Distinct element estimation using k-th min

In the lecture, we studied the algorithm named Idealized F_0 estimation (slide 19). The algorithm uses a random hash function to map elements from the stream to float values between 0 and 1. Ultimately, it maintains the smallest hash value V and outputs $\frac{1}{V} - 1$ as the estimate $\tilde{F}_0$ for the number of distinct elements.

This algorithm uses the idea that the expected value of the smallest hash value is $\frac{1}{F_0+1}$, where F_0 is the number of distinct elements. In fact, we can generally use the k-th smallest hash value V_k for $k = 1, 2, \ldots$. We will use the results from exercise 4 to conduct experiments to see how different k values affect the accuracy of your estimate.

[Optional]: Let m be the length of the stream. You can maintaining the k-th smallest element in an unsorted list in time $O(m \log k)$ using min heap, see https://docs.python.org/3/library/heapq.html.

In [146...

```
# Import packages needed.
import random, math
import numpy as np
import matplotlib.pyplot as plt
```

To test the effect of k, we must first implement a function that takes a data sequence, hash each element to a value between 0 and 1, and returns the k-th smallest hash value. Python has a built-in hash function hash() that takes any hashable object and returns an integer hash. To convert a hash value to a float, use modular the hash with a large int and divide by it, for instance, $MAXINT = 2^{63} - 1$.

In [147...

```
import sys
MAXINT = sys.maxsize
```

In [148...

```
def kth_smallest_hash_value(input_list, k):
#     Write your code here
    lst = []
    for x in range(0, len(input_list)):
        hashedVal = hash(input_list[x])
        hashedVal = (hashedVal % MAXINT)/MAXINT
        lst.append(hashedVal)
    lst.sort()
    value = lst[k-1]
    return value
```

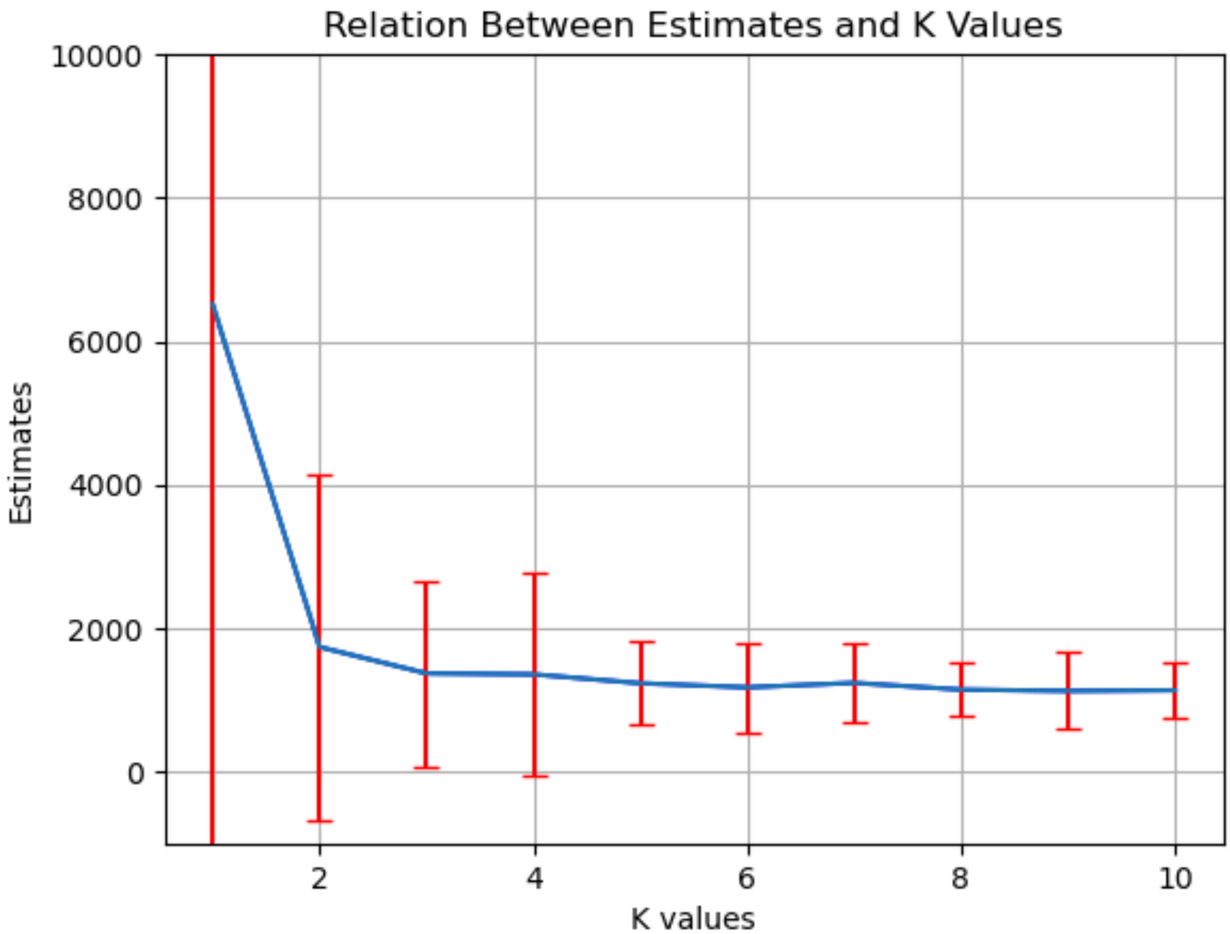
Now let us test k values between 1 to 10. For each k, we will generate a list of 1000 random **strings** using `str(random.uniform(0,100))`, and estimate its cardinality via the returned value from the function `kth_smallest_hash_value` you implemented. For each k, repeat this process 100 times and record the average and std of the estimates. Finally, generate a plot with error bars to show the relation between estimates and k values. Note that the std for small k can be very large, so you may need to set plt.ylim(-1000, 10000) to cap the y-axis for better visualization.

In [156…

```
# Write your code here

average = []
std = []
k_values = []
for k in range(1, 11):
    k_values.append(k)
    lowValue = 0
    lowValueArray = []
    for x in range(0,100):
        ranString = []
        for y in range(0,1000):
            ranString.append(str(random.uniform(0,100)))
        lowValue = kth_smallest_hash_value(ranString,k)
        lowValue = k/lowValue - 1
        lowValueArray.append(lowValue)
    averageLowValue = np.mean(lowValueArray)
    average.append(averageLowValue)
    std.append(np.std(lowValueArray))

plt.plot(k_values, average, color = "blue")

plt.grid()
plt.xlabel("K values")
plt.ylabel("Estimates")
plt.title("Relation Between Estimates and K Values")
plt.ylim(-1000,10000)
plt.errorbar(k_values,average,yerr = std, ecolor = "red", capsize = 4)
```

Out[156]:

```
<ErrorbarContainer object of 3 artists>
```




```
In [ ]:
```

The median trick useful technique (slide 13)

Please implement the function `median_trick` below.

In [158…

```
def median_trick(generator, expectation, var, eps, delta):
    '''
    Input:
        generator - a function that generates one sample from a distribution
        expectation - Expectation of the distribution
        var - Variance of the distribution
        eps - epsilon (accuracy parameter) as defined in slide 13
        delta - delta (confidence parameter) as defined in slide 13
    Output:
        estimated value Q
    '''
    # Write your code here
    t_val = math.ceil(math.log(1/delta,10))
    k_val = math.ceil(var/(expectation**2 * eps**2))

    averages = []
    for x in range(t_val):
        avg = []
        for y in range(k_val):
            number = generator()
            avg.append(number)
        averages.append(np.mean(avg))

    return np.median(averages)
```

Now we want to test the function with the following idea. Assume Q=2. The unbiased estimator, X of Q, generates estimates that follow a normal distribution with variance equal to 1. The generator for X is already given below as `normal_generator`. Please generate two plots below.

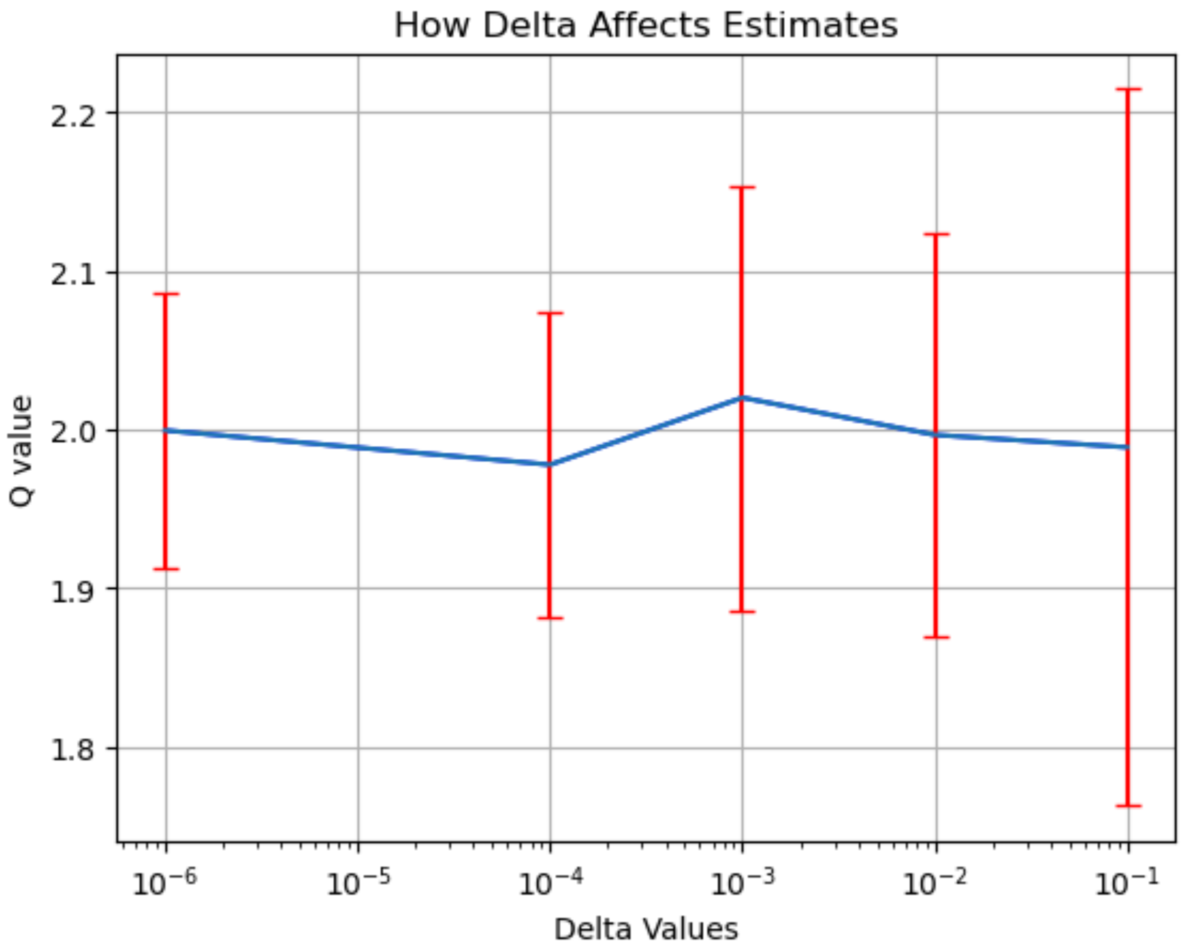
- Set eps=0.1, and test how the delta affects the estimates. Range delta in [1e-6, 1e-4, 1e-3, 0.01, 0.1]; repeat the estimation 100 times for each delta value. Generate a plot with std as error bars to show how the average estimates change as the delta changes.
- Set delta=0.1, and test how the epsilon affects the estimates. Range epsilon in [0.01, 0.02, 0.05, 0.1, 0.2]; repeat the estimation 100 times for each epsilon value. Generate a plot with std as error bars to show how the average estimates change as the epsilon changes.

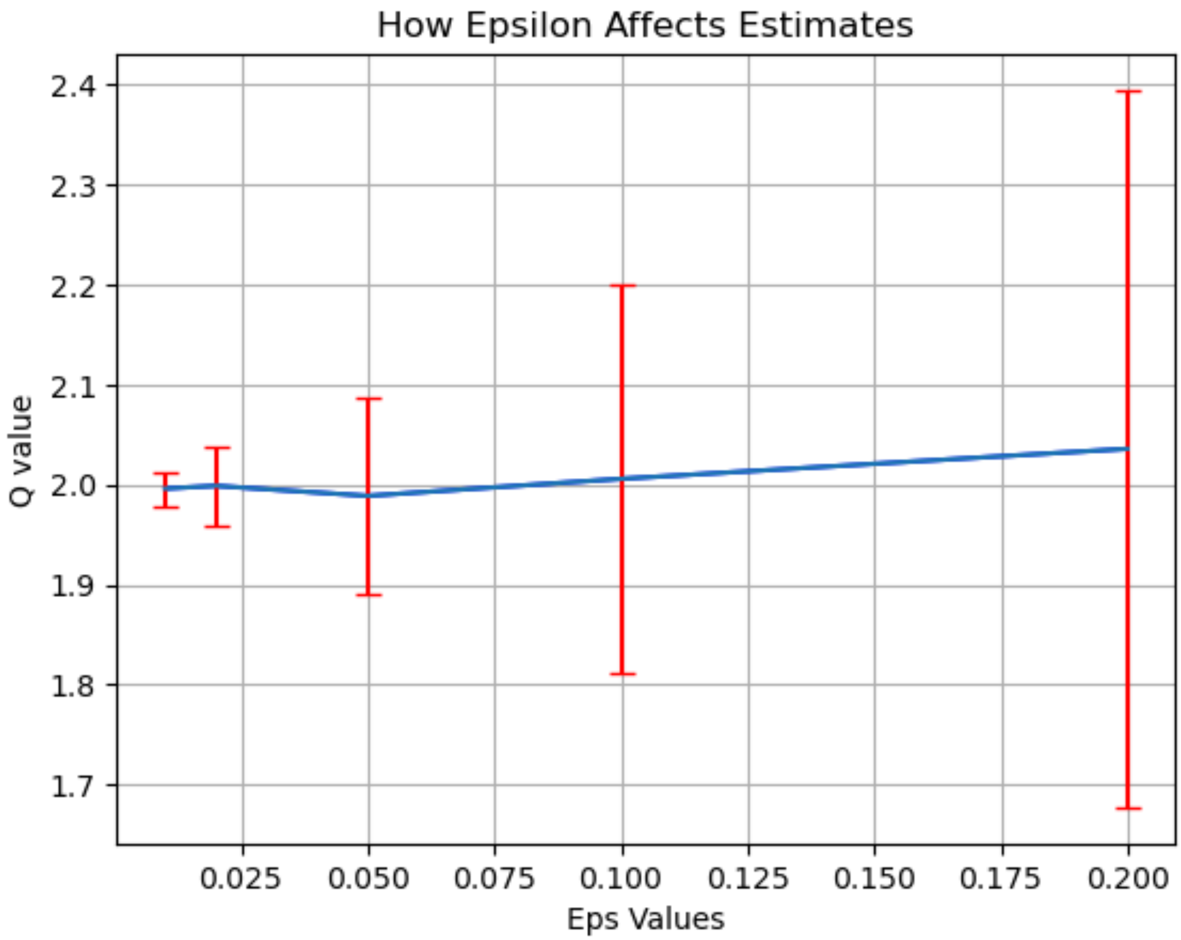
In [151…

```
# Don't change
def normal_generator():
    return np.random.normal(2,1)
```

In [159…

```
# Write your code here
eps = 0.1
delta = [1e-6, 1e-4, 1e-3, 0.01, 0.1]
delta_average_lst = []
delta_std_lst = []

for x in delta:
    delta_lst = []
    for z in range(100):
        y = median_trick(normal_generator, 2, 1, eps, x)
        delta_lst.append(y)
    delta_average_lst.append(np.mean(delta_lst))
    delta_std_lst.append(np.std(delta_lst))

plt.plot(delta, delta_average_lst, color = "blue")
plt.grid()
plt.xscale('log')
plt.ylabel("Q value")
plt.xlabel("Delta Values")
plt.title("How Delta Affects Estimates")
plt.errorbar(delta,delta_average_lst,yerr = delta_std_lst, ecolor = "red", c
plt.show()



delta = 0.1
eps = [0.01, 0.02, 0.05, 0.1, 0.2]
eps_average_lst = []
eps_std_lst = []

for x in eps:
    eps_lst = []
    for z in range(100):
        y = median_trick(normal_generator, 2, 1, x, delta)
        eps_lst.append(y)
    eps_average_lst.append(np.mean(eps_lst))
    eps_std_lst.append(np.std(eps_lst))

plt.plot(eps, eps_average_lst, color = "blue")
plt.grid()
plt.ylabel("Q value")
plt.xlabel("Eps Values")
plt.title("How Epsilon Affects Estimates")
plt.errorbar(eps,eps_average_lst,yerr = eps_std_lst, ecolor = "red", capsize
plt.show()
```


How Delta Affects Estimates
Q value
2.2
2.1
2.0
1.9
1.8
Delta Values
10^{-6}
10^{-5}
10^{-4}
10^{-3}
10^{-2}
10^{-1}



Morris Algorithm (slide 45)

Morris algorithm maintains a counter c that, for every element in the stream, itself increments by 1 with probability $\frac{1}{2^c}$. In the end, it outputs an estimate as $2^c - 1$.

In this section, we will change the base of this counter (slide 51). Instead of using 2 only, we use any base $1 + \alpha$. We now increase the counter c with probability $\frac{1}{(1+\alpha)^c}$. First, let us implement the function `morris_update_base_alpha` below. **This function is called whenever we see an element from the stream to update the counter.**

In [153…

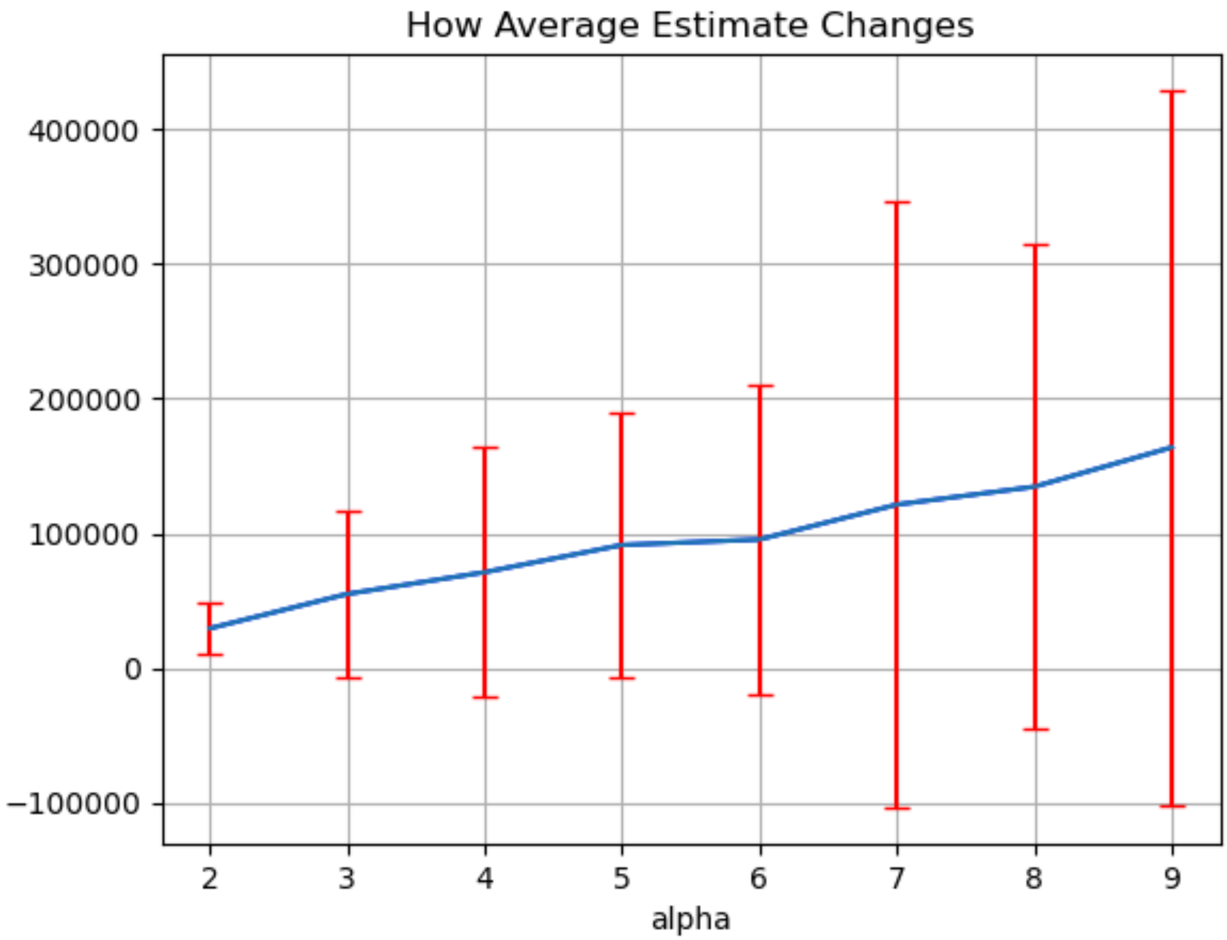
```
def morris_update_base_alpha(counter, alpha):
    '''
    Input:
        counter - current value of counter c
        alpha - as defined in slide 51 alpha
    Output:
        updated value of counter c
    '''
    denominator = (1+alpha)**counter
    numerator = 1
    value = numerator/denominator


    random_n = random.random()
    if(random_n < value):
        return counter + 1
    else:
        return counter
```

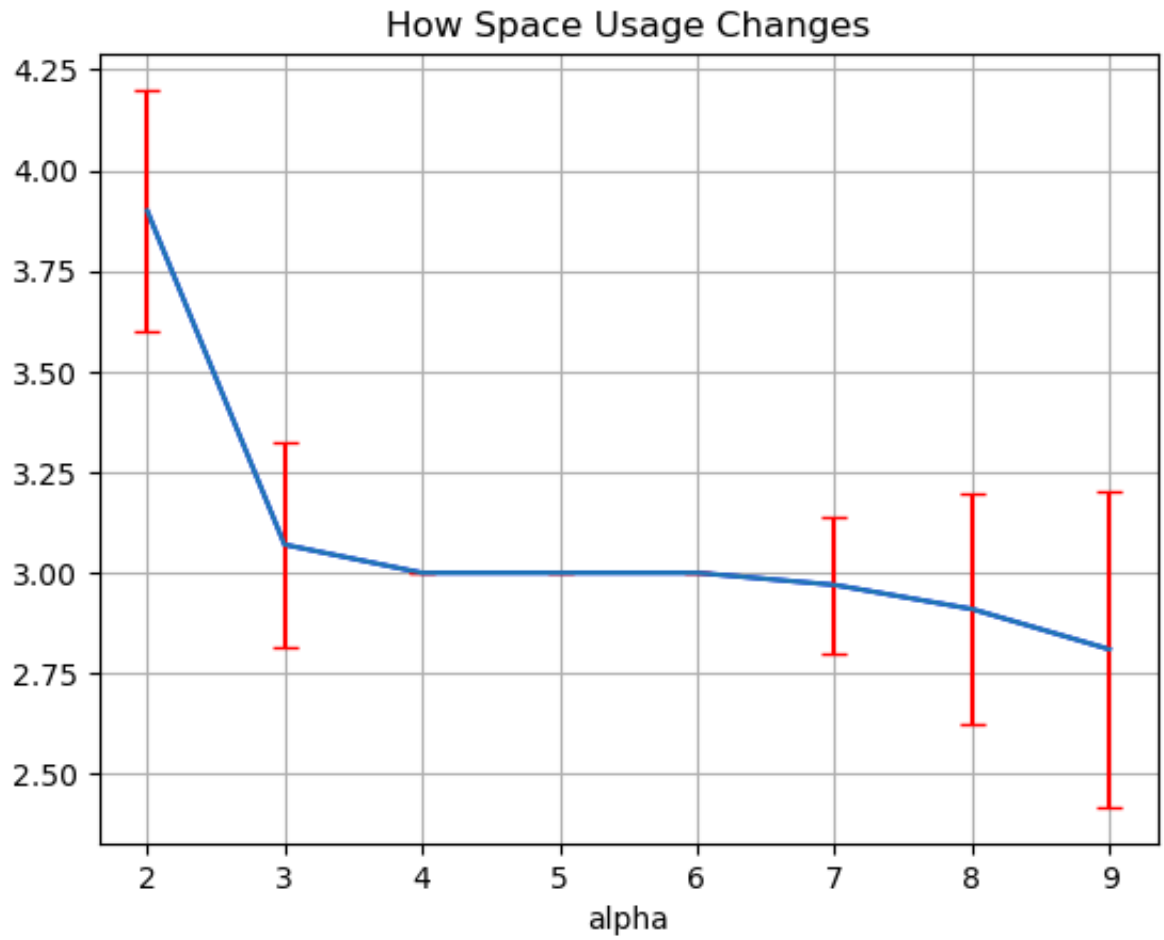
Now let us test the function with the edge list file "soc-hamsterster.edges" in the same folder. Reading the file line by line in python can generate a stream of strings. Counting the number of strings/lines in this file tells us the number of edges of this "soc-hamsterster" graph. Let us try different alpha values ranging from 2 to 9. Again, for each alpha, estimate the number of lines in the edge list file using the morris algorithm (the key component of which is `morris_update_base_alpha`), and repeat this 100 times. Besides, check how many bits are needed to maintain the counter via `math.ceil(math.log(counter, 2))` at the end of each estimation. Finally, generate two plots with std as error bars to show

- How the average estimate changes as the alpha value increases.
- How the space usage (in bits) changes as the alpha value increases.

In [167…

```
file = open('soc-hamsterster.edges','r')
count = 0
line_count = 0
for line in file:
    line_count += 1
    words = line.split()
    count += int(words[1])

alpha = []
average_estimate = []
std_estimate = []

space_average_estimate = []
space_std_estimate = []

for a in range(2, 10):
    alpha.append(a)
    value_lst = []
    space_lst = []

    for y in range(100):
        counter = 0
        for x in range(line_count):
            counter = morris_update_base_alpha(counter,a)

        value = (1+a)**counter - 1
        value_lst.append(value)

        space = math.ceil(math.log(counter,2))
        space_lst.append(space)



    average_estimate.append(np.mean(value_lst))
    std_estimate.append(np.std(value_lst))

    space_average_estimate.append(np.mean(space_lst))
    space_std_estimate.append(np.std(space_lst))

plt.plot(alpha, average_estimate, color = "blue")
plt.grid()
plt.xlabel("alpha")
plt.title("How Average Estimate Changes")
plt.errorbar(alpha,average_estimate,yerr = std_estimate, ecolor = "red", cap
plt.show()

plt.plot(alpha, space_average_estimate, color = "blue")
plt.grid()
plt.xlabel("alpha")
plt.title("How Space Usage Changes")
plt.errorbar(alpha,space_average_estimate,yerr = space_std_estimate, ecolor
plt.show()
```


How Average Estimate Changes
400000
300000
200000
100000
0
−100000
2
3
4
5
6
7
8
9
alpha


How Space Usage Changes
4.25
4.00
3.75
3.50
3.25
3.00
2.75
2.50
2
3
4
5
6
7
8
9
alpha

```
In [ ]:
```