# CS 170 — Lecture: Streaming Algorithms and Counting Distinct Elements

## 1 The streaming model

Picture the log of a big web service: IP addresses hitting a router, search queries arriving at a server, words of a text appearing one after another. The data is enormous — far bigger than the memory of the machine processing it — but we only need a few summary statistics, not the data itself. The streaming model is the right abstraction.

**Definition 1** (Streaming model)**.** The input is a sequence $x_1, x_2, \ldots, x_N$ of elements from some universe $\mathcal{U}$. An algorithm sees the elements one at a time, in order. After seeing $x_i$, it updates a small state (using only $s \ll N$ bits of memory) and discards $x_i$. At the end, it outputs an answer based on its final state.

The crucial constraint is the memory budget. Concretely, we will try to use only $O(\log N)$ or $O(\log^2 N)$ bits — tiny compared to storing the whole stream.

**Examples we will, and won't, handle.** Counting the total length $N$ of the stream is easy: maintain a counter, which takes $O(\log N)$ bits. Computing the *sum* of the elements (if they are numbers) is similarly easy. These are "easy" because they decompose additively. The questions we care about today are harder:

- (Today's warmup) Count $N$ in even *less* memory — $O(\log \log N)$ bits — if we are willing to accept an approximate answer. This is *Morris's algorithm.*
- (Main event) Count the number of *distinct* elements seen in the stream, again using $O(\log^2 N)$ bits. Storing the set of distinct elements takes $\Omega(N \log |\mathcal{U}|)$ bits, so some cleverness is needed.

Both rely on randomization. In both, the hash functions we built last lecture will be the source of our random bits.

## 2 Warmup: Morris's approximate counter

We want to count $N$, the length of the stream, using much less than $\log N$ bits. That sounds impossible, because $N$ itself takes $\log N$ bits to write down. The trick is: we will store a surrogate $X$ that is roughly $\log_2 N$ — and $\log_2 N$ takes only $\log \log N$ bits.

> **Morris's algorithm.** Initialize $X \leftarrow 0$. On each stream element: with probability $2^{-X}$, set $X \leftarrow X + 1$. Otherwise leave $X$ unchanged. At the end, output the estimate $\widehat{N} = 2^X - 1$.

Intuition: $X$ goes up less and less often as it grows, so it "slows down" in exactly the right way to track $\log_2 N$.

**Proposition 2.** *After processing $n$ stream elements, $\mathbb{E}[2^{X_n}] = n + 1$. So $\widehat{N} = 2^{X_n} - 1$ is an unbiased estimator of $n$.*

*Proof.* Let $X_n$ denote the value of $X$ after $n$ elements. Condition on $X_n$; the next update rule gives

$$\mathbb{E}\big[2^{X_{n+1}} \mid X_n\big] \;=\; 2^{-X_n} \cdot 2^{X_n+1} \;+\; (1 - 2^{-X_n}) \cdot 2^{X_n} \;=\; 2 \cdot 1 \;+\; 2^{X_n} - 1 \;=\; 2^{X_n} + 1.$$

Taking expectations, $\mathbb{E}[2^{X_{n+1}}] = \mathbb{E}[2^{X_n}]+1$. Since $\mathbb{E}[2^{X_0}] = 2^0 = 1$, by induction $\mathbb{E}[2^{X_n}] = n+1$. □

So $\widehat{N}$ has the right expectation. How much memory did we use? $X_n$ is typically about $\log_2 n$, so storing $X_n$ takes about $\log \log n$ bits. For a stream of $n = 2^{32}$ elements, that is about 5 bits of state — incredibly cheap.

**Remark 3.** We will not prove a concentration bound for Morris's estimator — an unbiased estimator is all we are claiming. The algorithm is already a little surprising: we just reduced counting from $\log N$ to $\log \log N$ bits at the cost of some variance.

## 3 The count-distinct problem

The real target of the lecture is the following.

**Definition 4** ($F_0$ / count-distinct)**.** Given a stream $x_1, x_2, \ldots, x_N \in \mathcal{U}$, let

$$d \;=\; \big|\{\, x_1, x_2, \ldots, x_N \,\}\big|$$

be the number of *distinct* elements in the stream. The goal is to estimate $d$ in small space.

Why is this nontrivial? Several natural ideas fail:

- *Maintain a counter, incrementing when we see a "new" element.* To know if $x_i$ is new we would have to compare against everything we have seen, i.e. store the whole stream.
- *Maintain an array of $|\mathcal{U}|$ bits, one per possible element.* That is linear in the universe, which may be $2^{64}$ IP addresses or every possible 128-bit user ID.
- *Sample elements at random.* If one element repeats a million times and another appears once, a uniform sample will nearly always miss the rare one.

We are going to solve this using a single random hash value per distinct element. The whole algorithm fits in a few lines.

## 4 The min-hash algorithm

Let $\mathcal{H}$ be a pairwise-independent hash family $\mathcal{U} \to \{0, 1, \ldots, M-1\}$, as constructed last lecture, with $M$ chosen much larger than any $d$ we could encounter (say $M = N^3$). For convenience of analysis we will think of the output as a real number in $[0, 1]$ via $h(x)/M$; pretend, in this lecture, that $h(x)/M$ is a uniform element of $[0, 1]$.

**Min-hash algorithm.**

1. Pick $h \in \mathcal{H}$ uniformly at random, at the start of the stream.
2. Maintain a single variable $V$, initialized to $V \leftarrow +\infty$ (in practice, to $M$).
3. On each stream element $x$: set $V \leftarrow \min(V,\ h(x))$.
4. At the end: output $\widehat{d} = M/V - 1$.

Total state: the description of $h$ (two elements of $\mathbb{F}_p$, i.e. $O(\log N)$ bits) plus the single variable $V$ (another $O(\log N)$ bits). So the algorithm uses $O(\log N)$ bits of memory.

Notice a key property: $V$ only depends on the *set* of distinct stream elements, not on multiplicities. If $x$ appears 17 times, $h(x)$ is the same each time; feeding it in once or 17 times produces the same $V$. Duplicates are harmless — which is exactly what we need.

## 5 Why the algorithm works (in expectation)

Let $y_1, \ldots, y_d$ be the distinct elements in the stream, and let $Y_i = h(y_i)/M \in [0,1]$ be their (idealized-uniform) hash values. Then at the end,

$$V/M \ = \ \min_{1 \le i \le d} Y_i.$$

Write $U \coloneqq V/M = \min_i Y_i$, the minimum of $d$ uniform $[0,1]$ random variables.

**Lemma 5.** *If $Y_1, \ldots, Y_d$ are independent and uniform on $[0,1]$, then $\mathbb{E}[\min_i Y_i] = \frac{1}{d+1}$.*

*Proof.* For any nonnegative random variable $U \in [0,1]$,

$$\mathbb{E}[U] \ = \ \int_0^1 \mathbb{P}[U \ge t]\, dt.$$

For the minimum of $d$ independent uniforms, $\mathbb{P}[U \ge t] = \mathbb{P}[\text{all } Y_i \ge t] = (1-t)^d$. So

$$\mathbb{E}[U] \ = \ \int_0^1 (1-t)^d\, dt \ = \ \left[ -\frac{(1-t)^{d+1}}{d+1} \right]_0^1 \ = \ \frac{1}{d+1}. \qquad \square$$

Plugging in: the final value of $V/M$ has expectation $1/(d+1)$. Inverting, the estimator $M/V - 1$ is "morally" $d$ — we should read this as: *the minimum hash value $V/M$ is a sharp ruler whose typical size is $1/(d+1)$, and reading off its reciprocal recovers $d$.*

**Remark 6** (Honesty about the analysis)**.** Two things are worth flagging, because a careful student will notice them.

First, Lemma **??** assumed the $Y_i$ are *fully* independent, whereas our hash family only guarantees pairwise independence. The pairwise case can be analysed rigorously, but it is messier — it is exactly the kind of second-moment argument this lecture is trying to avoid. In practice pairwise-independent hashing "behaves like uniform" well enough for this algorithm, and this is precisely the gap where Flajolet–Martin and HyperLogLog improve the analysis.

Second, $\mathbb{E}[M/V]$ is not exactly $1/\mathbb{E}[V/M]$, because expectation does not commute with $t \mapsto 1/t$. So strictly speaking $\widehat{d} = M/V - 1$ is not an unbiased estimator of $d$; its *typical value* is near $d$, and that is what the one-line derivation above captures. Making this precise is another version of the concentration question we are skipping.

## 6 A worked example

Suppose $M = 1000$, and the stream is

$$3,\ 5,\ 3,\ 9,\ 3,\ 5,\ 2.$$

The distinct elements are $\{2, 3, 5, 9\}$, so the true answer is $d = 4$.

Say our random hash gave the values

$$h(3) = 720, \quad h(5) = 280, \quad h(9) = 410, \quad h(2) = 195.$$

Then after processing the stream, $V = \min = h(2) = 195$, and we output

$$\widehat{d} \;=\; M/V - 1 \;=\; 1000/195 - 1 \;\approx\; 4.13.$$

Very close to $d = 4$. Of course a different random $h$ could have given, say, $h(2) = 900$, in which case $V = h(5) = 280$ and $\widehat{d} = 1000/280 - 1 \approx 2.57$, an underestimate. Lemma **??** says that on average it comes out right.

## 7 Putting it together

For the count-distinct problem:

- We cannot afford to store the distinct elements directly.
- A pairwise-independent hash — the same family we built last lecture — lets us replace each distinct element by a random number in $[0, M)$, and the *smallest* such number captures the count $d$ "for free," through the identity $\mathbb{E}[\min_i Y_i] = 1/(d+1)$.
- The algorithm uses one hash function and one number: $O(\log N)$ bits of state.
- We proved only that the estimator is correct on average. A concentration bound, and a practical variant (HyperLogLog), build on exactly this foundation.

The moral of these two lectures together: *a well-chosen random hash function is a source of structured randomness that costs almost nothing to store and that makes worst-case inputs look average.* In Lecture 1 that bought us $O(1)$-time hash tables; today it bought us a distinct-count algorithm in logarithmic space.

---

*Acknowledgements: the min-hash estimator is due to Flajolet–Martin (1985); the analysis and presentation follow the standard textbook treatment (cf. past CS 170 notes and Dasgupta–Papadimitriou–Vazirani).*