Space Efficient Algorithms for Distinct Elements in a Data Stream

Johnny Chen Naveen Sunkavally David Woodruff

Abstract

In a recent paper in RANDOM three algorithms were presented to (ϵ, δ)-approximate the number of distinct elements in a data stream with different space/time bounds. The paper also posed open problems about space lower bounds for this problem. In this paper we describe these and several older algorithms, provide proofs for the existing space lower bounds, and document our attempts at solving the open problems. In particular, drawing from several other papers, we suggest why common approaches do not seem to yield progress for this problem.

1 Introduction

Let $a = a_1, \ldots, a_n$ be a sequence of elements from a universe of size m, which we denote by $[m] = \{1, \ldots, m\}$. In this paper we will be interested in counting the number of distinct elements F_0 in a in the streaming model. In practice, m and n are often large and algorithms must execute quickly using little space. This restriction often makes exact and/or deterministic algorithms too inefficient, as it appears to be the case for the distinct elements problem. Therefore we consider randomized (ϵ, δ)-approximation algorithms.

1.1 Applications

Computing the number of distinct elements is very valuable to the database community. Certain query optimizers can use F_0 to find the number of unique values in a database possessing a certain attribute, without having to sort all the values. Internet routers may want to gather the number of distinct destination addresses passing through them, but can only see the data once and have very limited memory. In specific genome problems the length of the gene sequence is fixed and the number of distinct genes within that sequence is desired.

With commercial databases approaching the size of 100 terabytes, distinct element algorithms may need to use minimal space and time per element. It is infeasible to make multiple passes over the data since the sheer amount of time to look at the data is prohibitive. Unfortunately, for a stream with F_0 distinct elements, there is a provable lower bound of $\Omega(m)$ space on any deterministic algorithm which computes F_0 exactly. Even if we are content with a $1 \pm \epsilon$ approximation to F_0, any deterministic algorithm must still use $\Omega(m)$ space. It is not until we introduce randomization that things start to improve. We will present extremeley efficient randomized approximation algorithms for computing F_0.

1.2 Overview

This paper focuses on space issues. A recent paper [2] gives algorithms running in space $\tilde{O}(1/\epsilon^2 + \log m)$. The best lower bounds are $\Omega(\log m)$ [1] and $\Omega(1/\epsilon)$ [3]; the authors in [2] ask whether there is an algorithm matching these bounds or a new lower bound can be proved. In this paper we document our attempts at answering this question, presenting also the relevant algorithms from [2] and older literature, as well as the proofs of the lower bounds.

In section 2 we define the problem, fix notation and discuss some conventions. In section 3 we give a survey of algorithms for distinct elements in the one-pass stream model [1],[2],[10]. In particular, our presentation of the algorithms in [2] is more explicit and detailed than the original. Next we prove the two known lower bounds for randomized algorithms in the one-pass stream model. Indeed, there is a gap between the lower bound $1/\epsilon$ and the bound given in [2] of $1/\epsilon^2 + \log m$. The apparent weakness of the reduction suggests that $1/\epsilon$ may not be tight, and motivates the search for better algorithms. The rest of the paper illustrates our efforts. In section 5 we explain some attempts at one pass algorithms. The following section describes natural approaches like sampling and divide-and-conquer. Finally, we give a strategy for an algorithm using several passes.

2 Preliminaries

A stream is a sequence $a = a_1, \ldots, a_n$ of elements from a universe of size m, which we denote by $[m] = \{1, \ldots, m\}$. For $i \in [m]$, let m_i be the number of times i occurs. The *kth moment of a*, $F_k(a)$ is defined as

$$F_k(a) = \sum_{i \in [m]} m_i^k.$$

Of particular interest is F_0, the number of distinct elements in a.

All algorithms will be allowed exactly one pass from left to right over the input elements. Elements that have passed by that an algorithm has not stored are lost forever. Note that the algorithm cannot change the order in which items are streamed to it. The order of the elements and their distribution are assumed to be adversarial. We will use the log-cost RAM in our analysis. When we use $\tilde{O}()$ notation, we follow the convention in [2] of ignoring sublogarithmic factors and factors of $O(\log \frac{1}{\epsilon})$.

We first clarify the relationship between m and n. Most of our bounds will be in terms of $O(\log m)$ instead of $O(\log n)$, which is clearly advantageous in applications where the stream size n is larger than the universe m. If on the other hand $m > n$, we simply hash each value in the stream to $[n^c]$ for some constant $c > 2$ which gives high probability that there are no collisions. This takes $O(\log(m+n))$ time per element, and lets us assume $\log m = O(\log n)$ in what follows.

For any binary string x, we let $TRAIL(x)$ denote the number of rightmost zeros of x, e.g., $TRAIL(01000) = 3$ and $TRAIL(001010) = 0$. The set of distinct elements of stream a will be denoted $B = \{b_1, \ldots b_{F_0}\}$.

3 Known Algorithms

3.1 A Preliminary Algorithm Using Fully Random Hash Functions

We begin with an $O(\log m \log(\frac{\log m}{\delta}))$-space algorithm [10] for computing an approximation $\hat{F}_0$ of F_0 within a factor of 2, i.e. $\frac{1}{2}F_0 \leq \hat{F}_0 \leq 2F_0$. with probability of error less than δ. The algorithm illustrates the power of hashing for this problem, and the idea of looking at whether a certain hash bin stays empty or not is used extensively in [2]. Unfortunately, the space bound ignores the memory necessary to store the fully random hash functions used in the algorithm. [2] also encounters this problem but deals with it by using hash functions that are chosen to be just random enough for the (ϵ, δ) approximation to hold.

FR$(a_1, a_2, \ldots a_n, \delta)$

1. $\forall i, 1 \leq \log m$, let $t_i = 2^i$.
2. $\forall i, j,\ 1 \leq i \leq \log m$ and $1 \leq j \leq 200 \log(\frac{\log m}{\delta})$ let $h_{i,j} : [m] \to [t_i]$ represent a fully random hash function and initialize boolean variables $b_{i,j} \leftarrow 0$.
3. For each stream item a_k
   - $\forall i, j$, if $h_{i,j}(a_k) = 0$, set $b_{i,j} = 1$
4. for $i = 1$ to $\log m$
   - if more than $\frac{1}{5}$ of the values $b_{i,1} \ldots b_{i,200\log(\frac{\log m}{\delta})}$ are 0,

  - break, return $\hat{F}_0 = 2^i$ as best estimate for F_0.

5. return m as best estimate for F_0

Theorem 1 *Algorithm **FR** outputs a factor-2 approximation to F_0.*

Proof For a factor-2 approximation, it suffices to choose the best estimate $\hat{F}_0$ from among the set $S = \{2, 4, 8, 16 \ldots m\}$. For every $t_i \in S$, **FR** computes whether $F_0 \leq t_i$ or $F_0 \geq 2t_i$ with probability of error at most $\frac{\delta}{\log m}$. The computation takes place as follows: Consider any value $b_{i,j}$. After the stream has been processed, $b_{i,j} = 0$ only if none of the F_0 distinct values in the stream hash to bin 0 for the hash function $h_{i,j}$. For a fully random hash function, $P(b_{i,j} = 0) = (1 - \frac{1}{t_i})^{F_0}$. Now, if $F_0 \leq t_i$, then $P(b_{i,j} = 0) \geq (1 - \frac{1}{t_i})^{t_i} \geq .25$. If $F_0 \geq 2t_i$, then $P(b_{i,j} = 0) \leq (1 - \frac{1}{t_i})^{2t_i} \leq e^{-2} \approx .13534$. By running many different hashes, we can use these probabilities to distinguish upto any arbitrary level of precision. In particular, after running $200 \log(\frac{\log m}{\delta})$ hashes, we conclude that $F_0 \leq t_i$ if more than $\frac{1}{5}$ of the values are 0s; from Chernoff, it is easy to see that this many hashes is sufficient to determine whether $F_0 \leq t_i$ or $F_0 \geq 2t_i$ with probability of error less than $\frac{\delta}{\log m}$. We union bound over all possible values of t_i to get a total error probability less than δ. Assuming all decisions are correct for each t_i, we simply check one by one which t_i is closest to F_0 and output that value as our best esimate. ■

Ignoring the space required to store the description of the hash functions, the total space used equals $\tilde{O}(\log m \log(\frac{1}{\delta}))$. The time for processing each element is dominated by the time necessary to compute all the hash functions and equals $\tilde{O}(\log^2 m \log \frac{1}{\delta})$.

3.2 A Constant Factor Approximation Scheme

We now present an $O(\log m)$-space algorithm [1] for computing an estimate $\hat{F}_0$ to F_0 with $\frac{1}{c}F_0 \leq \hat{F}_0 \leq cF_0$ that errs with probability at most $\frac{2}{c}$, for any constant $c > 2$. This algorithm is better than **FR** in that it only makes use of linear hash functions which can be stored in $O(\log m)$ space. Recall that our input is a stream of up to m distinct elements, comprising $m \log m$ memory bits, which is exponentially larger than the space this algorithm uses. The algorithm will pick a random pairwise independent hash function $h : [m] \to [N]$ for some N (specified below) and apply it to each element a_i of the data stream. If there are few collisions, one would expect $\frac{N}{\min_{j=1}^{n} h(a_j)}$ to be a good approximation to F_0 because, intuitively, h is splitting up the discrete interval $[0, N]$ into $\frac{1}{F_0}$ equally sized buckets. The following algorithm/analysis formalizes this:

A1$(a_1, \ldots a_n)$

1. Choose a prime p between n and $2n$.
2. Choose $a \in \{1, \ldots, p-1\}$ and $b \in \{0, \ldots, p-1\}$ uniformly at random and define $h(x) = ax + b \bmod p$.
3. $R \leftarrow 0$
4. for $i = 1$ to n,
   - if $TRAIL(h(a_i)) > R$, set $R \leftarrow TRAIL(h(a_i))$
5. Output $Y = 2^R$

Theorem 2 *Algorithm **A1** outputs Y such that $\frac{1}{c}F_0 \leq Y \leq cF_0$ with probability at least $1 - \frac{2}{c}$.*

Proof Let r be an arbitrary nonnegative integer less than $\lceil \log p \rceil$. For each distinct element $x \in B$, let W_x be an indicator which is 1 iff $TRAIL(h(x)) \geq r$. Since $h(x)$ is uniformly distributed over $\mathbb{F}_p$, $\mathbf{E}[W_x] = \frac{1}{2^r}$ since every sequence of rightmost bits is (roughly) equally likely. Set $Z_r = \sum_x W_x$. By linearity of expectation, $\mathbf{E}[Z_r] = \sum_x \mathbf{E}[W_x] = \frac{F_0}{2^r}$. By Markov's inequality, if $2^r > cF_0$,

$$\mathbf{Pr}[Z_r > 0] = \mathbf{Pr}[Z_r > c\frac{F_0}{2^r}] = \mathbf{Pr}[Z_r > c\mathbf{E}[Z_r]] < \frac{1}{c}$$

By pairwise independence of h, $\mathbf{Var}[Z_r] = \sum_x \mathbf{Var}[W_x] = F_0 \cdot \frac{1}{2^r}(1-\frac{1}{2^r}) < \mathbf{E}[Z_r]$. By Chebyshev's inequality, if $c2^r < F_0$,

$$\mathbf{Pr}[Z_r = 0] = \mathbf{Pr}[\mathbf{E}[Z_r] - Z_r = \mathbf{E}[Z_r]] \leq \mathbf{Pr}[|\mathbf{E}[Z_r] - Z_r| \geq \mathbf{E}[Z_r]] \leq \frac{\mathbf{Var}[Z_r]}{(\mathbf{E}[Z_r])^2} < \frac{1}{\mathbf{E}[Z_r]} = \frac{2^r}{F_0} < \frac{1}{c}$$

Set r to be the value R computed by A. Now, A chooses R such that for all integers $s > R$, $\mathbf{Pr}[Z_s > 0] = 0$ since otherwise there would exist an x with $s > R$ rightmost bits set to 0, but R is maximal. Hence, by the union bound, the above two inequalities imply $\mathbf{Pr}[\frac{Y}{F_0} > c \text{ or } \frac{Y}{F_0} < \frac{1}{c}] < \frac{2}{c}$. ■

Theorem 3 *Algorithm **A1** uses $O(\log n)$ memory bits and takes $\tilde{O}(\log n)$ time per element.*

Proof We need $d = O(\log n)$ bits for p, a, b, and $O(\log \log n)$ bits to keep R, resulting in $O(\log n)$ space. For each element, we need $O(\log n \log \log n) = \tilde{O}(\log n)$ time to compute the multiplication in h and $O(\log n)$ time for the addition. Computing $TRAIL$ and updating R take $O(\log n)$ time. ■

3.3 Three Progressively Better (ϵ, δ)-Approximation Algorithms

We will run each of the algorithms that follow $O(\log \frac{1}{\delta})$ times in parallel and take the median of the results. For brevity, we have omitted this step from the pseudocode.

3.3.1 Algorithm A2

We have shown how to achieve an approximation to F_0 with error cF_0 in $O(\log m)$ space. We now wish to improve our estimate to a $1 \pm \epsilon$ factor with a $1 - \delta$ confidence, where ϵ, δ are user-specified parameters. In [2] three algorithms are presented to do this, each offering progressively better space. The first algorithm is a natural generalization of [1]. Instead of computing the minimum, we keep track of $t = \Theta(\frac{1}{\epsilon^2})$ smallest hash values in a balanced binary tree and output $\frac{tm^3}{v}$, where v is the t-th smallest distinct value. We expect this to work for the same reason the algorithm in [1] did. The algorithm follows:

A2$(a_1, \ldots, a_n, \epsilon)$

1. Pick a random pairwise independent hash function $h : [m] \to [m^3]$ (i.e., of the form $ax + b \bmod p$ for $m^3 \leq p \leq 2m^3$).
2. $t \leftarrow \lceil \frac{96}{\epsilon^2} \rceil$.
3. $BST \leftarrow$ new balanced binary tree.
4. for $i = 1$ to t,
    - insert(BST, $h(a_i)$)
5. for $i = t + 1$
    - if $h(a_i) < \max(BST)$,
        - remove(BST, $\max(BST)$)

– insert(BST, $h(a_i)$)

6. Output $Y = \frac{tm^3}{\max(BST)}$

Theorem 4 *Algorithm **A2** outputs Y with $(1-\epsilon)F_0 \leq Y \leq (1+\epsilon)F_0$ with probability at least $1-\delta$.*

Proof Assume $\frac{1}{m^3} < \frac{\epsilon t}{4F_0}$, which since $F_0 \leq m$ and $t \geq \frac{96}{\epsilon^2}$, is true provided $m \geq \sqrt{\frac{\epsilon}{24}}$:

Case 1: A2 outputs $Y > (1+\epsilon)F_0$.
There must exist at least t distinct elements that are all smaller than $\frac{tm^3}{F_0(1+\epsilon)} \leq \frac{tm^3(1-\frac{\epsilon}{2})}{F_0}$, as otherwise, if v is the tth smallest element, it would be at least $\frac{tm^3}{F_0(1+\epsilon)}$ so that $Y = \frac{tm^3}{v} \leq F_0(1+\epsilon)$. For any $b_i \in B$ (recall B is the set of distinct elements in the data stream),

$$\mathbf{Pr}[h(b_i) \leq \frac{(1-\frac{\epsilon}{2})tm^3}{F_0}] \leq \frac{(1-\frac{\epsilon}{2})t}{F_0} + \frac{1}{m^3} < \frac{(1-\frac{\epsilon}{2})t}{F_0} + \frac{\epsilon t}{4F_0} \leq \frac{(1-\frac{\epsilon}{4})t}{F_0}$$

where we used the fact that $h(b_i)$ is uniformly distributed and took into account rounding errors. Let $X_i, 1 \leq i \leq F_0$, be an indicator random variable which is 1 if and only if $h(b_i) < \frac{(1-\frac{\epsilon}{2})tm^3}{F_0}$. As argued, $\mathbf{E}[X_i] \leq \frac{(1-\frac{\epsilon}{4})t}{F_0}$. By linearity of expectation, letting $X = \sum_{i=1}^{F_0} X_i$ we get $\mathbf{E}[X] \leq (1-\frac{\epsilon}{4})t$. We compute the variance of X so that we can apply Chebyshev. By pairwise independence of h, the X_is are pairwise independent so that $\mathbf{Var}[X] = \sum_{i=1}^{F_0} \mathbf{Var}[X_i] \leq (1-\frac{\epsilon}{4})t$. Therefore by Chebyshev's inequality,

$$\mathbf{Pr}[X > t] = \mathbf{Pr}[X > \frac{\epsilon t}{4} + (1-\frac{\epsilon t}{4})] \leq \mathbf{Pr}[|X - \mathbf{E}[X]| \geq \frac{\epsilon t}{4}] \leq \frac{16\mathbf{Var}[X]}{\epsilon^2 t^2} \leq \frac{16(1-\frac{\epsilon}{4})t}{\epsilon^2 t^2} \leq \frac{16}{\epsilon^2 7} \leq \frac{16}{96} = \frac{1}{6}$$

Case 2: A2 outputs $Y < (1-\epsilon)F_0$.
There must be strictly less than t distinct elements that are smaller than $\frac{tm^3}{F_0(1-\epsilon)} \leq \frac{(1+\epsilon)tm^3}{F_0}$, as otherwise, if v is the tth smallest element, it would be at most $\frac{tm^3}{F_0(1-\epsilon)}$ so that $Y = \frac{tm^3}{v} \geq F_0(1-\epsilon)$. For any b_i,

$$\frac{(1+\frac{\epsilon}{2})t}{F_0} \leq \frac{(1+\epsilon)t}{F_0} - \frac{1}{m^3} \leq \mathbf{Pr}[h(b_i) \leq \frac{(1+\epsilon)tm^3}{F_0}] \leq \frac{(1+\epsilon)t}{F_0} + \frac{1}{m^3} \leq \frac{(1+\frac{3\epsilon}{2})t}{F_0}$$

where again we take into account rounding errors and use the fact that $\frac{1}{m^3} < \frac{\epsilon t}{4F_0}$. As in case 1, let $X_i, 1 \leq i \leq n$, be an indicator which is 1 iff $h(b_i) \leq \frac{(1+\epsilon)tm^3}{F_0}$ and define $X = \sum_{i=1}^{n} X_i$. We have that $\mathbf{E}[X] \geq t(1+\frac{\epsilon}{2})$ from the bounds above, and also $\mathbf{Var}[X] = \sum_{i=1}^{n} \mathbf{Var}[X_i] \leq \sum_{i=1}^{n} \mathbf{E}[X_i] \leq t(1+\frac{3\epsilon}{2})$. Rewriting,

$$\mathbf{Pr}[X < t] = \mathbf{Pr}[-X > -t] = \mathbf{Pr}[-X+\mathbf{E}[X] > -t+\mathbf{E}[X]] \leq \mathbf{Pr}[-X+\mathbf{E}[X] \geq -t+(t+\frac{t\epsilon}{2})] \leq \mathbf{Pr}[|X-\mathbf{E}[X]| \geq \frac{t\epsilon}{2}]$$

Therefore by Chebyshev's inequality,

$$\mathbf{Pr}[X < t] \leq \mathbf{Pr}[|X - \mathbf{E}[X]| \geq \frac{t\epsilon}{2}] \leq \frac{4\mathbf{Var}[X]}{\epsilon^2 t^2} \leq \frac{4t(1+\frac{3\epsilon}{2})}{\epsilon^2 t^2} \leq \frac{12}{\epsilon^2 t} < \frac{1}{6}$$

Hence, by the union bound, the probability that either case occurs is at most $\frac{1}{3}$. Since h has range $[m^3]$, the probability that it is not injective is at most $\frac{1}{m}$, so that **A1** succeeds with probability at least $\frac{2}{3} - \frac{1}{m}$. This probability can be amplified in the standard way to $1-\delta$ by running $O(\log \frac{1}{\delta})$ copies in parallel and taking the median of the results. ■

Theorem 5 *Algorithm **A2** uses $O(\frac{1}{\epsilon^2}\log n \log\frac{1}{\delta})$ memory bits with worst-case $\tilde{O}(\log m)$ time per element.*

Proof The size of BST is at most $t = \theta(\frac{1}{\epsilon^2})\log m$, and the hash function results in an additive $\log m$ to the space. Running $O(\log\frac{1}{\delta})$ parallel copies gives the overall space bound. Since BST is a balanced binary tree, the time per element is $O(\log\frac{1}{\epsilon}\log m)$ since all operations on BST take $\log|BST|$, where $|BST|$ is the number of elements stored in BST. ■

We have achieved space $O(\frac{1}{\epsilon^2}\log n\log\frac{1}{\delta})$. Later we will show space lower bounds of $\Omega(\frac{1}{\epsilon})$ and $\Omega(\log n)$. It is not known if $\Omega(\frac{1}{\epsilon^2})$ is a lower bound. Hence, it is possible that this algorithm has space which is the product of two lower bounds for this problem. Can we do better? Assuming $\Omega(\frac{1}{\epsilon^2})$ is a lower bound, the best possible algorithm would take $O((\frac{1}{\epsilon^2}+\log m)\log\frac{1}{\delta})$ space. Notice that $\frac{1}{\epsilon^2}$ and $\log m$ are added together rather than multiplied.

3.3.2 Algorithm A3

The second algorithm in [2] achieves this space bound modulo logarithmic factors. Rather than estimate F_0 directly, it instead estimates a related quantity which is easier to compute in the stream model. Specifically, we know that for completely random hash functions $h : [m] \to [R]$, $r = \mathbf{Pr}_h[h^{-1}(0)\cap B \neq \emptyset] = 1-(1-\frac{1}{R})^{F_0}$. What's crucial is that we've found an event which algebraically depends on F_0 that we can approximate without knowing F_0. Taking logarithms one can solve for an approximation to F_0 given an approximation to r. Since $R = cF_0$ for some small integer constant, $(1-\frac{1}{R})^{F_0} \approx \frac{1}{e}^{\frac{1}{c}}$, so that r is a constant in expectation, which is necessary for a good approximation of this event to translate to a good approximation of F_0, as we will prove below.

It turns out that r will indeed be approximable in the stream model. The reader may wonder why we choose to approximate r instead of $1-r$. The reason is purely formulaic; the techniques below for approximating r do not work for $1-r$. It remains open whether there is an analysis for $1-r$. The algorithm, explanation, and analysis follow:

A3$(a_1, \ldots, a_n, \epsilon, \delta)$

1. In parallel run the following two subroutines:
   - $R \leftarrow 2 * 25 * \mathbf{A1}(a_1, \ldots, a_n, \epsilon)$
   - (a) $\gamma \leftarrow \min(\frac{1}{e}-\frac{1}{3}, \frac{\epsilon}{6c})$
     (b) $t \leftarrow \left\lceil \frac{\log(\frac{2}{\gamma})}{\log 5} \right\rceil$
     (c) $k \leftarrow \frac{18000}{\epsilon^2}$
     (d) Choose a prime p_1 between m^{t+1} and $2m^{t+1}$.
     (e) Let $d \leftarrow \lceil \log m \rceil$.
     (f) Let $q(x)$ be an irreducible polynomial of degree d over $\mathbb{F}_2$.
     (g) Choose $a \in \{1, p_1-1\}$ and $b \in \{0, p_1-1\}$ uniformly at random.
     (h) Define the master hash function $h : [k] \to [p_1]$ as $h(x) = ax + b \bmod p_1$.
     (i) Initialize $y_j = 0$ for all $1 \leq j \leq k$.
     (j) for $i = 1$ to n,
        i. for $j = 1$ to k,
           A. $z \leftarrow h(j)$. Restrict z to its last $(t+1)\log m$ bits and chop it up into $t+1$ bit strings each of size $\log m$ bits. Let the successive bit strings be denoted z_i, $0 \leq i \leq t$. Define $w_j : [k] \to F_2^d$ to be $z_t x^t + z_{t-1}x^{t-1} + \ldots z_1 a_1 + z_0$ (arithmetic in F_2^d is done modulo the polynomial q).

B. If $TRAIL(w_j(a_i)) > y_j$, $y_j \leftarrow TRAIL(w_j(a_i))$.

2. $X(H_R) \leftarrow 0$
3. for $i = 1$ to k,
   (a) if $y_i \geq R$, $X(H_R) = X(H_R) + 1$.
4. $X(H_R) = \frac{X(H_R)}{k}$.
5. Output $Y = \frac{\ln(1-X(H_R))}{\ln(1-\frac{1}{R})}$

This algorithm constructs k t-wise independent hash functions and finds the average number of them $X(H_R)$ which map at least one element of the stream to 0. By appropriate choice of k and t, this gives a close approximation to the probability that a completely random hash function $v : [m] \rightarrow [R]$ maps at least one item to 0, which as we will show below, gives a good approximation to F_0. Unfortunately, we do not know R until we have read the entire stream. To combat this, for each hash function w_j, the above algorithm computes $y_j = \max_{a_i} TRAIL(w_j(a_i))$. Using the fact that the restriction of w_j to its last l bits is also a hash function, when we eventually learn R we need only compare the maximum index y_j to $\log R$ in order to learn if the restriction of y_j to its last $\log R$ bits maps at least one element of the stream to 0. Note that the restriction of a hash function to its last l bits is not generally a hash function. However, if its range is is a power of 2, as in our case, this result holds. Also note that **A1** returns R which is a power of 2.

In this algorithm we have a so-called master hash function h. When we read each stream element, we use the master hash function h to get k t-wise independent hash functions which are each polynomials with coefficents based on the bits returned by h. We then evaluate each of these hash functions on the current stream element and update the largest number of bits y_j from the right which are 0, if necessary. Since we need to evaluate each of k hash functions on each a_i, and since extracting each function from h is $O(\log m)$ (in the log-cost RAM) time, we do not lose any time by not storing each hash function explicitly. However, we replace a potential $O(\frac{1}{\epsilon^2}\log m)$ term in the space count with an $\tilde{O}(\log m)$ term. The following lemma says that approximating r is a good way to approximate F_0.

Lemma 6 *Let $c > 2$ be a constant and let $\epsilon > 0$. Suppose R and F_0 are such that $\frac{1}{2c} \leq \frac{F_0}{R} \leq \frac{1}{2}$. Then if $|r - \tilde{r}| \leq \gamma = \min(\frac{1}{e} - \frac{1}{3}, \frac{\epsilon}{6c})$, then $|F_0 - Y| \leq \epsilon F_0$.*

Proof We prove this using some well-known bounds and a bit of calculus. We have:

$$R \geq 2F_0 \rightarrow R \geq 2 \rightarrow \frac{1}{R} \leq \frac{1}{2} \rightarrow 1 - \frac{1}{R} \geq e^{\frac{-2}{R}} \rightarrow r = 1 - (1 - \frac{1}{R})^{F_0} \leq 1 - e^{\frac{-2F_0}{R}} \leq 1 - \frac{1}{e}$$

We also need: $\gamma \leq \frac{1}{e - \frac{1}{3}} \rightarrow r + \gamma < \frac{2}{3} \rightarrow \frac{1}{1-(r+\gamma)} < 3$ and $\frac{-1}{\ln(1-\frac{1}{R})} \leq R$. The calculus we use is that for any continuous function f, $|f(x) - f(\tilde{x})| \leq \epsilon|\sup_{y\in(x,\tilde{x})} f'(y)|$ for $\tilde{x}$ close to x. Specifically, for $f(x) = \ln(1-x)$, this yields $|f(x) - f(\tilde{x})| \leq \frac{|x-\tilde{x}|}{1-\max(x,\tilde{x})}$. The proof follows from the same line in [2]:

$$|F_0 - Y| = \frac{|\ln(1-r) - \ln(1-\tilde{r})|}{-\ln(1-\frac{1}{R})} \leq \frac{R|r-\tilde{r}|}{1-(r+\gamma)} \leq 3R\gamma \leq \frac{6cF_0\epsilon}{6c} = \epsilon F_0$$

■

Theorem 7 *Algorithm **A3** outputs Y with $(1-\epsilon)F_0 \leq Y \leq (1+\epsilon)F_0$ with probability at least $1-\delta$.*

Proof From the above lemma, we see that we have reduced the problem to one of approximating r. Now, r is defined in terms of completely random hash functions, which are not known to be efficiently constructible. Instead **A3** chooses our hash functions from a family of hash functions $\mathcal{H}$ which are t-wise independent. Define $p = \mathbf{Pr}_{h\in\mathcal{H}}[h^{-1}(0) \cap B \neq \emptyset]$. Let $\mathcal{H}_i \subseteq \mathcal{H}$ be the subset of hash functions of $\mathcal{H}$ that map $b_i \in B$ to 0. Then $p = |\frac{\cup_{i=1}^{n}\mathcal{H}_i}{\mathcal{H}}|$ since p counts the number of hash functions which map some element to 0, divided by the total number of hash functions. We expand p using the principle of inclusion-exclusion, and bound $|\frac{\cup_{i=1}^{n}\mathcal{H}_i}{\mathcal{H}}|$ between successive terms of the expansion. Since the terms alternate signs, for odd t we have:

$$\sum_{i=1}^{t-1}(-1)^{l+1}(\sum_{i_1<\ldots i_l}\mathbf{Pr}_{h\in\mathcal{H}}[h \in \mathcal{H}_{i_1}\cap\ldots\cap\mathcal{H}_{i_l}]) \leq p \leq \sum_{i=1}^{t}(-1)^{l+1}(\sum_{i_1<\ldots i_l}\mathbf{Pr}_{h\in\mathcal{H}}[h \in \mathcal{H}_{i_1}\cap\ldots\cap\mathcal{H}_{i_l}])$$

Since we have t-wise indepndence, $\mathbf{Pr}_{h\in\mathcal{H}}[h \in \mathcal{H}_{i_1}\cap\ldots\cap\mathcal{H}_{i_l}] = \binom{F_0}{l}R^{-l}$ since the probability ranges over $\binom{F_0}{l}$ subsets, and h has probability R^{-1} of being in any one of them (i.e., of mapping a particular element to 0), so by t-wise independence, it has probability R^{-l} of being in l all of them, for $l \leq t$. One can also expand the expression $r = 1-(1-\frac{1}{R})^{F_0}$ using binomial expansion and bound it between the same terms:

$$\sum_{l=1}^{t-1}(-1)^{l+1}\binom{F_0}{l}R^{-l} \leq r = 1-(1-\frac{1}{R})^{F_0} = \sum_{l=1}^{F_0}(-1)^{l+1}\binom{F_0}{l}R^{-l} \leq \sum_{l=1}^{t}(-1)^{l+1}\binom{F_0}{l}R^{-l}$$

If t is large enough, the difference between the two terms bounding r and p can be made arbitrarily small, implying that r can be made arbitrarily close to p (since they are sandwiched between the same terms). **A3** chooses $t = \left\lceil\frac{\log(\frac{2}{\gamma})}{\log 5}\right\rceil$, which makes $|p-r| \leq \frac{\gamma}{2}$, where $\gamma = \min(\frac{1}{e}-\frac{1}{3}, \frac{\epsilon}{6c})$. **A3** then approximates p by $X(H_R) = \frac{1}{k}|\{j|h_j^{-1}(0)\cap B \neq \emptyset\}|$. k is chosen to be suitably large so that by the law of large numbers, the average computed in $X(H_R)$ will be close to its expectation, which is just p. For $k = \frac{18000}{\epsilon^2}$, Chebyshev's inequality gives $\mathbf{Pr}[|X(H_R)-p| > \frac{\gamma}{2}] \leq \frac{1}{20}$. Now, setting $c=5$ in **A1** gives an error probability of $\frac{2}{5}$, since, $\frac{1}{c}F_0 \leq R' \leq cF_0 \rightarrow 2c\frac{1}{c}F_0 \leq 2cR' \leq 2c^2F_0 \rightarrow 2F_0 \leq R \leq 2c^2F_0$ with probability at least $1-\frac{2}{5}$, where we let R' denote what is returned by **A1**. Adding this to $\frac{1}{20}$ gives the total error probability, which is $\frac{9}{20}$. Hence, with constant probability, **A3** is correct, so that repeating it $\log\frac{1}{\delta}$ times and taking the median of the outputs gives an (ϵ, δ) approximation scheme. ■

Theorem 8 *Algorithm **A3** uses $\tilde{O}((\frac{1}{\epsilon^2}+\log n)\log\frac{1}{\delta})$ space and worst-case $\tilde{O}(\frac{1}{\epsilon^2}\log m)$ time per element.*

Proof **A1** takes $O(\log m)$ space. The master hash function takes $O(t\log m) = \tilde{O}(\log m)$ space. Storing y_j for $O(\frac{1}{\epsilon^2})$ values takes $O(\frac{1}{\epsilon^2}\log\log m) = \tilde{O}(\frac{1}{\epsilon^2})$ space. Finally, running $O(\log\frac{1}{\delta})$ copies in parallel yields the overall space bound. The time in **A3** is dominated by accessing the master hash function $O(\frac{1}{\epsilon^2})$ times for each stream element, yielding an overall $O(\frac{1}{\epsilon^2}\log m)$ time per elment. ■

3.3.3 Algorithm A4

Although the space has dramatically improved, the time per element has increased by a multiplicative factor of $\frac{1}{\epsilon^2}$. The third algorithm in [2] drops this back down to $\tilde{O}(\log m)$, at least in the amortized sense.

In a sense this algorithm is a generalization of algorithm **A3**. The algorithm chooses a pairwise independent hash function $h : [m] \rightarrow [m]$, and finds the minimal restriction (defined as in the previous algorithm) h_t for which $|h_t^{-1}\cap B| \leq \frac{576}{\epsilon^2}$. Here h_t refers to h restricted to its last t coordinates, which is also a hash function (since we assume m is a power of 2 in this algorithm). Intuitively, we are restricting the space to size 2^t, and then only considering elements which are in one bucket in that space, namely, the elements that hash to 0. We scale back up to the original space by multiplying by 2^t. The algorithm follows:

$\mathbf{A4}(a_1, \ldots, a_n, \epsilon, \delta)$

1. $T \leftarrow$ new array of size $m+1$.
2. Initialize $T(i)$ to a new balanced binary search tree for $0 \leq i \leq m$.
3. Let $d = \log m$, where we assume m is a power of 2. Choose an irreducible polynomial $q(x)$ of degree d over $\mathbb{F}_2$. We will work in $\mathbb{F}_2^d$, i.e., the field with reductions modulo $q(x)$.
4. Choose $a \in \mathbb{F}_2^d - 0$, $b \in \mathbb{F}_2^d$ uniformly at random. Define $h : [m] \to \mathbb{F}_2^d$ as $h(x) = ax + b \bmod q$.
5. Find a prime p between $\lceil 3\left(\frac{(\log m+1)576}{\epsilon^2}\right)^2 \rceil$ and $2\lceil 3\left(\frac{(\log m+1)576}{\epsilon^2}\right)^2 \rceil$.
6. Choose $a \in \{1, \ldots, p-1\}$, $b \in \{0, \ldots, p-1\}$ uniformly at random and define $g(x) = ax + b \bmod p$.
7. $t \leftarrow 0$.
8. for $i = 1$ to n,
   (a) if $TRAIL(h(a_i)) \geq t$, insert$(T(TRAIL(h(a_i))), g(a_i))$.
   (b) if $|T| > \frac{576}{\epsilon^2}$,
      i. $T_i \leftarrow$ new balanced binary search tree for all i, $0 \leq i \leq t$ (garbage collect old values).
      ii. $t \leftarrow t + 1$.
9. Output $Y = |T| \cdot 2^t$.

There are a couple of cute tricks in the above algorithm. When we get a buffer overflow (i.e., the size of T becomes too large) and need to increment t, we need to find the new set of distinct elements with $t+1$ rightmost 0s. Instead of rescanning the entire stream, we observe that $h_{t+1}^{-1}(0) \subseteq h_t^{-1}(0)$ so that we simply empty the tth entry of T. Also, the idea of using a second hash function g to store the description of distinct elements in T improves the space dramatically (from a $\frac{1}{\epsilon^2} \log m$ term to a $\frac{1}{\epsilon^2} \log\log m$ term). We argue correctness and efficiency:

Theorem 9 *Algorithm* ***A4*** *outputs* Y *with* $(1-\epsilon)F_0 \leq Y \leq (1+\epsilon)F_0$ *with probability at least* $1-\delta$.

Proof We note that there are two sources of error. One comes from g having collisions. The first observation is that the above algorithm applies g on at most $(\log m + 1)\frac{c}{\epsilon^2}$ distinct elements, since t can be incremented at most $\log m$ times, and in the worst case the entire buffer is flushed each time. However, the range of g is $[3((\log m + 1)\frac{c}{\epsilon^2})^2]$ so the expected number of collisions is less than $\frac{(\log m+1)^2 c^2}{2\epsilon^4} \cdot \frac{\epsilon^4}{3c^2(\log m+1)^2} = \frac{1}{6}$. We will assume there are no collisions in the remainder, and add $\frac{1}{6}$ to the total error probability (union bound). Computing the remaining probability of error is an exercise in pairwise independence and Chebyshev's inequality, as in the previous algorithms. See [2] for the details. Letting t' be the value of t that **A4** terminates with, we obtain $\mathbf{Pr}[|X_{t'} \cdot 2^{t'} - F_0| > \epsilon F_0] = \frac{1}{6}$. Hence, the overall algorithm succeeds with probability at least $1 - (\frac{1}{6} + \frac{1}{6}) = \frac{2}{3}$. Running $O(\log \frac{1}{\delta})$ copies in parallel and taking the median gives the desired (ϵ, δ) approximation scheme. ■

Theorem 10 *Algorithm* ***A4*** *uses* $\tilde{O}((\frac{1}{\epsilon^2} + \log n)\log\frac{1}{\delta})$ *space and amortized* $\tilde{O}(\log m \log \frac{1}{\delta})$ *time per element.*

Proof Hash functions h and g collectively consume $\tilde{O}(\log m)$ space. The array T has $\log m$ indices so it consumes at least $O(\log m)$ space, and at any given time it contains at most $\frac{1}{\epsilon^2}$ elements, each of which is an index into a bit position of $h(a_i)$, so it requires at most $\frac{1}{\epsilon^2} \cdot \log\log m$ bits. Hence, in total, the space is

$\tilde{O}((\frac{1}{\epsilon^2} + \log n) \log \frac{1}{\delta})$ since we run $\log \frac{1}{\delta}$ copies in parallel. To see the time bound, note that every element in the stream is inserted at most once and removed at most once from the buffer, each of which takes $O(\log m)$ time. So even though it takes $O(\log m + \frac{1}{\epsilon^2})$ time to empty a buffer, if we charge the removal of an item to when it was inserted, we have an amortized $O(\log m)$ time per stream element. Since we do this $\log \frac{1}{\delta}$ times in parallel, the total amortized time per element is $O(\log m \log \frac{1}{\delta})$. ■

This algorithm is the best known (in terms of space and time) for solving the distinct elements problem in the stream model in one pass.

4 Lower bounds

In this section we give the proofs of two lower bounds for F_0 in a one-pass streaming model. These proofs rely on reductions from problems in a communication complexity framework. It is notable that the reductions are not particularly powerful, suggesting that tighter bounds may hold.

Definition 11 *Fix the alphabet and stream size m, n, respectively. Define $DS_{\epsilon,\delta}(F_0)$ to be the amount of space used by an optimal (space) randomized algorithm that (ϵ, δ)-approximates F_0 for streams of length n over an alphabet of size m in one pass.*

There are two known space lower bounds for $DS_{\epsilon,\delta}(F_0)$: $\Omega(\log m)$, given in [1], and $\Omega(1/\epsilon^2)$, a recent result given in [3]. We present the proofs of both bounds in this section. Central to the proofs is the idea of communication complexity. Let $f : \mathcal{X} \times \mathcal{Y} \to \{0, 1\}$ be a Boolean function. We will consider two parties, Alice and Bob, receiving x and y respectively, who try to compute $f(x, y)$. In the protocols we consider, Alice computes some function $A(x)$ of x and sends the result to Bob. Bob then attempts to compute $f(x, y)$ from $A(x)$ and y. Note that only one message is sent, and it can only be from Alice to Bob.

Definition 12 *For each randomized protocol Π as described above for computing f, the communication cost of Π is the expected length of the longest message sent from Alice to Bob over all inputs. The **δ-error randomized communication complexity** of f, $R_\delta(f)$, is the communication cost of the optimal protocol computing f with error probability δ (that is, $\Pr[\Pi(x, y) \neq f(x, y)] \leq \delta$).*

For deterministic protocols with input distribution μ, define $D_{\mu,\delta}(f)$, the **δ-error μ-distributional communication complexity** of f, to be the communication cost of an optimal such protocol. Using the Yao Minimax Principle, $R_\delta(f)$ is bounded from below by $D_{\mu,\delta}$ for any μ [9]. The following is a famous result of communication complexity.

Theorem 13 *Let $EQ : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ such that $EQ(x, y) = 1$ iff $x = y$. Then $R_\delta(EQ) = \Theta(\log n)$.*

4.1 The $\Omega(\log m)$ Lower Bound

The idea is that EQ can be reduced to computing F_0. The lower bound then follows from Theorem 13. Let S be a family of $t = 2^{\Omega(n)}$ size $m/4$ subsets of $[m]$, such that $|x \cap y| \leq m/8$ for every distinct $x, y \in S$. The existence of S for sufficiently large n follows from the Probabilistic Method. Let $\mathcal{A}$ be a randomized (ϵ, δ)-approximation algorithm for streams over $[m]$. A protocol for EQ : $S \times S \to \{0, 1\}$ works as follows: on input (x, y), Alice computes $\mathcal{A}(x)$ and sends the state of $\mathcal{A}$ to Bob. Bob continues the computation from this point on y with $\mathcal{A}$. He returns 1 if the $\mathcal{A}$ returns a value greater than $(1 - \epsilon)3n/8$.

If $x = y$ then $F_0(x, y) = n/4$. If $x \neq y$ then $F_0(x, y) \geq 3n/8$. Therefore the protocol is correct. Now suppose that the (randomized) communication complexity of the protocol is less than $\log t$. By Pigeonhole, there exists $x \neq y$ such that $\mathcal{A}(x)$ is at the same state as $\mathcal{A}(y)$. Therefore Bob will see the same answer for $\mathcal{A}(x, x)$ and $\mathcal{A}(y, x)$. But this is impossible since $x \neq y$. Therefore we have:

Theorem 14 *For $\epsilon < .1$, $DS_{\epsilon,\delta}(F_0) = \Omega(\log m)$.*

4.2 The $\Omega(1/\epsilon)$ Lower Bound

The key result presented in this section is the following:

Theorem 15 (Bar-Yossef) *For every n, m, $0 < \delta < 1$, and $1/n \leq \epsilon \leq 1/6$,*

$$DS_{\epsilon,\delta}(F_0) = \Omega\left(\frac{1}{\epsilon} \cdot (1 - H_2(\delta))\right).$$

The proof involves a reduction from the (one-way) communication complexity of a set-disjointness problem. Before giving the proof, we must define VC dimension [8].

Let $\mathcal{H} = \{h : \mathcal{X} \to \{0,1\}\}$ be a family of Boolean functions on a domain $\mathcal{X}$. Each $h \in \mathcal{H}$ can be viewed as a $|\mathcal{X}|$-bit string $h_1 \dots h_{|\mathcal{X}|}$.

Definition 16 *For a subset $\mathcal{S} \subseteq \mathcal{X}$, the* ***shatter coefficient*** *of $\mathcal{S}$ is given by $|\{h|_{\mathcal{S}}\}_{h \in \mathcal{H}}|$, the number of distinct bitstrings obtained by restricting $\mathcal{H}$ to $\mathcal{S}$. If the shatter coefficient of $\mathcal{S}$ is $2^{|\mathcal{S}|}$, then $\mathcal{S}$ is* ***shattered*** *by $\mathcal{H}$. The* ***VC dimension*** *of $\mathcal{H}$, VCD($\mathcal{H}$), is the size of the largest subset $\mathcal{S} \subseteq \mathcal{X}$ shattered by $\mathcal{H}$.*

Let $f : \mathcal{X} \times \mathcal{Y} \to \{0,1\}$ be a Boolean function. The family of functions $f_{\mathcal{X}} : \mathcal{Y} \to \{0,1\}$ is defined by $f_x(y) = f(x,y)$ for all $x \in \mathcal{X}, y \in \mathcal{Y}$. The following Theorem lower bounds the (one-way) communcation complexity of f in information theory terms.

The proof of Theorem 15 uses a reduction from the following problem.

Definition 17 *Fix n and let $1/n \leq \epsilon \leq 1/2$. Let $T \subseteq \mathcal{X} \times \mathcal{Y}$, be where $\mathcal{X} = \{x \subseteq [n] : |x| = n/2\}$, $\mathcal{Y} = \{y \subseteq [n] : |y| = \epsilon n\}$ and if $(x,y) \in T$ then either $x \cap y = \emptyset$ or $y \subseteq x$. The* **ϵ-set-disjointness problem** *is defined as $DISJ_{n,\epsilon} : T \to \{0,1\}$, where $DISJ_{n,\epsilon}(x,y) = 1$ iff $y \subseteq x$.*

In other words, given the promise that x has size $n/2$, y has size ϵn and either they are disjoint or $y \subseteq x$, DISJ(x,y) indicates if $y \subseteq x$. We will use the randomized communication complexity of the ϵ-set-disjointness problem. We first find the VC Dimension of $f_{\mathcal{X}}$, where $f = \mathrm{DISJ}_{n,\epsilon}$.

Lemma 18 *Let $f = DISJ_{n,\epsilon}$, where for convenience we assume $n\epsilon$ divides $n/2$. Then $VCD(f_{\mathcal{X}}) = \frac{1}{2\epsilon}$.*

Proof Let $\ell = 1/2\epsilon$. First we argue that $VCD(f_{\mathcal{X}}) \leq \ell$. Suppose not; then there is a subset $\mathcal{S}$ with $\ell + 1$ elements shattered by $f_{\mathcal{X}}$. Therefore for some $x \in \mathcal{X}$, $f(x,y) = \mathrm{DISJ}(x,y) = 1$ for all $y \in \mathcal{S}$, implying $y \subseteq x$ for all $y \in \mathcal{S}$. Since $|x| = n/2$, each y has ϵn elements, and there are $\ell + 1$ such y, by the Pigeonhole Principle there are two elements $y_1, y_2 \in \mathcal{S}$ which have nonempty intersection. But it is impossible to split y_1 and y_2; that is, there is no x such that $y_1 \subseteq x$ and $y_2 \cap x = \emptyset$ or vice versa. This implies $\mathcal{S}$ is not shattered by $f_{\mathcal{X}}$, a contradiction. Therefore $VCD(f_{\mathcal{X}}) \leq \ell$.

Next we demonstrate a subset $\mathcal{S} \subseteq \mathcal{Y}$ of size ℓ that is shattered by $f_{\mathcal{X}}$. Recall what this means: for every subcollection $\mathcal{S}'$ of $\mathcal{S}$ there is some $x \in \mathcal{X}$ such that $y \subseteq x$ for $y \in \mathcal{S}'$ and $y \cap x = \emptyset$ for $y \in \mathcal{S} - \mathcal{S}'$. To this end, let $y_1, \dots, y_\ell$ be disjoint, equally sized subsets contained in $[n/2]$. Since $\ell \cdot \epsilon n = n/2$ (and we assume ϵn divides $n/2$) this is possible. Now for each $\ell-$bit sequence $b = b_1 \dots b_\ell$, define x_b so that $y_i \subseteq x_b$ iff $b_i = 1$; fill each x_b with elements from $\{n/2+1, \dots, n\}$ arbitrarily so that $|x_b| = n/2$ for all b. Since the y_i are disjoint, for every subcollection $\mathcal{S}' \subseteq \mathcal{S}$ there is a x_b such that $y \subseteq x$ for $y \in \mathcal{S}'$ and $y \cap x = \emptyset$ for $y \in \mathcal{S} - \mathcal{S}'$. Therefore $\{f_{x_b}\}$ shatters $\mathcal{S}$, so $VCD(f_{\mathcal{X}}) \geq \ell$.

It follows that $VCD(f_{\mathcal{X}}) = \ell$. ■

The following Theorem lower bounds the (one-way) communcation complexity of f in terms of information theory.

Theorem 19 *For every $f : \mathcal{X} \times \mathcal{Y} \to \{0,1\}$ and every $0 < \delta < 1$, there exists a distribution μ on $\mathcal{X} \times \mathcal{Y}$ such that*

$$D_{\mu,\delta}(f) \geq \ell(1 - H_2(\delta)),$$

where $\ell = VCD(f_{\mathcal{X}})$.

Proof Let $\mathcal{X}' \subseteq \mathcal{X}$ and $\mathcal{Y}' \subseteq \mathcal{Y}$ such that $|\mathcal{Y}'| = \ell$ and $f_{\mathcal{X}'}$ shatters $\mathcal{Y}$. In particular, $|\mathcal{X}'| = 2^\ell$. The existence of such subsets follows from the definition of VC Dimension. The input distribution we consider is $\mu = \mu_{\mathcal{X}} \times \mu_{\mathcal{Y}}$ where $\mu_{\mathcal{X}'}$ and $\mu_{\mathcal{Y}'}$ are uniform and independent over $\mathcal{X}'$ and $\mathcal{Y}'$, respectively. Let X, Y be chosen from μ. Consider a one-way protocol for f where Alice sends $A = A(X)$ to Bob, who estimates $f(X,Y)$ from $A(X)$ and Y. Suppose this protocol has complexity $\mathcal{D}$ and error δ. Recall Fano's Inequality from information theory [5]: *Suppose a message X drawn from a domain $\mathcal{X}$ is sent through a noisy channel; denote the received message by Y, drawn from the set $\mathcal{Y}$. For any reconstruction function $g : \mathcal{Y} \to \mathcal{X}$ that recovers X from Y with error δ satisfies:*

$$H_2(\delta) + \delta \log(|\mathcal{X}| - 1) \geq H_2(X|Y).$$

Applying Fano's Inequality (ignoring the second term) here gives:

$$H_2(\delta) \geq H_2(f(X,Y)|A(X),Y) = \frac{1}{\ell} \sum_{y \in \mathcal{Y}'} H(f(X,y)|A(X), Y = y),$$

by conditioning on Y and using the fact that $\mu_{\mathcal{Y}'}$ is uniform. By independence, $H(f(X,y)|A(X), Y = y) = H(f(X,y)|A(X))$. By the subadditivity of entropy, the sum is greater than the entropy of $\{f(X,y)\}_{y\in\mathcal{Y}'}|A(X)$. Applying the entropy chain rule, $H(\{f(X,y)\}_{y\in\mathcal{Y}'}|A(X)) \geq H(\{f(X,y)\}_{y\in\mathcal{Y}'}) - H(A(X))$. Therefore

$$\ell \cdot H_2(\delta) \geq H_2(\{f(X,y)\}_{y\in\mathcal{Y}'}) - H_2(A(X)).$$

Since $\mathcal{X}'$ shatters $\mathcal{Y}'$ and X is uniform on $\mathcal{X}'$, $\{f(X,y)\}_{y\in\mathcal{Y}'}$ is uniform over ℓ-bit strings. Therefore $H_2(\{f(X,y)\}_{y\in\mathcal{Y}'}) = \ell$. Since the entropy of $A(X)$ is at most $\mathcal{D}$, the result follows. ■

We get the following Corollary from Lemma 18 and Theorem 19:

Corollary 20

$$R_\delta(DISJ_{n,\epsilon}) \geq \frac{1}{2\epsilon}(1 - H_2(\delta)).$$

Proof Again set $f = \mathrm{DISJ}_{n,\epsilon}$. From Theorem 19, there exists a distribution μ on $\mathcal{X} \times \mathcal{Y}$ such that

$$D_{\mu,\delta}(f) \geq VCD(f_{\mathcal{X}})(1 - H_2(\delta)).$$

By Lemma 18, $\mathrm{VCD}(f_{\mathcal{X}}) = \frac{1}{2\epsilon}$. To complete the proof, we note that the Yao Minimax Principle implies $R_\delta(f) = \max_\mu D_{\mu,\delta}(f)$; that is, to prove a lower bound on the randomized communication complexity, we choose an input distribution μ and prove a lower bound for the deterministic communication complexity under μ. The result follows. ■

We are now in a position to prove Theorem 15. Since randomized (one-way) communication complexity is a lower bound on the amount of space used by a protocol/algorithm, a reduction from $\mathrm{DISJ}_{n,\epsilon}$ to determining F_0 implies that the bound in Corollary 20 applies to the space of any algorithm computing F_0 in the stream model.

Proof Let $\mathcal{A}$ be a space-optimal randomized algorithm that (ϵ, δ) approximates F_0 on streams from a universe of size n, and let $S = \mathrm{DS}_{\epsilon,\delta}(F_0)$ be the space used by $\mathcal{A}$. We use $\mathcal{A}$ to construct a one-way communication protocol Π for $\mathrm{DISJ}_{n',\epsilon'}$, where $n' = 2n/(1+6\epsilon)$ and $\epsilon' = 3\epsilon$.

Let $a = (x, y)$ be an instance of $\mathrm{DISJ}_{n',\epsilon'}$. Note the following: if $x \cap y = \emptyset$ then $F_0(a) = n'/2 + \epsilon' n'$, so with probability at least $1 - \delta$, $\mathcal{A}(a) > (1-\epsilon)(n'/2 + \epsilon' n')$. However, if $y \subseteq x$ then $F_0(a) = n'/2$, so with probability at least $1 - \delta$, $\mathcal{A}(a) < (1+\epsilon)n'/2$. By our choice of n' and ϵ', $(1+\epsilon)n'/2 < (1-\epsilon)(n'/2 + \epsilon' n')$. The definition of Π follows naturally.

The protocol will break the computation of $\mathcal{A}(a)$ into two parts. First, Alice runs $\mathcal{A}$ on the first $n'/2$ elements of a; note this is x. Alice records the state of $\mathcal{A}$ at this point and sends it to Bob. Bob runs $\mathcal{A}$ from this state on the rest $(\epsilon' n')$ of a. If $\mathcal{A}$ returns a value greater than $(1-\epsilon)(n'/2 + \epsilon' n')$, then Bob sets $\mathrm{DISJ}(a) = 0$, and 1 otherwise. As noted, with probability at most $1 - \delta$ Bob will compute $\mathrm{DISJ}(x, y)$ incorrectly. The result follows from Corollary 20. ■

5 Some Initial One-pass Attempts

In this section we present some of our naive attempts at achieving the $O(\frac{1}{\epsilon}\text{polylog}(m))$ space bound using only one pass. First, we begin by analyzing good algorithms constructed for some special cases: $F_0 = O(\log m)$ and $F_0 = \theta(m)$. The algorithms are instructive in analyzing tradeoffs over the range of possible values for F_0, and when combined provide a $O(\frac{\sqrt{m}}{\epsilon}\log m \log(1/\delta))$ space algorithm for all F_0.

We also provide another slightly more involved $O(\frac{\sqrt{m}}{\epsilon}\log m \log^2(1/\delta))$-space algorithm that is based on sampling and choosing an appropriate event estimator. The poor space achieved in this algorithm illustrates the difficulties inherent in using a purely sampling-based approach.

5.1 $F_0 = O(\log m)$

If $F_0 = O(\log m)$, we can afford to allocate as much space as needed to store all the distinct elements in the stream. The main idea of the algorithm **Naive-Storing** below is to simply maintain a data structure (we use a balanced binary search tree) containing the current distinct elements seen so far, and to update the data structure as necessary when we encounter a new distinct element. After the stream has been processed, we simply output the number of elements in the data structure as our best estimate for F_0.

Naive-Storing$(a_1, a_2, \ldots a_n, \delta)$

1. For each i, $1 \leq i \leq \log(1/\delta)$, initialize an empty balanced binary search tree BST_i and a counter $c_i \leftarrow 0$ to indicate the number of elements in BST_i.
2. Let $t = O(\log m)$ be a threshold for the maximum number of items that we allow in any BST_i.
3. Pick a prime p, $4t^2 \leq p \leq 8t^2$.
4. For each i, $1 \leq i \leq \log(1/\delta)$, choose a random element $a_i \neq 0$ in the field $GF(p)$ to define a nearly universal hash function $h_i = a_i x \bmod p$.
5. For each stream element a_k.
    - $\forall i$, if $h_i(a_k)$ does not exist in BST_i and $c_i < t$, add $h_i(a_k)$ to BST_i and increment c_i by 1.
6. Output $\hat{F_0} = max_i\ c_i$ as best estimate for F_0.

The purpose of hashing is to reduce the description of an element a_k in the stream from $O(\log m)$ bits to $O(\log(\log m))$ bits; this can be done because we are assuming that $F_0 = O(\log m)$. We choose our hash functions to hash to tables of size $O(\log^2 m)$ in the hope that we might get at least one perfect hash function; if we get a perfect hash function then all $F_0 = O(\log m)$ distinct values in the stream will have distinct hashes, and we can store the hashes instead of the values to save space. From Corollary 8.20, for the type of nearly universal hash family used in the above algorithm, at least half the choices of hash functions will be perfect. Choosing $O(\log(1/\delta))$ hash functions then ensures that at least one hash function will be perfect with probability $1-\delta$. The total space used equals $\tilde{O}(\log m \log(1/\delta))$. The time for processing each element is dominated by the time necessary to compute all the hashes: $\tilde{O}(\log(1/\delta)\log m)$.

The algorithm presented here is attractive because it can always be run in parallel with algorithms suited for the case that $F_0 = \Omega(\log m)$ without harming the space bound (except for by a constant factor). It is also attractive because it never uses no more space as necessary. Furthermore, the algorithm can signal failure when any BST_i has more elements than the threshold, so that we can know in which cases its estimates for F_0 are invalid. This facet of the algorithm will be used later when combining it with the naive Monte Carlo algorithm presented next..

5.2 $F_0 = \theta(m)$

Suppose we know that $F_0 \geq cm$ for some constant c. Then we can use the naive Monte Carlo sampling approach to obtain a good estimate for F_0. If we sample $k = O(\frac{1}{\epsilon^2})$ values from the universe $[m]$ and check how many of them appear in the data stream, we can achieve a $(1 \pm \epsilon)$ approximation to F_0 with a constant probability of error. We let $X_i = 1$ if the ith sample from the universe appears in the stream and 0 otherwise. Let $S = \sum_i X_i$. Our estimator $\hat{F}_0 = \frac{Sm}{k}$. To achieve a $(1 \pm \epsilon)$ approximation to F_0, we must achieve a $(1 \pm \epsilon)$ approximation to S. From Chebyshev,

$$P(|S - E[S]| > \epsilon E[S]) \leq \frac{k\frac{F_0}{m}(1 - \frac{F_0}{m})}{\epsilon^2 k^2 (\frac{F_0}{m})^2} = \frac{\frac{m}{F_0} - 1}{\epsilon^2 k}$$

Since $\frac{m}{F_0}$ is restricted to be at most c, $k = O(1/\epsilon^2)$ samples are sufficient to approximate F_0 with a constant probability of error. We can achieve a (ϵ, δ) approximation with the usual median-finding procedure by using $O(\log(1/\delta))$ times as many samples. The total space necessary is $O(\frac{1}{\epsilon^2}\log(1/\delta)\log m)$ bits to store each of the $O(\frac{1}{\epsilon^2}\log(1/\delta))$ values for whose presence we want to check for in the stream. To generate the random samples from the universe, it is sufficient to use a pairwise-independent pseudorandom generator of the form $ax + b \bmod p$, where p is a prime which is $\theta(m)$ and a and b are randomly chosen elements of $GF(p)$. The processing time per stream element is dominated by the time spent comparing it to the $O(\frac{1}{\epsilon^2}\log(1/\delta))$ samples. If we use $O(\log(1/\delta))$ balanced binary search trees to store the samples, we can bring down the processing time per element to $\tilde{O}(\log m \log(1/\delta))$.

5.3 Combining Naive-Storing and Monte Carlo

We have presented above two algorithms whose space requirements are good provided certain constraints hold. When $F_0 = O(\log m)$, a storage approach works well while when $F_0 = \theta(m)$, a sampling approach works well. Since both these algorithms work well in very different regimes, it seems logical that we can combine them in order to do better. The algorithm **NSMC** does exactly that and achieves a $O(\frac{1}{\epsilon}\sqrt{m}\log(1/\delta)\log(m))$ space bound.

NSMC$(a_1, a_2 \ldots a_n, \epsilon, \delta)$

1. Initialize $O(\log(1/\delta))$ empty balanced binary search trees BST_i. The BST_i's will be used for the Monte-Carlo part of the algorithm. In addition, initialize one empty binary balanced search tre BST_{NS}, which will be used for the **Naive-Storing** part of the algorithm. Let $c \leftarrow 0$ represent the number of elements stored in BST_{NS}. Set the threshold t to be $\theta(\frac{1}{\epsilon}\sqrt{m})$.

2. Choose a prime p between m and $2m$, and randomly generate two elements a and b from the field $GF(p)$. Use the function $ax + b \bmod p$ to generate $O(\frac{1}{\epsilon}\sqrt{m}\log(1/\delta))$ pairwise independent samples from $[m]$ and insert t samples into each BST_i.

3. Initialize $\theta(t\log(1/\delta))$ boolean random variables $b_{i,j} \leftarrow 0$ which will indicate whether the jth element inserted into BST_i has appeared in the stream.

4. Initialize boolean overflow flag $f \leftarrow 0$.

5. For each stream item a_k,
   - If a_k does not appear in BST_{NS} and $c < t$, insert a_k into BST_{NS}. If $c = t$ and this is the $t + 1$st distinct element seen, set $f = 1$.
   - $\forall i$, search for a_k in BST_i. If a_k matches the jth inserted element of BST_i, set $b_{i,j} = 1$.

6. If $f = 0$ return $\hat{F}_0 = c$. Otherwise compute $\log(1/\delta)$ estimates $\hat{F}_0^i = \frac{m}{t}\sum_j b_{i,j}$. Return the median of the $\hat{F}_0^i$'s as the estimator $\hat{F}_0$.

The algorithm basically runs the **Naive-Storing** and Monte-Carlo algorithms in parallel. The major difference is that we have eliminated the hash functions in the original **Naive-Storing** algorithm because very little can be gained from reducing the description size of each element; in the worst case the description size for each element appearing in the stream would still be $O(\log m)$. What we have gained is that we no longer need $O(\log(1/\delta))$ balanaced binary search trees to compensate for the fact that the hash functions may not be perfect.

Theorem 21 *Algorithm **NSMC** returns a (ϵ, δ) approximation to F_0.*

Proof The correctness of the algorithm is as follows: If $f = 0$, then BST_{NS} must have $\leq t$ elements in it. Since we are not using the hashing scheme in the original **Naive-Storing** algorithm, we can be certain with probability 1 that the overflow flag is set correctly. The analysis just becomes that of a deterministic algorithm and our estimate $\hat{F}_0$ is exact. If $f = 1$, then it must be the case that $F_0 > t = \theta(\frac{1}{\epsilon}\sqrt{m})$. From the analysis of the Monte-Carlo algorithm, we have that $k = O(\frac{\frac{m}{F_0}-1}{\epsilon^2})$ samples are sufficient to achieve an approximation to F_0 with a constant probability of error. Since $F_0 = \Omega(\frac{1}{\epsilon}\sqrt{m})$, we have that k should be $\theta(\frac{1}{\epsilon}\sqrt{m})$. But each BST_i has $\theta((\frac{1}{\epsilon}\sqrt{m}))$ samples so we satisfy the requirement for k. As usual, we keep $O(\log(1/\delta))$ balanced binary search trees and return the median of our estimates to reduce the probability of error to δ. ■

The threshold t is in fact optimal for minimizing the total space used by the algorithm. It was obtained by setting the space requirement for the **Naive-Storing** algorithm equal to the space requirement necessary for the Monte-Carlo algorithm.

Theorem 22 *Algorithm **NSMC** uses $\theta(\frac{1}{\epsilon}\sqrt{m}\log(1/\delta)\log m)$ space and takes $\tilde{O}(\log^2 m \log(1/\delta))$ time to process each element.*

Proof BST_{NS} can store at most $O(\frac{1}{\epsilon}\sqrt{m})$ distinct elements from the stream, and each BST_i stores $\theta(\frac{1}{\epsilon}\sqrt{m})$ samples from the universe. Since there are $O(\log(1/\delta))$ BST_i's, the total space usage is $\theta(\frac{1}{\epsilon}\sqrt{m}\log(1/\delta)\log m)$ bits. The processing time per stream element a_k depends on the search time in the binary trees. Since we are using balanced binary search trees, the number of samples a_k is compared to is at most $O(\log(\log(\frac{1}{\epsilon}\sqrt{m})))$ for each binary search tree. Since there are $O(\log(1/\delta))$ BST's and a comparison takes $O(\log m)$ time, the total processing time per element is $\tilde{O}(\log^2 m \log(1/\delta))$. ■

When $\frac{1}{\epsilon} = \omega(\sqrt{m})$, algorithm **NSMC** performs strictly better than all three algorithms in [2] in terms of space usage. Unfortunately, for this case, the naive $O(m)$-space determinstic outperforms **NSMC** and all three algorithms from [2].

5.4 A Second Sampling-Based Algorithm

In this section we present a sampling-based algorithm that achieves $O(\frac{1}{\epsilon}\sqrt{m}\log^2(1/\delta)\log m)$ space and $\tilde{O}(\frac{1}{\epsilon}\sqrt{m}\log^2(1/\delta)\log m)$ processing time per element. The space and time achieved are atrocious, even worse than the **NSMC** algorithm, but it may be of interest to understand the algorithm to avoid taking an approach similar to it in the future. The algorithm is based on choosing a collisions-based events estimator over a sample of elements of size $t = O(\sqrt{m}\frac{1}{\epsilon}\log(1/\delta))$. The key fact is that each element in this sample is equally likely to be one of the F_0 distinct stream elements.

SS$(a_1, a_2, \ldots a_n, \delta, \epsilon)$

1. Let $t = O(\frac{1}{\epsilon}\sqrt{m}\log(1/\delta))$. Choose an appropriate prime $p = \theta(m^2)$. For $1 \leq i \leq O(\log(1/\delta)t)$, randomly and independently generate a hash function $h_i : [m] \to [p]$ from the family of universal hash functions of the form $ax + b \bmod p$.

2. $\forall i$, let $v_i \leftarrow \infty$ denote the minimum hash value seen so far in the stream for hash function h_i, and let $s_i \leftarrow NULL$ denote the first element from the stream that was hashed to the value v_i. Let boolean $b_i \leftarrow 0$ be used to indicate if two or more distinct elements have hashed to v_i.

3. For each stream element a_k

   - $\forall i$, if $h_i(a_k) < v_i$, set $v_i = h_i(a_k)$, $s_i = a_k$, and $b_i = 0$. If $h_i(a_k) = v_i$ and $a_k = s_i$, set $b_i = 1$.

4. Let set $T \leftarrow \{\}$. $\forall i$, if $b_i = 0$, add s_i to T.

5. Break up T into $O(\log(1/\delta))$ disjoint sets T^k, each with an equal number of elements. Let t_i^k denote the ith element of T^k. Let random variable $X_{i,j}^k = 1$ if $t_i^k = t_j^k$ and 0 otherwise.

6. $\forall k$, efficiently compute $C^k = \sum_{i,j,i<j} X_{i,j}^k$.

7. Let C^m equal the median of the values C^k, and return $\hat{F}_0 = \frac{\binom{|T^m|}{2}}{C^m}$.

Theorem 23 *Algorithm **SS** produces a (ϵ, δ) approximation for F_0.*

Proof The algorithm begins by generating a set T containing at least $t = O(\frac{1}{\epsilon}\sqrt{m}\log(1/\delta))$ samples. Each element is independently and equally likely to be one of the F_0 distinct stream elements. The main idea is that, provided that our hash function produces a unique minimum, the element that has the minimum hash value is equally likely to be any of the F_0 distinct elements. We want our hash functions to be have unique minima to avoid the problem of having to distinguish duplicates in the stream. The boolean values b_i in the algorithm are used to determine which hash functions actually have a unique minimum. If we choose a perfect hash function, then we are guaranteed to have a unique minimum. From Chernoff and Corollary 8.20, we need to choose $O(\log(1/\delta)t)$ hash functions to guarantee with error probability $\theta(\delta)$ that we will get at least t valid samples.

Next we argue that t samples are sufficient to produce an (ϵ, δ) approximation to F_0. If there are t samples in T, then each subset T^k is guaranteed to have at least $t' = \Omega(\sqrt{m}\frac{1}{\epsilon})$ samples. WLOG, consider the case where $\forall k, |T^k| = t'$. We have denoted t_i^k as the ith sample of T^k, and indicator random variable $X_{i,j}^k, i \neq j$ equals 1 if $t_i^k = t_j^k$. $C^k = \sum_{i,j} X_{i,j}$ is the sum of all the possible pairs of elements in T^k which have equal values, or collide. Intuitively, the number of collisions should increase with the number of distinct elements in the stream, so C^k should be a good estimator for F_0. For any k, the expected value of $C^k = \binom{t'}{2}\frac{1}{F_0}$ since $P(X_{i,j}^k = 1) = \frac{1}{F_0}$. It is easy to see that for fixed k, the $X_{i,j}^k$'s are pairwise independent, so the variance of $C^k = \binom{t'}{2}\frac{1}{F_0}(1 - \frac{1}{F_0})$. From Chebyshev, $O(\sqrt{m}\frac{1}{\epsilon})$ samples is sufficient to achieve an estimate to the expected number of collisions in a sample of t' elements with a constant probability of error (we substitute for F_0 its maximum value m). With $O(\log(1/\delta))$ subsets T^k, we can achieve an (ϵ, δ) approximation to the expected number of collisions using the usual median-finding procedure. Now, because $\binom{t'}{2}$ can be treated as a constant and because an $(\epsilon/2, \delta)$ approximation to the expected number of collisions provides an (ϵ, δ) approximation to F_0, if we choose the constants appropriately we can generate an (ϵ, δ) estimate for F_0. ■

Theorem 24 *The algorithm **SS** uses $O(\log^2(1/\delta)\frac{1}{\epsilon}\sqrt{m}\log m)$ bits of space and processes each element in $\tilde{O}(\log^2(1/\delta)\frac{1}{\epsilon}\sqrt{m}\log m)$ time.*

Proof The space consumption is dominated by the number of hash functions and the values stored with each hash function: namely the description of the hash function, v_i, s_i, and b_i. In total, we use $O(\log^2(1/\delta)\frac{1}{\epsilon}\sqrt{m})$ hash functions, each needing $O(\log m)$ bits of space. The processing time per element is dominated by the amount of time necessary to compute all the hash functions. Since a multiply takes $\tilde{O}(\log m)$ time, the total processing time per element equals $\tilde{O}(\log^2(1/\delta)\frac{1}{\epsilon}\sqrt{m}\log m)$. ■

We have not described exactly how each C^k is computed. Using a balanced binary search tree, it is probably possible to compute C^k in much less time than the naive approach of comparing every pair of different elements.

The $\sqrt{m}$ factor in the space bound is reminiscent of the birthday paradox. With a universe of size F_0, a subset of size $\sqrt{F_0}$ elements drawn with replacement from $[F_0]$ has at least two elements of equal value with constant probability. Intuitively, if we have asymptotically less than $\sqrt{F_0}$ elements, then we should expect all the elements in the set to be distinct; in this case, all we can hope for is an underestimate for F_0.

The space in the algorithm suffers primarily because there really isn't a way to sample uniformly from the F_0 distinct elements until the entire stream has been processed; and, furthermore, even if we could sample while processing the stream, we need to be able to compute the estimator as we process the stream as well. In general, the algorithm illustrates the problems with using a sampling-based approach for this problem.

6 Coupling Sampling with Divide and Conquer Algorithms

Algorithms **NMSC** and **SS** show that it is important to distinguish between the number of samples required and the amount of space used by an algorithm. Most (ϵ, δ)-randomized approximation algorithms which sample the input require $\Omega(\frac{1}{\epsilon^2})$ samples to achieve the desired confidence level. However, these algorithms are not required to store all the samples before computing and returning a final estimate. Instead, they can partially compute the estimate along the way, and throw away samples as they go, saving space. To illustrate what is meant by this, we first outline an elegant median-finding technique from [7] in the stream model.

The technique works by sampling $\Omega\left(\frac{1}{\epsilon^2}\right)$ elements of the input stream and returning an approximation to the median of the sample as an estimate to the median of the entire stream. As the samples are taken from the stream they are fed into a deterministic divide and conquer approximation algorithm D. In each subproblem, D takes in some number b of buffers, each of size k, and collapses them into a single buffer of size k, containing the elements of rank $\frac{k}{2} + jb, 0 \leq j < k$, of the sorted subproblems (i.e., we take all bk elements from the bk buffers, we sort them, and we extract the elements of the specified rank, and put them in a single buffer and free up the input buffers). The exact strategy of when to collapse buffers is left up to the algorithm. Different collapsing strategies give different error guarantees. In [7] the following theorem is proven:

Theorem 25 *For any stream of size n, there exists an algorithm that with probability at least $1-\delta$, computes an approximation to the median with absolute error at most ϵn in a single pass using $\tilde{O}(\frac{1}{\epsilon} \log n)$ bits of memory.*

We run into two problems when we try to apply the above technique to the distinct elements problem. The first is that we cannot simply sample some subset of the stream to concentrate on. In [4] the following theorem is proven:

Theorem 26 *Any algorithm which computes an $(\epsilon, \delta)-$approximation to the number of distinct elements in a data stream of n elements must examine at least $\frac{n \ln \frac{1}{\delta}}{\epsilon^2 \left(2\epsilon^2 + \ln \frac{1}{\delta}\right)}$ stream elements in the worst case.*

Hence, we need to examine almost the entire stream in the worst case to accurately compute the number of distinct elements. On top of this, there do not even exist good deterministic divide and conquer algorithms for approximating F_0 in the stream model, where good is taken to mean efficient in space. This negative result for computing F_0 is summarized in the following theorem, shown in [1]:

Theorem 27 *Any deterministic algorithm that outputs, given a stream of $\frac{m}{2}$ elements of a universe of size m, an estimate Y such that $|Y - F_0| \leq .1F_0$, must use at least $\Omega(m)$ bits of space*

Note that even though we did not state this theorem in terms of ϵ, the requirement on space only gets worse for $\epsilon \leq .1$ and the $\Omega(m)$ space bound is already prohibitive. Hence, we see that the "sampling followed by deterministic divide and conquer approach" used for median-finding does not look like a promising technique for estimating F_0. Note that in particular the above theorem gives an $\Omega(m)$ space lower bound for any deterministic exact algorithm. $O(m \log m)$ is clearly an upper bound since one can always create a table to keep track of a count of the number of times each distinct element occurs.

Another attempt one might try is to *reduce* the problem of computing F_0 to that of finding an appropriate quantile in a data stream. The above median-finding procedure presented in [7] generalizes to an $O(\frac{1}{\epsilon}\log m)$-space algorithm for computing arbitrary quantiles. Recall that algorithm **A2** kept track of the smallest $O(\frac{1}{\epsilon^2})$ hashed distinct values and then output an estimate based solely on a single value, which had rank $\lceil\frac{96}{\epsilon^2}\rceil$ amongst the image of the hash function applied to the entire stream. What if instead we were to use a quantile-finding algorithm to approximately find the $\lceil\frac{96}{\epsilon^2}\rceil$th hashed value in the stream, instead of storing a linked list as in **A2**? The problem is that the quantile-finding algorithm of [7] computes the specified quantile over all elements of the stream, not just the *distinct* elements, which is what we want. Unfortunately we do not see a way of reducing this problem to that of finding a specified quantile over nondistinct elements.

7 Multiple Pass Algorithms

Another interesting idea would be to relax the model to allow for a small number of passes over the data. It's conceivable that we could first try to approximate F_0 in two passes, for instance, and then devise clever data structures to perform the computations we had previously done in the second pass in parallel with the first one. Note that this might have been how the authors of [2] approached algorithm **A3**, namely, they might have initially assumed R was computed in the first pass before coming up with their one-pass solution (which was achieved by recording indices, then taking hash function restrictions with their knowledge of R).

Although we did not achieve a novel 2-pass solution to this problem, we made the simple observation that in algorithm **A3** (recall that this algorithm achieves the best known space for computing F_0), one can split up the computation of $X(H_R)$ over multiple passes. Suppose we make $l+1$ passes. We use the first pass solely to compute R. Then we run a slightly modified form of **A3** in the remaining l passes. Note that not computing R in each pass drops a $\frac{1}{\epsilon^2}\log\log m$ factor to a $\frac{1}{\epsilon^2}$ factor in the space bound, since, instead of remembering y_j, we can compare to $TRAIL(w_j(a_i)) to \log R$ immediately and keep a single bit denoting whether a stream element hashed to 0 under $w_j = h(j)$ (so that we do not add 1 to $X(H_R)$ more than once for the same $h(j)$).

Now in each successive pass, instead of setting $k = \frac{18000}{\epsilon^2}$ in step b of **A3**, we set $k = \frac{18000}{l\epsilon^2}$. In each pass we will use a different master hash function $h_i, 2 \leq i \leq l+1$, each with domain $[k]$. Then in each pass we compute $|\{j | h_i(j)^{-1}(0) \cap B \neq \emptyset\}|, 1 \leq j \leq k$, divide by $\frac{18000}{\epsilon^2}$, and add the result to a running sum. At the end we will have the same value $X(H_R)$ as the unaltered **A3** would compute, but we only used $\tilde{O}((\frac{1}{l\epsilon^2} + \log m)\log\frac{1}{\delta})$ space in each pass, and hence in total. Intuitively, we have split up the computation of an average into computing an average of averages. For a real world application, suppose that we do not care how much time we spend per element, but are only limited by space. Using this technique, we can knock down the space of **A3** to $\tilde{O}(\log m \log\frac{1}{\delta})$ by setting $\frac{1}{l\epsilon^2} = \log m$ and solving for l. We see that if we allow for $\frac{1}{\epsilon^2 \log m}$ passes, we only use $O(\log m)$ space per iteration. Note that achieving minimal space could be important in real life when we need to make due with how much physical RAM we have while streaming over some massive dataset, perhaps through an internet connection.

8 Conclusion

In this paper we considered the problem of providing a (ϵ, δ) approximation for the number of distinct elements F_0 in a stream under extreme space constraints. With a lower space bound of $\Omega(m)$ for any deterministic algorithm for this problem, it is natural to consider efficient randomized algorithms. We began with a survey of the current state-of-the-art algorithms [2] and a review of the tightest known lower bounds [1], [3]. We then tried to address two related open questions posed in [2]: 1) can we construct a $\frac{1}{\epsilon}\text{polylog}(m)$-space algorithm for this problem, or 2) can we prove a $\Omega(\frac{1}{\epsilon^2})$ space lower bound?

Although we failed in both pursuits, we gained several insights into the nature of the problem. One key insight is that any sampling-based approach is probably doomed to failure. Too many samples are required for storage in the worst-case to produce a good estimate; this was illustrated in algorithms **NSMC** and **SS** and proven in [4]. Even combining sampling with a strategy like divide and conquer to avoid storing the samples seems to not provide an adequate solution.

A common thread in the most efficient algorithms for this problem is that they try use as an estimator for F_0 the probability of hashing to a specific bin. The approach is very robust to adversaries, requires very little space to compute, and is easy to compute on the fly. It seems that if we were to come up with a better algorithm for this problem, this estimator itself will not be improved. A better approach might be to view the problem in terms of data structures: all the algorithms in [2] essentially improve existing algorithms by using better data structures and clever hashing tricks. In all of this, of course, we have to also consider the possibility that the $\Omega(\frac{1}{\epsilon^2})$ space bound might be tight.

Another insight we gained is that it is surprisingly difficult to come up with algorithms that do better than the ones in [2] even if we relax the problem to allow for multiple passes. This is partially because the best algorithms already run several smaller algorithms in parallel; for instance, **A3** runs **A1** and does other processing on the side to refine the estimate from **A1**. It is a non-trivial matter to determine exactly what information should be stored on the first pass of the algorithm that could aid in future passes.

References

[1] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the 28th Annual ACM Symposum on the Theory of Computing*, p. 20-29, 1996.

[2] Z. Bar Yossef, T.S. Jayram, Ravi Kumar, D. Sivakumar, and Luca Trevisan. Counting distinct elements in a data stream. *RANDOM 2002, 6th. International Workshop on Randomization and Approximation Techniques in Computer Science*, p. 1-10, 2002.

[3] Z. Bar Yossef. The complexity of massive data set computations. Ph.D. Thesis, U.C. Berkeley, 2002.

[4] M. Charikar, S. Chaudhuri, R. Motwani, and V. Narasayya. Towards estimation error guarantees for distinct values. In *Proceedings of the 19th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, p. 268-279, 2000.

[5] T.M. Cover and J.A. Thomas, *Elements of Information Theory.* New York: Wiley, 1991.

[6] R.M. Karp, C. H. Papadimitriou, S. Shenker. A simple algorithm for finding frequent elements in streams and bags. Preprint, available at `http://www.cs.berkeley.edu/~ christos/iceberg.ps`.

[7] G. Manku, S. Rajagopalan, and B. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *Proceedings of SIGMOD '98*, p. 426-435. ACM Press, 1998.

[8] V. N. Vapnik and A. Y. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications,* XVI(2):264-280, 1971.

[9] A. C-C. Yao. Lower bounds by probabilistic arguments. In *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science*, p. 420-428, 1983.

[10] D. Ying. Computing Distinct Values. Lecture 1 for CS 361A at Stanford University. Available at `http://theory.stanford.edu/~ rajeev/cs361.html`, 2001.