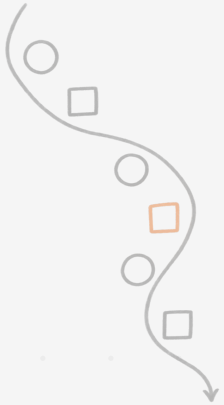


New Computational Models ▸ Streaming Algorithms

Lecture 16

- Background & Motivation
- Heavy Hitters
- Distinct Elements





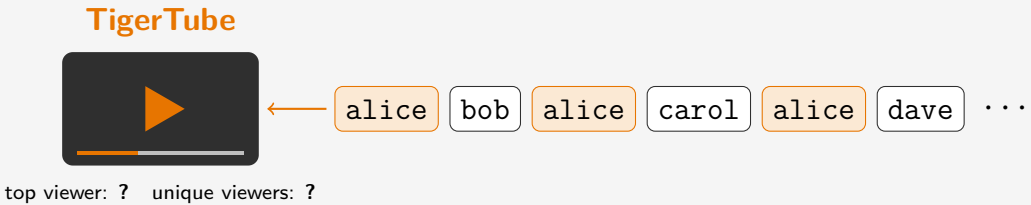
New Computational Models ▸ Streaming Algorithms

Lecture 16

- Background & Motivation
- Heavy Hitters
- Distinct Elements

Toy Problem: Building TigerTube

Suppose you are building **TigerTube**, the hot new platform for Princeton videos: lecture recordings, a cappella concerts, squirrels. Every time someone opens a video, your server sees their ID:



Under each video, you want to know, live:

- Is one account behind many of the views? (Maybe someone is refreshing their own video 500 times.)
- How many *different* people have watched it?

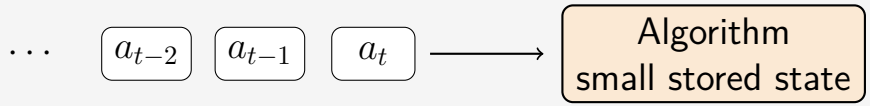
The catch: TigerTube goes viral! A single video can get billions of views, so you cannot afford the memory to store its whole viewer list, and the counts must update as views happen.

For each video, you see each view once, in order, and can keep only a small summary. This is the **streaming model**.

The Streaming Model

Elements come from a universe Σ with $|\Sigma| = N$, and arrive one at a time as a stream $a_1, a_2, a_3, \ldots$ with each $a_i \in \Sigma$.

(TigerTube: Σ = all user IDs, a_i = the user behind the i-th view.)



Definition: Streaming algorithm

- When a_t arrives, update the state using only the old state and a_t. Then a_t is gone: we **cannot** read it again.
- The stream length is **not** known in advance.
- At every time t, answer a question about $a_1, \ldots, a_t$ using only the state.

We measure **space**: the number of bits in the state, as a function of t and N.

Example: Maintaining the Sum

Take the universe $\Sigma = \{1, \ldots, N\}$. We want the sum of all elements seen so far.

Algorithm: Running sum

Initialize $sum \leftarrow 0$.

When a_t arrives, update

$$sum \leftarrow sum + a_t.$$

One variable is enough, but that is not the same as constant space: the number of bits needed to store sum grows with t.

Space Is Measured in Bits

Since $|\Sigma| = N$, an element of Σ can be encoded using $\lceil \log_2 N \rceil$ bits.

After t arrivals,

$$0 \leq sum \leq tN.$$

The sum register therefore needs at most

$$\lceil \log_2(tN + 1) \rceil = O(\log_2 t + \log_2 N)$$

bits.

The stream grows by one whole element at every step; the memory needed for its sum grows only logarithmically.

Example: Maintaining the Maximum

Again take $\Sigma = \{1, \ldots, N\}$.

Algorithm: Running maximum

Initialize $max \leftarrow 0$.

When a_t arrives:

- If $a_t > max$, set $max \leftarrow a_t$.
- Otherwise, discard a_t.

We store one value between 0 and N:

$$\text{space} = \lceil \log_2(N+1) \rceil = O(\log_2 N).$$

What about maintaining the **median**? Think about what we would need to remember.

The Space We Are Aiming For

Storing even one arbitrary element of Σ takes $\Theta(\log_2 N)$ bits. For tasks that must retain an element, this is a natural benchmark.

A trivial solution: Store the entire stream.

$$\text{space} = O(t \log_2 N).$$

Our goal: Use much less space, ideally close to $O(\log_2 t + \log_2 N)$.

Two useful ideas: allow **approximation**, and use **hashing and randomness** to build small summaries.





New Computational Models ▸ Streaming Algorithms

Lecture 16

- Background & Motivation
- Heavy Hitters
- Distinct Elements

Problem: Heavy Hitters

Now Σ is any universe with $|\Sigma| = N$ (e.g., TigerTube user IDs). Fix a threshold $0 < \varepsilon < 1$.

Definition: Frequency and heavy hitters

After t arrivals, the frequency of $e \in \Sigma$ is

$$\mathrm{Count}_t(e) = \big|\{i \in \{1, \ldots, t\} : a_i = e\}\big|.$$

An element e is an ε-heavy hitter if

$$\mathrm{Count}_t(e) > \varepsilon t.$$

Goal: At every time t, maintain a short list containing all the ε-heavy hitters.

How Many Heavy Hitters Can There Be?

Lemma: Few heavy hitters

At every time $t > 0$, the number of ε-heavy hitters is at most $\lceil 1/\varepsilon \rceil - 1$.

Proof

Let H be the set of heavy hitters, and assume $H \neq \emptyset$. Their occurrences are disjoint, so

$$t \;\geq\; \sum_{e \in H} \mathrm{Count}_t(e) \;>\; |H| \cdot \varepsilon t.$$

Hence $|H| < 1/\varepsilon$, and the largest integer strictly below $1/\varepsilon$ is $\lceil 1/\varepsilon \rceil - 1$.

So the list we want to output is always short, even if the stream is very long.

First: Finding a Majority Element

A majority element is a $1/2$-heavy hitter: it occurs more than $t/2$ times.

Algorithm: One candidate and one counter

Initialize $memory \leftarrow \perp$ and $counter \leftarrow 0$.

When a_t arrives:

1. If $counter = 0$, set $memory \leftarrow a_t$ and $counter \leftarrow 1$.
2. Otherwise, if $a_t = memory$, increment the counter.
3. Otherwise, decrement the counter and discard a_t.

Return the element in $memory$ as the candidate.

Caveat: If a majority element exists, it is always returned (we prove this soon). But the output may **not** be a majority element: on a, b, c the algorithm returns c.

Exact Answers: A Hard Stream

Exact answers can be expensive, even for a majority element. Consider the stream

$$\underbrace{s_1, s_2, \ldots, s_m}_{\text{distinct, forming a set } S} \;,\; \underbrace{x, x, \ldots, x}_{m \text{ copies}}.$$

Here $t = 2m$, and x is a majority element exactly when $x \in S$: its count is $m + 1 > t/2$ if $x \in S$, and only m otherwise.

Different sets need different memory states.

- Suppose $S \neq S'$ lead to the same state. Pick $x \in S \setminus S'$.
- The algorithm then sees the same m copies of x, so it gives the same answer in both cases. But x is a majority only after S.

There are $\binom{N}{m}$ possible sets, and b bits give only 2^b states:

$$b \;\geq\; \log_2 \binom{N}{m} \;\geq\; \log_2 \left(\frac{N}{m}\right)^m \;\geq\; m = \frac{t}{2} \qquad \text{when } N \geq 2m.$$

The Relaxation: No False Negatives

Exact answers can need memory linear in t, so we relax the goal: we allow the list to contain extra elements.

- **False positives are allowed:** An output element need not actually be a heavy hitter.
- **False negatives are not allowed:** Every heavy hitter must appear in the output.

Our approximation is a short **candidate list** that never misses a heavy hitter.

There Is at Most One Majority Element

Lemma: Uniqueness

At every time $t > 0$, there is at most one majority element.

Proof

If two distinct elements e and f were both majority elements, then

$$\mathrm{Count}_t(e) > t/2, \qquad \mathrm{Count}_t(f) > t/2.$$

Their occurrences are disjoint, so

$$t \geq \mathrm{Count}_t(e) + \mathrm{Count}_t(f) > t,$$

a contradiction.

Why Decrementing the Counter Works

Algorithm: Recall: one candidate and one counter

When a_t arrives:

1. If $counter = 0$, set $memory \leftarrow a_t$ and $counter \leftarrow 1$.
2. Otherwise, if $a_t = memory$, increment the counter.
3. Otherwise, decrement the counter and discard a_t.

Interpret the counter as the number of *uncanceled copies* of the element in memory.

When a different element arrives, cancel it against one stored copy:

one ~~stored copy of~~ x

new ~~element~~ ~~$y \neq x$~~

The processed stream can be partitioned into canceled pairs of distinct elements and the copies still represented by the counter.

Every time we cancel a copy of the true majority element, we also cancel a different element.

The Majority Element Survives

Lemma: Correctness

At every time $t > 0$, if a majority element exists, it is in memory.

Proof

Suppose the majority element e is not in memory. All of its more than $t/2$ occurrences must have been canceled.

Each was canceled together with a distinct occurrence of some element other than e. Thus more than

$$t/2 + t/2 = t$$

elements were canceled, a contradiction.

If there is no majority element, the algorithm makes no claim about its candidate.

Space for the Majority Algorithm

- **Memory:** One element of Σ (or the empty marker). This uses $O(\log_2 N)$ bits.
- **Counter:** An integer between 0 and t. This uses $O(\log_2 t)$ bits.

$$\text{total space} = O(\log_2 N + \log_2 t).$$

The key idea is **cancellation**. Next, we keep several candidates instead of one to find all the ε-heavy hitters.

Heavy Hitters: Back to the General Problem

Set

$$K = \left\lceil \frac{1}{\varepsilon} \right\rceil - 1.$$

Keep K slots, each storing a candidate and a counter:

$$T[1], \ldots, T[K], \qquad C[1], \ldots, C[K].$$

Initially, all slots are empty and all counters are 0. A slot with counter 0 is treated as empty.

The majority algorithm used one candidate. Now we keep a bounded list of candidates and cancel larger groups.

Algorithm: Finding Heavy Hitters

Algorithm: Update when a_t arrives

1. If a_t is already in an occupied slot j, increment $C[j]$.
2. Otherwise, if an empty slot j exists, set
$$T[j] \leftarrow a_t, \qquad C[j] \leftarrow 1.$$
3. Otherwise, decrement all K counters by 1 and discard a_t. Treat any slot whose counter becomes 0 as empty.

Output the candidates in the occupied slots.

Exercise: Check that for $\varepsilon = 1/2$, this is the majority algorithm, with a zero counter treated as an empty slot.

How Accurate Are the Counters?

Definition: Estimated frequency

After t arrivals, define

$$\mathrm{est}_t(e) = \begin{cases} C[j] & \text{if } e = T[j] \text{ in an occupied slot } j, \\ 0 & \text{if } e \text{ is not stored.} \end{cases}$$

Lemma: Additive error

For every element e and every time t,

$$0 \le \mathrm{Count}_t(e) - \mathrm{est}_t(e) \le \frac{t}{K+1} \le \varepsilon t.$$

Proof: Count the Cancellation Steps

Goal: For every e and t, $0 \le \mathrm{Count}_t(e) - \mathrm{est}_t(e) \le \frac{t}{K+1} \le \varepsilon t$.

$\mathrm{Count}_t(e)$: true number of occurrences of e so far. $\mathrm{est}_t(e)$: counter of e (0 if not stored).

Proof

Let D_t be the number of decrement-all steps in the first t arrivals, and let S_t be the sum of the counters.

- An ordinary update increases the counter sum by 1.
- A decrement-all step decreases the sum by K and discards the arriving element: it cancels $K+1$ elements in total.

The sum starts at 0. Of the t arrivals, D_t are decrement-all steps (each $-K$) and the other $t - D_t$ are ordinary updates (each $+1$), so

$$S_t = (t - D_t) \cdot 1 - D_t \cdot K = t - (K+1)D_t.$$

Since $S_t \ge 0$, we get $D_t \le \frac{t}{K+1}$.

Proof: Bound the Error for One Element

Proof

The counters record occurrences that have not been canceled. Thus $\mathrm{est}_t(e) \le \mathrm{Count}_t(e)$.

At a decrement-all step, the K stored candidates are distinct, and the arriving element is different from all of them. We cancel at most one occurrence of any particular e.

Therefore

$$\begin{aligned} 0 \le \mathrm{Count}_t(e) - \mathrm{est}_t(e) &\le D_t \\ &\le \frac{t}{K+1} \\ &\le \varepsilon t. \end{aligned}$$

The last inequality uses $K + 1 = \lceil 1/\varepsilon \rceil \ge 1/\varepsilon$.

Every Heavy Hitter Is in the List

Theorem: No false negatives

At every time t, the list contains all the ε-heavy hitters and has at most $K = \lceil 1/\varepsilon \rceil - 1$ elements.

Proof

If $\mathrm{Count}_t(e) > \varepsilon t$, then the error bound gives

$$\mathrm{est}_t(e) \geq \mathrm{Count}_t(e) - \varepsilon t > 0.$$

So e occupies a slot and is output.

With K elements and counters, the space is

$$O\left(\frac{1}{\varepsilon}(\log_2 N + \log_2 t)\right) \text{ bits.}$$

False positives are allowed; the algorithm is deterministic.





New Computational Models ▸ Streaming Algorithms

Lecture 16

- Background & Motivation
- Heavy Hitters
- Distinct Elements

Problem: Counting Distinct Elements

Given the stream $a_1, a_2, \ldots, a_t$, maintain the number

$$k = \left|\{a_1, a_2, \ldots, a_t\}\right|$$

of distinct elements seen so far.

Exact counting: In the worst case, the required memory grows linearly with the stream length, for a sufficiently large universe.

Our goal: A constant-factor estimate $\widehat{k}$:

$$\frac{k}{C} \leq \hat{k} \leq Ck$$

for some fixed constant $C > 1$.

Approximation Alone Is Not Enough

A limitation: Even for a constant-factor approximation, deterministic streaming algorithms can require $\Omega(t)$ bits of memory in the worst case, when the universe is sufficiently large.

The deterministic approach cannot always give us the space savings we want.

There is a simple **randomized** algorithm. The idea is to hash the elements and remember a minimum.

Algorithm: Remember the Minimum Hash

Idealized hashing: Choose a fixed hash function h such that distinct elements have independent uniform hash values in $[0, 1]$. Repeated occurrences of an element have the same hash value.

Algorithm: Minimum hash

Initialize $Z \leftarrow 1$.

When a_t arrives, compute $h(a_t)$ and update

$$Z \leftarrow \min\{Z, h(a_t)\}.$$

Output: the estimate $\widehat{k} = 1/Z$.

After t arrivals,

$$Z = \min_{1 \le i \le t} h(a_i).$$

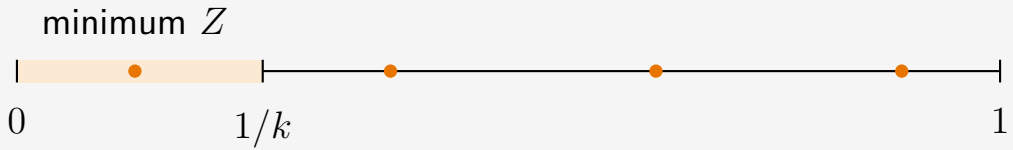
Duplicates do not change the minimum.

The idealized hashing assumption can be removed using standard tricks, such as pairwise-independent hash functions with finite precision.

Why Should a Minimum Reveal the Count?

If there are k distinct elements, Z is the minimum of k independent uniform values in $[0, 1]$.

Picture each distinct element as a dart thrown at $[0, 1]$. Repeats hit the same spot, so there are exactly k darts.



More darts push the leftmost one closer to 0:

- $k = 1$: Z is uniform on $[0, 1]$, typically around $1/2$.
- $k = 1000$: $\Pr[Z > 0.01] = 0.99^{1000} \leq e^{-10} < 0.0001$.

A small minimum is evidence that many distinct elements have arrived.

How Small Is the Minimum?

Lemma: Expected minimum

The minimum Z of k independent uniform values in $[0, 1]$ satisfies $\mathbb{E}[Z] = \frac{1}{k+1}$.

Why: Z is the length of the interval $[0, Z]$, and *lengths in* $[0, 1]$ *are probabilities*.



- Throw one more dart U. It lands in $[0, z]$ with probability exactly z. So $\mathbb{E}[Z] = \Pr[U < Z]$: the chance that a new dart lands to the left of all k darts.
- That means U is the leftmost of $k + 1$ independent darts. By symmetry, each is equally likely to be the leftmost: $\Pr[U < Z] = \frac{1}{k+1}$.

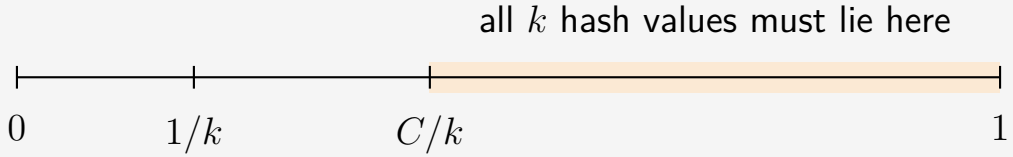
Since $Z \approx 1/k$, the output $\widehat{k} = 1/Z$ should be close to k. An average is not enough: we need to show that Z is neither much larger nor much smaller than $1/k$.

The Minimum Is Unlikely to Be Too Large

Lemma: Upper tail of the minimum

For any constant $C > 0$,

$$\Pr\left[Z > \frac{C}{k}\right] \leq e^{-C}.$$



If the minimum exceeds C/k, every distinct element hashes above C/k.

Proof: Bound the Probability All Hashes Are Large

Lemma: Upper tail of the minimum

For any constant $C > 0$, $\Pr\left[Z > \frac{C}{k}\right] \leq e^{-C}$.

Proof

If $C/k \geq 1$, the event is impossible. Otherwise, a single uniform hash value exceeds C/k with probability $1 - C/k$.

The k distinct elements have independent hash values, so

$$\Pr\left[Z > \frac{C}{k}\right] = \left(1 - \frac{C}{k}\right)^k \leq \left(e^{-C/k}\right)^k = e^{-C}.$$

We used $1 - x \leq e^{-x}$. For example, any $C \geq \ln 100$ gives $e^{-C} \leq 1/100$.

The Minimum Is Unlikely to Be Too Small

Lemma: Lower tail of the minimum

For $0 < \delta < 1$,

$$\Pr\left[Z < \frac{\delta}{k}\right] \leq \delta.$$



Proof

If $Z < \delta/k$, at least one of the k distinct elements hashes below δ/k. Each does so with probability δ/k, so by the union bound,

$$\Pr\left[Z < \frac{\delta}{k}\right] \leq k \cdot \frac{\delta}{k} = \delta.$$

A Constant-Factor Estimate

Choose a constant $C \geq 100$ and set $\delta = 1/C$. The two bounds give

$$\Pr\left[Z > \frac{C}{k}\right] \leq e^{-C}, \qquad \Pr\left[Z < \frac{1}{Ck}\right] \leq \frac{1}{C}.$$

Except with probability at most $e^{-C} + 1/C < 0.02$,

$$\frac{1}{Ck} \leq Z \leq \frac{C}{k}.$$

Taking reciprocals, our estimate $\widehat{k} = 1/Z$ satisfies

$$\frac{k}{C} \leq \widehat{k} \leq Ck$$

with probability greater than 0.98, for any fixed nonempty stream prefix.

The $1/C$ lower tail is what forces C to be large. The extensions at the end of the lecture fix this.

Extension: Keep the s Smallest Hashes

Theorem: $(1 \pm \varepsilon)$-approximation

Store the s smallest distinct hash values, and let $Z_{(s)}$ be the largest of them. For some $s = O(1/\varepsilon^2)$, the estimate $\hat{k} = s/Z_{(s)}$ satisfies

$$(1-\varepsilon)k \leq \hat{k} \leq (1+\varepsilon)k$$

with probability at least 0.99.

Intuition:

- About s of the k hash values fall below s/k, so $Z_{(s)} \approx s/k$.
- To make $\hat{k}$ too large, s values (not just one) must land too close to 0. The number below a threshold is binomial with mean $\approx s$, so it concentrates.

Want to read more? This algorithm and its analysis appear in the paper by Bar-Yossef, Jayram, Kumar, Sivakumar, and Trevisan, *Counting Distinct Elements in a Data Stream* (2002), which analyzes it without the idealized hashing assumption.

Extension: Median of Independent Copies

Theorem: Exponentially small failure probability

Run m independent copies of the minimum-hash algorithm and output the median of their estimates. Then

$$\frac{k}{3} \leq \hat{k} \leq 3k$$

except with probability at most $e^{-m/70}$.

Intuition:

- With $C = 3$, each copy fails with probability at most $\frac{1}{3} + e^{-3} < 0.39 < \frac{1}{2}$.
- The median fails only if at least half of the copies fail. By a Chernoff bound (Lecture 6), this is exponentially unlikely in m.

Want to read more? Repeating and taking the median appears in the paper by Alon, Matias, and Szegedy, *The Space Complexity of Approximating the Frequency Moments* (1996). It is one of the papers that started the study of streaming algorithms.