Chapter 4

Streaming Algorithms I: Distinct Elements

Our next step is to see a few more applications of the probabilistic toolkits we developed in previous chapters to designing randomized algorithms. For this, we go to the *streaming* model of computation.

4.1 Streaming Distinct Elements Problem

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.

The Model. We now define the streaming model formally as introduced in [AMS96]. The input consists of n elements $e_1, e_2, \ldots, e_n$, where each e_i belongs to some universe $[m]$ of 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.

Before getting to our main problem, let us mention a standard warm-up puzzle in the streaming model (see also Exercise 4.2):

- **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.

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

Problem 4.1. Given a stream of n elements from the universe $[m]$, output the number of *distinct* elements in the stream, denoted by F_0 (this quantity is also often called the 0-*th frequency moment* of the input).

For example, if $m = 5$, and the stream is $1, 2, 2, 1, 5, 4, 2, 2, 1$, then the answer is $F_0 = 4$.

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 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 directly cover this topic in this book, 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 chapter, we will consider a relaxed version of the problem, sometimes called **threshold testing** variant of the problem, defined as follows:

Problem 4.2. At the beginning of the stream, you are given a value $\widetilde{F_0} \in [m]$ 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 $F_0 \geqslant \widetilde{F_0}$;
- *No* if $F_0 < (1 - \varepsilon) \cdot \widetilde{F_0}$; and,
- either *Yes* or *No* if F_0 is between these two thresholds.

We prove the following result for this problem.

Theorem 4.3. *There is a streaming algorithm for Problem 4.2 that uses $O(\log m/\varepsilon^2)$ space and outputs the answer correctly with probability at least $2/3$.*

Theorem 4.3 gives a much more efficient solution than the naive approaches described earlier.

Simplifying assumption ($\widetilde{F_0} \geqslant 100/\varepsilon^2$). When proving Theorem 4.3, we can assume that $\widetilde{F_0} \geqslant 100/\varepsilon^2$ without loss of generality. This is because otherwise, we can simply use the deterministic $O(\widetilde{F_0} \cdot \log m)$-space naive algorithm that stores all the distinct elements it sees and answers *Yes* as soon as it sees $\widetilde{F_0}$ of them; when $\widetilde{F_0} < 100/\varepsilon^2$, this algorithm only requires $O(\log(m)/\varepsilon^2)$ space which is sufficient for our purpose.

4.1.1 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 "far fewer" than K darts at this board. What does this have 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 $\approx_\varepsilon 1/\widetilde{F_0}$; we then simply count the number of times we hit the target. If $F_0 \geqslant \widetilde{F_0}$, we expect this counter to be "large" whereas when $F_0 < (1-\varepsilon)\cdot\widetilde{F_0}$, we expect it to be a "small" number. How do we simulate this random dart process? By hashing the *universe* to a certain range! Formally,

Algorithm 4.4. An algorithm for threshold testing distinct elements in Problem 4.2 for a given $\widetilde{F_0} \in [m]$ and $\varepsilon \in (0,1)$:

(*i*) Let $t := 12/\varepsilon^2$ and pick a **hash function** $h : [m] \to [\widetilde{F_0}/t]$ uniformly at random from the set of all functions from $[m] \to [\widetilde{F_0}/t]$ (we will revisit this step later in Section 4.2).

(*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.

Note: In the algorithm, we simply store at most $(1-\varepsilon/2)\cdot t \leqslant t$ distinct elements hashed to 1 and if even more numbers are hashed, we simply ignore them, as we can already answer *Yes* based on the stored elements.

Going back to our intuition from earlier, for each *distinct* number in the stream, $h(\cdot)$ has a chance of hitting 1 with probability $t/\widetilde{F_0}$. As such, if $F_0 \geqslant \widetilde{F_0}$, then it is very likely that a "good number" of the elements will be hashed to 1, but if $F_0 < (1-\varepsilon)\cdot\widetilde{F_0}$ that number should be considerably lower (both cases in a probabilistic sense). See Figure 4.1 for an illustration.

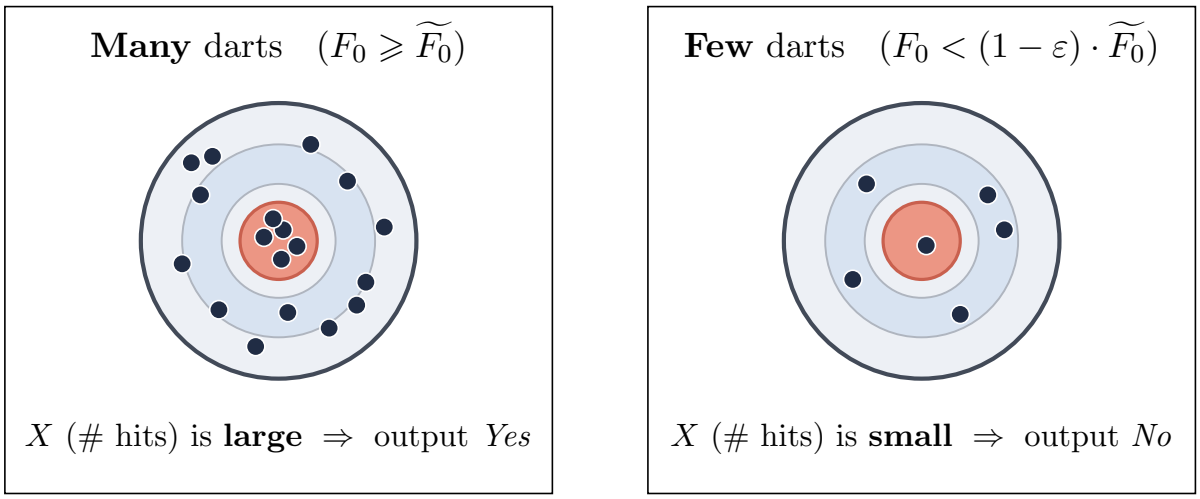

Figure 4.1: Each distinct element is a dart that hits the bullseye with probability $t/\widetilde{F_0}$. The counter X is the number of hits, which concentrates around its mean $\mathbb{E}\left[\widetilde{F_0}\right] = F_0 \cdot t/\widetilde{F_0}$.

You may ask what is the role of t, i.e., why did we pick t to be $\Theta(1/\varepsilon^2)$ and not, say, 1?[1] 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 need for a larger t. We will get back to this after the proof as this points to a general probabilistic analysis technique that is worth highlighting more.

We will talk about the space complexity of this algorithm later. For now, we should only note

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

that *as it is*, this algorithm requires prohibitively large space to store the hash function h and hence is not space-efficient. The rest of the algorithm however 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.

4.1.2 Formal Analysis

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

Lemma 4.5. *On any input stream and for any choice of parameters $\varepsilon \in (0,1)$ and $\widetilde{F_0} \geqslant 100/\varepsilon^2$:*

- *If $F_0 \geqslant \widetilde{F_0}$, then Algorithm 4.4 outputs Yes with probability at least $2/3$;*
- *If $F_0 < (1-\varepsilon) \cdot \widetilde{F_0}$, then Algorithm 4.4 outputs No with probability at least $2/3$.*

Proof. Let $\{e_1, \ldots, e_{F_0}\}$ denote the distinct elements in the stream. For $i \in [F_0]$, 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 4.4 is:

$$X = \sum_{i=1}^{F_0} X_i.$$

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

$$\begin{aligned}
\mathbb{E}[X] &= \mathbb{E}\left[\sum_{i=1}^{F_0} X_i\right] = \sum_{i=1}^{F_0} \mathbb{E}[X_i] && \text{(by the equation above and linearity of expectation)} \\
&= \sum_{i=1}^{F_0} \Pr(h(e_i) = 1) && \text{(by the definition of indicator } X_i\text{)} \\
&= \sum_{i=1}^{F_0} \frac{t}{\widetilde{F_0}} && \text{(as } h(\cdot) \text{ maps each element to a uniformly random position in } [\widetilde{F_0}/t]\text{)} \\
&= F_0 \cdot \frac{t}{\widetilde{F_0}}. && (4.1)
\end{aligned}$$

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

$$\begin{aligned}
\mathrm{Var}[X] &= \mathrm{Var}\left[\sum_{i=1}^{F_0} X_i\right] && \text{(again, by the equation } X = \textstyle\sum_i X_i\text{)} \\
&= \sum_{i=1}^{F_0} \mathrm{Var}[X_i] && \text{(by linearity of variance for } \textit{independent} \text{ variables in Fact 2.7)} \\
&\leqslant \sum_{i=1}^{F_0} \mathbb{E}\left[X_i^2\right] && \text{(by the definition of variance } \mathrm{Var}[X_i] = \mathbb{E}\left[X_i^2\right] - (\mathbb{E}[X_i])^2\text{)} \\
&= \sum_{i=1}^{F_0} \mathbb{E}[X_i] && \text{(as } X_i^2 = X_i \text{ since } X_i \in \{0,1\}\text{)} \\
&= \mathbb{E}[X]. && (4.2)
\end{aligned}$$

We can now analyze each case separately.

Case I: when $F_0 \geqslant \widetilde{F_0}$: In this case, by the bound on the expectation in Eq (4.1), 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 because:

$$\begin{aligned} X &< (1-\varepsilon/2)\cdot t = (1-\varepsilon/2)\cdot t - \mathbb{E}[X] + \mathbb{E}[X] \\ &\leqslant (1-\varepsilon/2)\cdot\mathbb{E}[X] - \mathbb{E}[X] + \mathbb{E}[X] && \text{(as } \mathbb{E}[X] \geqslant t \text{ in this case)} \\ &= -(\varepsilon/2)\,\mathbb{E}[X] + \mathbb{E}[X], \end{aligned}$$

which implies that $\mathbb{E}[X] - X > (\varepsilon/2)\cdot\mathbb{E}[X]$, and thus $|X - \mathbb{E}[X]| > (\varepsilon/2)\cdot\mathbb{E}[X]$ as well.

Thus, we are now precisely at the domain of concentration inequalities. In particular, recall Chebyshev's inequality of Proposition 2.5. We can apply Chebyshev's inequality to our random variable X to get:

$$\begin{aligned} \Pr(\text{Algorithm 4.4 says } \textit{No} \text{ in Case I}) &\leqslant \Pr(|X-\mathbb{E}[X]| > (\varepsilon/2)\cdot\mathbb{E}[X]) && \text{(as described above)} \\ &\leqslant \frac{4\cdot\mathrm{Var}[X]}{\varepsilon^2\cdot\mathbb{E}[X]^2} \\ &&& \text{(by Chebyshev's inequality of Proposition 2.5 for } b = (\varepsilon/2)\cdot\mathbb{E}[X]) \\ &\leqslant \frac{4}{\varepsilon^2\cdot\mathbb{E}[X]} \\ &&& \text{(by the upper bound of } \mathbb{E}[X] \text{ on variance in Eq (4.2))} \\ &\leqslant \frac{4}{\varepsilon^2\cdot(12/\varepsilon^2)} = \frac{1}{3}. \\ &&& \text{(by the lower bound of } t \text{ on expectation and the choice of } t = 12/\varepsilon^2) \end{aligned}$$

This proves the first bullet of the lemma statement.

Case II: when $F_0 < (1-\varepsilon)\cdot\widetilde{F_0}$: In this case, by the bound on the expectation in Eq (4.1),

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

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.[2] We thus have,

$$\begin{aligned} \Pr(\text{Algorithm 4.4 says } \textit{Yes} \text{ in Case II}) &\leqslant \Pr(|X-\mathbb{E}[X]| > (\varepsilon/2)\cdot t) && \text{(as described above)} \\ &\leqslant \frac{4\cdot\mathrm{Var}[X]}{\varepsilon^2\cdot t^2} \\ &&& \text{(by Chebyshev's inequality of Proposition 2.5 for } b = (\varepsilon/2)\cdot t) \\ &\leqslant \frac{4\cdot\mathbb{E}[X]}{\varepsilon^2\cdot t^2} \\ &&& \text{(by the upper bound of } \mathbb{E}[X] \text{ on variance in Eq (4.2))} \\ &\leqslant \frac{4\cdot(1-\varepsilon)}{\varepsilon^2\cdot(12/\varepsilon^2)} < \frac{1}{3}. \\ &&& \text{(by the upper bound of } \mathbb{E}[X] < (1-\varepsilon)\cdot t \text{ and the choice of } t = 12/\varepsilon^2) \end{aligned}$$

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

[2] Notice that this time, we bound the deviation as a function of t and *not* $\mathbb{E}[X]$; this is because, in this case, it is possible for $\mathbb{E}[X]$ 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 $\mathbb{E}[X]$ can be much larger than t in that case).

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.

Remark. Let us now revisit the choice of $t \approx 1/\varepsilon^2$. As we showed in the calculations above—and it is also easy to see directly—there is a gap of $\varepsilon \cdot t$ in the expected value of the random variable X we use in Algorithm 4.4 between the *Yes* case and *No* case. But then, for this "difference in expectation" to materialize as a probabilistic bound, we need this quantity to be *larger* than the *standard deviation* (square root of variance) of X: this is needed so that "typical" statistical changes in the value of X do not "override" the difference in expectation we have. But then, the standard deviation of X with t values is $\approx \sqrt{t}$ in our case. Thus, we need $\varepsilon t \gtrsim \sqrt{t}$ which tells us we need $t \gtrsim 1/\varepsilon^2$. This is precisely what our analysis did formally also.

4.2 Independence for Space: 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 4.4 this amounts to $\omega(m)$ space which makes the whole algorithm entirely useless! Fortunately however, we can make the analysis of Algorithm 4.4 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.

Definition 4.6. A family $\mathcal{H} = \{h : [a] \to [b]\}$ is called a ***k*-wise independent** family of hash functions if for all pairwise distinct $x_1, \dots, x_k \in [a]$ and all $y_1, \dots, y_k \in [b]$,

$$\Pr_{h \sim \mathcal{H}} \left(\bigwedge_{i=1}^{k} h(x_i) = y_i \right) = \frac{1}{b^k}.$$

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 it typically does).

Proposition 4.7. *Let $\mathcal{H} = \{h : [a] \to [b]\}$ be a k-wise independent family, and $h \sim \mathcal{H}$ chosen at random. Let $x_1, \dots, x_k \in [a]$ be arbitrary pairwise distinct elements. Then:*

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

Proof. 1. We only prove this for $i = 1$; the rest is symmetric. Let $y_1 \in [b]$. Observe that

$$\Pr_{h \sim \mathcal{H}}(h(x_1) = y_1) = \sum_{y_2, \dots, y_k \in [b]} \Pr_{h \sim \mathcal{H}} \left(\bigwedge_{i=1}^{k} h(x_i) = y_i \right) = \sum_{y_2, \dots, y_k \in [b]} \frac{1}{b^k} = \frac{b^{k-1}}{b^k} = \frac{1}{b},$$

by partitioning the sample space for the first equality and applying the definition of k-wise independent family for the second one.

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

$$\Pr_{h \sim \mathcal{H}} \left(\bigwedge_{i=1}^{k} h(x_i) = y_i \right) = \frac{1}{b^k} = \prod_{i=1}^{k} \Pr_{h \sim \mathcal{H}}(h(x_i) = y_i). \qquad \square$$

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):

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

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

So, why this is a k-wise independent family? 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 $x_1, \ldots, x_{k-1}$, the 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 book).

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. We will not prove this result here; see, e.g. [CW79] for a proof.

Proposition 4.8. *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.*

4.2.1 Improving Space Complexity of Algorithm 4.4

To make Algorithm 4.4 space-efficient, we are going to replace the truly random hash function $h : [m] \to [\widetilde{F_0}/t]$ with a *pair*-wise independent hash function instead. By Proposition 4.8, this means we can store h in only $O(\log m)$ bits (as $\widetilde{F_0}/t \leqslant m$) 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 properties of h:

- In Eq (4.1) to compute the expected value of X. In particular, we used the fact that for each element e in the universe, $\Pr(h(e) = 1) = t/\widetilde{F_0}$, namely, that $h(e)$ is distributed uniformly over the range of h. This continues to hold for any pairwise independent hash function also as proven in Proposition 4.7.

  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 (4.2) 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 4.9. *Let $X_1, \ldots, X_n$ be a family of n* pairwise *independent variables. Then,*

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

Proof. We have

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

which proves the result. □

Therefore the modification of Algorithm 4.4, by replacing h with a pairwise independent hash functions, works as before and solves our problem, but with a much better space of $O(\log(m)/\varepsilon^2)$ bits. This concludes the proof of Theorem 4.3.

4.3 The Original Distinct Elements Problem?

There is a simple way to go from Theorem 4.3 to the original problem of estimating the number of distinct elements in Problem 4.1: run the algorithm of Theorem 4.3 *in parallel* for thresholds

$$\widetilde{F_0} \in \left\{1, (1+\varepsilon), (1+\varepsilon)^2, (1+\varepsilon)^3, \ldots, m\right\};$$

then, find the *largest* choice of $\widetilde{F_0}$ for which the algorithm returns *Yes*, and output it as the estimate of F_0 for the stream. This increases 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 4.3 in this process is also correct, it is easy to see that the returned answer will be a $(1+3\varepsilon)$-approximation of F_0. 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. See Figure 4.2 for an illustration.

An important caveat is that, as stated, Theorem 4.3 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. But, we already know how to do this: run $O(\log m)$ copies of this algorithm in parallel on the stream, and return the median answer. As we showed previously in Problem 2.11, this now ensures that the new algorithm errs with probability at most $m^{-\Theta(1)}$. Thus, we can do a union bound over all choices of threshold,

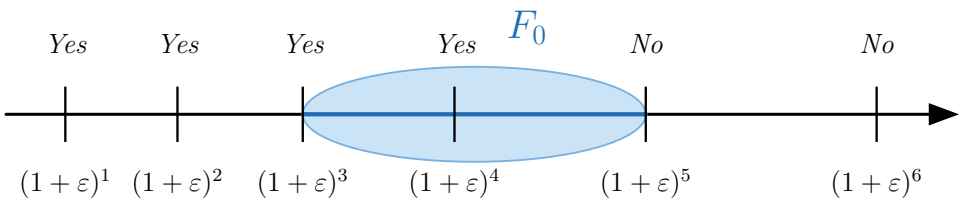

Figure 4.2: An illustration of the reduction from the estimation problem to threshold testing. Assuming all *Yes*/*No* responses are correct, the true estimate should lie somewhere in the marked (blue) region. The reason for the gap is that in threshold testing, the answer is arbitrary when neither case is satisfied.

of which there are fewer than m; thus, the probability that even one fails can be upper bounded by $m^{-\Theta(1)}$ and thus the above strategy works. Overall, this leads to an algorithm with space

$$O(\frac{\log m}{\varepsilon^2}) \cdot O(\frac{\log m}{\varepsilon}) \cdot O(\log m) = \text{poly}(\log m, 1/\varepsilon)\,.$$

In the exercises, we see a better strategy for this problem that uses fewer $\log m$ and $1/\varepsilon$ factors.

Finally, we note that the above strategy of going from threshold testing to the original estimation problem is a generic idea and can be applied to many other settings and problems.

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

The algorithm we discussed in this chapter was inspired by an algorithm of [BJK+02]. This algorithm was later improved in a series of papers, culminating in the asymptotically optimal algorithm of [KNW10] with space complexity $O(\frac{1}{\varepsilon^2} + \log m)$ bits.

Exercises

Exercise 4.1. The following streaming algorithm, called **reservoir sampling**, is for sampling an element uniformly at random from a stream of elements $e_1, e_2, \cdots$ with an *unknown* length.

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.

Prove that this algorithm outputs an element uniformly at random from the stream. Determine the space complexity of this algorithm.

Exercise 4.2. This exercise generalizes the warm-up puzzle from the beginning of the chapter. You are given $n-k$ *distinct* numbers from $[n]$, arriving in a stream of length $n-k$ in an arbitrary order, and your goal is to output the k *missing* numbers once the stream ends.

1. For $k = 1$, show how to recover the single missing number in $O(\log n)$ bits of space.
2. For general k, suppose that, as the stream arrives, you maintain the power sums $\sum_x x^j$ of the elements x seen so far for all $j \in [k]$. Show how to recover all k missing numbers from these quantities, and prove that the algorithm uses $O(k^2 \log n)$ bits of space.

3. (♦) Improve the space to $O(k \log n)$ bits for the general k case.
4. (♦) Improve the space even further to $O(k \log (n/k))$ bits for the general k case. Then, prove that this is the information-theoretically optimal space for this problem.

Exercise 4.3. Early in the chapter we asserted, without proof, that computing the number of distinct elements *exactly* and *deterministically* requires large space. You will prove this in this exercise.

Let $\mathcal{A}$ be a deterministic streaming algorithm that, on every stream of elements from $[m]$, outputs the exact number of distinct elements, and suppose the stream length may be as large as $n \geqslant m$. Prove that $\mathcal{A}$ must use $\Omega(m)$ bits of memory.

Hint: run $\mathcal{A}$ on the elements of a subset $S \subseteq [m]$ of size $m/2$, each fed once. If $\mathcal{A}$ uses fewer than $\log_2 \binom{m}{m/2} = m - O(\log m)$ bits, then two *distinct* subsets $S \neq S'$ of size $m/2$ leave $\mathcal{A}$ in the identical memory state. Pick an element $a \in S \setminus S'$, append it to both streams, and compare the true answers to what $\mathcal{A}$ must output.

Exercise 4.4 (♦). Extend the proof in Exercise 4.3 to prove that even deterministic algorithms that output a 1.01-approximation to the number of distinct elements require $\Omega(m)$ space.

Hint: Instead of working with all subsets $S \subseteq [m]$ with size $m/2$ work with a "large" family $\mathcal{F}$ of them that are additionally "far" from each other, namely, for $S_i, S_j \in \mathcal{F}$ for $i \neq j$, $|S_i \cap S_j| \leqslant m/3$. Show that there is a family $\mathcal{F}$ with $2^{\Omega(m)}$ many sets and use this to prove the result (you will also need to modify the "appending" part of the previous argument).