COMPSCI 514: ALGORITHMS FOR DATA SCIENCE

Cameron Musco

University of Massachusetts Amherst. Fall 2019.

Lecture 8

LOGISTICS

- 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.

Next Time: Random compression methods for high dimensional vectors. The Johnson-Lindenstrauss lemma.

- 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.

	Hash Table	Bloom Filters	MinHash Similarity Search	Distinct Elements
Goal	Check if x is a duplicate of any y in database and return y.	Check if x is a duplicate of y in database.	Check if x is a duplicate of any y in database and return y.	Count # of items, excluding duplicates.
Space	$O(n)$ items	$O(n)$ bits	$O(n \cdot t)$ items (when t tables used)	$O\left(\frac{\log \log n}{\epsilon^2}\right)$
Query Time	$O(1)$	$O(1)$	Potentially o(n)	NA
Approximate Duplicates?	✗	✗	✓	✗

All different variants of detecting duplicates/finding matches in large datasets. An important problem in many contexts!

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 $n = 9$, $k = 3$:

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
5	12	3	3	4	5	5	10	3

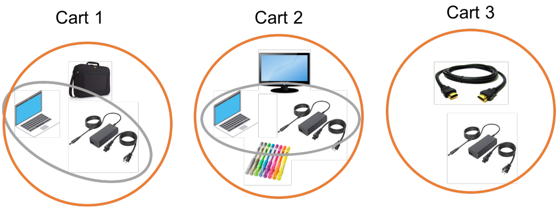
- What is the maximum number of items that must be returned? At most k items with frequency $\geq \frac{n}{k}$.
- Think of $k = 100$. Want items appearing $\geq 1\%$ of the time.
- Easy with $O(n)$ space – store the count for each item and return the one that appears $\geq n/k$ times.
- Can we do it 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.

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

- Basically k-Frequent items for $k = 2$ (and assume a single item has a strict majority.)

BOYER-MOORE ALGORITHM

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.

c=0, m=⊥

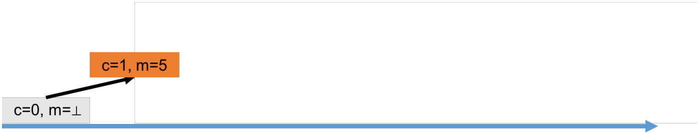
x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
5	12	3	5	4	5	5	10	5	5

BOYER-MOORE ALGORITHM

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.



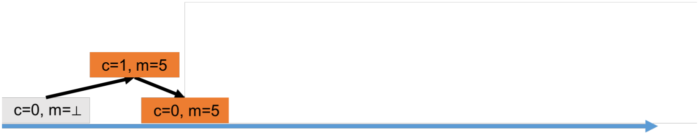
x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
5	12	3	5	4	5	5	10	5	5

BOYER-MOORE ALGORITHM

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.

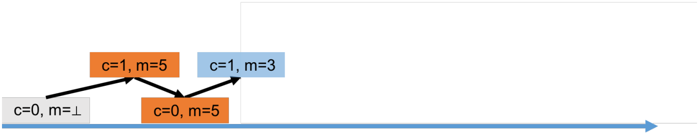


x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
5	12	3	5	4	5	5	10	5	5

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.



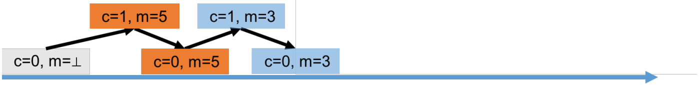
x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
5	12	3	5	4	5	5	10	5	5

BOYER-MOORE ALGORITHM

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.

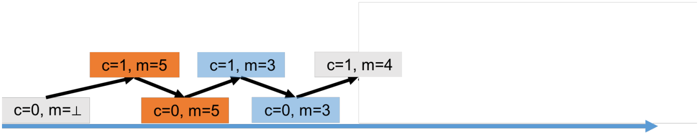


x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
5	12	3	5	4	5	5	10	5	5

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.

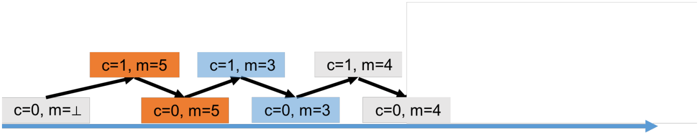


x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
5	12	3	5	4	5	5	10	5	5

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.



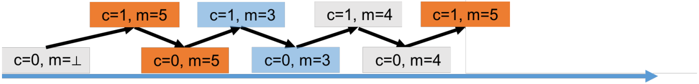
x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
5	12	3	5	4	5	5	10	5	5

BOYER-MOORE ALGORITHM

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.



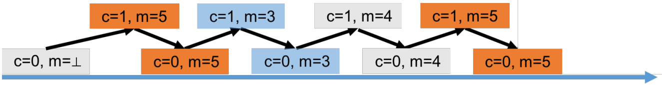
x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
5	12	3	5	4	5	5	10	5	5

BOYER-MOORE ALGORITHM

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.



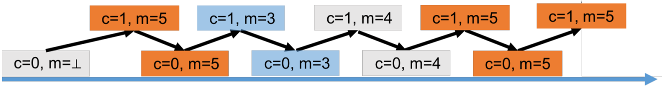
x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
5	12	3	5	4	5	5	10	5	5

BOYER-MOORE ALGORITHM

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.



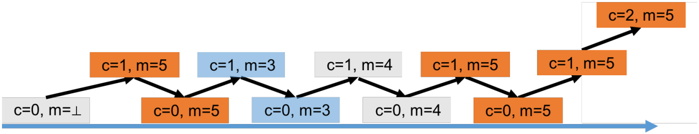
x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
5	12	3	5	4	5	5	10	5	5

BOYER-MOORE ALGORITHM

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.



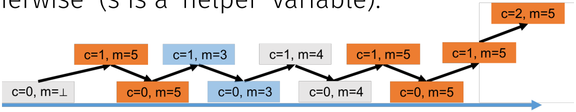
x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
5	12	3	5	4	5	5	10	5	5

Boyer-Moore Voting 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 $c := c - 1$.

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).



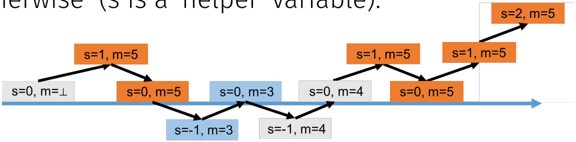
x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
5	12	3	5	4	5	5	10	5	5

Boyer-Moore Voting 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 $c := c - 1$.

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).



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

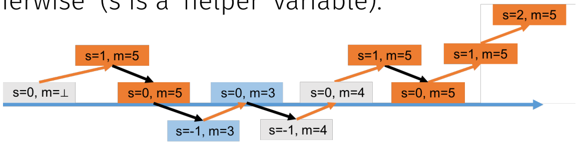
CORRECTNESS OF BOYER-MOORE

Boyer-Moore Voting 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 $c := c - 1$.

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).



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

Boyer-Moore Voting 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 $c := c - 1$.

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 at appears at least $\frac{n}{k}$ times.

Boyer-Moore Voting Algorithm:

- Initialize count $c := 0$, majority element $m := \perp$
- 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.

c_1=0, m_1=⊥

c_2=0, m_1=⊥

c_3=0, m_1=⊥

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
5	12	3	3	4	5	5	10	3

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.

c_1=1, m_1=5 ●

c_2=0, m_1=⊥

c_3=0, m_1=⊥

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
5	12	3	3	4	5	5	10	3

MISRA-GRIES ALGORITHM

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.

$c_1=1$, $m_1=5$

$c_2=1$, $m_1=12$

$c_3=0$, $m_1=\perp$

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
5	12	3	3	4	5	5	10	3

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.

c_1=1, m_1=5 ●

c_2=1, m_1=12 ●

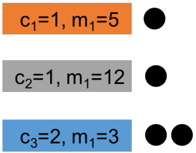
c_3=1, m_1=3 ●

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
5	12	3	3	4	5	5	10	3

MISRA-GRIES ALGORITHM

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.



x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
5	12	3	3	4	5	5	10	3

MISRA-GRIES ALGORITHM

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.

c_1=0, m_1=5

c_2=0, m_1=12

c_3=1, m_1=3

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
5	12	3	3	4	5	5	10	3

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.

c_1=1, m_1=5

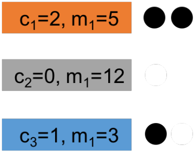
c_2=0, m_1=12

c_3=1, m_1=3

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
5	12	3	3	4	5	5	10	3

Misra-Gries Summary:

- Initialize counts $c_1, \dots, c_k := 0$, elements $m_1, \dots, m_k := \perp$.
- For $i = 1, \dots, n$
  - 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.



x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
5	12	3	3	4	5	5	10	3

MISRA-GRIES ALGORITHM

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.

c_1=2, m_1=5

c_2=1, m_1=10

c_3=1, m_1=3

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
5	12	3	3	4	5	5	10	3

MISRA-GRIES ALGORITHM

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.

c_1=2, m_1=5

c_2=1, m_1=10

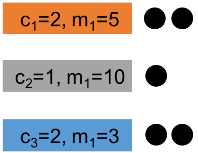
c_3=2, m_1=3

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
5	12	3	3	4	5	5	10	3

MISRA-GRIES ALGORITHM

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.



x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
5	12	3	3	4	5	5	10	3

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).

x_1	x_2	x_3	x_4	x_5	x_6	...	$x_{n-n/k+1}$	...	x_n
3	12	9	27	4	101		3		3

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).

(ϵ, 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 $r := \lceil k/\epsilon \rceil$
- Initialize counts $c_1, \ldots, c_r := 0$, elements $m_1, \ldots, m_r := \perp$.
- 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.
- Return any m_j with $c_j \geq (1 - \epsilon) \cdot \frac{n}{k}$.

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

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

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

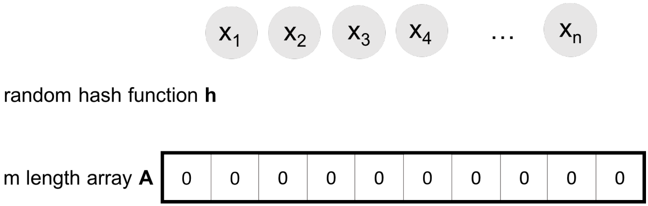
Implication: If $f(m_j) \geq \frac{n}{k}$, then $c_j \geq (1 - \epsilon) \cdot \frac{n}{k}$ so the item is returned.
If $f(m_j) < (1 - \epsilon) \cdot \frac{n}{k}$, then $c_j < (1 - \epsilon) \cdot \frac{n}{k}$ so the item is not returned.

Upshot: The (ϵ, 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.

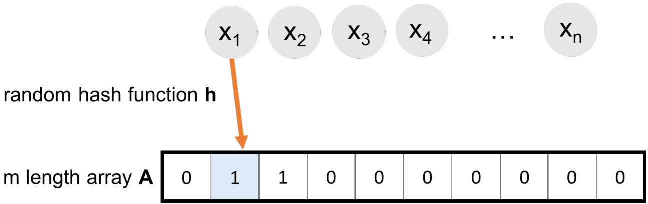
A common alternative to the Misra-Gries approach is the count-min sketch: a randomized method closely related to bloom filters.

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



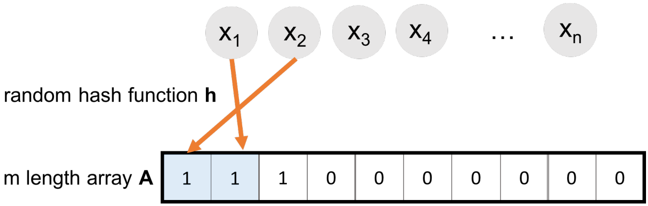
A common alternative to the Misra-Gries approach is the count-min sketch: a randomized method closely related to bloom filters.

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



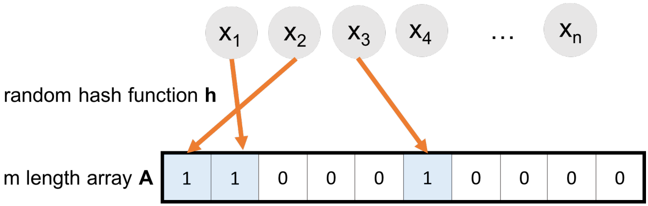
A common alternative to the Misra-Gries approach is the count-min sketch: a randomized method closely related to bloom filters.

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



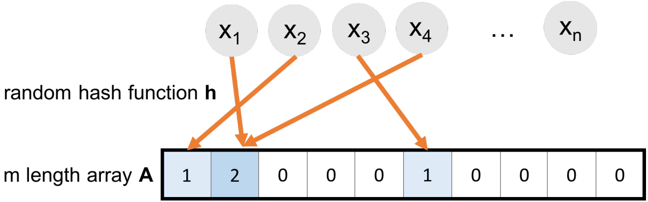
A common alternative to the Misra-Gries approach is the count-min sketch: a randomized method closely related to bloom filters.

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



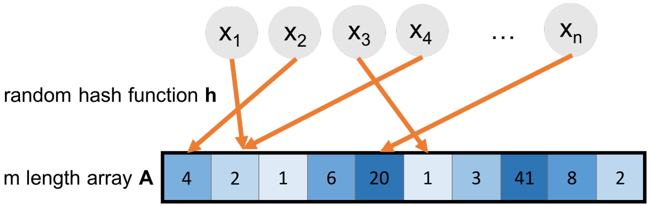
A common alternative to the Misra-Gries approach is the count-min sketch: a randomized method closely related to bloom filters.

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



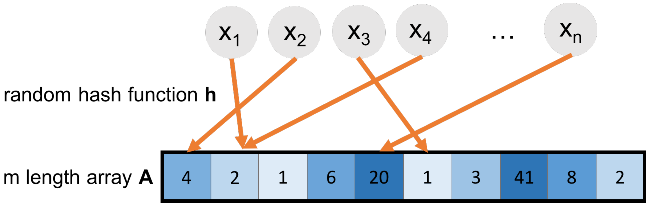
A common alternative to the Misra-Gries approach is the count-min sketch: a randomized method closely related to bloom filters.

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



A common alternative to the Misra-Gries approach is the count-min sketch: a randomized method closely related to bloom filters.

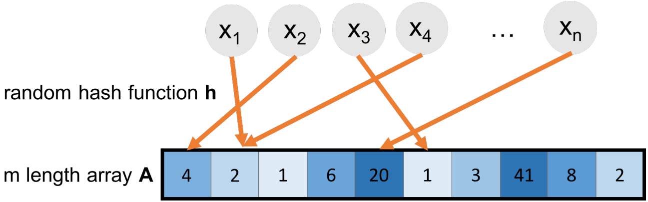
- A major advantage: easily distributed to processing on multiple servers.



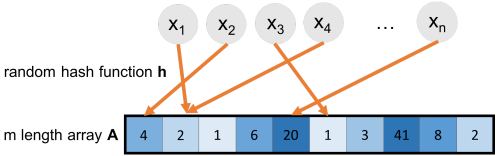
Will use $A[\mathbf{h}(x)]$ to estimate $f(x)$, the frequency of x in the stream. I.e., $|\{x_i : x_i = x\}|$.

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



Use $A[\mathbf{h}(x)]$ to estimate $f(x)$

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

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

$f(x)$: frequency of x in the stream (i.e., number of items equal to x). $\mathbf{h}$: random hash function. m: size of count-min sketch array.

COUNT-MIN SKETCH ACCURACY

$$A[\mathbf{h}(x)] = f(x) + \underbrace{\sum_{y \neq x : \mathbf{h}(y) = \mathbf{h}(x)} f(y)}_{\text{error in frequency estimate}}.$$

Expected Error:

$$\mathbb{E}\left[\sum_{y \neq x : \mathbf{h}(y) = \mathbf{h}(x)} f(y)\right] = \sum_{y \neq x} \Pr(\mathbf{h}(y) = \mathbf{h}(x)) \cdot f(y)$$

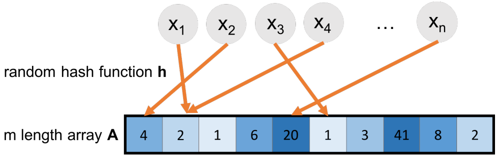
$$= \sum_{y \neq x} \frac{1}{m} \cdot f(y) = \frac{1}{m} \cdot (n - f(x)) \leq \frac{n}{m}$$

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

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

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.

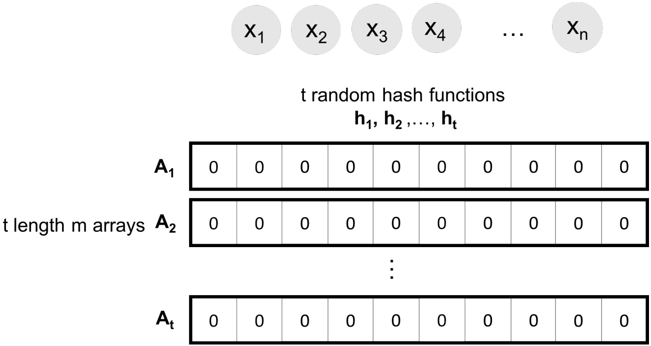


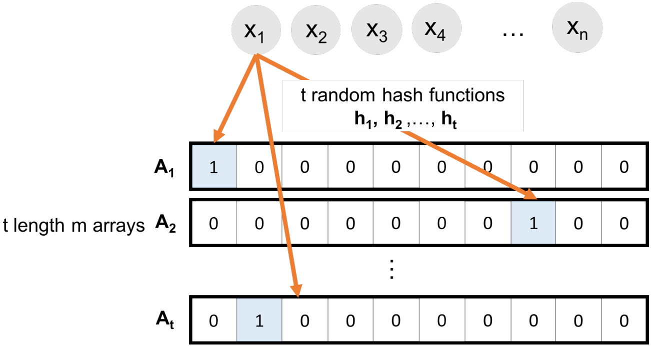
Claim: For any x, with probability at least $2/3$,

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

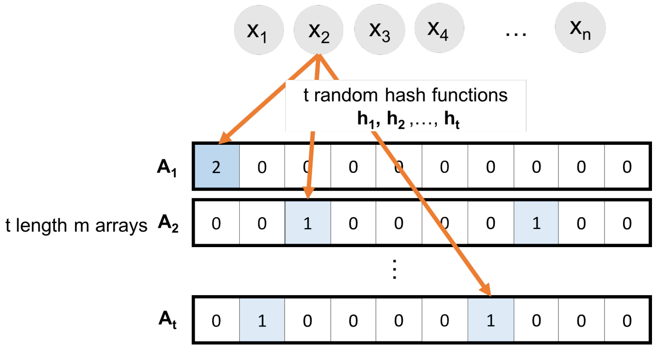
To solve the (ϵ, k)-Frequent elements problem, set $m = \frac{6k}{\epsilon}$.
How can we improve the success probability? Repetition.

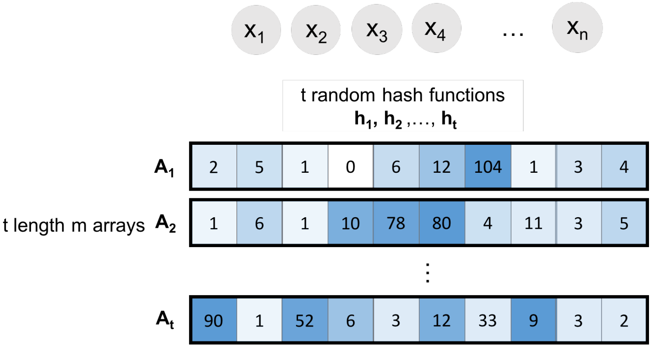
> $f(x)$: frequency of x in the stream (i.e., number of items equal to x). $\mathbf{h}$: random hash function. m: size of count-min sketch array.

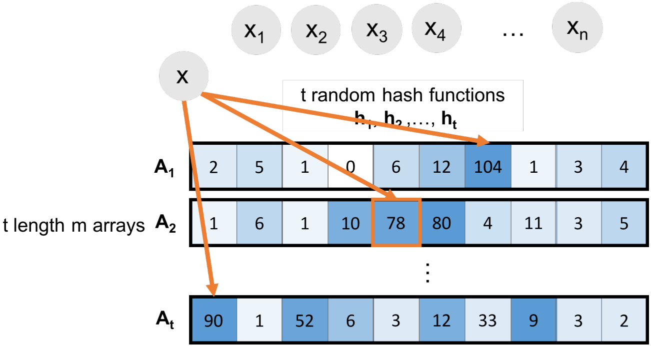

x1 x2 x3 x4 ... xn
t random hash functions
h1, h2 ,..., ht
A1
t length m arrays A2
At
0 0 0 0 0 0 0 0 0 0


x_1 x_2 x_3 x_4 … x_n
t random hash functions
$\mathbf{h_1, h_2, \dots, h_t}$
$\mathbf{A_1}$
1 0 0 0 0 0 0 0 0 0
t length m arrays
$\mathbf{A_2}$
0 0 0 0 0 0 0 1 0 0
$\mathbf{A_t}$
0 1 0 0 0 0 0 0 0 0

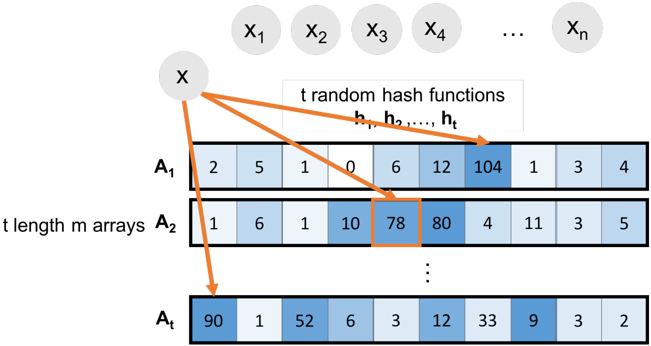
COUNT-MIN SKETCH ACCURACY




x_1
x_2
x_3
x_4
...
x_n
t random hash functions
$\mathbf{h_1, h_2, \ldots, h_t}$
t length m arrays
$\mathbf{A_1}$
2 5 1 0 6 12 104 1 3 4
$\mathbf{A_2}$
1 6 1 10 78 80 4 11 3 5
$\mathbf{A_t}$
90 1 52 6 3 12 33 9 3 2

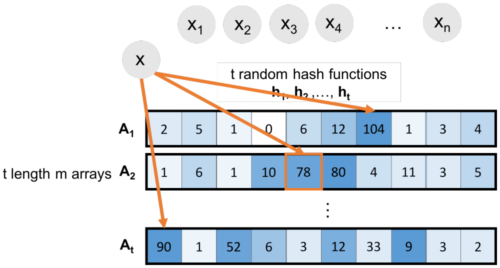


Estimate $f(x)$ with $\tilde{f}(x) = \min_{i \in [t]} A_i[\mathbf{h}_i(x)]$. (count-min sketch)



Estimate $f(x)$ with $\tilde{f}(x) = \min_{i\in[t]} A_i[\mathbf{h}_i(x)]$. (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 $\tilde{f}(x) = \min_{i \in [t]} A_i[\mathbf{h}_i(x)]$

- For every x and $i \in [t]$, we know that for $m = O(k/\epsilon)$, with probability $\geq 2/3$:

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

- 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 $\frac{\epsilon n}{k}$ with probability $\geq 1 - \delta$ in $O\left(\log(1/\delta) \cdot k/\epsilon\right)$ space.

- Accurate enough to solve the (ϵ, 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?