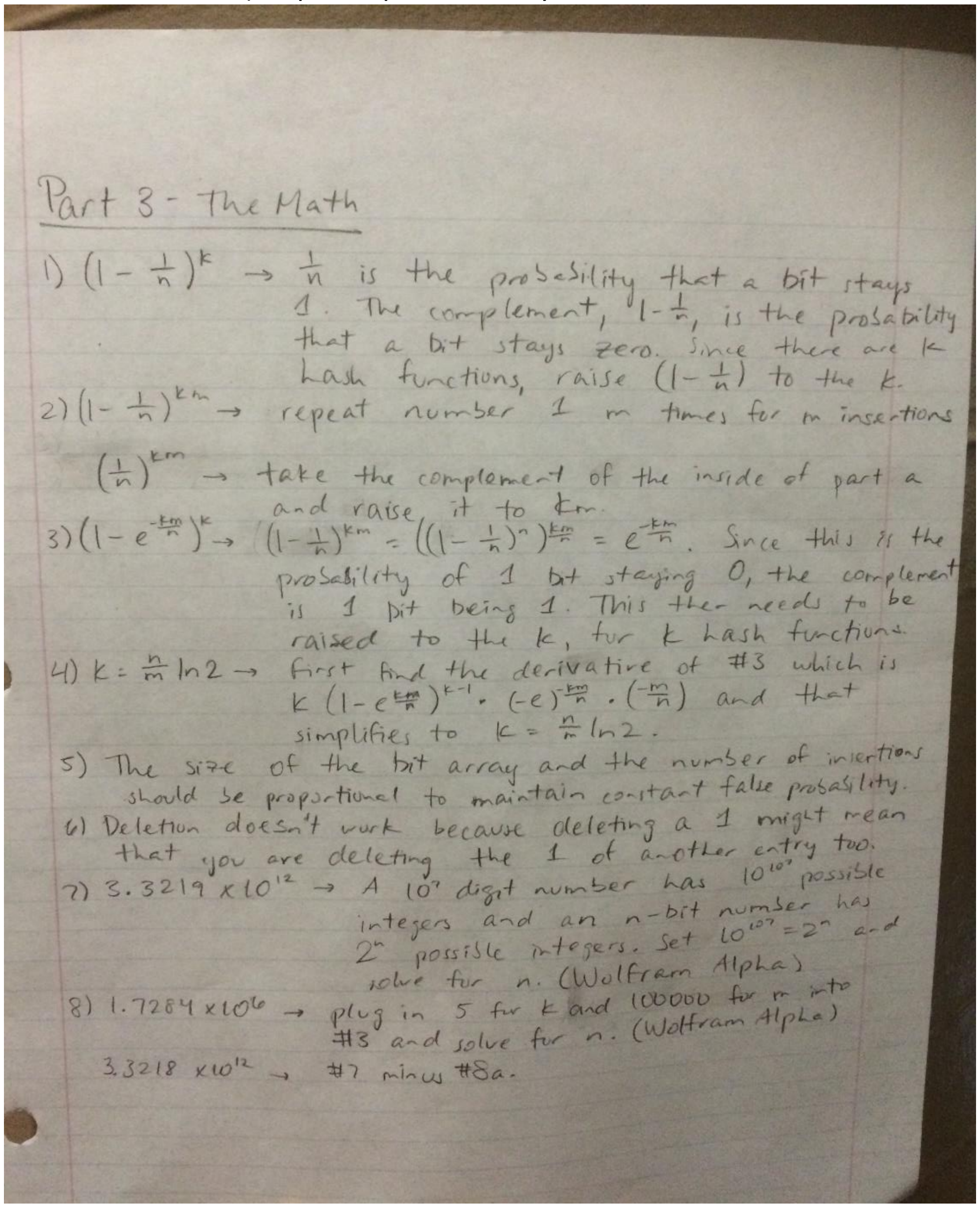
CS 237 Lab 2
Vidya Akavoor
Collaborators: Sreeja Keesara, Lauren DiSalvo
Sources:
http://pages.cs.wisc.edu/~cao/papers/summary-cache/node8.html
https://www.youtube.com/watch?v=bEmBh1HtYrw
https://en.wikipedia.org/wiki/Bloom filter#Optimal number of hash functions
http://web.stanford.edu/class/archive/cs/cs161/cs161.1152/Lecture-9.pdf
https://www.wolframalpha.com/
Late Days Used: 0
Late Days Left: 3

Part 3 – The Math (I copied a picture of my work because I had written it out before)



Part 3 - The Math

1) $(1-\frac{1}{n})^k$ → $\frac{1}{n}$ is the probability that a bit stays 1. The complement, $1-\frac{1}{n}$, is the probability that a bit stays zero. Since there are k hash functions, raise $(1-\frac{1}{n})$ to the k.

2) $(1-\frac{1}{n})^{km}$ → repeat number 1 m times for m insertions

$(\frac{1}{n})^{km}$ → take the complement of the inside of part a and raise it to km.

3) $(1-e^{\frac{-km}{n}})^k$ → $(1-\frac{1}{n})^{km} = ((1-\frac{1}{n})^n)^{\frac{km}{n}} = e^{\frac{-km}{n}}$. Since this is the probability of 1 bit staying 0, the complement is 1 bit being 1. This then needs to be raised to the k, for k hash functions.

4) $k = \frac{n}{m}\ln 2$ → first find the derivative of #3 which is $k(1-e^{\frac{km}{n}})^{k-1} \cdot (-e)^{\frac{-km}{n}} \cdot (\frac{-m}{n})$ and that simplifies to $k = \frac{n}{m}\ln 2$.

5) The size of the bit array and the number of insertions should be proportional to maintain constant false probability.

6) Deletion doesn't work because deleting a 1 might mean that you are deleting the 1 of another entry too.

7) 3.3219×10^{12} → A 10^7 digit number has 10^{10^7} possible integers and an n-bit number has 2^n possible integers. Set $10^{10^7} = 2^n$ and solve for n. (Wolfram Alpha)

8) 1.7284×10^6 → plug in 5 for k and 100000 for m into #3 and solve for n. (Wolfram Alpha)

3.3218×10^{12} → #7 minus #8a.

Part 4 – Python and Statistics

9) a.
```
def insert(self, key):
         for x in range(self.numHashFunctions):
             hashValue = self.hash.hash_i(key,x)
             self.bitArray[hashValue] = 1
```
b.
```
def lookup(self, key):
        for x in range(self.numHashFunctions):
            hashValue = self.hash.hash_i(key,x)
            if self.bitArray[hashValue] == 0:
                return False
        return True
```
10)
```
def rand_inserts(self):
        self.insert(str(rnd.randint(0, 100000000)))
```
11)
```
bf1 = BloomFilter(4095, 10)
    for z in range(600):
        bf1.rand_inserts()
```
Using the equation from number 3, the theoretical false positive probability is approximately 0.07229.

12) The average false probability with 10 hash functions is 0.0722467

(all code is together, below all answers; output is below)

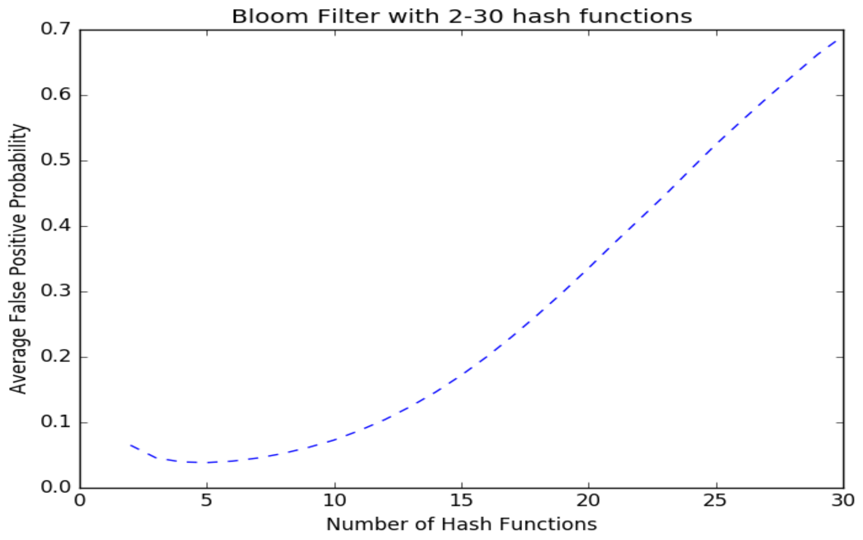
output:
```
>>> runfile('C:/Users/vid82/OneDrive/Documents/CS/CS237/bloom-SkeletonCode/Bloom-Filter.py', wdir='C:/Users/vid82/OneDrive/Documents/CS/CS237/bloom-SkeletonCode')

Reloaded modules: bloomFilterHash

0.0722467

>>>
```

13)



The plot tells us that after about 5 hash functions, as the number of hash functions goes up, the average false probability goes up too. This is because, assuming the bit array size stays constant, the more hash functions you have, the more bits get set with each insertion and then the more likely it is that a lookup that isn't in the bloom filter will still get the right combination of 1's.

Part 5 – Independence and Hash Functions

14) Pr[A ∩ B] = Pr[A] * Pr[B] = (1/2) * (1/2) = (1/4) → A and B are independent

Pr[B ∩ C] = Pr[B] * Pr[C] = (1/2) * (1/2) = (1/4) → B and C are independent

Pr[A ∩ C] = Pr[A] * Pr[C] = (1/2) * (1/2) = (1/4) → A and C are independent

15) If the second flip matches the first flip (as in ½ probability times ½ probability which is ¼ probability) then C (both flips are the same) is guaranteed to be 1 so the probability is ¼ while A times B times C is 1/8 and since they are not equal, A and B and C are not independent.

16)
```
def insert(self, key):
        for x in range(self.numHashFunctions):
            if self.pairwise:
                hashValue = self.hash.pairwiseHash_i(key,x)
            else:
                hashValue = self.hash.hash_i(key,x)
            self.bitArray[hashValue] = 1
    def lookup(self, key):
        for x in range(self.numHashFunctions):
            if self.pairwise:
                hashValue = self.hash.pairwiseHash_i(key,x)
            else:
                hashValue = self.hash.hash_i(key,x)
            if self.bitArray[hashValue] == 0:
                return False
        return True
```

17) The theoretical false probability is .04002.

18) The first number is the average and the second is the max.

output:

```
>>> runfile('C:/Users/vid82/OneDrive/Documents/CS/CS237/bloom-SkeletonCode/Bloom-Filter.py', wdir='C:/Users/vid82/OneDrive/Documents/CS/CS237/bloom-SkeletonCode')

Reloaded modules: bloomFilterHash

(0.039856299999999997, 0.050900000000000001)

>>>
```

19) The first number is the average and the second is the max.

output:

```
>>> runfile('C:/Users/vid82/OneDrive/Documents/CS/CS237/bloom-SkeletonCode/Bloom-Filter.py', wdir='C:/Users/vid82/OneDrive/Documents/CS/CS237/bloom-SkeletonCode')

Reloaded modules: bloomFilterHash

(0.063545699999999997, 0.075200000000000003)
```

20) The pairwise independent hash functions give a higher false probability than the independent hash functions

Part 6 – Security of Hash Functions

21) With the adversarial insert, the average false probability is very similar to question 18, but the max is much smaller than question 18.

output:

```
>>> runfile('C:/Users/vid82/OneDrive/Documents/CS/CS237/bloom-SkeletonCode/Bloom-Filter.py', wdir='C:/Users/vid82/OneDrive/Documents/CS/CS237/bloom-SkeletonCode')

Reloaded modules: bloomFilterHash

(0.0303082, 0.037600000000000001)
```

22) With the adversarial insert and lookup, the average false probability is 1 which means that the adversary was able to find all the test lookups in the bloom filter.

output:

```
>>> runfile('C:/Users/vid82/OneDrive/Documents/CS/CS237/bloom-SkeletonCode/Bloom-Filter.py', wdir='C:/Users/vid82/OneDrive/Documents/CS/CS237/bloom-SkeletonCode')

Reloaded modules: bloomFilterHash

(1.0, 1.0)
```