CSC473: Advanced Algorithm Design Winter 2018

Weeks 4-5: Streaming and Parallel Algorithms

Aleksandar Nikolov

1 Motivation and Model

In the next few lectures we will study algorithms which are *extremely* efficient, in terms of space, and usually in terms of time complexity as well. This is motivated by real-world scenarios in which we want to process enormous amounts of data quickly. For example:

- A network service provider wants to keep track of the traffic going through the network, in order to detect unusual patterns, which may be a sign of malicious activity. For example, an increased traffic to a particular IP address may suggest a Denial of Service attack is under way. Such unusual traffic patterns should be detected almost in real time, however the throughput of the network is so large that it's not feasible to store all the traffic data and analyze it. So, you need a solution which works online (i.e. processes the traffic as it passes through the routers), and accurately summarizes the data in a very concise "sketch."
- Processing data in a very large database can be very expensive in terms of time. Often it's a good idea to quickly compute some rough statistics about the data before embarking on a more expensive analysis. Because of its size, the database usually cannot be entirely loaded into the computer memory, and most of it will be on a relatively slow hard disk. The fastest way to process such a database is to do so sequentially, making only a few passes over the data, as opposed jumping back and forth between different records. So, we are interested in statistics which can be computed approximately in a few sequential passes over the data.

Scenarios like these motivate the simple and elegant streaming computation model, described below. In the last week of the course we will see that the techniques we develop for streaming algorithms will also be useful when designing algorithms that work on parallel systems.

In the streaming model, the algorithm $\mathcal{A}$ works in time steps, and receives one update per time step. In the simplest version of the model, an update is just an integer in the set $\{1, \dots, n\}$, which is meant as an identifier of some object. For example, the integer can encode an IP address. The sequence of updates is called the *stream.* The total number of updates m is called the length of the stream, and may or may not be explicitly given to the algorithm. After receiving an update, and before receiving the next one, $\mathcal{A}$ updates is memory. Moreover, at any point during its execution, the algorithm should be able to report an answer to a given question, e.g. report the number of distinct integers seen so far. The crucial constraint is that, at any time step, the algorithm is only allowed to store in its memory a number of bits which is bounded by a polynomial in $\log n$ and $\log m$. I.e. the algorithm's memory may never exceed $O(\log^c(nm))$ bits, where c is some fixed constant, say 1 or 2. This constraint models the fact that, in the applications we mentioned above, the size of the data (corresponding to m), and the size of the universe it is coming from (corresponding to n), are very large. While the model does not pose an explicit constraint on the running time, it's desirable that updates are as efficient as possible.

Often when we analyze algorithm in the streaming model, we count the number of words used by the algorithm, where a word is a block of $O(\log(nm))$ bits, which can store, e.g. a single variable, or a single cell in an array. We will adopt this convention in these notes.

Many fundamental problems in the streaming model are conveniently summarized by the *frequency vector* f, which gives the number of times each element of the universe appears in the stream. I.e. if we have a stream $\sigma = (i_1, \ldots, i_t)$ consisting of updates in $\{1, \ldots, n\}$, then $f \in \mathbb{Z}^n$ is defined by $f_i = |\{t : i_t = i\}|$. Note that the streaming algorithm will not actually store the frequency vector f, because that would require space linear in n. However, it helps to refer to the vector when defining problems or analyzing algorithms.

In some versions of the model, updates can have a richer meaning. For example, in the *turnstile* model, an update is a pair (i, s), where $i \in \{1, \ldots, n\}$, and $s \in \{-1, 1\}$. As before, i is just an identifier; s indicates "entering," when $s = +1$, or "leaving," when $s = -1$. For example, i can identify a particular subway station, and s can indicate whether at the given time step a customer has entered, or left the station. In the turnstile model we define the frequency vector f for a stream $\sigma = \{(i_1, s_1), \ldots, (i_m, s_m)\}$ by $f_i = \sum_{t:i_t=i} s_t$.

Exercise 1. *Suppose the stream has length $n-1$, and consists of $n-1$ of the integers $\{1, \ldots, n\}$. All updates are distinct, and the integers can appear in an arbitrary order. Design an algorithm which makes a single pass over the data, keeps only a constant number of words of memory, and finds the missing integer from the stream. Adapt your solution to finding two missing integers in a stream of $n-2$ distinct updates from $\{1, \ldots, n\}$.*

Exercise 2. *Give an algorithm which samples a random element in a stream $\sigma = (i_1, \ldots, i_m)$ of updates in $\{1, \ldots, n\}$ so that the probability that element i is sampled equals $\frac{|\{t:i_t=i\}|}{m}$. The algorithm should keep only a single variable, which at every time step is equal to the random sample from the portion of the stream seen so far. Do* not *assume the algorithm knows the length of the stream.*

2 Frequent Elements

It should be surprising to you that it is at all possible to do anything in the streaming model. Often approximation and randomization are both absolutely necessary to solve a problem. We will start with a rare example of a problem for which there are efficient deterministic algorithms.

As a warm-up, we consider the Majority problem. You are given a stream $\sigma = (i_1, \ldots, i_m)$ of updates in $[n] = \{1, \ldots n\}$. If there exists an $i \in [n]$ such that more than half the updates in σ are equal to i, the algorithm should output i. If no such majority element exists, the algorithm can output any element. (Notice that only one such element can exist.) The algorithm below (due to Boyer and Moore) solves this problem with *only two words of memory*:

```
MAJORITY($\sigma$)
$element = i_1$
$count = 1$
for $t = 2$ to $m$
    if $element == i_t$
        $count = count + 1$
    elseif $count > 0$
        $count = count - 1$
    else $element = i_t$
        $count = 1$
return $element$
```

Theorem 1. *If there exists an element i such that more than half the updates in σ are equal to i, then the* MAJORITY *algorithm outputs i. Moreover, at any point during the execution of the algorithm, $f_{element} \leq count + m/2$.*

Proof. We group the updates in the stream in pairs as follows. At the start, we leave the first update i_1 unpaired and move to $t = 2$. If $i_t = element$, or if $i_t \neq element$ but $count = 0$, we leave the update i_t unpaired for now and move on to the next value of t. If $i_t \neq element$ and $count > 0$, we pair i_t with one of the prior updates i_s, $s < t$, for which $i_s = element$, and i_s is still unpaired. Then we move on to the next value of t. The idea behind this pairing procedure is that we view the event of *count* decreasing at time step t as i_t "taking away" one of the prior instances of *element*.

The following fact is easily shown by induction:

- At any time step t, *count* is equal to the number of yet unpaired updates equal to *element*. For all $j \neq element$, all updates equal to j are paired.

Let i be as in the statement of the theorem. Observe that for every pair (i_s, i_t) we have $i_s \neq i_t$. So, since there are at most $m/2$ pairs in the pairing, at most $m/2$ of the updates equal to i are paired. This means that at least $f_i - m/2 > 0$ updates equal to i are left unpaired at the end of the execution of the algorithm. By the claim above, this means the final value of *element* returned by the algorithm is i, and $count \geq f_i - m/2$. Notice that this also proves the claim after "moreover" for the majority element i. If there is no majority element, that claim holds trivially. □

If we want to also verify that the element i returned by the algorithm is a majority element, we can just make a second pass over the stream and just count the number of occurrences of i.

Next we study a generalization of MAJORITY which finds all elements that appear in more than $1/k$ fraction of the updates. This algorithm is due to Misra and Gries.

```
Frequent($\sigma, k$)
$S = \emptyset$
for $t = 1$ to $m$
    if $\exists x \in S$ such that $x.\mathit{elem} == i_t$
        $x.\mathit{count} = x.\mathit{count} + 1$
    elseif $|S| < k - 1$
        Create an element $x$ with $x.\mathit{elem} = i_t$ and $x.\mathit{count} = 1$
        $S = S \cup x$
    else for $x \in S$
            $x.\mathit{count} = x.\mathit{count} - 1$
            if $x.\mathit{count} == 0$
                $S = S \setminus \{x\}$
Return $S$
```

Theorem 2. *The set S output by* Frequent *contains all $i \in [n]$ such that $f_i > m/k$. Moreover, for any $x \in S$, $f_{x.\mathit{elem}} \le x.\mathit{count} + m/k$.*

Proof. We group the updates in the stream in groups of size k. Let us say that an element $i \in [n]$ is represented in S if there exists an $x \in S$ such that $x.\mathit{elem} = i$. Starting from $t = 1$, we look at the following cases:

1. If i_t is represented in S, we leave i_t ungrouped for now, and move to the next value of t.

2. Similarly, if i_t is not represented in S, but $|S| < k - 1$, we leave i_t ungrouped for now, and move to the next value of t.

3. If i_t is not represented in S and $|S| = k - 1$, then we group i_t with $k - 1$ previously ungrouped distinct updates represented in S. I.e. we find $k - 1$ updates, occurring at times $s_1, \ldots, s_{k-1}$, all preceding i_t and ungrouped, with the property that $i_{s_1}, \ldots, i_{s_{k-1}}$ are all distinct, and represented in S. We group i_t with $i_{s_1}, \ldots, i_{s_{k-1}}$ and move to the next value of t.

It is easy to prove the following crucial claim by induction:

- At any time step t, for any $x \in S$, $x.\mathit{count}$ is equal to the number of yet ungrouped updates equal to $x.\mathit{elem}$. For all $j \in [n]$ not represented in S, all updates equal to j are grouped.

Let i be such that $f_i > m/k$. Observe that the groups are defined so that they contain distinct elements of $[n]$. Because there can be at most m/k groups, at least $f_i - m/k > 0$ updates equal to i are left ungrouped. By the claim above, this means that i is represented in S, and that $x.\mathit{count} \ge f_i - m/k$, where $x \in S$ is such that $x.\mathit{elem} = i$. This also proves the claim after "moreover" for all i such that $f_i > m/k$. The claim is trivial for all other elements. □

Notice that Frequent(σ, k) will output all elements with $f_i > m/k$, but it may also output some infrequent elements. To be sure which elements are have frequency greater than m/k, and which do not, you need to make another pass over the stream.

Exercise 3. *Show how to implement* FREQUENT *so that each update is as efficient as possible. What data structure would you use to represent S in memory? What is the worst-case time complexity of an update? What is the amortized time complexity of an update (i.e. the total time taken by the algorithm, divided by m).*

Exercise 4. *Design an algorithm in the streaming model that, given values ϕ and ε such that $0 < \varepsilon < \phi < 1$, outputs a set $S \subseteq [n]$ such that:*

1. *If i is such that $f_i > \phi m$, then $i \in S$;*
2. *If i is such that $f_i < (\phi - \varepsilon)m$, then $i \notin S$.*

Your algorithm should use $O(\frac{1}{\varepsilon})$ words of memory.

3 Distinct Elements Count

In the distinct elements count problem we want to know how many distinct integers we have seen in the stream so far. In terms of the frequency vector, we want to approximate $F_0 = |\{i : f_i > 0\}|$. For this problem it turns out that we need to allow both approximation and randomization in order to satisfy the constraints of the streaming model.

3.1 Warm-Up

We first give an algorithm which makes the (unrealistic) assumption that we know a number $\tilde{F_0}$ such that

$$F_0 \le \tilde{F_0} \le 2F_0. \tag{1}$$

Our goal in this case is to refine this "rough" estimate $\tilde{F_0}$ to a much more precise one by processing the stream. In the next subsection we will see how to remove this unrealistic assumption by building on the ideas from this subsection.

The algorithm to refine the rough estimate $\tilde{F_0}$ works by a simple sampling strategy:

```
DISTINCT-SIMPLE($\sigma, k, \tilde{F_0}$)
$S = \emptyset$
$d = \lceil \log_2(\tilde{F_0}/k) \rceil$
$L = \lceil \log_2 n \rceil$
Pick a hash function $h : [n] \to \{0,1\}^L$
for $t = 1$ to $m$
    if $h(i_t) \in 0^d\{0,1\}^{L-d}$ and $i_t \notin S$
        $S = S \cup \{i_t\}$
return $\hat{F_0} = 2^d \cdot |S|$
```

A few clarifications are in order. In the analysis we assume that h behaves like a random function. I.e., we assume that $h(1), \ldots, h(n)$ are n independent random variables, uniformly distributed in $\{0,1\}^L$. This is essentially the simple uniform hashing assumption you may remember from your data structures course. The assumption can be removed using a construction similar to universal hashing, and we discuss this a little bit later. We also clarify the notation: by $h(i_t) \in 0^d\{0,1\}^{L-d}$ we simply mean that the leftmost d bits of $h(i_t)$ are all 0.

So, in words, the algorithm works as follows: it uses the hash function to keep a set S which contains each element that appears in the stream with probability 2^{-d}. In expectation, S then contains 2^{-d} fraction of the elements that appear in the stream, so our estimate of the number of distinct elements is $2^d|S|$. The value d is chosen so that the expected size of S is at most k: this is the only place where we use $\tilde{F_0}$; if we did not know such an estimate, we would not know how to set d. There is a natural trade-off here: the larger we pick k (and, therefore, the smaller we pick d), the more space our algorithm uses, but the more accurate its estimate becomes.

To analyze the algorithm, we will recall a basic concept from probability theory: *variance*. The variance of a real-valued random variable X is defined as $\mathrm{Var}(X) = \mathbb{E}[(X - \mathbb{E}[X])^2]$. It measures how much X deviates from its expectation on average. The following calculation is often very useful:

$$\begin{aligned}\mathrm{Var}(X) = \mathbb{E}[(X - \mathbb{E}[X])^2] &= \mathbb{E}[X^2 - 2X \cdot \mathbb{E}[X] + \mathbb{E}[X]^2]\\ &= \mathbb{E}[X^2] - 2\mathbb{E}[X]^2 + \mathbb{E}[X]^2\\ &= \mathbb{E}[X^2] - \mathbb{E}[X]^2.\end{aligned}$$

A basic fact about variance is that the variance of the sum of independent random variables is equal to the sum of their variances.

Proposition 3. *Let $X_1, \dots, X_n$ be independent random variables, and let $X = \sum_{i=1}^n X_i$. Then $\mathrm{Var}(X) = \sum_{i=1}^n \mathrm{Var}(X_i)$.*

Proof. For simplicity, let us define new random variables $Y_i = X_i - \mathbb{E}[X_i]$, and $Y = \sum_{i=1}^n Y_i$. Notice $Y_1, \dots, Y_n$ are also independent, that $Y = X - \mathbb{E}[X]$, $\mathbb{E}[Y_i] = 0$ for all i, $\mathbb{E}[Y] = 0$, and each of $Y, Y_1, \dots, Y_n$ has the same variance as $X, X_1, \dots, X_n$, respectively. Then, it is enough to prove the proposition for Y and $Y_1, \dots, Y_n$. We have:

$$\begin{aligned}\mathrm{Var}(Y) = \mathbb{E}[Y^2] &= \mathbb{E}[(Y_1 + \ldots + Y_n)^2]\\ &= \sum_{i=1}^n \mathbb{E}[Y_i^2] + \sum_{i \neq j} \mathbb{E}[Y_i Y_j]\\ &= \sum_{i=1}^n \mathbb{E}[Y_i^2] + \sum_{i \neq j} \mathbb{E}[Y_i]\mathbb{E}[Y_j]\\ &= \sum_{i=1}^n \mathbb{E}[Y_i^2] = \sum_{i=1}^n \mathrm{Var}(Y_i).\end{aligned}$$

The third line above follows from the assumption that Y_i and Y_j are independent for each $i \neq j$. □

The main reason variance is useful is that it gives us some control on how far a random variable can be from its expectation. Let us first recall Markov's inequality, proved in the lecture notes on Locality Sensitive Hashing.

Theorem 4 (Markov's Inequality)**.** *Let $Z \geq 0$ be a random variable. Then, for any $z > 0$,*

$$\mathbb{P}(Z > z) < \frac{\mathbb{E}[Z]}{z}.$$

We will use Markov's inequality to prove Chebyshev's inequality.

Theorem 5 (Chebyshev's Inequality)**.** *Let X be a random variable. For any $t > 0$,*

$$\mathbb{P}(|X - \mathbb{E}[X]| > t) < \frac{\mathrm{Var}(X)}{t^2}.$$

Proof. Define the random variable $Z = (X - \mathbb{E}[X])^2$. By definition, $Z \geq 0$. Notice that $\mathrm{Var}(X) = \mathbb{E}[Z]$ and that

$$|X - \mathbb{E}[X]| > t \Leftrightarrow Z > t^2.$$

Then, by Markov's inequality,

$$\mathbb{P}(|X - \mathbb{E}[X]| > t) = \mathbb{P}(Z > t^2) < \frac{\mathrm{Var}(X)}{t^2}.$$

□

Notice that, unlike Markov's inequality, Chebyshev's inequality does not need to assume that the random variable is non-negative. Moreover, Chebyshev's inequality bounds the probability that the random variable is far from its expectation in either direction.

Exercise 5. *Give, for each value of $t > 0$, a random variable X so that $\mathbb{E}[X] = 0$, $\mathrm{Var}(X) = 1$, and such that Chebyshev's inequality holds with equality, i.e. $\mathbb{P}(|X - \mathbb{E}[X]| \geq t) = \frac{1}{t^2}$.*

Exercise 6. *Let $X_1, \dots, X_n$ be n random bits, where $\mathbb{P}(X_i = 1) = p$. Assume that $X_1, \dots, X_n$ are mutually independent. What is the variance of $-X_1 + X_2 - X_3 + \ldots + (-1)^n X_n$?*

Exercise 7. *Suppose we roll a fair die n times, and let X be the sum of the numbers that appeared over the n rolls. Use Chebyshev's inequality to bound the probability that $|X - 3.5n| > 0.5n$.*

Exercise 8. *Let $A[1\,..\,n]$ be an array of* distinct *integers. Recall that the* rank *of an integer x in A equals r if and only if there are exactly $r - 1$ integers in A that are strictly smaller than x.*

Consider k indexes $i_1, \dots, i_k$, sampled independently and uniformly at random with replacement from $\{1, \dots, n\}$. Let ℓ be the number of integers in $A[i_1], \dots, A[i_k]$, counted with repetition, that have rank $> \left(\frac{1}{2} + \varepsilon\right) n$. What is the expected value of ℓ (as a function of ε)? What is the variance of ℓ?

Use Chebyshev's inequality to give an upper bound on the probability that the median of $A[i_1], \dots, A[i_k]$ has rank $> \left(\frac{1}{2} + \varepsilon\right) n$ in A.

Exercise 9. *Given a permutation $\sigma = \sigma_1, \dots, \sigma_n$ of $[n]$, let $\mathrm{inv}(\sigma)$ be the number of distinct pairs $i < j$ such that $\sigma_i > \sigma_j$. I.e. $\mathrm{inv}(\sigma)$ is the number of pairs of integers that are inverted in σ. If σ is a uniformly random permutation of $[n]$, then what is $\mathbb{E}[\mathrm{inv}(\sigma)]$? What is $\mathrm{Var}(\mathrm{inv}(\sigma))$?*

HINT*: Use the indicator random variables X_{ij}, defined for any $1 \leq i < j \leq n$, where $X_{ij} = 1$ if $\sigma_i > \sigma_j$, and $X_{ij} = 0$ otherwise. Compute $\mathbb{E}[X_{ij}]$, and use this to calculate $\mathbb{E}[\mathrm{inv}(\sigma)]$. For $\mathrm{Var}(\mathrm{inv}(\sigma))$, expand $\mathbb{E}[\mathrm{inv}(\sigma)^2]$ and compute the quantities*

1. $\mathbb{E}[X_{ij}X_{k\ell}]$ *where* $i \neq k$ *and* $j \neq \ell$*;*

2. $\mathbb{E}[X_{ij}X_{i\ell}]$ *where* $j \neq \ell$*;*

3. $\mathbb{E}[X_{ij}X_{j\ell}]$*;*

4. $\mathbb{E}[X_{ij}^2]$*.*

Use these expectations to compute $\mathbb{E}[\mathrm{inv}(\sigma)^2]$ *and* $\mathrm{Var}(\mathrm{inv}(\sigma))$*.*

We are now ready to analyze Distinct-Simple.

Theorem 6. *If* $\tilde{F}_0$ *satisfies* (1)*, then with probability at least* $1/2$*, the estimate* $\hat{F}_0$ *output by* Distinct-Simple$(\sigma, \tilde{F}_0, k)$ *satisfies*

$$\left(1 - \frac{\sqrt{8}}{\sqrt{k}}\right) F_0 \leq \hat{F}_0 \leq \left(1 + \frac{\sqrt{8}}{\sqrt{k}}\right) F_0.$$

Moreover, the algorithm uses at most $O(k)$ *words of memory in expectation.*

Proof. Let D be the set of those $i \in [n]$ that appear in the stream, i.e. $D = \{i : f_i > 0\}$. By this definition, $F_0 = |D|$. For any $i \in D$, let X_i be the indicator random variable which equals 1 if $i \in S$, and 0 otherwise. We have $\mathbb{E}[X_i] = \mathbb{P}(i \in S) = 2^{-d}$ because $h(i)$ is a uniformly random string in $\{0,1\}^L$ and exactly 2^{L-d} out of all the 2^L such strings have their d leftmost bits set to 0. It follows that

$$\mathbb{E}[|S|] = \mathbb{E}\left[\sum_{i \in D} X_i\right] = \sum_{i \in D} \mathbb{P}(i \in S) = 2^{-d} F_0.$$

Since $F_0 \leq \tilde{F}_0$, and we chose d so that $2^{-d}\tilde{F}_0 \leq k$, we have $\mathbb{E}[|S|] = 2^{-d}F_0 \leq 2^{-d}\tilde{F}_0 \leq k$. Since storing S dominates the space usage of the algorithm, it follows that the algorithm uses at most $O(k)$ words of memory in expectation. It remains to analyze how close $\hat{F}_0$ is to F_0.

It is easy to see that $\hat{F}_0$ *equals* F_0 in expectation: $\mathbb{E}[\hat{F}_0] = \mathbb{E}[2^d|S|] = F_0$. However, this is not enough to show that $\hat{F}_0$ is close to F_0: a random variable could be very far from its expectation with high probability. This is where Chebyshev's inequality comes to the rescue: if we know that the variance of $\hat{F}_0$ is small, then we know it is unlikely to be too far from its expectation.

Recall that we assumed $h(1), \ldots, h(n)$ are all independent, and, therefore, the random variables X_i defined for each $i \in D$ are also independent. By Proposition 3, it follows that

$$\mathrm{Var}(|S|) = \mathrm{Var}\left(\sum_{i \in D} X_i\right) = \sum_{i \in D} \mathrm{Var}(X_i).$$

For any $i \in D$,

$$\mathrm{Var}(X_i) = \mathbb{E}[X_i^2] - \mathbb{E}[X_i]^2 \leq \mathbb{E}[X_i^2] = \mathbb{E}[X_i] = 2^{-d}.$$

Here, we used the fact that $\mathbb{E}[X_i]^2 \geq 0$, and that $X_i^2 = X_i$ because $X_i \in \{0,1\}$. Plugging this into the equation for $\mathrm{Var}(|S|)$, we get that $\mathrm{Var}(|S|) \leq 2^{-d}F_0 = \mathbb{E}[|S|]$.

We are ready to finish the proof. Let $\varepsilon = \frac{\sqrt{8}}{\sqrt{k}}$. By Chebyshev's inequality,

$$\begin{aligned}\mathbb{P}(|\hat{F}_0 - F_0| \geq \varepsilon F_0) &= \mathbb{P}(|\hat{F}_0 - \mathbb{E}[\hat{F}_0]| \geq \varepsilon \mathbb{E}[\hat{F}_0]) \\ &= \mathbb{P}(||S| - \mathbb{E}[|S|]| \geq \varepsilon \mathbb{E}[|S|]) \\ &\leq \frac{\mathrm{Var}(|S|)}{\varepsilon^2 \mathbb{E}[|S|]^2} \leq \frac{1}{\varepsilon^2 \mathbb{E}[|S|]}.\end{aligned}$$

I.e. if the expected size of S is not too small, we have a large probability of an accurate estimate. (This makes intuitive sense: a large sample gives a better estimate.) By the choice of d we know that $2^d \leq 2\tilde{F}_0/k$, which, after re-arranging, gives $2^{-d}\tilde{F} \geq k/2$. Therefore, by (1),

$$\mathbb{E}[|S|] = 2^{-d} F_0 \geq 2^{-d-1} \tilde{F}_0 \geq \frac{k}{4}.$$

Substituting into the expression we got from Chebyshev's inequality, we have

$$\mathbb{P}(|\hat{F}_0 - F_0| \geq \varepsilon F_0) \leq \frac{4}{\varepsilon^2 k} = \frac{1}{2},$$

as we wanted. □

Exercise 10. *Use Exercise 8 to analyze the following algorithm: given input array $A[1\,..\,n]$ of distinct integers, the algorithm samples k indexes $i_1, \dots, i_k$ independently and uniformly at random with replacement from $\{1, \dots, n\}$. Then it computes and outputs the median X of $A[i_1], \dots, A[i_k]$. Show that if $k \geq \frac{4}{\varepsilon^2}$, then the output X of the of the algorithm has rank between $\left(\frac{1}{2} - \varepsilon\right) n$ and $\left(\frac{1}{2} + \varepsilon\right) n$ in A with probability at least $\frac{1}{2}$.*

3.2 Adaptive Sampling

Next we describe a single pass streaming algorithm for distinct counts which does not assume we know an estimate $\hat{F}_0$. We describe one of the simpler solutions for this problem, known as Adaptive Sampling, as the procedure Distinct.

```
Distinct($\sigma, k$)
$S = \emptyset$
$d = 0$
$L = \lceil \log_2 n \rceil$
Pick a hash function $h : [n] \to \{0,1\}^L$
for $t = 1$ to $m$
    if $h(i_t) \in 0^d\{0,1\}^{L-d}$ and $i_t \notin S$
        $S = S \cup \{i_t\}$
    while $|S| > k$
        $d = d + 1$
        $T = \emptyset$
        for $j \in S$
            if $h(j) \in 0^d\{0,1\}^{L-d}$
                $T = T \cup \{j\}$
            $S = T$
return $\hat{F}_0 = 2^d \cdot |S|$
```

This algorithm is a variant of DISTINCT-SIMPLE which sets the value d, and, consequently, the sampling rate, adaptively. The algorithm keeps the invariant that $|S| \le k$, and, whenever S exceeds size k, we increase d by 1, and drop, in expectation, half of its elements. At the end, we expect S to hold about $2^{-d}F_0$ elements, so we output $2^d|S|$ as our estimate of F_0. Notice that the algorithm is guaranteed to use $O(k)$ words of memory.

We are now ready to analyze DISTINCT. Warning: this is one of the more involved probabilistic analyses you will see in this course, and it is somewhat heavy in calculations.

Theorem 7. *Let $k \ge 144$. With probability at least $1/2$, the estimate $\hat{F}_0$ output by* DISTINCT(σ, k) *satisfies*

$$\Big(1 - \frac{4}{\sqrt{k}}\Big)F_0 \le \hat{F}_0 \le \Big(1 + \frac{4}{\sqrt{k}}\Big)F_0.$$

Proof. Let D be the set of those $i \in [n]$ that appear in the stream, i.e. $D = \{i : f_i > 0\}$. By this definition, $F_0 = |D|$. Let us define the sets $S_0, S_1, \ldots, S_L$ by $S_\ell = \{i \in D : h(i) \in 0^\ell\{0,1\}^{L-l}\}$, and observe that $S_L \subseteq S_{L-1} \subseteq \ldots \subseteq S_0 = D$. It is easy to show that the final value of d computed by DISTINCT equals $\min\{\ell : |S_\ell| \le k\}$, and the estimate output by the algorithm equals $2^d|S_d|$.

Using a calculation analogous to the one in Theorem 6, we get

$$\begin{aligned} \mathbb{E}[|S_\ell|] &= 2^{-\ell}F_0, \\ \mathrm{Var}(|S_\ell|) &\le 2^{-\ell}F_0 = \mathbb{E}[|S_\ell|]. \end{aligned}$$

Let $\varepsilon = \frac{4}{\sqrt{k}} \le \frac{1}{3}$, and let a be the largest integer such that $(1-\varepsilon)2^{-a}F_0 > k$, and let b be the smallest integer such that $(1+\varepsilon)2^{-b}F_0 \le k$. Verify that $b \ge a$, and $b - a \le 1 + \log_2 \frac{1+\varepsilon}{1-\varepsilon} \le 2$, i.e. $b \in \{a, a+1, a+2\}$.

Since $\mathbb{E}[|S_a|] = 2^{-a}F_0$, we have, by Chebyshev's inequality

$$\begin{aligned} \mathbb{P}(|S_a| \le k) &\le \mathbb{P}(|S_a| < (1-\varepsilon)2^{-a}F_0) \\ &= \mathbb{P}(\mathbb{E}[|S_a|] - |S_a| > \varepsilon\mathbb{E}[|S_a|]) \\ &< \frac{\mathrm{Var}(|S_a|)}{\varepsilon^2\mathbb{E}[|S_a|]^2} \le \frac{1}{\varepsilon^2\mathbb{E}[|S_a|]}. \end{aligned}$$

Since $E[|S_a|] > k$, and $\varepsilon^2 = \frac{16}{k}$, the right hand side is at most $1/16$. By an analogous calculation:

$$\mathbb{P}(\big||S_b| - \mathbb{E}[|S_b|]\big| > \varepsilon 2^{-b}F_0) = \mathbb{P}(\big||S_b| - \mathbb{E}[|S_b|]\big| > \varepsilon\mathbb{E}[|S_b|]) < \frac{1}{\varepsilon^2\mathbb{E}[|S_b|]}.$$

By the choice of b, $(1+\varepsilon)2^{-b}F_0 \ge k/2$ (or we would've chosen a smaller b), so $\mathbb{E}[|S_b|] = 2^{-b}F_0 \ge 3k/8$. Since $\varepsilon^2 = \frac{16}{k}$, the right hand side of the inequality above is at most $1/6$. Another analogous calculation shows that, if $b = a+2$, then

$$\mathbb{P}(\big||S_{b-1}| - \mathbb{E}[|S_{b-1}|]\big| > \varepsilon 2^{-b+1}F_0) < \frac{1}{6}.$$

By the union bound, with probability at least $1 - 1/6 - 1/6 - 1/16 > 1/2$, we have

$$\begin{aligned} |S_a| &> k, \\ \big||S_{b-1}| - \mathbb{E}[|S_{b-1}|]\big| &\le \varepsilon 2^{-b+1}F_0, \\ \big||S_b| - \mathbb{E}[|S_b|]\big| &\le \varepsilon 2^{-b}F_0. \end{aligned}$$

The last inequality and the choice of b imply that $|S_b| \le k$. Therefore, at the end of the execution of the algorithm, $d = b$ or $d = b-1$, and the algorithm outputs the estimate $\hat{F}_0 \in \{2^{b-1}|S_{b-1}|, 2^b|S_b|\}$. If $\hat{F}_0 = 2^{b-1}|S_{b-1}|$, we have

$$\left|\hat{F}_0 - F_0\right| = 2^{b-1}\big||S_{b-1}| - \mathbb{E}[|S_{b-1}|]\big| \le \varepsilon F_0.$$

Analogously, if $\hat{F}_0 = 2^b|S_b|$, we have

$$\left|\hat{F}_0 - F_0\right| = 2^{b}\big||S_{b}| - \mathbb{E}[|S_{b}|]\big| \le \varepsilon F_0.$$

This completes the proof of the theorem. □

The constants in the proof are not chosen to be the tightest possible, but rather to make the calculations relatively painless. Another way to state the result we proved is that, using Distinct, we can approximate F_0 up to a factor $1 \pm \varepsilon$ using $O(\frac{1}{\varepsilon^2})$ words of memory, or, equivalently, $O(\frac{\log n}{\varepsilon^2})$ bits of memory. More sophisticated algorithms are known which use $O(\frac{1}{\varepsilon^2} + \log n)$ bits of memory; this latter bound is the best possible in the worst case.

Exercise 11. *The theorem above does not show any guarantee when $k = 1$. Prove that there exists a constant C, independent of n, m or the stream, so that if $\hat{F}_0$ is the estimate output by* Distinct$(\sigma, 1)$*, then, with probability at least* $1/2$,

$$\frac{1}{C}F_0 \le \hat{F}_0 \le CF_0$$

Let us finally make a remark about making the assumptions on the hash function in the algorithm more realistic. Carefully checking the calculation in Proposition 3 shows that it is enough if the random variables $X_1, \ldots, X_n$ are *pairwise independent*, i.e. if for every $i \ne j$ and every two values x, x', we have $\mathbb{P}(X_i = x, X_j = x') = \mathbb{P}(X_i = x)\mathbb{P}(X_j = x')$. (This is not the same as full independence: take for example X_1 and X_2 to be independent and uniform in $\{0,1\}$, and take X_3 to be the XOR of X_1 and X_2.)

This observation inspires the following definition, which is closely related to the definition of a universal hashing family.

Definition 8. *A family $\mathcal{H}$ of functions from $[n]$ to $\{0,1\}^L$ is a* pairwise independent *hash family if for all $i, j \in [n]$, $i \ne j$, and for any two bit-strings $x, x' \in \{0,1\}^L$, we have*

$$\mathbb{P}(h(i) = x, h(j) = x') = \frac{1}{2^{2L}},$$

where the probability is taken over picking a uniformly random $h \in \mathcal{H}$.

A pairwise independent hash family is also a universal family, but not necessarily the other way around. Using a little bit of algebra, it is easy to construct pairwise independent hash family of size $2^{2L} = O(n^2)$. Picking a random function from this family requires only picking $2L = O(\log n)$ random bits, and, moreover, the value of the hash function on any element of $[n]$ can be quickly computed from the random bits. So, in Distinct we will take h to be a random function from such a pairwise independent hash family. This will only increase our space complexity by a constant number of words, and will not affect the correctness guarantees of the algorithm, because in the analysis we only used pairwise independence.

4 Parallel Algorithms

4.1 The Model

In this section we explore another computational model which addresses the challenges of processing big data. We still consider a setting in which the data we want to process is too large to fit on a machine. However, we are going to assume that we possess a cluster of many machines: enough so that the total space on them fits all our data, and potentially slightly more than that. For concreteness, let us say that the input data is a set of n integers, we have $\sqrt{n}$ machines, and each of them can keep $O(\sqrt{n})$ integers in its local storage. Assume that the input starts out partitioned arbitrarily among the machines. Each machine can compute on the part of the input in its local storage. Since no machine sees the entire input, we cannot hope to solve even very simple problems, unless we allow the machines to exchange messages. However, communication between the machines is usually slow: sending data over a network is much slower than internal computation. For this reason, we will try to design algorithms that process all the data with as little communication between the machines as possible.

Let us describe the model more formally. Assume that the size of the input is n (measured, for example, in words of memory). We have m machines, and each of them has local storage of s words of memory. We assume that $ms = \Omega(n)$, and sometimes even allow for $ms = \Omega(n^{1+\varepsilon})$ for some relatively small constant $\varepsilon > 0$. The input starts out partitioned arbitrarily between the m machines. The computation proceeds in *rounds*. In a single round, each machine can execute any polynomial time algorithm on its local storage. Once all machines have finished their local computation, they are allowed to exchange messages. All machines simultaneously send messages to the other machines (assume the machines are numbered). The total size of the messages sent or received by a machine must not exceed s. Once the messages are exchanged, the round is complete. In the next round the local storage of each machine contains the union of the messages it received in the previous round together with the contents of its local storage from the previous round. We have the constraint that the local storage of any machine must not exceed s at any time. To satisfy this constraint, a machine can discard information from its local storage during the algorithm. Our main goal is to complete the entire computation in as few rounds as possible, preferably a constant number of rounds.

This model is an abstraction and simplification of real-world systems like MapReduce and Hadoop. The point here is not to perfectly model the actual systems, just like the point of the RAM model is not to be a perfect model of a physical computer. Our goal instead is to have a model which is simple enough to allow us to analyze our algorithms, and yet captures some of the algorithmic challenges of designing algorithms for parallel systems. We should note that there are other theoretical models of parallel computation, like the various flavors of the PRAM, and the BSP model of Valiant. The model above can be seen as a simplification of BSP.

4.2 Simple Algorithms

First a warm-up exercise.

Exercise 12. *Give a parallel algorithm which finds the maximum and the sum of n integers in a constant number of rounds when $s = m = \Theta(\sqrt{n})$. At the end, one specially designated machine must have the output written in its local memory.*

Now let us look at something a bit more interesting: the prefix sums problem. Our input is n pairs of integers, $(1, x_1), \ldots, (n, x_n)$, and our goal is to compute, for all $1 \leq i \leq n$, the sum $x_1 + \ldots, x_i$. We assume we have $s = m = \Theta(\sqrt{n})$. Notice that no machine has enough space for the entire output, so we will be satisfied having the different pieces of the output stored on different machines, as long as some prefix sum is stored by some machine.

A standard algorithm is as follows:

1. Exchange messages so that $(1, x_1), \ldots, (\sqrt{n}, x_{\sqrt{n}})$ are stored on the first machine, $(\sqrt{n} + 1, x_{\sqrt{n}+1}), \ldots, (2\sqrt{n}, x_{2\sqrt{n}})$ on the second machine, etc.
2. In the next round, the i-th machine computes $x_{(i-1)\sqrt{n}+1} + \ldots + x_j$ for all $(i-1)\sqrt{n} + 1 \leq j \leq i\sqrt{n}$. Moreover, the i-th machine sends to all machines numbered higher than itself the sum of all integers in its local memory.
3. In the next and final round, the local storage of the i-th machine contains the sums of the integers on machines $1, \ldots, i-1$. It can add up these sums to get $x_1 + \ldots + x_{(i-1)\sqrt{n}}$. The local storage of the i-th machine also contains the sums $x_{(i-1)\sqrt{n}+1} + \ldots + x_j$ for all $(i-1)\sqrt{n}+1 \leq j \leq i\sqrt{n}$ from the previous round. From these, the i-th machine can compute the prefix sums $x_1 + \ldots + x_j$ for all $(i-1)\sqrt{n} + 1 \leq j \leq i\sqrt{n}$.

An illustrative example is shown in Figure 1.

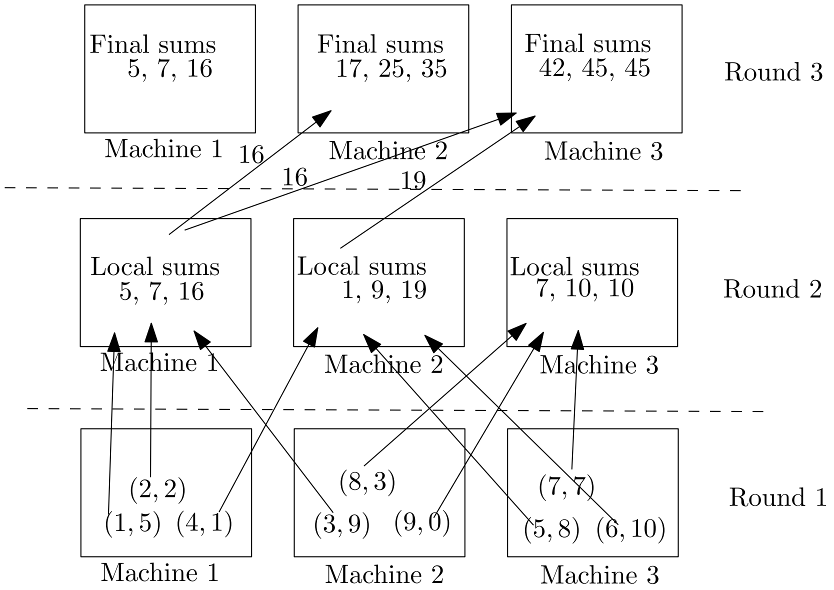

Figure 1: An example of the prefix sums algorithm.

Exercise 13. *Suppose the input is a weighted graph G on n nodes and $\Theta(n^2)$ edges. Assume that $s = \Theta(n^{3/2})$ and $m = \Theta(n)$. Initially, each machine has n edges of the graph in its local storage, where each edge is represented as a pair of vertices with a weight. Give a parallel algorithm that computes the minimum spanning tree of G in a constant number of rounds.*

4.3 Using Streaming Algorithms in Parallel

Many streaming algorithms can be turned into parallel algorithms without much effort. For example, suppose that our input is a sequence $\sigma = (\sigma_1, \ldots, \sigma_n)$ of n integers, which could possibly repeat. Each of our $m = \Theta(\sqrt{n})$ machines has $\Theta(\sqrt{n})$ integers from σ. We want to estimate the number of distinct integers in σ.

We show how to use the adaptive sampling algorithm DISTINCT to achieve this goal. First one machine samples a random hash function and sends it to all other machines: this way all machines use the same hash function. (**This can be done in two rounds: see how?**) Then, we run DISTINCT on each machine with parameter $k = \Theta(\frac{1}{\varepsilon^2})$, and once machine i has processed its input, it can send the set of at most k elements $S^{(i)}$ computed by DISTINCT on its local input, together with the final value $d^{(i)}$ of d used to compute $S^{(i)}$, to a specially designated machine, say the first one. (To make sure the notation is clear, we point out that $S^{(i)}$ is the set of integers on the i-th machine whose hash value given by h starts with $d^{(i)}$ 0's, and $d^{(i)}$ is chosen so that $|S^{(i)}| \leq k$.) As long as ε is a constant, the total number of words sent to the first machine is $O(\varepsilon^{-2}m) = O(\sqrt{n})$, and does not exceed the storage requirement.

It remains for the first machine to finish the computation. We claim that it can exactly simulate the DISTINCT algorithm when run on the entire sequence σ. Recall from the proof of Theorem 7 the notation D for the set of distinct integers in σ, and the sets $S_\ell = \{i : h(i) \in 0^\ell\{0,1\}^{L-l}\}$. Recall further that the final value of d computed by DISTINCT equals $\min\{\ell : |S_\ell| \leq k\}$, and the estimate output by the algorithm equals $2^d|S_d|$. Observe that for any $\ell \geq \max\{d^{(1)}, \ldots, d^{(m)}\}$, $S_\ell \subseteq S^{(1)} \cup \ldots \cup S^{(m)}$, and, moreover, that $d \geq \max\{d^{(1)}, \ldots, d^{(m)}\}$. So, the first machine can just go over the different possible values for d, from $\max\{d^{(1)}, \ldots, d^{(m)}\}$ to L, and check for each one whether $|S_d| \leq k$. The smallest d for which this is true gives us our desired output $2^d|S_d|$. Since we already proved in Theorem 7 that DISTINCT computed an estimate which is within a $(1 \pm \varepsilon)$ multiplicative factor from the true number of distinct elements, we now also have a parallel algorithm with the same guarantee.

Exercise 14. *Show how to use a constant number of additional rounds of computation to refine the approximation factor above from* $(1 \pm \varepsilon)$ *to* $(1 \pm \frac{C}{\sqrt{s}})$*, for a constant* C*.*

5 Impossibility Results

When designing algorithms it is useful to also be aware of impossibility results, which tell you what algorithmic problems are in fact solvable. You have seen some examples of this: there is no algorithm that takes an arbitrary program and input and decides if the program halts on that input; there is no comparison-based sorting algorithm that makes $o(n \log n)$ comparisons; there is no algorithm that computes the minimum vertex cover of an arbitrary graph, unless $\mathsf{P} = \mathsf{NP}$. Next we will discuss similar impossibility results in the streaming model. Unlike the results above, we will focus on *space lower bounds.* We will see that various assumptions we had to make above were necessary: if we drop them, then we need $\Omega(n)$ bits of space to solve our problem.

One frustrating aspect of the MAJORITY algorithm above is that we cannot be sure whether the element output by the algorithm actually is a majority element or not, unless we do a second pass over the stream. A natural question is whether there is a small space algorithm in the streaming model which can detect if a majority element exists in the stream. Our first impossibility result shows that this is impossible.

Theorem 9. *Any deterministic algorithm in the streaming model which decides whether a given stream σ has a majority element, i.e. an element i such that $f_i > \frac{m}{2}$, must use $\Omega(n)$ bits of space.*

The key to proving Theorem 9 is to think about a simple communication game. In this game, Alice has an object X from some large set $\mathcal{X}$. She wants to send a single message M to Bob, who has to use M to decode X. Alice and Bob can only communicate once: Alice sends M to Bob, and that's it. How short can the message M be, in the worst case?

More formally, Alice gets an input $X \in \mathcal{X}$ and sends to Bob a message $M(X) \in \{0,1\}^*$ (i.e. M is a function that maps $\mathcal{X}$ to strings of bits). Then Bob outputs $D(M)$, where D is some function that maps strings of bits to $\mathcal{X}$. It must be the case that $D(M(X)) = X$, otherwise Bob doesn't decode the message correctly.

Lemma 10. *In the game above, there must exist some $X \in \mathcal{X}$ such that $M(X)$ has length at least $\lceil \log_2 |\mathcal{X}| \rceil$ bits.*

Proof. Let $\mathcal{M} = \{M(X) : X \in \mathcal{X}\}$ be the set of all possible messages sent by Alice. Suppose towards contradiction that every message in $\mathcal{M}$ were of length at most $L < \lceil \log_2 |\mathcal{X}| \rceil$ bits. Then $|\mathcal{M}| \le 2^L < |\mathcal{X}|$, so, by the pigeonhole principle, there must exist two different $X, Y \in \mathcal{X}$ for which $M(X) = M(Y)$. This means that $X = D(M(X)) = D(M(Y)) = Y$, which is a contradiction. □

Note that Lemma 10 holds no matter what $\mathcal{X}$ is. All we use about $\mathcal{X}$ is its size.

We can now deduce Theorem 9 from Lemma 10 by showing that we could use a streaming algorithm to come up with M and D in the communication game above.

Proof of Theorem 9. Let $\mathcal{A}$ be a streaming algorithm that, on every stream σ uses at most s bits of memory in the worst case and decides if the stream has a majority element or not. We will use $\mathcal{A}$ to design M and D for the Alice and Bob communication game. We will take $\mathcal{X}$ to be the powerset of $[n]$, i.e. the set of all possible subsets of $[n]$. Notice that $|\mathcal{X}| = 2^n$. On input a subset X of $[n]$, Alice constructs a partial stream σ' whose updates consist of the elements of X listed in some arbitrary order. She feeds σ' to $\mathcal{A}$, and after $\mathcal{A}$ is done processing σ', she sends the contents of the memory of $\mathcal{A}$ to Bob together with the size of X: this is her message $M(X)$. Notice that the length of $M(X)$ is $s + \lfloor \log_2(n+1) \rfloor$: s bits to encode the memory, and $\lfloor \log_2(n+1) \rfloor$ bits to encode the size of X.

To decode the message, Bob constructs n streams $\sigma''_1, \dots, \sigma''_n$, where σ''_i consists of $|X|$ copies of i. Then, for each i, Bob restarts the execution of $\mathcal{A}$ with the memory contents he received from Alice, and feeds it σ''_i. At the end of the stream, $\mathcal{A}$ will be able to decide if the stream $\sigma = (\sigma', \sigma''_i)$ has a majority element. Notice that this happens if and only if $i \in X$. After doing this for every $i \in [n]$, Bob learns exactly the elements of X, as required by the communication game. An illustration of this construction is given in Figure 2.

By Lemma 10, we have that the size of the message sent by Alice must be at least n in the worst case. I.e. we have $s + \lfloor \log_2(n+1) \rfloor \ge n$, which implies $s = \Omega(n)$. □

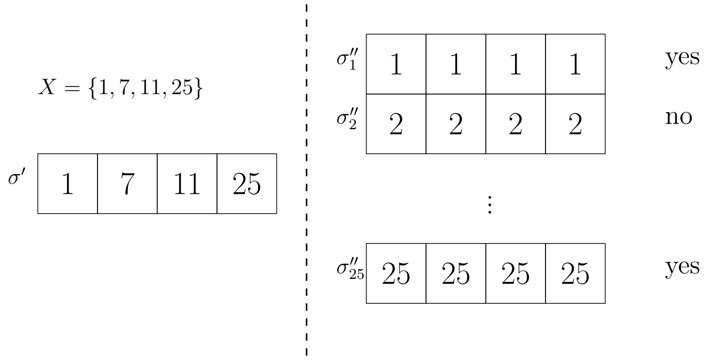

Figure 2: An illustration of the proof of Theorem 9. The "yes" and "no" labels on the far right indicate whether there is a majority element.

There is a subtlety in this proof: it is crucial that Bob can use the memory contents of $\mathcal{A}$ many times, by feeding different streams of updates to the algorithm. This is where we used the fact that $\mathcal{A}$ processes the updates in the stream one by one: because of this we can stop $\mathcal{A}$ at any time and use its memory to check what the algorithm would do if the stream is continued in different possible ways.

Exercise 15. *Use the same technique to show that an algorithm which outputs an element i if $f_i > m/2$, but does not output i if $f_i < m/4$, must use at least $\Omega(\log n)$ bits of memory.*

This technique can be used for many problems. Next we use it to show that a streaming algorithm for the distinct element count problem cannot be both deterministic and exact (and use small space).

Theorem 11. *Any deterministic algorithm in the streaming model which computes exactly the number of distinct elements F_0 in a given stream σ must use $\Omega(n)$ bits of space.*

Proof. Assume again that $\mathcal{A}$ is a streaming algorithm that, on every stream σ uses at most s bits of memory in the worst case and exactly computes F_0. The reduction we use is almost the same as in the proof of Theorem 9. Let again $\mathcal{X}$ be the powerset of $[n]$. Given a subset X of $[n]$, Alice constructs σ' and her message $M(X)$ in the exact same way as in the proof of Theorem 9. What is different is Bob's decoding procedure. Bob once again constructs streams $\sigma''_1, \dots, \sigma''_n$, but this time σ''_i consists of just a single copy of i. If $i \in X$, then $F_0 = |X|$; otherwise, $F_0 = |X| + 1$. So, by restarting the execution of $\mathcal{A}$ using the memory contents he received from Alice, and, for each i separately, feeding the additional update σ''_i to $\mathcal{A}$, and getting F_0 returned by the algorithm, Bob can decide, for each $i \in [n]$, whether $i \in X$. This allows Bob to reconstruct X.

By Lemma 10, we have, as before, that $s + \lfloor \log_2(n+1) \rfloor \geq n$, i.e. $s = \Omega(n)$. □

Our algorithm for estimating F_0 is both approximate and randomized. Is it possible to have a deterministic approximation algorithm for F_0? It turns out that it is not, but to show this, we will need an additional lemma.

Lemma 12. *There exists a collection $\mathcal{X}$ of subsets of $[n]$ such that $\log_2 |\mathcal{X}| = \Omega(n)$, and for any two distinct $X, Y \in \mathcal{X}$ we have $|Y \setminus X| \geq \frac{n}{8}$.*

We will omit the proof of this lemma here, because it goes beyond the scope of these lecture notes. (Just to be clear, there is nothing special about the number 8, and it can be replaced by any other number less than 4 at the cost of changing the hidden constant in the $\Omega()$ notation.) Using the lemma, however, we will prove the following theorem.

Theorem 13. *Any deterministic algorithm in the streaming model which, given a stream σ with F_0 distinct elements, computes a number $\hat{F}_0$ satisfying*

$$F_0 \leq \hat{F}_0 < \frac{9}{8} F_0$$

must use $\Omega(n)$ bits of space.

Proof. Assume that $\mathcal{A}$ is a streaming algorithm that satisfies the assumption of the theorem and uses at most s bits of memory in the worst case. For the reduction, this time we choose $\mathcal{X}$ to be the collection of subsets of $[n]$ from Lemma 12. Given $X \in \mathcal{X}$, Alice constructs her message $M(X)$ in the exact same way as in the proof of Theorem 11. Once again, the difference will be in how Bob decodes her message. For every $Y \in \mathcal{X}$, Bob creates a stream σ''_Y consisting of the elements of Y in some arbitrary order; then, he restarts the execution of $\mathcal{A}$ with the memory contents he received from Alice, and feeds σ''_Y to $\mathcal{A}$. The stream (σ', σ''_Y) has exactly $|X \cup Y| = |X| + |Y \setminus X|$ distinct elements. If $Y = X$, then $F_0 = |X \cup Y| = |X|$, so the value $\hat{F}_0$ returned by $\mathcal{A}$ satisfies $\hat{F}_0 < \frac{9}{8} F_0 = \frac{9}{8}|X|$. If $Y \neq X$, then $F_0 = |X \cup Y| \geq |X| + \frac{n}{8} \geq \frac{9}{8}|X|$, and, therefore, we have $\hat{F}_0 \geq F_0 \geq \frac{9}{8}|X|$. Therefore, Bob can decide whether $X = Y$ by comparing the approximation $\hat{F}_0$ returned by the algorithm to $\frac{9}{8}|X|$. By doing this for every $Y \in \mathcal{X}$, Bob can find X.

By Lemma 10, we have that $s + \lfloor \log_2(n+1) \rfloor \geq cn$, where c is a constant. This implies $s = \Omega(n)$. □