1 Introduction

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

$$3, 1, 17, 4, -9, 32, 101, 3, -722, 3, 900, 4, 32, \ldots \qquad (\diamond)$$

- 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.

2 Sampling vs. Hashing

It is natural that we use sampling for some of these problems: after all, we want to whittle down the amount of data to some manageable size. But sometimes you should be careful about how you sample (and why hashing may be good). Here's an example from the Mining of Massive Datasets book.

Suppose you want to figure out the number of "uniques", i.e., the elements that occur exactly once times. One way is: pick 10% of the stream *by picking each element of the stream independently at random with probability* 0.1, look at the number of uniques in the sample, and scale up the answer by 10. But this will mis-calcluate the answer. To see why, suppose the stream of length n has $n/2$ distinct elements that appear just once (the uniques), and $n/4$ more distinct elements that appear exactly twice. (So the correct answer is $n/2$.) The sampled stream has expected length $\frac{1}{10}n$. But now we expect to see $0.1 \times n/2 + n/4(2 \cdot (0.1) - (0.1)^2) \approx \frac{1}{10}n \cdot (1 - \frac{1}{40})$ uniques—the second term takes into account the chance that we see one but not both copies of a repeat. So our estimate will be that $\frac{39}{40}n$ elements in the stream were uniques, which is very wrong! And by a Chernoff bound, this is pretty much what you will see with very high probability (as n gets large).

The problem, of course, was that we were making independent sampling decisions for each element of the stream. What we should have done is to make sure that if an element was sampled, all copies of it were sampled too. And one way of doing this: pick a hash function that maps the universe to the range $[10] = \{0, 1, \ldots, 9\}$. And take the elements that map to 0, say, as part of your sample. Now, if the hash function is (at least) 1-wise independent (i.e., each value in the range is equally likely), then we'll get a 10% sample of the stream that will maintain the fraction of duplicates. *Everything is in expectation*, of course, and the variance of this estimator gets higher, so we have to work around that.

3 A Warm-Up: Majority Element

An element e is called the majority element at time t if it occurs (strictly) more than half the time in the stream until that time. Note that the majority element may change over time. There is at

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

most one majority element; there may be none.

So now it's really a data structure to maintain a multiset. There are two operations: `add(i)`, which adds a copy of element i to the multiset, and `query(i)`, which reports whether i is a majority element at the current time.

Suppose we want an algorithm that is correct all the time. One way to do this is to keep a table of size $|U|$, and keep a tally for each element. Takes $|U| \log n$ bits. Or keep the n elements you have seen, uses $n \log |U|$ bits. Both wasteful. But necessary: we claim you need $\Omega(\min(n, |U|))$ space for a stream of length n over universe U. Here's a sketch for the case $n = |U|$. Let the first $n/2$ stream elements be some $n/2$ distinct elements from U, randomly chosen. Then give $n/2$ copies of a random element i from U. Finally, query i. Now i is a majority element if and only if it also appears in the first half. So loosely, the algorithm has to remember which elements came in the first half—i.e., which of the $\binom{n}{n/2} \approx \frac{2^n}{\sqrt{n}}$ possible options appeared in the first half. And that will take at least $\log_2 \frac{2^n}{\sqrt{n}} = \Omega(n)$ bits.[2]

However, *suppose we allow false positives: if there is a majority element you must answer correctly, else you can answer anyhow you want.* Here's an algorithm (due to R. Boyer and J.S. Moore) that keeps very little information, and spits out at each time its estimate of the majority element. (One element in a variable called `memory`, and one counter.)

```
Initially, memory = empty and counter = 0.

When element a_t arrives
  if (counter == 0)
    set memory = a_t and counter = 1
  else
    if a_t == memory
      counter++
    else
      counter--
      discard a_t
```

At the end, return the element in `memory`.

Suppose there is no majority element, then we'll output some element — that's OK, since we want to output a set of size 1 that contains the majority element (if any). We need to ensure that we do output the majority element, if there is one.

Observe: when we discard an element a_t, we also throw away another (different) element as well. (Decrementing the counter is like throwing away one copy of the element in `memory`.) So every time we throw away a copy of the majority element, we throw away another element too. Since there are fewer than half non-majority elements, we cannot throw away all the majority elements.

4 Streams as Vectors, and Additions/Deletions

A convenient abstraction will be to view the current multiset as a vector (in high dimensional space). Since each element in the stream belongs to the universe U, you can imagine the multieset

[2] This information-theoretic argument can be formalized fairly easily, but we won't do it here.

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 added until time t. (Hence, $x_i^0 = 0$ for all $i \in U$.) When we see `add(i)`, we increment x_i by 1.

This brings us a natural extension of the model: we have `add(i)` which adds a copy of i to the multiset, `delete(i)` which removes a copy of i from the multiset, and `query(i)`, which now asks for how many copies of i are currently in the multiset. [3] We 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, $U = \{A, B, C\}$ and suppose the stream looked like:

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

then the value of $\mathbf{x}$ would initially be $(0, 0, 0)$, then $(1, 0, 0)$, $(1, 1, 0)$, $(2, 1, 0)$, $(2, 0, 0)$, $(1, 0, 0)$, $(1, 0, 1)$, ...

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

- The question of "heavy hitters" is to estimate the "large" entries in the vector $\mathbf{x}$.
- The total number of elements currently in the system is just $\|\mathbf{x}\| := \sum_{i=1}^{|U|} x_i$. (This is easy.)
- The number of distinct elements is the number of non-zero entries in $\mathbf{x}$.
- We might also want to estimate the norms $\|x\|_2, \|x\|_p$ of the vector $\mathbf{x}$.

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

5 Answering Queries with Inserts and Deletes

How would we implement `query(i)` in this model, now that we allow elements to be added and deleted from the multiset? In fact, what guarantees can we hope from a query operation? Here is one that is fairly useful.

> `query(i)`: output the value of x_i^t with an error of at most $\varepsilon\|\mathbf{x}^t\|_1$, i.e., ε times the size of the multiset at the current time. In symbols: output an estimate
>
> $$\widetilde{x}_i^t \in x_i^t \pm \varepsilon\|\mathbf{x}^t\|_1.$$

In other words, the counts are allowed to be wrong, but not by "too much".

Define the ε-heavy-hitters at time t to be the elements i such that $x_i > \varepsilon\|\mathbf{x}\|_1$. Given an implementation of `query(i)` as above, we are guaranteed that if i is an ε-heavy-hitter then the estimate is strictly positive. (Of course there could be other elements j for which the estimate may be positive but which are not ε-hhs, since we may have false positives.) But this is a step in the right direction.

So how should we implement this `query()` procedure?

[3] 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.

Exercise: solve the ε-heavy-hitters problem using ideas from Section 3 in the model elements are only added to the system, not removed.

Aside: Would sampling work? Suppose we want to find the ε-heavy-hitters: should we just sample some small fraction of the elements and hope to see all the heavy hitters in there? This is not clear how to maintain a small sample size when we allow the stream to contain deletions. Imagine that N copies of a arrive, then they all depart, then we get a stream of length $\sqrt{N}$ containing just b. The only heavy hitter at the end is b. But unless we sample each element with probability more than $1/\sqrt{N}$, we don't expect to see any b's. Here we will see how to get away with storing only poly$(1/\varepsilon)$ elements.

5.1 A Hashing-Based Solution: First Cut

We're going to be using hashing for this approach, simple and effective. We'll worry about what properties we need for our hash functions later, for now assume we have a hash function $h : U \to [M] = \{0, 1, \ldots, M-1\}$ for some suitably large integer M. Maintain an array $A[0 \ldots M-1]$ capable of storing non-negative integers.

```
If add(i))            // i.e., x^t_i = x^{t-1}_i + 1
   A[h(i)]++;
If del(i)             // i.e., x^t_i = x^{t-1}_i - 1
   A[h(i)]--;
```

What about `query(i)`: what is our estimate for x_i^t? It is

$$\widetilde{x}_i^t := A(h(i)).$$

In words, we look at the location $h(i)$ where i gets mapped using the hash function h, and look at the count $A[h(i)]$ stored at that location. What does it contain? It contains the current count x_i for element i for sure. But added to it is the current count for any other element that gets mapped to that location. In math:

$$A(h(j)) = \sum_{j \in U} x_j^t \cdot \mathbf{1}(h(j) = h(i)),$$

where $\mathbf{1}$(some condition) is a function that evaluates to 1 when the condition in the parentheses is true, and 0 if it is false. We can rewrite this as

$$A(h(e)) = x_i^t + \sum_{j \neq i} x_j^t \cdot \mathbf{1}(h(j) = h(i)), \tag{1}$$

or using the definition of the estimate, the error is

$$\widetilde{x}_i^t - x_i^t = \sum_{j \neq i} x_j^t \cdot \mathbf{1}(h(j) = h(i)). \tag{2}$$

It is too much to hope that no other elements $j \neq i$ hashed to location $h(i)$ and the error evaluates to zero. What's the expected error? Now we need to assume something good about the hash functions. Assume that the hash function h is a random draw from a universal family. Recall the definition of universal:

Definition 1 *A family H of hash functions from $U \to [M]$ is universal if for any pair of distinct keys $x_1, x_2 \in U$ with $x_1 \neq x_2$,*

$$\Pr[h(x_1) = h(x_2)] \leq \frac{1}{M}.$$

We gave a construction where each hash function in the family used $(\lg M) \cdot (\lg |U|)$ bits to specify.

Good. So we drew a hash function h from this universal hash family H, and we used it to map elements to locations $\{0, 1, \ldots, M-1\}$. What is its expected error? Taking expectation in (2),

$$\begin{aligned} E\left[\sum_{j \neq i} x_i^t \cdot \mathbf{1}(h(j) = h(i))\right] &= \sum_{j \neq i} x_i^t \cdot E\left[\mathbf{1}(h(j) = h(e))\right] && (3) \\ &= \sum_{j \neq i} x_i^t \cdot \Pr[h(j) = h(i)] \\ &\leq \sum_{j \neq i} x_i^t \cdot (1/M) && (4) \\ &= \frac{\|\mathbf{x}^t\|_1 - x_i^t}{M} \leq \frac{\|\mathbf{x}^t\|_1}{M}. \end{aligned}$$

We used linearity of expectations in equality (3). To get (4) from the previous line, we used the definition of a universal hash family.

That's pretty awesome. (But perhaps not so surprising, once you think about it.) If we just hash the vector down into a smaller vector of size $M = 1/\varepsilon$ then we get expected error $\varepsilon \|\mathbf{x}^t\|_1$, which sounds great. Actually, we'd like to do better—instead of just low *expected error*, we'd like to get low error with high probability. So let's see how to improve things.

5.2 Amplification of the Success Probability

Any ideas how to *amplify* the probability that we are close to the expectation? Very often independent repetition is a great idea.

Let us pick m hash functions $h_1, h_2, \ldots, h_\ell$ *independently* from the universal hash family H.[4] Each $h_i : U \to \{0, 1, \ldots, M-1\}$. We now also have ℓ arrays $A_1, A_2, \ldots, A_\ell$, one for each hash function. The algorithm now just uses the k^{th} hash function to choose a location in the k^{th} array, and increments or decrements the same as before.

```
When update a_t arrives
  For each k from 1..l
    If (a_t == (add, i))
      then A_k[h_k(i)]++;
    else // a_t == (delete, i)
      A_k[h_k(i)]--;
```

And what is our new estimate for the number of copies of element e in our active set? It is

$$\widetilde{x}_i^t := \min_{k=1}^{\ell} A_k(h_k(i)).$$

In other words, each (h_k, A_k) pair gives us an estimate, and we take the least of these. It makes perfect sense — the estimates are all *over*estimates, so taking the least of these is the right thing

[4] If we use the hash function construction given in the previous lecture, this means the $(\lg M) \cdot (\lg |U|)$-bit matrices for each of the ℓ hash functions must be filled with independent random bits.

to do.[5]

But how much better is this estimator? Let's do the math.

What is the chance that one single estimator has error more than $2\|\mathbf{x}^t\|_1/M$? The expected error is at most $\|\mathbf{x}^t\|_1/M$, so by Markov's inequality,

$$\Pr\left[error > 2 \cdot \frac{\|\mathbf{x}^t\|_1}{M}\right] \leq \frac{1}{2}.$$

So the chance that all of the ℓ repetitions have more than $2\|\mathbf{x}^t\|_1/M$ error is

$$\begin{aligned}
&\Pr[\text{each of } \ell \text{ repetitions have error } \geq 2\|\mathbf{x}^t\|_1/M] \\
&= \prod_{k=1}^{\ell} \Pr[k^{th} \text{ repetition had error } \geq 2\|\mathbf{x}^t\|_1/M] \\
&\leq (1/2)^{\ell}.
\end{aligned}$$

The first equality there used the independence of the hash function choices. And so our estimate $\widetilde{x}_i^t$ has error at most $2\|\mathbf{x}^t\|_1/M$ with probability at least $1 - (1/2)^{\ell}$.

5.2.1 Final Bookkeeping

Let's set the parameters now. Set $M = 2/\varepsilon$, so that the error bound $2\|\mathbf{x}^t\|_1/M = \varepsilon\|\mathbf{x}^t\|_1$. Set $\ell = \lg 1/\delta$, then the failure probability is $(1/2)^{\ell} = \delta$, and our query will succeed with probability at least $1 - \delta$. [6] Then for any t and i, the estimate $\widetilde{x}_i^t$ satisfies

$$\Pr\left[|\widetilde{x}_i^i - x_i^t| \leq \varepsilon\|\mathbf{x}^t\|_1\right] \geq 1 - \delta.$$

Just as we wanted. And the total space usage is

$$\ell \cdot M \text{ counters} = O(\log 1/\delta) \cdot O(1/\varepsilon) = O(1/\varepsilon \log 1/\delta) \text{ counters}.$$

Each counter has to store at most $\lg T$-bit numbers after T time steps. [7]

Space for Hash Functions: We need to store the ℓ hash functions as well. How much space does that use? The construction from the previous lecture used $s := (\lg M) \cdot (\lg U)$ bits per hash function. Since $M = 2/\varepsilon$, the total space used for all the ℓ functions is

$$\ell \cdot s = O(\log 1/\delta) \cdot (\lg 1/\varepsilon) \cdot (\lg U) \text{ bits}.$$

In summary, using about $1/\varepsilon \times$ poly-logarithmic factors space, and very simple hashing ideas, we could maintain the counts of elements in a data stream under both arrivals and departures (up to an error of $\varepsilon\|\mathbf{x}^t\|_1$). This algorithm is called the Count-Min sketch, and is due to Cormode and Muthukrishnan.

[5] This is very much like a Bloom filter, which just maintains membership, whereas this maintains counts. But the idea is very similar.

[6] How small should you make δ? Depends on how many queries you want to do. Suppose you want to make a query a million times a day, then you could make $\delta = 1/10^9 \approx 1/2^{30}$ to get a 1-in-1000 chance that even one of your answers has high error. Our space varies linearly as $\lg 1/\delta$, so setting $\delta = 1/10^{18}$ instead of $1/10^9$ doubles the space usage, but drops the error probability by a factor of billion.

[7] So a 32-counter can handle a data stream of length 4 billion. If that is not enough, there are ways to reduce this space usage as well, look online for "approximate counters".

6 Takeaway Messages

Today we saw the data-streaming model, where the input (the "data stream") has size n which is much larger than the amount of working memory (typically $O(\text{poly}\log n)$. Moreover, you are allowed just a single pass over the data, so you have to figure out what information about the stream to store at each time.

1. Sampling may be a reasonable idea for some problems but not for all. In Section 2 we saw an example where pure sampling (e.g., "retain each stream element independently with probability $1/M$") changes the answer. A good alternative is hashing: e.g, "hash the stream elements into a table of size M. Retain elements which hash to position 0." Note this ensures that all copies of a given element is treated the same.

2. Even simple problems (computing the median, finding if there is an element that appears more than 50% of the time, etc) become more challenging when you restrict yourself to this streaming model. In fact, computing things exactly may be impossible with low space.

3. *Approximation* and *Randomness* then become your only hope. You may be able to solve the problem approximately—e.g., for the majority element problem, you allowed *false positives*. For heavy-hitters you allowed randomization and additive error in the counts.

4. Suppose you want to maintain the some statistic.

5. The streaming model often forces you to come up with simple answers: having small space often removes complicated algorithms from consdieration. (Its like you are putting blinkers on, and focusing.) Hence algorithms developed here may be useful even when you have lots of memory.

6. The more things change, the more they remain the same. Data streaming was first considered in the 60s and 70s when data was on tape, and moreover computer memory was small. That model went out of fashion, but then was back in the 2000s with the era of big data.

7. There is a lot of work on data streaming, both in theory and in practice. Check out work by our own David Woodruff (and his monograph on Linear Algebra in Data Streaming), and many other online courses.

7 Optional: Distinct Elements

Our second example today will be to compute the number of distinct elements seen in the data stream. (Imagine there are no deletions, we are in the addition-only model.) So this number is the number of non-zeroes in the vector $\mathbf{x}^t$. Often this is called the zero-norm of $\mathbf{x}^t$, where we define

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

How should we do this?

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 2 (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.[8]

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.

7.1 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. But lBy 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.

7.2 The Algorithm

Assume we have a hash family H with hash functions $h : U \to \{0, 1, \ldots, M-1\}$. 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), ..., 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 = M·s / L_t.
```

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

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 formally reasons that this estimate is any good.

Theorem 3 *Consider some time t. If H is a 2-universal hash family mapping $U \to \{0, 1, \ldots, M-1\}$, and M is large enough, then both the following guarantees hold:*

$$\Pr[D_t > 2\,\|\mathbf{x}^t\|_0] \leq \frac{3}{s}, \text{ and} \tag{5}$$

$$\Pr[D_t < \frac{\|\mathbf{x}^t\|_0}{2}] \leq \frac{3}{s}. \tag{6}$$

We will prove this in the next section. First, some observations. Firstly, setting $s = 12$ means that the estimate D_t lies within $[\frac{\|\mathbf{x}^t\|_0}{2}, 2\|\mathbf{x}^t\|_0]$ with probability at least $1-(3/s+3/s) = 1-(1/4+1/4) = 1/2$. (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.

Finally, observe we now use the stronger assumption that that the hash family is 2-*universal* or *pairwise-independent*. Recall the definition?

Definition 4 (2-Universal Hash Family) *A family H of hash functions from $U \to R$ is 2-universal if for any pair of distinct keys $x_1 \neq x_2$ and any set of values $v_1, v_2 \in R$,*

$$\Pr[h(x_1) = v_1 \wedge h(x_2) = v_2] = \frac{1}{|R|^2}.$$

In other words, if we just look at two keys, the probability that they map to two particular values v_1, v_2 in the range R is the same as what we would get if we were to map these elements completely randomly and independently to locations in the R.

7.3 Proof of Theorem 3

Now for the proof of the theorem. We'll prove bound (6), the other bound (5) 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,

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

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

$$X_i = \begin{cases} 1 & \text{if } h(e_i) \leq 2sM/d \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

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

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

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

$$\Pr[X_i = 1] = \frac{\lfloor sM/2d \rfloor}{M} \geq \frac{s}{2d} - \frac{1}{M}. \tag{8}$$

By linearity of expectations,

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

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,

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

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.

7.3.1 Enter Chebyshev

To do better, the final ingredient is Chebyshev's Inequality, which you recall from the previous lecture. For a random variable Z with mean μ and variance σ^2,

$$\Pr\left[\,|Z - \mu| \geq c\,\sigma\,\right] \leq \frac{1}{c^2}.$$

A convenient way to rewrite Chebyshev's Inequality is

$$\Pr\left[\,|Z - \mu| \geq C\,\mu\,\right] \leq \frac{\sigma^2}{(C\,\mu)^2}. \tag{9}$$

Applying Chebyshev's inequality is useful when the variance of the random variable Z is small. Fortunately, we have some useful facts about variances we can use.

- $\text{Var}(\sum_i Z_i) = \sum_i \text{Var}(Z_i)$ for pairwise-independent random variables Z_i. (Why?)
- And when Z_i is a $\{0,1\}$ random variable, $\text{Var}(Z_i) \leq E[Z_i]$. (Why?)

Applying this to our random variables $X = \sum_i X_i$, we get

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

(The first inequality used that the X_i were pairwise independent, since the hash function was 2-universal.)

Is this variance "low" enough? Let's plug into Chebyshev's inequality (9) and find out.

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

Which is precisely what we want for the bound (5). The proof for the bound (6) 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 $\Pr[X < \frac{E[X]}{(1+\varepsilon)}]$. 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.

7.4 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 construction from previous lectures (and the homework) uses $O((\lg M) + (\lg U))$ 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.