Acknowledgements

These lecture notes began as rough scribe notes for a Fall 2009 offering of the course “Data Stream Algorithms” at Dartmouth College. The initial scribe notes were prepared mostly by students enrolled in the course in 2009. Subsequently, during a Fall 2011 offering of the course, I edited the notes heavily, bringing them into presentable form, with the aim being to create a resource for students and other teachers of this material.

I would like to acknowledge the initial effort by the 2009 students that got these notes started: Radhika Bhasin, Andrew Cherne, Robin Chhetri, Joe Cooley, Jon Denning, Alina Djamankulova, Ryan Kingston, Ranganath Kondapally, Adrian Kostrubiak, Konstantin Kutzkow, Aarathi Prasad, Priya Natarajan, and Zhenghui Wang. Other significant contributions were made by 2015 students Sagar Kale and Suman Bera.
## Contents

0 Preliminaries: The Data Stream Model ................................................. 4
  0.1 The Basic Setup ........................................................................ 4
  0.2 The Quality of an Algorithm’s Answer ........................................ 4
  0.3 Variations of the Basic Setup ....................................................... 5

1 Finding Frequent Items Deterministically ........................................ 6
  1.1 The Problem ............................................................................ 6
  1.2 Frequency Estimation: The Misra–Gries Algorithm ....................... 6
  1.3 Analysis of the Algorithm ......................................................... 7
  1.4 Finding the Frequent Items ....................................................... 8
  Exercises ....................................................................................... 8

2 Estimating the Number of Distinct Elements .................................... 9
  2.1 The Problem ............................................................................ 9
  2.2 The Tidemark Algorithm ........................................................... 9
  2.3 The Quality of the Algorithm’s Estimate ..................................... 10
  2.4 The Median Trick ................................................................. 11
  Exercises ....................................................................................... 11

3 A Better Estimate for Distinct Elements ......................................... 13
  3.1 The Problem ........................................................................... 13
  3.2 The BJKST Algorithm ............................................................. 13
  3.3 Analysis: Space Complexity .................................................... 14
  3.4 Analysis: The Quality of the Estimate ..................................... 14
  3.5 Optimality ............................................................................. 15
  Exercises ....................................................................................... 15

4 Approximate Counting .................................................................... 17
  4.1 The Problem ........................................................................... 17
  4.2 The Algorithm ....................................................................... 17
4.3 The Quality of the Estimate .......................................................... 18
4.4 The Median-of-Means Improvement .............................................. 19
Exercises ......................................................................................... 20

5 Finding Frequent Items via (Linear) Sketching .................................. 21
5.1 The Problem .................................................................................. 21
5.2 Sketches and Linear Sketches ......................................................... 21
5.3 The CountSketch ........................................................................... 22
  5.3.1 The Quality of the Basic Sketch’s Estimate .......................... 22
  5.3.2 The Final Sketch .................................................................... 23
5.4 The Count-Min Sketch .................................................................... 24
  5.4.1 The Quality of the Algorithm’s Estimate .......................... 24
5.5 Comparison of Frequency Estimation Methods ............................ 25
Exercises ......................................................................................... 26

6 Estimating Frequency Moments ...................................................... 27
6.1 Background and Motivation ............................................................ 27
6.2 The (Basic) AMS Estimator for $F_k$ ............................................ 27
6.3 Analysis of the Basic Estimator ...................................................... 28
6.4 The Final Estimator and Space Bound .......................................... 30
  6.4.1 The Soft-O Notation .............................................................. 30

7 The Tug-of-War Sketch .................................................................... 31
7.1 The Basic Sketch ........................................................................... 31
  7.1.1 The Quality of the Estimate .................................................. 32
7.2 The Final Sketch ........................................................................... 32
  7.2.1 A Geometric Interpretation ................................................. 32
Exercises ......................................................................................... 33

8 Estimating Norms Using Stable Distributions .................................. 34
8.1 A Different $\ell_2$ Algorithm ............................................................ 34
8.2 Stable Distributions ..................................................................... 35
8.3 The Median of a Distribution and its Estimation .......................... 35
8.4 The Accuracy of the Estimate ......................................................... 36
8.5 Annoying Technical Details ........................................................... 37
  8.5.1 An Open Question ................................................................ 37
Preliminaries: The Data Stream Model

0.1 The Basic Setup

In this course, we are concerned with algorithms that compute some function of a massively long input stream $\sigma$. In the most basic model (which we shall call the *vanilla streaming model*), this is formalized as a sequence $\sigma = \langle a_1, a_2, \ldots, a_m \rangle$, where the elements of the sequence (called *tokens*) are drawn from the universe $[n] := \{1, 2, \ldots, n\}$. Note the two important size parameters: the stream length, $m$, and the universe size, $n$. If you read the literature in the area, you will notice that some authors interchange these two symbols. In this course, we shall consistently use $m$ and $n$ as we have just defined them.

Our central goal will be to process the input stream using a small amount of *space* $s$, i.e., to use $s$ bits of random-access working memory. Since $m$ and $n$ are to be thought of as “huge,” we want to make $s$ much smaller than these; specifically, we want $s$ to be *sublinear* in both $m$ and $n$. In symbols, we want

$$s = o\left(\min\{m, n\}\right).$$

The holy grail is to achieve

$$s = O\left(\log m + \log n\right),$$

because this amount of space is what we need to store a constant number of tokens from the stream and a constant number of counters that can count up to the length of the stream. Sometimes we can only come close and achieve a space bound of the form $s = \text{polylog}(\min\{m, n\})$, where $f(n) = \text{polylog}(g(n))$ means that there exists a constant $c > 0$ such that $f(n) = O((\log g(n))^c)$.

The reason for calling the input a stream is that we are only allowed to access the input in “streaming fashion.” That is, we do not have random access to the tokens and we can only scan the sequence in the given order. We do consider algorithms that make $p$ *passes* over the stream, for some “small” integer $p$, keeping in mind that the holy grail is to achieve $p = 1$. As we shall see, in our first few algorithms, we will be able to do quite a bit in just one pass.

0.2 The Quality of an Algorithm’s Answer

The function we wish to compute—$\phi(\sigma)$, say—is often real-valued. We shall typically seek to compute only an *estimate* or *approximation* of the true value of $\phi(\sigma)$, because many basic functions can provably not be computed exactly using sublinear space. For the same reason, we shall often allow *randomized* algorithms than may err with some small, but controllable, probability. This motivates the following basic definition.

**Definition 0.2.1.** Let $\mathcal{A}(\sigma)$ denote the output of a randomized streaming algorithm $\mathcal{A}$ on input $\sigma$; note that this is a random variable. Let $\phi$ be the function that $\mathcal{A}$ is supposed to compute. We say that the algorithm $(\epsilon, \delta)$-estimates $\phi$ if
we have
\[
\Pr \left[ \left| \frac{\mathcal{A}(\sigma)}{\phi(\sigma)} - 1 \right| > \epsilon \right] \leq \delta.
\]

Notice that the above definition insists on a multiplicative approximation. This is sometimes too strong a condition when the value of \( \phi(\sigma) \) can be close to, or equal to, zero. Therefore, for some problems, we might instead seek an additive approximation, as defined below.

**Definition 0.2.2.** In the above setup, the algorithm \( \mathcal{A} \) is said to \((\epsilon, \delta)^+\)-estimate \( \phi \) if we have
\[
\Pr \left[ \| \mathcal{A}(\sigma) - \phi(\sigma) \| > \epsilon \right] \leq \delta.
\]

We have mentioned that certain things are *provably* impossible in sublinear space. Later in the course, we shall study how to prove such impossibility results. Such impossibility results, also called *lower bounds*, are a rich field of study in their own right.

### 0.3 Variations of the Basic Setup

Quite often, the function we are interested in computing is some statistical property of the *multiset* of items in the input stream \( \sigma \). This multiset can be represented by a frequency vector \( \mathbf{f} = (f_1, f_2, \ldots, f_n) \), where
\[
f_j = |\{i : a_i = j\}| = \text{number of occurrences of } j \text{ in } \sigma.
\]

In other words, \( \sigma \) implicitly defines this vector \( \mathbf{f} \), and we are then interested in computing some function of the form \( \Phi(\mathbf{f}) \). While processing the stream, when we scan a token \( j \in [n] \), the effect is to increment the frequency \( f_j \). Thus, \( \sigma \) can be thought of as a sequence of *update instructions*, updating the vector \( \mathbf{f} \).

With this in mind, it is interesting to consider more general updates to \( \mathbf{f} \): for instance, what if items could both “arrive” and “depart” from our multiset, i.e., if the frequencies \( f_j \) could be both incremented and decremented, and by variable amounts? This leads us to the *turnstile model*, in which the tokens in \( \sigma \) belong to \([n] \times \{-L, \ldots, L\}\), interpreted as follows:

Upon receiving token \( a_i = (j, c) \), update \( f_j \leftarrow f_j + c \).

Naturally, the vector \( \mathbf{f} \) is assumed to start out at \( \mathbf{0} \). In this generalized model, it is natural to change the role of the parameter \( m \): instead of the stream’s length, it will denote the maximum number of items in the multiset at any point of time. More formally, we require that, at all times, we have
\[
\|\mathbf{f}\|_1 = |f_1| + \cdots + |f_n| \leq m.
\]

A special case of the turnstile model, that is sometimes important to consider, is the *strict turnstile model*, in which we assume that \( \mathbf{f} \geq \mathbf{0} \) at all times. A further special case is the *cash register model*, where we only allow positive updates: i.e., we require that every update \( (j, c) \) have \( c > 0 \).
Finding Frequent Items Deterministically

Our study of data stream algorithms begins with statistical problems: the input stream describes a multiset of items and we would like to be able to estimate certain statistics of the empirical frequency distribution (see Section 0.3). We shall study a number of such problems over the next several units. In this unit, we consider one such problem that admits a particularly simple and deterministic (yet subtle and powerful) algorithm.

1.1 The Problem

We are in the vanilla streaming model. We have a stream \( \sigma = (a_1, \ldots, a_n) \), with each \( a_i \in [n] \), and this implicitly defines a frequency vector \( f = (f_1, \ldots, f_n) \). Note that \( f_1 + \cdots + f_n = m \).

In the MAJORITY problem, our task is as follows. If \( \exists j : f_j > m/2 \), then output \( j \), otherwise, output “⊥”.

This can be generalized to the FREQUENT problem, with parameter \( k \), as follows: output the set \( \{ j : f_j > m/k \} \).

In this unit, we shall limit ourselves to deterministic algorithms for this problem. If we further limit ourselves to one-pass algorithms, even the simpler problem, MAJORITY, provably requires \( \Omega(\min\{m, n\}) \) space. However, we shall soon give a one-pass algorithm—the Misra–Gries Algorithm [MG82]—that solves the related problem of estimating the frequencies \( f_j \). As we shall see,

1. the properties of Misra–Gries are interesting in and of themselves, and
2. it is easy to extend Misra–Gries, using a second pass, to then solve the FREQUENT problem.

Thus, we now turn to the FREQUENCY-ESTIMATION problem. The task is to process \( \sigma \) to produce a data structure that can provide an estimate \( \hat{f}_a \) for the frequency \( f_a \) of a given token \( a \in [n] \). Note that the query \( a \) is given to us only after we have processed \( \sigma \). This problem can also be thought of as querying the frequency vector \( f \) at the point \( a \), so it is also known as the POINT-QUERY problem.

1.2 Frequency Estimation: The Misra–Gries Algorithm

As with all one-pass data stream algorithms, we shall have an initialization section, executed before we see the stream, a processing section, executed each time we see a token, and an output section, where we answer question(s) about the stream, perhaps in response to a given query.

This algorithm (see Algorithm 1) uses a parameter \( k \) that controls the quality of the answers it gives. (Looking ahead: to solve the FREQUENT problem with parameter \( k \), we shall run the Misra–Gries algorithm with parameter \( k \).) It maintains an associative array, \( A \), whose keys are tokens seen in the stream, and whose values are counters associated with these tokens. We keep at most \( k - 1 \) counters at any time.
Algorithm 1 The Misra–Gries frequency estimation algorithm

Initialize:
1: \( A \leftarrow \) (empty associative array)

Process (token \( j \)) :
2: if \( j \in \text{keys}(A) \) then
3: \( A[j] \leftarrow A[j] + 1 \)
4: else if |\text{keys}(A)| < \( k - 1 \) then
5: \( A[j] \leftarrow 1 \)
6: else
7: for all \( \ell \in \text{keys}(A) \) do
8: \( A[\ell] \leftarrow A[\ell] - 1 \)
9: if \( A[\ell] = 0 \) then remove \( \ell \) from \( A \)

Output (query \( a \)) :
10: if \( a \in \text{keys}(A) \) then report \( \hat{f}_a = A[a] \) else report \( \hat{f}_a = 0 \)

1.3 Analysis of the Algorithm

To process each token quickly, we could maintain the associative array \( A \) using a balanced binary search tree. Each key requires \( \lceil \log n \rceil \) bits to store and each value requires at most \( \lceil \log m \rceil \) bits. Since there are at most \( k - 1 \) key/value pairs in \( A \) at any time, the total space required is \( O(k(\log m + \log n)) \).

We now analyze the quality of the algorithm’s output.

Pretend that \( A \) consists of \( n \) key/value pairs, with \( A[j] = 0 \) whenever \( j \) is not actually stored in \( A \) by the algorithm. Consider the increments and decrements to \( A[j] \)'s as the stream comes in. For bookkeeping, pretend that upon entering the loop at line 7, \( A[j] \) is incremented from 0 to 1, and then immediately decremented back to 0. Further, noting that each counter \( A[j] \) corresponds to a set of occurrences of \( j \) in the input stream, consider a variant of the algorithm (see Algorithm 2) that explicitly maintains this set. Of course, actually doing so is horribly space-inefficient!

Algorithm 2 Thought-experiment algorithm for analysis of Misra–Gries

Initialize:
1: \( B \leftarrow \) (empty associative array)

Process (stream-position \( i \), token \( j \)) :
2: if \( j \in \text{keys}(B) \) then
3: \( B[j] \leftarrow B[j] \cup \{i\} \)
4: else
5: \( B[j] \leftarrow \{i\} \)
6: if |\text{keys}(B)| = \( k \) then
7: for all \( \ell \in \text{keys}(B) \) do
8: \( B[\ell] \leftarrow B[\ell] \setminus \min(B[\ell]) \) \( \triangleright \) forget earliest occurrence of \( \ell \)
9: if \( B[\ell] = \emptyset \) then remove \( \ell \) from \( B \)

Output (query \( a \)) :
10: if \( a \in \text{keys}(B) \) then report \( \hat{f}_a = |B[a]| \) else report \( \hat{f}_a = 0 \)

Notice that after each token is processed, each \( A[j] \) is precisely the cardinality of the corresponding set \( B[j] \). The increments to \( A[j] \) (including the aforementioned pretend ones) correspond precisely to the occurrences of \( j \) in the stream. Thus, \( \hat{f}_j \leq f_j \).

On the other hand, decrements to counters (including the pretend ones) occur in sets of \( k \). To be precise, referring to Algorithm 2, whenever the sets \( B[\ell] \) are shrunk, exactly \( k \) of the sets shrink by one element each and the removed
elements are \( k \) distinct stream positions. Focusing on a particular item \( j \), each decrement of \( A[j] \) is “witnessed” by a collection of \( k \) distinct stream positions. Since the stream length is \( m \), there can be at most \( m/k \) such decrements. Therefore, \( \hat{f}_j \geq f_j - m/k \). Putting these together we have the following theorem.

**Theorem 1.3.1.** The Misra–Gries algorithm with parameter \( k \) uses one pass and \( O(k(\log m + \log n)) \) bits of space, and provides, for any token \( j \), an estimate \( \hat{f}_j \) satisfying

\[
f_j - m/k \leq \hat{f}_j \leq f_j.
\]

### 1.4 Finding the Frequent Items

Using the Misra–Gries algorithm, we can now easily solve the FREQUENT problem in one additional pass. By the above theorem, if some token \( j \) has \( f_j > m/k \), then its corresponding counter \( A[j] \) will be positive at the end of the Misra–Gries pass over the stream, i.e., \( j \) will be in \( \text{keys}(A) \). Thus, we can make a second pass over the input stream, counting exactly the frequencies \( f_j \) for all \( j \in \text{keys}(A) \), and then output the desired set of items.

Alternatively, if limited to a single pass, we can solve FREQUENT in an approximate sense: we may end up outputting items that are below (but not too far below) the frequency threshold. We explore this in the exercises.

## Exercises

### 1-1

Let \( \hat{m} \) be the sum of all counters maintained by the Misra–Gries algorithm after it has processed an input stream, i.e., \( \hat{m} = \sum_{\ell \in \text{keys}(A)} A[\ell] \). Prove that the bound in Theorem 1.3.1 can be sharpened to

\[
f_j - m - \hat{m}/k \leq \hat{f}_j \leq f_j.
\]

### 1-2

Items that occur with high frequency in a dataset are sometimes called heavy hitters. Accordingly, let us defined the HEAVY-HITTERS problem, with real parameter \( \varepsilon > 0 \), as follows. The input is a stream \( \sigma \). Let \( m, n, f \) have their usual meanings. Let

\[
\text{HH}_\varepsilon(\sigma) = \{ j \in [n] : f_j \geq \varepsilon m \}
\]

be the set of \( \varepsilon \)-heavy hitters in \( \sigma \). Modify Misra–Gries to obtain a one-pass streaming algorithm that outputs this set “approximately” in the following sense: the set \( H \) it outputs should satisfy

\[
\text{HH}_\varepsilon(\sigma) \subseteq H \subseteq \text{HH}_{\varepsilon/2}(\sigma).
\]

Your algorithm should use \( O(\varepsilon^{-1}(\log m + \log n)) \) bits of space.

### 1-3

Suppose we have run the (one-pass) Misra–Gries algorithm on two streams \( \sigma_1 \) and \( \sigma_2 \), thereby obtaining a summary for each stream consisting of \( k \) counters. Consider the following algorithm for merging these two summaries to produce a single \( k \)-counter summary.

1. Combine the two sets of counters, adding up counts for any common items.
2. If more than \( k \) counters remain:
   1. \( c \leftarrow \) value of \((k+1)\)th counter, based on increasing order of value.
   2. Reduce each counter by \( c \) and delete all keys with non-positive counters.

Prove that the resulting summary is good for the combined stream \( \sigma_1 \circ \sigma_2 \) (here “\( \circ \)” denotes concatenation of streams) in the sense that frequency estimates obtained from it satisfy the bounds given in Eq. (1.1).
Estimating the Number of Distinct Elements

We continue with our study of data streaming algorithms for statistical problems. Previously, we focused on identifying items that are particularly dominant in a stream, appearing with especially high frequency. Intuitively, we could solve this in sublinear space because only a few items can be dominant and we can afford to throw away information about non-dominant items. Now, we consider a very different statistic: namely, how many distinct tokens (elements) appear in the stream. This is a measure of how “spread out” the stream is. It is not intuitively clear that we can estimate this quantity well in sublinear space, because we can’t afford to ignore rare items. In particular, merely sampling some tokens from the stream will mislead us, since a sample will tend to pick up frequent items rather than rare ones.

2.1 The Problem

As in Unit 1, we are in the vanilla streaming model. We have a stream \( \sigma = (a_1, \ldots, a_n) \), with each \( a_i \in [n] \), and this implicitly defines a frequency vector \( f = (f_1, \ldots, f_n) \). Let \( d = |\{ j : f_j > 0 \}| \) be the number of distinct elements that appear in \( \sigma \).

In the DISTINCT-ELEMENTS problem, our task is to output an \((\epsilon, \delta)\)-estimate (as in Definition 0.2.1) to \( d \).

It is provably impossible to solve this problem in sublinear space if one is restricted to either deterministic algorithms (i.e., \( \delta = 0 \)), or exact algorithms (i.e., \( \epsilon = 0 \)). Thus, we shall seek a randomized approximation algorithm. In this unit, we give a simple algorithm for this problem that has interesting, but not optimal, quality guarantees. Despite being sub-optimal, it is worth studying because

- the algorithm is especially simple;
- it introduces us to two ingredients used in tons of randomized algorithms, namely, universal hashing and the median trick;
- it introduces us to probability tail bounds, a basic technique for the analysis of randomized algorithms.

2.2 The Tidemark Algorithm

The idea behind the algorithm is originally due to Flajolet and Martin [FM85]. We give a slightly modified presentation, due to Alon, Matias and Szegedy [AMS99]. Since that paper designs several other algorithms as well (for other problems), it’s good to give this particular algorithm a name more evocative than “AMS algorithm.” I call it the tidemark algorithm because of how it remembers information about the input stream. Metaphorically speaking, each token has an opportunity to raise the “water level” and the algorithm simply keeps track of the high-water mark, just as a tidemark records the high-water mark left by tidal water.
For an integer $p > 0$, let $\text{zeros}(p)$ denote the number of zeros that the binary representation of $p$ ends with. Formally,

$$\text{zeros}(p) = \max\{i : 2^i \text{ divides } p\}.$$ 

Our algorithm’s key ingredient is a 2-universal hash family, a very important concept that will come up repeatedly. If you are unfamiliar with the concept, working through Exercises 2-1 and 2-2 is strongly recommended. Once we have this key ingredient, our algorithm is very simple.

**Algorithm 3** The tidemark algorithm for the number of distinct elements

**Initialize:**
1: Choose a random hash function $h : [n] \rightarrow [n]$ from a 2-universal family
2: $z \leftarrow 0$

**Process (token $j$):**
3: if $\text{zeros}(h(j)) > z$ then $z \leftarrow \text{zeros}(h(j))$

**Output:** $2^{z+\frac{1}{2}}$

The basic intuition here is that we expect 1 out of the $d$ distinct tokens to hit $\text{zeros}(h(j)) \geq \log d$, and we don’t expect any tokens to hit $\text{zeros}(h(j)) \gg \log d$. Thus, the maximum value of $\text{zeros}(h(j))$ over the stream—which is what we maintain in $z$—should give us a good approximation to $\log d$. We now analyze this.

### 2.3 The Quality of the Algorithm’s Estimate

Formally, for each $j \in [n]$ and each integer $r \geq 0$, let $X_{r,j}$ be an indicator random variable for the event “$\text{zeros}(h(j)) \geq r$,” and let $Y_r = \sum_{j : f_j > 0} X_{r,j}$. Let $T$ denote the value of $z$ when the algorithm finishes processing the stream. Clearly,

$$Y_r > 0 \iff T \geq r. \quad \quad (2.1)$$

We can restate the above fact as follows (this will be useful later):

$$Y_r = 0 \iff T \leq r - 1. \quad \quad (2.2)$$

Since $h(j)$ is uniformly distributed over the $(\log n)$-bit strings, we have

$$\mathbb{E} X_{r,j} = \mathbb{P}\{\text{zeros}(h(j)) \geq r\} = \mathbb{P}\{2^r \text{ divides } h(j)\} = \frac{1}{2^r}.$$ 

We now estimate the expectation and variance of $Y_r$ as follows. The first step of Eq. (2.3) below uses the pairwise independence of the random variables $\{X_{r,j}\}_{j \in [n]}$, which follows from the 2-universality of the hash family from which $h$ is drawn.

$$\mathbb{E} Y_r = \sum_{j : f_j > 0} \mathbb{E} X_{r,j} = \frac{d}{2^r}. \quad \quad (2.3)$$

Thus, using Markov’s and Chebyshev’s inequalities respectively, we have

$$\mathbb{P}\{Y_r > 0\} = \mathbb{P}\{Y_r \geq 1\} \leq \frac{\mathbb{E} Y_r}{1} = \frac{d}{2^r} \quad \text{and} \quad (2.4)$$

$$\mathbb{P}\{Y_r = 0\} \leq \mathbb{P}\{|Y_r - \mathbb{E} Y_r| \geq d/2^r\} \leq \frac{\text{Var} Y_r}{(d/2^r)^2} \leq \frac{2^r}{d}. \quad \quad (2.5)$$
Let \( d \) be the estimate of \( d \) that the algorithm outputs. Then \( d = 2^{T/2} \). Let \( a \) be the smallest integer such that \( 2^{a+\frac{1}{2}} \geq 3d \). Using eqs. (2.1) and (2.4), we have

\[
P\{d \geq 3d\} = P\{T \geq a\} = P\{Y_a > 0\} \leq \frac{d}{2^a} \leq \frac{\sqrt{2}}{3}.
\] (2.6)

Similarly, let \( b \) be the largest integer such that \( 2^{b+\frac{1}{2}} \leq d/3 \). Using Eqs. (2.2) and (2.5), we have

\[
P\{d \leq d/3\} = P\{T \leq b\} = P\{Y_{b+1} = 0\} \leq \frac{2^{b+1}}{d} \leq \frac{\sqrt{2}}{3}.
\] (2.7)

These guarantees are weak in two ways. Firstly, the estimate \( \hat{d} \) is only of the “same order of magnitude” as \( d \), and is not an arbitrarily good approximation. Secondly, these failure probabilities in eqs. (2.6) and (2.7) are only bounded by the rather large \( \sqrt{2}/3 \approx 0.47\% \). Of course, we could make the probabilities smaller by replacing the constant “3” above with a larger constant. But a better idea, that does not further degrade the quality of the estimate \( \hat{d} \), is to use a standard “median trick” which will come up again and again.

### 2.4 The Median Trick

Imagine running \( k \) copies of this algorithm in parallel, using mutually independent random hash functions, and outputting the median of the \( k \) answers. If this median exceeds \( 3d \), then at least \( k/2 \) of the individual answers must exceed \( 3d \), whereas we only expect \( k\sqrt{2}/3 \) of them to exceed \( 3d \). By a standard Chernoff bound, this event has probability \( 2^{-\Omega(k)} \).

Similarly, the probability that the median is below \( 3d/3 \) is also \( 2^{-\Omega(k)} \).

Choosing \( k = \Theta(\log(1/\delta)) \), we can make the sum of these two probabilities work out to at most \( \delta \). This gives us an \((O(1), \delta)\)-estimate for \( d \). Later, we shall give a different algorithm that will provide an \((\epsilon, \delta)\)-estimate with \( \epsilon \to 0 \).

The original algorithm requires \( O(\log n) \) bits to store (and compute) a suitable hash function, and \( O(\log \log n) \) more bits to store \( x \). Therefore, the space used by this final algorithm is \( O((\log(1/\delta)) \cdot \log n) \). When we reattack this problem with a new algorithm, we will also improve this space bound.

### Exercises

These exercises are designed to get you familiar with the very important concept of a 2-universal hash family, as well as give you constructive examples of such families.

Let \( X \) and \( Y \) be finite sets and let \( Y^X \) denote the set of all functions from \( X \) to \( Y \). We will think of these functions as “hash” functions. [The term “hash function” has no formal meaning; strictly speaking, one should say “family of hash functions” or “hash family” as we do here.] A family \( \mathcal{H} \subseteq Y^X \) is said to be 2-universal if the following property holds, with \( h \in \mathcal{H} \) picked uniformly at random:

\[
\forall x, x' \in X \ \forall y, y' \in Y \ \left( x \neq x' \Rightarrow P_h \{ h(x) = y \land h(x') = y' \} = \frac{1}{|Y|^2} \right).
\]

We shall give two examples of 2-universal hash families from the set \( X = \{0, 1\}^n \) to the set \( Y = \{0, 1\}^k \) (with \( k \leq n \)).

2-1 Treat the elements of \( X \) and \( Y \) as column vectors with 0/1 entries. For a matrix \( A \in \{0, 1\}^{k \times n} \) and vector \( b \in \{0, 1\}^k \), define the function \( h_{A,b} : X \to Y \) by \( h_{A,b}(x) = Ax + b \), where all additions and multiplications are performed mod 2.

Prove that the family of functions \( \mathcal{H} = \{ h_{A,b} : A \in \{0, 1\}^{k \times n}, b \in \{0, 1\}^k \} \) is 2-universal.
Identify $X$ with the finite field $\mathbb{F}_{2^n}$ using an arbitrary bijection—truly arbitrary: e.g., the bijection need not map the string $0^n$ to the zero element of $\mathbb{F}_{2^n}$. For elements $a, b \in X$, define the function $g_{a,b} : X \to Y$ as follows:

$$g_{a,b}(x) = \text{rightmost } k \text{ bits of } f_{a,b}(x), \quad \text{where}$$

$$f_{a,b}(x) = ax + b, \quad \text{with addition and multiplication performed in } \mathbb{F}_{2^n}.$$

Prove that the family of functions $\mathcal{G} = \{g_{a,b} : a, b \in \mathbb{F}_{2^n}\}$ is 2-universal. Is the family $\mathcal{G}$ better or worse than $\mathcal{H}$ in any sense? Why?
A Better Estimate for Distinct Elements

3.1 The Problem

We revisit the \textsc{distinct-elements} problem from Unit 2, giving a better solution, in terms of both approximation guarantee and space usage. We also seek good time complexity. Thus, we are again in the \textit{vanilla streaming model}. We have a stream \(\sigma = \langle a_1, a_2, a_3, \ldots, a_m \rangle\), with each \(a_i \in [n]\), and this implicitly defines a frequency vector \(f = (f_1, \ldots, f_n)\). Let \(d = |\{ j : f_j > 0 \}|\) be the number of distinct elements that appear in \(\sigma\). We want an \((\varepsilon, \delta)\)-approximation (as in Definition 0.2.1) to \(d\).

3.2 The BJKST Algorithm

In this section we present the algorithm dubbed BJKST, after the names of the authors: Bar-Yossef, Jayram, Kumar, Sivakumar and Trevisan [BJK’02]. The original paper in which this algorithm is presented actually gives three algorithms, the third (and, in a sense, “best”) of which we are presenting. The “zeros” notation below is the same as in Section 4.2. The values \(b\) and \(c\) are universal constants that will be determined later, based on the desired guarantees on the algorithm’s estimate.

\begin{algorithm}
\caption{The BJKST algorithm for \textsc{distinct-elements}}
\begin{algorithmic}[1]
\REQUIRE
\STATE Initialize: 
\STATE 1: Choose a random hash function \(h : [n] \rightarrow [n]\) from a 2-universal family 
\STATE 2: Choose a random hash function \(g : [n] \rightarrow [be^{-4}\log^2 n]\) from a 2-universal family 
\STATE 3: \(z \leftarrow 0\) 
\STATE 4: \(B \leftarrow \emptyset\)
\ENSURE
\STATE Process (token \(j\)):
\STATE 5: if \(\text{zeros}(h(j)) \geq z\) then 
\STATE 6: \(B \leftarrow B \cup \{(g(j), \text{zeros}(h(j)))\}\)
\STATE 7: while \(|B| \geq c/e^2\) do 
\STATE 8: \(z \leftarrow z + 1\)
\STATE 9: shrink \(B\) by removing all \((\alpha, \beta)\) with \(\beta < z\)
\STATE Output: \(|B|/2^z\)
\end{algorithmic}
\end{algorithm}

Intuitively, this algorithm is a refined version of the tidemark algorithm from Section 4.2. This time, rather than simply tracking the maximum value of \(\text{zeros}(h(j))\) in the stream, we try to determine the size of the bucket \(B\) consisting
of all tokens \( j \) with \( \text{zeros}(h(j)) \geq z \). Of the \( d \) distinct tokens in the stream, we expect \( d/2^z \) to fall into this bucket. Therefore \(|B|2^z\) should be a good estimate for \( d \).

We want \( B \) to be small so that we can store enough information (remember, we are trying to save space) to track \(|B|\) accurately. At the same time, we want \( B \) to be large so that the estimate we produce is accurate enough. It turns out that letting \( B \) grow to about \( \Theta(1/\varepsilon^2) \) in size is the right tradeoff. Finally, as a space-saving trick, the algorithm does not store the actual tokens in \( B \) but only their hash values under \( g \), together with the value of \( \text{zeros}(h(j)) \) that is needed to remove the appropriate elements from \( B \) when \( B \) must be shrunk.

We now analyze the algorithm in detail.

### 3.3 Analysis: Space Complexity

We assume that \( 1/\varepsilon^2 = o(m) \); otherwise, there is no point to this algorithm! The algorithm has to store \( h, g, z, \) and \( B \). Clearly, \( h \) and \( g \) dominate the space requirement. Using the finite-field-arithmetic hash family from Exercise 2-2 for our hash functions, we see that \( h \) requires \( O(\log n) \) bits of storage. The bucket \( B \) has its size capped at \( O(1/\varepsilon^2) \). Each tuple \((\alpha, \beta)\) in the bucket requires \( \log(\beta e^{-1}\log^2 n) = O(\log(1/\varepsilon) + \log\log n) \) bits to store the hash value \( \alpha \), which dominates the \( \lceil \log\log n \rceil \) bits required to store the number of zeros \( \beta \).

Overall, this leads to a space requirement of \( O(\log n + (1/\varepsilon^2)(\log(1/\varepsilon) + \log\log n)) \).

### 3.4 Analysis: The Quality of the Estimate

The entire analysis proceeds under the assumption that storing hash values (under \( g \)) in \( B \), instead of the tokens themselves, does not change \(|B|\). This is true whenever \( g \) does not have collisions on the set of tokens to which it is applied. By choosing the constant \( b \) large enough, we can ensure that the probability of this happening is at most 1/6, for each choice of \( h \) (you are asked to flesh this out in Exercise 3-1). Thus, making this assumption adds at most 1/6 to the error probability. We now analyze the rest of the error, under this no-collision assumption.

The basic setup is the same as in Section 2.3. For each \( j \in [n] \) and each integer \( r \geq 0 \), let \( X_{r,j} \) be an indicator random variable for the event \( \text{"zeros}(h(j)) \geq r\)\" and let \( Y_r = \sum_{j \in [n]} X_{r,j} \). Let \( T \) denote the value of \( z \) when the algorithm finishes processing the stream, and let \( \hat{d} \) denote the estimate output by the algorithm. Then we have

\[
Y_T = \text{value of } |B| \text{ when the algorithm finishes, and} \\
\hat{d} = Y_T 2^T.
\]

Proceeding as in Section 2.3, we obtain

\[
\forall r : \quad \mathbb{E} Y_r = \frac{d}{2^r} ; \quad \text{Var} Y_r \leq \frac{d}{2^r} . \tag{3.1}
\]

Notice that if \( T = 0 \), then the algorithm never incremented \( z \), which means that \( d < c/\varepsilon^2 \) and \( \hat{d} = |B| = d \). In short, the algorithm computes \( d \) exactly in this case.

Otherwise \( (T \geq 1) \), we say that a \( \text{FAIL} \) event occurs if \( \hat{d} \) is not a \((1 \pm \varepsilon)\)-approximation to \( d \). That is,

\[
\text{FAIL} \iff |Y_T 2^T - d| \geq \varepsilon d \iff \left| Y_T - \frac{d}{2^T} \right| \geq \frac{\varepsilon d}{2^T}.
\]

We can estimate this probability by summing over all possible values \( r \in \{1, 2, \ldots, \log n\} \) of \( T \). For the small values of \( r \), a failure will be unlikely when \( T = r \), because failure requires a large deviation of \( Y_r \) from its mean. For the large values of \( r \), simply having \( T = r \) is unlikely. This is the intuition for splitting the summation into two parts below. We need to choose the threshold that separates "small" values of \( r \) from "large" ones and we do it as follows.

Let \( s \) be the unique integer such that

\[
\frac{12}{\varepsilon^2} \leq \frac{d}{2^r} < \frac{24}{\varepsilon^2}. \tag{3.2}
\]
Then we calculate
\[
\mathbb{P}(\text{FAIL}) = \sum_{r=1}^{\log n} \mathbb{P} \left\{ \left| Y_r - \frac{d}{2^r} \right| \geq \frac{\varepsilon d}{2^r} \wedge T = r \right\} \\
\leq \sum_{r=1}^{\log n} \mathbb{P} \left\{ \left| Y_r - \frac{d}{2^r} \right| \geq \frac{\varepsilon d}{2^r} \right\} + \sum_{r=1}^{\log n} \mathbb{P} \{ T = r \} \\
= \sum_{r=1}^{s-1} \mathbb{P} \left\{ \left| Y_r - \frac{d}{2^r} \right| \geq \frac{\varepsilon d}{2^r} \right\} + \mathbb{P} \{ T \geq s \} \quad \triangleright \text{by eq. (3.1)} \\
= \sum_{r=1}^{s-1} \mathbb{P} \left\{ \left| Y_r - \frac{d}{2^r} \right| \geq \frac{\varepsilon d}{2^r} \right\} + \mathbb{P} \{ Y_{s-1} \geq c/\varepsilon^2 \}. \tag{3.3}
\]

Bounding the terms in (3.3) using Chebyshev’s inequality and Markov’s inequality, respectively, we continue:
\[
\mathbb{P}(\text{FAIL}) \leq \sum_{r=1}^{s-1} \frac{\text{Var} Y_r}{(\varepsilon d/2^r)^2} + \frac{\mathbb{E} Y_{s-1}}{c/\varepsilon^2} \\
\leq \sum_{r=1}^{s-1} \frac{2^r}{\varepsilon^2 d} + \frac{\varepsilon^2 d}{c2^{s-1}} \\
\leq \frac{2^r + 2\varepsilon^2 d}{\varepsilon^2 d} \\
\leq \frac{1}{\varepsilon^2} \cdot \frac{\varepsilon^2}{12} + \frac{2\varepsilon^2}{c} \cdot \frac{24}{\varepsilon^2} \quad \triangleright \text{by eq. (3.2)} \\
\leq \frac{1}{6}, \tag{3.4}
\]

where the final bound is achieved by choosing a large enough constant \( c \).

Recalling that we had started with a no-collision assumption for \( g \), the final probability of error is at most \( 1/6 + 1/6 = 1/3 \). Thus, the above algorithm \((\varepsilon, \frac{1}{3})\)-approximates \( d \). As before, by using the median trick, we can improve this to an \((\varepsilon, \delta)\)-approximation for any \( 0 < \delta \leq 1/3 \), at a cost of an \( O(\log(1/\delta)) \)-factor increase in the space usage.

### 3.5 Optimality

This algorithm is very close to optimal in its space usage. Later in this course, when we study lower bounds, we shall show both an \( \Omega(\log n) \) and an \( \Omega(\varepsilon^{-2}) \) bound on the space required by an algorithm that \((\varepsilon, \frac{1}{3})\)-approximates the number of distinct elements. The small gap between these lower bounds and the above upper bound was subsequently closed by Kane, Nelson, and Woodruff [KNW10]: using considerably more advanced ideas, they achieved a space bound of \( O(\varepsilon^{-2} + \log n) \).

### Exercises

3-1 Let \( \mathcal{H} \subseteq Y^X \) be a 2-universal hash family, with \(|Y| = cM^2\), for some constant \( c > 0 \). Suppose we use a random function \( h \in \mathcal{H} \) to hash a stream \( \sigma \) of elements of \( X \), and suppose that \( \sigma \) contains at most \( M \) distinct elements. Prove that the probability of a collision (i.e., the event that two distinct elements of \( \sigma \) hash to the same location) is at most \( 1/(2c) \).

3-2 Recall that we said in class that the buffer \( B \) in the BJKST Algorithm for \textsc{distinct-elements} can be implemented cleverly by not directly storing the elements of the input stream in \( B \), but instead, storing the hash values of these elements under a secondary hash function whose range is of size \( cM^2 \), for a suitable \( M \).
Using the above result, flesh out the details of this clever implementation. (One important detail that you must describe: how do you implement the buffer-shrinking step?) Plug in \( c = 3 \), for a target collision probability bound of \( 1/(2c) = 1/6 \), and figure out what \( M \) should be. Compute the resulting upper bound on the space usage of the algorithm. It should work out to

\[
O \left( \log n + \frac{1}{\varepsilon^2} \left( \log \frac{1}{\varepsilon} + \log \log n \right) \right).
\]
Approximate Counting

While observing a data stream, we would like to know how many tokens we have seen. Just maintain a counter and increment it upon seeing a token: how simple! For a stream of length \( m \), the counter would use \( O(\log m) \) space which we said (in eq. 2) was the holy grail.

Now suppose, just for this one unit, that even this amount of space is too much and we would like to solve this problem using \( o(\log m) \) space: maybe \( m \) is really huge or maybe we have to maintain so many counters that we want each one to use less than the trivial amount of space. Since the quantity we wish to maintain—the number of tokens seen—has \( m+1 \) possible values, maintaining it exactly necessarily requires \( \geq \log(m+1) \) bits. Therefore, our solution will have to be approximate.

4.1 The Problem

We are, of course, in the vanilla streaming model. In fact, the identities of the tokens are now immaterial. Therefore, we can think of the input as a stream \((1, 1, 1, \ldots)\). Our goal is to maintain an approximate count of \( n \), the number of tokens so far, using \( o(\log m) \) space, where \( m \) is some promised upper bound on \( n \). Note that we’re overriding our usual meaning of \( n \): we shall do so for just this unit.

4.2 The Algorithm

The idea behind the algorithm is originally due to Morris [Sr.78], so the approximate counter provided by it is called a “Morris counter.” We now present it in a more general form, using modern notation.

**Algorithm 5** The Morris counter

**Initialize:**
1: \( x \leftarrow 0 \)

**Process (token):**
2: \( \text{with probability } 2^{-x} \text{ do } x \leftarrow x + 1 \)

**Output:** \( 2^{x} - 1 \)

As stated, this algorithm requires \( \lceil \log m \rceil \) space after all, because the counter \( x \) could grow up to \( m \). However, as we shall soon see, \( x \) is extremely unlikely to grow beyond \( 2 \log m \) (say), so we can stay within a space bound of \( \lceil \log \log m \rceil + O(1) \) by aborting the algorithm if \( x \) does grow larger than that.
4.3 The Quality of the Estimate

The interesting part of the analysis is understanding the quality of the algorithm’s estimate. Let $C_n$ denote the (random) value of $2^x$ after $n$ tokens have been processed. Notice that $\hat{n}$, the algorithm’s estimate for $n$, equals $C_n - 1$. We shall prove that $\mathbb{E}C_n = n + 1$, which shows that $\hat{n}$ is an unbiased estimator for $n$. This by itself is not enough: we would like to prove a concentration result stating that $\hat{n}$ is unlikely to be too far from $n$. Let’s see how well we can do.

To aid the analysis, it will help to rewrite Algorithm 5 in the numerically equivalent (though space-inefficient) way shown below.

Algorithm 6 Thought-experiment algorithm for analysis of Morris counter

Initialize:
1: $c \leftarrow 1$

Process (token):
2: with probability $1/c$ do $c \leftarrow 2c$

Output: $c - 1$

Clearly, the variable $c$ in this version of the algorithm is just $2^x$ in Algorithm 5, so $C_n$ equals $c$ after $n$ tokens. Here is the key lemma in the analysis.

Lemma 4.3.1. For all $n \geq 0$, $\mathbb{E}C_n = n + 1$ and $\text{Var}C_n = n(n-1)/2$.

Proof. Let $Z_i$ be an indicator random variable for the event that $c$ is increased in line 2 upon processing the $(i+1)$th token. Then, $Z_i \sim \text{Bern}(1/C_i)$ and $C_{i+1} = (1 + Z_i)C_i$. Using the law of total expectation and the simple formula for expectation of a Bernoulli random variable,

$\mathbb{E}C_{i+1} = \mathbb{E}\mathbb{E}[(1 + Z_i)C_i | C_i] = \mathbb{E} \left( 1 + \frac{1}{C_i} \right) C_i = 1 + \mathbb{E}C_i$.

Since $\mathbb{E}C_0 = 1$, this gives us $\mathbb{E}C_n = n + 1$ for all $n$. Similarly,

$\mathbb{E}C_{i+1}^2 = \mathbb{E}\mathbb{E}[(1 + 2Z_i + Z_i^2)C_i^2 | C_i] = \mathbb{E}\mathbb{E}[(1 + 3Z_i)C_i^2 | C_i] \quad \triangleright \text{since } Z_i^2 = Z_i$

$= \mathbb{E} \left( 1 + \frac{3}{C_i} \right) C_i^2 \quad \triangleright \text{by our above formula for } \mathbb{E}C_i$

$= \mathbb{E}C_i^2 + 3(i+1)$.

Since $\mathbb{E}C_0^2 = 1$, this gives us $\mathbb{E}C_n^2 = 1 + \sum_{i=1}^n 3i = 1 + 3n(n+1)/2$. Therefore,

$\text{Var}C_n = \mathbb{E}C_n^2 - (\mathbb{E}C_n)^2 = 1 + \frac{3n(n+1)}{2} - (n+1)^2 = \frac{n(n-1)}{2}$,

upon simplification.\[\]

Unfortunately, this variance is too large for us to directly apply the median trick introduced in Section 2.4. (Exercise: try it out! See what happens if you try to bound $P\{\hat{n} < n/100\}$ using Chebyshev’s inequality and Lemma 4.3.1). However, we can still get something good out of Algorithm 5, in two different ways.

- We could tune a parameter in the algorithm, leading to a lower variance at the cost of higher space complexity. This technique, specific to this algorithm, is explored further in the exercises.
- We could plug the algorithm into a powerful, widely-applicable template that we describe below.
4.4 The Median-of-Means Improvement

As noted above, we have an estimator (\( \hat{n} \), in this case) for our quantity of interest (\( n \), in this case) that is unbiased but with variance so large that we are unable to get an \(( \varepsilon, \delta)\)-estimate (as in Definition 0.2.1). This situation arises often. A generic way to deal with it is to take our original algorithm (Algorithm 5, in this case) to be a basic estimator and then build from it a final estimator that runs several independent copies of the basic estimator in parallel and combines their output. The key idea is to first bring the variance down by averaging a number of independent copies of the basic estimator, and then apply the median trick.

**Lemma 4.4.1.** There is a universal positive constant \( c \) such that the following holds. Let random variable \( X \) be an unbiased estimator for a real quantity \( Q \). Let \( \{X_{ij}\}_{i \in [t], j \in [k]} \) be a collection of independent random variables with each \( X_{ij} \) distributed identically to \( X \), where

\[
\begin{align*}
t &= c \log \frac{1}{\delta}, \quad \text{and} \\
k &= \frac{3 \text{Var} X}{\varepsilon^2 (\mathbb{E} X)^2}.
\end{align*}
\]

Let \( Z = \text{median}_{i \in [t]} \left( \frac{1}{k} \sum_{j=1}^{k} X_{ij} \right) \). Then, we have

\[
\mathbb{P}\{ |Z - Q| \geq \varepsilon Q \} \leq \delta, \quad \text{i.e., } Z \text{ is an } (\varepsilon, \delta)\text{-estimate for } Q.
\]

Thus, if an algorithm can produce \( X \) using \( s \) bits of space, then there is an \((\varepsilon, \delta)\)-estimation algorithm using

\[
O\left( s \cdot \frac{\text{Var} X}{(\mathbb{E} X)^2} \cdot \frac{1}{\varepsilon^2} \log \frac{1}{\delta} \right)
\]

bits of space.

**Proof.** For each \( i \in [t] \), let \( Y_i = k^{-1} \sum_{j=1}^{k} X_{ij} \). Then, by linearity of expectation, we have \( \mathbb{E} Y_i = Q \). Since the variables \( X_{ij} \) are (at least) pairwise independent, we have

\[
\text{Var} Y_i = \frac{1}{k^2} \sum_{j=1}^{k} \text{Var} X_{ij} = \frac{\text{Var} X}{k}.
\]

Applying Chebyshev’s inequality, we obtain

\[
\mathbb{P}\{ |Y_i - Q| \geq \varepsilon Q \} \leq \frac{\text{Var} Y_i}{(\varepsilon Q)^2} = \frac{\text{Var} X}{k \varepsilon^2 (\mathbb{E} X)^2} = \frac{1}{3}.
\]

Now an application of a Chernoff bound (exactly as in the median trick from Section 2.4) tells us that for an appropriate choice of \( c \), we have \( \mathbb{P}\{ |Z - Q| \geq \varepsilon Q \} \leq \delta \).

Applying the above lemma to the basic estimator given by Algorithm 5 gives us the following result.

**Theorem 4.4.2.** For a stream of length at most \( m \), the problem of approximately counting the number of tokens admits an \((\varepsilon, \delta)\)-estimation in \( O(\log \log m \cdot \varepsilon^{-2} \log^{-1} 2 \log m) \) space.

**Proof.** If we run the basic estimator as is (according to Algorithm 5), using \( s \) bits of space, the analysis in Lemma 4.3.1 together with the median-of-means improvement (Lemma 4.4.1) gives us a final \((\varepsilon, \delta)\)-estimator using

\[
O(\varepsilon^{-2} \log^{-1} 2 \log m) \text{ space.}
\]

Now suppose we tweak the basic estimator to ensure \( s = O(\log \log m) \) by aborting if the stored variable \( x \) exceeds \( 2\log m \). An abortion would imply that \( C_n \geq m^2 \geq n^2 \). By Lemma 4.3.1 and Markov’s inequality,

\[
\mathbb{P}\{ C_n \geq n^2 \} \leq \frac{\mathbb{E} C_n}{n^2} = \frac{n + 1}{n^2}.
\]

Then, by a simple union bound, the probability that any one of the \( \Theta(\varepsilon^{-2} \log^{-1}) \) parallel runs of the basic estimator aborts is at most \( o(1) \). Thus, a final estimator based on this tweaked basic estimator produces an \((\varepsilon, \delta + o(1))\)-estimate within the desired space bound.

\[\square\]
Exercises

4-1 Here is a different way to improve the accuracy of the basic estimator in Algorithm 5. Observe that the given algorithm roughly tracks the logarithm (to base 2) of the stream length. Let us change the base from 2 to $1 + \beta$ instead, where $\beta$ is a small positive value, to be determined. Update the pseudocode using this idea, ensuring that the output value is still an unbiased estimator for $n$.

Analyze the variance of this new estimator and show that, for a suitable setting of $\beta$, it directly provides an $(\varepsilon, \delta)$-estimate, using only $\log \log m + O(\log \varepsilon^{-1} + \log \delta^{-1})$ bits of space.

4-2 Show how to further generalize the version of the Morris counter given by the previous exercise to solve a more general version of the approximate counting problem where the stream tokens are positive integers and a token $j$ is to be interpreted as “add $j$ to the counter.” As usual, the counter starts at zero. Provide pseudocode and rigorously analyze the algorithm’s output quality and space complexity.

4-3 Instead of using a median of means for improving the accuracy of an estimator, what if we use a mean of medians? Will it work just as well, or at all?
Finding Frequent Items via (Linear) Sketching

5.1 The Problem

We return to the FREQUENT problem that we studied in Unit 1: given a parameter $k$, we seek the set of tokens with frequency $> m/k$. The Misra–Gries algorithm, in a single pass, gave us enough information to solve FREQUENT with a second pass: namely, in one pass it computed a data structure which could be queried at any token $j \in [n]$ to obtain a sufficiently accurate estimate $\hat{f}_j$ to its frequency $f_j$. We shall now give two other one-pass algorithms for this same problem, that we can call FREQUENCY-ESTIMATION.

5.2 Sketches and Linear Sketches

Let $MG(\sigma)$ denote the data structure computed by Misra–Gries upon processing the stream $\sigma$. In ??, we saw a procedure for combining two instances of this data structure that would let us space-efficiently compute $MG(\sigma_1 \circ \sigma_2)$ from $MG(\sigma_1)$ and $MG(\sigma_2)$, where “$\circ$” denotes concatenation of streams. Clearly, it would be desirable to be able to combine two data structures in this way, and when it can be done, such a data structure is called a sketch.

**Definition 5.2.1.** A data structure $DS(\sigma)$ computed in streaming fashion by processing a stream $\sigma$ is called a sketch if there is a space-efficient combining algorithm $COMB$ such that, for every two streams $\sigma_1$ and $\sigma_2$, we have

$$COMB(DS(\sigma_1), DS(\sigma_2)) = DS(\sigma_1 \circ \sigma_2).$$

However, the Misra–Gries has the drawback that it does not seem to extend to the turnstile (or even strict turnstile) model. In this unit, we shall design two different solutions to the FREQUENT problem that do generalize to turnstile streams. Each algorithm computes a sketch of the input stream in the above sense, but these sketches have an additional important property that we now explain.

Since algorithms for FREQUENT are computing functions of the frequency vector $f(\sigma)$ determined by $\sigma$, their sketches will naturally be functions of $f(\sigma)$. It turns out that for the two algorithms in this unit, the sketches will be linear functions. That’s special enough to call out in another definition.

**Definition 5.2.2.** A sketching algorithm “sk” is called a linear sketch if, for each stream $\sigma$ over a token universe $[n]$, $sk(\sigma)$ takes values in a vector space of dimension $\ell = \ell(n)$, and $sk(\sigma)$ is a linear function of $f(\sigma)$. In this case, $\ell$ is called the dimension of the linear sketch.

Notice that the combining algorithm for linear sketches is to simply add the sketches (in the appropriate vector space). A data stream algorithm based on a linear sketch naturally generalizes from the vanilla to the turnstile model. If the arrival of a token $j$ in the vanilla model causes us to add a vector $v_j$ to the sketch, then an update $(j, c)$ in the turnstile model is handled by adding $cv_j$ to the sketch: this handles both cases $c \geq 0$ and $c < 0$. 
We now describe the first of our sketching algorithms, called CountSketch, which was introduced by Charikar, Chen and Farach-Colton [CCFC04]. We start with a basic sketch that already has most of the required ideas in it. This sketch takes an accuracy parameter $\varepsilon$ which should be thought of as small and positive.

### Algorithm 7 CountSketch: basic estimator

**Initialize:**
1. $C[1\ldots k] \leftarrow 0$, where $k := 3/\varepsilon^2$
2. Choose a random hash function $h : [n] \rightarrow [k]$ from a 2-universal family
3. Choose a random hash function $g : [n] \rightarrow \{-1, 1\}$ from a 2-universal family

**Process**(token $(j,c)$):
4. $C[h(j)] \leftarrow C[h(j)] + cg(j)$

**Output**(query $a$):
5. report $\hat{f}_a = g(a)C[h(a)]$

The sketch computed by this algorithm is the array of counters $C$, which can be thought of as a vector in $\mathbb{Z}^k$. Note that for two such sketches to be combinable, they must be based on the same hashes $h$ and $g$.

### 5.3.1 The Quality of the Basic Sketch’s Estimate

Fix an arbitrary token $a$ and consider the output $X = \hat{f}_a$ on query $a$. For each token $j \in [n]$, let $Y_j$ be the indicator for the event “$h(j) = h(a)$”. Examining the algorithm’s workings we see that a token $j$ contributes to the counter $C[h(a)]$ iff $h(j) = h(a)$, and the amount of the contribution is its frequency $f_j$ times the random sign $g(j)$. Thus,

$$X = g(a) \sum_{j=1}^n f_j g(j)Y_j = f_a + \sum_{j\in [n]\setminus \{a\}} f_j g(a)g(j)Y_j.$$

Since $g$ and $h$ are independent, we have

$$\mathbb{E}[g(j)Y_j] = \mathbb{E}[g(j)] \cdot \mathbb{E}Y_j = 0 \cdot \mathbb{E}Y_j = 0. \quad (5.1)$$

Therefore, by linearity of expectation, we have

$$\mathbb{E}X = f_a + \sum_{j\in [n]\setminus \{a\}} f_j g(a) \mathbb{E}[g(j)Y_j] = f_a. \quad (5.2)$$

Thus, the output $X = \hat{f}_a$ is an unbiased estimator for the desired frequency $f_a$.

We still need to show that $X$ is unlikely to deviate too much from its mean. For this, we analyze its variance. By 2-universality of the family from which $h$ is drawn, we see that for each $j \in [n] \setminus \{a\}$, we have

$$\mathbb{E}Y_j^2 = \mathbb{E}Y_j = \mathbb{P}\{h(j) = h(a)\} = \frac{1}{k}. \quad (5.3)$$
Next, we use 2-universality of the family from which \( g \) is drawn, and independence of \( g \) and \( h \), to conclude that for all \( i, j \in [n] \) with \( i \neq j \), we have

\[
\mathbb{E}[g(i)g(j)Y_j] = \mathbb{E}g(i) \cdot \mathbb{E}g(j) \cdot \mathbb{E}Y_j = 0 \cdot 0 \cdot \mathbb{E}Y_j = 0.
\] (5.4)

Thus, we calculate

\[
\text{Var}X = 0 + g(a)^2 \text{Var} \left[ \sum_{j \in [n] \setminus \{a\}} f_j g(j) Y_j \right]
\]

\[
= \mathbb{E} \left[ \sum_{j \in [n] \setminus \{a\}} f_j^2 Y_j^2 + \sum_{i \neq j \in [n] \setminus \{a\}} f_i f_j g(i) g(j) Y_i Y_j \right] - \left( \sum_{j \in [n] \setminus \{a\}} f_j \mathbb{E}[g(j)Y_j] \right)^2
\]

\[
= \sum_{j \in [n] \setminus \{a\}} \frac{f_j^2}{k} + 0 - 0 \quad \text{(by (5.3), (5.4), and (5.1))}
\]

\[
= \frac{\|f\|^2_2 - f_a^2}{k},
\] (5.5)

where \( f = f(\sigma) \) is the frequency distribution determined by \( \sigma \). From (5.2) and (5.5), using Chebyshev’s inequality, we obtain

\[
\mathbb{P} \left\{ |\hat{f}_a - f_a| \geq \epsilon \sqrt{\|f\|^2_2 - f_a^2} \right\} = \mathbb{P} \left\{ |X - \mathbb{E}X| \geq \epsilon \sqrt{\|f\|^2_2 - f_a^2} \right\}
\]

\[
\leq \frac{\text{Var}[X]}{\epsilon^2 (\|f\|^2_2 - f_a^2)}
\]

\[
= \frac{1}{ke^2}
\]

\[
= \frac{1}{3}.
\] (5.6)

5.3.2 The Final Sketch

The sketch that is commonly referred to as “Count Sketch” is in fact the sketch obtained by applying the median trick (see Section 2.4) to the above basic sketch, bringing its probability of error down to \( \delta \), for a given small \( \delta > 0 \). Thus, the Count Sketch can be visualized as a two-dimensional array of counters, with each token in the stream causing several counter updates. For the sake of completeness, we spell out this final algorithm in full below.

As in Section 2.4, a standard Chernoff bound argument proves that this estimate \( \hat{f}_a \) satisfies

\[
\mathbb{P} \left\{ |\hat{f}_a - f_a| \geq \epsilon \|f_{-a}\|_2 \right\} \leq \delta
\] (5.7)

With a suitable choice of hash family, we can store the hash functions above in \( O(t \log n) \) space. Each of the \( tk \) counters in the sketch uses \( O(\log m) \) space. This gives us an overall space bound of \( O(t \log n + tk \log m) \), which is

\[
O \left( \frac{1}{\epsilon^2} \log \frac{1}{\delta} \cdot (\log m + \log n) \right).
\]
We focus on the case when each token \( f \) is better than that of Count Sketch by a 1.5.4 The Count-Min Sketch

Another solution to FREQUENCY-ESTIMATION is the so-called “Count-Min Sketch”, which was introduced by Cormode and Muthukrishnan [CM05]. As with the Count Sketch, this sketch too takes an accuracy parameter \( \varepsilon \) and an error probability parameter \( \delta \). And as before, the sketch consists of a two-dimensional \( t \times k \) array of counters, which are updated in a very similar manner, based on hash functions. The values of \( t \) and \( k \) are set, based on \( \varepsilon \) and \( \delta \), as shown below.

Algorithm 8 CountSketch: final estimator

**Initialize:**
1. \( C[1 \ldots t][1 \ldots k] \leftarrow \hat{0} \), where \( k := 3/\varepsilon^2 \) and \( t := O(\log(1/\delta)) \)
2. Choose \( t \) independent hash functions \( h_1, \ldots, h_t : [n] \rightarrow [k] \), each from a 2-universal family
3. Choose \( t \) independent hash functions \( g_1, \ldots, g_t : [n] \rightarrow [k] \), each from a 2-universal family

**Process** (token \( (j, c) \)):
4. for \( i \leftarrow 1 \) to \( t \) do
5. \( C[i][h_i(j)] \leftarrow C[i][h_i(j)] + c \cdot g_i(j) \)

**Output** (query \( a \)):
6. report \( \hat{f}_a = \text{median}_{1 \leq i \leq t} g_i(a) \cdot C[i][h_i(a)] \)

5.4 The Count-Min Sketch

Note how much simpler this algorithm is, as compared to Count Sketch! Also, note that its space usage is

\[
O\left(\frac{1}{\varepsilon} \log \frac{1}{\delta} \cdot (\log m + \log n)\right),
\]

which is better than that of Count Sketch by a 1/\( \varepsilon \) factor. The place where Count-Min Sketch is weaker is in its approximation guarantee, which we now analyze.

5.4.1 The Quality of the Algorithm’s Estimate

We focus on the case when each token \( (j, c) \) in the stream satisfies \( c > 0 \), i.e., the cash register model. Clearly, in this case, every counter \( C[i][h_i(a)] \), corresponding to a token \( a \), is an overestimate of \( f_a \). Thus, we always have

\[
f_a \leq \hat{f}_a,
\]

where \( \hat{f}_a \) is the estimate of \( f_a \) output by the algorithm.

For a fixed \( a \), we now analyze the excess in one such counter, say in \( C[i][h_i(a)] \). Let the random variable \( X_i \) denote this excess. For \( j \in [n] \setminus \{a\} \), let \( Y_{i,j} \) be the indicator of the event “\( h_i(j) = h_i(a) \)”. Notice that \( j \) makes a contribution to

\[
\sum_{j \in [n] \setminus \{a\}} Y_{i,j}.
\]
the counter iff \( Y_{i,j} = 1 \), and when it does contribute, it causes \( f_j \) to be added to this counter. Thus,

\[
X_i = \sum_{j \in [n] \setminus \{a\}} f_j Y_{i,j}.
\]

By 2-universality of the family from which \( h_i \) is drawn, we compute that \( \mathbb{E} Y_{i,j} = 1/k \). Thus, by linearity of expectation,

\[
\mathbb{E} X_i = \sum_{j \in [n] \setminus \{a\}} \frac{f_j}{k} = \frac{\|f\|_1 - f_a}{k} = \frac{\|f - f_a\|_1}{k}.
\]

Since each \( f_j \geq 0 \), we have \( X_i \geq 0 \), and we can apply Markov’s inequality to get

\[
\Pr\{X_i \geq \varepsilon \|f - f_a\|_1\} \leq \frac{\varepsilon}{k \mathbb{E} X_i} \leq \frac{1}{2},
\]

by our choice of \( k \).

The above probability is for one counter. We have \( t \) such counters, mutually independent. The excess in the output, \( \hat{f}_a - f_a \), is the minimum of the excesses \( X_i \), over all \( i \in [t] \). Thus,

\[
\Pr\{\hat{f}_a - f_a \geq \varepsilon \|f - f_a\|_1\} = \Pr\{\min\{X_1, \ldots, X_t\} \geq \varepsilon \|f - f_a\|_1\}
\]

\[
= \Pr\left\{ \bigwedge_{i=1}^{t} (X_i \geq \varepsilon \|f - f_a\|_1) \right\}
\]

\[
= \prod_{i=1}^{t} \Pr(X_i \geq \varepsilon \|f - f_a\|_1)
\]

\[
\leq \frac{1}{2^t},
\]

and using our choice of \( t \), this probability is at most \( \delta \). Thus, we have shown that, with high probability,

\[
f_a \leq \hat{f}_a \leq f_a + \varepsilon \|f - f_a\|_1,
\]

where the left inequality always holds, and the right inequality fails with probability at most \( \delta \).

The reason this estimate is weaker than that of Count Sketch is that its deviation is bounded by \( \varepsilon \|f - f_a\|_1 \), rather than \( \varepsilon \|f - f_a\|_2 \). For all vectors \( z \in \mathbb{R}^n \), we have \( |z|_1 \geq |z|_2 \). The inequality is tight when \( z \) has a single nonzero entry. It is at its weakest when all entries of \( z \) are equal in absolute value: the two norms are then off by a factor of \( \sqrt{n} \) from each other. Thus, the quality of the estimate of Count Sketch gets better (in comparison to Count-Min Sketch) as the stream’s frequency vector gets more “spread out”.

### 5.5 Comparison of Frequency Estimation Methods

At this point, we have studied three methods to estimate frequencies of tokens in a stream. The following table throws in a fourth method, and compares these methods by summarizing their key features.

<table>
<thead>
<tr>
<th>Method</th>
<th>( f_a - \hat{f}_a \in \cdots )</th>
<th>Space, ( O(\cdot) )</th>
<th>Error Probability</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Misra–Gries</td>
<td>([- \varepsilon |f - f_a|_1, 0])</td>
<td>( \frac{1}{\varepsilon} (\log m + \log n) )</td>
<td>0 (deterministic)</td>
<td>Cash register</td>
</tr>
<tr>
<td>CountSketch</td>
<td>([- \varepsilon |f - f_a|_2, \varepsilon |f - f_a|_2] )</td>
<td>( \frac{1}{\varepsilon^2} \log \frac{1}{\delta} (\log m + \log n) )</td>
<td>( \delta ) (overall)</td>
<td>Turnstile</td>
</tr>
<tr>
<td>Count-Min Sketch</td>
<td>([0, \varepsilon |f - f_a|_1])</td>
<td>( \frac{1}{\varepsilon^2} \log \frac{1}{\delta} (\log m + \log n) )</td>
<td>( \delta ) (upper bound)</td>
<td>Cash register</td>
</tr>
<tr>
<td>Count/Median</td>
<td>([- \varepsilon |f - f_a|_1, \varepsilon |f - f_a|_1] )</td>
<td>( \frac{1}{\varepsilon^2} \log \frac{1}{\delta} (\log m + \log n) )</td>
<td>( \delta ) (overall)</td>
<td>Turnstile</td>
</tr>
</tbody>
</table>

The claims in the first row can be proved by analyzing the Misra–Gries algorithm from Lecture 1 slightly differently. The last row refers to an algorithm that maintains the same data structure as the Count-Min Sketch, but answers queries by reporting the median of the absolute values of the relevant counters, rather than the minimum. It is a simple (and instructive) exercise to analyze this algorithm and prove the claims in the last row.
Exercises

5-1 Prove that for every integer \( n \geq 1 \) and every vector \( \mathbf{z} \in \mathbb{R}^n \), we have \( \|\mathbf{z}\|_1 / \sqrt{n} \leq \|\mathbf{z}\|_2 \leq \|\mathbf{z}\|_1 \). For both inequalities, determine when equality holds.

5-2 Write out the Count/Median algorithm formally and prove that it satisfies the properties claimed in the last row of the above table.

5-3 Our estimate of \( \ell \|\mathbf{f}_{-a}\|_2 \) on the absolute error of CountSketch is too pessimistic in many practical situations where the data are highly skewed, i.e., where most of the weight of the vector \( \mathbf{f} \) is supported on a small “constant” number of elements. To make this precise, we define \( \mathbf{f}^{\text{res}(\ell)}_{-a} \) to be the \((n-1)\)-dimensional vector obtained by dropping the \( a \)th entry of \( \mathbf{f} \) and then setting the \( \ell \) largest (by absolute value) entries to zero.

Now consider the CountSketch estimate \( \hat{\mathbf{f}}_a \) as computed by the algorithm in Sec 5.3.2 with the only change being that \( k \) is set to \( 6/\epsilon^2 \). Prove that
\[
P\left\{ |\hat{\mathbf{f}}_a - \mathbf{f}_a| \geq \epsilon \|\mathbf{f}^{\text{res}(\ell)}_{-a}\|_2 \right\} \leq \delta,
\]
where \( \ell = 1/\epsilon^2 \).

5-4 Consider a stream \( \sigma \) in the turnstile model, defining a frequency vector \( \mathbf{f} \geq 0 \). The Count-Min Sketch solves the problem of estimating \( f_j \), given \( j \), but does not directly give us a quick way to identify, e.g., the set of elements with frequency greater than some threshold. Fix this.

In greater detail: Let \( \alpha \) be a constant with \( 0 < \alpha < 1 \). We would like to maintain a suitable summary of the stream (some enhanced version of Count-Min Sketch, perhaps?) so that we can, on demand, quickly produce a set \( S \subseteq [n] \) satisfying the following properties w.h.p.: (1) \( S \) contains every \( j \) such that \( f_j \geq \alpha F_1 \); (2) \( S \) does not contain any \( j \) such that \( f_j < (\alpha - \epsilon)F_1 \). Here, \( F_1 = F_1(\sigma) = \|\mathbf{f}\|_1 \). Design a data stream algorithm that achieves this. Your space usage, as well as the time taken to process each token and to produce the set \( S \), should be polynomial in the usual parameters, \( \log m \), \( \log n \), and \( 1/\epsilon \), and may depend arbitrarily on \( \alpha \).
Unit 6

Estimating Frequency Moments

6.1 Background and Motivation

We are in the vanilla streaming model. We have a stream \( \sigma = \langle a_1, \ldots, a_m \rangle \), with each \( a_j \in [n] \), and this implicitly defines a frequency vector \( f = f(\sigma) = (f_1, \ldots, f_n) \). Note that \( f_1 + \cdots + f_n = m \). The \( k \)th frequency moment of the stream, denoted \( F_k(\sigma) \) or simply \( F_k \), is defined as follows:

\[
F_k := \sum_{j=1}^{n} f_j^k = \|f\|_k^k. \tag{6.1}
\]

Using the terminology “\( k \)th” suggests that \( k \) is a positive integer. But in fact the definition above makes sense for every real \( k > 0 \). And we can even give it a meaning for \( k = 0 \), if we first rewrite the definition of \( F_k \) slightly: \( F_k = \sum_{j : f_j > 0} f_j^k \). With this definition, we get

\[
F_0 = \sum_{j : f_j > 0} f_j^0 = \{ j : f_j > 0 \},
\]

which is the number of distinct tokens in \( \sigma \). (We could have arrived at the same result by sticking with the original definition of \( F_k \) and adopting the convention \( 0^0 = 0 \).

We have seen that \( F_0 \) can be \((\varepsilon, \delta)\)-approximated using space logarithmic in \( m \) and \( n \). And \( F_1 = m \) is trivial to compute exactly. Can we say anything for general \( F_k \)? We shall investigate this problem in this and the next few lectures.

By way of motivation, note that \( F_2 \) represents the size of the self join \( r \bowtie r \), where \( r \) is a relation in a database, with \( f_j \) denoting the frequency of the value \( j \) of the join attribute. Imagine that we are in a situation where \( r \) is a huge relation and \( n \), the size of the domain of the join attribute, is also huge; the tuples can only be accessed in streaming fashion (or perhaps it is much cheaper to access them this way than to use random access). Can we, with one pass over the relation \( r \), compute a good estimate of the self join size? Estimation of join sizes is a crucial step in database query optimization.

The solution we shall eventually see for this \( F_2 \) estimation problem will in fact allow us to estimate arbitrary equi-join sizes (not just self joins). For now, though, we give an \((\varepsilon, \delta)\)-approximation for arbitrary \( F_k \), provided \( k \geq 2 \), using space sublinear in \( m \) and \( n \). The algorithm we present is due to Alon, Matias and Szegedy [AMS99], and is not the best algorithm for the problem, though it is the easiest to understand and analyze.

6.2 The (Basic) AMS Estimator for \( F_k \)

We first describe a surprisingly simple basic estimator that gets the answer right in expectation, i.e., it is an unbiased estimator. Eventually, we shall run many independent copies of this basic estimator in parallel and combine the results to get our final estimator, which will have good error guarantees.
The estimator works as follows. Pick a token from the stream $\sigma$ uniformly at random, i.e., pick a position $J \in_R [m]$. Count the length, $m$, of the stream and the number, $r$, of occurrences of our picked token $a_J$ in the stream from that point on: $r = |\{j \geq J : a_j = a_J\}|$. The basic estimator is then defined to be $m(r^k - (r - 1)^k)$.

The catch is that we don’t know $m$ beforehand, and picking a token uniformly at random requires a little cleverness, as seen in the pseudocode below.

**Algorithm 10 AMS basic estimator for frequency moments**

**Initialize:**
1. $(m, r, a) \leftarrow (0, 0, 0)$

**Process (token $j$):**
2. $m \leftarrow m + 1$
3. with probability $1/m$ do
4. $a \leftarrow j$
5. $r \leftarrow 0$
6. if $j = a$ then
7. $r \leftarrow r + 1$

**Output:** $m(r^k - (r - 1)^k)$

This algorithm uses $O(\log m)$ bits to store $m$ and $r$, plus $\lceil \log n \rceil$ bits to store the token $a$, for a total space usage of $O(\log m + \log n)$. Although stated for the vanilla streaming model, it has a natural generalization to the cash register model. It is a good homework exercise to figure this out.

### 6.3 Analysis of the Basic Estimator

First of all, let us agree that the algorithm does indeed compute $r$ as described above. For the analysis, it will be convenient to think of the algorithm as picking a random token from $\sigma$ in two steps, as follows.

1. Pick a random token value, $a \in [n]$, with $P(a = j) = f_j/m$ for each $j \in [n]$.
2. Pick one of the $f_a$ occurrences of $a$ in $\sigma$ uniformly at random.

Let $A$ and $R$ denote the (random) values of $a$ and $r$ after the algorithm has processed $\sigma$, and let $X$ denote its output. Taking the above viewpoint, let us condition on the event $A = j$, for some particular $j \in [n]$. Under this condition, $R$ is equally likely to be any of the values $\{1, \ldots, f_j\}$, depending on which of the $f_j$ occurrences of $j$ was picked by the algorithm. Therefore,

$$E[X \mid A = j] = E[m(R^k - (R - 1)^k) \mid A = j] = \sum_{i=1}^{f_j} \frac{1}{f_j} \cdot m(i^k - (i - 1)^k) = \frac{m}{f_j} (f_j^k - 0^k).$$

By the law of total expectation,

$$E[X] = \sum_{j=1}^{n} P(A = j) E[X \mid A = j] = \sum_{j=1}^{n} \frac{f_j}{m} \cdot \frac{m}{f_j} \cdot f_j^k = F_k.$$

This shows that $X$ is indeed an unbiased estimator for $F_k$.

We shall now bound $\text{Var} \, X$ from above. Calculating the expectation as before, we have

$$\text{Var} \, X \leq E[X^2] = \sum_{j=1}^{n} \frac{f_j}{m} \sum_{i=1}^{f_j} \frac{1}{f_j} \cdot m^2(i^k - (i - 1)^k)^2 = m \sum_{j=1}^{n} \sum_{i=1}^{f_j} \left(\frac{i^k - (i - 1)^k}{f_j}\right)^2.$$

(6.2)
By the mean value theorem (from basic calculus), for all $x \geq 1$, there exists $\xi(x) \in [x - 1, x]$ such that
\[
x^k - (x-1)^k = k\xi(x)^{k-1} \leq k^{k-1},
\]
where the last step uses $k \geq 1$. Using this bound in (6.2), we get
\[
\begin{align*}
\text{Var}_X &\leq m \sum_{j=1}^{n} \sum_{i=1}^{f_j} k^{k-1} \left( i^k - (i-1)^k \right) \\
&\leq m \sum_{j=1}^{n} k f_j^{k-1} \sum_{i=1}^{f_j} \left( i^k - (i-1)^k \right) \\
&= m \sum_{j=1}^{n} k f_j^{k-1} f_j^k \\
&= k F_1 F_{2k-1}.
\end{align*}
\]

For reasons we shall soon see, it will be convenient to bound $\text{Var}_X$ by a multiple of $(E_X)^2$, i.e., $F_k^2$. To do so, we shall use the following lemma.

**Lemma 6.3.1.** Let $n > 0$ be an integer and let $x_1, \ldots, x_n \geq 0$ and $k \geq 1$ be reals. Then
\[
\left( \sum_{i=1}^{n} x_i \right) \left( \sum_{i=1}^{n} x_i^{2k-1} \right) \leq n^{1-1/k} \left( \sum_{i=1}^{n} x_i^k \right)^2,
\]
where all the summations range over $i \in [n]$.

**Proof.** We continue to use the convention that summations range over $i \in [n]$. Let $x_\ast = \max_{i \in [n]} x_i$. Then, we have
\[
x_\ast^{k-1} = \left( x_\ast^k \right)^{(k-1)/k} \leq \left( \sum_{i=1}^{n} x_i^k \right)^{(k-1)/k}.
\]
Since $k \geq 1$, by the power mean inequality (or directly, by the convexity of the function $x \mapsto x^k$), we have
\[
\frac{1}{n} \sum_{i=1}^{n} x_i \leq \left( \frac{1}{n} \sum_{i=1}^{n} x_i^k \right)^{1/k}.
\]
Using (6.4) and (6.5) in the second and third steps (respectively) below, we compute
\[
\left( \sum_{i=1}^{n} x_i \right) \left( \sum_{i=1}^{n} x_i^{2k-1} \right) \leq \left( \sum_{i=1}^{n} x_i \right) \left( x_\ast^{k-1} \sum_{i=1}^{n} x_i^k \right) \\
\leq \left( \sum_{i=1}^{n} x_i^k \right) \left( \sum_{i=1}^{n} x_i^k \right)^{(k-1)/k} \left( \sum_{i=1}^{n} x_i^k \right)^{(k-1)/k} \\
\leq n^{1-1/k} \left( \sum_{i=1}^{n} x_i^k \right)^{1/k} \left( \sum_{i=1}^{n} x_i^k \right)^{(k-1)/k} \left( \sum_{i=1}^{n} x_i^k \right) \\
= n^{1-1/k} \left( \sum_{i=1}^{n} x_i^k \right)^2,
\]
which completes the proof. \qed

Using the above lemma in (6.3), with $x_j = f_j$, we get
\[
\text{Var}_X \leq k F_1 F_{2k-1} \leq kn^{1-1/k} F_k^2.
\]

\[\text{(6.6)}\]
6.4 The Final Estimator and Space Bound

Our final $F_k$ estimator, which gives us good accuracy guarantees, is obtained by combining several independent basic estimators and using the median-of-means improvement (Lemma 4.4.1 in Section 4.4). By that lemma, we can obtain an $(\varepsilon, \delta)$-estimator for $F_k$ by combining $O(r\varepsilon^{-2}\log \delta^{-1})$ independent copies of the basic estimator $X$, where

$$r = \frac{\text{Var} X}{(EX)^2} = \frac{kn^{1-1/k}F_k}{F_k^2} = kn^{1-1/k}.$$ 

As noted above, the space used to compute each copy of $X$, using Algorithm 10, is $O(\log m + \log n)$, leading to a final space bound of

$$O\left(\frac{1}{\varepsilon^2} \log \frac{1}{\delta} \cdot kn^{1-1/k} (\log m + \log n)\right) = \tilde{O}(\varepsilon^{-2} n^{1-1/k}).$$

(6.7)

6.4.1 The Soft-O Notation

For the first time in these notes, we have a space bound that is sublinear, but not polylogarithmic, in $n$ (or $m$). In such cases it is convenient to adopt an $\tilde{O}$-notation, also known as a “soft-O” notation, which suppresses factors polynomial in $\log m$, $\log n$, $\log \varepsilon^{-1}$, and $\log \delta^{-1}$. We have adopted this notation in eq. (6.7), leading to the memorable form $\tilde{O}(\varepsilon^{-2} n^{1-1/k})$. Note that we are also treating $k$ as a constant here.

The above bound is good, but not optimal, as we shall soon see. The optimal bound (upto polylogarithmic factors) is $\tilde{O}(\varepsilon^{-2} n^{1-2/k})$ instead; there are known lower bounds of $\Omega(n^{1-2/k})$ and $\Omega(\varepsilon^{-2})$. We shall see how to achieve this better upper bound in a subsequent unit.
The Tug-of-War Sketch

At this point, we have seen a sublinear-space algorithm — the AMS estimator — for estimating the $k$th frequency moment, $F_k = f_1^k + \cdots + f_n^k$, of a stream $\sigma$. This algorithm works for $k \geq 2$, and its space usage depends on $n$ as $\tilde{O}(n^{1-1/k})$. This fails to be polylogarithmic even in the important case $k = 2$, which we used as our motivating example when introducing frequency moments in the previous lecture. Also, the algorithm does not produce a sketch in the sense of Section 5.2.

But Alon, Matias and Szegedy [AMS99] also gave an amazing algorithm that does produce a sketch—a linear sketch of merely logarithmic size—which allows one to estimate $F_2$. What is amazing about the algorithm is that seems to do almost nothing.

7.1 The Basic Sketch

We describe the algorithm in the turnstile model.

\begin{algorithm}
\caption{Tug-of-War Sketch for $F_2$}
\begin{algorithmic}
\State \textbf{Initialize:}
\State 1. Choose a random hash function $h : \mathbb{N} \rightarrow \{-1, 1\}$ from a 4-universal family
\State 2. $z \leftarrow 0$
\State \textbf{Process (token $(j, c)$):}
\State 3. $z \leftarrow z + c h(j)$
\State \textbf{Output:} $z^2$
\end{algorithmic}
\end{algorithm}

The sketch is simply the random variable $z$. It is pulled in the positive direction by those tokens $j$ that have $h(j) = 1$ and is pulled in the negative direction by the rest of the tokens; hence the name Tug-of-War Sketch. Clearly, the absolute value of $z$ never exceeds $f_1 + \cdots + f_k = m$, so it takes $O(\log m)$ bits to store this sketch. It also takes $O(\log n)$ bits to store the hash function $h$, for an appropriate 4-universal family.
7.1.1 The Quality of the Estimate

Let \( Z \) denote the value of \( z \) after the algorithm has processed \( \sigma \). For convenience, define \( Y_j = h(j) \) for each \( j \in [n] \). Then \( Z = \sum_{j=1}^{n} f_j Y_j \). Note that \( Y_j^2 = 1 \) and \( \mathbb{E} Y_j = 0 \), for each \( j \). Therefore,

\[
\mathbb{E} Z^2 = \mathbb{E} \left[ \sum_{j=1}^{n} f_j^2 Y_j^2 + \sum_{j \neq \ell} f_j f_\ell Y_j Y_\ell \right] = \sum_{j=1}^{n} f_j^2 + \sum_{j \neq \ell} f_j f_\ell \mathbb{E} Y_j Y_\ell = F_2,
\]

where we used the fact that \( \{ Y_j \}_{j \in [n]} \) are pairwise independent (in fact, they are 4-wise independent, because \( h \) was picked from a 4-universal family). This shows that the algorithm’s output, \( Z^2 \), is indeed an unbiased estimator for \( F_2 \).

The variance of the estimator is \( \text{Var} Z^2 = \mathbb{E} Z^4 - (\mathbb{E} Z^2)^2 = \mathbb{E} Z^4 - F_2^2 \). We bound this as follows. By linearity of expectation, we have

\[
\mathbb{E} Z^4 = \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} \sum_{\ell=1}^{n} f_i f_j f_k f_\ell \mathbb{E} [Y_i Y_k Y_j Y_\ell].
\]

Suppose one of the indices in \( (i, j, k, \ell) \) appears exactly once in that 4-tuple. Without loss of generality, we have \( i \notin \{ j, k, \ell \} \). By 4-wise independence, we then have \( \mathbb{E} [Y_i Y_k Y_j Y_\ell] = \mathbb{E} Y_i \cdot \mathbb{E} [Y_j Y_k Y_\ell] = 0 \), because \( \mathbb{E} Y_i = 0 \). It follows that the only potentially nonzero terms in the above sum correspond to those 4-tuples \( (i, j, k, \ell) \) that consist either of one index occurring four times, or else two distinct indices occurring twice each. Therefore we have

\[
\mathbb{E} Z^4 = \sum_{j=1}^{n} f_j^4 \mathbb{E} Y_j^4 + 6 \sum_{i=1}^{n} \sum_{j=i+1}^{n} f_i^2 f_j^2 \mathbb{E} [Y_i^2 Y_j^2] = F_4 + 6 \sum_{i=1}^{n} \sum_{j=i+1}^{n} f_i^2 f_j^2,
\]

where the coefficient “6” corresponds to the \( \binom{4}{2} = 6 \) permutations of \( (i, i, j, j) \) with \( i \neq j \). Thus,

\[
\text{Var} Z^2 = F_4 - F_2^2 + 6 \sum_{i=1}^{n} \sum_{j=i+1}^{n} f_i^2 f_j^2 \\
= F_4 - F_2^2 + 3 \left( \left( \sum_{j=1}^{n} f_j^2 \right)^2 - \sum_{j=1}^{n} f_j^4 \right) \\
= F_4 - F_2^2 + 3 (F_2^2 - F_4) \leq 2 F_2^2.
\]

7.2 The Final Sketch

As before, having bounded the variance, we can design a final sketch from the above basic sketch by a median-of-means improvement. By Lemma 4.4.1, this will blow up the space usage by a factor of

\[
\frac{\text{Var} Z^2}{(\mathbb{E} Z^2)^2} \cdot O \left( \frac{1}{\varepsilon^2 \log \frac{1}{\delta}} \right) \leq \frac{2 F_2^2}{F_2^2} \cdot O \left( \frac{1}{\varepsilon^2 \log \frac{1}{\delta}} \right) = O \left( \frac{1}{\varepsilon^2 \log \frac{1}{\delta}} \right)
\]

in order to give an \((\varepsilon, \delta)\)-estimate. Thus, we have estimated \( F_2 \) using space \( O(\varepsilon^{-2} \log(\delta^{-1})(\log m + \log n)) \), with a sketching algorithm that in fact computes a linear sketch.

7.2.1 A Geometric Interpretation

The AMS Tug-of-War Sketch has a nice geometric interpretation. Consider a final sketch that consists of \( t \) independent copies of the basic sketch. Let \( \mathbf{M} \in \mathbb{R}^{t \times n} \) be the matrix that “transforms” the frequency vector \( f \) into the \( t \)-dimensional sketch vector \( y \). Note that \( \mathbf{M} \) is not a fixed matrix but a random matrix with \( \pm 1 \) entries: it is drawn from a certain distribution described implicitly by the hash family. Specifically, if \( M_{ij} \) denotes the \((i, j)\)-entry of \( \mathbf{M} \), then \( M_{ij} = h_i(j) \), where \( h_i \) is the hash function used by the \( i \)th basic sketch.
Let $t = 6/\varepsilon^2$. By stopping the analysis in Lemma 4.4.1 after the Chebyshev step (and before the “median trick” Chernoff step), we obtain that

$$\mathbb{P}\left\{ \frac{1}{t} \sum_{i=1}^{t} y_i^2 - F_2 \geq \varepsilon F_2 \right\} \leq \frac{1}{3},$$

where the probability is taken with respect to the above distribution of $M$, resulting in a random sketch vector $y = (y_1, \ldots, y_t)$. Thus, with probability at least $2/3$, we have

$$\|\frac{1}{\sqrt{t}} M f\|_2 = \frac{1}{\sqrt{t}} \|y\|_2 \in \left[ \sqrt{1 - \varepsilon} \|f\|_2, \sqrt{1 + \varepsilon} \|f\|_2 \right] \subseteq \left[ (1 - \varepsilon) \|f\|_2, (1 + \varepsilon) \|f\|_2 \right].$$

This can be interpreted as follows. The (random) matrix $M/\sqrt{t}$ performs a dimension reduction, simplifying an $n$-dimensional vector $f$ to a $t$-dimensional sketch $y$—with $t = O(1/\varepsilon^2)$—while preserving $\ell_2$-norm within a $(1 \pm \varepsilon)$ factor. Of course, this is only guaranteed to happen with probability at least $2/3$. But clearly this correctness probability can be boosted to an arbitrary constant less than 1, while keeping $t = O(1/\varepsilon^2)$.

The “amazing” AMS sketch now feels quite natural, under this geometric interpretation. We are using dimension reduction to maintain a low-dimensional image of the frequency vector. This image, by design, has the property that its $\ell_2$-length approximates that of the frequency vector very well. Which of course is what we’re after, because the second frequency moment, $F_2$, is just the square of the $\ell_2$-length.

Since the sketch is linear, we now also have an algorithm to estimate the $\ell_2$-difference $\|f(\sigma) - f(\sigma')\|_2$ between two streams $\sigma$ and $\sigma'$.

**Exercises**

7-1 In Section 6.1, we noted that $F_2$ represents the size of a self join in a relational database and remarked that our $F_2$ estimation algorithm would allow us to estimate arbitrary equi-join sizes (not just self joins). Justify this by designing a sketch that can scan a relation in one streaming pass such that, based on the sketches of two different relations, we can estimate the size of their join. Explain how to compute the estimate.

Recall that for two relations (i.e., tables in a database) $r(A, B)$ and $s(A, C)$, with a common attribute (i.e., column) $A$, we define the join $r \bowtie s$ to be a relation consisting of all tuples $(a, b, c)$ such that $(a, b) \in r$ and $(a, c) \in s$. Therefore, if $f_{r,j}$ and $f_{s,j}$ denote the frequencies of $j$ in the first columns (i.e., “$A$”-columns) of $r$ and $s$, respectively, and $j$ can take values in $[n]$, then the size of the join is $\sum_{j=1}^{n} f_{r,j} f_{s,j}$.
Estimating Norms Using Stable Distributions

As noted at the end of Lecture 7, the AMS Tug-of-War sketch allows us to estimate the $\ell_2$-difference between two data streams. Estimating similarity metrics between streams is an important class of problems, so it is nice to have such a clean solution for this specific metric.

However, this raises a burning question: Can we do the same for other $\ell_p$ norms, especially the $\ell_1$ norm? The $\ell_1$-difference between two streams can be interpreted (modulo appropriate scaling) as the variational distance (a.k.a., statistical distance) between two probability distributions: a fundamental and important metric. Unfortunately, although our log-space $F_2$ algorithm automatically gave us a log-space $\ell_2$ algorithm, the trivial log-space $F_1$ algorithm works only in the cash register model and does not give an $\ell_1$ algorithm at all.

It turns out that thinking harder about the geometric interpretation of the AMS Tug-of-War Sketch leads us on a path to polylogarithmic space $\ell_p$ norm estimation algorithms, for all $p \in (0, 2]$. Such algorithms were given by Indyk [Ind06], and we shall study them now. For the first time in this course, it will be necessary to gloss over several technical details of the algorithms, so as to have a clear picture of the important ideas.

### 8.1 A Different $\ell_2$ Algorithm

The length-preserving dimension reduction achieved by the Tug-of-War Sketch is reminiscent of the famous Johnson-Lindenstrauss Lemma [JL84, FM88]. One high-level way of stating the JL Lemma is that the random linear map given by a $t \times n$ matrix whose entries are independently drawn from the standard normal distribution $\mathcal{N}(0, 1)$ is length-preserving (up to a scaling factor) with high probability. To achieve $1 \pm \varepsilon$ error, it suffices to take $t = O(1/\varepsilon^2)$.

Let us call such a matrix a JL Sketch matrix. Notice that the sketch matrix for the Tug-of-War sketch is a very similar object, except that

1. its entries are uniformly distributed in $\{-1, 1\}$: a much simpler distribution;
2. its entries do not have to be fully independent: 4-wise independence in each row suffices; and
3. it has a succinct description: it suffices to describe the hash functions that generate the rows.

The above properties make the Tug-of-War Sketch “data stream friendly”. But as a thought experiment one can consider an algorithm that uses a JL Sketch matrix instead. It would give a correct algorithm for $\ell_2$ estimation, except that its space usage would be very large, as we would have to store the entire sketch matrix. In fact, since this hypothetical algorithm calls for arithmetic with real numbers, it is unimplementable as stated.

Nevertheless, this algorithm has something to teach us, and will generalize to give (admittedly unimplementable) $\ell_p$ algorithms for each $p \in (0, 2]$. Later we shall make these algorithms realistic and space-efficient. For now, we consider the basic sketch version of this algorithm, i.e., we maintain just one entry of $Mf$, where $M$ is a JL Sketch matrix. The pseudocode below shows the operations involved.
Let \( X \) denote the value of \( x \) when this algorithm finishes processing \( \sigma \). Then \( X = \sum_{j=1}^{n} f_j Y_j \). From basic statistics, using the independence of the collection \( \{Y_j\}_{j \in [n]} \), we know that \( X \) has the same distribution as \( \|f\|_2 Y \), where \( Y \sim \mathcal{N}(0, 1) \). This is a fundamental property of the normal distribution.\(^1\) Therefore, we have \( \mathbb{E}[X^2] = \|f\|_2^2 = F_2 \), which gives us our unbiased estimator for \( F_2 \).

### 8.2 Stable Distributions

The fundamental property of the normal distribution that was used above has a generalization, which is the key to generalizing this algorithm. The next definition captures the general property.

**Definition 8.2.1.** Let \( p > 0 \) be a real number. A probability distribution \( \mathcal{D}_p \) over the reals is said to be \( p \)-stable if for all integers \( n \geq 1 \) and all \( c = (c_1, \ldots, c_n) \in \mathbb{R}^n \), the following property holds. If \( X_1, \ldots, X_n \) are independent and each \( X_i \sim \mathcal{D}_p \), then \( c_1 X_1 + \cdots + c_n X_n \) has the same distribution as \( c \cdot X \), where \( X \sim \mathcal{D}_p \) and

\[
\bar{c} = (c_1^p + \cdots + c_n^p)^{1/p} = \|c\|_p.
\]

The concept of stable distributions dates back to Lévy [Lév54] and is more general than what we need here. It is known that \( p \)-stable distributions exist for all \( p \in (0, 2] \), and do not exist for any \( p > 2 \). The fundamental property above can be stated simply as: “The standard normal distribution is 2-stable.”

Another important example of a stable distribution is the Cauchy distribution, which can be shown to be 1-stable. Just as the standard normal distribution has density function

\[
\phi(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2},
\]

the Cauchy distribution also has a density function expressible in closed form as

\[
c(x) = \frac{1}{\pi(1+x^2)}.
\]

But what is really important to us is not so much that the density function of \( \mathcal{D}_p \) be expressible in closed form, but that it be easy to generate random samples drawn from \( \mathcal{D}_p \). The Chambers-Mallows-Stuck method [CMS76] gives us the following simple algorithm. Let

\[
X = \frac{\sin(p \theta)}{(\cos \theta)^{1/p}} \left( \frac{\cos((1-p) \theta)}{\ln(1/r)} \right)^{(1-p)/p},
\]

where \( (\theta, r) \in \mathbb{R} \times [-\pi/2, \pi/2] \times [0, 1] \). Then the distribution of \( X \) is \( p \)-stable.

Replacing \( \mathcal{N}(0, 1) \) with \( \mathcal{D}_p \) in the above pseudocode, where \( \mathcal{D}_p \) is \( p \)-stable, allows us to generate a random variable distributed according to \( \mathcal{D}_p \) “scaled” by \( \|f\|_p \). Note that the scaling factor \( \|f\|_p \) is the quantity we want to estimate. To estimate it, we shall simply take the median of a number of samples from the scaled distribution, i.e., we shall maintain a sketch consisting of several copies of the basic sketch and output the median of (the absolute values of) the entries of the sketch vector. Here is our final “idealized” sketch.

### 8.3 The Median of a Distribution and its Estimation

To analyze this algorithm, we need the concept of the median of a probability distribution over the reals. Let \( \mathcal{D} \) be an absolutely continuous distribution, let \( \phi \) be its density function, and let \( X \sim \mathcal{D} \). A median of \( \mathcal{D} \) is a real number \( \mu \) that satisfies

\[
\frac{1}{2} = \Pr[X \leq \mu] = \int_{-\infty}^{\mu} \phi(x) \, dx.
\]

\(^1\)The proof of this fact is a nice exercise in calculus.
The distributions that we are concerned with here are nice enough to have uniquely defined medians; we will simply speak of the median of a distribution. For such a distribution \( \mathcal{D} \), we will denote this unique median as \( \text{median}(\mathcal{D}) \).

For a distribution \( \mathcal{D} \), with density function \( \phi \), we denote by \( |\mathcal{D}| \) the distribution of the absolute value of a random variable drawn from \( \mathcal{D} \). It is easy to show that the density function of \( |\mathcal{D}| \) is \( \psi \), where
\[
\psi(x) = \begin{cases} 
2\phi(x), & \text{if } x \geq 0 \\
0, & \text{if } x < 0.
\end{cases}
\]

For \( p \in (0, 2] \) and \( c \in \mathbb{R} \), let \( \phi_{p,c} \) denote the density function of the distribution of \( c|X| \), where \( X \sim \mathcal{D}_p \), and let \( \mu_{p,c} \) denote the median of this distribution. Note that
\[
\phi_{p,c}(x) = \frac{1}{c} \phi_{p,1} \left( \frac{x}{c} \right), \quad \text{and} \quad \mu_{p,c} = c \mu_{p,1}.
\]

Let \( X_i \) denote the final value of \( x_i \) after the above algorithm has processed \( \sigma \). By the earlier discussion, and the definition of \( p \)-stability, we see that \( X_i \equiv \|f\|_p X_i \), where \( X_i \sim \mathcal{D}_p \). Therefore, \( |X_i|/\text{median}(|\mathcal{D}_p|) \) has a distribution whose density function is \( \phi_{p,\lambda} \), where \( \lambda = \|f\|_p / \text{median}(|\mathcal{D}_p|) = \|f\|_p / \mu_{p,1} \). Thus, the median of this distribution is \( \mu_{p,\lambda} = \lambda \mu_{p,1} = \|f\|_p \).

The algorithm — which seeks to estimate \( \|f\|_p \) — can thus be seen as attempting to estimate the median of an appropriate distribution by drawing \( t = O(\varepsilon^{-2} \log(\delta^{-1})) \) samples from it and outputting the sample median. We now show that this does give a fairly accurate estimate.

### 8.4 The Accuracy of the Estimate

**Lemma 8.4.1.** Let \( \varepsilon > 0 \), and let \( \mathcal{D} \) be a distribution over \( \mathbb{R} \) with density function \( \phi \), and with a unique median \( \mu > 0 \). Suppose that \( \phi \) is absolutely continuous on \( [(1 - \varepsilon)\mu, (1 + \varepsilon)\mu] \) and let \( \phi_* = \min \{ \phi(x) : x \in [(1 - \varepsilon)\mu, (1 + \varepsilon)\mu] \} \). Let \( Y = \text{median}_{1 \leq i \leq t} X_i \), where \( X_1, \ldots, X_t \) are independent samples from \( \mathcal{D} \). Then
\[
\Pr[|Y - \mu| \geq \varepsilon \mu] \leq 2 \exp \left( -\frac{2}{3} \varepsilon^2 \mu^2 t \right).
\]

**Proof.** We bound \( \Pr[Y < (1 - \varepsilon)\mu] \) from above. A similar argument bounds \( \Pr[Y > (1 + \varepsilon)\mu] \) and to complete the proof we just add the two bounds.

Let \( \Phi(y) = \int_{-\infty}^y \phi(x) \, dx \) be the cumulative distribution function of \( \mathcal{D} \). Then, for each \( i \in [t] \), we have
\[
\Pr[X_i < (1 - \varepsilon)\mu] = \int_{-\infty}^{(1 - \varepsilon)\mu} \phi(x) \, dx - \int_{(1 - \varepsilon)\mu}^\mu \phi(x) \, dx
\[
= \frac{1}{2} - \Phi(\mu) + \Phi((1 - \varepsilon)\mu)
\[
= \frac{1}{2} - \varepsilon \mu \phi(\xi),
\]
for some \( \xi \in [(1 - \varepsilon)\mu, \mu] \), where the last step uses the mean value theorem and the fundamental theorem of calculus: \( \Phi' = \phi \). Let \( \alpha \) be defined by
\[
\left( \frac{1}{2} - \varepsilon \mu \phi(\xi) \right) (1 + \alpha) = \frac{1}{2}.
\]

Let \( N = |\{ i \in [t] : X_i < (1 - \varepsilon)\mu \} | \). By linearity of expectation, we have \( \mathbb{E}[N] = \left( \frac{1}{2} - \varepsilon \mu \phi(\xi) \right) t \). If the sample median, \( Y \), falls below a limit \( \lambda \), then at least half the \( X_i \)s must fall below \( \lambda \). Therefore
\[
\Pr[Y < (1 - \varepsilon)\mu] \leq \Pr[N \geq t/2] = \Pr[N \geq (1 + \alpha) \mathbb{E}[N]] \leq \exp(-\mathbb{E}[N] \alpha^2 / 3),
\]
by a standard Chernoff bound. Now, from (8.1), we derive \( \mathbb{E}[N] \alpha = \varepsilon \mu \phi(\xi) \) and \( \alpha \geq 2 \varepsilon \mu \phi(\xi) \). Therefore
\[
\Pr[Y < (1 - \varepsilon)\mu] \leq \exp \left( -\frac{2}{3} \varepsilon^2 \mu^2 \phi(\xi)^2 t \right) \leq \exp \left( -\frac{2}{3} \varepsilon^2 \mu^2 t \right). \]
To apply the above lemma to our situation we need an estimate for $\phi_*$. We will be using the lemma with $\phi = \phi_{p, \lambda}$ and $\mu = \mu_{p, \lambda} = \|f\|_p$, where $\lambda = \|f\|_p / \mu_{p, 1}$. Therefore,

$\mu \phi_* = \mu_{p, \lambda} \cdot \min \{ \phi_{p, \lambda}(x) : x \in [(1 - \varepsilon)\mu_{p, \lambda}, (1 + \varepsilon)\mu_{p, \lambda}] \}
= \lambda \mu_{p, 1} \cdot \min \left\{ \frac{1}{\lambda} \phi_{p, 1}\left(\frac{x}{\lambda}\right) : x \in [(1 - \varepsilon)\mu_{p, 1}, (1 + \varepsilon)\mu_{p, 1}] \right\}
= \mu_{p, 1} \cdot \min \{ \phi_{p, 1}(y) : y \in [(1 - \varepsilon)\mu_{p, 1}, (1 + \varepsilon)\mu_{p, 1}] \},$

which is a constant depending only on $p$: call it $c_p$. Thus, by Lemma 8.4.1, the output $Y$ of the algorithm satisfies

$$\Pr \left[ |Y - \|f\|_p| \geq \varepsilon \|f\|_p \right] \leq \exp \left( -\frac{2}{3} \varepsilon^2 c_p^2 \right) \leq \delta,$$

for the setting $t = (3/(2c_p^2))e^{-2\log(\delta^{-1})}$.

### 8.5 Annoying Technical Details

There are two glaring issues with the “idealized” sketch we have just discussed and proven correct. As stated, we do not have a proper algorithm to implement the sketch, because

- the sketch uses real numbers, and algorithms can only do bounded-precision arithmetic; and
- the sketch depends on a huge matrix — with $n$ columns — that does not have a convenient implicit representation.

We will not go into the details of how these matters are resolved, but here is an outline.

We can approximate all real numbers involved by rational numbers with sufficient precision, while affecting the output by only a small amount. The number of bits required per entry of the matrix $M$ is only logarithmic in $n$, $1/\varepsilon$ and $1/\delta$.

We can avoid storing the matrix $M$ explicitly by using a pseudorandom generator (PRG) designed to work with space-bounded algorithms. One such generator is given by a theorem of Nisan [Nis90]. Upon reading an update to a token $j$, we use the PRG (seeded with $j$ plus the initial random seed) to generate the $j$th column of $M$. This transformation blows up the space usage by a factor logarithmic in $n$ and adds $1/n$ to the error probability.

There is actually a third issue: there does not seem to be a theorem in the literature that proves that $c_p$ (see the analysis above) is nonzero! We tacitly assumed this. For $c_1$, we can show this by explicit calculation, because we have a convenient closed-form expression for the density function of the Cauchy distribution. Indyk [Ind06] resolves this issue by changing the algorithm for $\ell_p$ ($p \neq 1$) slightly in a way that allows him to perform the necessary calculations. Have a look at his paper for the details on this, and the other two matters discussed above.

### 8.5.1 An Open Question

Life would be a lot easier if there were some way to generate entries of a suitable $\ell_p$-sketching matrix $M$ from $O(1)$-wise independent random variables, similar to the way 4-wise independent collections of $\pm 1$-valued random variables worked for $\ell_2$. Is there? Then we would not have to rely on the heavy hammer of Nisan’s PRG, and perhaps we could make do with very simple arithmetic, as in the Tug-of-War Sketch.
Bibliography


