CS369E: Communication Complexity (for Algorithm Designers) Lecture #1: Data Streams: Algorithms and Lower Bounds*

Tim Roughgarden[†]

January 8, 2015

1 Preamble

This class is mostly about impossibility results — lower bounds on what can be accomplished by algorithms. However, our perspective will be unapologetically that of an algorithm designer.[1] We'll learn lower bound technology on a "need-to-know" basis, guided by fundamental algorithmic problems that we care about (perhaps theoretically, perhaps practically). That said, we will wind up learning quite a bit of complexity theory — specifically, communication complexity — along the way. We hope this viewpoint makes this course and these notes complementary to the numerous excellent courses, books [10, 11], and surveys (e.g. [12, 13, 6, 15]) on communication complexity.[2] The theme of communication complexity lower bounds also provides a convenient excuse to take a guided tour of numerous models, problems, and algorithms that are central to modern research in the theory of algorithms but missing from many algorithms textbooks: streaming algorithms, space-time trade-offs in data structures, compressive sensing, sublinear algorithms, extended formulations for linear programs, and more.

Why should an algorithm designer care about lower bounds? The best mathematical researchers are able to work on an open problem simultaneously from both sides. Even if you have a strong prior belief about whether a given mathematical statement is true or false, failing to prove one direction usefully informs your efforts to prove the other. (And for most

*©2015, Tim Roughgarden.

[†]Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: `tim@cs.stanford.edu`.

[1]Already in this lecture, over half our discussion will be about algorithms and upper bounds!

[2]See Pătraşcu [14] for a series of four blog posts on data structures that share some spirit with our approach.

us, the prior belief is wrong surprisingly often!) In algorithm design, working on both sides means striving simultaneously for better algorithms and for better lower bounds. For example, a good undergraduate algorithms course teaches you both how to design polynomial-time algorithms and how to prove that a problem is NP-complete — since when you encounter a new computational problem in your research or workplace, both are distinct possibilities. There are many other algorithmic problems where the fundamental difficulty is not the amount of time required, but rather concerns communication or information transmission. The goal of this course is to equip you with the basic tools of communication complexity — its canonical hard problems, the canonical reductions from computation in various models to the design of low-communication protocols, and a little bit about its lower bound techniques — in the service of becoming a better algorithm designer.

This lecture and the next study the *data stream* model of computation. There are some nice upper bounds in this model (see Sections 4 and 5), and the model also naturally motivates a severe but useful restriction of the general communication complexity setup (Section 7). We'll cover many computational models in the course, so whenever you get sick of one, don't worry, a new one is coming up around the corner.

2 The Data Stream Model

The data stream model is motivated by applications in which the input it best thought of as a firehose — packets arriving to a network switch at a torrential rate, or data being generated by a telescope at a rate of one exobyte per day. In these applications, there's no hope of storing all the data, but we'd still like to remember useful summary statistics about what we've seen.

Alternatively, for example in database applications, it could be that data is not thrown away but resides on a big, slow disk. Rather than incurring random access costs to the data, one would like to process it sequentially once (or a few times), perhaps overnight, remembering the salient features of the data in a limited main memory for real-time use. The daily transactions of Amazon or Walmart, for example, could fall into this category.

Formally, suppose data elements belong to a known universe $U = \{1, 2, \ldots, n\}$. The input is a stream $x_1, x_2, \ldots, x_m \in U$ of elements that arrive one-by-one. Our algorithms will not assume advance knowledge of m, while our lower bounds will hold even if m is known a priori. With space $\approx m \log_2 n$, it is possible to store all of the data. The central question in data stream algorithms is: what is possible, and what is impossible, using a one-pass algorithm and much less than $m \log n$ space? Ideally, the space usage should be sublinear or even logarithmic in n and m. We're not going to worry about the computation time used by the algorithm (though our positive results in this model have low computational complexity, anyway).

Many of you will be familiar with a streaming or one-pass algorithm from the following common interview question. Suppose an array A, with length m, is promised to have a *majority element* — an element that occurs strictly more than $m/2$ times. A simple one-pass algorithm, which maintains only the current candidate majority element and a counter for it

— so $O(\log n + \log m)$ bits — solves this problem. (If you haven't seen this algorithm before, see the Exercises.) This can be viewed as an exemplary small-space streaming algorithm.[3]

3 Frequency Moments

Next we introduce *the* canonical problems in the field of data stream algorithms: computing the *frequency moments* of a stream. These were studied in the paper that kickstarted the field [1], and the data stream algorithms community has been obsessed with them ever since.

Fix a data stream $x_1, x_2, \ldots, x_m \in U$. For an element $j \in U$, let $f_j \in \{0, 1, 2, \ldots, m\}$ denote the number of times that j occurs in the stream. For a non-negative integer k, the *kth frequency moment* of the stream is

$$F_k := \sum_{j \in U} f_j^k. \tag{1}$$

Note that the bigger k is, the more the sum in (1) is dominated by the largest frequencies. It is therefore natural to define

$$F_\infty = \max_{j \in U} f_j$$

as the largest frequency of any element of U.

Let's get some sense of these frequency moments. F_1 is boring — since each data stream element contributes to exactly one frequency f_j, $F_1 = \sum_{j \in U} f_j$ is simply the stream length m. F_0 is the number of distinct elements in the stream (we're interpreting $0^0 = 0$) — it's easy to imagine wanting to compute this quantity, for example a network switch might want to know how many different TCP flows are currently going through it. F_∞ is the largest frequency, and again it's easy to imagine wanting to know this — for example to detect a denial-of-service attack at a network switch, or identify the most popular product on Amazon yesterday. Note that computing F_∞ is related to the aforementioned problem of detecting a majority element. Finally, $F_2 = \sum_{j \in U} f_2^2$ is sometimes called the "skew" of the data — it is a measure of how non-uniform the data stream is. In a database context, it arises naturally as the size of a "self-join" — the table you get when you join a relation with itself on a particular attribute, with the f_j's being the frequencies of various values of this attribute. Having estimates of self-join (and more generally join) sizes at the ready is useful for query optimization, for example. We'll talk about F_2 extensively in the next section.[4]

It is trivial to compute all of the frequency moments in $O(m \log n)$ space, just by storing the x_i's, or in $O(n \log m)$, space, just by computing and storing the f_j's (a $\log m$-bit counter for each of the n universe elements). Similarly, F_1 is trivial to compute in $O(\log m)$ space (via a counter), and F_0 in $O(n)$ space (via a bit vector). Can we do better?

[3]Interestingly, the promise that a majority element exists is crucial. A consequence of the next lecture is that there is no small-space streaming algorithm to check whether or not a majority element exists!

[4]The problem of computing F_2 and the solution we give for it are also quite well connected to other important concepts, such as compressive sensing and dimensionality reduction.

Intuitively, it might appear difficult to improve over the trivial solution. For F_0, for example, it seems like you have to know which elements you've already seen (to avoid double-counting them), and there's an exponential (in n) number of different possibilities for what you might have seen in the past. As we'll see, this is good intuition for deterministic algorithms, and for (possibly randomized) exact algorithms. Thus, the following positive result is arguably surprising, and very cool.[5]

Theorem 3.1 ([1]) *Both F_0 and F_2 can be approximated, to within a $(1 \pm \epsilon)$ factor with probability at least $1 - \delta$, in space*

$$O\left((\epsilon^{-2}(\log n + \log m) \log \tfrac{1}{\delta}\right). \tag{2}$$

Theorem 3.1 refers to two different algorithms, one for F_0 and one for F_2. We cover the latter in detail below. Section 5 describes the high-order bit of the F_0 algorithm, which is a modification of the earlier algorithm of [8], with the details in the exercises. Both algorithms are randomized, and are approximately correct (to within $(1 \pm \epsilon)$) most of the time (except with probability δ). Also, the $\log m$ factor in (2) is not needed for the F_0 algorithm, as you might expect. Some further optimization are possible; see Section 4.3.

The first reason to talk about Theorem 3.1 is that it's a great result in the field of algorithms — if you only remember one streaming algorithm, the one below might as well be the one.[6] You should never tire of seeing clever algorithms that radically outperform the "obvious solution" to a well-motivated problem. And Theorem 3.1 should serve as inspiration to any algorithm designer — even when at first blush there is no non-trivial algorithm for problem in sight, the right clever insight can unlock a good solution.

On the other hand, there unfortunately are some important problems out there with no non-trivial solutions. And it's important for the algorithm designer to know which ones they are — the less effort wasted on trying to find something that doesn't exist, the more energy is available for formulating the motivating problem in a more tractable way, weakening the desired guarantee, restricting the problem instances, and otherwise finding new ways to make algorithmic progress. A second interpretation of Theorem 3.1 is that it illuminates why such lower bounds can be so hard to prove. A lower bound is responsible for showing that every algorithm, even fiendishly clever ones like those employed for Theorem 3.1, cannot make significant inroads on the problem.

4 Estimating F_2: The Key Ideas

In this section we give a nearly complete proof of Theorem 3.1 for the case of $F_2 = \sum_{j \in U} f_2^2$ estimation (a few details are left to the Exercises).

[5]The Alon-Matias-Szegedy paper [1] ignited the field of streaming algorithms as a hot area, and for this reason won the 2005 Gödel Prize (a "test of time"-type award in theoretical computer science). The paper includes a number of other upper and lower bounds as well, some of which we'll cover shortly.

[6]Either algorithm, for estimating F_0 or for F_2, could serve this purpose. We present the F_2 estimation algorithm in detail, because the analysis is slightly slicker and more canonical.

4.1 The Basic Estimator

The high-level idea is very natural, especially once you start thinking about randomized algorithms.

1. Define a randomized unbiased estimator of F_2, which can be computed in one pass. Small space seems to force a streaming algorithm to lose information, but maybe it's possible to produce a result that's correct "on average."

2. Aggregate many independent copies of the estimator, computed in parallel, to get an estimate that is very accurate with high probability.

This is very hand-wavy, of course — does it have any hope of working? It's hard to answer that question without actually doing some proper computations, so let's proceed to the estimator devised in [1].

The Basic Estimator:[7]

1. Let $h : U \to \{\pm 1\}$ be a function that associates each universe element with a random sign. On a first reading, to focus on the main ideas, you should assume that h is a totally random function. Later we'll see that relatively lightweight hash functions are good enough (Section 4.2), which enables a small-space implementation of the basic ideas.

2. Initialize $Z = 0$.

3. Every time a data stream element $j \in U$ appears, add $h(j)$ to Z. That is, increment Z if $h(j) = +1$ and decrement Z if $h(j) = -1$.[8]

4. Return the estimate $X = Z^2$.

Remark 4.1 A crucial point: since the function h is fixed once and for all before the data stream arrives, an element $j \in U$ is treated consistently every time it shows up. That is, Z is either incremented every time j shows up or is decremented every time j shows up. In the end, element j contributes $h(j)f_j$ to the final value of Z.

First we need to prove that the basic estimator is indeed unbiased.

Lemma 4.2 *For every data stream,*

$$\mathbf{E}_h[X] = F_2.$$

[7]This is sometimes called the "tug-of-war" estimator.

[8]This is the "tug of war," between elements j with $h(j) = +1$ and those with $h(j) = -1$.

Proof: We have

$$\begin{aligned}
\mathbf{E}[X] &= \mathbf{E}\big[Z^2\big] \\
&= \mathbf{E}\left[\left(\sum_{j\in U} h(j)f_j\right)^2\right] \\
&= \mathbf{E}\left[\sum_{j\in U} \underbrace{h(j)^2}_{=1} f_j^2 + 2\sum_{j<\ell} h(j)h(\ell)f_jf_\ell\right] \\
&= \underbrace{\sum_{j\in U} f_j^2}_{=F_2} + 2\sum_{j<\ell} f_jf_\ell \underbrace{\mathbf{E}_h[h(j)h(\ell)]}_{=0} \qquad (3) \\
&= F_2, \qquad (4)
\end{aligned}$$

where in (3) we use linearity of expectation and the fact that $h(j) \in \{\pm 1\}$ for every j, and in (4) we use the fact that, for every distinct j, ℓ, all four sign patterns for $(h(j), h(\ell))$ are equally likely. ■

Note the reason for both incrementing and decrementing in the running sum Z — it ensures that the "cross terms" $h(j)h(\ell)f_jf_\ell$ in our basic estimator $X = Z^2$ cancel out in expectation. Also note, for future reference, that the only time we used the assumption that h is a totally random function was in (4), and we only used the property that all four sign patterns for $(h(j), h(\ell))$ are equally likely — that h is "pairwise independent."

Lemma 4.2 is not enough for our goal, since the basic estimator X is not guaranteed to be close to its expectation with high probability. A natural idea is to take the average of many independent copies of the basic estimator. That is, we'll use t independent functions $h_1, h_2, \ldots, h_t : U \to \{\pm 1\}$ to define estimates $X_1, \ldots, X_t$. On the arrival of a new data stream element, we update all t of the counters $Z_1, \ldots, Z_t$ appropriately, with some getting incremented and others decremented. Our final estimate will be

$$Y = \frac{1}{t}\sum_{i=1}^{t} X_i.$$

Since the X_i's are unbiased estimators, so is Y (i.e., $\mathbf{E}_{h_1,\ldots,h_t}[Y] = F_2$). To see how quickly the variance decreases with the number t of copies, note that

$$\mathbf{Var}[Y] = \mathbf{Var}\left[\frac{1}{t}\sum_{i=1}^{t} X_i\right] = \frac{1}{t^2}\sum_{i=1}^{t}\mathbf{Var}[X_i] = \frac{\mathbf{Var}[X]}{t},$$

where X denotes a single copy of the basic estimator. That is, averaging reduces the variance by a factor equal to the number of copies. Unsurprisingly, the number of copies t (and in the end, the space) that we need to get the performance guarantee that we want is governed by the variance of the basic estimator. So there's really no choice but to roll up our sleeves and compute it.

Lemma 4.3 *For every data stream,*

$$\mathbf{Var}_h[X] \leq 2F_2^2.$$

Lemma 4.3 states the standard deviation of the basic estimator is in the same ballpark as its expectation. That might sound ominous, but it's actually great news — a constant (depending on ϵ and δ only) number of copies is good enough for our purposes. Before proving Lemma 4.3, let's see why.

Corollary 4.4 *For every data stream, with* $t = \frac{2}{\epsilon^2\delta}$*,*

$$\mathbf{Pr}_{h_1,\ldots,h_t}[Y \in (1 \pm \epsilon) \cdot F_2] \geq 1 - \delta.$$

Proof: Recall that *Chebyshev's inequality* is the inequality you want when bounding the deviation of a random variable from its mean parameterized by the number of standard deviations. Formally, it states that for every random variable Y with finite first and second moments, and every $c > 0$,

$$\mathbf{Pr}[|Y - \mathbf{E}[Y]\,| > c] \leq \frac{\mathbf{Var}[Y]}{c^2}. \tag{5}$$

Note that (5) is non-trivial (i.e., probability less than 1) once c exceeds Y's standard deviation, and the probability goes down quadratically with the number of standard deviations. It's a simple inequality to prove; see the separate notes on tail inequalities for details.

We are interested in the case where Y is the average of t basic estimators, with variance as in Lemma 4.3. Since we want to guarantee a $(1 \pm \epsilon)$-approximation, the deviation c of interest to us is ϵF_2. We also want the right-hand side of (5) to be equal to δ. Using Lemma 4.3 and solving, we get $t = 2/\epsilon^2\delta$.[9] ■

We now stop procrastinating and prove Lemma 4.3.

Proof of Lemma 4.3: Recall that

$$\mathbf{Var}[X] = \mathbf{E}\left[X^2\right] - \Bigg(\underbrace{\mathbf{E}[X]}_{=F_2 \text{ by Lemma 4.2}}\Bigg)^2. \tag{6}$$

Zooming in on the $\mathbf{E}[X^2]$ term, recall that X is already defined as the square of the running sum Z, so X^2 is Z^4. Thus,

$$\mathbf{E}\left[X^2\right] = \mathbf{E}\left[\left(\sum_{j \in U} h(j) f_j\right)^4\right]. \tag{7}$$

Expanding the right-hand side of (7) yields $|U|^4$ terms, each of the form $h(j_1)h(j_2)h(j_3)h(j_4)f_{j_1}f_{j_2}f_{j_3}f_{j_4}$. (Remember: the h-values are random, the f-values are

[9]The dependence on $\frac{1}{\delta}$ can be decreased to logarithmic; see Section 4.3.

fixed.) This might seem unwieldy. But, just as in the computation (4) in the proof of Lemma 4.2, most of these are zero in expectation. For example, suppose j_1, j_2, j_3, j_4 are distinct. Condition on the h-values of the first three. Since $h(j_4)$ is equally likely to be +1 or -1, the conditional expected value (averaged over the two cases) of the corresponding term is 0. Since this holds for all possible values of $h(j_1), h(j_2), h(j_3)$, the unconditional expectation of this term is also 0. This same argument applies to any term in which some element $j \in U$ appears an odd number of times. Thus, when the dust settles, we have

$$\begin{aligned}\mathbf{E}\left[X^2\right] &= \mathbf{E}\left[\sum_{j\in U} \underbrace{h(j)^4}_{=1} f_j^4 + 6\sum_{j<\ell} \underbrace{h(j)^2}_{=1}\underbrace{h(\ell)^2}_{=1} f_j^2 f_\ell^2\right] \\ &= \sum_{j\in U} f_j^4 + 6\sum_{j<\ell} f_j^2 f_\ell^2,\end{aligned} \tag{8}$$

where the "6" appears because a given $h(j)^2h(\ell)^2f_j^2f_\ell^2$ term with $j < \ell$ arises in $\binom{4}{2} = 6$ different ways.

Expanding terms, we see that

$$F_2^2 = \sum_{j\in U} f_j^4 + 2\sum_{j<\ell} f_j^2 f_\ell^2$$

and hence

$$\mathbf{E}\left[X^2\right] \le 3F_2^2.$$

Recalling (6) proves that $\mathbf{Var}[X] \le 2F_2^2$, as claimed. ■

Looking back over the proof of Lemma 4.3, we again see that we only used the fact that h is random in a limited way. In (8) we used that, for every set of four distinct universe elements, their 16 possible sign patterns (under h) were equally likely. (This implies the required property that, if j appears in a term an odd number of times, then even after conditioning on the h-values of all other universe elements in the term, $h(j)$ is equally likely to be +1 or -1.) That is, we only used the "4-wise independence" of the function h.

4.2 Small-Space Implementation via 4-Wise Independent Hash Functions

Let's make sure we're clear on the final algorithm.

1. Choose functions $h_1, \ldots, h_t : U \to \{\pm 1\}$, where $t = \frac{2}{\epsilon^2\delta}$.

2. Initialize $Z_i = 0$ for $i = 1, 2, \ldots, t$.

3. When a new data stream element $j \in U$ arrives, add $h_i(j)$ to Z_i for every $i = 1, 2, \ldots, t$.

4. Return the average of the Z_i^2's.

Last section proved that, if the h_i's are chosen uniformly at random from all functions, then the output of this algorithm lies in $(1 \pm \epsilon)F_2$ with probability at least $1 - \delta$.

How much space is required to implement this algorithm? There's clearly a factor of $\frac{2}{\epsilon^2 \delta}$, since we're effectively running this many streaming algorithms in parallel, and each needs its own scratch space. How much space does each of these need? To maintain a counter Z_i, which always lies between $-m$ and m, we only need $O(\log m)$ bits. But it's easy to forget that we have to also store the function h_i. Recall from Remark 4.1 the reason: we need to treat an element $j \in U$ consistently every time it shows up in the data stream. Thus, once we choose a sign $h_i(j)$ for j we need to remember it forevermore. Implemented naively, with h_i a totally random function, we would need to remember one bit for each of the possibly $\Omega(n)$ elements that we've seen in the past, which is a dealbreaker.

Fortunately, as we noted after the proofs of Lemmas 4.2 and 4.3, our entire analysis has relied only on 4-wise independence — that when we look at an arbitrary 4-tuple of universe elements, the projection of h on their 16 possible sign patterns is uniform. (Exercise: go back through this section in detail and double-check this claim.) And happily, there are small families of simple hash functions that possess this property.

Fact 4.5 *For every universe U with $n = |U|$, there is a family $\mathcal{H}$ of 4-wise independent hash functions (from U to $\{\pm 1\}$) with size polynomial in n.*

Fact 4.5 and our previous observations imply that, to enjoy an approximation of $(1 \pm \epsilon)$ with probability at least $1 - \delta$, our streaming algorithm can get away with choosing the functions $h_1, \ldots, h_t$ uniformly and independently from $\mathcal{H}$.

If you've never seen a construction of a k-wise independent family of hash functions with $k > 2$, check out the Exercises for details. The main message is to realize that you shouldn't be scared of them — heavy machinery is not required. For example, it suffices to map the elements of U injectively into a suitable finite field (of size roughly $|U|$), and then let $\mathcal{H}$ be the set of all cubic polynomials (with all operations occurring in this field). The final output is then +1 if the polynomial's output (viewed as an integer) is even, and -1 otherwise. Such a hash function is easily specified with $O(\log n)$ bits (just list its four coefficients), and can also be evaluated in $O(\log n)$ space (which is not hard, but we won't say more about it here).

Putting it all together, we get a space bound of

$$O\left(\underbrace{\frac{1}{\epsilon^2 \delta}}_{\#\text{ of copies}} \cdot \left(\underbrace{\log m}_{\text{counter}} + \underbrace{\log n}_{\text{hash function}}\right)\right). \tag{9}$$

4.3 Further Optimizations

The bound in (9) is worse than that claimed in Theorem 3.1, with a dependence on $\frac{1}{\delta}$ instead of $\log \frac{1}{\delta}$. A simple trick yields the better bound. In Section 4.1, we averaged t copies of the basic estimator to accomplish two conceptually different things: to improve the approximation ratio to $(1 \pm \epsilon)$, for which we suffered an $\frac{1}{\epsilon^2}$ factor, and to improve the success probability to $1-\delta$, for which we suffered an additional $\frac{1}{\delta}$. It is more efficient to implement these improvements one at a time, rather than in one shot. The smarter implementation first uses $\approx \frac{1}{\epsilon^2}$ copies to obtain an approximation of $(1 \pm \epsilon)$ with probably at least $\frac{2}{3}$ (say). To boost the success probability from $\frac{2}{3}$ to $1-\delta$, it is enough to run $\approx \log \frac{1}{\delta}$ different copies of this solution, and then take the *median* of their $\approx \log \frac{1}{\delta}$ different estimates. Since we expect at least two-thirds of these estimates to lie in the interval $(1 \pm \epsilon)F_2$, it is very likely that the median of them lies in this interval. The details are easily made precise using a Chernoff bound argument; see the Exercises for details.

Second, believe it or not, the $\log m$ term in Theorem 3.1 can be improved to $\log \log m$. The reason is that we don't need to count the Z_i's exactly, only approximately and with high probability. This relaxed counting problem can be solved using *Morris's algorithm*, which can be implemented as a streaming algorithm that uses $O(\epsilon^{-2} \log \log m \log \frac{1}{\delta})$ space. See the Exercises for further details.

5 Estimating F_0: The High-Order Bit

Recall that F_0 denotes the number of distinct elements present in a data stream. The high-level idea of the F_0 estimator is the same as for the F_2 estimator above. The steps are to define a basic estimator that is essentially unbiased, and then reduce the variance by taking averages and medians. (Really making this work takes an additional idea; see [3] and the Exercises.)

The basic estimator for F_0 — originally from [8] and developed further in [1] and [3] — is as simple as but quite different from that used to estimate F_2. The first step is to choose a random permutation h of U.[10] Then, just remember (using $O(\log n)$ space) the minimum value of $h(x)$ that ever shows up in the data stream.

Why use the minimum? One intuition comes from the suggestive match between the idempotence of F_0 and of the minimum — adding duplicate copies of an element to the input has no effect on the answer.

Given the minimum $h(x)$-value in the data stream, how do we extract from it an estimate of F_0, the number of distinct elements? For intuition, think about the uniform distribution on $[0,1]$ (Figure 1). Obviously, the expected value of one draw from the distribution is $\frac{1}{2}$. For two i.i.d. draws, simple calculations show that the expected minimum and maximum are $\frac{1}{3}$ and $\frac{2}{3}$, respectively. More generally, the expected order statistics of k i.i.d. draws split the interval into $k+1$ segments of equal length. In particular, the expected minimum is $\frac{1}{k+1}$. In other words, if you are told that some number of i.i.d. draws were taken from the uniform distribution on $[0,1]$ and the smallest draw was c, you might guess that there were roughly $1/c$ draws.

Translating this idea to our basic F_0 estimator, if there are k distinct elements in a data stream $x_1, \ldots, x_m$, then there are k different (random) hash values $h(x_i)$, and we expect the smallest of these to be roughly $|U|/k$. This leads to the basic estimator $X =$

[10]Or rather, a simple hash function with the salient properties of a random permutation.

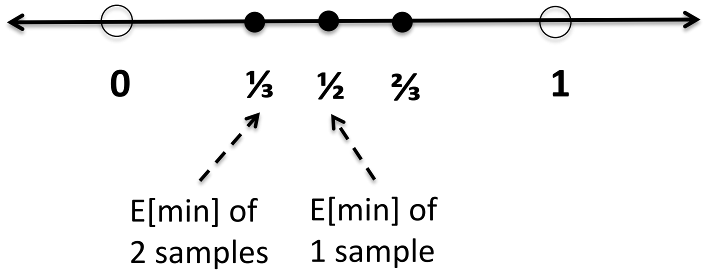

Figure 1: The expected order statistics of i.i.d. samples from the uniform distribution on the unit interval are spaced out evenly over the interval.

$|U|/(\min_{i=1}^{m} h(x_i))$. Using averaging and an extra idea to reduce the variance, and medians to boost the success probability, this leads to the bound claimed in Theorem 3.1 (without the $\log m$ term). The details are outlined in the exercises.

Remark 5.1 You'd be right to ask if this high-level approach to probabilistic and approximate estimation applies to all of the frequency moments, not just F_0 and F_2. The approach can indeed be used to estimate F_k for all k. However, the variance of the basic estimators will be different for different frequency moments. For example, as k grows, the statistic F_k becomes quite sensitive to small changes in the input, resulting in probabilistic estimators with large variance, necessitating a large number of independent copies to obtain a good approximation. More generally, no frequency moment F_k with $k \notin \{0, 1, 2\}$ can be computed using only a logarithmic amount of space (more details to come).

6 Can We Do Better?

Theorem 3.1 is a fantastic result. But a good algorithm designer is never satisfied, and always wants more. So what are the weaknesses of the upper bounds that we've proved so far?

1. We only have interesting positive results for F_0 and F_2 (and maybe F_1, if you want to count that). What about for $k > 2$ and $k = \infty$?

2. Our F_0 and F_2 algorithms only approximate the corresponding frequency moment. Can we compute it exactly, possibly using a randomized algorithm?

3. Our F_0 and F_2 algorithms are randomized, and with probability δ fail to provide a good approximation. (Also, they are Monte Carlo algorithms, in that we can't tell when they fail.) Can we compute F_0 or F_2 deterministically, at least approximately?

4. Our F_0 and F_2 algorithms use $\Omega(\log n)$ space. Can we reduce the dependency of the

space on the universe size?[11]

5. Our F_0 and F_2 algorithms use $\Omega(\epsilon^{-2})$ space. Can the dependence on ϵ^{-1} be improved? The ϵ^{-2} dependence can be painful in practice, where you might want to take $\epsilon = .01$, resulting in an extra factor of 10,000 in the space bound. An improvement to $\approx \epsilon^{-1}$, for example, would be really nice.

Unfortunately, we can't do better — the rest of this lecture and the next (and the exercises) explain why *all* of these compromises are necessary for positive results. This is kind of amazing, and it's also pretty amazing that we can prove it without overly heavy machinery. Try to think of other basic computational problems where, in a couple hours of lecture and with minimal background, you can explain complete proofs of both a non-trivial upper bound and an unconditional (independent of P vs. NP, etc.) matching lower bound.[12]

7 One-Way Communication Complexity

We next describe a simple and clean formalism that is extremely useful for proving lower bounds on the space required by streaming algorithms to perform various tasks. The model will be a quite restricted form of the general communication model that we study later — and this is good for us, because the restriction makes it easier to prove lower bounds. Happily, even lower bounds for this restricted model typically translate to lower bounds for streaming algorithms.

In general, communication complexity is a sweet spot. It is a general enough concept to capture the essential hardness lurking in many different models of computation, as we'll see throughout the course. At the same time, it is possible to prove numerous different lower bounds in the model — some of these require a lot of work, but many of the most important ones are easier that you might have guessed. These lower bounds are "unconditional" — they are simply true, and don't depend on any unproven (if widely believed) conjectures like $P \neq NP$. Finally, because the model is so clean and free of distractions, it naturally guides one toward the development of the "right" mathematical techniques needed for proving new lower bounds.

In (two-party) communication complexity, there are two parties, Alice and Bob. Alice has an input $\mathbf{x} \in \{0,1\}^a$, Bob an input $\mathbf{y} \in \{0,1\}^b$. Neither one has any idea what the other's input is. Alice and Bob want to cooperate to compute a Boolean function (i.e., a predicate) $f : \{0,1\}^a \times \{0,1\}^b \to \{0,1\}$ that is defined on their joint input. We'll discuss several examples of such functions shortly.

For this lecture and the next, we can get away with restricting attention to *one-way communication protocols*. All that is allowed here is the following:

[11]This might seem like a long shot, but you never know. Recall our comment about reducing the space dependency on m from $O(\log m)$ to $O(\log\log m)$ via probabilistic approximate counters.

[12]OK, comparison-based sorting, sure. And we'll see a couple others later in this course. But I don't know of that many examples!

1. Alice sends Bob a message $\mathbf{z}$, which is a function of her input $\mathbf{x}$ only.
2. Bob declares the output $f(\mathbf{x}, \mathbf{y})$, as a function of Alice's message $\mathbf{z}$ and his input $\mathbf{y}$ only.

Since we're interested in both deterministic and randomized algorithms, we'll discuss both deterministic and randomized 1-way communication protocols.

The *one-way communication complexity* of a Boolean function f is the minimum worst-case number of bits used by any 1-way protocol that correctly decides the function. (Or for randomized protocols, that correctly decides it with probability at least 2/3.) That is, it is

$$\min_{\mathcal{P}} \max_{\mathbf{x},\mathbf{y}} \{\text{length (in bits) of Alice's message } \mathbf{z} \text{ when Alice's input is } \mathbf{x}\},$$

where the minimum ranges over all correct protocols.

Note that the one-way communication complexity of a function f is always at most a, since Alice can just send her entire a-bit input $\mathbf{x}$ to Bob, who can then certainly correctly compute f. The question is to understand when one can do better. This will depend on the specific function f. For example, if f is the parity function (i.e., decide whether the total number of 1s in $(\mathbf{x}, \mathbf{y})$ is even or odd), then the 1-way communication complexity of f is 1 (Alice just sends the parity of $\mathbf{x}$ to Bob, who's then in a position to figure out the parity of $(\mathbf{x}, \mathbf{y})$).

8 Connection to Streaming Algorithms

If you care about streaming algorithms, then you should also care about 1-way communication complexity. Why? Because of the unreasonable effectiveness of the following two-step plan to proving lower bounds on the space usage of streaming algorithms.

1. Small-space streaming algorithms imply low-communication 1-way protocols.
2. The latter don't exist.

Both steps of this plan are quite doable in many cases.

Does the connection in the first step above surprise you? It's the best kind of statement — genius and near-trivial at the same time. We'll be formal about it shortly, but it's worth remembering a cartoon meta-version of the connection, illustrated in Figure 2. Consider a problem that can be solved using a streaming algorithm S that uses space only s. How can we use it to define a low-communication protocol? The idea is for Alice and Bob to treat their inputs as a stream $(\mathbf{x}, \mathbf{y})$, with all of $\mathbf{x}$ arriving before all of $\mathbf{y}$. Alice can feed $\mathbf{x}$ into S without communicating with Bob (she knows $\mathbf{x}$ and S). After processing $\mathbf{x}$, S's state is completely summarized by the s bits in its memory. Alice sends these bits to Bob. Bob can then simply restart the streaming algorithm S seeded with this initial memory, and then feed his input $\mathbf{y}$ to the algorithm. The algorithm S winds up computing some function of $(\mathbf{x}, \mathbf{y})$, and Alice only needs to communicate s bits to Bob to make it happen. The communication cost of the induced protocol is exactly the same as the space used by the streaming algorithm.

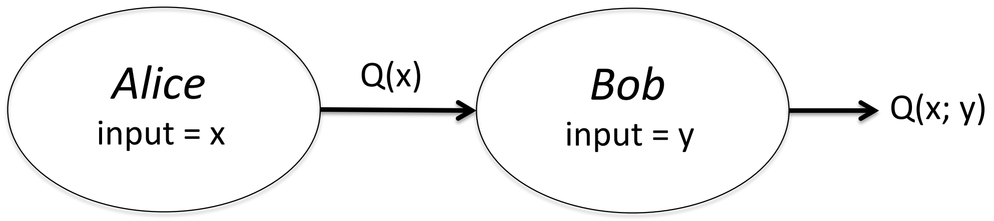

Figure 2: Why a small-space streaming algorithm induces a low-communication one-way protocol. Alice runs the streaming algorithm on her input, sends the memory contents of the algorithm to Bob, and Bob resumes the execution of the algorithm where Alice left off on his input.

9 The Disjointness Problem

To execute the two-step plan above to prove lower bounds on the space usage of streaming algorithms, we need to come up with a Boolean function that (i) can be reduced to a streaming problem that we care about and (ii) does not admit a low-communication one-way protocol.

9.1 Disjointness Is Hard for One-Way Communication

If you only remember one problem that is hard for communication protocols, it should be the Fproblem. This is the canonical hard problem in communication complexity, analogous to satisfiability (SAT) in the theory of NP-completeness. We'll see more reductions from the Fproblem than from any other in this course.

In an instance of Disjointness, both Alice and Bob hold n-bit vectors $\mathbf{x}$ and $\mathbf{y}$. We interpret these as characteristic vectors of two subsets of the universe $\{1, 2, \ldots, n\}$, with the subsets corresponding to the "1" coordinates. We then define the Boolean function $DISJ$ in the obvious way, with $DISJ(\mathbf{x}, \mathbf{y}) = 0$ if there is an index $i \in \{1, 2, \ldots, n\}$ with $x_i = y_i = 1$, and $DISJ(\mathbf{x}, \mathbf{y}) = 1$ otherwise.

To warm up, let's start with an easy result.

Proposition 9.1 *Every deterministic one-way communication protocol that computes the function $DISJ$ uses at least n bits of communication in the worst case.*

That is, the trivial protocol is optimal among deterministic protocols.[13] The proof follows pretty straightforwardly from the Pigeonhole Principle — you might want to think it through before reading the proof below.

Formally, consider any 1-way communication protocol where Alice always sends at most $n-1$ bits. This means that, ranging over the 2^n possible inputs $\mathbf{x}$ that Alice might have, she

[13]We'll see later that the communication complexity remains n even when we allow general communication protocols.

only sends 2^{n-1} distinct messages. By the Pigeonhole Principle, there are distinct messages $\mathbf{x}^1$ and $\mathbf{x}^2$ where Alice sends the same message $\mathbf{z}$ to Bob. Poor Bob, then, has to compute $DISJ(\mathbf{x}, \mathbf{y})$ knowing only $\mathbf{z}$ and $\mathbf{y}$ and not knowing $\mathbf{x}$ — $\mathbf{x}$ could be $\mathbf{x}^1$, or it could be $\mathbf{x}^2$. Letting i denote an index in which $\mathbf{x}^1$ and $\mathbf{x}^2$ differ (there must be one), Bob is really in trouble if his input $\mathbf{y}$ happens to be the ith basis vector (all zeroes except $y_i = 1$). For then, whatever Bob says upon receiving the message $\mathbf{z}$, he will be wrong for exactly one of the cases $\mathbf{x} = \mathbf{x}^1$ or $\mathbf{x} = \mathbf{x}^2$. We conclude that the protocol is not correct.

A stronger, and more useful, lower bound also holds.

Theorem 9.2 *Every randomized protocol*[14] *that, for every input* $(\mathbf{x}, \mathbf{y})$*, correctly decides the function DISJ with probability at least* $\frac{2}{3}$*, uses* $\Omega(n)$ *communication in the worst case.*

The probability in Theorem 9.2 is over the coin flips performed by the protocol (there is no randomness in the input, which is "worst-case"). There's nothing special about the constant $\frac{2}{3}$ in the statement of Theorem 9.2 — it can be replaced by any constant strictly larger than $\frac{1}{2}$.

Theorem 9.2 is certainly harder to prove than Proposition 9.1, but it's not too bad — we'll kick off next lecture with a proof.[15] For the rest of this lecture, we'll take Theorem 9.2 on faith and use it to derive lower bounds on the space needed by streaming algorithms.

9.2 Space Lower Bound for F_∞ (even with Randomization and Approximation)

Recall from Section 6 that the first weakness of Theorem 3.1 is that it applies only to F_0 and F_2 (and F_1 is easy). The next result shows that, assuming Theorem 9.2, there is no sublinear-space algorithm for computing F_∞, even probabilistically and approximately.

Theorem 9.3 ([1]) *Every randomized streaming algorithm that, for every data stream of length* m*, computes* F_∞ *to within a* $(1 \pm .2)$ *factor with probability at least* $2/3$ *uses space* $\Omega(\min\{m, n\})$.

Theorem 9.3 rules out, in a strong sense, extending our upper bounds for F_0, F_1, F_2 to all F_k. Thus, the different frequency moments vary widely in tractability in the streaming model.[16]

Proof of Theorem 9.3: The proof simply implements the cartoon in Figure 2, with the problems of computing F_∞ (in the streaming model) and DISJOINTNESS (in the one-way communication model). In more detail, let S be a space-s streaming algorithm that for

[14] There are different flavors of randomized protocols, such as "public-coin" vs. "private-coin" versions. These distinctions won't matter until next lecture, and we elaborate on them then.

[15] A more difficult and important result is that the communication complexity of DISJOINTNESS remains $\Omega(n)$ even if we allow arbitrary (not necessarily one-way) randomized protocols. We'll use this stronger result several times later in the course. We'll also briefly discuss the proof in Lecture #4.

[16] For finite k strictly larger than 2, the optimal space of a randomized $(1 \pm \epsilon)$-approximate streaming algorithm turns out to be roughly $\Theta(n^{1-1/2k})$ [2, 4, 9]. See the exercises for a bit more about these problems.

every data stream, with probability at least $2/3$, outputs an estimate in $(1 \pm .2)F_\infty$. Now consider the following one-way communication protocol $\mathcal{P}$ for solving the Disjointness problem (given an input $(\mathbf{x}, \mathbf{y})$):

1. Alice feeds into S the indices i for which $x_i = 1$; the order can be arbitrary. Since Alice knows S and $\mathbf{x}$, this step requires no communication.

2. Alice sends S's current memory state σ to Bob. Since S uses space s, this can be communicated using s bits.

3. Bob resumes the streaming algorithm S with the memory state σ, and feeds into S the indices i for which $y_i = 1$ (in arbitrary order).

4. Bob declares "disjoint" if and only if S's final answer is at most $4/3$.

To analyze this reduction, observe that the frequency of an index $i \in \{1, 2, \ldots, n\}$ in the data stream induced by $(\mathbf{x}, \mathbf{y})$ is 0 if $x_i = y_i = 0$, 1 if exactly one of x_i, y_i is 1, and 2 if $x_i = y_i = 2$. Thus, F_∞ of this data stream is 2 if $(\mathbf{x}, \mathbf{y})$ is a "no" instance of Disjointness, and is at most 1 otherwise. By assumption, for every "yes" (respectively, "no") input $(\mathbf{x}, \mathbf{y})$, with probability at least $2/3$ the algorithm S outputs an estimate that is at most 1.2 (respectively, at least $2/1.2$); in this case, the protocol $\mathcal{P}$ correctly decides the input $(\mathbf{x}, \mathbf{y})$. Since $\mathcal{P}$ is a one-way protocol using s bits of communication, Theorem 9.2 implies that $s = \Omega(n)$. Since the data stream length m is n, this reduction also rules out $o(m)$-space streaming algorithms for the problem. ■

Remark 9.4 (The Heavy Hitters Problem) Theorem 9.3 implies that computing the maximum frequency is a hard problem in the streaming model, at least for worst-case inputs. As mentioned, the problem is nevertheless practically quite important, so it's important to make progress on it despite this lower bound. For example, consider the following relaxed version, known as the "heavy hitters" problem: for a parameter k, if there are any elements with frequency bigger than m/k, then find one or all such elements. When k is constant, there are good solutions to this problem: the exercises outline the "Mishra-Gries" algorithm, and the "Count-Min Sketch" and its variants also give good solutions [5, 7].[17] The heavy hitters problem captures many of the applications that motivated the problem of computing F_∞.

9.3 Space Lower Bound for Randomized Exact Computation of F_0 and F_2

In Section 6 we also criticized our positive results for F_0 and F_2 — to achieve them, we had to make two compromises, allowing approximation and a non-zero chance of failure. The reduction in the proof of Theorem 9.3 also implies that merely allowing randomization is not enough.

[17]This does not contradict Theorem 9.2 — in the hard instances of F_∞ produced by that proof, all frequencies are in $\{0, 1, 2\}$ and hence there are no heavy hitters.

Theorem 9.5 ([1]) *For every non-negative integer $k \neq 1$, every randomized streaming algorithm that, for every data stream, computes F_∞ exactly with probability at least $2/3$ uses space $\Omega(\min\{n, m\})$.*

The proof of Theorem 9.5 is almost identical to that of Theorem 9.2. The reason the proof of Theorem 9.2 rules out approximation (even with randomization) is because F_∞ differs by a factor of 2 in the two different cases ("yes" and "no" instances of F). For finite k, the correct value of F_k will be at least slightly different in the two cases, which is enough to rule out a randomized algorithm that is exact at least two-thirds of the time.[18]

The upshot of Theorem 9.5 is that, even for F_0 and F_2, approximation is essential to obtain a sublinear-space algorithm. It turns out that randomization is also essential — every deterministic streaming algorithm that always outputs a $(1 \pm \epsilon)$-estimate of F_k (for any $k \neq 1$) uses linear space [1]. The argument is not overly difficult — see the Exercises for the details.

10 Looking Backward and Forward

Assuming that randomized one-way communication protocols require $\Omega(n)$ communication to solve the DISJOINTNESS problem (Theorem 9.2), we proved that some frequency moments (in particular, F_∞) cannot be computed in sublinear space, even allowing randomization and approximation. Also, both randomization and approximation are essential for our sublinear-space streaming algorithms for F_0 and F_2.

The next action items are:

1. Prove Theorem 9.2.

2. Revisit the five compromises we made to obtain positive results (Section 6). We've showed senses in which the first three compromises are necessary. Next lecture we'll see why the last two are needed, as well.

References

[1] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–147, 1999.

[2] Z. Bar-Yossef, T. S. Jayram, R. Kumar, and D. Sivakumar. An information statistics approach to data stream and communication complexity. In *Proceedings of the 43rd Symposium on Foundations of Computer Science (FOCS)*, pages 209–218, 2002.

[18] Actually, this is not quite true (why?). But if Bob also knows the number of 1's in Alice's input (which Alice can communicate in $\log_2 n$ bits, a drop in the bucket), then the exact computation of F_k allows Bob to distinguish "yes" and "no" inputs of DISJOINTNESS (for any $k \neq 1$).

[3] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and Luca Trevisan. Counting distinct elements in a data stream. In *Proceedings of the 6th International Workshop on Randomization and Approximation Techniques (RANDOM)*, pages 1–10, 2002.

[4] A. Chakrabarti, S. Khot, and X. Sun. Near-optimal lower bounds on the multi-party communication complexity of set disjointness. In *Proceedings of the 18th Annual IEEE Conference on Computational Complexity*, pages 107–117, 2003.

[5] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. *Theoretical Computer Science*, 312(1):3–15, 2004.

[6] A. Chattopadhyay and T. Pitassi. The story of set disjointness. *SIGACT News*, 41(3):59–85, 2010.

[7] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[8] P. Flajolet and G. N. Martin. Probabilistic counting. In *Proceedings of the 24th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 76–82, 1983.

[9] P. Indyk and D. P. Woodruff. Optimal approximations of the frequency moments of data streams. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing (STOC)*, pages 202–208, 2005.

[10] S. Jukna. *Boolean Function Complexity: Advances and Frontiers.* Springer, 2012.

[11] E. Kushilevitz and N. Nisan. *Communication Complexity.* Cambridge, 1996.

[12] T. Lee and A. Shraibman. Lower bounds in communication complexity. *Foundations and Trends in Theoretical Computer Science*, 3(4):263–399, 2009.

[13] L. Lovász. Communication complexity: A survey. In B. H. Korte, editor, *Paths, Flows, and VLSI Layout*, pages 235–265. Springer-Verlag, 1990.

[14] M. Pătraşcu. Communication complexity I, II, III, IV. WebDiarios de Motocicleta blog posts dated May 23, 2008; March 25, 2009; April 3, 2009; April 8, 2009.

[15] A. A. Razborov. Communication complexity. In D. Schleicher and M. Lackmann, editors, *An Invitation to Mathematics: From Competitions to Research*, pages 97–117. Springer, 2011.