15-859E: Advanced Algorithms **CMU, Spring 2015**
Lecture #20: Streaming Computation: Computing Moments March 2, 2015
Lecturer: Anupam Gupta Scribe: Anupam Gupta

Today's lecture will be about a slightly different computational model called the *data streaming* model. In this model you see elements going past in a "stream", and you have very little space to store things. For example, you might be running a program on an Internet router, the elements might be IP Addresses, and you have limited space. You certainly don't have space to store all the elements in the stream. The question is: which functions of the input stream can you compute with what amount of time and space? (For this lecture, we will focus on space, but similar questions can be asked for update times.)

We will denote the stream elements by

$$a_1, a_2, a_3, \ldots, a_t, \ldots$$

We assume each stream element is from alphabet U and takes b bits to represent. For example, the elements might be 32-bit integers IP Addresses. We imagine we are given some function, and we want to compute it continually, on every prefix of the stream. Let us denote $a_{[1:t]} = \langle a_1, a_2, \ldots, a_t \rangle$.

Let us consider some examples. Suppose we have seen the integers

$$3, 1, 17, 4, -9, 32, 101, 3, -722, 3, 900, 4, 32, \ldots \tag{$\diamond$}$$

- Computing the sum of all the integers seen so far? $F(a_{[1:t]}) = \sum_{i=1}^{t} a_i$. We want the outputs to be
$$3, 4, 21, 25, 16, 48, 149, 152, -570, -567, 333, 337, 369, \ldots$$
If we have seen T numbers so far, the sum is at most $T2^b$ and hence needs at most $O(b + \log T)$ space. So we can just keep a counter, and when a new element comes in, we add it to the counter.

- How about the maximum of the elements so far? $F(a_{[1:t]}) = \max_{i=1}^{t} a_i$. Even easier. The outputs are:
$$3, 1, 17, 17, 17, 32, 101, 101, 101, 101, 900, 900, 900$$
We just need to store b bits.

- The median? The outputs on the various prefixes of ($\diamond$) now are
$$3, 1, 3, 3, 3, 3, 4, 3, \ldots$$
And doing this will small space is a lot more tricky.

- ("distinct elements") Or the number of distinct numbers seen so far? You'd want to output:
$$1, 2, 3, 4, 5, 6, 7, 7, 8, 8, 9, 9, 9 \ldots$$

- ("heavy hitters") Or the elements that have appeared most often so far? Hmm...

You can imagine the applications of the data-stream model. An Internet router might see a lot of packets whiz by, and may want to figure out which data connections are using the most space? Or how many different connections have been initiated since midnight? Or the median (or the 90^{th} percentile) of the file sizes that have been transferred. Which IP connections are "elephants" (say the ones that have used more than 0.01% of your bandwidth)? Even if you are not working at "line speed",[1] but just looking over the server logs, you may not want to spend too much time to find out the answers, you may just want to read over the file in one quick pass and come up with an answer. Such an algorithm might also be cache-friendly. But how to do this?

Two of the recurring themes will be:

- Approximate solutions: in several cases, it will be impossible to compute the function exactly using small space. Hence we'll explore the trade-offs between approximation and space.
- Hashing: this will be a very powerful technique.

1 Streams as Vectors, and Additions/Deletions

An important abstraction will be to view the stream as a vector (in high dimensional space). Since each element in the stream is an element of the universe U, you can imagine the stream at time t as a vector $\mathbf{x}^t \in \mathbb{Z}^{|U|}$. Here

$$\mathbf{x}^t = (x_1^t, x_2^t, \ldots, x_{|U|}^t)$$

and x_i^t is the number of times the i^{th} element in U has been seen until time t. (Hence, $x_i^0 = 0$ for all $i \in U$.) When the next element comes in and it is element j, we increment x_j by 1.

This brings us a extension of the model: we could have another model where each element of the stream is either a new element, or an old element departing.[2] Formally, each time we get an *update* a_t, it looks like $(\mathtt{add}, e)$ or $(\mathtt{del}, e)$. We usually assume that for each element, the number of deletes we see for it is at most the number of adds we see — the running counts of each element is non-negative. As an example, suppose the stream looked like:

$$(\mathtt{add}, A), (\mathtt{add}, B), (\mathtt{add}, A), (\mathtt{del}, B), (\mathtt{del}, A), (\mathtt{add}, C), \ldots$$

and if A was the first element of U, then x_1 would be $1, 1, 2, 2, 1, 1, \ldots$.

This vector notation allows us to formulate some of the problems more easily:

- The total number of elements currently in the system is just $\|\mathbf{x}\| := \sum_{i=1}^{|U|} x_i$. (This is easy.)
- We might also want to estimate the norms $\|\mathbf{x}\|_2, \|\mathbf{x}\|_p$ of the vector $\mathbf{x}$.
- The number of distinct elements is the number of non-zero entries in $\mathbf{x}$. (You'll see one way to do this in the next HW.)

Let's consider the (non-trivial) problems one by one.

[1] Such a router might see tens of millions of packets per second.

[2] In data stream jargon, the addition-only model is called the *cash-register* model, whereas the model with both additions and deletions is called the *turnstile model.* I will not use this jargon.

2 Computing Moments

Recall that $\mathbf{x}^t$ was the vector of frequencies of elements seen so far. Several interesting problems can be posed as computing various norms of $\mathbf{x}^t$: in particular the 2-norm

$$\|\mathbf{x}^t\|_2 = \sqrt{\sum_{i=1}^{|U|} (x_i^t)^2},$$

and the 0-norm (which is not really a norm)

$$\|\mathbf{x}^t\|_0 := \text{ number of non-zeroes in } \mathbf{x}^t.$$

For ease of notation, we use the following notation: $F_0 := \|\mathbf{x}^t\|_0$, and for $p \geq 1$,

$$F_p := \sum_{i=1}^{|U|} (x_i^t)^p. \tag{20.1}$$

Today we'll see a way to compute F_2; we'll see ways to compute F_0 (and maybe extensions from F_2 to F_p) in the homeworks.

2.1 Computing F_2

The "second moment" F_2 of the stream is often called the "surprise number" (since it captures how uneven the data is). This is also the *size of the self-join*. Clearly we can store the entire vector $\mathbf{x}$ and compute this, but we'll have to store $|U|$ counts. How to reduce this space?

Here's an algorithm:

```
Pick a random hash function from some hash family $H$ mapping $U \to \{-1, +1\}$.
Maintain counter $C$, which starts off at zero.
    On update $(add, i) \in U$, increment the counter $C \to C + h(i)$.
    On update $(delete, i) \in U$, decrement the counter $C \to C - h(i)$.
    On query about the value of $F_2$, reply with $C^2$.
```

Aside: This estimator is often called the "tug-of-war" estimator: the hash function randomly partitions the elements into two parties (those mapping to 1, and those to -1), and the counter keeps the difference between the sizes of the two parties.

2.1.1 Properties of the Hash Family

Definition 20.1 (k-universal hash family)**.** H is *k-universal* (also called *uniform* and *k-wise independent*) mapping U to some set R if all *distinct* $i_1, \ldots, i_k \in U$ and for any values $\alpha_1, \ldots, \alpha_k \in R$,

$$\Pr_{h \leftarrow H}\left[\bigwedge_{j=1..k} (h(i_j) = \alpha_j)\right] = \frac{1}{|R|^k}. \tag{20.2}$$

We want our hash family to be 4-universal from U to $R = \{-1, 1\}$: this implies the following.

- at for any i,
$$\Pr_{h \leftarrow H}[h(i) = 1] = \Pr_{h \leftarrow H}[h(i) = -1] = \frac{1}{2}.$$
- for distinct i, j, k, l, we have
$$E[h(i) \cdot h(j) \cdot h(k) \cdot h(l)] = E[h(i)] \cdot E[h(j)] \cdot E[h(k)] \cdot E[h(l)].$$
$$E[h(i) \cdot h(j)] = E[h(i)] \cdot E[h(j)].$$

2.2 The Analysis

Hence, having seen the stream that results in the frequency vector $\mathbf{x} \in \mathbb{Z}_{\geq 0}^{|U|}$, the counter will have the value $C = \sum_{i \in U} x_i \, h(i)$.

2.2.1 The Expectation

Does $E[C^2]$ at least have the right expectation? It does:

$$\begin{aligned} E[C^2] &= E[\sum_{i,j} (h(i)x_i \cdot h(j)x_j)] = \sum_{i,j} x_i x_j E[(h(i) \cdot h(j))] \\ &= \sum_i x_i^2 E[h(i) \cdot h(i)] + \sum_{i \neq j} \sum_{i,j} x_i x_j E[h(i)] \cdot E[h(j)] \\ &= \sum_i x_i^2 = F_2. \end{aligned}$$

So in expectation we are correct!

2.2.2 The Variance

Recall that $\mathrm{Var}(C^2) = E[(C^2)^2] - E[C^2]^2$, so let us calculate

$$\begin{aligned} E[(C^2)^2] = E[\sum_{p,q,r,s} h(p)h(q)h(r)h(s)x_p x_q x_r x_s] &= \\ &= \sum_p x_p^4 E[h(p)^4] + 6 \sum_{p<q} x_p^2 x_q^2 E[h(p)^2 h(q)^2] + \text{ other terms} \\ &= \sum_p x_p^4 + 6 \sum_{p<q} x_p^2 x_q^2. \end{aligned}$$

This is because all the other terms have expectation zero. Why? The terms like $E[h(p)h(q)h(r)h(s)]$ where p, q, r, s are all distinct, all become zero because of 4-universality. Terms like $E[h(p)^2 h(r)h(s)]$ become zero for the same reason. It is only terms like $E[h(p)^2 h(q)^2]$ and $E[h(p)^4]$ that survive, and since $h(p) \in \{-1, 1\}$, they have expectation 1.

So

$$\mathrm{Var}(C^2) = \sum_p x_p^4 + 6 \sum_{p<q} x_p^2 x_q^2 - (\sum_p x_p^2)^2 = 4 \sum_{p<q} x_p^2 x_q^2 \leq 2E[C^2]^2.$$

What does Chebyshev say then?

$$\mathbf{Pr}[|C^2 - E[C^2]| > \varepsilon E[C^2]] \leq \frac{\mathrm{Var}(C^2)}{(\varepsilon E[C^2])^2} \leq \frac{2}{\varepsilon^2}.$$

This is pretty pathetic: since ε is usually less than 1, the RHS usually more than 1.

2.2.3 Now to Reduce the Variance

But if we take a collection of k such independent counters $C_1, C_2, \ldots, C_k$, and given a query, take their average $\overline{C} = \frac{1}{k} \sum_i C_i$, and return $\overline{C}^2$. The expectation of the average remains the same, but the variance falls by a factor of k. And we get

$$\mathbf{Pr}[|\overline{C}^2 - E[\overline{C}^2]| > \varepsilon E[\overline{C}^2]] \leq \frac{\mathrm{Var}(\overline{C}^2)}{(\varepsilon E[\overline{C}^2])^2} \leq \frac{2}{k\varepsilon^2}.$$

So, our probability of error on any query is at most δ if we take $k = \frac{2}{\varepsilon^2\delta}$.

In summary: if we keep $k = \frac{2}{\varepsilon^2\delta}$ independent counters, each one uses a 4-universal hash function (which requires $O(\log U)$ random bits to store), then we are correct on any query w.p. $1-\delta$.

Aside: A bunch of students (Jason, Anshu, Aram) proposed that for the p^{th}-moment calculation we should use $2p$-wise independent hash functions from U to R, where $R = \{1, \omega, \omega^2, \ldots, \omega^{p-1}\}$, the p primitive roots of unity. Again, we set $C := \sum_{i \in U} x_i h(i)$, and return the real part of C^p as our estimate. This approach has been explored by Ganguly in this paper. Some calculations (and elbow-grease) show that $E[C^p] = F_p$, but it seems that naively $\mathbf{Var}(C^p)$ tends to grow like F_2^p instead of F_k^p; this leads to pretty bad bounds. Ganguly's paper gives some ways of controlling the variance.

BTW, there is a lower bound saying that any algorithm that outputs a 2-approximation for F_k requires at least $|U|^{1-2/k}$ bits of storage. Hence, while we just saw that for $k = 2$, we can get away with just $O(\log |U|)$ bits to get a $O(1)$-estimate, for $k > 2$ things are much worse.

2.3 A Matrix View of our Estimator

Here's a equivalent, convenient way of looking at this estimator. Take a matrix S of dimensions $k \times D$, where $D = |U|$. Observe that S is a "fat and short" matrix, since $k = O(\varepsilon^{-2}\delta^{-1})$ is small and $D = |U|$ is huge. Now pick k independent hash functions $h_1, h_2, \ldots, h_k$ from the 4-universal hash family, and set

$$S_{ij} := \frac{1}{\sqrt{k}} h_i(j).$$

You should check that the vector $(C_1, C_2, \ldots, C_k)^\intercal$ of values of the k counters is nothing other than

$$\sqrt{k} \cdot S\mathbf{x}$$

And that the estimate $\overline{C}^2 = \left(\frac{1}{k}\sum_{i=1}^k C_i\right)^2$ you return is just

$$\|S\mathbf{x}\|_2^2$$

Aside: Observe this is very similar to the construction for JL: we're now a bit worse since we've got a taller matrix with $k = O(\varepsilon^{-2}\delta^{-1})$ rows instead of $k = O(\varepsilon^{-2}\log\delta^{-1})$.

However, the matrix entries are not required to be fully independent (as in JL), just 4-wise independent. If we required full independence, we would need to store kD bits of information. This way we need to store k different 4-wise independence hash functions, which means a total of $O(k \log D)$ bits.

Let us record two more properties of this construction:

Theorem 20.2 (Tug-of-War Sketch)**.** *Take a $k \times D$ matrix S whose columns are 4-wise independent $\{\frac{1}{\sqrt{k}}, \frac{-1}{\sqrt{k}}\}$-valued random variables. Then for any $\mathbf{x}, \mathbf{y} \in \mathbb{R}^D$, we have*

- $E[\,\langle S\mathbf{x}, S\mathbf{y}\rangle\,] = \langle \mathbf{x}, \mathbf{y}\rangle$.
- $\mathrm{Var}(\,\langle S\mathbf{x}, S\mathbf{y}\rangle\,) = \frac{2}{k} \cdot \|\mathbf{x}\|_2^2 \|\mathbf{y}\|_2^2$.

The proofs are very similar to the calculations in Sections 2.2.1 and 2.2.2; note that plugging in $\mathbf{y} = \mathbf{x}$ gives us exactly the expression we got above.

Citations: This scheme is due to the Gödel prize winning paper of Noga Alon, Yossi Matias, and Mario Szegedy. There has been a lot of interesting work on moment estimation: see, e.g., this STOC 2011 paper of Daniel Kane, Jelani Nelson, Ely Porat and David Woodruff on getting lower bounds for ℓ_p-norms of the vector x, and the many references therein.

3 An Application: Approximate Matrix Multiplication

Suppose we want to multiply two square matrices $A, B \in \mathbb{R}^{n \times n}$. But we are happy with approximate solutions, we don't really want the precise answer. We are happy with some matrix C such that

$$C \approx AB.$$

How should we measure the error? We could require that the answer is close entry-wise to the actual answer. That is a much harder problem. Let's aim for something weaker: we want the "aggregate error" to be small in some sense.

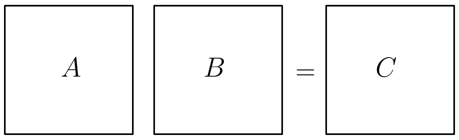
Formally, the *Frobenius norm* of matrix M is

$$\|M\|_F := \sqrt{\sum_{i,j} M_{ij}^2}.$$

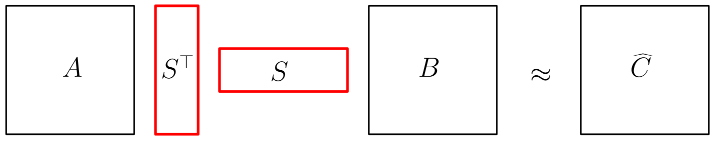
(It's as though you think of the matrix as just a vector and look at its Euclidean length.) Say the guarantee for approximate matrix multiplication we want is that

$$\|C - AB\|_F^2 \leq \text{ small.}$$

Here's the idea: we want to do this—



This normally takes $O(n^3)$ time. Suppose we could find a "fat and short" matrix S (say a $k \times n$ matrix, for $k \ll n$) such that $AB \approx AS^\intercal SB$. By associativity of matrix multiplication, we could first compute $(AS^\intercal)$ and (SB) in times $O(n^2k)$, and then multiply the results in time $O(nk^2)$.



And what is this matrix S? We claim the matrix S from the previous section works pretty well, where we set $D = n$. The idea is this. The ij^{th} entry of the product AB is the dot product of the i^{th} row $A_{i\star}$ of A with the j^{th} column $B_{\star j}$ of B. But the properties of S from Theorem 20.2 give us that

$$E[(A_{i\star}S^\intercal) \cdot (SB_{\star j})] = A_{i\star} \cdot B_{\star j} = (AB)_{ij}$$

Indeed, entries of the error matrix $Y = (AB) - (AS^\intercal SB)$ satisfy

$$E[Y_{ij}] = 0$$

and

$$E[Y_{ij}^2] = \text{Var}(Y_{ij}) + E[Y_{ij}]^2 = \text{Var}(Y_{ij}) \leq \tfrac{2}{k}\|A_{i\star}\|_2^2\|B_{\star j}\|_2^2.$$

So

$$E[\|AB - AS^\intercal SB\|_F^2] = E[\sum_{ij} Y_{ij}^2] = \sum_{ij} E[Y_{ij}^2] = \tfrac{2}{k} \sum_{ij} \|A_{i\star}\|_2^2 \|B_{\star j}\|_2^2$$
$$= \tfrac{2}{k} \|A\|_F^2 \|B\|_F^2.$$

Finally, setting $k = \frac{2}{\varepsilon^2 \delta}$ and using Markov's inequality, you can say that for any fixed $\varepsilon > 0$, we can compute an approximate matrix product $C := AS^\intercal SB$ such that

$$\|AB - C\|_F \leq \varepsilon \cdot \|A\|_F \|B\|_F \qquad \text{with probability at least } 1 - \delta,$$

in time $O(\frac{n^2}{\varepsilon^2 \delta})$. (If the matrix is sparse and contains only $M \ll n^2$ entries, the time for the matrix product can be reduced further.)

Citations: The approximate matrix product question has been considered often, e.g., by Cohen and Lewis using a random-walks approach. This algorithm is due to Tamás Sarlós; his paper gives better results as well as extensions to computing SVDs faster. Better bounds have subsequently been given by Clarkson and Woodruff.

4 Optional: Computing F_0, the Number of Distinct Elements*

Our last example today will be to compute F_0, the number of distinct elements seen in the data stream, but in the addition-only model, with no deletions. (We'll see another approach in a HW.)

4.1 A Simple Lower Bound

Of course, if we store $\mathbf{x}$ explicitly (using $|U|$ space), we can trivially solve this problem exactly. Or we could store the (at most) t elements seen so far, again we could give an exact answer. And indeed, we cannot do much better if we want no errors. Here's a proof sketch for deterministic algorithms (one can extend this to randomized algorithms with some more work).

Lemma 20.3 (A Lower Bound)**.** *Suppose a deterministic algorithm correctly reports the number of distinct elements for each sequence of length at most N. Suppose $N \leq 2|U|$. Then it must use at least $\Omega(N)$ bits of space.*

Proof. Consider the situation where first we send in some subset S of $N-1$ elements distinct elements of U. Look at the information stored by the algorithm. We claim that we should be able to use this information to identify exactly which of the $\binom{|U|}{N-1}$ subsets of U we have seen so far. This would require

$$\log_2 \binom{|U|}{N-1} \geq (N-1)\big(\log_2 |U| - \log_2(N-1)\big) = \Omega(N)$$

bits of memory.[3]

OK, so why should we be able to uniquely identify the set of elements until time $N-1$? For a contradiction, suppose we could not tell whether we'd seen S_1 or S_2 after $N-1$ elements had come in. Pick any element $e \in S_1 \setminus S_2$. Now if we gave the algorithm e as the N^{th} element, the number of distinct elements seen would be N if we'd already seen S_2, and $N-1$ if we'd seen S_1. But the algorithm could not distinguish between the two cases, and would return the same answer. It would be incorrect in one of the two cases. This contradicts the claim that the algorithm always correctly reports the number of distinct elements on streams of length N. ∎

OK, so we need an approximation if we want to use little space. Let's use some hashing magic.

[3]We used the approximation that $\binom{m}{k} \geq \left(\frac{m}{k}\right)^k$, and hence $\log_2 \binom{m}{k} \geq k(\log_2 m - \log_2 k)$.

4.2 The Intuition

Suppose there are $d = \|\mathbf{x}\|_0$ distinct elements. If we randomly map d distinct elements onto the line $[0,1]$, we expect to see the smallest mapped value at location $\approx \frac{1}{d}$. (I am assuming that we map these elements *consistently*, so that multiple copies of an element go to the same place.) So if the smallest value is δ, one estimator for the number of elements is $1/\delta$.

This is the essential idea. To make this work (and analyze it), we change it slightly: The variance of the above estimator is large. By the same argument, for any integer s we expect the s^{th} smallest mapped value at $\frac{s}{d}$. We use a larger value of s to reduce the variance.

4.3 The Algorithm

Assume we have a hash family H with hash functions $h : U \to [M]$. (We'll soon figure out the precise properties we'll want from this hash family.) We will later fix the value of the parameter s to be some large constant. Here's the algorithm:

```
Pick a hash function $h$ randomly from $H$.
If query comes in at time $t$
  Consider the hash values $h(a_1), h(a_2), \dots, h(a_t)$ seen so far.
    Let $L_t$ be the $s^{th}$ smallest distinct hash value $h(a_i)$ in this set.
  Output the estimate $D_t = \frac{M \cdot s}{L_t}$.
```

The crucial observation is: it does not matter if you see an element e once or multiple times — the algorithm will behave the same, since the output depends on what *distinct* elements we've seen so far. Also, maintaining the s^{th} smallest element can be done by remembering at most s elements. (So we want to make s small.)

How does this help? As a thought experiment, if you had d distinct darts and threw them in the continuous interval $[0, M]$, you would expect the location of the s^{th} smallest dart to be about $\frac{s \cdot M}{d}$. So if the s^{th} smallest dart was at location ℓ in the interval $[0, M]$, you would be tempted to equate $\ell = \frac{s \cdot M}{d}$ and hence guessing $d = \frac{s \cdot M}{\ell}$ would be a good move. Which is precisely why we used the estimate

$$D_t = \frac{M \cdot s}{L_t}.$$

Of course, all this is in expectation—the following theorem argues that this estimate is good with reasonable probability.

Theorem 20.4. *Consider some time t. If H is a uniform 2-universal hash family mapping $U \to [M]$, and M is large enough, then both the following guarantees hold:*

$$\mathbf{Pr}[D_t > 2\,\|\mathbf{x}^t\|_0] \leq \frac{3}{s}, \text{ and} \tag{20.3}$$

$$\mathbf{Pr}[D_t < \frac{\|\mathbf{x}^t\|_0}{2}] \leq \frac{3}{s}. \tag{20.4}$$

We will prove this in the next section. First, some observations. Firstly, we now use the stronger assumption that that the hash family 2-*universal*; recall the definition from Section 2.1.1. Next, setting $s = 8$ means that the estimate D_t lies within $[\frac{\|\mathbf{x}^t\|_0}{2}, 2\|\mathbf{x}^t\|_0]$ with probability at least $1 - (1/4 + 1/4) = 1/2$. (And we can boost the success probability by repetitions.) Secondly, we will see that the estimation error of a factor of 2 can be made $(1 + \varepsilon)$ by changing the parameters s and k.

4.4 Proof of Theorem 20.4

Now for the proof of the theorem. We'll prove bound (20.4), the other bound (20.3) is proved identically. Some shorter notation may help. Let $d := \|\mathbf{x}^t\|_0$. Let these d distinct elements be $T = \{e_1, e_2, \ldots, e_d\} \subseteq U$.

The random variable L_t is the s^{th} smallest distinct hash value seen until time t. Our estimate is $\frac{sM}{L_t}$, and we want this to be at least $d/2$. So we want L_t to be *at most* $\frac{2sM}{d}$. In other words,

$$\mathbf{Pr}[\text{ estimate too low }] = \mathbf{Pr}[D_t < d/2] = \mathbf{Pr}[L_t > \frac{2sM}{d}].$$

Recall T is the set of all d $(= \|\mathbf{x}^t\|_0)$ distinct elements in U that have appeared so far. How many of these elements in T hashed to values greater than $2sM/d$? The event that $L_t > 2sM/d$ (which is what we want to bound the probability of) is the same as saying that fewer than s of the elements in T hashed to values smaller than $2sM/d$. For each $i = 1, 2, \ldots, d$, define the indicator

$$X_i = \begin{cases} 1 & \text{if } h(e_i) \leq 2sM/d \\ 0 & \text{otherwise} \end{cases} \tag{20.5}$$

Then $X = \sum_{i=1}^{d} X_i$ is the number of elements seen that hash to values below $2sM/d$. By the discussion above, we get that

$$\mathbf{Pr}\left[L_t < \frac{2sM}{d}\right] \leq \mathbf{Pr}[X < s].$$

We will now estimate the RHS.

Next, what is the chance that $X_i = 1$? The hash $h(e_i)$ takes on each of the M integer values with equal probability, so

$$\mathbf{Pr}[X_i = 1] = \frac{\lfloor sM/2d \rfloor}{M} \geq \frac{s}{2d} - \frac{1}{M}. \tag{20.6}$$

By linearity of expectations,

$$E[X] = E\left[\sum_{i=1}^{d} X_i\right] = \sum_{i=1}^{d} E\left[X_i\right] = \sum_{i=1}^{d} \mathbf{Pr}\left[X_i = 1\right] \geq d \cdot \left(\frac{s}{2d} - \frac{1}{M}\right) = \left(\frac{s}{2} - \frac{d}{M}\right).$$

Let's imagine we set M large enough so that d/M is, say, at most $\frac{s}{100}$. Which means

$$E[X] \geq \left(\frac{s}{2} - \frac{s}{100}\right) = \frac{49\, s}{100}.$$

So by Markov's inequality,

$$\mathbf{Pr}\left[X > s\right] = \mathbf{Pr}\left[X > \frac{100}{49} E[X]\right] \leq \frac{49}{100}.$$

Good? Well, not so good. We wanted a probability of failure to be smaller than $2/s$, we got it to be slightly less than $1/2$. Good try, but no cigar.

4.4.1 Enter Chebyshev

Recall that $\mathbf{Var}(\sum_i Z_i) = \sum_i \mathbf{Var}(Z_i)$ for pairwise-independent random variables Z_i. (Why?) Also, if Z_i is a $\{0,1\}$ random variable, $\mathbf{Var}(Z_i) \leq E[Z_i]$. (Why?) Applying these to our random variables $X = \sum_i X_i$, we get

$$\mathbf{Var}(X) = \sum_i \mathbf{Var}(X_i) \leq \sum_i E[X_i] = E(X).$$

(The first inequality used that the X_i were pairwise independent, since the hash function was 2-universal.) Is this variance "low" enough? Plugging into Chebyshev's inequality, we get:

$$\mathbf{Pr}[X > s] = \mathbf{Pr}[X > \frac{100}{49}\mu_X] \leq \mathbf{Pr}[|X - \mu_X| > \frac{50}{49}\mu_X] \leq \frac{\sigma_X^2}{(50/49)^2\mu_X^2} \leq \frac{1}{(50/49)^2\mu_X} \leq \frac{3}{s}.$$

Which is precisely what we want for the bound (20.3). The proof for the bound (20.4) is similar and left as an exercise.

> **Aside:** If you want the estimate to be at most $\frac{\|\mathbf{x}^t\|_0}{(1+\varepsilon)}$, then you would want to bound $\mathbf{Pr}[X < \frac{E[X]}{(1+\varepsilon)}]$. Similar calculations should give this to be at most $\frac{3}{\varepsilon^2 s}$, as long as M was large enough. In that case you would set $s = O(1/\varepsilon^2)$ to get some non-trivial guarantees.

4.5 Final Bookkeeping

Excellent. We have a hashing-based data structure that answers "number of distinct elements seen so far" queries, such that each answer is within a multiplicative factor of 2 of the actual value $\|\mathbf{x}^t\|_0$, with small error probability.

Let's see how much space we actually used. Recall that for failure probability 1/2, we could set $s = 12$, say. And the space to store the s smallest hash values seen so far is $O(s \lg M)$ bits. For the hash functions themselves, the standard constructions use $O((\lg M) + (\lg U))$ bits per hash function. So the total space used for the entire data structure is

$$O(\log M) + (\lg U) \text{ bits.}$$

What is M? Recall we needed to M large enough so that $d/M \leq s/100$. Since $d \leq |U|$, the total number of elements in the universe, set $M = \Theta(U)$. Now the total number of bits stored is

$$O(\log U).$$

And the probability of our estimate D_t being within a factor of 2 of the correct answer $\|\mathbf{x}^t\|_0$ is at least 1/2.