IE531: Algorithms for Data Analytics

© Professor Ramavarapu "RS" Sreenivas

Industrial and Enterprise Systems Engineering, The Grainger College of Engineering, UIUC

Lesson 6: Algorithms for Massive Data Problems: Streaming, Sketching, and Sampling

The best analogy to use in the context of streaming data models is this -- a (massively long) train with an a bunch of cars (with appropriate attributes) is going past you, and you are to make some educated guesses on the statistical features of the train (and its cars) as it runs past your eyes.

More precisely, we are concerned with algorithms that compute some function of a massively long input stream (analogous to the "train") . In the *vanilla streaming model*, this is formalized as a sequence $\langle a_1, a_2, \ldots, a_n \rangle$, where each $a_i \in \{1, 2, \ldots m\}$. The stream length is n and the universe-size is m, and both these integers are very very large numbers. Some folks refer to the stream $\langle a_1, a_2, \ldots, a_n \rangle$ as a multiset or a *bag*. We can associate a *frequency vector* $(f_1, f_2, \ldots f_m)$ with the stream $\langle a_1, a_2, \ldots, a_n \rangle$, where $f_j = card(\{i \mid a_i = j\})$ (i.e. f_j is the number of occurrences of j in the stream $\langle a_1, a_2, \ldots, a_n \rangle$).

You will see two terms being used in the context of streaming data models -- *turnstile*- and *cash-register* models. The notation is not standardized (AFAIK). In the turnstile model the only a finite-window of the (continuous) data-stream is available to the user -- this is like saying you can only remember the last n-many cars of the train you have seen go past you, in the train-analogy. In the cash-register model, you can store a small number of variables that keep track of a few important things over the entire history of the data-stream that has gone past your eyes.

Section 1: The `MAJORITY` and `FREQUENT` Problem

We have a stream $\langle a_1, a_2, \ldots, a_n \rangle$ where each $a_i \in \{1, 2, \ldots m\}$. We have the associated frequency vector $(f_1, f_2, \ldots f_m)$, where $\sum_{i=1}^{n} f_i = n$. For the `MAJORITY` problem, we need to to find a j such that $f_j > n/2$, if it exits. If there is no such j then we can output anything. For the `FREQUENT` problem, we have to output a set $\{j \mid f_j > n/k\}$, if it exists (otherwise, it is OK to output some junk).

Section 1.1: Misra-Gries Family of Algorithms

Let us look at the `MAJORITY` problem first. We have two variables that we store. The first-variable is the *key* (which is either a member of $\{1, 2, \ldots, m\}$ or a null-entity). The second-variable is an integer (which is a count). We start with an empty key, and a count of zero.

Every time an element $a_i = j$ of the data-stream is observed, if the key is empty we set the value of the key to j, and we initialize the count to 1. If the key is not empty, and equal to j, we increment the count by 1. If the key is not empty, and not equal to j, we decrement the count by 1 -- if the count

becomes zero as a result of this decrementing, we set the key to null-entity. It is not hard to see that if there is a majority-element, it will be the value of the key.

For the FREQUENT problem, we have $k-1$ keys, with associated counts. The keys are initialized to null-entities, their counts being zero. Every time an element of $a_i = j$ of the data-stream is observed, and if j is one of the keys, we increment is counter by 1. If j is not one of the keys, and there is an null-entity key, we set it equal to j with a count of 1. If j is not one of the keys, and there is no null-entity keys, we decrement the count of all counters. If any counter is zero as a result of this decrementing, we set its key to be a null-entity.

The MAJORITY problem is a special case of the FREQUENT problem, where $k = 2$. There is a wealth of theoretical results on the Misra-Gries family of algorithms (i.e. the ones described above that solves the MAJORITY and FREQUENT problems). I am skipping all of it. That said, it is important to note that these algorithms only use $O(k(\log n + \log m))$-bits of memory/space and they effectively solve the FREQUENT problem.

Section 1.2: Approximate solution to a variation of the FREQUENT problem

We have a stream $\langle a_1, a_2, \ldots, a_n \rangle$ where each $a_i \in \{1, 2, \ldots m\}$. We have the associated frequency vector $(f_1, f_2, \ldots f_m)$, where $\sum_{i=1}^{n} f_i = n$. In this version of the problem, we wish to maintain an approximation $(\widehat{f}_1, \widehat{f}_2, \ldots \widehat{f}_m)$ to the frequency vector $(f_1, f_2, \ldots f_m)$, such that $\forall j, \mid f_j - \widehat{f}_j \mid \leq \epsilon n$, for a desired $\epsilon > 0$.

For this, we keep l-many keys (with their associated counters), where $l \ll n$. By design, we have at least one empty key (and zeroed counter) at any instant.

Like before (with FREQUENT problem), if the observed stream-element is one of the keys, we increment its associated counter. If it is not in one of the key, then

1. We add it to the list of keys (note: there is always one empty key, by design).
2. To ensure there is at least one empty key, we take the median value of the counters (let us call it δ_t at the t-step of the algorithm), and subtract the median-value from all counters. We delete/empty all keys with a counter value that is less-than-or-equal-to zero.

At any point in this algorithm $\widehat{f}_j$ is the value of the counter associated with j (if it exists).
It follows that $\widehat{f}_j \leq f_j$ (because we might have deleted/emptied the key associated with j at some point in the algorithm). It also follows that $\widehat{f}_j \geq f_j - \sum_t \delta_t$, which in turn implies $\sum_t \delta_t \geq f_j - \widehat{f}_j$.
We also have,

$$0 \leq \sum_j \widehat{f}_j \leq \sum_t \left(1 - \frac{l}{2}\delta_t\right) = n - \frac{l}{2}\sum_t \delta_t \Rightarrow \sum_t \delta_t \leq \frac{2n}{l},$$

if $l = 2/\epsilon$, then $\forall j, \epsilon n \geq \sum_t \delta_t \geq f_j - \widehat{f}_j$, which accomplishes what we need. This idea has been used by several researchers to form what are called *sketches* of larger data streams or larger objects. You may want to look at how this concept has been used in the paper on Simple and Deterministic Matrix Sketches by Edo Liberty.

The sample code shown below illustrates the Misr-Gries procedure.

In [1]:
```
# IE531: Algorithms for Data Analytics
# Written by Prof. R.S. Sreenivas
#
```

```
# Illustration of the Misra-Gries Procedure
# This is a modified version of https://github.com/rhiever/big-data-hw/blob/master/Mi
#
# For a given stream-size, and epsilon-value, we pick L-many counters, where L = 2/ep
# This should result in an approximate-frequency count such that |real-freq - approx-
#

import numpy.random as nrandom
import math
from collections import Counter, defaultdict

n = 100000
epsilon = 0.05
L = math.ceil(2/epsilon)
print ('Epsilon = ' + str(epsilon))
print ('Stream-length (n) = ' + str(n))
print ('Misra-Gries requires ' + str(L) + '-many counters')
print ('We expect |real-freq - approximate-freq | <= ' + str(math.ceil(n*epsilon)))

data_stream = nrandom.randint(low=0, high=101, size=n)
data_counts = Counter(data_stream)

print ('\n')
print ('Rank' + '\t' + 'Item #' + '\t' + 'Real-Frequency')

for rank, token in enumerate(sorted(data_counts, key=data_counts.get, reverse=True)):
    rank += 1
    if rank > 1.5*L:
        break

    print (str(rank) + '\t' + str(token) + '\t' + str(data_counts[token]))

print ('\n')
print ("=== MISRA GRIES RANKS ===")
print ('Rank' + '\t' + 'Item #' + '\t' + 'Approx-Freq.' + '\t' + 'Cond. (True/False)?

mg_counts = defaultdict(int)

for token in data_stream:
    if len(mg_counts.keys()) < L:
        mg_counts[token] += 1

    elif token in mg_counts:
        mg_counts[token] += 1

    else:
        for key in list(mg_counts.keys()):
            mg_counts[key] -= 1
            if mg_counts[key] == 0:
                del mg_counts[key]


for rank, token in enumerate(sorted(mg_counts, key=mg_counts.get, reverse=True)):
    rank += 1
    print (str(rank) + '\t' + str(token) + '\t' + str(mg_counts[token]) + '\t\t'  +
           str(abs(mg_counts[token] - data_counts[token]) <= (epsilon*n)))
```

```
Epsilon = 0.05
Stream-length (n) = 100000
Misra-Gries requires 40-many counters
We expect |real-freq - approximate-freq | <= 5000


Rank    Item #  Real-Frequency
1       28      1107
2       52      1073
3       79      1068
4       3       1057
5       17      1055
6       83      1042
7       29      1038
8       56      1037
9       77      1035
10      33      1034
11      35      1030
12      84      1029
13      30      1028
14      26      1026
15      73      1024
16      93      1023
17      96      1023
18      12      1021
19      2       1020
20      95      1020
21      75      1017
22      32      1017
23      64      1016
24      88      1016
25      70      1015
26      40      1014
27      81      1014
28      43      1013
29      18      1012
30      76      1011
31      20      1010
32      89      1009
33      59      1008
34      65      1007
35      31      1007
36      19      1006
37      42      1005
38      47      1004
39      36      1003
40      5       1003
41      90      1002
42      82      1000
43      9       998
44      61      997
45      8       997
46      16      994
47      22      994
48      34      993
49      71      993
50      38      992
51      41      991
52      92      991
53      78      990
54      63      989
55      87      989
56      46      988
57      99      988
58      23      986
59      58      983
```

```
60      25      980


=== MISRA GRIES RANKS ===
Rank    Item #  Approx-Freq.    Cond. (True/False)?
1       58      3               True
2       51      2               True
3       64      2               True
4       94      2               True
5       38      2               True
6       69      2               True
7       13      2               True
8       74      2               True
9       68      2               True
10      70      2               True
11      22      1               True
12      12      1               True
13      49      1               True
14      79      1               True
15      85      1               True
16      72      1               True
17      83      1               True
18      98      1               True
19      62      1               True
20      1       1               True
21      36      1               True
22      60      1               True
23      28      1               True
24      54      1               True
25      75      1               True
26      21      1               True
27      67      1               True
28      55      1               True
29      42      1               True
30      24      1               True
31      43      1               True
```

Section 2: Hashing

This is an interesting, long, and involved topic. We are going to do the bare minimum to get ourselves a taste of its utility in the big data domain. Suppose you had n (where n is very very very large) objects stored/indexed in a systematic manner. You will encounter scenarios where you are given an object and you are asked if this object-at-hand is in your inventory of stored objects. In the absence of any structure to the storage/inventory process, in the worst-case you will need to examine all n objects in your inventory before you declare the object-at-hand is either present/absent. In a sentence -- the *average-time* (not the worst-case time) for the problem of storing/checking-inventory with caching is constant (i.e. $O(1)$ for fixed utilization/load) -- and this makes it very apropos to Big Data.

Section 2.1: Hashing in Python

You should read this Wikipedia article on how Hashing is done in practice. For the present it will suffice if you understood the hash as an integer-value of fixed-size that identifies an object (something like a serial-number, for example). Some objects cannot be hashed in Python (you will figure that out when you try it, and you get an error). These integer-values can be used as a "proxy" for the object when it involves searching/inventory etc. The code shown below illustrates these concepts. As noted here the hash values in Python 3.x will change with each run. That is, it is hard

to "guess" what the hash value would be (it involves pseudorandom number generators with different initial-seeds, which we have seen before).

In [2]:
```
# IE531: Algorithms for Data Analytics
# Written by Prof. R.S. Sreenivas
#
# Illustration of Hashing
# Taken mostly from https://stackoverflow.com/questions/17585730/what-does-hash-do-in
#
import sys
print(sys.hash_info)

print(hash("Look at me!"))
f = "Look at me!"
print(hash(f))
## this has an extra exclamation -- a completely different hash value!
g = "Look at me!!"
print(hash(g))
#
# To learn more, see https://stackoverflow.com/questions/37612524/when-is-hashn-n-in-
```

```
sys.hash_info(width=64, modulus=2305843009213693951, inf=314159, nan=0, imag=1000003,
algorithm='siphash24', hash_bits=64, seed_bits=128, cutoff=0)
3656977634236051124
3656977634236051124
-7598075605624298862
```

Section 2.2: Chaining

It would help to think of the problem as follows -- we have n-many objects (called *keys*; denoted by $K = \{k_1, k_2, \ldots, k_n\}$) that we wish to store (for easy retrieval/checking later on). We have a *hash function* $h : K \rightarrow \{0, 1, \ldots, m-1\}$ that assigns a slot/spot on a *hash-table* of size m (cf. figure 1). The anthropomorphic analogy goes as follows -- when k_i arrives for the first-time, it is assigned to the slot $h(k_i)$ in the hash-table. If this slot is already occupied by a bunch of keys (knowns as a *collision*), then k_i is placed in the beginning/end of a chain of keys associated with the slot (cf. figure 1 for an illustration). For an arbitrary, heretofore unseen k_i, the hash function assigns k_i to any one of the slots $\{0, 1, \ldots, m-1\}$ with equal probability -- that is, $Prob\{h(k_i) = j\} = \frac{1}{m}$ for any k_i and any j. A quick Google-search or a title-search at Amazon will show you that a lot has been written about the design of hash functions. For this discourse, you might as well assume k_i is an unsigned integer and $h(k_i) = (a \times k_i + b)_{mod\, m}$ for a carefully selected a and b.

Figure 1: Illustration of chaining.

Suppose we have built the chains associated with the hash table and we have a key k_i in our hands, and we want to know if we have already seen it earlier -- we would check if k_i is present in the chain associated with $h(k_i)$ before we declare success/failure. The following observation notes that an unsuccessful search will take $\Theta(1 + \alpha)$ time on an average, where $\alpha = \frac{n}{m}$.

> **Observation 1**: On an average, an unsuccessful search will take $\Theta(1 + \alpha)$, where $\alpha = \frac{n}{m}$.
> **Proof (Sketch)**: Any key is equally likely to be assigned to any of the m slots. If we have built the table already -- it has n objects assigned to m slots, which means the average number of objects assigned to a slot is $\alpha = \frac{n}{m}$. We will know k_i is not in the

table after looking at all elements in the chain associated with its slot -- this will take $\Theta(1 + \text{length of chain})$ time. Therefore, the average running time is $\Theta(1 + \text{average length of chain})$ time. That is, the procedure is $\Theta(1 + \alpha)$.

The next observation states that a successful search will also take $\Theta(1 + \alpha)$ time on an average, where $\alpha = \frac{n}{m}$.

> **Observation 2**: On an average, a successful search will take $\Theta(1 + \alpha)$, where $\alpha = \frac{n}{m}$.
> **Proof (Sketch)**: Let us denote the key by k_i (and it exists in the hash table). When k_i was first inserted into the table, let us say it was the i-th key among the sequence of n-many keys inserted in total. At the time k_i was inserted into the table, the average length of the chain associated with a slot of the table is $\frac{i-1}{m}$ (why?). Given the value of i, the average number of steps before we find k_i in the chain will be $\Theta(1 + \frac{i-1}{m})$. Now, the variable i takes on values over the set $\{1, 2, \ldots, n\}$ and the probability that i takes on any one of these values is $\frac{1}{n}$. Therefore, the average running time for a successful search is given by
>
> $$\frac{1}{n}\sum_{i=1}^{n}\left(1 + \frac{i-1}{m}\right) = 1 + \frac{1}{nm}\sum_{i=1}^{n}(i-1) = 1 + \frac{\alpha}{2} - \frac{1}{2m}.$$
>
> Hence the observation.

The take-away from all this is the following -- if $\alpha = \frac{n}{m}$ is fixed, then successful/unsuccessful searches take a constant time, on an average. This is very attractive for big data applications.

Section 2.3: Mathematical Analysis of Chaining

Let us suppose we have n keys, and a hash-table of size m. Assume we have perfect uniform hashing. The probability of seeing an i-long chain is given by the expression

$$\begin{aligned} Prob\{\text{there is a chain with length } i\} &= \binom{n}{i}\left(1 - \frac{1}{m}\right)^{n-i}\left(\frac{1}{m}\right)^{i} \\ &= \binom{n}{i}\frac{(m-1)^{n-i}}{m^n} \end{aligned}$$

As before, we have n keys and a hash-table of size m. Let us just look at the first-slot of the hash-table for this discussion. We are to compute the probability that any one item hashes to this slot. Let X_i be the RV that counts the number of keys that hash to the first-slot of the hash-table of the i-th key. That is, $X_i = 0$ or $X_i = 1$ (i.e. either the i-th key was not-assigned/assigned to the first-slot). It is not hard to see that $E\{X_i\} = \frac{1}{m}$. Let $X = X_1 + X_2 + \cdots + X_n$,then $E\{X\}$ is the average number of keys that hash to the first-slot of the hash table. Since the insertion of each key is independent of the others (by assumption), it follows that $E\{X\} = \frac{n}{m}$. There is nothing special about just looking at the first-slot, we will see the same average-value for all slots.

We are now going to compute the number of empty slots in the hash-table. Following the above discussion, the probability that any slot will be empty after we hashed the first key is $(1 - \frac{1}{m})$. Since each hashing-activity is independent of the others, the probability that none of the n-many keys are hashed to a slot in the hash-table is $(1 - \frac{1}{m})^n$. Suppose $X_i = 1$ (resp. $X_i = 0$) if the i-th slot of the hash table is empty (resp. non-empty) after all n-many keys have been hashed. Then the number of empty slots (after all n-many keys have been hashed) is $X_1 + X_2 + \cdots + X_m$. This would mean

$$E\{X_1 + X_2 + \cdots + X_m\} = E\{X_1\} + E\{X_2\} + \cdots + E\{X_m\} = m\left(1 - \frac{1}{m}\right)^n.$$

If $n = m$, then above expression (i.e. average number of empty slots in the hash table after n-many keys have been inserted) simplifies to $n(1 - \frac{1}{n})^n$. This would mean that for large n (and $m = n$, in this discussion), we expect to see $1/e$ of the n-many slots to remain empty. Note that if $n = m$ (and n is large) we expect to see one item per slot in the hash table. All this means is that the (n/e) -empty slots will be "balanced" out by other slots that have more than one item in them.

Continuing with the discussion -- a "*collision*" occurs when a key is hashed to an occupied slot in the hash table. We are going to compute the expected number of collisions. The number of collisions will equal the total number of keys (i.e. n) minus the number of occupied slots (because the first-key hashed to an occupied-slot did not collide with any other key when it was inserted). Therefore

$$\begin{aligned} E\{\text{collisions}\} &= n - E\{\text{occupied slots}\} \\ &= n - m + E\{\text{empty slots}\} \\ &= n - m - m(1 - 1/m)^n. \end{aligned}$$

Let M_i be the event that the maximum chain length (after n-many keys have been inserted into a hash table of length m) is i. Then

$$Prob\{M_i\} \leq m \binom{n}{i} \frac{(m-1)^{n-i}}{m^n}$$

If $m = n$, then the above expression simplifies to

$$Prob\{M_i\} \leq n \binom{n}{i} \frac{(n-1)^{n-i}}{n^n}$$

The rough argument for this is as follows -- the probability that the longest-chain (of length i) is at the first slot of the hash table is less than the probability that there are i-many keys hashed to the first slot, while less than i-many keys are hashed to the other slots. That is, we are just looking at the first slot of the hash table (and assuming this slot has the largest chain of length i; while other slots have equal-or-less-than-i long chains). The probability of this special-event is bounded above by

$$\binom{n}{i} \frac{(m-1)^{n-i}}{m^n}$$

The compound-event that some slot has the maximum chain length of i, is no more than the m-many sum of similar expression above. It can be shown that when $m = n$,

$$\binom{n}{i} \frac{(n-1)^{n-i}}{n^n} \leq \frac{e^i}{i^i}$$

and therefore the probability that the maximum chain length is i is at most $\frac{ne^i}{i^i}$. Additionally, when $m = n$, and we has n-many items into a hash table of size m, the expected maximum chain length is $O(\log n / \log \log n)$.

Section 2.4: Open Addressing

In this scheme $\alpha = \frac{n}{m} < 1$ -- that is, the number of keys you wish to store is less than the number "buckets." In addition, there are m-many hash functions (as opposed to just one, as with chaining)

$h_i : K \rightarrow \{1, 2, \ldots, m\}$ where $i \in \{1, 2, \ldots, m\}$, where each function satisfies the property that $Prob\{h_i(\bullet) = j\} = \frac{1}{m}$. One way in which this can be done is to let $h_i(\bullet)$ be a permutation of the set $\{1, 2, \ldots, m\}$. For example consider the hash functions $h_i(\bullet), i \in \{1, 2, \ldots, 9\}$ that arise out of six (randomly assigned) permutations as shown in figure 2. Note, $n = 6$ and $m = 9$, for this example. For each k_i, starting with $j = 1$, we place k_i at the spot/bucket $h_j(k_i)$ only if it is unoccupied; otherwise, we let $j \leftarrow j + 1$ and try again. We are guaranteed that this process will eventually find an empty/unoccupied spot/bucket for k_i (why?). The final outcome would be the following assignments: $k_1 \rightarrow 3, k_2 \rightarrow 1, k_3 \rightarrow 4, k_4 \rightarrow 5, k_5 \rightarrow 9$ and $k_6 \rightarrow 7$.


Forwarding
Figure 2: Open Addressing Illustration, where n = 6 and m = 9. The m-many hash functions are obtained using a randomly assigned permutation of the set {1,2,...,m} to each key.

In this context, the number of times we have to find a few hash function (because the previous one assigned the key to an occupied bucket) is how we measure running time. In the parlance, we are using the "number of probes" to measure running time. The following observations are about the average number of probes we need for unsuccessful/successful attempts at finding a key.

> **Observation 3**: If $\alpha = \frac{n}{m} < 1$, the average number of probes in an unsuccessful search is at most $\frac{1}{1-\alpha}$.
> **Proof (Sketch)**: If you have an unsuccessful search, then we must have gone through a series of probes, where (a) all but the last probe took us to an occupied slot, and (b) the slot resulting from the last probe was empty. In this context, let us suppose
>
> $$\begin{aligned} p_i &= Prob\{\text{the last bucket after } \mathbf{exactly}\ i \text{ probes was empty}\} \\ &= Prob\{\mathbf{exactly}\ i \text{ probes accessed occupied-slots}\}. \end{aligned}$$
>
> By definition, $\forall i > n, p_i = 0$. Then, the average number of probes is $1 + \sum_{i=0}^{\infty} ip_i$, which is what we need to compute. To get there, let us define
>
> $$\begin{aligned} q_i &= Prob\{\mathbf{atleast}\ i \text{ probes found empty buckets}\} \\ &= Prob\{\mathbf{atleast}\ i \text{ probes accessed occupied-slots}\}. \end{aligned}$$
>
> Then
>
> $$\sum_{i=0}^{\infty} ip_i = \sum_{i=1}^{\infty} q_i$$
>
> Why? If we have a discrete RV $X \in \{0, 1, \ldots\}$, then
> $E\{X\} = \sum_{i=0}^{\infty} i \times Prob\{X = i\} = \sum_{i=0}^{\infty} i \times (Prob\{X \geq i\} - Prob\{X \geq i + 1\}) = \sum_{i=}^{\infty}$
> .
>
> We know that $q_1 = \frac{n}{m}, q_2 = \frac{n}{m}\frac{n-1}{m-1}, \cdots, q_i = \frac{n}{m}\frac{n-1}{m-1}\cdots\frac{n-i+1}{m-i+1}$. That is, $q_i \leq (\frac{n}{m})^i (= \alpha^i)$. Therefore, $1 + \sum_{i=1}^{\infty} q_i \leq 1 + \alpha + \alpha^2 + \cdots = \frac{1}{1-\alpha}$, which establishes the observation.

To see the import of this observation, if $\alpha = 0.5$ (i.e. the hash table is 50% full), then the average number of probes before we figure out something is not in the hash table is less than 2. Likewise, if $\alpha = 0.9$ (i.e. the table is 90% full), the average number of probes before we know that the key is not in the table is less than 10. Note, $\alpha = \frac{n}{m}$, which means the ratio of n and m (not their numerical values) play a role in deciding this average.

The proof of the above observation also serves as a proof of the following observation.

> **Observation 4**: Inserting a new element/key requires at most $\frac{1}{1-\alpha}$ probes, on an average.

The following observation is about the average number of probes in a successful search.

> **Observation 5**: The average number of probes in a successful search is at most $\frac{1}{\alpha}\ln\frac{1}{1-\alpha}+\frac{1}{\alpha}$
> **Proof (Sketch)**: Let us denote the key by k (and it exists in the hash table). When k was first inserted into the table, let us say it was the $(i+1)$-th key among the sequence of n-many keys inserted in total. From the previous observations, the average number of probes used to place k is at most $\frac{1}{1-i/m}=\frac{m}{m-i}$, for a known i. Averaging over all i's we get $\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i}=\frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i}=\frac{1}{\alpha}(H_m-H_{m-n})$, where $H_i=\sum_{j=1}^{i}\frac{1}{j}$ is the i-th *Harmonic Number*.
>
> It is known that $\ln i\leq H_i\leq(\ln i+1)$, consequently
>
> $$\frac{1}{\alpha}(H_m-H_{m-n})\leq\frac{1}{\alpha}(1+\ln m-\ln(m-n))=\frac{1}{\alpha}\ln\frac{1}{1-\alpha}+\frac{1}{\alpha}.$$

To appreciate the import of this observation, let us say $\alpha=0.5$ (i.e. the hash table is 50% full), then the average number of probes is less than 3.38; if $\alpha=0.9$, the average number of probes is less than 3.67. That is, for almost all reasonable instances, the average number of probes is less than 4.

One thing that it is a hassle with open addressing is that it is difficult to erase/remove items (why?).

Section 2.5: Bloom Filters

In the hashing methods described earlier, if we have a key that is in (resp. not in) the table, we will definitely know that is in (resp. not in) the table after executing a sequence of chain/probe operations. In the general setting, we have a set of objects S and we are asked if $x\in S$?. The hashing methods will answer this question without error in case $x\in S$ or if $x\notin S$; but, these can be slow for some applications. For example. an automatic spell-checker has to work as you type -- hashing is not an option here.

With *Bloom Filters*, we are willing to put up with an occasional error if $x\notin S$. That is, if $x\in S$, we will know this for sure; if $x\notin S$, we might make a mistake and say $x\in S$ on occasion. This *false-positive error* issue is something that is tolerable when it comes to spell-checking, password-security, etc. The Bloom Filter is essentially an n-dimensional boolean vector that has to be constructed for the set S. This done with the help of k-many hash functions $\{h_1(\bullet),h_2(\bullet),\ldots,h_k(\bullet)\}$, where $\forall i\in\{1,2,\ldots,n\},h_i:S\rightarrow\{1,2,\ldots,n\}$, and $Prob\{h_i(x)=j\}=\frac{1}{n},\forall i\in\{1,2,\ldots,n\},\forall x\in S$.

Here is how we build the Bloom Filter (i.e. the n-dimensional boolean vector). We start with a vector of all zeros. For each $x\in S,\forall i\in\{1,2,\ldots,k\}$, we make sure the $h_i(x)$-th bit of the Bloom Filter equal to 1. Membership in S is tested by the following property:

$$(x\in S)\Leftrightarrow(h_i(x)=1,\forall i\in\{1,2,\ldots,k\}).$$

A *false negative error*, where $x\in S$, but we declare $x\notin S$, cannot occur under this arrangement. However, a *false-positive error*, where $x\notin S$, but we mistakenly declare $x\in S$, can occur. Figure 3 presents an illustrative example.


Forwarding
Figure 3: An illustration of the possibility of a false positive error in the con- struction of a Bloom Filter. Bits 2, 4, 5 and 8 are set to 1 by the two hash functions for $y_1 \in S$ and $y_2 \in S$. But, for y_3 does not belong to S, the same two hash functions check/assign y_3 to bits 4 and 8. Under this construction there will be a false positive error for y_3.

To get at the probability of a false positive error, let us suppose n is the size/dimension of the Bloom Filter, $m = card(S)$. and k is the number of hash functions used in the construction of the filter. The design of the Bloom Filter will pick a value for n and k that guarantees the probability of a false positive error is below an acceptable value. We note that for

$$
\begin{aligned}
x \in S, i \in \{1,2,\ldots,k\}, j \in \{1,2,\ldots,n\}, Prob\{h_i(x) \neq j\} &= 1 - \frac{1}{n} \\
\Rightarrow x \in S, j \in \{1,2,\ldots,n\}, Prob\{\forall i \in \{1,2,\ldots,k\}, h_i(x) \neq j\} &= \left(1 - \frac{1}{n}\right)^k \\
\Rightarrow j \in \{1,2,\ldots,n\}, Prob\{\forall x \in S, \forall i \in \{1,2,\ldots,k\}, h_i(x) \neq j\} &= \left(1 - \frac{1}{n}\right)^{km} \\
&= \left\{\left(1 - \frac{1}{n}\right)^n\right\}^{km/n} \\
&\approx e^{-km/n} \\
\Rightarrow j \in \{1,2,\ldots,n\}, Prob\{j\text{-th bit is set to 1 by some hash function}\} &= (1 - e^{-km/n}) \\
\Rightarrow Prob\{\text{False Positive Error}\} &= (1 - e^{-km/n})^k.
\end{aligned}
$$

If we set $k = \frac{n}{m} \ln 2 \approx (0.7n)/m$, then

$$
Prob\{\text{False Positive Error}\} = (1 - e^{-km/n})^k = (1/2)^k = (1/2)^{(0.7n/m)} = (0.615)^{n/m}.
$$

If $Prob\{\text{False Positive Error}\} = p$, then

$$
p = (0.615)^{n/m} \Rightarrow \frac{n}{m} = \frac{\log_2 p}{\log_2 0.615} = 0.7 \log_2 p.
$$

if $p \leq \frac{1}{1000}$ then $\frac{n}{m} \approx 0.7 \log_2 (1/1000) \approx 7$ and $k = 0.7 \times 7 \approx 5$. That is, if we are happy with a 1/1000 chance of a false positive error, then $k \approx 5$ and $n/m \approx 7$, which is quite good.

Section 2.6: Deletions in Chaining, Open-Addressing and Bloom Filters

Deleting a key in chaining is straightforward (but laborious). If we have to delete a key k_i, we search the chain connected to the slot identified by $h(k_i)$. We are guaranteed to find k_i somewhere in this chain. We delete k_i, and if it is not at the end of the chain, we have to "re-attach" the two split-chains in a predictable manner, and we are good to go.

The process of deleting keys in open-addressing is a little more complex. Take a look at my flipped classroom video on Compass where I did an example of open-addressing. Suppose we deleted k_2 and left its previously occupied slot empty afterwards, we will have a false negative error if we searched for k_3 afterwards (why?). This can be remedied by not leaving the deleted item's slot empty -- we fill it with a dummy-variable. There is a predictable narrative that is needed here -- I trust you can take care of it on your own.

Deletions in Bloom Filters

Consider a Bloom filter that uses three hash functions for each key/object -- $\{h_i(\bullet)\}_{i=1}^{3}$. Let us suppose k_1, k_2 are two keys/objects where $h_2(k_1) = h_1(k_2)$ (that is, k_1 and k_2 share a common bit-position in the Bloom filter, through different hash functions). If we deleted k_1 by zeroing out the three bit positions it hashes to, we will have a false negative error when we search for k_2.

Let us suppose we did not delete k_1 or k_2 from the Bloom Filter, but $k_3 \notin S$ results in a false positive error (note: Bloom Filters can have false positive errors). Let us say we deleted the false positive k_3 for some reason -- we will have at least one false negative error (i.e. we will say something is not in S while it really \underline{is} in S). For this reason, we should not delete any false-positive cases. Or, stated differently, all deletions of keys/objects from Bloom Filters must only be done when the last item in the inventory is removed, and there is none of it left anymore.

The *Counting Bloom Filter* is a modification of the standard Bloom Filter where each slot is a counter (as opposed to a bit that is zero/one). Each slot counts the number of keys/objects that have been hashed to that slot. If an object is to be removed, we just subtract one from the counter associated with each slot the object is hashed to. A little thought should convince you that with a counter for each slot (as opposed to a single-bit) we will not have false negative errors after deletions.

Following the notation of the Bloom Filter section -- let us suppose $card(S) = m$, and the size of the Counting Bloom Filter is n. Let us suppose we use k-many hash functions to place each object into the Counting Bloom Filter. Since there are m objects and k-many hash functions, we can envision a set of mk-many boxes with an integer inside each box that identifies the slot that is assigned to an object from via some hash function. The probability of a specific slot in the Counting Bloom Filter having i-many objects hashed on to it, is given by

$$\binom{mk}{i}\left(\frac{1}{n}\right)^{i}\left(1-\frac{1}{n}\right)^{n-i} = \binom{mk}{i}\frac{(n-1)^{n-i}}{n^n} \leq \binom{mk}{i}\frac{1}{n^i} \leq \frac{(mk)^i}{i!}\frac{1}{n^i}.$$

From Stirling's Approximation we have $i! \approx \sqrt{2\pi i}(i/e)^i$, therefore $1/i! \leq \frac{e^i}{i^i}$, and the above expression simplifies to

$$\left(\frac{mke}{in}\right)^{i}$$

if $k = \frac{n}{m}\ln 2$ and $i = 16$, the above expression evaluates to 1.37×10^{-15}, which is "practically zero." To count up to $i = 16$, we need four bits -- in terms of bits, the Counting Bloom Filter will be $4n$ (as opposed to n-bits for the "regular" Bloom Filter introduced earlier).

The code shown below illustrates the construction of Bloom Filters using the PyProbables API,

In [3]:
```
# IE531: Algorithms for Data Analytics
# Written by Prof. R.S. Sreenivas
#
# Illustration of Bloom Filters using PyProbables
# Taken from https://pyprobables.readthedocs.io/en/latest/quickstart.html#example-usa
#

from probables import (BloomFilter)
blm = BloomFilter(est_elements=1000000, false_positive_rate=0.05)
with open('Data/Siddhartha_Herman_Hesse', 'r', encoding = 'latin-1') as fp:
    for line in fp:
        for word in line.split():
            blm.add(word.lower())  # add each word to the bloom filter!
            # end reading in the file
words_to_check = ['govinda', 'borzoi', 'pray', 'fleches', 'rain']
```

```
for word in words_to_check:
    print (blm.check(word))
```

```
True
False
False
False
True
```

In [4]:
```
# Exporint the Bloom Filter and checking for words afterwards
blm.export('Data/Siddhartha_Herman_Hesse.blm')
blm2 = BloomFilter(filepath='Data/Siddhartha_Herman_Hesse.blm')
print (blm2.check('sutler'))
print (blm2.check('venerable'))
```

```
False
True
```

In [5]:
```
# Trying out the Count-Min Sketch
# from https://pyprobables.readthedocs.io/en/latest/quickstart.html#count-min-sketch
# with some changes
#
from probables import (CountMinSketch)
cms = CountMinSketch(width=100000, depth=5)
with open('Data/Siddhartha_Herman_Hesse', 'r', encoding = 'latin-1') as fp:
    for line in fp:
        for word in line.split():
            cms.add(word.lower())  # add each to the count-min sketch!
words_to_check = ['govinda', 'venerable', 'pray', 'samanas']
for word in words_to_check:
    print(cms.check(word))  # prints: 108, 21, 0, 38
```

```
108
21
0
38
```

Section 3: COUNT-DISTINCT Problem

In this problem we are asked to estimate the number m based on observations of the stream $\langle a_1, a_2, \ldots, a_n \rangle$, where $a_i \in \{1, 2, \ldots, m\}$. Keep in mind that n and m are quite large, and the standard-approach of keeping track of the number of distinct objects we have seen up to now, is not an option.

Section 3.1: The Flajolet-Martin Algorithm

If we used a hash function on each a_i that we saw in the stream, and if these hash functions (are perfect and) assign distinct objects to distinct buckets, we can look at the buckets to infer the number of distinct items in the stream. This is the driver behind the *Flajolet-Martin Algorithm*, which supposes that the more different elements we see in the stream, the more different hash-values we shall see. Whenever we apply a hash function h to a stream element a_i, the bit string $h(a_i)$ will end in some number of 0's, possibly none. Call this number the tail length for a_i and h. Let R be the maximum tail length of any a_i seen so far in the stream. Then we shall use estimate 2^R for the number of distinct elements seen in the stream.

The probability that a given stream element a_i has $h(a_i)$ ending in at least R 0's is 2^{-R}. Suppose there are m distinct elements in the stream. Then the probability that none of them has tail length at least r is

$$\left(1-\frac{1}{2^r}\right)^m = \left(\left(1-\frac{1}{2^r}\right)^{2^r}\right)^{m2^{-r}} \approx e^{-m/2^r}.$$

Therefore, the probability that one of them has a tail length of at least r is $(1-e^{-m/2^r})$. If $m \gg 2^r$, this expression equals 1 -- that is, the probability that we will find a tail of length at least R approaches 1. If $m \ll 2^r$, this expression equals 0 -- that is, the probability that we will find a tail of length at least r approaches 0. From an "*engineering viewpoint*," if R is the maximum tail length of any a_i, the estimate of $m \approx 2^R$ is unlikely to be either much too high or much too low.

Some Practical Issues

Just relying on a single estimate of $m \approx 2^R$ is not prudent. The standard-approach of taking the average of several $m_i \approx 2^{R_i}$ will not work either -- as the arithmetic-mean is highly susceptible to outliers (and there can be some outlandishly large outliers here). If we used the median (which is immune to large outliers), we have the problem of the estimate of m being a power-of-2. The solution is to group the trials into "bins" -- the arithmetic-mean of the n-estimate for each bin is computed first; following this, the median of the arithmetic-means of the set of bins is presented as the n-estimate.

The other thing you will see (cf. this Wikipedia page, for example) is that Flajolet and Marin suggest the use of $2^R/\Phi$ as the m-estimate, where $\Phi \approx 0.77351$. The exact derivation of this "correction" factor can be found in their original paper.

Also, there is nothing sacrosanct about looking for a tail of zeros, we could have looked for a tail of ones (or, a tail of repeating zeros and ones, for that matter). The analysis would still be the same.

There are several custom-built packages that implements (variants of) these procedures. You can take a look at the illustrative examples in each of these sites. I am skipping them in the interest of time (and you do not need me to do the cut-n-paste exercise).

PDSA: Probabilistic Data Structures and Algorithms in Python

Cardinality Estimation using the Flajolet-Martin algorithm

Here is some sample code that implements the *Flajolet-Martin Algorithm*, that uses xxhash. We have to take many estimates of the number of words in the (small) Twitter Data set file. The correct answer

---

In [9]:
```
# IE531: Algorithms for Data Analytics
# Written by Prof. R.S. Sreenivas
#
# Illustration of the Flajolet-Martin Algorithm for estimating the number of distinct
# It was inspired by the Python code here https://stackoverflow.com/questions/6552486
#
import xxhash
import math
import numpy as np
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
/var/folders/4d/qfln2dxs7x575056pj_hd31m0000gn/T/ipykernel_3937/3077270848.py in <cel
l line: 7>()
      5 # It was inspired by the Python code here https://stackoverflow.com/question
s/65524861/flajolet-martin-algorithm-implementation
      6 #
----> 7 import xxhash
      8 import math
      9 import numpy as np

ModuleNotFoundError: No module named 'xxhash'
```

In [ ]:
```python
def return_trailing_zeroes(s):
    s = str(s)
    rev = s[::-1]
    count = 0

    for i in rev:
        if (i == '0'):
            count = count + 1
        else:
            break

    return count
```

In [ ]:
```python
def gethash_xxhash(line):
    num=abs(xxhash.xxh32(line, np.random.randint(0, 2**32-1)).intdigest())
    return num
```

In [ ]:
```python
no_of_trials = 100
average = 0
for i in range(no_of_trials) :
    fp=open("Data/Twitter_Words.txt","r")
    h_max=0

    for line in fp:
        hash_value_1 = gethash_xxhash(line)
        binary_1 = format(hash_value_1, '032b')
        t1 = return_trailing_zeroes(binary_1)

        if t1>h_max:
            h_max=t1

    average = average + (2**h_max)
    fp.close()

print ('The average number of distinct words after ' + str(no_of_trials) + ' trials =
print ('The correct answer (and I will show it using C++ code in Class) is 3709')
print ('The estimate will be off, it will perform better with files with more words')
print ('It would be too slow to try it with Python, I will show some C++ Code in clas
```

Section 4: FREQUENCY MOMENTS Problem

We have a stream $\langle a_1, a_2, \ldots, a_n \rangle$ where each $a_i \in \{1, 2, \ldots m\}$. We have the associated frequency vector $(f_1, f_2, \ldots f_m)$, where $\sum_{i=1}^{m} f_i = n$. We are looking for an algorithm that gives us an estimate of the k-th frequency moment: $F_k = \sum_{i=1}^{m} f_i^k$.

Note that if $k = 0$, and we take $0^0 = 0$, the above problem reduces to the COUNT DISTINCT problem, which was discussed earlier. If $k = 1$, then the above problem reduces to the problem of

estimating n, which is trivially solved by a counter.

For $k \geq 2$, we do the following -- we pick an uniform random position of the stream (i.e. pick some $i \in \{1, 2, \ldots, n\}$). Let us suppose $a_i = j$. We then count the number of times j appears in the subsequent portion of the stream. Let us say we saw r-many j's in the stream afterwards. At the end of the stream, we output $n(r^k - (r-1)^k)$ as our estimate of F_k. Usually, we will run many copies of this estimator to get a final estimate with better guarantees.

To see why this algorithm works, it would help to interpret the above algorithm as involving two "random events." The first, involves the selection of j as the steam-item, which appears a total of f_j -many times in the stream, to be tracked subsequently. The second, involves which of the f_j-many stream-positions of j is to be tracked subsequently.

The probability of picking j is $\frac{f_j}{n}$. Following this, if we picked i-th among the f_j-many positions j occurs in the stream, then $r = (f_j - i + 1)$ (why?). The expected value of $n(r^k - (r-1)^k)$ is given by

$$\begin{aligned} E\{n(r^k - (r-1)^k)\} &= \sum_{j=1}^{m} \frac{f_j}{n} \left\{ \sum_{i=1}^{f_j} \frac{1}{f_j} \left[n \left((f_j - i + 1)^k - (f_j - i)^k \right) \right] \right\} \\ &= \sum_{j=1}^{m} \underbrace{\sum_{i=1}^{f_j} \left\{ (f_j - i + 1)^k - (f_j - i)^k \right\}}_{= f_j^k \ (\text{"Telescope"})} \\ &= \sum_{j=1}^{m} f_j^k. \end{aligned}$$

It is not hard to show

$$Var\{n(r^k - (r-1)^k)\} = n \sum_{j=1}^{m} \sum_{i=1}^{f_j} \left(i^k - (i-1)^k \right)^2,$$

which can be very large. There are several methods in the literature (and many are cookie-cutter variance reduction techniques) to bring down the variance. I am skipping it in the interest of time.

Section 5: Random Sampling of Streams

Keep in mind that we do not know what n is going to be, as we observe the stream. For this reason, we have to pick the sampled stream-positions carefully. We pick the sampled-positions from the early part of the stream, we will be biased in favor of those values that appeared in the beginning. If we waited for too long, then we might not have enough to improve the quality of our estimate.

Let us say we want s-many estimates of F_k running at any time. So far, we have seen k-many stream positions ($k \leq n$). When the $(k+1)$-th stream sample arrives, we pick that sample as the one to be tracked, with probability $\frac{s}{k+1}$. If it is picked, then we throw away (uniformly random) one of the s-many running-estimates and replace it with a_{k+1}. If it not picked, we continue with the previous s-many estimates (as before).

The analysis below is a little unorthodox -- it shows that after the $(k+1)$-th element arrives, effectively the probability of selecting any one of the first k-many elements is also $\frac{s}{k+1}$ (i.e. the s-many estimates are chosen uniformly at random over the $(k+1)$-long stream).

Assume before the $(k+1)$-th stream element arrived, the s-many estimates are chosen uniformly at random for the (previous) k-many stream elements. This would mean the probability of these (previous) k-many stream elements is $\frac{s}{k}$.

Now, suppose the $(k+1)$-th stream element arrives, the probability of picking the $(k+1)$-th stream element is $\frac{s}{k+1}$, by design. If the $(k+1)$-th stream element is picked, one of the (previous) s-many estimates will be dropped. This would mean that $(s-1)$-many of the estimates are the previous/old estimates, and the $(k+1)$-th stream element is the extra/newly-added estimate. The probability of the older estimates gets scaled by $\frac{s-1}{s}$, as a consequence. Therefore, the probability of selecting each of the first k positions under this sampling scheme is

$$\left\{ \underbrace{\left(1 - \frac{s}{k+1}\right)}_{\text{Prob. } (k+1)\text{-th not picked}} + \overbrace{\left(\frac{s}{k+1}\right)}^{\text{Prob. } (k+1)\text{-th picked}} \underbrace{\left(\frac{s-1}{s}\right)}_{\text{reduction factor}} \right\} \left(\frac{s}{k}\right)$$

which simplifies as

$$\left\{1 - \frac{s}{k+1} + \frac{s-1}{k+1}\right\} \left(\frac{s}{k}\right) = \left\{\frac{k+1-s+s-1}{k+1}\right\} \left(\frac{s}{k}\right) = \frac{s}{k+1}.$$

In [ ]: