COMP 480/580 — Probabilistic Algorithms and Data Structure Sep 30, 2025

Lecture 10

Lecturer: Anshumali Shrivastava *Scribe By: Sahil Joshi (sj157), Gavin Ni (gjn1)*

1 Motivation

In many modern applications, data arrives in a stream, generated in real time (eg., Google Streams, Twitter Feed, Internet Traffic). However, this presents a few key challenges:

- Algorithms must be extremely fast. If an algorithm is slower than the rate at which the data is generated, it will never be able to keep up with the backlog.
- Only one pass over the data must be allowed, since data is continuous, and we donot know the size of data stream in advance.
- Memory of the buffer (or storage) must be small relative to stream size.

One approach is to maintain a uniform sample of the stream (reservoir sampling) to approximate statistics. However, for identifying frequent elements (heavy hitters), uniform sampling is generally suboptimal. Unless the sample is large enough to reliably capture frequent items, estimates can have high variance relative to their frequencies.

2 Reservoir Sampling

Definition 1 (Reservoir sampling) *We want a uniform random sample of size k from a stream $x_1, x_2, \ldots$ of* unknown *length n. Reservoir sampling keeps a small "reservoir" R of k items and updates it so that, after seeing i items, each of the i items is in R with probability k/i.*

Reservoir sampling can answer questions about the distribution of keywords, but it is inefficient for heavy hitter detection, since you would need a very large reservoir to reliably capture frequent items, making the memory requirements scale poorly with stream size.

Algorithm (one pass, $O(k)$ space, $O(1)$ update).

1. **Initialize:** Put the first k items into R.
2. **For each new item x_i with $i > k$:**
   (a) Keep x_i with probability k/i.
   (b) If kept, pick a uniform index $j \in \{1, \ldots, k\}$ and set $R[j] \leftarrow x_i$.

Tiny example ($k = 2$). Stream A, B, C, D, E arrives.

1. After A, B: $R = \{A, B\}$.
2. $i = 3$ (item C): keep with prob $2/3$. Suppose we keep and randomly replace $B \Rightarrow R = \{A, C\}$.
3. $i = 4$ (D): keep with prob $2/4 = 1/2$. Suppose we *do not* keep $\Rightarrow R = \{A, C\}$.
4. $i = 5$ (E): keep with prob $2/5$. Suppose we keep and replace A (chosen index $j = 1$) $\Rightarrow R = \{E, C\}$.

Different random draws give different samples, but *in expectation* each of A, B, C, D, E ends up in R with probability $2/5$.

When to (not) use it: Great for estimating means, proportions, distinct examples, etc. For *heavy hitters*, a small uniform sample can miss frequent-but-not-dominant items; use frequency-aware sketches (e.g., Count–Min) when top-k matters.

Key properties: One pass; expected $O(1)$ work per item; memory $O(k)$; uniform sample without knowing n in advance.

3 Majority Element Problem

> Given an array $A[1..n]$, find an element that occurs in more than $n/2$ positions using $O(1)$ space and $O(n)$ time.

Boyer–Moore Majority Vote 2: Maintain a *candidate* c and a *counter* t.

1. Initialize $t \leftarrow 0$.
2. For each $x \in A$:
   (a) If $t = 0$, set $c \leftarrow x$ and $t \leftarrow 1$.
   (b) Else if $x = c$, set $t \leftarrow t + 1$; otherwise set $t \leftarrow t - 1$.

At the end, c is a *majority candidate*. If a majority element exists, then c equals it.

Correctness: Let m be the majority element that occurs more than $n/2$ times. The key insight is that each non-majority element can "cancel" at most one occurrence of m, but since m occurs more than half the time, there will always be at least one uncanceled occurrence of m at the end. More formally, consider the algorithm as pairing up different elements. When we see different elements, they cancel each other (count decreases). Since m occurs more than all other elements combined, even in the worst-case pairing where each m is paired with a different element, there will be at least one m left uncanceled at the end. The algorithm maintains the invariant that if there is a majority element, it will be the candidate when the algorithm terminates. This is because:

1. When the counter is positive, the candidate has occurred more times than all other elements in the prefix processed so far.
2. The majority element cannot be completely "canceled out" since it appears more than half the time.

Complexity. We need only one pass over the input array, so time is $O(n)$; only c, t, and the final count are stored, so space is $O(1)$.

4 Hashed Counting & Count Min Sketch

By nature, many real world distributions are skewed: a few elements occur very frequently, while the majority are rare. This works to our advantage in analyzing data streams, and is the base assumption in sketching algorithms, which approximate counts efficiently. We can hash each stream element into an array of counters.

- On arrival of element i, increment the counter at $h(i)$.
- Estimated frequency $\hat{C}_i = C_{h(i)}$.

If the hash is perfect, the estimate is exact. In practice, however, collisions can cause error.

Error Analysis (Assuming 1 hash function)

Let C denote the observed counter at bucket $h(i)$ and let C_i be the true count of item i.

$$C \;=\; C_i \;+\; \sum_{j \neq i} \mathbf{1}\{h(j) = h(i)\}\, C_j$$

By linearity of expectation and $\Pr[h(j) = h(i)] = \frac{1}{R}$, where R is the range of the hash function

$$\mathbb{E}[C] \;=\; C_i \;+\; \sum_{j \neq i} \frac{1}{R}\, C_j \;=\; C_i \;+\; \frac{1}{R} \sum_{j \neq i} C_j.$$

Since collisions only add mass, the error is nonnegative:

$$\mathbb{E}[|C - C_i|] \;=\; \frac{1}{R} \sum_{j \neq i} C_j \;\leq\; \frac{1}{R} \sum_{j} C_j.$$

With R counters (hash range), probability of collision is $\frac{1}{R}$, so:

$$\mathbb{E}[|C - C_i|] \leq \frac{1}{R} \sum_{j} C_j$$

Thus, error decreases with larger R.

Error Reduction (With more than 1 hash function):

To reduce collision error:

- Use k independent hash functions $h_1, \ldots, h_k$.
- Maintain k arrays of counters.
- On arrival of i, increment all k counters.
- Estimate frequency by:

$$\hat{C}_i = \min_{j=1}^{k} C_{h_j(i)}$$

We take the minimum because each count is always overestimated due to collisions, so taking the minimum gives a good bound.

Let $\Sigma = \sum_{i=1}^{n} C_i$ (total stream length) and $\epsilon = 1/R$. Let us define per-item error $= |\hat{C}_i - C_i|$,

$$\mathbb{E}[\text{error}] \leq \epsilon\Sigma$$

Using Markov's inequality:

$$\Pr(\text{error} \geq 2\epsilon\Sigma) \leq \tfrac{1}{2}$$

For k hash functions, probability of error exceeding $2\epsilon\Sigma$ is at most $(\frac{1}{2})^k$. Thus, error can be reduced arbitrarily by increasing k.

Union bound for error over all items

Assume the per-item guarantee

$$\Pr\big(|\hat{C}_i - C_i| \geq 2\epsilon\,\Sigma\big) \;\leq\; 2^{-k} \quad \text{for each } i \in \{1, \ldots, n\}.$$

Let $A_i := \{|\hat{C}_i - C_i| \leq 2\epsilon\,\Sigma\}$. Then

$$\begin{aligned}
\Pr\Big(\forall i :\, |\hat{C}_i - C_i| \leq 2\epsilon\,\Sigma\Big) &= \Pr\Big(\bigcap_{i=1}^{n} A_i\Big) \\
&= 1 - \Pr\Big(\bigcup_{i=1}^{n} A_i^{\mathrm{c}}\Big) \\
&\geq 1 - \sum_{i=1}^{n} \Pr(A_i^{\mathrm{c}}) \qquad \text{(union bound)} \\
&= 1 - \sum_{i=1}^{n} \Pr\big(|\hat{C}_i - C_i| \geq 2\epsilon\,\Sigma\big) \\
&\geq 1 - n \cdot 2^{-k}
\end{aligned}$$

Therefore, choosing

$$k \;\geq\; \left\lceil \log_2 \frac{n}{\delta} \right\rceil$$

ensures

$$\Pr\Big(\forall i :\, |\hat{C}_i - C_i| \leq 2\epsilon\,\Sigma\Big) \;\geq\; 1 - \delta$$

5 Finding Heavy Hitters: Count–Min Sketch + Heap (Streaming Example)

Setup

Goal: maintain the top-2 heavy hitters from the stream

`A, B, A, C, A, B, D, A, C, B, E, A, B, F.`

Data structures:

- **Count–Min Sketch (CMS):** $k = 3$ (independent hashes h_1, h_2, h_3), range $R = 4$ counters per row.
- **Min-heap H of size $S = 2$:** stores pairs $(\hat{C}_x, \text{item } x)$, keyed by $\hat{C}_x$ (tie-break by item ID).

CMS update: on item x, increment the three counters at indices $h_1(x), h_2(x), h_3(x)$. CMS query: $\hat{C}_x = \min\{C^{(1)}_{h_1(x)}, C^{(2)}_{h_2(x)}, C^{(3)}_{h_3(x)}\}$ (an upper bound on the true count C_x).

Streaming Trace

We illustrate the *online* evolution of H; the exact numeric values of $\hat{C}_x$ depend on collisions, but the logic is deterministic.

1. **A** arrives: update CMS; $\hat{C}_A = 1$. Heap has room $\Rightarrow H = \{(1, A)\}$.
2. **B** arrives: update CMS; $\hat{C}_B = 1$. Heap has room $\Rightarrow H = \{(1, A), (1, B)\}$.
3. **A** arrives: update CMS; typically $\hat{C}_A \geq 2$ (exactly 2 if no collisions on A's buckets so far). Update A's key $\Rightarrow H = \{(1, B), (\hat{C}_A, A)\}$.
4. **C** arrives: update CMS; $\hat{C}_C \geq 1$. Heap full; compare with current minimum (the entry for B). If $\hat{C}_C$ is *strictly* larger than the heap minimum, replace it; otherwise discard C.
5. **A** arrives: update CMS; $\hat{C}_A$ increases monotonically (no less than previous). Update A's key in H.
6. **B** arrives: update CMS; $\hat{C}_B$ increases; update B's key in H.
7. **D** arrives: get $\hat{C}_D \geq 1$; compare to heap minimum and discard unless it strictly exceeds the minimum.
8. Continue similarly for the rest of the stream. Each time an item x arrives:
   - Update its three CMS counters.
   - Compute $\hat{C}_x$.
   - If $x \in H$, update its key; else if $|H| < 2$, insert; else if $\hat{C}_x$ exceeds the current heap minimum (or ties and wins by the tie-break), replace the minimum.

At the end, the true frequencies are

$$C_A = 5, \quad C_B = 4, \quad C_C = 2, \quad C_D = 1, \quad C_E = 1, \quad C_F = 1.$$

CMS guarantees $\hat{C}_x \geq C_x$ for all x (collisions only add). The heap therefore returns the two items with the largest *estimates*, which in this stream are A and B (their estimates may be slightly above 5 and 4 due to collisions).

Procedure (CMS + Heap, per update)

1. For incoming x: increment $C^{(r)}_{h_r(x)}$ for each $r \in \{1, 2, 3\}$.
2. Compute $\hat{C}_x = \min\{C^{(1)}_{h_1(x)}, C^{(2)}_{h_2(x)}, C^{(3)}_{h_3(x)}\}$.
3. If $x \in H$, update its key to $\hat{C}_x$ and restore heap order.
4. Else if $|H| < S$, insert $(\hat{C}_x, x)$.
5. Else if $\hat{C}_x$ is greater than the current heap minimum (or ties and wins by a fixed tie-break), pop the minimum and insert $(\hat{C}_x, x)$.

Key Points

- The heap keeps only the top-S *estimates*; CMS provides estimates for all items without storing per-item counters.
- Space: $O(kR + S)$; per-update time: $O(k + \log S)$.
- Estimates are *upper bounds*; larger k (independent hashes) make large overestimates rarer.
- Specify a deterministic tie-break in the heap (e.g., by item ID) to make replacements unambiguous.

6 Summary

- Reservoir sampling works for unbiased random samples, but is inefficient for the "heavy hitters" problem.
- The Boyer Moore algorithm finds majority elements in $O(1)$ space.
- Count Min Sketch provides approximate frequency estimation. Very useful for the "heavy hitters" problem.
- Tradeoffs:
  - Larger hash range (R) $\Rightarrow$ smaller error.
  - More hash functions (k) $\Rightarrow$ higher confidence.

References

[1] Wikipedia: Boole's inequality.

[2] Boyer–Moore majority vote algorithm