log(p(1-p))$')
             plt.xlabel('log(quartile gap between 75% and 25%)')
             plt.title('Log of the gap between the 0.75 and 0.25 quartiles against $\ln p + \ln(1-p)$')
             plt.show()
```

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

```
Question 2 part (i)
Calculating for p = 0.1
Calculating for p = 0.3
Calculating for p = 0.5
Calculating for p = 0.7
Calculating for p = 0.9
```




YOUR COMMENTS HERE:

## Part (j): Standard Deviation

This suggests another normalization. We call $\sqrt{p(1-p)}$ the "standard deviation" for a 0-1 coin toss with probability $p$ of being 1.

Plot a histogram of $\frac{S_k - kp}{\sqrt{k}\sqrt{p(1-p)}}$ for $k = 1000$ with bin size of 0.1. Do this again for $k = 10, 100, 4000$ and for $p = 0.1, 0.3, 0.5, 0.7, 0.9$ as before. You should have one plot for each value of $p$, so there should be 5 plots in total.

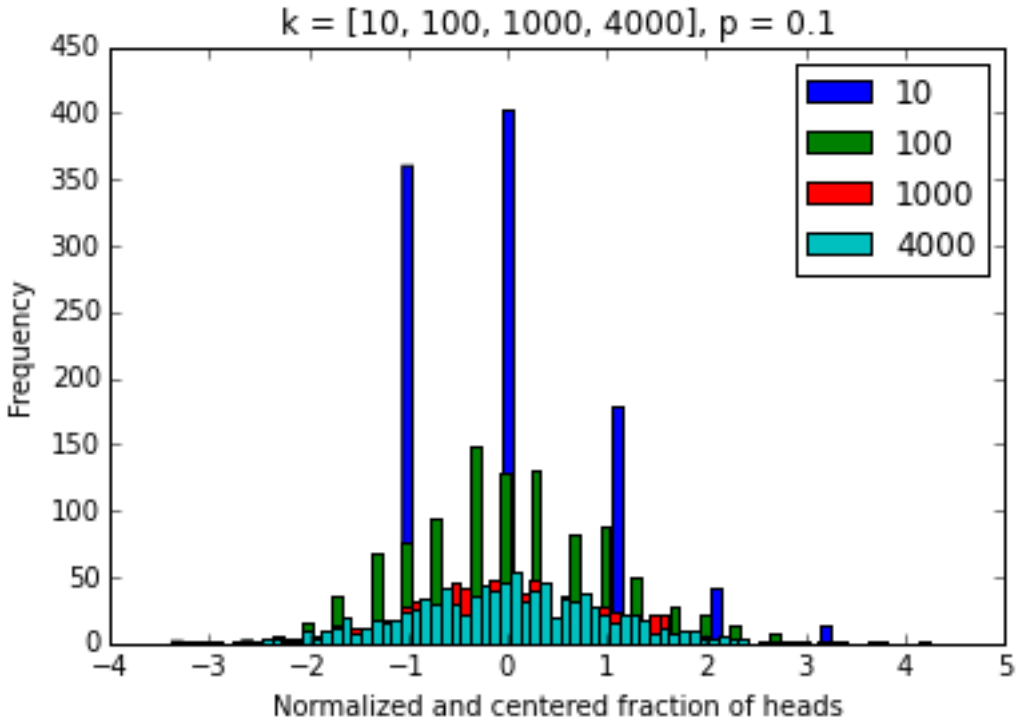
What do you observe?

```
In [21]: def partJ(pranges=[0.1,0.3,0.5,0.7,0.9], kranges=[10,100,1000,4000], m=1000):
             """
             Code for Q2 part (j)
             """

             print 'Question 2 part (j):'
             bin_width = 0.1

             for p in pranges:
                 print ('Probability of head p = %.1f'%p)

                 # YOUR CODE HERE
                 std = math.sqrt(p*(1-p))
                 results = {}
                 for k in kranges:
                     results[k] = [(Sk -k*p)/(math.sqrt(k)*std) for Sk in run_many_trials(p, k, m)]
                     plt.hist(results[k], bins=np.arange(min(results[k]), max(results[k])+bin_width*2,
                              label=str(k), histtype='barstacked')

                 # END YOUR CODE HERE
```

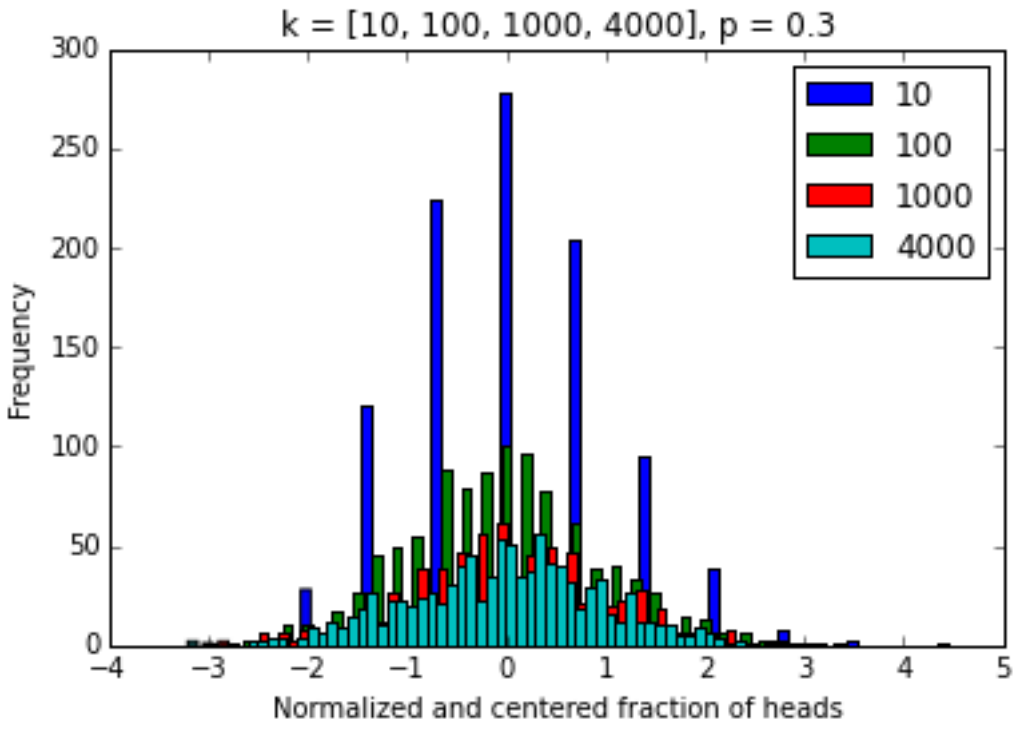
```
        plt.legend()
        plt.ylabel('Frequency')
        plt.xlabel('Normalized and centered fraction of heads')
        plt.title('k = %s, p = %.1f' % (str(kranges), p))
        plt.show()
```

```
In [22]: partJ()
```

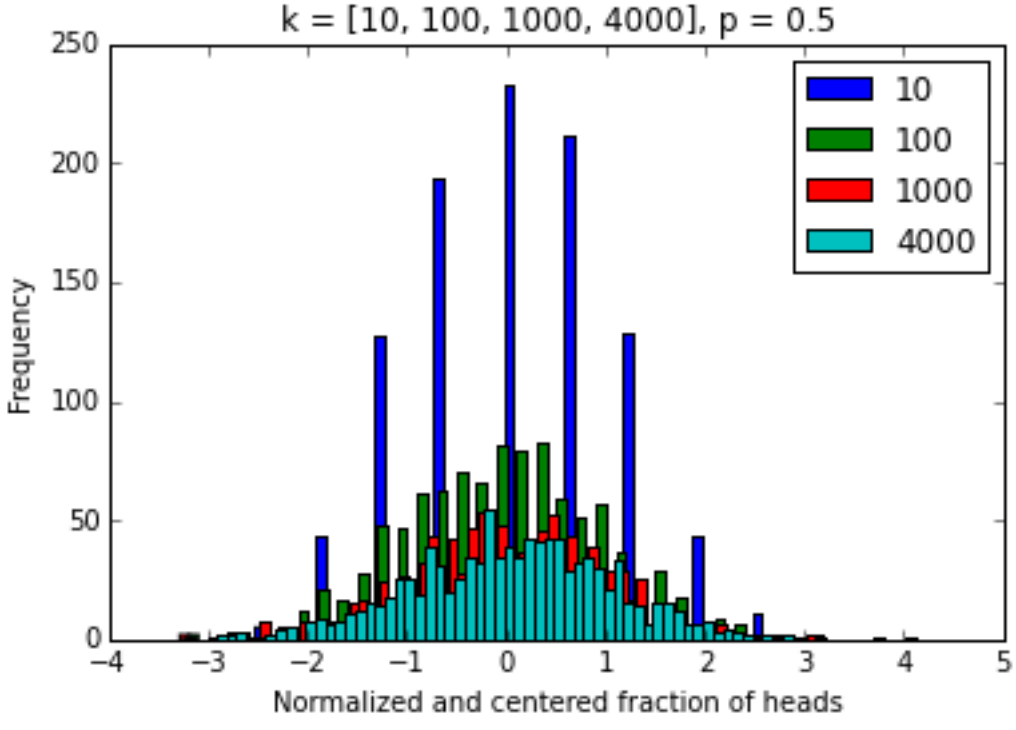
```
Question 2 part (j):
Probability of head p = 0.1
```



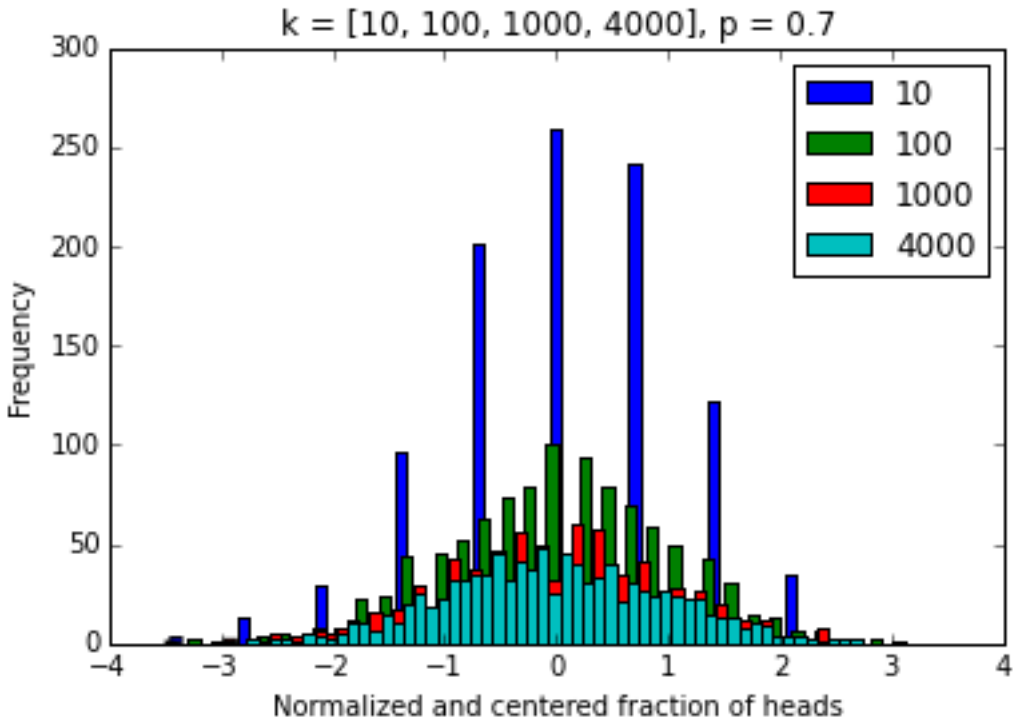

```
Probability of head p = 0.3
```



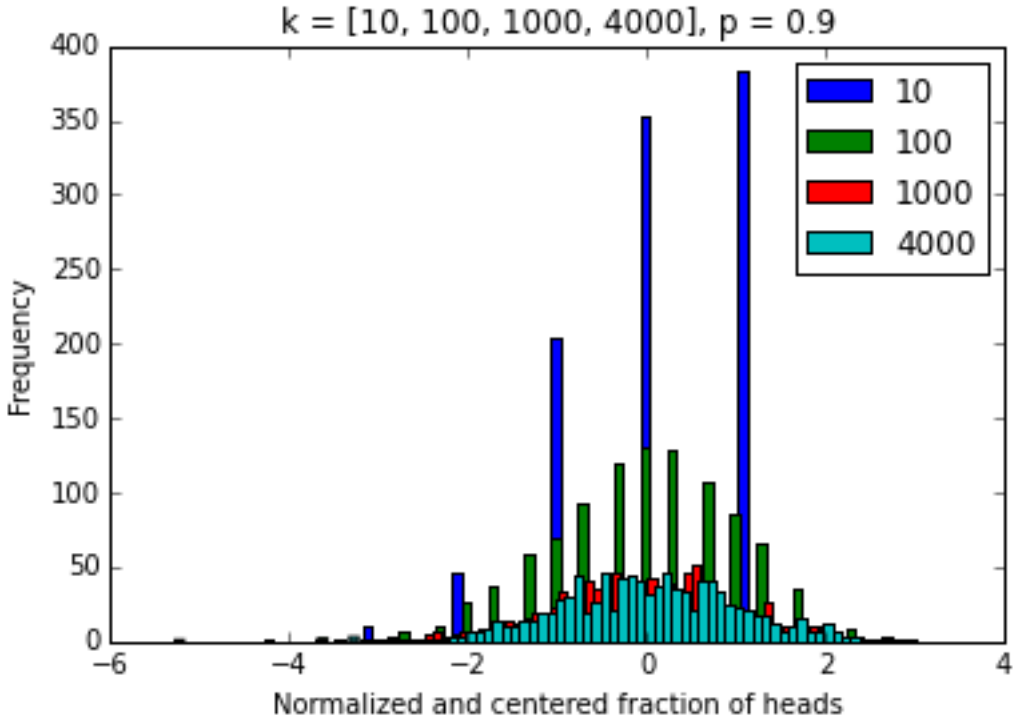

```
Probability of head p = 0.5
```



```
Probability of head p = 0.7
```




```
Probability of head p = 0.9
```



YOUR COMMENTS HERE:

## Part (k): Normalized q-curve Redux

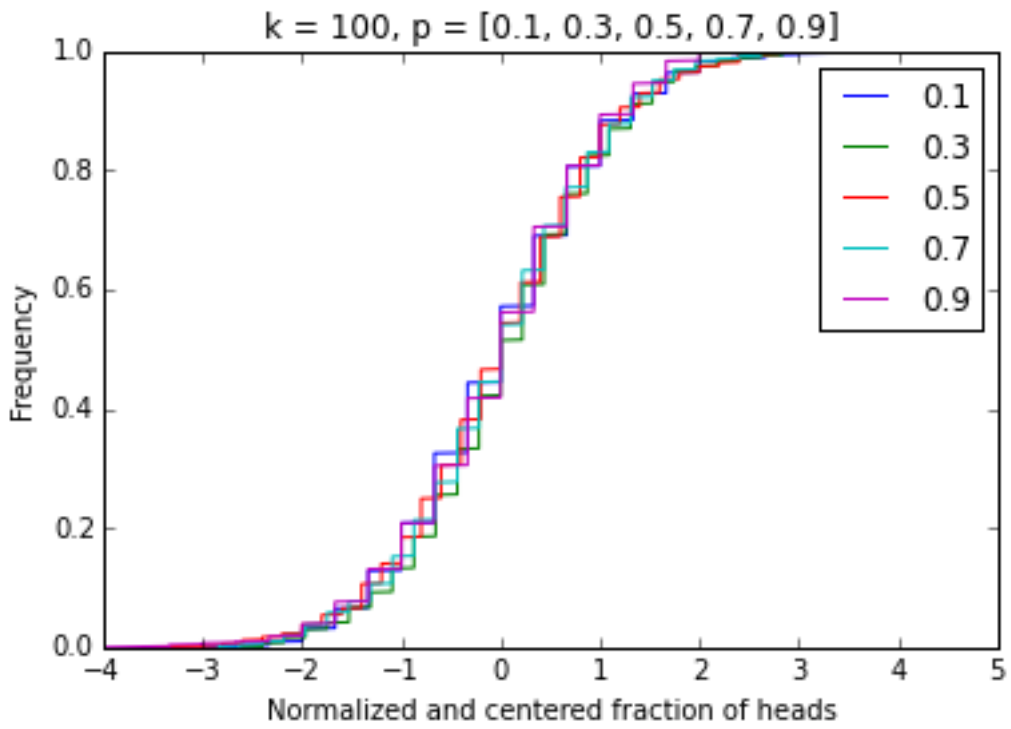
Use the same normalization as the previous part and make the cliff-face plots corresponding to part (d) of last week's lab. To be precise, look at the range $d = -3$ to $d = +3$ and plot how often (out of the $m$ runs — as a fraction between 0 and 1) $\frac{S_k - kp}{\sqrt{k}\sqrt{p(1-p)}}$ is less than $d$.

This should be an increasing curve. Here, put all the different $p$ plots together but have three different plots for $k = 100, 1000, 4000$. (You already know from earlier plots that the different $k$'s track each other.)

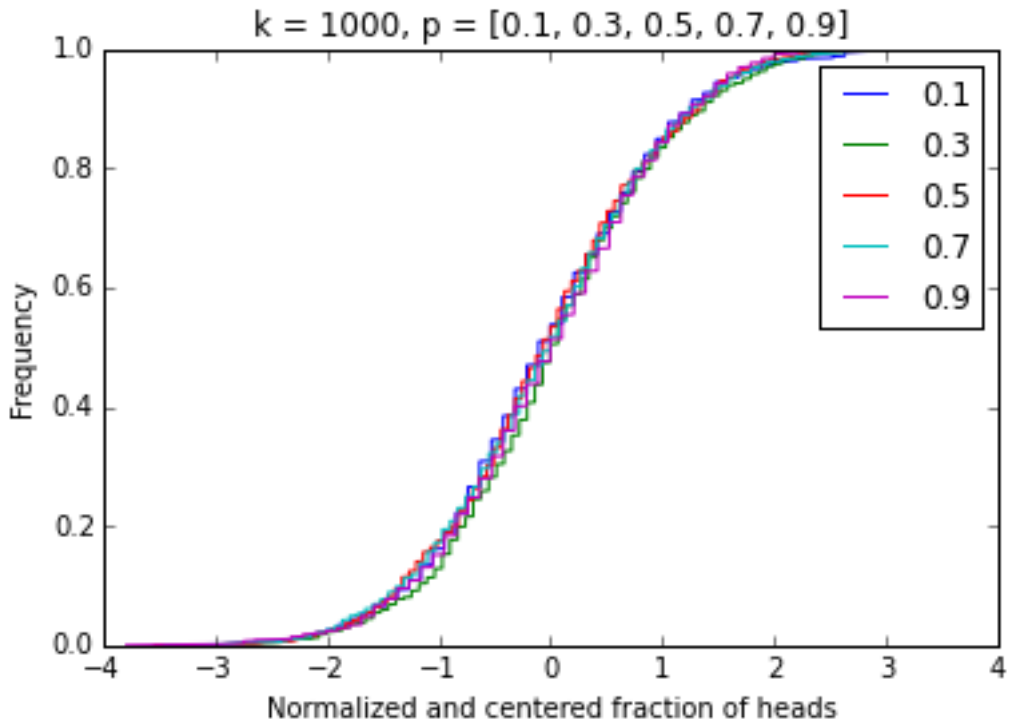
```
In [23]: def partK(pranges=[0.1,0.3,0.5,0.7,0.9], kranges=[100,1000,4000], m=1000):
             """
             Code for Q2 part (k)
             """

             print 'Question 2 part (k):'

             for k in kranges:
                 plt.clf()

                 # YOUR CODE HERE
                 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 run_many_trials(p, k, m)]
                     results[p].sort()
                     plt.plot(results[p], np.linspace(0, 1, m), label=str(p))

                 # END YOUR CODE HERE
                 plt.legend()
                 plt.ylabel('Frequency')
                 plt.xlabel('Normalized and centered fraction of heads')
                 plt.title('k = %i, p = %s' % (k, str(pranges)))
                 plt.show()
```

```
In [24]: partK()
```

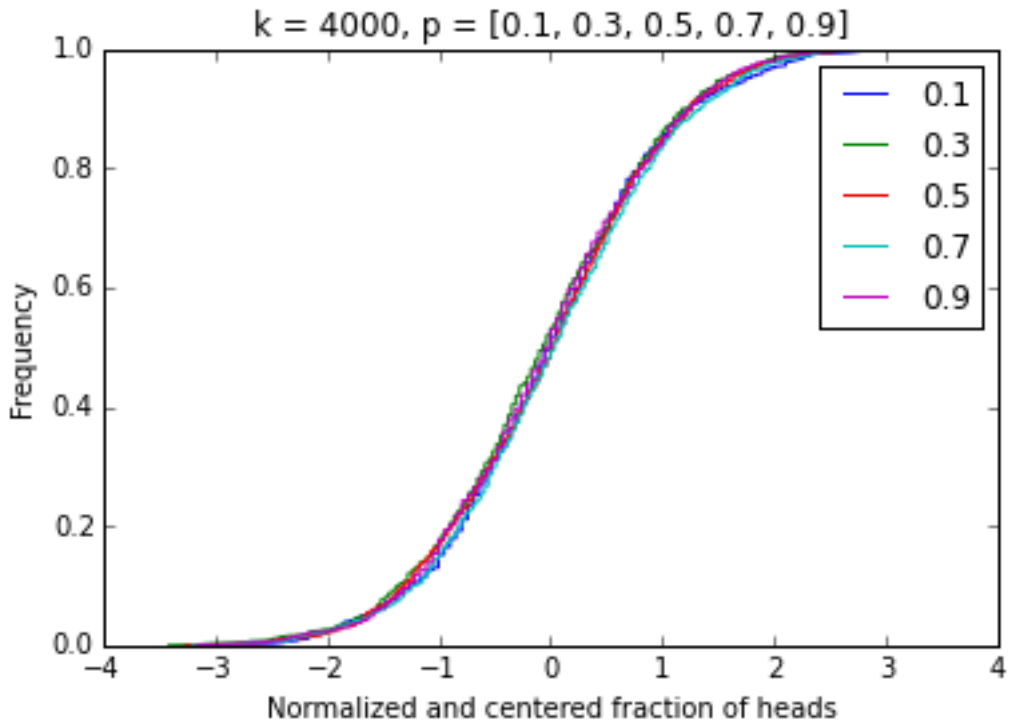
```
Question 2 part (k):
Number of trials k = 100
Probability of head p = 0.1
Probability of head p = 0.3
Probability of head p = 0.5
Probability of head p = 0.7
Probability of head p = 0.9
```




```
Number of trials k = 1000
Probability of head p = 0.1
Probability of head p = 0.3
Probability of head p = 0.5
Probability of head p = 0.7
Probability of head p = 0.9
```




```
Number of trials k = 4000
Probability of head p = 0.1
Probability of head p = 0.3
Probability of head p = 0.5
Probability of head p = 0.7
Probability of head p = 0.9
```




## Part (1): $n$ choose $k$

Lastly, we would like to explore a function that defines how many ways you can choose $k$ distinct objects out of $n$ possible objects. This is written as $\binom{n}{k}$ and is read aloud as "$n$ choose $k$". We define $\binom{n}{k} = \frac{n!}{k!(n-k)!}$. For example, $\binom{5}{3} = \frac{5!}{3!(5-3)!} = \frac{5 \cdot 4 \cdot 3 \cdot 2 \cdot 1}{(3 \cdot 2 \cdot 1) \cdot (2 \cdot 1)} = \frac{120}{12} = 10$. We wish to explore this function. Plot the value $\binom{50}{k}$ on the $y$-axis and $k$ on the $x$-axis for $0 \leq k \leq 50$.

Does this constantly grow as $k$ gets larger? What does the shape of the graph remind you of?

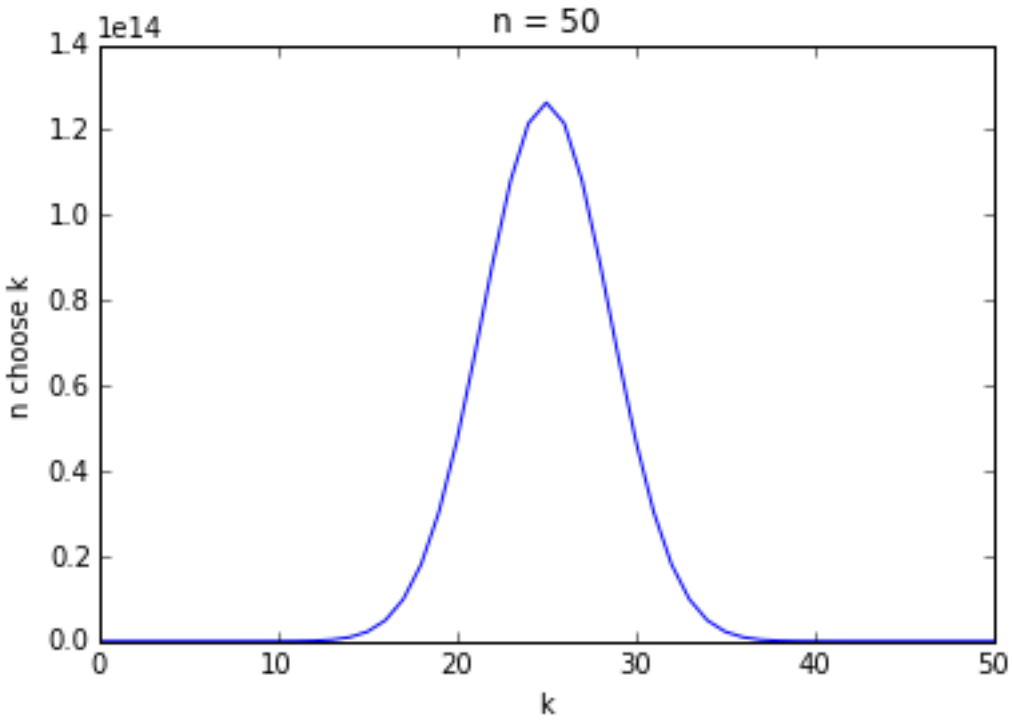
*Hint*: Implement the function `choose(n, k)` using `math.factorial`.

```
In [25]: def choose(n, k):
             """
             Computes n choose k
             """

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

             # YOUR CODE HERE
             return math.factorial(n)/(math.factorial(k)*math.factorial(n-k))
```

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

```
In [26]: choose(5, 3) == 10

Out[26]: True

In [27]: choose(17, 1) == 17

Out[27]: True
```

```
In [28]: def partL(n=50):
             """
             Code for Q2 part (l)

             YOUR CODE HERE
             """

             plt.plot(range(0,n+1), [choose(n, k) for k in xrange(0, n+1)])
             plt.xlabel('k')
             plt.ylabel('n choose k')
             plt.title('n = %i' % n)
             plt.show()

In [29]: partL()
```




YOUR COMMENTS HERE:

**Congratulations**! You are done with Virtual Lab 10.

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.