CS 514: Advanced Algorithms II – Sublinear Algorithms **Rutgers: Spring 2020**

Lecture 7

March 3, 2020

Instructor: Sepehr Assadi *Scribe: Abraham Gale and Guido Tagliavini*

1 Streaming Algorithms

We now move from sublinear time algorithms to the related area of streaming algorithms. These algorithms are another family of sublinear algorithms but instead of *time* as in sublinear time algorithms, they focus on *space* of algorithms. A streaming algorithm processes its inputs in small chunks, one at a time, and thus does not need to store the entire input in one place. For motivation, consider a router in a network: the router needs to process a massive number of packets using a limited memory much smaller than what allows for storing all the packets it sees during its process.

In the following, we give a formal definition of the model and a simple example of a streaming algorithm. We then switch to considering one of the first problems considered in the streaming model, the *distinct element* problem, and design a streaming algorithm for that. We conclude this lecture by describing the general family of *frequency estimation* problems.

Streaming Model of Computation

We now define the model formally. The input consists of n elements $e_1, e_2, \ldots, e_n$, where each e_i belongs to some universe $\mathcal{U}$, that are received one at a time by the algorithm, sequentially. Every time a new element is received, the previous one is erased, so the algorithm only has access to the most recent element. The algorithm has a local memory available, separate from the input which is (ideally) much smaller than the input (and so we cannot store the input entirely by the end of the stream). As such, the goal in this model is to design algorithms that use only a small amount of memory compared to the input size, typically (but not always) of size $\text{poly}(\log n, \log |\mathcal{U}|)$ bits.

Warm-Up: Uniform Sampling in a Stream

Let us start with the following simple problem.

Problem 1. Given a stream of elements from the universe $[m]$, sample an element e_i uniformly at random.

If we know the length n of the input in advance, this is trivial – simply sample a number $i \in [n]$, keep a counter of the number of elements seen so far, and store the i-th element. This algorithm uses $O(\log n + \log m)$ bits of space: $O(\log n)$ for the index i and the counter, and $O(\log m)$ for the element. But what if we do not know the length n of the stream? The following algorithm, *Reservoir Sampling*, solves this problem.

Reservoir Sampling:

1. Let s be the selected element. Initially $s \leftarrow e_1$.
2. When e_i arrives, set $s \leftarrow e_i$ with probability $\frac{1}{i}$.
3. When the stream ends, return s.

The space complexity of this algorithm is also $O(\log n + \log m)$ bits: $O(\log n)$ for storing the counter i, and $O(\log m)$ space for storing the chosen number s.

Lemma 1. *In reservoir sampling algorithm,* $\Pr(s = e_i) = 1/n$ *for every* e_i *in the stream.*

Proof. We have

$$\begin{aligned}
\Pr(s = e_i) &= \Pr(s \leftarrow e_i \text{ and no other element is chosen as } s) \\
&= \Pr(s \leftarrow e_i) \cdot \Pr(\text{no other element is chosen as } s) && \text{(because every step is independent)} \\
&= \frac{1}{i} \cdot \Pr(\text{no other element is chosen}) && \text{(because we select } e_i \text{ as } s \text{ with probability } 1/i) \\
&= \frac{1}{i} \cdot \prod_{j=i+1}^{n} \left(1 - \Pr(s \leftarrow e_j)\right) && \text{(again, because every step is independent)} \\
&= \frac{1}{i} \cdot \prod_{j=i+1}^{n} \left(1 - \frac{1}{j}\right) && \text{(again, because we select } e_j \text{ as with probability } 1/j) \\
&= \frac{1}{i} \cdot \prod_{j=i+1}^{n} \frac{j-1}{j} \\
&= \frac{1}{i} \cdot \frac{i}{i+1} \cdot \frac{i+1}{i+2} \cdots \frac{n-1}{n} \\
&= \frac{1}{n},
\end{aligned}$$

concluding the proof. □

We leave it as an easy exercise to the reader to show how one can modify this algorithm to sample k numbers from the stream (both with or without repetition) in $O(k \cdot (\log n + \log m))$ space.

An interesting question? The following question seems interesting to consider. The $O(\log m)$ space in the algorithm is clearly necessary just to output the answer. But what about the $\log n$ term? Can one prove that if we are required to output a number *truly* uniformly at random[1] from n numbers we also need the extra $O(\log n)$ factor in the space?

2 Distinct Element Counting

We now consider one of the first (and highly influential) problems considered in the streaming model, namely, the distinct element (counting) problem.

Problem 2. Given a stream of n elements from the universe $[m]$, output the number of *distinct* elements in the stream, denoted by DE.

For example, if $m = 5$, and the stream is $1, 2, 2, 1, 5, 4, 2, 2, 1$, then the answer is $\mathsf{DE} = 4$.

There are two naive solutions to this problem:

- Store the entire universe: Use a bitmap with m bits. Every time we see a new element, mark it. This requires $O(m)$ bits.
- Store the entire stream: Store a set of all the elements we receive. This requires $O(n \log m)$ bits.

[1] As opposed to sample each element with probability, say, $\frac{1 \pm o(1)}{n}$.

These type of straightforward solutions are applicable to most streaming problems.

What about an algorithm using $\text{poly}(\log n, \log m)$ bits? In the next lecture we will show that this is not possible without randomization and approximation, by proving the following lower bounds:

- Every deterministic algorithm requires $\Omega(m)$ bits, even if it is a, say, 1.1-approximation.
- Every exact randomized algorithm requires $\Omega(n)$ bits.

Therefore, to find a sublinear space streaming algorithm we need to allow for both approximation and randomization. In particular, we require our algorithm to output a number $\widetilde{\mathsf{DE}}$ such that

$$\Pr\left(|\mathsf{DE} - \widetilde{\mathsf{DE}}| \leq \varepsilon \cdot \mathsf{DE}\right) \geq 1 - \delta.$$

Next we will give two algorithms that solve this problem using $\text{poly}(\log n, \log m, 1/\varepsilon, \log(1/\delta))$ bits of space.

2.1 A Geometric Search Problem

Our first algorithm will use a standard trick. The idea is to introduce a variable T, that takes values in $[m]$, and for each T decide whether DE is *approximately* smaller or larger than T. More specifically, we want a randomized streaming algorithm Alg, that given a threshold T:

- if $\mathsf{DE} \leq T$, Alg answers *No* with probability $\geq 1 - \delta$;
- if $\mathsf{DE} > 2T$, Alg answers *Yes* with probability $\geq 1 - \delta$.

We will refer to Alg as an algorithm to (threshold) *test* DE. Assuming Alg answers correctly most of the time, the sequence of outputs over an increasing sequence of thresholds will be roughly bitonic, and the stationary point will be approximately DE. Since Alg is not committed to a specific output when $\mathsf{DE} \in (T, 2T]$, its answer when the threshold is near DE can go either way. See Figure 1.

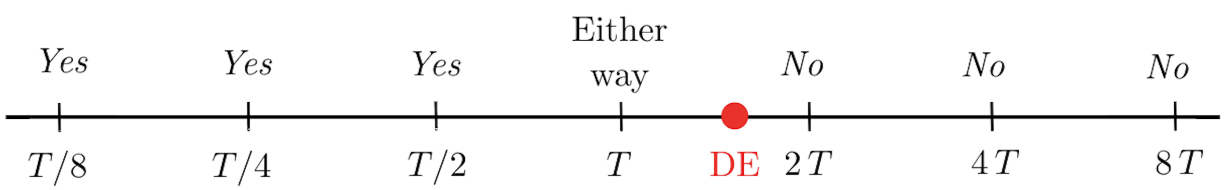

Figure 1: Alg's answer for geometrically increasing thresholds. The output is approximately monotonic.

Before we solve this threshold version of the problem, we show how to use such an algorithm to construct an approximation algorithm for the original problem.

Algorithm:

1. For each $i \in \{0, \ldots, \log m\}$, let $T_i := 2^i$.
2. Let Alg be a randomized streaming algorithm for testing DE.
3. Run Alg for each of the T_i's (in parallel) with confidence parameter $\delta' = \delta/(2 \log m)$.
4. Let $\widetilde{\mathsf{DE}}$ be the smallest T_i such that Alg answered *No* on T_i. Return $\widetilde{\mathsf{DE}}$.

The space complexity of this algorithm is $O(\log m)$ times the space complexity of *Alg* (for confidence parameter $\delta' = O(\delta/\log m)$). So if we design a small-space algorithm *Alg* for the threshold version of the problem, we will also obtain a small-space algorithm for the original problem to within $\operatorname{poly}\log(m)$ factors.

Lemma 2. *The algorithm outputs a 2-approximation of* DE *with probability at least* $1-\delta$.

Proof. Let i be the largest integer such that $T_i < \mathsf{DE}$ and so $\mathsf{DE} < T_{i+1}$. The output is a 2-approximation if and only if $\widetilde{\mathsf{DE}} \in \{T_i, T_{i+1}\}$.

Consider the events:

- $\mathcal{E}_1$: when $\widetilde{\mathsf{DE}} \leq T_{i-1}$; and,
- $\mathcal{E}_2$: when $\widetilde{\mathsf{DE}} \geq T_{i+2}$.

Then, $\Pr\left(|\widetilde{\mathsf{DE}} - \mathsf{DE}| > 2\,\mathsf{DE}\right) = \Pr(\mathcal{E}_1 \cup \mathcal{E}_2) = \Pr(\mathcal{E}_1) + \Pr(\mathcal{E}_2)$. On the one hand,

$$\begin{aligned}
\Pr(\mathcal{E}_1) &= \Pr(\textit{Alg} \text{ answers } \textit{No} \text{ for some } T_j, \text{ with } j \leq i-1) \\
&\leq \sum_{j=0}^{i-1} \Pr(\textit{Alg} \text{ answers } \textit{No} \text{ for } T_j) && \text{(by union bound)} \\
&\leq \sum_{j=0}^{i-1} \frac{\delta}{2\log m} && \text{(because } \mathsf{DE} > 2\cdot T_j \text{ and the probability that } \textit{Alg} \text{ errs is at most } \delta/(2\log m)) \\
&\leq \log m \cdot \frac{\delta}{2\log m} = \frac{\delta}{2}.
\end{aligned}$$

On the other hand,

$$\begin{aligned}
\Pr(\mathcal{E}_2) &= \Pr(\textit{Alg} \text{ answers } \textit{Yes} \text{ for all } T_j \text{ with } j \leq i+1) \\
&\leq \Pr(\textit{Alg} \text{ answers } \textit{Yes} \text{ for } T_{i+1}) \\
&\leq \frac{\delta}{2\log m}. && \text{(because } \mathsf{DE} < T_{i+1} \text{ and the probability that } \textit{Alg} \text{ errs is } \delta/(2\log m))
\end{aligned}$$

Thus $\Pr\left(|\widetilde{\mathsf{DE}} - \mathsf{DE}| > 2\cdot\mathsf{DE}\right) \leq \delta/2 + \delta/2\log m \leq \delta$. ◻

Remark. The approximation factor 2 in this algorithm can be improved to $1+\varepsilon$ for any $\varepsilon > 0$, by reducing the geometric ratio of the thresholds to $1+\varepsilon$, and solving the threshold version of the problem for distinguishing between T and $(1+\varepsilon)\cdot T$. We omit the details as we are going to see a more space-efficient algorithm for this task anyway.

Remark. Notice that our algorithm in this section (and in particular Lemma 2) did not use any details of *Alg* (i.e., treated it as a black-box) or for that matter even any particular knowledge the problem at hand, namely, estimating DE. As such, this trick of combining a threshold-testing version of a problem with geometric searching can be applied to other problems as well and forms a standard technique in streaming algorithms (as well as many other algorithms).

An Algorithm for Threshold Testing DE

The following algorithm solves the threshold testing problem, for a given threshold T. We may assume that $T \geq 100$, since for any $T = O(1)$ we can simply use the deterministic $O(T)$-space naive algorithm that counts the number of distinct elements up to T, and answers *Yes* as soon as it sees $T + 1$ of them.

Algorithm: A preliminary version of *Alg* used in the previous section.

1. Pick a truly random hash function $h : [m] \to [T]$ (i.e., each element in $[m]$ is hashed independently to some number in $[T]$ chosen uniformly at random).
2. For each element in the stream, check $h(e_i)$. If $h(e_i) = 1$, return *Yes*.
3. If the stream ends, return *No*.

We will consider the space complexity of this algorithm later in the section. For now, we should only note that *as it is*, this algorithm requires prohibitively large space to store the hash function h and hence is not space-efficient. Regardless, it forms the main building block for our final algorithm and thus we focus on proving its correctness in the following.

The intuition behind the algorithm is as follows: for each *distinct* number in the stream, $h(\cdot)$ has a chance of hitting 1 with probability $1/T$. As such, if $\mathsf{DE} > 2T$, then it is very likely that one of the values will be hashed to 1, but if $\mathsf{DE} \leq T$ that probability is lower.

Although we need *Alg* to answer correctly with confidence $1 - \delta$, this algorithm, as is, does not quite get us that far. We will first prove that *Alg* answers correctly with constant nonzero probability, and then show how to bump up that probability to $1 - \delta$.

Lemma 3. *Alg satisfies the following:*

- *if* $\mathsf{DE} \leq T$*, then Alg answers No with probability* $\geq 1/e^{1.01}$*;*
- *if* $\mathsf{DE} > 2T$*, then Alg answers No with probability* $\leq 1/e^2$*.*

Proof. Let $i_1, \ldots, i_{\mathsf{DE}} \in [n]$ be the indices of the distinct elements of the stream. If $\mathsf{DE} > 2T$, then

$$\begin{aligned}
\Pr\left(\textit{Alg} \text{ answers } \textit{No}\right) &= \Pr_h(h(e_{i_j}) \neq 1 \text{ for all } j \in [\mathsf{DE}]) \\
&= \prod_{j=1}^{\mathsf{DE}} \Pr_h(h(e_{i_j}) \neq 1) \quad \text{(because } h \text{ is truly random and thus independent for each } e_{i_j}) \\
&= \prod_{j=1}^{\mathsf{DE}} \left(1 - \frac{1}{T}\right) \\
&= \left(1 - \frac{1}{T}\right)^{\mathsf{DE}} && \text{(by the inequality } 1 - x \leq e^{-x} \text{ for all } x) \\
&\leq \exp\left(-\mathsf{DE}/T\right) \\
&< \frac{1}{e^2}. && \text{(as } \mathsf{DE} > 2T)
\end{aligned}$$

Now suppose $\mathsf{DE} \leq T$. Just as in the previous case, we have

$$\Pr\left(\textit{Alg} \text{ answers } \textit{No}\right) = \left(1 - \frac{1}{T}\right)^{\mathsf{DE}}.$$

In this case, this is the probability that *Alg* gives the right answer, so we want a lower bound. We use the following upper bound for the exponential function to simplify the above bound.

Proposition 4. *Let $c > 1$ be any real number. Then $e^{-cx} \le 1 - x$ for all $x \in \left[0, \min\left\{\frac{1}{c}, 2 \cdot \frac{c-1}{c^2}\right\}\right]$.*

Proof. The inequality is trivial for $x = 0$. Let $x > 0$. Consider the Taylor series for e^{-cx}:

$$e^{-cx} = 1 - cx + \frac{(cx)^2}{2!} - \frac{(cx)^3}{3!} + \dots$$

This is an alternating series in which the terms are monotonically decreasing in absolute value, since

$$\frac{(cx)i}{i!} \ge \frac{(cx)^{i+1}}{(i+1)!} \iff \frac{(i+1)!}{i!} \ge \frac{(cx)^{i+1}}{(cx)^i} \qquad \text{(as } x, c > 0\text{)}$$

$$\iff i + 1 \ge cx \iff \frac{i+1}{c} \ge x, \qquad \text{(as } c > 0\text{)}$$

and the last inequality holds because $x \le 1/c$. Thus, we can bound the series by any partial sum up to any positive term. In particular,

$$e^{-cx} \le 1 - cx + \frac{(cx)^2}{2}.$$

An easy calculation shows that $1 - cx + \frac{(cx)^2}{2} \le 1 - x$ if and only if $x \le 2 \cdot \frac{c-1}{c^2}$. □

By setting $c = 1.01$ in Proposition 4, we have $1 - x \ge e^{-1.01\,x}$ for all $x \in [0, 0.01]$. Since $T \ge 100$, then $1/T \le 0.01$, and thus the inequality implies that $1 - 1/T \ge \exp(-1.01/T)$. Therefore,

$$\Pr(\textit{Alg} \text{ answers } \textit{No}) \ge \exp\left(-1.01\,\mathsf{DE}/T\right) \ge 1/e^{1.01}, \qquad \text{(since } \mathsf{DE} \le T\text{)}$$

concluding the proof. □

Lemma 3 implies that we "expect" to see *No* more often when $\mathsf{DE} \le T$ compared to when $\mathsf{DE} > 2T$. However, these probabilities are still far from our desired bounds of δ on the error (or even, say, (1/3)-error).

We are already familiar with the concept of amplifying probability of success from previous lectures. For instance, we have previously used independent repetition of one-sided error algorithms, and majority/median voting on algorithms with error probability "around" half. In the case of Lemma 3 however, *Alg* is a two-sided error algorithm with error probabilities that are in both cases "far from" half. Nevertheless, the previous approaches can be adjusted for this purpose as well.

One way to amplify the confidence of algorithm in deciding whether $T \le \mathsf{DE}$ or $T > 2\mathsf{DE}$, is to exploit the *additive* constant gap between the probability that *Alg* answers *No* in each case. Paraphrasing Lemma 3, if $\mathsf{DE} \le T$, then at least $1/e^{1.01}$ of the time *Alg* will output *No*, and if $\mathsf{DE} > 2T$, then at most $1/e^2 < 1/e^{1.01} - 0.22$ of the time *Alg* will output *No*. Thus, after running *Alg* sufficiently many times, we can tell which case we are in. We do so using the following algorithm.

Algorithm: A final version of *Alg* with amplified probability of success:

1. Let λ be any constant in $\left(\frac{1}{e^2}, \frac{1}{e^{1.01}}\right)$. Let $\lambda_1 := \lambda - 1/e^2$ and $\lambda_2 := 1/e^{1.01} - \lambda$.
2. Run *Alg*, $k = \ln(\frac{2}{\delta}) \cdot \frac{1}{2} \cdot \max\left\{\frac{1}{\lambda_1^2}, \frac{1}{\lambda_2^2}\right\}$ independent times, in parallel.
3. Let N_i be the indicator variable that is 1 if and only if *Alg* answers *No* in the i-th execution. Let $N := \sum_{i=1}^{k} N_i$.
4. If $N \ge \lambda \cdot k$, answer *No*. Otherwise, answer *Yes*.

Lemma 5. *Amplified Alg returns the right answer with probability* $\geq 1 - \delta$.

Proof. First suppose $\mathsf{DE} \leq T$, so the right answer in this case is *No*. The confidence guarantee of *Alg* when $\mathsf{DE} \leq T$ implies $\mathbb{E}[N_i] = \Pr(\textit{Alg} \text{ answers } \textit{No}) \geq 1/e^{1.01}$, and thus $\mathbb{E}[N] \geq k/e^{1.01}$. We have

$$\begin{aligned}
\Pr(\text{answer } \textit{Yes}) &= \Pr(N < \lambda\, k) \\
&= \Pr\left(N < k\left(\frac{1}{e^{1.01}} - \lambda_2\right)\right) \\
&\leq \Pr(N < \mathbb{E}[N] - k\lambda_2) && \text{(because } \mathbb{E}[N] \geq k/e^{1.01}\text{)} \\
&\leq \Pr(|N - \mathbb{E}[N]| \geq k\lambda_2) \\
&\leq 2 \cdot \exp\left(-\frac{2\,\lambda_2^2\, k^2}{k}\right) && \text{(by the additive Chernoff bound)} \\
&\leq \delta. && \text{(because } k \geq \ln(\tfrac{2}{\delta}) \cdot \tfrac{1}{2\,\lambda_2^2}\text{)}
\end{aligned}$$

The case $\mathsf{DE} > 2T$ is symmetric. □

As such, we can indeed amplify the success probability of original *Alg* to within $1 - \delta$ using $O(\ln(1/\delta))$ independent repetitions in parallel. Note that in terms of a streaming algorithm, this only increases the space by a factor of $O(\ln(1/\delta))$ compared to the original algorithm.

Space complexity of *Alg*? The algorithm we designed in this section is still *not* a space-efficient streaming algorithm – while the entire space of the algorithm throughout the stream is $O(\log m)$ to store e_i and compute $h(e_i)$, storing the *random bits* in h itself requires $O(m \log T)$ bits (which makes the space of the algorithm more than the naive $O(m)$ space!).

Recall that we encountered this issue before when designing LCAs as well – sometimes, the only bottleneck in space of our algorithms is to store random bits. Previously, we mentioned that this is often not problematic and can be remedied through use of some "pseudo-random numbers" instead of truly random numbers. In the next section, we will examine one such option (used extensively in streaming algorithms) in details.

2.2 Trading Independence for Space: Limited-Independence Hash Functions

Generating and storing a random function $h : [m] \to [T]$ requires $O(m \log T)$ bits, which is too much. Although the analysis of *Alg* relies on the full independence provided by the random hash function, we can make it work even when there is only "limited independence" in this hash function. Let us define this formally as follows.

Definition 6. A family $\mathcal{H} = \{h : [a] \to [b]\}$ is called a **k-wise independent** family of hash functions if for all pairwise distinct $x_1, \ldots, x_k \in [a]$ and all $y_1, \ldots, y_k \in [b]$,

$$\Pr_{h \sim \mathcal{H}}(h(x_1) = y_1 \wedge \cdots \wedge h(x_k) = y_k) = \frac{1}{b^k}.$$

Observe than a k-wise independent family may also be $(k+1)$-wise independent, i.e., the definition does not necessarily break for $k+1$ hash values (although for "interesting" families this is almost always the case). For instance, a truly random hash function is k-wise independent for all k.

Proposition 7. *Let* $\mathcal{H} = \{h : [a] \to [b]\}$ *be a* k*-wise independent, and* $h \sim \mathcal{H}$ *chosen at random. Let* $x_1, \ldots, x_k \in [a]$ *be arbitrary pairwise distinct elements. Then:*

1. $h(x_1)$ *is uniform over* $[b]$;
2. $h(x_1), \ldots, h(x_k)$ *are mutually independent.*

Proof. We prove each part separately.

1. Let $y_1 \in [b]$. Observe that

$$\Pr_{h \sim \mathcal{H}}(h(x_1) = y_1) = \sum_{y_2,\ldots,y_k \in [b]} \Pr_{h \sim \mathcal{H}}(h(x_1) = y_1 \wedge h(x_2) = y_2 \wedge \cdots \wedge h(x_k) = y_k)$$

(partitioning the sample space)

$$= \sum_{y_2,\ldots,y_k \in [b]} \frac{1}{b^k} = \frac{b^{k-1}}{b^k} = \frac{1}{b}. \qquad \text{(by definition of } k\text{-wise independent family)}$$

2. Let $y_1, \ldots, y_k \in [b]$. Since all $h(x_i)$ are uniform over $[b]$, it follows that

$$\Pr_{h \sim \mathcal{H}}(h(x_1) = y_1 \wedge \cdots \wedge h(x_k) = y_k) = \frac{1}{b^k} = \prod_{i=1}^{k} \Pr_{h \sim \mathcal{H}}(h(x_i) = y_i).$$

□

Example. Here is an example of a k-wise independent family of hash functions: $[p] \to [p]$ for any prime p.

k-wise Independent Family of Hash Functions:

1. $\mathcal{H}$ is the set of degree-$(k-1)$ polynomial functions over $\mathbb{F}_p$ (field of integers mod prime p). That is,

$$\mathcal{H} = \{h : [p] \to [p] \mid h(x) = c_{k-1}x^{k-1} + c_{k-2}x^{k-2} + \cdots + c_1 x + c_0, \text{ with } c_0, \ldots, c_{k-1} \in \mathbb{F}_p\}.$$

2. Sampling $h \sim \mathcal{H}$: Sample $c_0, \ldots, c_{k-1} \in \mathbb{F}_p$. Then h is the polynomial defined by this coefficients.

To see why this is a k-wise independent hash function, note that any degree-$(k-1)$ polynomial h is uniquely determined by having k of its values (i.e., k distinct $(x, h(x))$ pairs for $x \in \mathbb{F}_p$): if we fix only $k-1$ values of h on $x_1, \ldots, x_{k-1}$, value of $h(x_k)$ for any other x_k is still chosen uniformly at random from $\mathbb{F}_p$ (we omit the simple algebraic proof of this statement as it is not the focus of this lecture).

The important thing we would like to note about the family $\mathcal{H}$ is on how much space we need to store h. Since each function in the family is defined by k polynomial coefficients, the space required to generate and store it is only $O(k \log p)$ bits (as opposed to $O(p \log p)$ for a truly random hash function mapping $[p] \to [p]$). It is also possible to evaluate any such hash function in the same amount of space.

Although this family only works for $a = b = p$, we can in general construct families for arbitrary a and b.

Proposition 8. *For any a, b, there exists a k-wise independent family of hash functions mapping $[a] \to [b]$, that requires $O(k \cdot (\log a + \log b))$ bits.*

Remark. We can replace the truly random hash function $h : [m] \to [T]$ in algorithm of previous section with a 2-wise independent hash function that only requires $O(\log m + \log T) = O(\log m)$ bits to store. Using a slightly different analysis, one can then prove that the algorithm continues to output the correct answer with sufficiently large probability. This way we can obtain a streaming algorithm for the threshold testing problem using $O(\log m \cdot \log(1/\delta))$ space. By combining this with the geometric search algorithm, we get an algorithm for 2-approximation of DE using $O(\log^2 m \cdot (\log\log m + \log(1/\delta)))$ space. In the next section, we give a simpler and more direct algorithm for this problem.

2.3 A Better Algorithm for Estimating DE

To improve the space cost and achieve a $(1+\varepsilon)$-approximation to DE, we will combine the idea of limiting the independency with a more direct algorithm that does not rely on geometrically searching the range of possible answers.

The intuition is as follows: Suppose every time we receive a number from the universe $[m]$, we hash it and place it in one of m slots. To simplify things, suppose there are no collisions. At the end of the stream, DE slots will be non-empty and we "expect" them to be evenly distributed – the first nonempty slot should be approximately at m/DE, the second at $2 \cdot m/\mathsf{DE}$, and the t-th one at $t \cdot m/\mathsf{DE}$. Thus, if we can compute the t-th nonempty slot number X, we can use $X \approx t \cdot m/\mathsf{DE}$ to estimate DE (the reason we go for some relatively larger t as opposed to just $t = 1$ is to make the concentration results in the algorithm work as it will become evident shortly).

Algorithm:

1. Let $\mathcal{H}$ be a 2-wise independent family of hash functions mapping $[m] \to [m]$. Pick $h \in \mathcal{H}$ at random.
2. Let $t := 100/\varepsilon^2$. Maintain the t smallest pairs $(e_i, h(e_i))$ ordered by the hash value, breaking ties consistently but arbitrarily (note that we do *not* keep $(e_i, *)$ more than once).
3. Let X be the t-th smallest hash value among the pairs.
4. Return $\widetilde{\mathsf{DE}} := t \cdot m/X$.

By Proposition 8, we can generate and store h in $O(\log m)$ bits. Maintaining the t smallest pairs requires $O(t \cdot \log m)$ bits. Therefore, the algorithm uses $O(t \cdot \log m) = O(1/\varepsilon^2 \cdot \log m)$ bits.

Observe that the algorithm does not simply take $t = 1$, i.e., estimating via the minimum hash value. This is because the minimum yields higher variance than the t-th smallest, for $t > 1$, as the proof will reveal. Another characteristic of this algorithm is that, unlike most other algorithms we have seen so far in the course, it does not compute an unbiased estimator of the target DE, as $\mathbb{E}\left[\widetilde{\mathsf{DE}}\right] \neq \mathsf{DE}$.[2] We now prove the correctness of the algorithm.

Lemma 9. *In the algorithm above,* $\Pr\left(|\widetilde{\mathsf{DE}} - \mathsf{DE}| > \varepsilon \cdot \mathsf{DE}\right) \leq \frac{1}{50}$.

Proof. We will bound the upper and lower tails separately. For the lower tail, we have

$$\begin{aligned}
\Pr\left(\widetilde{\mathsf{DE}} < (1-\varepsilon) \cdot \mathsf{DE}\right) &= \Pr\left(t \cdot m/X < (1-\varepsilon) \cdot \mathsf{DE}\right) \\
&= \Pr\left(X > \frac{t \cdot m}{(1-\varepsilon) \cdot \mathsf{DE}}\right) \\
&= \Pr\left(\text{less than } t \text{ distinct elements hash to a value } \leq \frac{t \cdot m}{(1-\varepsilon) \cdot \mathsf{DE}}\right); \qquad (1)
\end{aligned}$$

The last equation holds because the only way for X, i.e., the value of t-th smallest hash-value in the stream, to be larger than some threshold $\tau := \frac{t \cdot m}{(1-\varepsilon) \cdot \mathsf{DE}}$, is that less than t numbers are hashed to $[1 : \tau)$.

For each $i \in [\mathsf{DE}]$, define the indicator variable Y_i that is 1 if the i-th distinct number seen in the stream hashes to a value $< \tau$, and 0 otherwise. Let $Y := \sum_{i=1}^{\mathsf{DE}} Y_i$. Since elements are hashed uniformly,

$$\mathbb{E}[Y_i] = \frac{t \cdot m}{(1-\varepsilon) \cdot \mathsf{DE}} \cdot \frac{1}{m} = \frac{t}{(1-\varepsilon) \cdot \mathsf{DE}},$$

[2] Recall that $\mathbb{E}\left[\frac{1}{X}\right] \neq \frac{1}{\mathbb{E}[X]}$ in general.

and therefore $\mathbb{E}[Y] = t/(1-\varepsilon)$. We can rewrite the probability from Eq (1) in terms of Y simply as $\Pr(Y < t)$. Thus,

$$\begin{aligned}
\Pr\left(\widetilde{\mathsf{DE}} < (1-\varepsilon)\cdot \mathsf{DE}\right) &= \Pr(Y < t) \\
&= \Pr(Y < (1-\varepsilon)\cdot \mathbb{E}[Y]) && \text{(since } \mathbb{E}[Y] = t/(1-\varepsilon)) \\
&\le \Pr(|Y - \mathbb{E}[Y]| \le \varepsilon\, \mathbb{E}[Y]). && (2)
\end{aligned}$$

Next we use a concentration bound on Y. Although Y is a sum of 0-1 random variables, we cannot use Chernoff bounds because our hash function h is only 2-wise independent, and thus the Y_i's are not mutually independent.[3] Instead, we can bound the variance and use Chebyshev's inequality. Recall that the variance is not linear in general, but it is if the variables are pairwise independent. Since h is 2-wise independent, it is easy to see that $Y_1, \ldots, Y_{\mathsf{DE}}$ are pairwise independent. Thus,

$$\begin{aligned}
\mathrm{Var}[Y] = \mathrm{Var}\left[\sum_{i=1}^{\mathsf{DE}} Y_i\right] &= \sum_{i=1}^{\mathsf{DE}} \mathrm{Var}[Y_i] && \text{(since } Y_i\text{'s are 2-wise independent)} \\
\le \sum_{i=1}^{\mathsf{DE}} \mathbb{E}\left[Y_i^2\right] &= \sum_{i=1}^{\mathsf{DE}} \mathbb{E}[Y_i] = \mathbb{E}[Y]. && \text{(since } Y_i\text{'s are indicator random variables and so } Y_i^2 = Y_i)
\end{aligned}$$

Using Chebyshev's inequality in Eq (2) yields

$$\Pr(|Y - \mathbb{E}[Y]| \le \varepsilon\, \mathbb{E}[Y]) \le \frac{\mathrm{Var}[Y]}{\varepsilon^2\, \mathbb{E}[Y]} \le \frac{\mathbb{E}[Y]}{\varepsilon^2\, \mathbb{E}[Y]^2} = \frac{1}{\varepsilon^2\, \mathbb{E}[Y]} = \frac{1-\varepsilon}{\varepsilon^2 t} \le \frac{1}{\varepsilon^2 t} = \frac{1}{100},$$

by the choice of t. As such, we conclude that $\Pr\left(\widetilde{\mathsf{DE}} < (1-\varepsilon)\cdot \mathsf{DE}\right) \le 1/100$.

For the upper tail, we can bound $\Pr\left(\widetilde{\mathsf{DE}} > (1+\varepsilon)\cdot \mathsf{DE}\right)$ symetrically[4]. Finally, by union bound, we have

$$\Pr\left(|\widetilde{\mathsf{DE}} - \mathsf{DE}| > \varepsilon \mathsf{DE}\right) \le 1/50,$$

concluding the proof. □

As always, we can run this algorithm $O(\ln(1/\delta))$ times, in parallel, and then pick the median to boost the probability of success up to $1-\delta$.

Theorem 10. *There is a streaming algorithm that outputs a $(1\pm\varepsilon)$-estimation of the number of distinct elements DE from a universe $[m]$ with probability at least $1-\delta$, using $O(\frac{1}{\varepsilon^2}\cdot \log m \cdot \log(1/\delta))$ bits of space.*

The distinct element problem was first studied by Flajolet and Martin in [4] (long before the formalization of the streaming model). This problem was revisited in the work of Alon, Matias, and Szegedy [1] that pioneered the streaming model. The algorithm we discussed in this section was proposed by Bar-Yossef, Jayram, Kumar, Sivakumar, and Trevisan [2] and was later improved in a series of work, culminating in the optimal algorithm of Kane, Nelson, and Woodruff [5] for this problem with space complexity $O(\frac{1}{\varepsilon^2} + \log m)$.

3 Frequency Moments Estimation

Let us now consider a general family of problems in the streaming model. Suppose we receive a stream of elements $e_1, \ldots, e_n$ from the universe $[m]$. The *frequency vector* $f \in \mathbb{N}^m$ of the stream is defined as the

[3] Although in this course we have only used Chernoff bounds that assume full independence, there are weaker variants that only require limited independence. Nevertheless, for our application it is easier (and sufficient) to simply use Chebyshev inequality instead.

[4] In this case, the argument and calculations are the same, but this is not always necessarily the case.

m-element vector such that the i-th coordinate f_i is the number of times i appears in the stream. A typical class of problems in streaming algorithms is that of computing some function of the frequency vector. Among such problems, an important class is that of *frequency moment* estimation problems: For any integer p, we define the *p-th frequency moment* as

$$F_p = \|f\|_p^p = \sum_{i=1}^{m} f_i^p.$$

We also define $F_0 := \|f\|_0$, i.e., number of non-zero entries of f, and $F_\infty = \|f\|_\infty$, i.e., the max of f.

The frequency moment estimation problem is defined as follows.

Problem 3. Given a stream of n elements from the universe $[m]$, defining a frequency vector f, and an integer parameter p, estimate F_p.

Particularly instances of this problem are the following:

- F_0 is the number of distinct elements in the stream, that is, DE.
- $F_1 = n$, the number of elements in the stream.
- $F_2 = \|f\|_2^2$. While F_2 may not have a 'direct' interpretation as F_0 and F_1, it is related to the *skewness* of the distribution of elements. Similar to the interpretation of the squared two-norm of a probability distribution as the probability of collision (Lecture 4), we can think of $\|f\|_2^2$ as a measure of non-uniformity (for an example of a practical application of F_2 used for anomaly detection in network traffic analysis, see [6]).
- $F_\infty := \max_i f_i$ is the most frequent element. This value is directly related to *heavy-hitters* detection in networks, which are flows with an abnormally large number of packets.

We already know how to estimate F_0, and F_1 can be trivially computed exactly in $O(\log n)$ bits[5]. Moreover, we will (hopefully) show later in the course that approximating F_p, with $p \geq 3$ (including $p = \infty$), requires $\Omega(n^{1-2/p})$ bits of space, and thus there is poly-logarithmic space algorithm for values of p larger than 2. What about F_2? This is the topic of our next section.

3.1 Second Frequency Moment Estimation

We turn our attention to the specific problem of estimating F_2. Let $f^{(i)}$ be the frequency vector after i elements have been received. The trivial algorithm for this problem is to maintain the entire $f^{(i)}$ and update it at each step, which requires $O(m \log n)$ bits. What if we only maintain a small "sketch" of $f^{(i)}$, that uses less bits, and that allows us to recover most of $f^{(i)}$?

Linear sketching: To realize this idea, we define a *linear sketch of* $f^{(i)}$ as a vector $Z \cdot f^{(i)}$, where Z is a $t \times m$ matrix chosen *independently* of f (and almost always randomly). Linear sketches are easy to maintain in the streaming model: since the frequency vector is updated linearly as $f^{(i)} = f^{(i-1)} + E_{e_i}$—$E_j$ is the m-coordinate vector that has 1 in the j-th coordinate and 0 everywhere else—we can update $Z \cdot f^{(i)} = Z \cdot f^{(i-1)} + Z \cdot E_{e_i}$ by linearity, hence, at the end of the stream have $Z \cdot f$.

In this approach, the number of rows t of Z controls the tradeoff between the number of bits required by the sketch and the "power" of $Z \cdot f^{(i)}$ in recovering information about $f^{(i)}$ – the smaller the t, the less bits are required, but (presumably) the less information we will recover from $f^{(i)}$. Finally, since Z is often always entirely consists of random bits, we typically store Z *implicitly* using limited-independence hash functions (as was done in the previous section).

[5] We can get away with $O(\log \log n + \log (1/\varepsilon))$ bits if we settle for a $(1 \pm \varepsilon)$-approximation [7, 3].

Remark. Linear sketching is an extremely powerful and versatile tool used in numerous areas of theoretical computer science (and beyond) such as: (*i*) streaming algorithms as we will see in this course, (*ii*) distributed algorithms that work on separate machines with limited communication: each machine can send a linear sketch of its input to a central coordinator (instead of the original input thus saving in communication) and the coordinator can merge all sketches by linearity and solve the problem, (*iii*) faster algorithms: by sketching the data first and then working on the much smaller sketch of the data instead of the original one, we can obtain faster algorithms, (*iv*) compressed sensing and sparse recovery as we already saw in Lecture 6 even though we did not name them explicitly as such.

We are now going to design a linear sketch for the F_2 estimation problem.

Algorithm:

1. Let $\mathcal{H}$ be a 4-wise independent family of hash functions mapping $[m] \rightarrow \{-1, 1\}$ and $t := 70/\varepsilon^2$.
2. For each $i \in [t]$, define the i-th row of the matrix $Z \in \{-1, 1\}^{t \times m}$ as follows:
   - Pick $h_i \in \mathcal{H}$ at random (independently of other rows) and let $Z_i := [h_i(1)\ h_i(2)\ \ldots\ h_i(m)]$.
3. Initialize the frequency vector f to all zeros. Update $Z \cdot f$ after every element of the stream using the linearity of the sketch described earlier.
4. At the end of the stream, compute $s_i := Z_i \cdot f$ for $i \in [t]$ and return $Y := \frac{1}{t} \sum_{i=1}^{t} s_i^2$.

The space complexity of this algorithm is $O(t \cdot \log m)$ bits for storing Z using t independent 4-wise independent hash functions (each $O(\log m)$ bits) and another $O(t \cdot \log n)$ bits for storing the t-dimensional vector $Z \cdot f$ (i.e., the sketch). Hence, the space complexity is $O(t \cdot (\log n + \log m)) = O(\frac{1}{\varepsilon^2} \cdot (\log n + \log m))$ bits.

We now prove the correctness of the algorithm in the following lemma.

Lemma 11. *For the random variable Y of the sketching algorithm,*

$$\Pr\left(|Y - F_2| \geq \varepsilon \cdot F_2\right) \leq \frac{1}{10}.$$

We prove this by computing expected value and bounding variance of each s_i^2 (which are distributed identically), extending these bounds to Y (as an average of t such random variables), and using Chebyshev's inequality to conclude the proof. We start with the expected values.

Claim 12. *Let s be distributed as any s_i in the algorithm; then, $\mathbb{E}\left[s^2\right] = F_2$.*

Proof. For brevity, in the following, we use $\overrightarrow{Z} = [z_1, \ldots, z_m]$ to denote a random row of the matrix Z such that $s = \overrightarrow{Z} \cdot f$. Note that by definition,

$$s^2 = \left(\sum_{i=1}^{m} z_i \cdot f_i\right)^2 = \sum_{i=1}^{m} \sum_{j=1}^{m} z_i \cdot z_j \cdot f_i \cdot f_j.$$

Using this, we can write,

$$\begin{aligned} \mathbb{E}\left[s^2\right] &= \sum_{i=1}^{m} \sum_{j=1}^{m} \mathbb{E}\left[z_i \cdot z_j\right] \cdot f_i \cdot f_j && \text{(by linearity of expectation)} \\ &= \sum_{i=1}^{m} \mathbb{E}\left[z_i^2\right] \cdot f_i^2 + \sum_{i \neq j} \mathbb{E}\left[z_i \cdot z_j\right] \cdot f_i \cdot f_j. \end{aligned}$$

Now note that $\mathbb{E}[z_i \cdot z_j] = \mathbb{E}[z_i] \cdot \mathbb{E}[z_j]$ for $i \neq j$ because z-values are 4-wise independent and thus z_i, z_j are independent (for this part of argument, 2-wise independence suffices and we only need 4-wise independence for later in the proof). We can thus simplify the above equation to,

$$\begin{aligned}
\mathbb{E}\left[s^2\right] &= \sum_{i=1}^{m} \mathbb{E}\left[z_i^2\right] \cdot f_i^2 + \sum_{i \neq j} \mathbb{E}[z_i] \cdot \mathbb{E}[z_j] \cdot f_i \cdot f_j && \text{(as } z_i \perp z_j \text{ for } i \neq j\text{)} \\
&= \sum_{i=1}^{m} f_i^2 && \text{(as } \mathbb{E}\left[z_i^2\right] = 1 \text{ and } \mathbb{E}[z_i] = \mathbb{E}[z_j] = 0\text{)} \\
&= F_2,
\end{aligned}$$

as desired. ☐

Claim 13. *Let s be distributed as any s_i in the algorithm; then,* $\mathbb{E}\left[s^2\right] \leq 7 \cdot F_2^2$.

Proof. Similar to the previous part, we have,

$$\begin{aligned}
\mathrm{Var}\left[s^2\right] \leq \mathbb{E}\left[s^4\right] &= \sum_{i,j,k,\ell} \mathbb{E}[z_i \cdot z_j \cdot z_k \cdot z_\ell] \cdot f_i \cdot f_j \cdot f_k \cdot f_\ell && \text{(by linearity of expectation)} \\
&= \sum_{i=1}^{m} \mathbb{E}\left[z_i^4\right] \cdot f_i^4 + \binom{4}{2} \sum_{i \neq j} \mathbb{E}\left[z_i^2 \cdot z_j^2\right] f_i^2 \cdot f_j^2 + \binom{4}{1} \sum_{i \neq j \neq k} \mathbb{E}\left[z_i^2 \cdot z_j \cdot z_k\right] f_i^2 \cdot f_j \cdot f_k \\
&\quad + \binom{4}{2} \sum_{i \neq j} \mathbb{E}\left[z_i^3 \cdot z_j\right] f_i^3 \cdot f_j + \sum_{i \neq j \neq k \neq \ell} \mathbb{E}[z_i \cdot z_j \cdot z_k \cdot z_\ell] \cdot f_i \cdot f_j \cdot f_k \cdot f_\ell
\end{aligned}$$

We can now again use the fact that z-values are 4-wise independent (this time we do need the 4-wise independence) and simplify the expectation terms as product of different terms. As such,

$$\begin{aligned}
\mathrm{Var}\left[s^2\right] &\leq \sum_{i=1}^{m} \mathbb{E}\left[z_i^4\right] \cdot f_i^4 + \binom{4}{2} \sum_{i \neq j} \mathbb{E}\left[z_i^2\right] \mathbb{E}\left[z_j^2\right] f_i^2 \cdot f_j^2 + \binom{4}{1} \sum_{i \neq j \neq k} \mathbb{E}\left[z_i^2\right] \mathbb{E}[z_j] \mathbb{E}[z_k] f_i^2 \cdot f_j \cdot f_k \\
&\quad + \binom{4}{2} \sum_{i \neq j} \mathbb{E}\left[z_i^3\right] \mathbb{E}[z_j] f_i^3 \cdot f_j + \sum_{i \neq j \neq k \neq \ell} \mathbb{E}[z_i] \mathbb{E}[z_j] \mathbb{E}[z_k] \mathbb{E}[z_\ell] \cdot f_i \cdot f_j \cdot f_k \cdot f_\ell \\
&= \sum_{i=1}^{m} f_i^4 + 6 \sum_{i \neq j} f_i^2 \cdot f_j^2 && \text{(}\mathbb{E}\left[z_i^4\right] = \mathbb{E}\left[z_i^2\right] = 1 \text{ and } \mathbb{E}\left[z_i^3\right] = \mathbb{E}[z_i] = 0\text{)} \\
&\leq (\sum_{i=1}^{m} f_i^2)^2 + 6 \cdot (\sum_{i=1}^{m} f_i^2)^2 \\
&= 7F_2^2,
\end{aligned}$$

as desired. ☐

We can now conclude the proof of Lemma 11.

Proof of Lemma 11. By linearity of expectation and Claim 12, $\mathbb{E}[Y] = F_2$, and since $\mathrm{Var}[Y] = \frac{1}{t}\mathrm{Var}[s]$ (by independence of s_i^2 variables) and Claim 13, $\mathrm{Var}[Y] \leq 7F_2^2/t$. Thus, by Chebyshev's inequality,

$$\begin{aligned}
\Pr(|Y - F_2| \geq \varepsilon \cdot F_2) &= \Pr(|Y - \mathbb{E}[Y]| \geq \varepsilon \cdot F_2) \\
&\leq \frac{7F_2^2}{t \cdot \varepsilon^2 \cdot F_2^2} = \frac{1}{10},
\end{aligned}$$

by the choice of $t = 70/\varepsilon^2$, finalizing the proof. ☐

As before, we can always boost the probability of success of this algorithm by running it $O(\ln(1/\delta))$ in parallel and taking the median answer. In terms of linear sketching, this corresponds to use a sketching matrix $Z' \in \{-1,1\}^{t' \times m}$ where $t' = O(\ln(1/\delta)) \cdot t$ and each $O(\ln(1/\delta))$ consecutive rows form one independent instance of the sketching matrix Z in the algorithm of this section. For recovery, we first recover the answer to each of Z-sub-matrices using the algorithm of this section and then take a median (so in a sense, the final answer is a proper combination of median and average of squared of entities of the sketching vector).

We can now conclude the following theorem.

Theorem 14. *There is a linear sketching streaming algorithm that outputs a $(1 \pm \varepsilon)$-estimation of F_2 with probability at least $1-\delta$, using $O(\frac{1}{\varepsilon^2} \cdot (\log m + \log n) \cdot \log(1/\delta))$ bits of space.*

This result was also first proved in the seminal paper of Alon, Matias, and Szegedy [1].

References

[1] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 20–29, 1996. 10, 14

[2] Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, D. Sivakumar, and Luca Trevisan. Counting distinct elements in a data stream. In *Randomization and Approximation Techniques, 6th International Workshop, RANDOM 2002, Cambridge, MA, USA, September 13-15, 2002, Proceedings*, pages 1–10, 2002. 10

[3] Philippe Flajolet. Approximate counting: A detailed analysis. *BIT*, 25(1):113–134, 1985. 11

[4] Philippe Flajolet and G. Nigel Martin. Probabilistic counting. In *24th Annual Symposium on Foundations of Computer Science, Tucson, Arizona, USA, 7-9 November 1983*, pages 76–82, 1983. 10

[5] Daniel M. Kane, Jelani Nelson, and David P. Woodruff. An optimal algorithm for the distinct elements problem. In *Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2010, June 6-11, 2010, Indianapolis, Indiana, USA*, pages 41–52, 2010. 10

[6] Balachander Krishnamurthy, Subhabrata Sen, Yin Zhang, and Yan Chen. Sketch-based change detection: methods, evaluation, and applications. In *Proceedings of the 3rd ACM SIGCOMM Internet Measurement Conference, IMC 2003, Miami Beach, FL, USA, October 27-29, 2003*, pages 234–247, 2003. 11

[7] Robert H. Morris Sr. Counting large numbers of events in small registers. *Commun. ACM*, 21(10):840–842, 1978. 11