CS 466/666: Algorithm Design and Analysis **University of Waterloo: Fall 2024**

Lecture 3

September 12, 2024

Instructor: Sepehr Assadi

Topics of this Lecture

1 Streaming Distinct Elements Problem

Our next step in this course is to see an example of Chebyshev's inequality in designing randomized algorithm in the *streaming* model of computation. 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.

1.1 The Streaming Model of Computation

We now define the streaming model formally as introduced by Alon, Matias, and Szegedy in [AMS96][1]. The input consists of n elements $e_1, e_2, \ldots, e_n$, where each e_i belongs to some universe $\mathcal{U}$ of m elements, 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).

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 $\text{poly}(\log n, \log m)$ bits.

[1]The 2005 Gödel Prize—one of the highest awards in Theoretical Computer Science—was awarded to Noga Alon, Yossi Matias, and Mario Szegedy for the introduction of this model. See the citation of the award here:

`https://eatcs.org/index.php/component/content/article/503`

Warm-Up: Before getting to our main problem, let us mention a standard warm-up puzzle:

- You are given $n-1$ *distinct* numbers from $[n]$ in a stream in some arbitrary order. Find the missing element in $O(\log n)$ bits of space.
- If you like to challenge yourself further, consider the same problem where you are given $n-k$ distinct numbers from $[n]$ and the goal is to find the k missing elements in $O(k^2 \cdot \log n)$ bits of space.
- If you really like to challenge yourself, solve the above problem in $O(k \log n)$ bits of space.

We will leave the answer to these questions as an exercise to the reader (*Note:* the first two questions are simple enough but the last one might be quite challenging without the "right" background—do not let that discourage you however!).

1.2 Distinct Elements Counting Problems

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

Problem 1. 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.

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

What about an algorithm using $\text{poly}(\log n, \log m)$ bits? While we will not cover this topic in this course, one can show that this is not possible without randomization and approximation, by proving:

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

Therefore, to find a sublinear space streaming algorithm we need to allow for both approximation and randomization. In addition, to make the problem a bit easier for us in this lecture, we will consider a relaxed version of the problem, sometimes called *threshold testing* (variant) of the problem, defined as follows:

Problem 2. At the beginning of the stream, you are given a value $\widetilde{\mathsf{DE}} \in [m]$ and a parameter $\varepsilon \in (0, 1)$. Then, you are given a stream of n numbers (possibly with repetitions) from a universe $[m]$. The goal is to, with probability at least $2/3$, output *Yes* if $\mathsf{DE} \geqslant \widetilde{\mathsf{DE}}$ and output *No* if $\mathsf{DE} < (1-\varepsilon) \cdot \widetilde{\mathsf{DE}}$; if the value of DE is between these two numbers, either answer is considered correct.

In this lecture, we will see an algorithm that solves this problem using

$$\text{poly}(\log n, \log m, 1/\varepsilon)$$

bits of space, which is much more efficient than the naive approaches above.

Simplifying assumption. Without loss of generality, we can assume that $\widetilde{\mathsf{DE}} \geqslant 100/\varepsilon^2$. This is because otherwise, we can simply use the deterministic $O(\widetilde{\mathsf{DE}} \cdot \log m)$-space naive algorithm that stores all the distinct elements it sees and answers *Yes* as soon as it sees $\widetilde{\mathsf{DE}} + 1$ of them; when $\widetilde{\mathsf{DE}} < 100/\varepsilon^2$, this algorithm only requires $O(\log m/\varepsilon^2)$ space which is sufficient for our purpose.

1.3 The Algorithm

Before getting to the actual algorithm, let us provide a vague intuition. Suppose there is a dartboard in front of us and each time we throw a dart at this board, the probability we hit our target is some $1/K$. Then, if we could throw "much more" than K darts, we *expect* to hit our target many more times compared to when we throw "much less" than K darts at this board. What does this have anything to do with our problem?

We are going to design a randomized process wherein each *distinct* element in the stream allows us to throw a new dart which hits the target with probability $\approx 1/\widetilde{\mathsf{DE}}$; we then, simply count the number of times we hit the target. If $\mathsf{DE} \geqslant \widetilde{\mathsf{DE}}$, we expect this counter to be "large" whereas when $\mathsf{DE} < (1-\varepsilon) \cdot \widetilde{\mathsf{DE}}$, we expect it to be a "small" number. How do we simulate this random dart process? By hashing the *universe* to a certain range! The algorithm is formally as follows.

Algorithm 1. An algorithm for threshold testing distinct elements for a given $\widetilde{\mathsf{DE}} \in [m]$ and $\varepsilon \in (0,1)$:

(i) Let $t := 12/\varepsilon^2$ and pick a **hash function** $h : [m] \to [\widetilde{\mathsf{DE}}/t]$ uniformly at random from the set of all functions from $[m] \to [\widetilde{\mathsf{DE}}/t]$.

(ii) Count the number of *distinct* elements in the stream that are hashed to 1, i.e., count the size of the set $\{e \mid h(e) = 1\}$, denoted by X.

(iii) Return *Yes* if the number X calculated above is at least $(1-\varepsilon/2) \cdot t$ or *No* otherwise.

Going back to our intuition from earlier, for each *distinct* number in the stream, $h(\cdot)$ has a chance of hitting 1 with probability $t/\widetilde{\mathsf{DE}}$. As such, if $\mathsf{DE} \geqslant \widetilde{\mathsf{DE}}$, then it is very likely that a "good number" of the element will be hashed to 1, but if $\mathsf{DE} \leqslant (1-\varepsilon) \cdot \widetilde{\mathsf{DE}}$ that number should be considerably lower (both cases in a probabilistic sense).

You may ask what is the role of t then, i.e., why did we pick t to be $\Omega(1/\varepsilon^2)$ and not simply, say, 1?[2] This is done for the purpose of *"variance reduction"*: see the calculation for the variance of the random variables and how it is used in the analysis to see the necessity of using a larger t.

We will talk about the space complexity of this algorithm later in the lecture. 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. Although the rest of the algorithm uses $O(t \cdot \log m) = O(\log m/\varepsilon^2)$ bits of space which is good enough for us. Regardless, almost all the main ideas of the algorithm are already here and thus we focus on proving its correctness in the following.

1.4 Formal Analysis

We now formalize this intuition and prove the correctness of the algorithm.

Lemma 1. *On any input stream and for any choice of parameters $\varepsilon \in (0,1)$ and $\widetilde{\mathsf{DE}} \geqslant 100/\varepsilon^2$:*

- *If $\mathsf{DE} \geqslant \widetilde{\mathsf{DE}}$, then Algorithm 1 outputs Yes with probability at least $2/3$;*
- *If $\mathsf{DE} < (1-\varepsilon) \cdot \widetilde{\mathsf{DE}}$, then Algorithm 1 outputs No with probability at least $2/3$.*

[2]Notice that we ideally want t to be as small as possible as the space of the algorithm depends linearly on t.

Proof. Let $\{e_1, \ldots, e_{\mathsf{DE}}\}$ denote the distinct elements in the stream. For $i \in [\mathsf{DE}]$, define the *indicator* random variable $X_i \in \{0,1\}$ which is 1 iff $h(e_i) = 1$. Under this definition, the random variable X in Algorithm 1 is:

$$X = \sum_{i=1}^{\mathsf{DE}} X_i.$$

We can thus calculate the expected value of X as follows:

$$\begin{aligned}
\mathbb{E}[X] &= \mathbb{E}\left[\sum_{i=1}^{\mathsf{DE}} X_i\right] && \text{(by the equation above)} \\
&= \sum_{i=1}^{\mathsf{DE}} \mathbb{E}[X_i] && \text{(by linearity of expectation)} \\
&= \sum_{i=1}^{\mathsf{DE}} \Pr(h(e_i) = 1) && \text{(by the definition of indicator } X_i\text{)} \\
&= \sum_{i=1}^{\mathsf{DE}} \frac{t}{\widetilde{\mathsf{DE}}} && \text{(as } h(\cdot) \text{ maps each element to a uniformly random position in } [\widetilde{\mathsf{DE}}/t]\text{)} \\
&= \mathsf{DE} \cdot \frac{t}{\widetilde{\mathsf{DE}}}. && (1)
\end{aligned}$$

For our analysis, we also need to bound the variance of X, which can be done as follows:

$$\begin{aligned}
\mathrm{Var}[X] &= \mathrm{Var}\left[\sum_{i=1}^{\mathsf{DE}} X_i\right] && \text{(again, by the equation } X = \textstyle\sum_i X_i\text{)} \\
&= \sum_{i=1}^{\mathsf{DE}} \mathrm{Var}[X_i] && \text{(variance of sum of \textit{independent} variables is sum of variances)} \\
&\leqslant \sum_{i=1}^{\mathsf{DE}} \mathbb{E}\left[X_i^2\right] && \text{(by the definition of variance } \mathrm{Var}[X_i] = \mathbb{E}\left[X_i^2\right] - (\mathbb{E}[X_i])^2\text{)} \\
&= \sum_{i=1}^{\mathsf{DE}} \mathbb{E}[X_i] && \text{(as } X_i^2 = X_i \text{ since } X_i \in \{0,1\}\text{)} \\
&= \mathbb{E}[X]. && (2)
\end{aligned}$$

We can now analyze each case separately.

Case I: when $\mathsf{DE} \geqslant \widetilde{\mathsf{DE}}$: In this case, by the bound on the expectation in Eq (1), we have,

$$\mathbb{E}[X] \geqslant t.$$

The algorithm will say *No* if $X < (1 - \varepsilon/2) \cdot t$; in this case, this requires X to deviate by at least $(\varepsilon/2) \cdot \mathbb{E}[X]$ from its expectation. This is exactly the topic of concentration inequalities. In particular, recall Chebyshev's inequality from the last lecture:

Proposition 2 (Chebyshev's Inequality). *For any random variable X and $b > 0$,*

$$\Pr(|X - \mathbb{E}[X]| \geqslant b) \leqslant \frac{\mathrm{Var}[X]}{b^2}.$$

We can apply Chebyshev's inequality to our random variable X to get:

$$\begin{aligned}
\Pr\left(\text{Algorithm 1 says } \textit{No} \text{ in Case I}\right) &\leqslant \Pr\left(\left|X - \mathbb{E}\left[X\right]\right| > (\varepsilon/2)\cdot\mathbb{E}\left[X\right]\right) && \text{(as described above)}\\
&\leqslant \frac{4\cdot\mathrm{Var}\left[X\right]}{\varepsilon^2\cdot\mathbb{E}\left[X\right]^2} && \text{(by Chebyshev's inequality of Proposition 2 for } b=(\varepsilon/2)\cdot\mathbb{E}\left[X\right])\\
&\leqslant \frac{4}{\varepsilon^2\cdot\mathbb{E}\left[X\right]} && \text{(by the upper bound of } \mathbb{E}\left[X\right] \text{ on variance in Eq (2))}\\
&\leqslant \frac{4}{\varepsilon^2\cdot(12/\varepsilon^2)} && \text{(by the lower bound of } t \text{ on expectation and the choice of } t=12/\varepsilon^2)\\
&= \frac{1}{3}.
\end{aligned}$$

This proves the first bullet of the lemma statement.

Case II: when $\mathsf{DE} < (1-\varepsilon)\cdot\widetilde{\mathsf{DE}}$: In this case, by the bound on the expectation in Eq (1), we have,

$$\mathbb{E}\left[X\right] < (1-\varepsilon)\cdot t.$$

The algorithm will say *Yes* if $X \geqslant (1-\varepsilon/2)\cdot t$; in this case, this requires X to deviate by at least $(\varepsilon/2)\cdot t$ from its expectation.[3] We thus have,

$$\begin{aligned}
\Pr\left(\text{Algorithm 1 says } \textit{Yes} \text{ in Case II}\right) &\leqslant \Pr\left(\left|X - \mathbb{E}\left[X\right]\right| > (\varepsilon/2)\cdot t\right) && \text{(as described above)}\\
&\leqslant \frac{4\cdot\mathrm{Var}\left[X\right]}{\varepsilon^2\cdot t^2} && \text{(by Chebyshev's inequality of Proposition 2 for } b=(\varepsilon/2)\cdot\mathbb{E}\left[X\right])\\
&\leqslant \frac{4\cdot\mathbb{E}\left[X\right]}{\varepsilon^2\cdot t^2} && \text{(by the upper bound of } \mathbb{E}\left[X\right] \text{ on variance in Eq (2))}\\
&\leqslant \frac{4\cdot(1-\varepsilon)}{\varepsilon^2\cdot(12/\varepsilon^2)} && \text{(by the upper bound of } \mathbb{E}\left[X\right] < (1-\varepsilon)\cdot t \text{ and the choice of } t=12/\varepsilon^2)\\
&< \frac{1}{3}.
\end{aligned}$$

This proves the second bullet of the lemma statement and concludes the whole proof. □

Thus, the algorithm has a probability of success of at least 2/3 for our problem, as desired. In the next part, we see how to fix the issue with the space complexity of the problem—in particular, how to replace the hash function h with something that we can store more efficiently.

2 Independence for Space: Limited-Independence Hash Functions

Generating and storing a random function $h : [a] \to [b]$ requires $O(a \log b)$ bits, which is often too costly. In the case of Algorithm 1 this amounts to $\omega(m)$ space which makes the whole algorithm entirely useless! Fortunately however, we can make the analysis of Algorithm 1 work even if the hash function h we picked is not *completely random*, but has some *limited independence* only. Let us define this formally as follows.

[3] Notice that this time, we bound the deviation as a function of t and *not* $\mathbb{E}\left[X\right]$; this is because, in this case, it is possible for $\mathbb{E}\left[X\right]$ to be very small and thus bounding the deviation just as a function of expectation will be too weak. You are also encouraged to check that in the previous case, bounding the deviation as a function of t does *not* work (because $\mathbb{E}\left[X\right]$ can be much larger than t in that case).

Definition 3. 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 that 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 $k \in [m]$.

Proposition 4. *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. *for every $i \in [k]$, $h(x_i)$ is uniform over $[b]$;*
2. *$h(x_1), \ldots, h(x_k)$ are mutually independent.*

Proof. 1. We only prove this for $i = 1$; the rest is symmetric. 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

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

This concludes the proof. □

Example. Let $k \geqslant 2$ be an integer and $p > k$ be a prime number. Here is an example of a k-wise independent family of hash functions mapping $[p] \to [p]$

A k-wise Independent Family of Hash Functions on $[p] \to [p]$ for a prime p:

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

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

2. Sample $c_0, \ldots, c_{k-1} \in \mathbb{F}_p$ and return h as the polynomial defined by these 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/course).

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 5. *For any integers $a, b \geqslant 1$, there exists a k-wise independent family of hash functions mapping $[a] \to [b]$, that requires $O(k \cdot (\log a + \log b))$ bits.*

2.1 Improving Space Complexity of Algorithm 1

To make Algorithm 1 space-efficient, we are going to replace the truly random hash function $h : [m] \to [\widetilde{\mathsf{DE}}/t]$ with a *pair*-wise independent hash function instead. By Proposition 5, this means we can store h in only $O(\log m)$ bits (as $\widetilde{\mathsf{DE}}/t \leqslant m$) and evaluate $h(\cdot)$ on each arriving element quickly and in a space-efficient manner still. The rest of the algorithm also needed $O(\log m/\varepsilon^2)$ bits, thus we now have a truly space-efficient algorithm for the problem.

But, what about the analysis? There were only two key places that we used the properties h:

- In Eq (1) to compute the expected value of X. In particular, we used the fact that for each element e in the universe, $\Pr(h(e_i) = 1) = t/\widetilde{\mathsf{DE}}$, namely, that $h(e_i)$ is distributed uniformly over the range of h. This continue to hold for any pair-wise independent hash function also as argued in Proposition 4.

  As an aside, this is an easy property to satisfy that holds even for "weaker" hash families, say *universal* hash families, or even *uniform* hash families—for instance, consider the family of functions $h : [a] \to [b]$ consisting of b functions $\{h_i \mid h_i(x) = i \;\forall x \in [a]\}$; this is obviously a "bad" hash family that maps *all* the elements to the same position, and still even this family is enough to satisfy the property.

- In Eq (2) to say that variance of the sum is equal to sum of the variances as X_i's are independent. We no longer have the independence property here. However, the conclusion, namely, that variance of the sum is equal to the sum of variances holds even for pairwise independent variables—which X_i's are because they are deterministic functions $h(e_i)$'s, which are pairwise independent—as we prove below.

  Proposition 6. *Let $X_1, \ldots, X_n$ be a family of n* pair-wise *independent variables. Then,*

  $$\mathrm{Var}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathrm{Var}\left[X_i\right].$$

  Proof. We have

  $$\begin{aligned}\mathrm{Var}\left[\sum_{i=1}^{n} X_i\right] &= \mathbb{E}\left[(\sum_{i=1}^{n} X_i)^2\right] - (\mathbb{E}\left[\sum_{i=1}^{n} X_i\right])^2 && \text{(by the definition of variance)}\\ &= \mathbb{E}\left[\sum_{i=1}^{n} X_i^2 + \sum_{i \neq j} X_i \cdot X_j\right] - \sum_{i=1}^{n} \mathbb{E}\left[X_i\right]^2 - \sum_{i \neq j} \mathbb{E}\left[X_i\right] \cdot \mathbb{E}\left[X_j\right] \\ & && \text{(by expanding the sums)}\\ &= \left(\sum_{i=1}^{n} \mathbb{E}\left[X_i^2\right] - \mathbb{E}\left[X_i\right]^2\right) + \left(\sum_{i \neq j} \mathbb{E}\left[X_i \cdot X_j\right] - \mathbb{E}\left[X_i\right] \cdot \mathbb{E}\left[X_j\right]\right) \\ & && \text{(by linearity of expectation and re-ordering the terms)}\\ &= \sum_{i=1}^{n} \mathrm{Var}\left[X_i\right] + 0,\end{aligned}$$

  (first term by definition of variance, second term because $X_i \perp X_j$ for pair-wise independent variables)

  which proves the result. □

This means that the modification of Algorithm 1, by replacing h with a pairwise independent hash functions, works exactly as before and thus solves our problem, but now with a much better space. This proves the following theorem.

Theorem 7. *There is a streaming algorithm for threshold testing number of distinct elements in Problem 2 that uses $O(\log m/\varepsilon^2)$ space and outputs the correct answer with probability at least $2/3$.*

3 The Original Distinct Elements Problem?

What about the original problem: estimating the number of distinct elements instead of threshold testing?

There is a simple way to go from Theorem 7 to that problem: run the algorithm of Theorem 7 *in parallel* for different thresholds $\widetilde{\mathsf{DE}} \in \{1, (1+\varepsilon), (1+\varepsilon)^2, (1+\varepsilon)^3, \ldots, m\}$; then, find the *largest* choice of $\widetilde{\mathsf{DE}}$ for which the algorithm returns *Yes*, and output that as the estimate of DE for the stream. Notice that this increasing the space by a factor of $O(\log_{(1+\varepsilon)}(m) = O((\log m)/\varepsilon)$ which is okay for our purpose. *Assuming* every application of Theorem 7 in this process is also correct, it is easy to see that the returned answer will be a $(1+\Theta(\varepsilon))$-approximation of DE – we can then use a smaller ε in the algorithm, if needed, by changing the space with a constant factor, and obtain a truly $(1+\varepsilon)$-approximation.

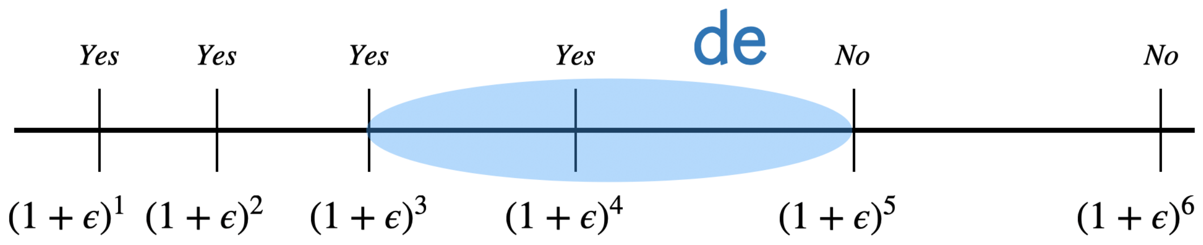

Figure 1: An illustration of the reduction from original estimation problem to threshold testing. Assuming all *Yes*/*No* responses are correct, the true estimate should lie somewhere in the marked (blue) region—the reason for the gap is to account for the fact that in the threshold testing, the answer can be arbitrary when neither case are satisfied.

An important caveat is that, as stated, Theorem 7 only guarantees $2/3$ probability of success and thus it is not going to be the case that all of its $O((\log m)/\varepsilon)$ invocations in this strategy return a correct answer. As such, we first need to *boost* the probability of success of the algorithm to a larger value before running this strategy—this will be the content of a future lecture.

Finally, we note that the above strategy of going from threshold testing to the original estimation problem, as well as boosting success of randomized algorithms, is a completely generic idea that has nothing to do with the distinct element problem we studied, and can be applied to most other settings and problems.

Remark. The distinct element problem—in this formulation—was first studied by Flajolet and Martin in [FM83], long before the formalization of the streaming model, as was revisited in the work of Alon, Matias, and Szegedy [AMS96] that pioneered the streaming model.

The algorithm we discussed in this lecture was inspired by an algorithm of Bar-Yossef, Jayram, Kumar, Sivakumar, and Trevisan [BJK+02]. This algorithm was later improved in a series of work, culminating in the asymptotically optimal algorithm of Kane, Nelson, and Woodruff [KNW10] with space complexity $O(\frac{1}{\varepsilon^2} + \log m)$ bits.

References

[AMS96] 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. 1, 8

[BJK$^+$02] 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, MA, USA, September 13-15, 2002*, pages 1–10, 2002. 8

[FM83] Philippe Flajolet and G. Nigel Martin. Probabilistic counting. In *24th Annual Symposium on Foundations of Computer Science, Arizona, USA, 7-9 November 1983*, pages 76–82, 1983. 8

[KNW10] 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. 8