

Finding duplicates in a data stream

Parikshit Gopalan*
University of Washington
& Microsoft Research SVC.
parik@cs.washington.edu

Jaikumar Radhakrishnan
TIFR, Mumbai.
jaikumar@tifr.res.in

Abstract

Given a data stream of length n over an alphabet $[m]$ where $n > m$, we consider the problem of finding a duplicate in a single pass. We give a randomized algorithm for this problem that uses $O((\log m)^3)$ space. This answers a question of Muthukrishnan [Mut05] and Tarui [Tar07], who asked if this problem could be solved using sub-linear space and one pass over the input. Our algorithm solves the more general problem of finding a positive frequency element in a stream given by frequency updates where the sum of all frequencies is positive. Our main tool is an Isolation Lemma that reduces this problem to the task of detecting and identifying a Dictatorial variable in a Boolean halfspace. We present various relaxations of the condition $n > m$, under which one can find duplicates efficiently.

1 Introduction

The problem of detecting duplicates in a large data set is a ubiquitous and well-studied problem. In databases, this problem arises in the context of detecting repeated records [GCM05, BM03], search engines encounter it in the context of detecting duplicated web-pages [BNW03], and in networking it arises in the context of monitoring network traffic in order to detect anomalies in it [MAA05, DR06]. In this paper, we consider the duplicate detection problem in the data stream model. This is a natural model for many of the applications above and has been studied in previous work [Mut05, MAA05, DR06, Tar07]. Given a stream of n numbers from an alphabet $[m]$, our goal is to find a duplicate in the data stream if one exists, in a single pass. While the items in the data stream might be more complex objects (like web-pages or database records), one can always map such objects to integers by hashing or other means, so this assumption is without loss of generality. In some applications, one would like to tell whether each new element has been seen previously or in some small window. However, an easy reduction

from set-disjointness shows that even of the problem of duplicate detection stated above is hard: it requires linear storage space even for randomized algorithms that can make multiple passes over the data.

This has motivated studying variants of the problem that might admit efficient solutions. One such variant of the problem that has been studied intensively is where some of the items occur with relatively high frequencies, and our goal is to find these frequent items [Cha08]. We will study the problem when the length n of the stream is larger than the alphabet m . A duplicate now is guaranteed by the pigeonhole principle and the goal is to find a duplicated element.

PROBLEM 1. FINDDUPLICATE: *Given an input stream $S = s_1s_2 \cdots s_n$ where $s_i \in [m]$ and $n > m$, find $a \in [m]$ which occurs at least twice.*

This problem is posed by Muthukrishnan in his survey on data stream algorithms [Mut05, Section 0.2.3, Puzzle 3: Pointer and Chaser], to illustrate the trade-off between random and sequential access (and also in a post on Fortnow's Complexity weblog [For05]). With random access to the input, the problem can be solved with $O(\log m)$ space. Muthukrishnan raised the question of whether similar space guarantees are possible for streaming algorithms, possibly by using multiple passes. For deterministic algorithms, this question was resolved negatively by Tarui [Tar07], who showed a lower bound of $\Omega(m^{\frac{1}{k}})$ when k passes are allowed and $n = m + 1$. The randomized complexity of this problem was open.

1.1 Related work Duplicate detection on data streams was considered by Metwally *et al.* [MAA05], their motivation was the detection of frauds in click-streams, the goal being to tell whether clicks to an advertisement are from genuine users or generated artificially by a computer. They do not make any assumptions about the relative size of m and n . They propose a solution using Bloom filters that detects all duplicates with good probability, but it requires $O(n)$ space. Another solution using a variant of Bloom filters was pro-

*Work done in part while visiting TIFR, Mumbai

posed by Deng and Rafiei [DR06].

We summarize the known bounds for FINDDUPLICATE below.

Deterministic: A simple one-pass deterministic algorithm is to keep a bit-vector $b \in \{0, 1\}^m$ to track which elements have been seen before, this needs space m . With k passes, one can reduce the space to $m^{\frac{1}{k}} \log(m^{\frac{1}{k}})$: we divide the alphabet into $m^{\frac{1}{k}}$ groups and compute the frequencies of each group in a single pass. One of the groups will occur more than $m^{\frac{1}{k}}$ times, we recurse on this group in the next pass. In particular, with $\log m$ passes, we need only $O(\log m)$ space.

Surprisingly, Tarui [Tar07] shows that when $n = m + 1$, this tradeoff between number of passes and space is near-optimal, by giving a lower bound of $\Omega(m^{\frac{1}{2k-1}})$. This bound is proved via an elegant reduction from the Karchmer-Wigderson communication game for monotone circuits computing majority [KN97]. Lower bounds for this game even with multiple passes follow from lower bounds for the size of small depth monotone circuits computing majority. But this lower bound does not extend to the case of $n \gg m$. Tarui raises the question of proving lower bounds when $n = 2m$.

Randomized: If $n > 2m$, if we sample a random element a_i , it will occur again with probability $\frac{1}{2}$. This generalizes to give a randomized algorithm that needs space $\frac{n}{n-m} \log m$. If n is close to m , this is linear space. It seems that no better upper bounds or non-trivial lower bounds were known for randomized algorithms.

A related problem that has been studied intensively in the data stream model is the problem of finding frequent items in a stream. We refer the reader to the recent survey by Charikar for the current state of the art [Cha08]. These algorithms typically do well on the heavy-hitters: items whose frequency is larger than some specified threshold. These algorithms do not solve the problem of duplicate detection, since as Metwally *et al.* point out, in the duplicate detection setting the vast majority of items that occur might be distinct, so there may not be any heavy-hitters [MAA05].

1.2 Our results We present a fairly complete characterization of the space complexity of FINDDUPLICATE for both deterministic and randomized algorithms that use a single pass. Our main algorithmic result is the first sub-linear space algorithm for FINDDUPLICATE: we give a randomized algorithm that uses $O((\log m)^3)$ space and update time. This settles a question raised by [Mut05, Tar07].

Our randomized algorithm will solve a more general problem described below. By a *stream of increments over an alphabet* $[m]$, we mean a sequence $\langle (i_j, c_j) \rangle_{j=1}^n$, where each $i_j \in [m]$ and $c_j \in \mathbb{Z}$. We think of c_j

(which could be negative) as the increment to the frequency of i_j . This is the standard turnstile model of Muthukrishnan [Mut05]. The *frequency vector* $f : [m] \rightarrow \mathbb{Z}$ associated with such a stream is given by $f[i] = \sum_{j:i_j=i} c_j$.

PROBLEM 2. FINDPOSITIVE: *Given a stream of increments of length n over the alphabet $[m]$ with the promise that $\sum_{i=1}^m f[i] > 0$, find an $i \in [m]$ such that $f[i] > 0$.*

THEOREM 1. *There is a one-pass randomized algorithm for FINDPOSITIVE that uses space $O(\log^2 n \log m)$.*

At each step our algorithm performs $O(\log n \log m)$ arithmetic operations on words of size $\log n$. There is a simple reduction from FINDDUPLICATE to FINDPOSITIVE, which gives the $O((\log m)^3)$ bound for FINDDUPLICATE. A suitable modification to this algorithm lets us find duplicates (if they exist), even when the length n is less than m . On streams of length $n < m$ our algorithm runs in space $O((m-n) \log^3 m)$, it finds a duplicate with probability $3/4$ if one exists, and returns Fail with probability $3/4$ if there are no duplicates. We also show that any constant-pass randomized algorithm for this problem requires space $\Omega(m-n)$.

Let us define

$$\ell_p(f) = \left(\sum_i |f[i]|^p \right)^{\frac{1}{p}},$$

$$\ell_p^+(f) = \left(\sum_{i:f[i]>0} |f[i]|^p \right)^{\frac{1}{p}}, \quad \ell_p^-(f) = \left(\sum_{i:f[i]<0} |f[i]|^p \right)^{\frac{1}{p}}.$$

Then FINDPOSITIVE can be restated as the problem of finding a positive frequency element when $\ell_1^+(f) > \ell_1^-(f)$. The following is a natural generalization of FINDPOSITIVE to the ℓ_p norm:

PROBLEM 3. ℓ_p -FINDPOSITIVE: *Given a stream of increments of length n over the alphabet $[m]$ such that $\ell_p^+(f) > \ell_p^-(f)$, find an $i \in [m]$ such that $f[i] > 0$.*

We give an algorithm that uses space $O((\log n \log m)^2)$ for ℓ_2 -FINDPOSITIVE.

THEOREM 2. *There is a one-pass randomized algorithm for ℓ_2 -FINDPOSITIVE that uses space $O((\log n \log m)^2)$.*

As an algorithm for FINDPOSITIVE, the requirement on the ℓ_2 norm is incomparable to the requirement on the ℓ_1 norm. However, for the instances of FINDPOSITIVE that are produced by our reduction from FINDDUPLICATE, the ℓ_2 requirement is weaker. In contrast, we show that any algorithm for ℓ_p -FINDPOSITIVE where $p > 2$ requires space polynomial in m .

THEOREM 3. *For $p > 2$, any c -pass algorithm for ℓ_p -FINDPOSITIVE requires space $\Omega\left(\frac{m^{1-2/p}}{c \log m}\right)$.*

We show a lower bound of $m - \log n$ on the space needed by one-pass deterministic algorithms for finding repeated elements for length n . This essentially gives a bound of m as long as $n = m^{O(1)}$. Our bound also applies to non-uniform deterministic algorithms, which are algorithms with read-only access to a clock which tells them the current position in the stream. Tarui shows a similar lower bound for one-pass algorithms using similar arguments, though his model of non-uniform algorithms is somewhat different from ours, so the results are incomparable [Tar07].

THEOREM 4. *Any one-pass deterministic algorithm for FINDDUPLICATE needs space $m - \log n$.*

This lower bound yields nothing when $n \geq 2^m$. It is natural to ask if efficient deterministic algorithms exist for this setting of parameters. For non-uniform algorithms, we show that the answer is yes, by giving an algorithm that uses space $\log m + O(1)$ for $n \geq 2^m$.

THEOREM 5. *There is non-uniform algorithm for FINDDUPLICATE using space $\log m + O(1)$ for $n \geq 2^m$.*

One can also tradeoff the space needed with the length n to essentially match the lower bound of $m - \log n$ for all n (see Theorem 20). In contrast, for uniform algorithms, we show that an $\Omega(m)$ lower bound holds for inputs of all lengths (see Theorem 17).

1.3 Sketch of the randomized algorithm The novelty of our algorithm is to relate the problem of finding a duplicate to questions about halfspaces on the Boolean hypercube. In addition, the algorithm uses by now standard tools such as pairwise independent hashing [AMS99, Cha08] and Nisan’s pseudorandom generator [Nis92, Ind06]. We give a sketch of the algorithm for FINDPOSITIVE in the ℓ_1 norm. Given the frequency vector f , we can associate to it the Boolean halfspace $f(x)$ mapping $\{\pm 1\}^n$ to $\{\pm 1\}$ given by $f(x) = \text{sgn}(\sum_i f[i]x_i)$. Given $S \subseteq [m]$, we define the halfspace $f^S(x) = \text{sgn}(\sum_{j \in S} f[j]x_j)$. The co-ordinate functions x_i are called dictatorships, note that they are also halfspaces.

We illustrate the connection between FINDPOSITIVE and halfspaces by a couple of examples:

1. There is a unique i so that $f[i] > 0$. In this case, we observe that $f[i] > \sum_{j \neq i} |f[j]|$. It follows that $f(x) = \text{sgn}(\sum_j f[j]x_j) = x_i$, so that f is the dictatorship of x_i .
2. There is a set D of $\frac{\ell_1^+(f)}{k}$ positive $f[i]$ s of size k , whereas all the others $f[i]$ s are negative. If we sample $S \subseteq [m]$ by picking each i independently with probability $\frac{1}{k}$, there is a constant probability

that we pick a unique $i \in D$ and further that $f^S(x)$ is the dictatorship of x_i . Thus sampling from $[m]$ at rate $\frac{1}{k}$ gives a dictatorship.

By sampling at a suitable rate, we reduce the general case of FINDPOSITIVE to finding a dictator in a halfspace. We sketch an argument that shows how sampling gives a dictatorship with probability at least $\Omega(\log^{-2} n)$. By a double-counting argument, there must always be some $k \in 1, \dots, \ell_1^+(f)$ so that there are about $\frac{\ell_1^+(f)}{k \log n}$ numbers of size at least k , which is only a factor $\log n$ below the best possible. Further, we can assume that k is a power of 2. So now, we guess this k and sample $S \subseteq [m]$ at a suitable rate. With probability $\Omega(\log^{-2} n)$, we are in the situation where $f^S(x) = x_i$ for some i such that $f[i] > k$. If we repeat this step $O(\log^2 n)$ times, the condition holds for one of the sets. By a more careful analysis, we can in fact show that one can isolate a dictatorship with probability $\Omega((\log n)^{-1})$. To implement this with small space in a streaming fashion, the sets S are picked pairwise independently as opposed to truly at random.

Assuming that $f^S(x)$ is indeed a dictatorship, there is a simple test to tell which $i \in [m]$ is the dictator, by taking $\log m$ inner products with suitably chosen vectors. But the problem is that there is a good chance that $f^S(x)$ is not a dictatorship, in which case the above test could return some arbitrary i . Our goal is to filter out such i ’s. We will in fact settle for a weaker guarantee: we reject any i so that $f[i] < 0$. Thus the algorithm might return a spurious dictator i even though $f^S(x)$ is not a dictatorship, but as long as $f[i] > 0$, this is good enough for FINDPOSITIVE.

For this verification step, we use the fact that if $f[i] \leq 0$, then $E_x[f^S(x)x_i] \leq 0$, whereas if $f^S(x) = x_i$ then $E_x[f^S(x)x_i] = 1$. Thus evaluating f^S on a few random strings is sufficient to distinguish between these cases. To implement this test in small space, we resort to Nisan’s pseudorandom generator for small space algorithms as opposed to truly random strings.

2 Preliminaries

There is a simple reduction from FINDDUPLICATE to FINDPOSITIVE. Given a data stream $S = \langle s_1, s_2, \dots, s_n \rangle$, in which we want to find a repeat, define a stream of increments

$$I = \langle (1, -1), (2, -1), \dots, (m, -1), (s_1, 1), (s_2, 1), \dots, (s_n, 1) \rangle.$$

Thus, to find a duplicate in the stream S , it is enough to locate an i such that $f[i] > 0$ for the frequency vector associated with the stream I . Further, we can assume that the stream S has length $m + 1$ by ignoring

the remainder of the stream. Hence for FINDDUPLICATE, our algorithms are parametrized by m only. In the rest of this section, we will deal exclusively with FINDPOSITIVE. Also, we will assume that the increments $c_j \in \{\pm 1\}$, as in the reduction from FINDDUPLICATE. But even allowing increments of size $\text{poly}(n)$ will only affect the complexity by constant factors.

DEFINITION 1. (RESTRICTIONS, DOT PRODUCTS)
 For a vector $x \in \mathbb{R}^m$, and a subset S of $[m]$, let the restriction of x to S be the vector x^S defined by $x_i^S = x_i$ if $i \in S$ and $x_i^S = 0$ if $i \notin S$. We define its inner product with f by $f \cdot x = \sum_{i=1}^n f[i] \cdot x_i$.

We define the Boolean functions $f(x) = \text{sgn}(f \cdot x)$ and $f^S(x) = \text{sgn}(f^S \cdot x)$ which map $\{\pm 1\}^m \rightarrow \{\pm 1\}$. Our algorithms proceed by computing the dot product of the frequency vector with suitably chosen random vectors $x \in \{\pm 1\}^m$. Assuming one can efficiently retrieve the coordinates of the vector x and check membership in the set S , then in one pass over the stream of increments $I = \langle (i_1, c_1), (i_2, c_2), \dots, (i_n, c_n) \rangle$ we can compute $(f^S \cdot x)$ by the following procedure:

ALGORITHM 1. COMPUTING DOT PRODUCTS

```

Let sum = 0.
For j = 1, 2, ..., n do
  if  $i_j \in S$  then sum = sum +  $x_{i_j} \cdot c_j$ ;
Return sum
  
```

Remark: The above code assumes that we have the stream of increments as input. However, when FINDDUPLICATE is reduced to FINDPOSITIVE, we need to start with m initial increments of -1 . Thus, when working with data streams directly, our algorithm needs a pre-processing stage taking time $\tilde{O}(m)$ for each such dot product computation. Alternatively, since the stream is of length $n > m$, we can spread this computation over n steps with negligible overhead per step.

Top level view of our algorithms Our algorithms share a common high-level structure, which we describe below. The algorithms have three main parts:

1. **Isolate:** This procedure generates a set S which is guaranteed to contain a dictator with probability $\Omega(\log m)$. This part does not read the input stream.
2. **Find:** This procedure finds a candidate dictator in a given S . Find(S) succeeds in finding the dictator if S has one, but otherwise might return an arbitrary element (possibly not in S).
3. **Verify:** Given a set S and a potential dictator i , this procedure accepts with high probability if i is indeed a dictator. If $f[i]$ is negative the procedure rejects with probability at least $\frac{1}{2}$.

We note that the notion of what a dictator is depends on the norm we are working with, hence the procedures Isolate, Find and Verify will also have to be defined accordingly.

3 Isolating A Dictator

DEFINITION 2. (r -DICTATORSHIP) Let $g : [m] \rightarrow \mathbb{R}^{\geq 0}$, $r \geq 1$ and $A \subseteq [m]$. We say that $S \subseteq [m]$ is an r -dictatorship with respect to (g, A) if there is an $i \in S \cap A$ such that $g[i] > r(\sum_{j \in S - \{i\}} g[j])$. We then refer to i as an r -dictator of S .

The following procedure (which is oblivious to g and A) is used to isolate an r -dictator with respect to (g, A) .

ALGORITHM 2. ISOLATE(D,L)

Output: $S \subseteq [m]$.

Step 1: Pick k uniformly from the set $\{0, 1, \dots, \lfloor \log L \rfloor\}$.

Step 2: Let $b = \langle b_1, b_2, \dots, b_m \rangle$ be a sequence of random pairwise independent bits, each taking the value 1 with probability $\frac{2^k}{DL}$. Return the subset $S \subseteq [m]$ whose characteristic vector is b .

To generate the set S , we need to generate at most m pairwise independent random bits, with a certain bias. It is well known [CG89] that this can be done using $O(\log m)$ truly random bits. Further, one can test whether $i \in S$ in time $O(\log m)$; we skip the details.

The following Lemma is our main Isolation Lemma. It is instructive to think of r, C as constants and $L = \text{poly}(n)$, in which case the Lemma states that if $\ell_1(g|_A) = \Omega(\ell_1(g))$, random sampling gives an r -dictatorship with probability $\Omega(1/\log n)$.

LEMMA 3.1. (Isolating a Dictator)

Let $g : [m] \rightarrow \mathbb{R}^{\geq 0}$, $[m] = A \cup [m] - A$, and L, C be positive integers such that

- (a) $\ell_1(g|_A) \leq L$, and
- (b) $\ell_1(g) \leq C\ell_1(g|_A)$.

Then Isolate($2Cr, L$) returns an r -dictatorship with respect to (g, A) probability at least $\frac{1}{8Cr(1+\lfloor \log L \rfloor)}$.

Proof: We write ℓ and $\ell(A)$ instead of $\ell_1(g)$ and $\ell_1(g|_A)$. For $j \in \{0, 1, \dots, \lfloor \log \ell(A) \rfloor\}$, let $N_j = \{a \in [m] : g[a] \geq 2^j\}$ and $n_j = |N_j|$; then a simple double-counting argument shows that

$$(3.1) \quad \ell(A) \leq \sum_{j=0}^{\lfloor \log \ell(A) \rfloor} 2^j n_j.$$

For $j \in \{0, 1, \dots, \lfloor \log \ell(A) \rfloor\}$, let $k_j = \left\lfloor \log \left(\frac{2^j}{\ell(A)} L \right) \right\rfloor$. Since $2^j \leq \ell(A)$, we have $k_j \in \{0, 1, \dots, \lfloor \log L \rfloor\}$; since $L \geq \ell(A)$, $\frac{2^j}{\ell(A)} L \leq 2^{k_j} \leq \frac{2^j}{\ell(A)} L$ and k_j is strictly increasing with j . In particular,

$$\begin{aligned} (3.2) \quad & \Pr[S \text{ has a dictator}] \\ & \geq \sum_{j=0}^{\lfloor \log \ell(A) \rfloor} \Pr[S \text{ has a dictator and } k = k_j] \\ & = \sum_{j=0}^{\lfloor \log \ell(A) \rfloor} \Pr[k = k_j] \cdot \Pr[S \text{ has a dictator} \mid k = k_j]. \end{aligned}$$

Let us bound the two factors on the right separately. By the definition of k , the first factor, $\Pr[k = k_j]$, is $\frac{1}{1 + \lfloor \log L \rfloor}$. To bound the second factor, we will show that for $j \in \{0, 1, \dots, \lfloor \log \ell(A) \rfloor\}$,

$$(3.3) \quad \Pr[S \text{ has a dictator} \mid k = k_j] \geq \frac{2^j n_j}{8Cr\ell(A)}.$$

Our lemma then follows by combining these bounds with (3.2). Equation 3.3 is proved by analyzing the probability that some element of N_j is a dictator. Fix $j \in \{0, 1, \dots, \lfloor \log \ell(A) \rfloor\}$. We have

$$\begin{aligned} & \Pr[S \text{ is an } r\text{-dictatorship} \mid k = k_j] \\ & \geq \sum_{a \in N_j} \Pr[a \text{ is an } r\text{-dictator in } S \mid k = k_j] \\ & \geq \sum_{a \in N_j} \Pr[a \in S \mid k = k_j] \times \\ (3.4) \quad & \Pr \left[\sum_{b \in S - \{a\}} |g[b]| < 2^j/r \mid a \in S, k = k_j \right] \end{aligned}$$

The second inequality holds since $a \in N_j$ hence $g[a] \geq 2^j$. The first factor on the right is precisely $\frac{2^{k_j}}{2CrL}$. To bound the second factor, fix $a \in [m]$ and note that since the bits b_i are pairwise independent, we have

$$\begin{aligned} & \mathbb{E} \left[\sum_{b \in S - \{a\}} |g[b]| \mid a \in S, k = k_j \right] \\ & = \mathbb{E} \left[\sum_{b \in S - \{a\}} |g[b]| \mid k = k_j \right] \\ & \leq \frac{\ell 2^{k_j}}{2CrL} \leq \frac{C\ell(A)}{2CrL} \frac{2^j}{\ell(A)} L \leq \frac{2^j}{2r}. \end{aligned}$$

Thus, by Markov's inequality,

$$\begin{aligned} & \Pr \left[\sum_{b \in S - \{a\}} |g[b]| \geq \frac{2^j}{r} \mid a \in S \text{ and } k = k_j \right] \leq \frac{1}{2} \\ \Rightarrow & \Pr \left[\sum_{b \in S - \{a\}} |g[b]| < \frac{2^j}{r} \mid a \in S \text{ and } k = k_j \right] \geq \frac{1}{2}. \end{aligned}$$

We now return to (3.4) with these bounds:

$$\begin{aligned} & \Pr[S \text{ has a } (g, A) \text{ dictator} \mid k = k_j] \\ & \geq \sum_{a \in N_j} \frac{2^{k_j}}{4CrL} \frac{1}{2} \geq \frac{2^j n_j}{8Cr\ell(A)}. \end{aligned}$$

This establishes (3.3) and completes the proof of the lemma. \blacksquare

4 FINDPOSITIVE in the ℓ_1 Norm

DEFINITION 3. (DICTATORSHIP) Let f be a frequency vector associated with the input stream and $S \subseteq [m]$. We say that f^S is a dictatorship if there exists $i \in S$ so that $f[i] > \sum_{j \in S: j \neq i} |f[j]|$.

The condition above ensures that $f^S(x) = x_i$.

Isolating a Dictatorship Define the function $g : [m] \rightarrow \mathbb{R}^{\geq 0}$, by $g[i] = |f[i]|$. Let A be the set of positive frequency elements $A = \{i : f[i] \geq 1\}$. So g satisfies the condition $\ell_1(g) < 2\ell_1(g|_A)$, hence we take $C = 2$. Since $\ell_1(f) \leq n$, we take $L = n$. Note that if S contains the 1-dictator i with respect to (g, A) , then $f^S(x)$ is the dictatorship of x_i . Applying Lemma 3.1, we conclude

COROLLARY 6. With probability $\Omega((\log n)^{-1})$, $\text{Isolate}(4, n)$ returns a set S such that f^S is a dictatorship.

Finding the Dictator Fix a binary encoding of $[m]$ using strings of length $\lceil \log m \rceil$ over the alphabet $\{\pm 1\}$. For $k = 1, 2, \dots, \lceil \log m \rceil$, let $w_k \in \{\pm 1\}^{\lceil \log m \rceil}$ be defined by

$w_k[i] = i$ -th coordinate in the binary representation of i , and let $\mathbf{w} = \langle w_1, \dots, w_{\lceil \log m \rceil} \rangle$. Let $\text{sgn}(f^S \cdot \mathbf{w}) = \langle \text{sgn}(f^S \cdot w_1), \dots, \text{sgn}(f^S \cdot w_{\lceil \log m \rceil}) \rangle$.

ALGORITHM 3. FIND(S)

Input: $S \subseteq [m]$.

Output: $i \in [m]$.

Return: $\text{sgn}(f^S \cdot \mathbf{w})$.

The following claim is immediate from the definitions.

LEMMA 4.1. If S contains the dictator i , then $\text{sgn}(f^S \cdot \mathbf{w})$ is the binary encoding of i .

Given a set S (as a pair) and an index $j \in \{1, \dots, \lceil \log m \rceil\}$, we can compute $f^S \cdot w_j$ in space $O(\log n)$. Thus $f^S \cdot \mathbf{w}$, and hence $\text{sgn}(f^S \cdot \mathbf{w})$ can be computed in space $O(\log m \log n)$, given the input stream. In the main algorithm we invoke $\text{Find}(S)$ on $O(\log n)$ sets S . The entire computation can be carried out in space $O((\log n)^2 \log m)$.

Verifying the Dictatorship We verify if i is indeed a dictator by estimating $E_x[f^S(x)x_i]$.

ALGORITHM 4. VERIFY(S, i)

Input: $S \subseteq [m], i \in [m]$.

Output: i or failure.

Step 1: If $i \notin S$, then return failure.

Step 2: Pick $x \in \{\pm 1\}^n$ at random. If $f^S(x) \neq x_i$, return failure.

LEMMA 4.2. Let f be a frequency vector. Let $i \in S$ be such that $f[i] \leq 0$. Then, for a uniformly random $x \in \{+1, -1\}^m$ $\Pr[f^S(x) = x_i] \leq \frac{1}{2}$.

This corresponds to the well-known fact that if the weight $f[i]$ given to a variable x_i in a halfspace is so that $f[i] \leq 0$, then $E_x[f(x)x_i] \leq 0$. The proof is omitted.

Verify can be implemented naively using $O(m)$ bits to generate and store the vector x , and then $O(\log n)$ bits to compute $f^S(x)$. In the main algorithm, we need to run Verify(S_j, i_j) $k = \lceil \log m \rceil$ times for $\ell = O(\log n)$ input pairs (S_j, i_j) . We can reuse the same strings x_1, \dots, x_k for all ℓ input pairs. We will now describe an implementation that uses space $O((\log n)^2 \log m)$.

We observe that we can use Nisan's generator [Nis92], generate all the k vectors x^i using $O((\log n)^2)$ random bits in space $O((\log n)^2)$. To see this, consider the following randomized computation (used only for the analysis).

Input: A frequency vector f and $\langle (S_j, i_j) : j = 1, 2, \dots, \ell \rangle$.

For $j = 1, 2, \dots, \ell$,

For $k = 1, 2, \dots, \lceil \log m \rceil$

If Verify(S_j, i_j) returns failure, skip to next j ;
return i_j .

Let x^i be the random vector used in the i -th invocation of Verify. In the naive implementation this would require n random bits for each invocation. However, in this sequential implementation, we don't need to store the random string; indeed, it is clear that the entire code can be implemented using space $O(\log n)$. Now the idea (following Indyk [Ind06]) is to use Nisan's generator instead. This generator uses $O((\log n)^2)$ space, and given an index $i \in [k]$ and $j \in [m]$, in $O((\log n)^2)$ space generates x_j^i . The distribution of the computation using vectors generated in this manner can be arranged to differ from the ideal distribution (when truly random bits are used) by at most, say, $\frac{1}{m}$. In the efficient implementation of the

main algorithm, we generate the vectors required in the various invocations of Verify in this pseudo-random manner, and conclude by comparing it to the code above that the output distribution remains essentially the same. Note however, that when the input is available as a stream, then we need to run the invocations of Verify in parallel, dedicating $O(\log n)$ bits for the dot product computation involved in each. Thus, Step 3 of the Main Algorithm can be implemented in space $O((\log n)^3)$.

4.1 The Final Algorithm Let us now put the pieces together to obtain the main algorithm.

ALGORITHM 5. FIND POSITIVE

Input: Input stream I = $\langle (i_1, c_1), (i_2, c_2), \dots, (i_n, c_n) \rangle$.

Step 1: Run Isolate($4, n$) $\ell = \theta(\log m)$ times and obtain sets S_1, S_2, \dots, S_ℓ .

Step 2: For $j = 1, 2, \dots, \ell$
 $i_j \leftarrow \text{Find}(S_j)$
For $k = 1, 2, \dots, \lceil \log m \rceil$
If Verify(S_j, i_j) returns failure,
skip to the next j ;
return Fail

THEOREM 7. With probability at least $\frac{3}{4}$, Algorithm 5 returns an index i with $f[i] > 0$. Further, the algorithm can be implemented with $O((\log n)^2 \log m)$ bits of space and $O((\log n)^2)$ random bits.

Proof: Each S_j obtained in Step 1 has a dictator with probability $\Omega(1/\log n)$ (see Corollary 6). Thus, the probability that no S_j has a dictator can be made (say) less than $\frac{1}{10}$ by taking $\ell = \theta(\log n)$ large enough. Let j_0 be the smallest index for which S_{j_0} has a dictator. In Step 2, i_{j_0} is set to this value. Thus (see Lemma 4.2), some index $j \leq j_0$ is returned by the algorithm. For some $j < j_0$, if $f[j] < 0$, then the probability that i_j is not rejected is at most $\frac{1}{m}$. Thus, with high probability an index with $f[i] > 0$ is returned by the algorithm.

To generate each S_j one requires $O(\log m)$ bits (see below). So, all ℓ copies can be generated using $O(\log n \log m)$ bits. Each invocation of Find requires $O(\log m \log n)$ bits, and thus Step 2 can be implemented in space $O((\log n)^2 \log m)$. We require $O(\log m \log n)$ parallel calls to the procedure Verify. Each invocation requires space $O(\log n)$, which gives the space bound. ■

4.2 Finding duplicates in shorter streams For any $C \geq 1$, we extend our algorithm for finding positives to work under the weaker promise that $\ell(f) < C\ell^+(f)$, using space $O(C(\log n)^2 \log m)$. Thus we can find positives as long as the vector f is not too *unbalanced*.

PROBLEM 4. FINDPOSITIVE(C): Given $C \geq 1$ and a stream of increments of length n over the alphabet $[m]$ with the promise that $\ell(f) < C\ell^+(f)$, find an $i \in [m]$ such that $f[i] > 0$.

The problem FINDPOSITIVE corresponds to taking $C = 2$. We invoke Isolate($2C, n$) and apply Lemma 3.1 with $r = 2, L = n$ and A the set of positive frequencies.

COROLLARY 8. With probability at least $\Omega((C \log n)^{-1})$, Isolate($2C, n$) returns a set S such that f^S is a dictatorship.

We generate $\ell = \Theta(C \log n)$ sets so that one of them is a dictatorship with constant probability, causing the space needed to increase by a factor of C . This gives the following theorem:

THEOREM 9. Given an instance of FINDPOSITIVE(C), Algorithm 5 (with the modifications described above) returns an index i with $f[i] > 0$ with probability at least $\frac{3}{4}$. Further, the algorithm runs in space $O(C(\log n)^2 \log m)$.

As an application, we show how to find duplicates in a stream of length $m - s$ using space $O(s(\log m)^3)$ if one exists, using a randomized one-pass algorithm.

PROBLEM 5. FINDDUPLICATE(n): Given an input stream $x_1 x_2 \dots x_n$ where $x_i \in [m]$, find a duplicate $a \in [m]$ if one exists, else return Fail.

THEOREM 10. There is a one-pass algorithm for FINDDUPLICATE($m - s$) that uses space $O(s(\log m)^3)$ and returns a duplicate with probability at least $\frac{3}{4}$ if one exists, and returns Fail with probability $\frac{3}{4}$ otherwise.

Proof: We use the same reduction as before: given $S = \langle s_1, s_2, \dots, s_n \rangle$, define a stream of increments

$$I = \langle (1, -1), (2, -1), \dots, (m, -1), \\ (s_1, 1), (s_2, 1), \dots, (s_n, 1) \rangle.$$

Observe that $|I| \leq 2m$. Let $C = s + 3$ and run the algorithm for FINDPOSITIVE(r).

It is easy to see that the algorithm is unlikely to return a number which is not a duplicate. So assume that X does contain a duplicate. Let the number of distinct symbols from $[m]$ that occur in X be $d \leq n - 1$. Then it follows that

$$\ell^+(f) = n - d, \ell^-(f) = m - d, \frac{\ell(f)}{\ell^+(f)} = \frac{m - d}{n - d} + 1.$$

Since $n < m$, this ratio is maximized when $d = n - 1$, and $\ell^+(f) = 1, \ell(f) = s + 2 < C\ell^+(f)$. So the algorithm will find a duplicate with probability at least $\frac{3}{4}$. ■

The dependence on s is essentially optimal:

LEMMA 11. Any randomized streaming algorithm for FINDDUPLICATE($m - s$) that makes a constant number of passes requires space $\Omega(s)$.

Proof: We reduce from two-player Set-Disjointness over a universe $[2s]$. It is well-known [KS92, Raz92] that the communication complexity of this problem is $\Omega(s)$, where we may assume that Alice and Bob hold sets A and B each of size $s/2$. We define the input stream I to consist of A , followed by B , followed $2s + 1, \dots, n + s$. Thus I is of length n , and $m = n + s$.

Clearly, I contains a repeat iff Alice and Bob's sets intersect. If there is a space S algorithm for FINDDUPLICATE($m - s$), it can be used to find this intersection: Alice runs it on A and passes the state of the algorithm to Bob who runs it on the rest of the stream. If there is a duplicate, it will be found with good probability. If not, the algorithm can return an arbitrary answer $i \in [m]$. Bob checks that $i \in B$ and sends it to Alice to confirm that $i \in A$. Thus an q -pass algorithm for FINDDUPLICATE($m - s$) yields an $q + 1$ -round protocol for Set-Disjointness on $[2s]$. ■

5 Finding Positives in ℓ_2

Our algorithm for the ℓ_2 -version relies on the notion of a Near-Dictator.

DEFINITION 4. (NEAR-DICTATOR) Let f be a frequency vector associated with the input stream and $S \subseteq [m]$. We say that $i \in S$ is a near-dictator for S if $f[i]^2 \geq 10(\sum_{j \in S \setminus \{i\}} |f[j]|^2)$.

While a Near-Dictator need not be a Dictator by our definition, but the following Lemma shows that $f^S(x)$ is close to being the Dictatorship of x_i .

LEMMA 12. If $i \in S$ is a Near-Dictator for S , then $\Pr_x[f^S(x) = x_i] \geq 0.9$. Further this is true even when $x \in \{\pm 1\}^n$ is drawn from a pairwise independent distribution.

Proof: Consider the random variable $W = \sum_{j \in S \setminus \{i\}} f[j]x_j$, where the x_j s are pairwise independent. Then

$$\begin{aligned} \mathbb{E}[W^2] &= \sum_{j \in S \setminus \{i\}} f[j]^2 \\ \Rightarrow \Pr[|W| \geq f[i]] &\leq \frac{\mathbb{E}[W^2]}{f[i]^2} \leq \frac{1}{10}. \end{aligned}$$

If the event $|W| < f[i]$ occurs, then $f^S(x) = x_i$, hence the claim is proved. ■

Isolating a Near-Dictator Define the function $g : [m] \rightarrow \mathbb{R}^{\geq 0}$ by $g[i] = f[i]^2$. As before, let A be the set of positive frequencies. Note that the $\ell_1(g|_A) \leq \ell_1(g) \leq \ell_2(f)^2 \leq n^2$, so we can take $L \leq n^2$. The condition $\ell_2^+(f) > \ell_2^-(f)$ ensure that $\ell_1(g|_A) \leq 2\ell_1(g)$ so we can take $C \leq 2$. If the set S is a 10-dictatorship with respect to (g, A) , then it follows that f^S is a Near-dictatorship. So we set $r = 10$ and apply Lemma 3.1 to get:

COROLLARY 13. *Let f be a frequency vector such that $\ell_2^-(f) < \ell_2^+(f)$. With probability $\Omega(\log n)$, $\text{Isolate}(40, n^2)$ returns a set S so that f^S is a Near-Dictatorship.*

Finding a Near-dictator Let $k = \lceil \log(m+1) \rceil$. Let us associate each $i \in [m]$ with a distinct non-empty subset $I \subset [k]$. For $y \in \{\pm 1\}^k$, and $I \subseteq [k]$, let $\chi_I(y)$ denote the parity of the bits in the subset I . Thus for each $i \in [m]$, the Hadamard encoding is given by $\{\chi_I(y)\}_{y \in \{\pm 1\}^k}$.

One can also view the Hadamard encoding as a map $\text{Had} : \{\pm 1\}^k \rightarrow \{\pm 1\}^m$ given by $\text{Had}(y) = \{\chi_I(y)\}_{I \subseteq [k]}$. Let us define $g : \{\pm 1\}^k \rightarrow \{\pm 1\}$ by $g(y) = f^S(\text{Had}(y))$. It is well known and easy to see that over a random choice of $y \in \{\pm 1\}^k$, the bits of $\text{Had}(y) \in \{\pm 1\}^m$ are pairwise independent, so by Lemma 12, $\Pr_y[g(y) = \chi_I(y)] \geq 0.9$. Thus if $i \in S$ is a near-dictator, the function $g(y)$ is at Hamming distance 0.1 from the Hadamard encoding of i . So we can view $g(y)$ as $\chi_I(y)$ corrupted by adversarial noise of rate 0.1. Our goal is to recover i by querying g at $\text{poly}(k) = \text{poly}(\log m)$ vectors y . This is done using a unique-decoder for Hadamard codes. For $y \in \{\pm 1\}^k$, let $B(y)$ denote the Hamming ball of radius 1 around it.

PROPOSITION 14. *There exists an algorithm Hadamard-Decode that takes as input the values of $g : \{\pm 1\}^k \rightarrow \{\pm 1\}$ at points in $C = \cup_{i=1}^k B(y_i)$ where the y_i 's are chosen randomly in $\{\pm 1\}^k$. If there exists I so that $\Pr_y[g(y) = \chi_I(y)] \geq 0.9$, then the algorithm returns I with probability $1 - c^k$ for some $c < 1$.*

The algorithm and its analysis are standard, so we skip the proof. We now describe the procedure ℓ_2 -Find.

ALGORITHM 6. ℓ_2 -FIND(S)

Input: A non-empty subset $S \subseteq [m]$.

Output: $i \in [m]$.

Sample y_1, \dots, y_k .

For every $y \in C = \cup_{i=1}^k B(y_i)$,

Set $x = \text{Had}(y)$.

Compute $g(y) = f^S(x)$.

Run Hadamard-Decode on $\{g(y)\}_{y \in T}$, and return the result.

Verifying Near-Dictatorships If $i \in S$ is a Near-dictator, then $\Pr_x[f^S(x) = x_i] \geq 0.9$. Whereas if $f[i] \leq 0$, then $\Pr_x[f^S(x) = x_i] \leq 0.5$. Thus we test whether $f^S(x) = x_i$ for many randomly chosen x 's, and accept if it holds for at least 0.8 of the them. The x s are generated using Nisan's generator as opposed to uniformly at random.

We now describe the Algorithm for ℓ_2 -FINDPOSITIVE.

ALGORITHM 7. ℓ_2 -FIND POSITIVE

Input: $I = \langle (i_1, c_1), (i_2, c_2), \dots, (i_n, c_n) \rangle$.

Step 1: Run $\text{Isolate}(40, n^2)$ $\ell = \theta(\log n)$ times and obtain sets S_1, S_2, \dots, S_ℓ .

Step 2: For $j = 1, 2, \dots, \ell$

$i_j \leftarrow \ell_2\text{-Find}(S_j)$

If $\Pr_x[\text{Verify}(S_j, i_j) \text{ accepts}] > 0.8$,

Accept and halt.

return Fail.

We estimate the probability that Verify accepts using $O(\log m)$ strings from Nisan's generator. This proves the following theorem:

THEOREM 15. *Given an input stream I over $[m]$ with frequency vector f where $\ell_2^-(f) < \ell_2^+(f)$, Algorithm 7 returns an index i with $f[i] > 0$ with probability at least $\frac{3}{4}$. Further, the algorithm can be implemented in space $O((\log n \log m)^2)$ with $O((\log n)^2)$ random bits.*

Comparison between the ℓ_1 and ℓ_2 Algorithms

While in general, the requirement that $\ell_1^+(f) > \ell_1^-(f)$ and $\ell_2^+(f) > \ell_2^-(f)$ are incomparable, for the instances generated by our reduction from FINDDUPLICATE, the ℓ_2 requirement is always weaker. Let S be the input instance of FINDDUPLICATE, with frequency vector f . Let $A \subset [m]$ be the set of repeated elements where $f[i] \geq 2$ and let $B \subset [m]$ be the set of omitted elements where $f[i] = 0$. Then if we run the reduction to FINDPOSITIVE, the ℓ_1 condition translates to $\sum_{i \in A} (f[i] - 1) > |B|$ whereas the ℓ_2 condition translates to $\sum_{i \in A} (f[i] - 1)^2 > |B|$ which is weaker.

5.1 Finding positives is hard in higher norms

We show that for $p > 2$, ℓ_p -FINDPOSITIVE is intractable for small space algorithms from multiplayer

Set-Disjointness. In the t -player set-disjointness problem, players P_1, \dots, P_t each holds subsets S_1, \dots, S_t of $[m]$, with the promise that the sets are either pairwise disjoint, or that they all intersect in a unique element $i \in [m]$. Chakrabarti *et al.* show that the total communication complexity of this problem is $\Omega(\frac{m}{t \log m})$ [CKS03]. We use this result to prove Theorem 3.

Proof: Let $t = m^{\frac{1}{p}} + 1$. Consider the input stream consisting of $\langle(1, -1), (2, -1), \dots, (m, -1)\rangle$ followed by the sets S_1, \dots, S_t presented as increments.

We claim that a c -pass algorithm for ℓ_p -FINDPOSITIVE gives an $c + 1$ -pass protocol for t -player set-disjointness, with the same space requirements. Assuming that the sets intersect at $[i]$, we have $f[i] \geq t - 1$, while $f[j] \in \{0, -1\}$ for all $j \neq i$. Thus we have $\ell_p^+(f) = t - 1 = m^{1/p}$ whereas $\ell_p^-(f) \leq (m - 1)^{1/p}$. Thus in this case an algorithm for ℓ_p -FINDPOSITIVE must return the element i . In the No case $\ell_p^+(f) = 0$, so the algorithm may return nothing or an arbitrary element $i \in [m]$. It is easy to verify if i is indeed an duplicate using an extra round.

Since the total communication complexity of set-disjointness is $\Omega(\frac{m}{t \log m})$, some player must send a message of size $\Omega(\frac{m}{ct^2 \log m})$ in a $c + 1$ round protocol, which implies the claim. ■

6 Lower bounds for deterministic algorithms

We present our arguments using the language of finite state machines. Q will denote the set of states of the finite state machine. If the alphabet is $[m]$, each state $q \in Q$ has m transitions originating at it, each transition labeled by a different symbol of $[m]$. On input $X = \langle x_1, x_2, \dots, x_n \rangle$, the machine passes through a sequence of states $q_0, q_1(X), \dots, q_n(X)$, where q_0 is the start state (independent of X), and for $i = 1, 2, \dots, n$, the unique transition out of q_{i-1} labeled x_i leads to q_i . The answer returned by the machine on input X is obtained by applying a function $\text{ans} : Q \rightarrow m \cup \{?\}$ to the final state $q_n(X)$.

PROPOSITION 6.1. *If there is a space s algorithm that finds duplicates in a data stream of length n over the alphabet $[m]$, then there is a finite state machine $\mathcal{A} = (Q, \delta, \text{ans})$ over the alphabet $[m]$ such that $|Q| \leq 2^s$, and for all $X \in [m]^n$, $\text{ans}(q_n(X))$ appears twice in X .*

Non-uniformity. We will also consider non-uniform algorithms, which we think of as algorithms equipped with a clock. When such an algorithm for processing a stream of n symbols is modeled by a finite state machine, it will be convenient to write the set of states as $S \times \{0, 1, \dots, n\}$; then each transition out of

a state of the form (q, i) leads to a state of the form $(q', i + 1)$. The rest is the same as before.

PROPOSITION 6.2. *If there is a space s algorithm with a clock that finds duplicates in a data stream of length n over the alphabet $[m]$, then there is a finite state machine $\mathcal{A} = (S \times \{0, 1, \dots, n\}, \delta, \text{ans})$ over $[m]$ such that $|S| \leq 2^s$, and for all $X \in [m]^n$, $\text{ans}(q_n(X))$ appears twice in X .*

These models are different from the one-pass model considered by Tarui [Tar07].

THEOREM 16. *If $\mathcal{A} = (Q = S \times [n], \delta, \text{ans})$ is a finite state machine with a clock for finding duplicates in data streams of length n over the alphabet $[m]$, then $|S| \geq m - \log n$.*

This bound leaves open the possibility of a small-space algorithm for $n \geq 2^m$. In the uniform case, we can prove a strong lower bound for all n .

THEOREM 17. *Let $\mathcal{A} = (Q, \delta, \text{ans})$ be a deterministic finite state machine over the alphabet $[m]$. Suppose there exists n_0 so that for all streams of length at least n_0 , the algorithm returns a duplicate. Then, $|Q| \geq 2^{\frac{n_0}{2} - 1}$.*

The proofs are omitted due to lack of space, they use the same technique as Tarui [Tar07] where we associate a vector to each state in Q .

7 A Non-Uniform Algorithm

We present a non-uniform branching program to find duplicates that uses space $\log m + O(1)$. For this algorithm it will be convenient to think of the input string as $X = x_0 x_1 \dots x_{n-1}$ (the indices starts at 0), where $n = 2^m$. We think of the read-only clock c as a vector of m bits, which is initialized to 0^m and holds the index i of the current symbol x_i being read: $c[1]$ is the least significant bit of the binary representation of i , and $c[m]$ its most significant bit. The algorithm has a $\lceil \log m \rceil$ -bit register ‘ a ’ that holds an element of $[m]$.

ALGORITHM 8. DETERMINISTIC ALGORITHM

1. Read the first input symbol (at this point $c = 0^n$).
 2. Move the current input symbol into the register a .
 3. If $c[a] = 1$, terminate with output a .
 4. Else, read the input until $c[a] = 1$, and goto 2.
-

LEMMA 18. *If the control arrives at Step 4, then it does eventually return to Step 2 (before the input runs out). If the algorithm terminates in Step 3, the register ‘a’ holds a duplicate element.*

Proof: When the control arrives at Step 4, $c[a] = 0$, and the loop reads input symbols until $c[a] = 1$. Since the binary representation of $n - 1$ is 1^n , in the end $c[a] = 1$, and the control must eventually return to Step 2. This establishes the first statement.

To justify the second statement, we first establish the following claim: When control reaches Step 2, if $c[x] = 1$ for some $x \in [m]$, then the symbol x must have appeared at least once before.

This claim is vacuously true initially (for $c = 0^n$). Assume the claim holds at some point when the control reaches Step 2; we wish to show that it holds when the control next returns to Step 2. The value of the clock does not change in Steps 2 and 3, so we just need to verify that the invariant is maintained in Step 4. At the beginning of Step 4, we have $c[a] = 0$ and the last symbol read is a . During Step 4, the bits $c[b]$ for $b > a$ are not disturbed, $c[a]$ is set to 1, and the bits $c[b]$ for $b < a$ are set to 0. Thus, the claim holds when the control next returns to Step 2. This immediately implies the second statement. ■

THEOREM 19. *Algorithm 8 uses $\log m + O(1)$ bits and finds a duplicate in a data stream of length 2^m .*

Proof: Note that each execution of Step 4 causes at least one more symbol of the input to be read. So, the algorithm either terminates in Step 3 or runs out of input in Step 4. By the first part of Lemma 18, the algorithm, in fact, terminates in Step 3. By the second part, the value output is indeed a duplicate. ■

It is possible to tradeoff length of the stream and storage space:

THEOREM 20. *For all $k \leq m$, there is a deterministic algorithm that find a duplicate in a data stream of length $k + 2^{m-k}$ using space $k + \lceil \log(m - k) \rceil$.*

Proof: We run Algorithm 8 with the alphabet restricted to the first $m - k$ symbols, and use k bits to record occurrences of the last k symbols. ■

The lower bound in Theorem 16 implies that any algorithm that finds duplicates in a data stream of length at least $k + 2^{m-k}$ requires at least $m - \log(k + 2^{m-k}) \geq k - \log k$ space.

Acknowledgment Jaikumar Radhakrishnan is grateful to Jun Tarui for introducing him to this problem,

and to Omer Reingold for useful discussions. Parikshit Gopalan would like to thank the USCIS for making his extended stay in India possible.

References

- [AMS99] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *JCSS*, 58(1):137–147, 1999.
- [BM03] M. Bilenko and R. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *Proc. 9th ACM SIGKDD*, pages 39–48, 2003.
- [BNW03] A. Z. Broder, M. Najork, and J. L. Wiener. Efficient URL caching for world wide web crawling. In *WWW*, 2003.
- [CG89] B. Chor and O. Goldreich. On the power of two-point based sampling. *J. Complexity*, 5(1):96–106, (1989).
- [Cha08] M. Charikar. Top- k frequent item maintenance over streams. *Survey article*, 2008.
- [CKS03] A. Chakrabarti, S. Khot, and X. Sun. Near-optimal lower bounds on the multiparty communication complexity of set-disjointness. In *Proc. 18th IEEE Conference on Computational Complexity*, pages 107–117, 2003.
- [DR06] F. Deng and D. Rafiei. Approximately detecting duplicates for streaming data using stable Bloom filters. In *SIGMOD Conference*, pages 25–36, 2006.
- [For05] L. Fortnow. Finding duplicates: Post on the Computational Complexity weblog, March 4th 2005.
- [GCM05] V. Ganti, S. Chaudhuri, and R. Motwani. Robust identification of fuzzy duplicates. In *Proc. 21st IEEE ICDE*, 2005.
- [Ind06] P. Indyk. Stable distributions, pseudorandom generators, embeddings and data stream computation. *J. ACM*, 53(3):307–323, 2006.
- [KN97] E. Kushilevitz and N. Nisan. *Communication Complexity*. Cambridge University Press, 1997.
- [KS92] B. Kalyanasundaram and G. Schnitger. The probabilistic communication complexity of set intersection. *SIAM Journal on Discrete Math*, 5(5):545–557, 1992.
- [MAA05] A. Metwally, D. Agrawal, and A. El Abbadi. Duplicate detection in click streams. In *WWW*, pages 12–21, 2005.
- [Mut05] S. Muthukrishnan. *Data Streams: Algorithms and Applications*. Now Publishers Inc., 2005.
- [Nis92] N. Nisan. Pseudorandom generators for space-bounded computation. *Combinatorica*, 12(4):449–461, 1992.
- [Raz92] A. A. Razborov. On the distributional complexity of disjointness. *Theoretical Computer Science*, 106(2):385–390, 1992.
- [Tar07] J. Tarui. Finding a duplicate and a missing item in a stream. In *TAMC*, pages 128–135, 2007.