COMPSCI 514: ALGORITHMS FOR DATA SCIENCE

Cameron Musco
University of Massachusetts Amherst. Fall 2019.
Lecture 8
· Problem Set 1 was due this morning in Gradescope.
· Problem Set 2 will be released tomorrow and due 10/10.
Last Class: Finished up MinHash and LSH.

- Application to fast similarity search.
- False positive and negative tuning with length $r$ hash signatures and $t$ hash table repetitions ($s$-curves).
- Examples of other locality sensitive hash functions (SimHash).

This Class:

- The Frequent Elements (heavy-hitters) problem in data streams.
- Misra-Gries summaries.
- Count-min sketch.

• Building on the idea of SimHash.

After That: Spectral Methods

• PCA, low-rank approximation, and the singular value decomposition.
• Spectral clustering and spectral graph theory.

Will use a lot of linear algebra. May be helpful to refresh.

• Vector dot product, addition, length. Matrix vector multiplication.
• Linear independence, column span, orthogonal bases, rank.
• Eigendecomposition.
<table>
<thead>
<tr>
<th></th>
<th>Hash Table</th>
<th>Bloom Filters</th>
<th>MinHash Similarity Search</th>
<th>Distinct Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Goal</strong></td>
<td>Check if x is a duplicate of any y in database and return y.</td>
<td>Check if x is a duplicate of y in database.</td>
<td>Check if x is a duplicate of any y in database and return y.</td>
<td>Count # of items, excluding duplicates.</td>
</tr>
<tr>
<td><strong>Space</strong></td>
<td>$O(n)$ items</td>
<td>$O(n)$ bits</td>
<td>$O(n \cdot t)$ items (when t tables used)</td>
<td>$O\left(\frac{\log \log n}{\epsilon^2}\right)$</td>
</tr>
<tr>
<td><strong>Query Time</strong></td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>Potentially $o(n)$</td>
<td>NA</td>
</tr>
<tr>
<td><strong>Approximate Duplicates?</strong></td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
</tr>
</tbody>
</table>

All different variants of detecting duplicates/finding matches in large datasets. An important problem in many contexts!
**The Frequent Items Problem**

*k*-Frequent Items (Heavy-Hitters) Problem: Consider a stream of *n* items $x_1, \ldots, x_n$ (with possible duplicates). Return any item that appears at least $\frac{n}{k}$ times. E.g., for $n = 9$, $k = 3$:

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$x_4$</th>
<th>$x_5$</th>
<th>$x_6$</th>
<th>$x_7$</th>
<th>$x_8$</th>
<th>$x_9$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>12</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>3</td>
</tr>
</tbody>
</table>

- What is the maximum number of items that must be returned? At most $k$ items with frequency $\geq \frac{n}{k}$.
- Think of $k = 100$. Want items appearing $\geq 1\%$ of the time.
- Easy with $O(n)$ space – store the count for each item and return the one that appears $\geq n/k$ times.
- Can we do it with less space? I.e., without storing all *n* items?
- Similar challenge as with the distinct elements problem.
Applications of Frequent Items:

- Finding top/viral items (i.e., products on Amazon, videos watched on Youtube, Google searches, etc.)
- Finding very frequent IP addresses sending requests (to detect DoS attacks/network anomalies).
- ‘Iceberg queries’ for all items in a database with frequency above some threshold.

Generally want very fast detection, without having to scan through database/logs. I.e., want to maintain a running list of frequent items that appear in a stream.
Association rule learning: A very common task in data mining is to identify common associations between different events.

- Identified via frequent itemset counting. Find all sets of $k$ items that appear many times in the same basket.
- Frequency of an itemset is known as its support.
- A single basket includes many different itemsets, and with many different baskets an efficient approach is critical. E.g., baskets are Twitter users and itemsets are subsets of who they follow.
**Majority:** Consider a stream of $n$ items $x_1, \ldots, x_n$, where a single item appears a majority of the time. Return this item.

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$x_4$</th>
<th>$x_5$</th>
<th>$x_6$</th>
<th>$x_7$</th>
<th>$x_8$</th>
<th>$x_9$</th>
<th>$x_{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>12</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

- Basically $k$-Frequent items for $k = 2$ (and assume a single item has a strict majority.)
**Boyer-Moore Voting Algorithm:** (our first *deterministic algorithm*)

- Initialize count $c := 0$, majority element $m := \bot$
- For $i = 1, \ldots, n$
  - If $c = 0$, set $m := x_i$ and $c := 1$.
  - Else if $m = x_i$, set $c := c + 1$.
  - Else if $m \neq x_i$, set $c := c - 1$.

Just requires $O(\log n)$ bits to store $c$ and space to store $m$.

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$x_4$</th>
<th>$x_5$</th>
<th>$x_6$</th>
<th>$x_7$</th>
<th>$x_8$</th>
<th>$x_9$</th>
<th>$x_{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>12</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>
Boyer-Moore Voting Algorithm: (our first *deterministic algorithm*)

- Initialize count $c := 0$, majority element $m := \perp$
- For $i = 1, \ldots, n$
  - If $c = 0$, set $m := x_i$ and $c := 1$.
  - Else if $m = x_i$, set $c := c + 1$.
  - Else if $m \neq x_i$, set $c := c - 1$.

Just requires $O(\log n)$ bits to store $c$ and space to store $m$. 

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$x_4$</th>
<th>$x_5$</th>
<th>$x_6$</th>
<th>$x_7$</th>
<th>$x_8$</th>
<th>$x_9$</th>
<th>$x_{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>12</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>
Boyermoore Voting Algorithm: (our first deterministic algorithm)

- Initialize count \(c := 0\), majority element \(m := \bot\)
- For \(i = 1, \ldots, n\)
  - If \(c = 0\), set \(m := x_i\) and \(c := 1\).
  - Else if \(m = x_i\), set \(c := c + 1\).
  - Else if \(m \neq x_i\), set \(c := c - 1\).

Just requires \(O(\log n)\) bits to store \(c\) and space to store \(m\).
Boyermoore Voting Algorithm: (our first deterministic algorithm)

- Initialize count \( c := 0 \), majority element \( m := \perp \)
- For \( i = 1, \ldots, n \)
  - If \( c = 0 \), set \( m := x_i \) and \( c := 1 \).
  - Else if \( m = x_i \), set \( c := c + 1 \).
  - Else if \( m \neq x_i \), set \( c := c - 1 \).

Just requires \( O(\log n) \) bits to store \( c \) and space to store \( m \).
Boyer-Moore Voting Algorithm: (our first deterministic algorithm)

- Initialize count $c := 0$, majority element $m := \bot$
- For $i = 1, \ldots, n$
  - If $c = 0$, set $m := x_i$ and $c := 1$.
  - Else if $m = x_i$, set $c := c + 1$.
  - Else if $m \neq x_i$, set $c := c - 1$.

Just requires $O(\log n)$ bits to store $c$ and space to store $m$. 

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$x_4$</th>
<th>$x_5$</th>
<th>$x_6$</th>
<th>$x_7$</th>
<th>$x_8$</th>
<th>$x_9$</th>
<th>$x_{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>12</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>
Boyer-Moore Voting Algorithm: (our first deterministic algorithm)

- Initialize count \( c := 0 \), majority element \( m := \perp \)
- For \( i = 1, \ldots, n \)
  - If \( c = 0 \), set \( m := x_i \) and \( c := 1 \).
  - Else if \( m = x_i \), set \( c := c + 1 \).
  - Else if \( m \neq x_i \), set \( c := c - 1 \).

Just requires \( O(\log n) \) bits to store \( c \) and space to store \( m \).
Boyer-Moore Voting Algorithm: (our first *deterministic algorithm*)

- Initialize count $c := 0$, majority element $m := \perp$
- For $i = 1, \ldots, n$
  - If $c = 0$, set $m := x_i$ and $c := 1$.
  - Else if $m = x_i$, set $c := c + 1$.
  - Else if $m \neq x_i$, set $c := c - 1$.

Just requires $O(\log n)$ bits to store $c$ and space to store $m$. 

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$x_4$</th>
<th>$x_5$</th>
<th>$x_6$</th>
<th>$x_7$</th>
<th>$x_8$</th>
<th>$x_9$</th>
<th>$x_{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>12</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>
Boyer-Moore Voting Algorithm: (our first deterministic algorithm)

- Initialize count $c := 0$, majority element $m := \bot$
- For $i = 1, \ldots, n$
  - If $c = 0$, set $m := x_i$ and $c := 1$.
  - Else if $m = x_i$, set $c := c + 1$.
  - Else if $m \neq x_i$, set $c := c - 1$.

Just requires $O(\log n)$ bits to store $c$ and space to store $m$. 
Boyer-Moore Voting Algorithm: (our first deterministic algorithm)

- Initialize count \( c := 0 \), majority element \( m := \perp \)
- For \( i = 1, \ldots, n \)
  - If \( c = 0 \), set \( m := x_i \) and \( c := 1 \).
  - Else if \( m = x_i \), set \( c := c + 1 \).
  - Else if \( m \neq x_i \), set \( c := c - 1 \).

Just requires \( O(\log n) \) bits to store \( c \) and space to store \( m \).
Boyer-Moore Voting Algorithm: (our first deterministic algorithm)

- Initialize count $c := 0$, majority element $m := \bot$
- For $i = 1, \ldots, n$
  - If $c = 0$, set $m := x_i$ and $c := 1$.
  - Else if $m = x_i$, set $c := c + 1$.
  - Else if $m \neq x_i$, set $c := c - 1$.

Just requires $O(\log n)$ bits to store $c$ and space to store $m$. 

\[ \begin{array}{cccccccccc}
  x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 & x_9 & x_{10} \\
  5   & 12  & 3   & 5   & 4   & 5   & 5   & 10  & 5   & 5 \\
\end{array} \]
Boyer-Moore Voting Algorithm: (our first deterministic algorithm)

- Initialize count \( c := 0 \), majority element \( m := \perp \)
- For \( i = 1, \ldots, n \)
  - If \( c = 0 \), set \( m := x_i \) and \( c := 1 \).
  - Else if \( m = x_i \), set \( c := c + 1 \).
  - Else if \( m \neq x_i \), set \( c := c - 1 \).

Just requires \( O(\log n) \) bits to store \( c \) and space to store \( m \).
Boyer-Moore Voting Algorithm:
• Initialize count $c := 0$, majority element $m := \bot$
• For $i = 1, \ldots, n$
  • If $c = 0$, set $m := x_i$ and $c := 1$.
  • Else if $m = x_i$, set $c := c + 1$.
  • Else if $m \neq x_i$, set $c := c - 1$.

Claim: The Boyer-Moore algorithm always outputs the majority element, regardless of what order the stream is presented in.

Proof: Let $M$ be the true majority element. Let $s = c$ when $m = M$ and $s = -c$ otherwise (s is a ‘helper’ variable).

Claim: The Boyer-Moore algorithm always outputs the majority element, regardless of what order the stream is presented in.
Boyer-Moore Voting Algorithm:
- Initialize count $c := 0$, majority element $m := \bot$
- For $i = 1, \ldots, n$
  - If $c = 0$, set $m := x_i$ and $c := 1$.
  - Else if $m = x_i$, set $c := c + 1$.
  - Else if $m \neq x_i$, set $c := c - 1$.

Claim: The Boyer-Moore algorithm always outputs the majority element, regardless of what order the stream is presented in.

Proof: Let $M$ be the true majority element. Let $s = c$ when $m = M$ and $s = -c$ otherwise (s is a ‘helper’ variable).
**Correctness of Boyer-Moore**

**Boyer-Moore Voting Algorithm:**
- Initialize count $c := 0$, majority element $m := \bot$
- For $i = 1, \ldots, n$
  - If $c = 0$, set $m := x_i$ and $c := 1$.
  - Else if $m = x_i$, set $c := c + 1$.
  - Else if $m \neq x_i$, set $c := c - 1$.

**Claim:** The Boyer-Moore algorithm always outputs the majority element, regardless of what order the stream is presented in.

**Proof:** Let $M$ be the true majority element. Let $s = c$ when $m = M$ and $s = -c$ otherwise (s is a ‘helper’ variable).

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$x_4$</th>
<th>$x_5$</th>
<th>$x_6$</th>
<th>$x_7$</th>
<th>$x_8$</th>
<th>$x_9$</th>
<th>$x_{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>12</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>
Boyer-Moore Voting Algorithm:

- Initialize count $c := 0$, majority element $m := \bot$
- For $i = 1, \ldots, n$
  - If $c = 0$, set $m := x_i$ and $c := 1$.
  - Else if $m = x_i$, set $c := c + 1$.
  - Else if $m \neq x_i$, set $c := c - 1$.

Claim: The Boyer-Moore algorithm always outputs the majority element, regardless of what order the stream is presented in.

Proof: Let $M$ be the true majority element. Let $s = c$ when $m = M$ and $s = -c$ otherwise (s is a ‘helper’ variable).

- $s$ is incremented each time $M$ appears. So it is incremented more than it is decremented (since $M$ appears a majority of times) and ends at a positive value. $\implies$ algorithm ends with $m = M$. 
**k-Frequent Items (Heavy-Hitters) Problem:** Consider a stream of \( n \) items \( x_1, \ldots, x_n \) (with possible duplicates). Return any item that appears at least \( \frac{n}{k} \) times.

**Boyer-Moore Voting Algorithm:**

- Initialize count \( c := 0 \), majority element \( m := \bot \)
- For \( i = 1, \ldots, n \)
  - If \( c = 0 \), set \( m := x_i \)
  - Else if \( m = x_i \), set \( c := c + 1 \).
  - Else if \( m \neq x_i \), set \( c := c - 1 \).
**$k$-Frequent Items (Heavy-Hitters) Problem:** Consider a stream of $n$ items $x_1, \ldots, x_n$ (with possible duplicates). Return any item at appears at least $\frac{n}{k}$ times.

**Misra-Gries Summary:**

- Initialize counts $c_1, \ldots, c_k := 0$, elements $m_1, \ldots, m_k := \perp$.
- For $i = 1, \ldots, n$
  - If $m_j = x_i$ for some $j$, set $c_j := c_j + 1$.
  - Else let $t = \arg \min c_j$. If $c_t = 0$, set $m_t := x_i$ and $c_t := 1$.
  - Else $c_j := c_j - 1$ for all $j$. 
Misra-Gries Summary:

- Initialize counts $c_1, \ldots, c_k := 0$, elements $m_1, \ldots, m_k := \perp$.
- For $i = 1, \ldots, n$
  - If $m_j = x_i$ for some $j$, set $c_j := c_j + 1$.
  - Else let $t = \arg \min c_j$. If $c_t = 0$, set $m_t := x_i$ and $c_t := 1$.
  - Else $c_j := c_j - 1$ for all $j$.

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$x_4$</th>
<th>$x_5$</th>
<th>$x_6$</th>
<th>$x_7$</th>
<th>$x_8$</th>
<th>$x_9$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>12</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>3</td>
</tr>
</tbody>
</table>
Misra-Gries Summary:

- Initialize counts $c_1, \ldots, c_k := 0$, elements $m_1, \ldots, m_k := \bot$.
- For $i = 1, \ldots, n$
  - If $m_j = x_i$ for some $j$, set $c_j := c_j + 1$.
  - Else let $t = \arg\min c_j$. If $c_t = 0$, set $m_t := x_i$ and $c_t := 1$.
  - Else $c_j := c_j - 1$ for all $j$.

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$x_4$</th>
<th>$x_5$</th>
<th>$x_6$</th>
<th>$x_7$</th>
<th>$x_8$</th>
<th>$x_9$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>12</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>3</td>
</tr>
</tbody>
</table>
Misra-Gries Summary:

- Initialize counts $c_1, \ldots, c_k := 0$, elements $m_1, \ldots, m_k := \perp$.
- For $i = 1, \ldots, n$
  - If $m_j = x_i$ for some $j$, set $c_j := c_j + 1$.
  - Else let $t = \arg \min c_j$. If $c_t = 0$, set $m_t := x_i$ and $c_t := 1$.
  - Else $c_j := c_j - 1$ for all $j$.

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$x_4$</th>
<th>$x_5$</th>
<th>$x_6$</th>
<th>$x_7$</th>
<th>$x_8$</th>
<th>$x_9$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>12</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>3</td>
</tr>
</tbody>
</table>
Misra-Gries Summary:

- Initialize counts $c_1, \ldots, c_k := 0$, elements $m_1, \ldots, m_k := \perp$.
- For $i = 1, \ldots, n$
  - If $m_j = x_i$ for some $j$, set $c_j := c_j + 1$.
  - Else let $t = \text{arg min } c_j$. If $c_t = 0$, set $m_t := x_i$ and $c_t := 1$.
  - Else $c_j := c_j - 1$ for all $j$.

| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | $x_8$ | $x_9$
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>12</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>3</td>
</tr>
</tbody>
</table>
Misra-Gries Summary:

- Initialize counts $c_1, \ldots, c_k := 0$, elements $m_1, \ldots, m_k := \perp$.
- For $i = 1, \ldots, n$
  - If $m_j = x_i$ for some $j$, set $c_j := c_j + 1$.
  - Else let $t = \arg \min c_j$. If $c_t = 0$, set $m_t := x_i$ and $c_t := 1$.
  - Else $c_j := c_j - 1$ for all $j$.

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>$x_2$</td>
<td>$x_3$</td>
<td>$x_4$</td>
<td>$x_5$</td>
<td>$x_6$</td>
<td>$x_7$</td>
<td>$x_8$</td>
<td>$x_9$</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
Misra-Gries Summary:

- Initialize counts $c_1, \ldots, c_k := 0$, elements $m_1, \ldots, m_k := \bot$.
- For $i = 1, \ldots, n$
  - If $m_j = x_i$ for some $j$, set $c_j := c_j + 1$.
  - Else let $t = \arg \min c_j$. If $c_t = 0$, set $m_t := x_i$ and $c_t := 1$.
  - Else $c_j := c_j - 1$ for all $j$.

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$x_4$</th>
<th>$x_5$</th>
<th>$x_6$</th>
<th>$x_7$</th>
<th>$x_8$</th>
<th>$x_9$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>12</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>3</td>
</tr>
</tbody>
</table>
Misra-Gries Summary:

- Initialize counts \( c_1, \ldots, c_k := 0 \), elements \( m_1, \ldots, m_k := \perp \).
- For \( i = 1, \ldots, n \)
  - If \( m_j = x_i \) for some \( j \), set \( c_j := c_j + 1 \).
  - Else let \( t = \arg \min c_j \). If \( c_t = 0 \), set \( m_t := x_i \) and \( c_t := 1 \).
  - Else \( c_j := c_j - 1 \) for all \( j \).

<table>
<thead>
<tr>
<th>( c_1 )</th>
<th>( m_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( c_2 )</th>
<th>( m_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( c_3 )</th>
<th>( m_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( x_3 )</th>
<th>( x_4 )</th>
<th>( x_5 )</th>
<th>( x_6 )</th>
<th>( x_7 )</th>
<th>( x_8 )</th>
<th>( x_9 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>12</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>3</td>
</tr>
</tbody>
</table>
Misra-Gries Summary:

- Initialize counts $c_1, \ldots, c_k := 0$, elements $m_1, \ldots, m_k := \bot$.
- For $i = 1, \ldots, n$
  - If $m_j = x_i$ for some $j$, set $c_j := c_j + 1$.
  - Else let $t = \text{arg min} c_j$. If $c_t = 0$, set $m_t := x_i$ and $c_t := 1$.
  - Else $c_j := c_j - 1$ for all $j$.

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$x_4$</th>
<th>$x_5$</th>
<th>$x_6$</th>
<th>$x_7$</th>
<th>$x_8$</th>
<th>$x_9$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>12</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>3</td>
</tr>
</tbody>
</table>
Misra-Gries Summary:

- Initialize counts $c_1, \ldots, c_k := 0$, elements $m_1, \ldots, m_k := \perp$.
- For $i = 1, \ldots, n$
  - If $m_j = x_i$ for some $j$, set $c_j := c_j + 1$.
  - Else let $t = \arg \min c_j$. If $c_t = 0$, set $m_t := x_i$ and $c_t := 1$.
  - Else $c_j := c_j - 1$ for all $j$.

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$x_4$</th>
<th>$x_5$</th>
<th>$x_6$</th>
<th>$x_7$</th>
<th>$x_8$</th>
<th>$x_9$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>12</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>3</td>
</tr>
</tbody>
</table>
Misra-Gries Summary:

- Initialize counts $c_1, \ldots, c_k := 0$, elements $m_1, \ldots, m_k := \perp$.
- For $i = 1, \ldots, n$
  - If $m_j = x_i$ for some $j$, set $c_j := c_j + 1$.
  - Else let $t = \arg \min c_j$. If $c_t = 0$, set $m_t := x_i$ and $c_t := 1$.
  - Else $c_j := c_j - 1$ for all $j$.

<table>
<thead>
<tr>
<th></th>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$x_4$</th>
<th>$x_5$</th>
<th>$x_6$</th>
<th>$x_7$</th>
<th>$x_8$</th>
<th>$x_9$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>12</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>3</td>
</tr>
</tbody>
</table>
Misra-Gries Summary:

- Initialize counts $c_1, \ldots, c_k := 0$, elements $m_1, \ldots, m_k := \perp$.
- For $i = 1, \ldots, n$
  - If $m_j = x_i$ for some $j$, set $c_j := c_j + 1$.
  - Else let $t = \arg\min c_j$. If $c_t = 0$, set $m_t := x_i$ and $c_t := 1$.
  - Else $c_j := c_j - 1$ for all $j$.

Claim: At the end of the stream, all items with frequency $\geq \frac{n}{k}$ are stored.
Claim: At the end of the stream, the Misra-Gries algorithm stores $k$ items, including all those with frequency $\geq \frac{n}{k}$.

Intuition:

• If there are exactly $k$ items, each appearing exactly $n/k$ times, all are stored (since we have $k$ storage slots).
• If there are $k/2$ items each appearing $\geq n/k$ times, there are $\leq n/2$ irrelevant items, being inserted into $k/2$ ‘free slots’.
• May cause $\frac{n/2}{k/2} = \frac{n}{k}$ decrement operations. Few enough that the heavy items (appearing $n/k$ times each) are still stored.

Anything undesirable about the Misra-Gries output guarantee? May have false positives – infrequent items that are stored.
**Issue:** Misra-Gries algorithm stores $k$ items, including all with frequency $\geq n/k$. But may include infrequent items.

- In fact, no algorithm using $o(n)$ space can output just the items with frequency $\geq n/k$. Hard to tell between an item with frequency $n/k$ (should be output) and $n/k - 1$ (should not be output).

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$x_4$</th>
<th>$x_5$</th>
<th>$x_6$</th>
<th>...</th>
<th>$x_{n-n/k+1}$</th>
<th>...</th>
<th>$x_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>12</td>
<td>9</td>
<td>27</td>
<td>4</td>
<td>101</td>
<td>...</td>
<td>3</td>
<td>...</td>
<td>3</td>
</tr>
</tbody>
</table>

$n/k - 1$ occurrences
**Issue:** Misra-Gries algorithm stores $k$ items, including all with frequency $\geq n/k$. But may include infrequent items.

- In fact, no algorithm using $o(n)$ space can output just the items with frequency $\geq n/k$. Hard to tell between an item with frequency $n/k$ (should be output) and $n/k - 1$ (should not be output).

**$(\epsilon, k)$-Frequent Items Problem:** Consider a stream of $n$ items $x_1, \ldots, x_n$. Return a set $F$ of items, including **all items that appear at least** $\frac{n}{k}$ **times and only items that appear at least** $(1 - \epsilon) \cdot \frac{n}{k}$ **times.**

- An example of relaxing to a ‘promise problem’: for items with frequencies in $[(1 - \epsilon) \cdot \frac{n}{k}, \frac{n}{k}]$ no output guarantee.
**Misra-Gries Summary:** (ε-error version)

- Let $r := \lceil k/\varepsilon \rceil$
- Initialize counts $c_1, \ldots, c_r := 0$, elements $m_1, \ldots, m_r := \bot$.
- For $i = 1, \ldots, n$
  - If $m_j = x_i$ for some $j$, set $c_j := c_j + 1$.
  - Else let $t = \arg\min j. c_j$. If $c_t = 0$, set $m_t := x_i$ and $c_t := 1$.
  - Else $c_j := c_j - 1$ for all $j$.
- Return any $m_j$ with $c_j \geq (1 - \varepsilon) \cdot \frac{n}{k}$.

**Claim:** For all $m_j$ with true frequency $f(m_j)$:

$$f(m_j) - \frac{\varepsilon n}{k} \leq c_j \leq f(m_j).$$

**Intuition:** # items stored $r$ is large, so relatively few decrements.

**Implication:** If $f(m_j) \geq \frac{n}{k}$, then $c_j \geq (1 - \varepsilon) \cdot \frac{n}{k}$ so the item is returned. If $f(m_j) \leq (1 - \varepsilon) \cdot \frac{n}{k}$, then $c_j < (1 - \varepsilon) \cdot \frac{n}{k}$ so the item is not returned.
**Upshot:** The $(\epsilon, k)$-Frequent Items problem can be solved via the Misra-Gries approach.

- Space usage is $\lceil k/\epsilon \rceil$ counts – $O\left(\frac{k \log n}{\epsilon}\right)$ bits and $\lceil k/\epsilon \rceil$ items.
- Deterministic approximation algorithm.
A common alternative to the Misra-Gries approach is the **count-min sketch**: a randomized method closely related to bloom filters.

- A major advantage: easily distributed to processing on multiple servers.

\[
\begin{array}{ccccccccc}
\mathbf{x}_1 & \mathbf{x}_2 & \mathbf{x}_3 & \mathbf{x}_4 & \ldots & \mathbf{x}_n \\
\end{array}
\]

random hash function \( h \)

\[
\begin{array}{cccccccccccc}
\mathbf{A} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]
A common alternative to the Misra-Gries approach is the **count-min sketch**: a randomized method closely related to bloom filters.

- A major advantage: easily distributed to processing on multiple servers.

**Diagram:**

- **Random hash function** $h$
- **m length array** $A$

- $X_1$, $X_2$, $X_3$, $X_4$, ..., $X_n$
A common alternative to the Misra-Gries approach is the **count-min sketch**: a randomized method closely related to bloom filters.

- A major advantage: easily distributed to processing on multiple servers.

\[
\begin{align*}
A_1 & \quad \cdots \quad A_s \\
\end{align*}
\]

Will use \( A[h(x)] \) to estimate \( f(x) \), the frequency of \( x \) in the stream. I.e., \( |f(x) : x_i = x| \).
A common alternative to the Misra-Gries approach is the **count-min sketch**: a randomized method closely related to bloom filters.

- A major advantage: easily distributed to processing on multiple servers.

![Diagram of count-min sketch](image)
A common alternative to the Misra-Gries approach is the **count-min sketch**: a randomized method closely related to bloom filters.

- A major advantage: easily distributed to processing on multiple servers.

![Diagram showing the count-min sketch algorithm](image.png)

- Build arrays $A_1, \ldots, A_s$ separately and then just set $A := A_1 + \ldots + A_s$.
- Will use $A[h(x)]$ to estimate $f(x)$, the frequency of $x$ in the stream. I.e., $|f(x)| = x_g$.
A common alternative to the Misra-Gries approach is the **count-min sketch**: a randomized method closely related to bloom filters.

- A major advantage: easily distributed to processing on multiple servers.

```
Build arrays A_1, ..., A_s separately and then just set A := A_1 + ... + A_s.
Will use A[h(x)] to estimate f(x), the frequency of x in the stream. I.e., |f(x) : x = x|.
```
A common alternative to the Misra-Gries approach is the **count-min sketch**: a randomized method closely related to bloom filters.

- A major advantage: easily distributed to processing on multiple servers.

Will use $A[h(x)]$ to estimate $f(x)$, the frequency of $x$ in the stream. I.e., $|\{x_i : x_i = x\}|$. 
A common alternative to the Misra-Gries approach is the **count-min sketch**: a randomized method closely related to bloom filters.

- A major advantage: easily distributed to processing on multiple servers. **Build arrays** $A_1, \ldots, A_s$ separately and then just set $A := A_1 + \ldots + A_s$.

Will use $A[h(x)]$ to estimate $f(x)$, the frequency of $x$ in the stream. I.e., $|\{x_i : x_i = x\}|$. 
Use $A[h(x)]$ to estimate $f(x)$

**Claim 1:** We always have $A[h(x)] \geq f(x)$. Why?

- $A[h(x)]$ counts the number of occurrences of any $y$ with $h(y) = h(x)$, including $x$ itself.
- $A[h(x)] = f(x) + \sum_{y \neq x: h(y) = h(x)} f(y)$.

$f(x)$: frequency of $x$ in the stream (i.e., number of items equal to $x$). $h$: random hash function. $m$: size of count-min sketch array.
A[h(x)] = \( f(x) + \sum_{\substack{y \neq x: h(y) = h(x)\}} f(y) \).

Expected Error:

\[
\mathbb{E} \left[ \sum_{\text{y \neq x: h(y) = h(x)}} f(y) \right] = \sum_{\text{y \neq x}} \Pr(h(y) = h(x)) \cdot f(y) = \sum_{\text{y \neq x}} \frac{1}{m} \cdot f(y) = \frac{1}{m} \cdot (n - f(x)) \leq \frac{n}{m}.
\]

What is a bound on probability that the error is \( \geq \frac{3n}{m} \)?

Markov’s inequality: \( \Pr \left[ \sum_{\text{y \neq x: h(y) = h(x)}} f(y) \geq \frac{3n}{m} \right] \leq \frac{1}{3} \).

What property of \( h \) is required to show this bound? 2-universal.

\( f(x) \): frequency of \( x \) in the stream (i.e., number of items equal to \( x \)). \( h \): random hash function. \( m \): size of count-min sketch array.
**Claim:** For any $x$, with probability at least $2/3$,

$$f(x) \leq A[h(x)] \leq f(x) + \frac{\epsilon n}{k}.$$

To solve the $(\epsilon, k)$-Frequent elements problem, set $m = \frac{6k}{\epsilon}$. How can we improve the success probability? **Repetition.**

- $f(x)$: frequency of $x$ in the stream (i.e., number of items equal to $x$).
- $h$: random hash function.
- $m$: size of count-min sketch array.
COUNT-MIN SKETCH ACCURACY

Estimate $f(x)$ with $\sim f(x) = \min_{i} \left[ t A_i[h_i(x)] \right]$. (count-min sketch)

Why min instead of median?
The minimum estimate is always the most accurate since they are all overestimates of the true frequency!
Estimate $f(x)$ with $\tilde{f}(x) = \min_i \left[ t \right] A_i[h_i(x)]$. (count-min sketch)

Why min instead of median?
The minimum estimate is always the most accurate since they are all overestimates of the true frequency!
Estimate $f(x)$ with $\tilde{f}(x) = \min_i [t] A_i [h_i(x)]$. (count-min sketch)

Why min instead of median?
The minimum estimate is always the most accurate since they are all overestimates of the true frequency!
Estimate $f(x)$ with $\tilde{f}(x) = \min_i A_i[h_i(x)]$. (Count-min sketch)

Why min instead of median? The minimum estimate is always the most accurate since they are all overestimates of the true frequency!
Estimate $f(x)$ with $\tilde{f}(x) = \min_{i \in [t]} A_i[h_i(x)]$. (count-min sketch)
Estimate $f(x)$ with $\tilde{f}(x) = \min_{i \in [t]} A_i[h_i(x)]$. (count-min sketch)

Why min instead of median? The minimum estimate is always the most accurate since they are all overestimates of the true frequency!
Estimate $f(x)$ by $\tilde{f}(x) = \min_{i \in [t]} A_i[h_i(x)]$

- For every $x$ and $i \in [t]$, we know that for $m = O(k/\epsilon)$, with probability $\geq 2/3$:
  $$f(x) \leq A_i[h_i(x)] \leq f(x) + \frac{\epsilon n}{k}.$$

- What is $\Pr[f(x) \leq \tilde{f}(x) \leq f(x) + \frac{\epsilon n}{k}]$? $1 - 1/3^t$.

- To have a good estimate with probability $\geq 1 - \delta$, set $t = \log(1/\delta)$.
Upshot: Count-min sketch lets us estimate the frequency of every item in a stream up to error $\frac{en}{k}$ with probability $\geq 1 - \delta$ in $O(\log(1/\delta) \cdot k/\epsilon)$ space.

- Accurate enough to solve the $(\epsilon, k)$-Frequent elements problem.
- Actually identifying the frequent elements quickly requires a little bit of further work.

One approach: Store potential frequent elements as they come in. At step $i$ remove any elements whose estimated frequency is below $i/k$. Store at most $O(k)$ items at once and have all items with frequency $\geq n/k$ stored at the end of the stream.
Questions on Frequent Elements?