1 Heavy Hitter Guarantees

1.1 The Streaming Model

As with the past few lectures, today we are working under the streaming model. This means that there is some stream of updates \((i_t, v_t)_{t=1}^T\) defining some vector \(x \in \mathbb{R}^n\) such that

\[
x_k = \sum_{t=1}^T \delta(i_t = k) v_t
\]

where \(\delta(i_t = k)\) is the indicator function for the statement \(i_t = k\). The goal in this model is to return some quantity or quantities related to \(x\) using as little memory as possible. We also assume that as \(x\) is updated, its entries never get too large, and can always be represented in \(O(\log n)\) bits.

1.2 \(\ell_1\) and \(\ell_2\) Guarantees

One such quantity relates to the definition of \(\ell_1\)- and \(\ell_2\)-guarantees, as defined in the last lecture:

1. \(\ell_1\)-guarantee:
   
   • Output a set containing all items \(j\) for which \(|x_j| \geq \phi |x|_1\)
   • The set should not contain any \(j\) with \(|x_j| \leq (\phi - \varepsilon)|x|_1\)

2. \(\ell_2\)-guarantee:
   
   • Output a set containing all items \(j\) for which \(|x_j^2| \geq \phi |x|_2^2\)
   • The set should not contain any \(j\) with \(|x_j^2| \leq (\phi - \varepsilon)|x|_2^2\)

1.3 Algorithm Intuition

1.3.1 A Simple Problem

Suppose you are promised that at the end of the stream, \(x_{i^*} = n\) and \(x_i \in \{0, 1\}\) for \(i \in \{1, 2, \ldots, n\}\) with \(i \neq i^*\). How do we find \(i^*\)?

The algorithm to solve this is as follows: Each index \(i \in \{1, \ldots, n\}\) has a bit representation, and we can partition these indices using these bits. That is, for \(j = 1, \ldots, \lfloor \log n \rfloor\), define \(A_j\) as the
set of all indices with jth bit equal to 0. Similarly, define \( B_j \) as the set of all indices with jth bit equal to 1. Then, we can define

\[
a_j = \sum_{i \in A_j} x_i
\]

and

\[
b_j = \sum_{i \in B_j} x_i.
\]

Note that \( A_j \) and \( B_j \) partition all indices, so \( i^* \) will be in either \( A_j \) or \( B_j \); if in \( A_j \), then \( a_j > b_j \); if in \( B_j \), then \( b_j > a_j \). Thus, the jth bit of \( i^* \) is 0 if \( a_j > b_j \), and 1 otherwise, so we can reconstruct \( i^* \) by keeping track of these \( a_j \)s and \( b_j \)s. Note that this algorithm keeps track of \( O(\log n) \) quantities, each with bit-size \( O(\log n) \). Thus, the entire algorithm uses \( O(\log^2 n) \) space.

1.3.2 A Less Simple Problem

Now we make a small modification to the prior question. Suppose you are promised that at the end of the stream, \( x_{i^*} = 100\sqrt{n \log n} \) and \( x_i \in \{0, 1\} \) for \( i \in \{1, 2, \ldots, n\} \) with \( i \neq i^* \). How do we find \( i^* \)?

The prior algorithm no longer works. In particular, for \( n \) large enough, we have \( \frac{n^2}{2} \geq 100\sqrt{n \log n} \). This means that if we try to partition the indices into the sets \( A_j \) and \( B_j \), then \( i^* \in A_j \) does not imply that \( a_j > b_j \); nor does \( i^* \in B_j \) imply that \( b_j > a_j \). Written another way, there is the issue that a large number of 1s added together can be larger than \( 100\sqrt{n \log n} \).

But there is an easy fix to this! By adding a little bit of randomness and error probability, we can ensure that large amounts of 1s are not added together. Multiplying randomly by \( \{1, -1\} \), we can make sure that these 1s cancel out while the large entry remains large in magnitude! That is, we additionally come up with a random sign function \( \sigma : [n] \to \{-1, 1\} \), and instead keep track of the quantities

\[
a_j = \sum_{i \in A_j} \sigma(i)x_i
\]

and

\[
b_j = \sum_{i \in B_j} \sigma(i)x_i
\]

for each \( j = 1, \ldots, \lfloor \log n \rfloor \). So for a fixed \( j \), WLOG assume that \( i^* \in A_j \). Due to the random signs, the probability that a bunch of 1s gang up and overwhelm the magnitude of \( 100\sqrt{n \log n} \) is low; so in turn, we can assume that the \( a_j \) or \( b_j \) of larger magnitude will be the one containing \( i^* \). More formally, we use a Hoeffding bound. The sum

\[
\sum_{i \in A_j \setminus \{i^*\}} \sigma(i)x_i
\]

can be represented as the sum of at most \( |A_j| \leq \frac{n}{2} \) i.i.d. variables with uniform distribution over \( \{-1, 1\} \) (i.e. 1/2 probability equal to -1, 1/2 probability equal to 1). With a Hoeffding bound\(^1\) we

---

\(^1\)The Hoeffding bound requires independent variables. This would mean that we would have to independently store each of the outputs of \( \sigma \), for a prohibitive space cost of \( O(n) \). Instead, we can use Lemma 17 from [this paper](#) setting \( \lambda = \sqrt{\log n} \).
have that
\[
P \left[ \left| \sum_{i \in A \setminus \{i^*\}} \sigma(i)x_i \right| \geq \sqrt{n \log \frac{1}{\delta}} \right] \leq e^{-\Theta(\log(1/\delta))} = \delta.
\]

So by setting \( \delta = \frac{1}{c \log n} \) and using a union bound over all \( j \), we have that the magnitude of \( 100 \sqrt{n \log n} \) is never overwhelmed in any of the \( k \) partitions with constant probability. So, for any \( j \), we can pick the bit value of 0, 1 corresponding to the \( a_j, b_j \) of larger magnitude. In this way we can return the correct bits of \( i^* \).

Note that this is essentially CountSketch (a signed hashing function), except with a deterministic hash function.

1.3.3 The Actual Problem: The \( \ell_2 \)-guarantee

So! The actual problem is just to find a way to achieve the \( \ell_2 \) guarantee. And as you can see from section 1.3.2, the idea is to use CountSketch. Note that a single CountSketch is composed of a sign function \( \sigma : [n] \rightarrow \{-1, 1\} \) and a hash function \( h : [n] \rightarrow [B] \), where \( B \) is the number of buckets to hash into. For ease of writing, we let \( \sigma_i = \sigma(i) \).

First, suppose we just use a single CountSketch with \( B \) buckets. Given update \( (i_t, v_t) \), we simply update the bucket \( h(i_t) \) by \( v_t \). Let \( c_j \) be the value of bucket \( j \) by the end of the stream, so \( c_j = \sum_{i=1}^{n} \delta(h(i) = j)\sigma_ix_i \). Then, for each index \( i \in [n] \), we can estimate \( x_i \) as \( \sigma_ic_{h(i)} \).

**Lemma 1.** \( \sigma_ic_{h(i)} \) is an unbiased estimator for \( x_i \).

**Proof:**

\[
E_{\sigma,h}[\sigma_ic_{h(i)}] = E_{\sigma,h}\left[ \sum_{j=1}^{n} \delta(h(j) = h(i))\sigma_i\sigma_jx_j \right]
\]

\[
= E_{\sigma,h} [\delta(h(i) = h(i))\sigma_i\sigma_ix_i] + E_{\sigma,h} \left[ \sum_{j=1,j \neq i}^{n} \delta(h(j) = h(i))\sigma_i\sigma_jx_j \right]
\]

\[
= E_{\sigma,h} [\delta(h(i) = h(i))\sigma_i\sigma_ix_i] + \sum_{j=1,j \neq i}^{n} E_{h}[\delta(h(j) = h(i))] E_{\sigma} [\sigma_j] x_j
\]

\[
= x_i
\]

The last equality is because \( E_{\sigma} [\sigma_i, \sigma_j] = 0 \) for \( i \neq j \).

Of course, expectation isn’t enough. We also want to bound the variance of this estimator \( \sigma_ic_{h(i)} \). It suffices to bound \( E[\sigma_i^2c_{h(i)}^2] \), which is at least as large as \( \text{Var}[\sigma_ic_{h(i)}] \). We have the following Lemma:

**Lemma 2.** \( \text{Var}[\sigma_ic_{h(i)}] \leq \frac{|x|^2}{B} \).
Proof:

\[
\mathbb{E}_{h,\sigma}[\sigma_i^2c_{h(i)}^2] = \mathbb{E}_{h,\sigma}[c_{h(i)}^2]
\]

\[
= \mathbb{E}_{h,\sigma}\left[\sum_{j=1}^{n} \delta(h(j) = h(i))\sigma_jx_j\right]^2
\]

\[
= \mathbb{E}_{h,\sigma}\left[\sum_{j=1}^{n} \sum_{k=1}^{n} \delta(h(j) = h(i))\delta(h(k) = h(i))\sigma_j\sigma_kx_jx_k\right]
\]

For \( j \neq k \), \( \mathbb{E}[\sigma_j\sigma_k] = 0 \), so we can eliminate those terms. Then the only nonzero terms are those with \( j = k \), and so

\[
\mathbb{E}_{h,\sigma}[\sigma_i^2c_{h(i)}^2] = \sum_{j=1}^{n} \mathbb{E}_h[\delta(h(j) = h(i))x_j^2] = \sum_{j=1}^{n} \frac{1}{B} x_j^2 = \frac{|x_i|^2}{B}
\]

In particular, we have

\[
\text{Var}[\sigma_i c_{h(i)}] \leq \frac{|x_i|^2}{B}.
\]

Now that we have a bound on the variance, we can use Chebychev’s inequality! In particular, with constant probability, the error of the estimate of a single index is \( O\left(\frac{|x_i|^2}{\sqrt{B}}\right) \). Anticipating a union bound across all \( n \) indices, we can also raise the error bound to \( O\left(\sqrt{n}\frac{|x_i|^2}{\sqrt{B}}\right) \) to achieve a failure probability of just \( O\left(\frac{1}{n}\right) \). Yet, this error bound is too large; so we will instead find another way to lower this failure probability.

To do so, we consider running \( O(\log n) \) independent CountSketches, each with \( B \) buckets; then, our estimate of \( x_i \) will just be the median of the \( \sigma_ic_{h(i)} \)s across all buckets. This allows us to maintain our error bound while lowering the failure probability using a bound on the concentration of the binomial distribution. More formally, with probability \( 1 - \frac{1}{\text{poly}(n)} \), the error of our estimate is within \( O\left(\frac{|x_i|^2}{\sqrt{B}}\right) \). In conclusion, all our estimates have error within \( O\left(\frac{|x_i|^2}{\sqrt{B}}\right) \) with probability \( 1 - \frac{1}{\text{poly}(n)} \).

Intuitively, this error bound of \( O\left(\frac{|x_i|^2}{\sqrt{B}}\right) \) is okay because we only care about heavy-hitters; and for a heavy hitter \( x_i \), \( |x_i| \) is comparable to \( |x_i|^2 \). More formally, let’s call \( \hat{x}_i \) our estimate of \( x_i \). Suppose we have access to \( |x_i|^2 \) and we return the indices \( i \) such that \( \hat{x}_i^2 \geq (\phi - \frac{1}{2}\epsilon)|x_i|^2 \). Consider a heavy hitter \( x_i \), so

\[
x_i^2 \geq \phi|x_i|^2.
\]

Then as long as \( O\left(\frac{|x_i|}{\sqrt{B}}\right) |x_i| \leq \frac{\epsilon}{6} |x_i|^2 \), we can be assured that our estimate of \( x_i^2 \) (given by \( \hat{x}_i^2 \)) will have error at most \( \frac{\epsilon}{3}|x_i|^2 \); and so we know that

\[
\hat{x}_i^2 \geq \left(\phi - \frac{\epsilon}{2}\right) |x_i|^2,
\]

in which case we correctly return \( i \).  

\(^2\)Note that the algorithm does not have access to \( |x_i|^2 \). This can be solved by sketching the 2-norm using CountSketch or random gaussian matrices, which does not introduce significant error nor increase the asymptotic space cost of the algorithm.
Similarly, if \( i \) is something that we do not want to return, then
\[
x_i^2 \leq (\phi - \epsilon)|x|^2.
\]
Once again if \( O \left( \frac{|x|}{\sqrt{B}} \right) |x_i| \leq \frac{\epsilon}{6}|x|^2 \), then our estimate of \( x_i^2 \) (given by \( \hat{x}_i^2 \)) will have error at most \( \frac{\epsilon}{3}|x|^2 \), so we have that
\[
\hat{x}_i^2 < \left( \phi - \frac{\epsilon}{2} \right)|x|^2,
\]
in which case we correctly do not return \( i \).

As a small note, the algorithm does not have access to \( |x|^2 \). Instead, we can also use CountSketch or a matrix of random Gaussian to estimate \( |x|^2 \); the additional error should not present too much of a problem.

So in conclusion, we want \( B \) large enough such that \( O \left( \frac{|x|}{\sqrt{B}} \right) |x| \leq \frac{\epsilon}{6}|x|^2 \). It suffices to choose \( B \) such that \( O \left( \frac{|x|}{\sqrt{B}} \right) |x| \leq \frac{\epsilon}{6}|x|^2 \), which means that we want \( B = O(\epsilon^{-2}) \).

Space-wise, we have \( O(\log n) \) runs of CountSketch, each with \( B \) buckets, each with entries of size at most \( O(\log n) \). Thus the entire algorithm uses \( O \left( \frac{\log^2 n}{\epsilon^2} \right) \) space.

**Theorem 1.** CountSketch approximates every \( x_i \) simultaneously up to \( O \left( \frac{|x - B/4|}{\sqrt{B}} \right) \) additive error, where \( x_{-B/4} \) is \( x \) after zero-ing out its top \( B/4 \) coordinates in magnitude.

**Proof:** For any \( i \), with probability at least \( \frac{3}{4} \), none of the largest \( B/4 \) indices are hashed to the same bucket as \( i \) (unless \( i \) is among the largest \( B/4 \) indices). The additional \( \frac{1}{4} \) failure probability for each entry can be washed away in the subsequent median probability bounds. Thus the resulting additive error of each estimate is proportional to \( O \left( \frac{|x - B/4|}{\sqrt{B}} \right) \) instead of \( O \left( \frac{|x|}{\sqrt{B}} \right) \), with only a constant blowup in failure probability.

Note that theorem 1 implies that if \( x \) is \( B/4 \)-sparse, then the error bound is 0, which means that we can reconstruct \( x \) perfectly with high probability!

### 1.4 A Different Problem: The \( \ell_1 \)-guarantee

Why do we care about the \( \ell_1 \)-guarantee? As mentioned in earlier lectures, the \( \ell_1 \)-guarantee is implied by the \( \ell_2 \)-guarantee; so what use is there in solving the former if we can already solve the latter? Well, the \( \ell_1 \)-guarantee is an easier problem, and can be solved with a deterministic algorithm (which not not true for the \( \ell_2 \)-guarantee).

To solve the \( \ell_1 \)-guarantee, we simply need an \( \epsilon \)-incoherent matrix; that is, an \( s \times n \) matrix \( S \) such that:

1. For every column \( S_i \) of \( S \), \( |S_i|^2 = 1 \).
2. For all pairs of columns \( S_i \) and \( S_j \) (\( i \neq j \)), \( |\langle S_i, S_j \rangle| \leq \epsilon \).

As a side note, we also want \( S \) to be storable with \( O(\log n) \) bits of space.
With such an $\epsilon$-incoherent matrix $S$, we can easily find an algorithm with the $\ell_1$-guarantee. First, we compute $S \cdot x$ in a stream using $O(s \log n)$ bits of space. Then, we can estimate $x_i$ with $\hat{x}_i = S_i^T S x$. Why does this work? Well, note that

$$\hat{x}_i = S_i^T S x = \sum_{j=1}^{s} \langle S_i, S_j \rangle x_j = x_i + \sum_{j=1, j \neq i}^{s} \langle S_i, S_j \rangle x_j = x_i \pm \epsilon |x|_1$$

That is, our estimate of $x_i$ has error at most $\epsilon |x|_1$. With such an error bound we can easily construct an algorithm that satisfies the $\ell_1$ guarantee.

In the next part of the lecture, we will construct an $\epsilon$-coherent matrix $S$ that only requires $O(\log n)$ bits of space to store.