# Assignment 3; Counting Bloom Filters

November 3, 2019

## 1 CS110 Fall 2019

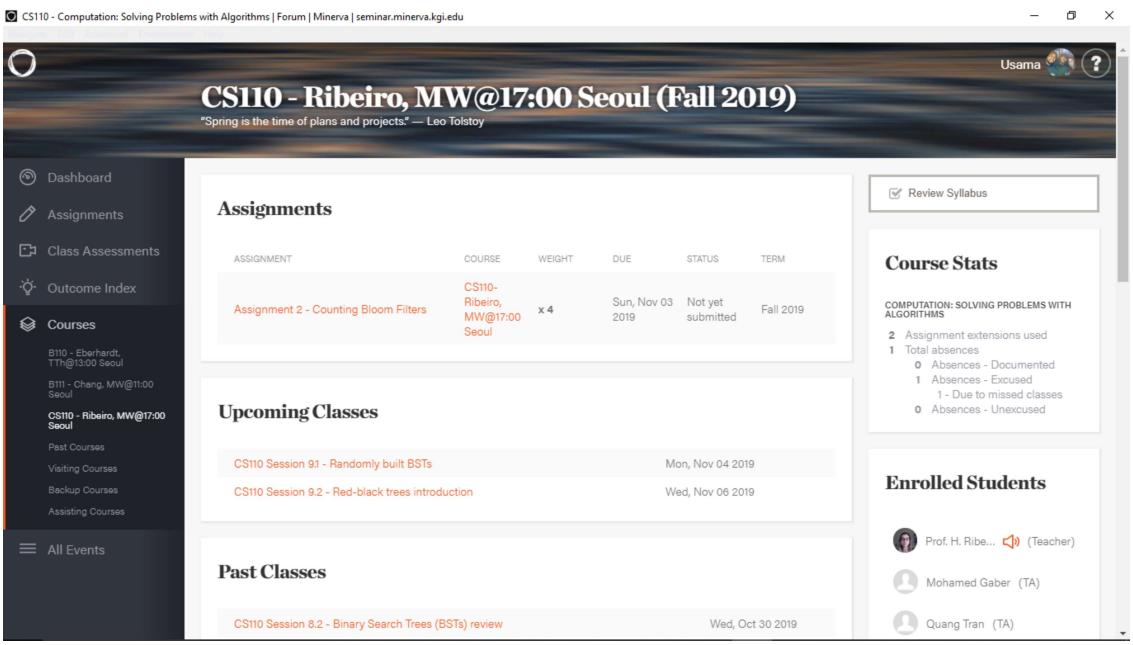
### 1.1 Assignment 3: Counting Bloom Filters

### 1.2 Question 1. [HCs #responsibility and #professionalism]

Take a screenshot of your CS110 dashboard on Forum where the following is visible: 1. your name. 2. your absences for the course have been set to excused up to session 7.2 (inclusively).

```
In [57]: from IPython.display import Image
         Image("Screenshot.png")
```

Out[57]:



## 1.3 Question 2. [#ComputationalSolutions, #DataStructures, #ComplexityAnalysis]

### 1.3.1 Give an overviewof the functionality of CBFs, listing all its operations (i.e., from initialize to indexing keys with a given false-positive rate). Give a few examples of computational applications that, in your consideration, can benefit from using CBFs. For every computational application, give meaningful explanations as to why it is a good candidate for CBF.

## 1.4 Definition: Bloom FIlters:

A Bloom FIlter is a very space efficient probablistic data structure that is used to test whether an object x is a member of a larger set U or not. Despite being highly space efficient it has some possibility of getiing a false positive because of collisions happening in the hash tables. Getting a false positive means that it will sometimes return that an object is "possibly in the set" while it isn't but it is very efficient in determining if the object is "definetely not in the set." So, to limit the probablility of getting these errors we use multiple hash functions such that when we want to enter an element X1 into our filter we hash that element from different arbitrary hash functions and for all the outputs we get form these hash functions we change the value in the bloom filter from 0 to 1. To get a more deeper insight, lets consider an example:

### 1.4.1 Example:

Suppose we have an element y1 (that can be a sequence of strings or an integer). Now we want to check whether this element is already present in our Bloom Filter which is made on universal set U. So to check this we will hash y1 with all the hash functions inside that bloom filter and check wheter all the corresponding values in the filter are 1. In case they all are one, Bloom Filter will tell us that the element is "probabilly in the set U". However, if even one of the corresponding filters is 0, Bloom Filter tells us that the element is "definitely not in the set U."

### 1.4.2 Functions and Characteristics of Bloom Filters:

A Normal or commonly known as Vanila Bloom Filter consists of two functions: 1. Add 2. Query

Normal Bloom Filters do not support delete function. If we were to try to delete an element from the Bloom Filter, and reset the corresponding values in the filter to, we might create a collision as other elements,that we don't want to delete, may also hash to those same bits, and while quering other elements we may get wrong answers even if the element is in the universal set. when it comes to complexity analyses Bloom filters have the following charachteristics:

1. Constant time to hash when an element is inserted which depends on the number of hash sunctions we choose for the filter
2. As We choose to initialize a Bloom Filter of a particular size, and do not change that size afterwards. So Bloom filter has a constant space as well

## 1.5 Counting Bloom FIlter:

The Counting Bloom Filter is a particular type of Bloom Filter having the following characteristics: 1. The counting bloom filter increments a bucket for every value that is hashed to it so it has positive integer valued buckets instead of binary buckets. 2. It supports delete function as deleting an element from the CBF decrements the buckets counters for all of its hash values.

#### 1.5.1 Functions of Counting Bloom FIlters:

#### 1.5.2 Add:

Adding an element to the Counting Bloom Filters requires the following operations: 1. Hash the element being inserted form all the hash functions in the CBF and obtain hash value . 2. For every hash value , retrieve the corresponding bucket in the CBF. 3. Increment the value of the bucket by 1. (All buckets start at 0).

#### 1.5.3 Query

If we want to look up the presence of an element in our CBF, it requires the following operations: 1. Hash the element being inserted form all the hash functions in the CBF and obtain hash value. 2. For every hash value , retrieve the corresponding bucket in the CBF. 3. If all the corresponding bucket values are greater than zero, then say that the element is probably in the CBF. If any value is zero, then declare that the element is definetely not in the CBF.

#### 1.5.4 Delete

If we want to delete an element from our CBF, the following operations are required: 1. Query that the element is actually in the CBF. If it is definetely not in the CBF, thendo nothing. 2. Hash the element being inserted form all the hash functions in the CBF and obtain hash value. 3. For every hash value , retrieve the corresponding bucket in the CBF. 4. For every bucket, decrement the counter.

#### 1.5.5 Uses of Bloom Filters and Counting Bloom Filters:

There are several used of Bloom filters and Counting Bloom Filters; Some of them are mentioned below

1. Spellcheckers used in late 90s used BF's to check whether a word is correct or not. At that time space was limited so the constant space characteristic of BF's helped them. Also that they would look up a word in a very less time considering English languae has about 100,000 words.
2. It was used as a passwords filter on different websites where we have to create an account. When a client would register, they look it up in the bloom filter where all the weak passwords or obvious passwords were already saved. If you get that, you have to choose another one.

### 1.6 Question 3. [#DataStructures, #PythonProgramming, #CodeReadability]

#### 1.6.1 Implement in Python 3your designed CBFs data structure and all its properties. Make sure you carefully organize your Python code, write meaningful comments, and provide a thorough justification for your choice of hash functions. Meaning, why did you think a particular hash function is a good hash function in the context of CBFs.

```
In [9]: import random
        import mmh3

        class Bloom(object):
```

```
    def __init__(self, indices, numhash):       #indices: the number of buckets/tables in
        self.indices = [0] * indices           #numhash: the number of hash functions we
        self.length = indices
        self.numhash = numhash

         # generate numhash number of hash functions
         # every hash function then is called via a lambda
         # with one argument, so you may call this e.g.
         # my_element_hash = self.hash_functions[0](my_element)
        self.hash_functions = [lambda item: mmh3.hash(item, number) for number in range(


#This Function adds an element to our bloom filter.iterates over every hash function,
#and increments the value in the corresponding bucket.
    def add(self, item):
        for hash_fn in self.hash_functions:
            index = hash_fn(item) % self.length
            self.indices[index] += 1


#This function iterates over all hash functions,if all corresponding bucket values are g
#return True (that is a probabilistic truth) otherwise, False with certainty
    def look_up(self, item):
        for hash_fn in self.hash_functions:
            index = hash_fn(item) % self.length
            if self.indices[index] == 0:
                return False
        return True


# This function first sees if we can actually delete the item,by determinining if the it
#then it iterates over all hash functions and decrements all corresponding bucket values
    def delete(self, item):
        if self.look_up(item):
            for hash_fn in self.hash_functions:
                index = hash_fn(item) % self.length
                # if the element is in the filter,then this index MUST be greater than z
                assert self.indices[index] > 0
                self.indices[index] -= 1

def tests():
    print("Running tests!")
    CBF_Test1 = Bloom(50, 5)
    assert CBF_Test1.numhash == len(CBF_Test1.hash_functions)
    CBF_Test1.add("Usama Puri")
     # query works for present element
    assert CBF_Test1.look_up("Usama Puri")
     # query works for absent element
```

```
        assert not CBF_Test1.look_up("CS110")
        CBF_Test1.add("CS110")
        assert CBF_Test1.look_up("CS110")
        CBF_Test1.add("CBF Assignment")
        CBF_Test1.delete("CS110")
         # query works for now-deleted element
        assert not CBF_Test1.look_up("CS110")
         # delete is idempotent for elements that aren't present
        temp = CBF_Test1.indices
        CBF_Test1.delete("CS110")
        CBF_Test1.delete("Usama Puri")
        assert CBF_Test1.indices == temp

        print("Tests passed!")

    tests()
```

```
Running tests!
Tests passed!
```

### 1.6.2 Explanation of Hash Functions

As already discussed in Question 2 that the posssibility of getting a false possitive depends on the hash functions being used in that BF Algorithm. So in order to minimize this possibility and make my algorithm more efficient, I used the mmh3 library to generate my hash functions in the above Algorithm. I would justify my choice by saying that MurmurHash3 generates a set of robust and arbitrary hash functions, and we can use mmh3 to initialize hash functions with different seeds, which allows us to arbitarily generate as many hash functions as we want, each of them uniformly distributed. So the more robust and uniformely distributed these functions are the lesser is the probability to get any false positive.

## 1.7 Question 4.

### 1.7.1 Using your own Python implementation from question 3, generate data to push into the CBFs. Provide an analysis, both on theoretical grounds and using experimental corroboration, of how your implementation scales in terms of:

In the following questions, we're asked to implement our code in determining the memory size as a function of the false positive rate, among others.However, this is not a simple linear relationship, as there are two more variables that should be considered in order to get the desired results: 1. The number of hash functions. We will keep it constant, so that we can identify the effects we've been asked to identify. We'll pick as 5 an arbitary number of hash functions. 2. The number of elements to be inserted into the CBF. We also need to keep this one constant. We'll pick 150 as an arbitrary number of elements.

### 1.7.2 a) Memory size as a function of false positive rate

### 1.7.3 Theoretical Implimentation

Consider the following variables: 1. A false positive rate $r$ 2. A memory size $m$ (this is the number of buckets in our array) 3. $k$ hash functions, where $k$ is fixed 4. $n$ elements to be inserted into the CBF.

Now We assume that each of the hash functions selects a bucket in the Counting Bloom Filters with equal probability (assuming uniform distribution). Then the probability that one hash function will increment a bucket is $\frac{1}{m}$. On the other hand, the probability that one hash function will not increment the given bucket is $1 - \frac{1}{m}$. Since we have $k$ number of hash functions, then these are independent events, and the probability that none of the hash functions increment that bucket is $1 - (1 - \frac{1}{m})^{kn}$.

Now if we now insert elements in the CBF, then the probability that a given bucket has never been incremented is $((1 - \frac{1}{m})^k)^n$. Again, we multiply because it is $n$ independent events.

Consequently, the probability that a given bucket has been incremented is $1 - (1 - \frac{1}{m})^{kn}$ . Since we have one bucket per hash function, the probability that for some element $y$ , for every single hash function that hashes to some bucket, that bucket has been incremented is $(1 - (1 - \frac{1}{m})^{kn})^k$. This is the probability of a false positive. So, now the probability of our false positive is calculated to be:

$r = (1 - (1 - \frac{1}{m})^{kn})^k$

However, As we were asked for the memory size as a function of the false positive rate and we've only discussed it the other way around so far, so writing m in terms of r will be:

$r = (1 - (1 - \frac{1}{m})^{kn})^k$

$r^{\frac{1}{k}} = 1 - (1 - \frac{1}{m})^{kn}$

$1 - r^{\frac{1}{k}} = (1 - \frac{1}{m})^{kn}$

$(1 - r^{\frac{1}{k}})^{\frac{1}{kn}} = 1 - \frac{1}{m}$

$1 - (1 - r^{\frac{1}{k}})^{\frac{1}{kn}} = \frac{1}{m}$

$\$1\frac{}{1-(1-r^{\frac{1}{k}})^{\frac{1}{kn}}=m\$}$

### 1.7.4 Practical Implementation

```
In [21]: import random
         import matplotlib.pyplot as pyplot

         #This function returns a random alphabetic string of length 10 so any combination of st
         def make_random_string():
             to_return = ''
             alphabet = 'abcdefghijklmnopqrstuvwxyz'
             for i in range(10):
                 to_return += random.choice(alphabet)
             return to_return

         #This function gets the false positive rate for a given Counting Bloom Filter
         def get_false_positive_rate(cbf, number_of_elements_to_insert):
             false_positives = 0
             number_of_tries = 100  # use set insertion to avoid duplicates when constructing ou
```

```
                                # set of strings in the BF. (Statistical corner case.)
            strings_in_bf = set()
            while len(strings_in_bf) < number_of_elements_to_insert:
                new_string = make_random_string()
                strings_in_bf.add(new_string)
                cbf.add(new_string)

             # now make 100 new random strings, that are NOT in the CBF
             # statistical corner case if one of our test strings
             # actually were in the CBF (so the false positive would be a false positive)
            strings_not_in_bf = set()
            while len(strings_not_in_bf) < number_of_tries:
                new_random_string = make_random_string()
                if not new_random_string in strings_in_bf:
                    strings_not_in_bf.add(new_random_string)

            for s in strings_not_in_bf:
                if cbf.look_up(s):
                    false_positives += 1

            return false_positives/number_of_tries
```

In [28]:
```
k = 5
n = 150
memory_sizes = range(1, 10000, 10)
# initialize one bloom filter for every memory size
bloom_filters = [Bloom(i, k) for i in memory_sizes]
false_positive_rates = [get_false_positive_rate(cbf, n) for cbf in bloom_filters]

pyplot.plot(false_positive_rates , memory_sizes)
pyplot.xlabel('False Positive Rate')
pyplot.ylabel('Memory size')
pyplot.title('Memory size as a function of False positive rate')
pyplot.show()
```


As we can see in the above graph that it drops off at a fairly large angle, as we were expecting it to. Here we can see that if we have more memory to store data then we will have very less False positive cases. On the other hand if we have very less memory to store data we will have a large number of false positives

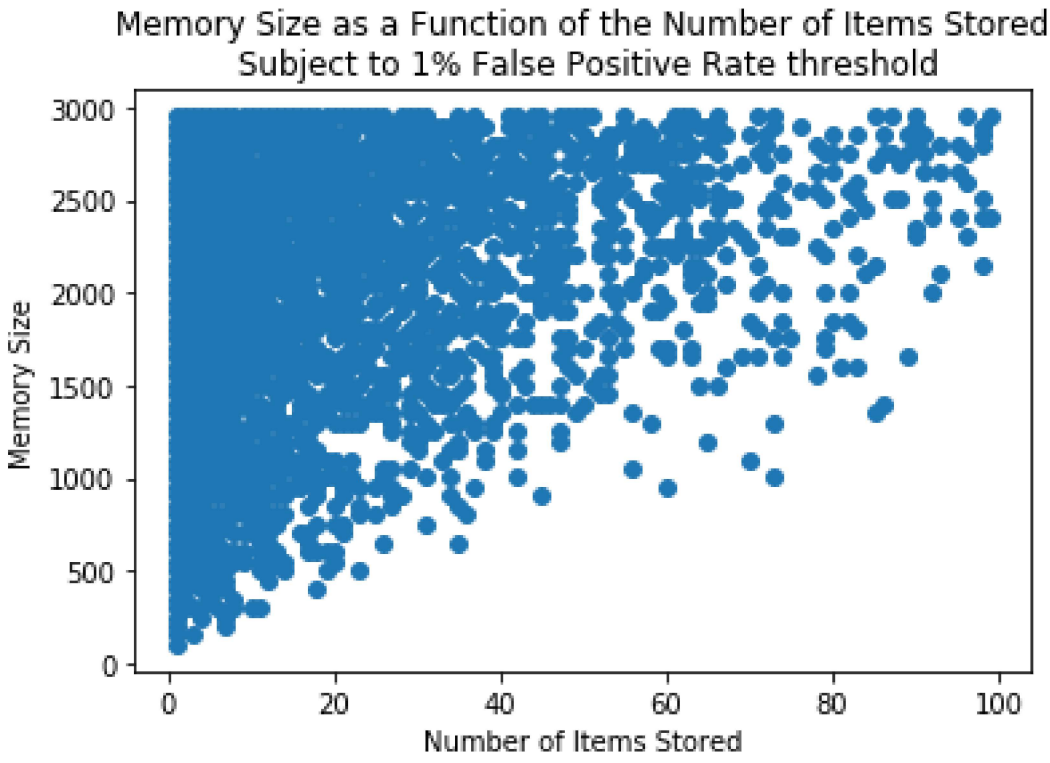
### 1.7.5 b) Memory size as a function of the number of items stored

Now we will try to show memory size as a function of item stored. As we already calculated, while keeping the false positive rate and number of hash functions constant the expression that captures this relation is: $m=1\frac{}{1-(1-r^{\frac{1}{k}})^{\frac{1}{kn}}}$

For $k$ and $r$ constant.Capturing this mathematically is really difficult, because we need to keep the Falsepositive Rate constant, for randomized experiments. But the real world is noisy. Below, we run a series of trials where we keep $k$ constant, and admit values $r$ of between 0% and 1%.

```
In [46]: k = 5
         fpr_min = 0.00
         fpr_max = 0.01
         values_of_n = range(1, 100, 1)
         memory_sizes = range(1, 3000, 50)
         # the number of items stored and memory size is an array of tuples n_m pairs
         # that keep a constant number of hash functions (5)
         # and an approximately constant false positive rate(0-1%)
         n_m_pairs = []
         for n in values_of_n:
```

```
# initialize one bloom filter for every memory size
# need to initialize here because we can't re-use it between trials
    bloom_filters = [Bloom(x, k) for x in memory_sizes]
# for this n, find the memory sizes that match our desired FPR
    false_positive_rates = [get_false_positive_rate(cbf, n) for cbf in bloom_filters]

    for index, fpr in enumerate(false_positive_rates):
        if (fpr_min <= fpr <= fpr_max):
            memory_size = memory_sizes[index]
            n_m_pairs.append((n, memory_size))

pyplot.scatter([n for (n,m) in n_m_pairs],[m for (n,m) in n_m_pairs])
pyplot.ylabel('Memory Size')
pyplot.xlabel('Number of Items Stored')
pyplot.title('Memory Size as a Function of the Number of Items Stored\n Subject to 1% F
pyplot.show()
```




The above graph shows that when we try to store one element, any memory size will work. When trying to store 20 elements, only memory sizes above 500 yield error rates below 1%. When trying to store 40 items, the threshold for reliability seems to be around 1000. Around 60, consistent success comes in at a memory size of around 2000, and beyond that, consistency is rare. This trend looks as if the memory size is approximately linear in thenumber of items stored, with the linear factor being roughly 20x.

### 1.7.6 c) Access time as a function of the false-positive rate

The access time is constant in the false positive rate, because it only depends on the number of hash functions. After we have choosen the number of hash functions when we initialize the CBF. It does not change afterwards.When we look up a CBF for the presence of an element, we hash that item with every hash function, which is a constant time operation. Then we look up the corresponding bucket for every hash. As we can index directly into the array, this is also a constant-time operation. Combining these two constant time operations yields another constant time operation, and we do this operation as many times as we have hash functions. Similarly, the access time is linear in the number of hash functions. This means that so long as we keep the number of hash functions constant, then the access time stays constant as well, regardless of how the false positive rate varies otherwise.

### 1.7.7 Digression

There is one corner-case to consider: when we vary the false positive rate by varying the number of hash functions, then we're also going to vary the access time. This relationship is hard to reason about intuitively, because even though a linear change in the number of hash functions will produce a linear change in the access time, it may produce a nonlinear change in the false positive rate, so we would end up trying to relate non-linear changes in false positive rate to linear changes in access time.

If we take our earlier expression $r = (1 - (1 - \frac{1}{m})^{kn})^k$ , we may say that since access time is proportionally linear to the number of hash functions , we may as well treat as the access time in this equation. In that case, we can rewrite our expression to give access time as a function of the false positive rate, i.e. as a function of . However, this is exceedingly hard. (I didn't manage to do it.)In any case, this is probably not proper to consider. What we're effectively saying is that "if you want to change the false positive rate by changing the number of hash functions, then it will also change the access time". In this scenario, the change in access time is a side-effect of the change in false-positive rate; the causal variable (one might call it a confounding variable) is the number of hash functions.

### 1.7.8 d) Access time as a function of the number of items stored

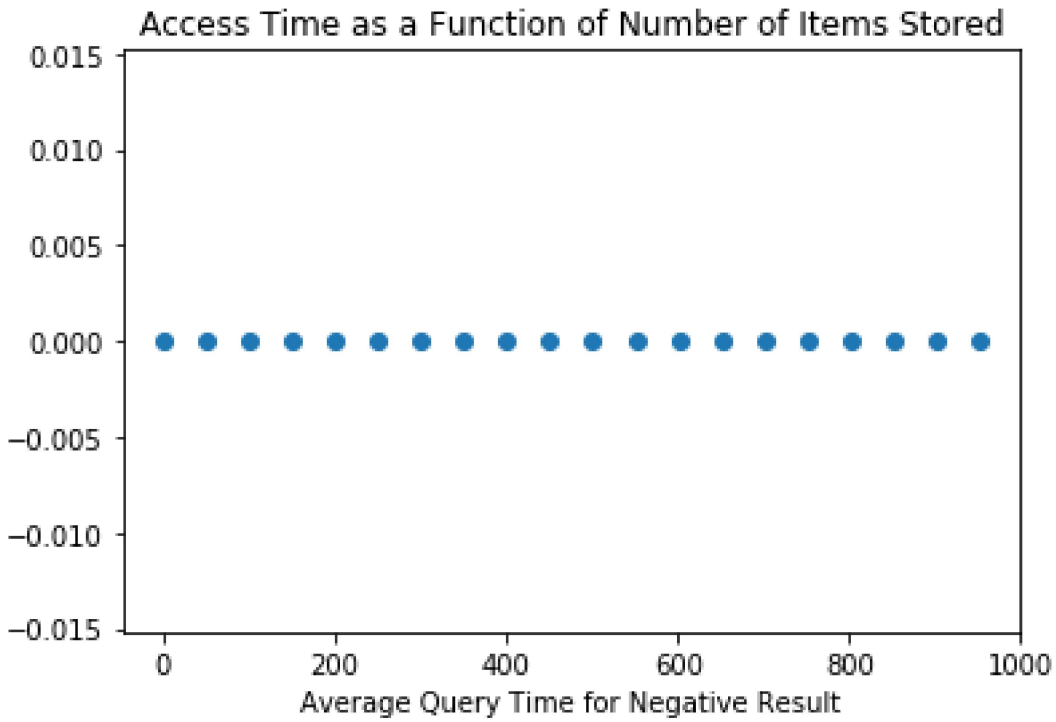
In the above explanation, we suggest that the number of items stored does not influence the access time. However, there is a corner-case: when we perform a membership look up for an element that is not in the CBF. We return False as soon as we hit a zero. Otherwise, we keep checking all the other hash values. This means that when very few items are stored, we have to conduct fewer worst-case and average-case comparisons than if many items are stored. (The best-case scenario, where we hit a zero as the first element, remains the same.)

Keeping the number of hash functions constant, this is the only way for the Access Time (i.e. membership look up) to take longer. We experiment with this below, keeping the memory size and number of hash functions constant. We do not need to worry about keeping the false positive rate constant, because, as per the above, we are interested only in negative resuls, which are not affected by the false positive rate.

```
In [49]: import time
         # We need a timed implementation of the CBF.
         # We'll subclass what we implemented earlier.
         class TimedCBF(Bloom):
```

```
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        # store lookup times here, in seconds
        self.times = []

    def query(self, item):
        start = time.time()
        result = super().look_up(item)
        end = time.time()
        # we are ONLY interested in negative results!
        if not result:
            self.times.append(end - start)

# we use 1000 hash functions this time, so that measuring the time is useful otherwise
k = 1000
m = 1000
values_of_n = range(1, 1000, 50)
access_times_by_n = []
# this code is conceptually similar to `get_false_positives`,
# but with a few important/subtle distinctions.
for n in values_of_n:
# need to initialize here because we can't re-use it between trials
    timed_cbf = TimedCBF(m, k)
    # insert n values into the timed_cbf
    # use set insertion to avoid duplicates.
    strings_in_timed_cbf = set()
    while len(strings_in_timed_cbf) < n:
        new_string = make_random_string()
        strings_in_timed_cbf.add(new_string)
        timed_cbf.add(new_string)

    number_of_samples = 100

    # now query for random, non-present strings into the timed_cbf
    # stop when we've obtained 100 negative results
    strings_not_in_bf = set()

    while len(timed_cbf.times) < number_of_samples:
        new_random_string = make_random_string()
        if not new_random_string in strings_in_timed_cbf:
            strings_not_in_bf.add(new_random_string)
            timed_cbf.query(new_random_string)

    # add the average time to our lookups array
    average = sum(timed_cbf.times)/len(timed_cbf.times)
    access_times_by_n.append(average)

pyplot.scatter(values_of_n, access_times_by_n,)
```

```
pyplot.xlabel('Number of Items Stored')
pyplot.xlabel('Average Query Time for Negative Result')
pyplot.title('Access Time as a Function of Number of Items Stored')
pyplot.show()
```



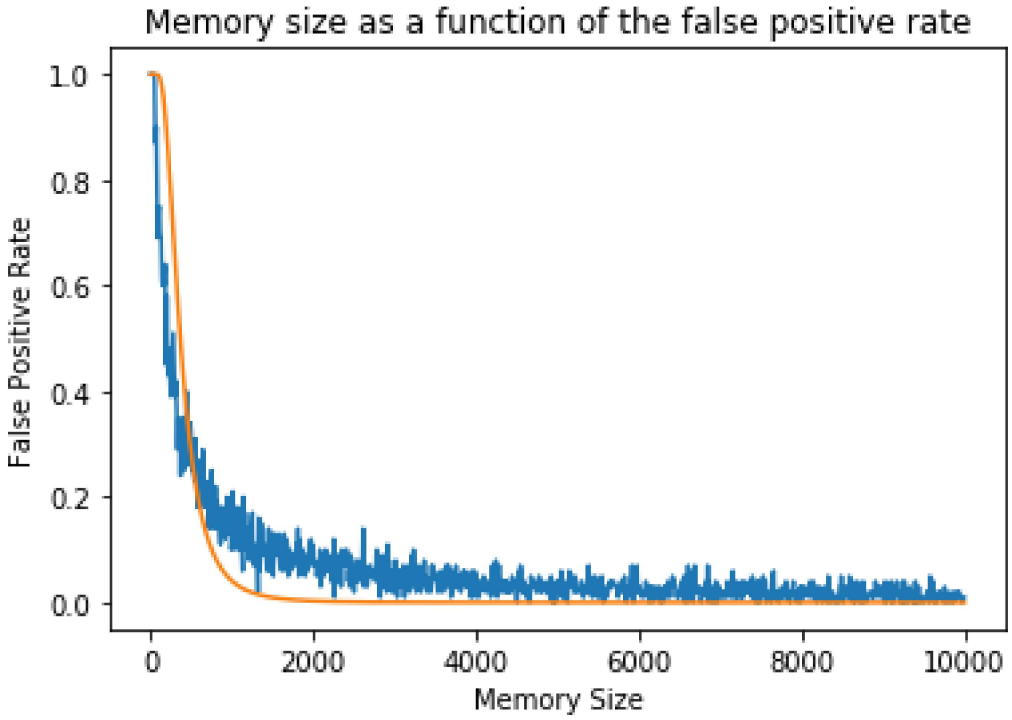

In the above graph it seems that in reality and in terms of CPU time, our access time per lookup is so fast that any concern around digression is simply dominated. This plot also shows average of one lookup per negative result: in other words, the first lookup is a zero, and a negative result is found instantly. That's not very useful, so I then tried expanding the number of hash functions, or the number of items to insert, such that the lookup times would go up, but those configurations very quickly wouldn't return at all -- i.e. none (or very few) of the cells in the CBF would have a zero, so my script would loop forever trying to find negative results. This means that timing this behavior for the case we're actually interested in requires a very delicate balance of memory size, number of hash functions, and number of elements inserted, such that we have some negative r

## 1.8 Question 5.

### 1.8.1 Produce a plot to show that your implementation's falsepositive rate matches the theoretically expected rate.

```
In [56]: k = 5
         n = 150
         memory_sizes = range(1, 10000, 10)
         # initialize one bloom filter for every memory size
         bloom_filters = [Bloom(x, k) for x in memory_sizes]
```

```
false_positive_rates = [get_false_positive_rate(cbf, n) for cbf in bloom_filters]
# false positive rates in theory, as given by our formula
computed_false_positive_rates = [(1 - (1 - (1/m))**(n*k))**k for m in memory_sizes]

pyplot.plot(memory_sizes,false_positive_rates)
pyplot.plot(memory_sizes,computed_false_positive_rates,)
pyplot.xlabel('Memory Size')
pyplot.ylabel('False Positive Rate')
pyplot.title('Memory size as a function of the false positive rate')
pyplot.show()

pyplot.plot(memory_sizes[:100],false_positive_rates[:100])
pyplot.plot(memory_sizes[:100],computed_false_positive_rates[:100])
pyplot.xlabel('Memory Size')
pyplot.ylabel('False Positive Rate')
pyplot.title('Memory size as a function of the false positive rate')
pyplot.show()
```




For this part i made 2 graphs one graph with *m=100000* and the other with *m=1000*.

Although the first graph looks roughly as desired, we note that the false positive rate actually appears to be significantly less than the theoretically expected false positive rate in the very start. Afterwards, around m=1000 , we note that the theoretical false positive rate drops below our actual false postive rate for a while. In other words, the theoretical exponential dropoff appears to be much harsher than it's turned out to be in our experiment. So, I zoomed in on the early part of the graph. The 2nd graph shows that our actual false positive rate is significantly less than the expected false positive rate for approximately the first 400 elements. Maybe the closed-form expression we arrived at earlier is inaccurate due to noise when n is close to m? It's also possible that our assumptions around independence of events are incorrect and the mmh3 hash functions are not as uniformly random as we assumed they are.

### 1.9 Question 6

The simplest corner case is that CBFs take much more space than regular Bloom Filters. For a regular Bloom Filter, its array is simply of bits -- 1s and 0s -- whereas CBFs require integers. An unsigned integer, between 0 and 65,535 requires 2 bytes (16 bits). An unsigned integer between 0 and 4,294,967,295 requires 4 bytes (32 bits). Depending on the number of elements you expect to see, a CBF may occupy more memory than a regular BF.

## 2 HC Applications:

**#Dataviz:** I used this HC to effectively create and analyze the graphs. One analyses that I liked the most was in question 5 where at first the graph showed to be agreeing with the theoratical

pattern but when i zoomed in the graph I saw that the values is significantly less than the expected value which at first was hard to notice due to large data.

**#Organization:** Organized the assignment by commenting on functions, providing detailed and structured justification, and making sure to justify any critique of the questions.

**#probability:** Here I also used this HC to calculate the approximate probability of getting a false positive by deriving an expression using probability that would calculate the false positive rate

# 3 Sources:


1. Bloom Filters The Basics 16 min. (2019). Retrieved 3 November 2019, from: https://www.youtube.com/watch?v=zYlxP7F3Z3c
2. Bloom Filters by Example. (2019). Retrieved 3 November 2019, from https://llimllib.github.io/bloomfilter-tutorial/
3. Wikipedia (n.d.). Bloom filter. Retrieved on 6th March 2019, from: https://en.wikipedia.org/wiki/Bloom_filter#Probability_of_false_positives (https://en.wikipedia.org/wiki/Bloom_filter#Probability_of_false_positives)
4. Appleby, A. (2011). Murmurhash3. Retrieved on 6th March 2019, from: https://github.com/aappleby/smhasher/wiki/MurmurHash3 (https://github.com/aappleby/smhasher/wiki/MurmurHash3)