COMPSCI 514: ALGORITHMS FOR DATA SCIENCE

Cameron Musco
University of Massachusetts Amherst. Spring 2020.
Lecture 7
LOGISTICS

- Problem Set 1 is due tomorrow at 8pm in Gradescope.
- No class next Tuesday (it’s a Monday at UMass).
• Problem Set 1 is due tomorrow at 8pm in Gradescope.
• No class next Tuesday (it’s a Monday at UMass).
• **Talk Today:** Vatsal Sharan at 4pm in CS 151. *Modern Perspectives on Classical Learning Problems: Role of Memory and Data Amplification.*
Last Class:
Last Class: Hashing for Jaccard Similarity

- MinHash for estimating the Jaccard similarity.
- Locality sensitive hashing (LSH).
- Application to fast similarity search.
Last Class: Hashing for Jaccard Similarity

- MinHash for estimating the Jaccard similarity.
- Locality sensitive hashing (LSH).
- Application to fast similarity search.

This Class:

- Finish up MinHash and LSH.
- The Frequent Elements (heavy-hitters) problem.
- Misra-Gries summaries.
Jaccard Similarity: \( J(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{\# \text{ shared elements}}{\# \text{ total elements}}. \)

Two Common Use Cases:

- **Near Neighbor Search**: Have a database of \( n \) sets/bit strings and given a set \( A \), want to find if it has high similarity to anything in the database. Naively \( \Omega(n) \) time.

- **All-pairs Similarity Search**: Have \( n \) different sets/bit strings. Want to find all pairs with high similarity. Naively \( \Omega(n^2) \) time.
MinHash(A) = \min_{a \in A} h(a) where h : U \to [0, 1] is a random hash.
\[ \text{MinHash}(A) = \min_{a \in A} h(a) \] where \( h : U \rightarrow [0, 1] \) is a random hash.

**Locality Sensitivity:** \[ \Pr[\text{MinHash}(A) = \text{MinHash}(B)] = J(A, B). \]
MINHASHING

\[ \text{MinHash}(A) = \min_{a \in A} h(a) \] where \( h : U \to [0, 1] \) is a random hash.

**Locality Sensitivity:** \( \Pr[\text{MinHash}(A) = \text{MinHash}(B)] = J(A, B). \)

Represents a set with a **single number** that captures Jaccard similarity information!
MinHash$(A) = \min_{a \in A} h(a)$ where $h : U \rightarrow [0, 1]$ is a random hash.

**Locality Sensitivity:** $\Pr[\text{MinHash}(A) = \text{MinHash}(B)] = J(A, B)$.

Represents a set with a single number that captures Jaccard similarity information!

Given a collision free hash function $g : [0, 1] \rightarrow [m],$

$$\Pr [g(\text{MinHash}(A)) = g(\text{MinHash}(B))] = J(A, B).$$
MinHash(A) = \min_{a \in A} h(a) \text{ where } h : U \rightarrow [0, 1] \text{ is a random hash.}

**Locality Sensitivity:** Pr[MinHash(A) = MinHash(B)] = J(A, B).

Represents a set with a single number that captures Jaccard similarity information!

Given a collision free hash function \( g : [0, 1] \rightarrow [m] \),

\[
\Pr \left[ g(\text{MinHash}(A)) = g(\text{MinHash}(B)) \right] = J(A, B).
\]

What is \( \Pr \left[ g(\text{MinHash}(A)) = g(\text{MinHash}(B)) \right] \) if \( g \) is not collision free?
**MinHashing**

MinHash\(A) = \min_{a \in A} h(a)\) where \(h: U \rightarrow [0, 1]\) is a random hash.

**Locality Sensitivity:** \(\Pr[\text{MinHash}(A) = \text{MinHash}(B)] = J(A, B)\).

Represents a set with a **single number** that captures Jaccard similarity information!

Given a collision free hash function \(g: [0, 1] \rightarrow [m]\),

\[
\Pr [g(\text{MinHash}(A)) = g(\text{MinHash}(B))] = J(A, B).
\]

What is \(\Pr [g(\text{MinHash}(A)) = g(\text{MinHash}(B))]\) if \(g\) is not collision free? Will be a bit larger than \(J(A, B)\).
When searching for similar items only search for matches that land in the same hash bucket.

- **False Negative**: A similar pair doesn’t appear in the same bucket.
- **False Positive**: A dissimilar pair is hashed to the same bucket.
When searching for similar items only search for matches that land in the same hash bucket.

- **False Negative:** A similar pair doesn’t appear in the same bucket.
- **False Positive:** A dissimilar pair is hashed to the same bucket.

Need to balance a small probability of false negatives (a high hit rate) with a small probability of false positives (a small query time.)
Balancing False Negatives/Positives with MinHash via repetition.

Create $t$ hash tables. Each is indexed into not with a single MinHash value, but with $r$ values, appended together. A length $r$ signature:

$$\text{MH}_{i,1}(x), \text{MH}_{i,2}(x), \ldots, \text{MH}_{i,r}(x).$$

Hit Rate: Given by the $s$-curve: $1 - (1 - s^r)^t$. 
Repetition and s-curve tuning can be used for fast similarity search with any similarity metric, given a locality sensitive hash function for that metric.
Repetition and s-curve tuning can be used for fast similarity search with any similarity metric, given a locality sensitive hash function for that metric.

- LSH schemes exist for many similarity/distance measures: hamming distance, cosine similarity, etc.
Repetition and s-curve tuning can be used for fast similarity search with any similarity metric, given a locality sensitive hash function for that metric.

- LSH schemes exist for many similarity/distance measures: hamming distance, cosine similarity, etc.
Repetition and s-curve tuning can be used for fast similarity search with any similarity metric, given a locality sensitive hash function for that metric.

- LSH schemes exist for many similarity/distance measures: hamming distance, cosine similarity, etc.
Repetition and s-curve tuning can be used for fast similarity search with any similarity metric, given a locality sensitive hash function for that metric.

- LSH schemes exist for many similarity/distance measures: hamming distance, cosine similarity, etc.

Cosine Similarity: \( \cos(\theta(x, y)) \)
Repetition and s-curve tuning can be used for fast similarity search with any similarity metric, given a locality sensitive hash function for that metric.

- LSH schemes exist for many similarity/distance measures: hamming distance, cosine similarity, etc.

Cosine Similarity: $\cos(\theta(x, y))$

- $\cos(\theta(x, y)) = 1$ when $\theta(x, y) = 0^\circ$ and $\cos(\theta(x, y)) = 0$ when $\theta(x, y) = 90^\circ$, and $\cos(\theta(x, y)) = -1$ when $\theta(x, y) = 180^\circ$
Repetition and s-curve tuning can be used for fast similarity search with any similarity metric, given a locality sensitive hash function for that metric.

- LSH schemes exist for many similarity/distance measures: hamming distance, cosine similarity, etc.

**Cosine Similarity:** \( \cos(\theta(x, y)) = \frac{\langle x, y \rangle}{\|x\|_2 \cdot \|y\|_2} \).

- \( \cos(\theta(x, y)) = 1 \) when \( \theta(x, y) = 0^\circ \) and \( \cos(\theta(x, y)) = 0 \) when \( \theta(x, y) = 90^\circ \), and \( \cos(\theta(x, y)) = -1 \) when \( \theta(x, y) = 180^\circ \)
SimHash Algorithm: LSH for cosine similarity.
SimHash Algorithm: LSH for cosine similarity.
SimHash Algorithm: LSH for cosine similarity.
SimHash Algorithm: LSH for cosine similarity.
SimHash Algorithm: LSH for cosine similarity.

Sim\(Hash(x) = \text{sign}(\langle x, t \rangle)\) for a random vector \(t\).
SimHash Algorithm: LSH for cosine similarity.

\[ \text{SimHash}(x) = \text{sign}(\langle x, t \rangle) \] for a random vector \( t \).

What is \( \Pr [\text{SimHash}(x) = \text{SimHash}(y)] \)?
What is \( \Pr[\text{SimHash}(x) = \text{SimHash}(y)] \)?
What is $\Pr[\text{SimHash}(x) = \text{SimHash}(y)]$?

$\text{SimHash}(x) \neq \text{SimHash}(y)$ when the plane separates $x$ from $y$. 
What is $\Pr[\text{SimHash}(x) = \text{SimHash}(y)]$?

$\text{SimHash}(x) \neq \text{SimHash}(y)$ when the plane separates $x$ from $y$. 

![Diagram showing the relationship between SimHash(x) and SimHash(y)]
What is \( \Pr[\text{SimHash}(x) = \text{SimHash}(y)] \)?

\( \text{SimHash}(x) \neq \text{SimHash}(y) \) when the plane separates \( x \) from \( y \).

\[
\Pr[\text{SimHash}(x) \neq \text{SimHash}(y)] = \frac{\theta(x,y)}{\pi}
\]
What is $\Pr [\text{SimHash}(x) = \text{SimHash}(y)]$?

$\text{SimHash}(x) \neq \text{SimHash}(y)$ when the plane separates $x$ from $y$.

- $\Pr [\text{SimHash}(x) \neq \text{SimHash}(y)] = \frac{\theta(x,y)}{\pi}$
- $\Pr [\text{SimHash}(x) = \text{SimHash}(y)] = 1 - \frac{\theta(x,y)}{\pi} \approx \frac{\cos(\theta(x,y)) + 1}{2}$
Many applications outside traditional similarity search. E.g., approximate neural net computation (Anshumali Shrivastava).
Many applications outside traditional similarity search. E.g., approximate neural net computation (Anshumali Shrivastava).
Many applications outside traditional similarity search. E.g., approximate neural net computation (Anshumali Shrivastava).

- Evaluating $\mathcal{N}(x)$ requires $|x| \cdot |\text{layer 1}| + |\text{layer 1}| \cdot |\text{layer 2}| + \ldots$ multiplications if fully connected.
Many applications outside traditional similarity search. E.g., approximate neural net computation (Anshumali Shrivastava).

- Evaluating $\mathcal{N}(x)$ requires $|x| \cdot |\text{layer 1}| + |\text{layer 1}| \cdot |\text{layer 2}| + \ldots$ multiplications if fully connected.

- Can be expensive, especially on constrained devices like cellphones, cameras, etc.
Many applications outside traditional similarity search. E.g., approximate neural net computation (Anshumali Shrivastava).

- Evaluating $\mathcal{N}(x)$ requires $|x| \cdot |\text{layer 1}| + |\text{layer 1}| \cdot |\text{layer 2}| + \ldots$ multiplications if fully connected.
- Can be expensive, especially on constrained devices like cellphones, cameras, etc.
- For approximate evaluation, suffices to identify the neurons in each layer with high activation when $x$ is presented.
Many applications outside traditional similarity search. E.g., approximate neural net computation (Anshumali Shrivastava).

- Evaluating $N(x)$ requires $|x| \cdot |\text{layer 1}| + |\text{layer 1}| \cdot |\text{layer 2}| + \ldots$ multiplications if fully connected.
- Can be expensive, especially on constrained devices like cellphones, cameras, etc.
- For approximate evaluation, suffices to identify the neurons in each layer with high activation when $x$ is presented.
Nonlinearity $\sigma$

$$n_i = \sigma \left( \sum_{j=1}^{m} w(x_j, n_i) \cdot x_j \right) = \sigma(\langle w_i, x \rangle)$$
Important neurons have high activation $\sigma(\langle w_i, x \rangle)$.
• Important neurons have high activation $\sigma(\langle w_i, x \rangle)$.
• Since $\sigma$ is typically monotonic, this means large $\langle w_i, x \rangle$. 

$X + w_i$ are similar under cosine similarity.
• Important neurons have high activation $\sigma(\langle w_i, x \rangle)$.
• Since $\sigma$ is typically monotonic, this means large $\langle w_i, x \rangle$.
• $\cos(\theta(w_i, x)) = \frac{\langle w_i, x \rangle}{\|w_i\| \|x\|}$. Thus these neurons can be found very quickly using LSH for cosine similarity search.
Important neurons have high activation $\sigma(\langle w_i, x \rangle)$.
- Since $\sigma$ is typically monotonic, this means large $\langle w_i, x \rangle$.
- $\cos(\theta(w_i, x)) = \frac{\langle w_i, x \rangle}{\|w_i\| \|x\|}$. Thus these neurons can be found very quickly using LSH for cosine similarity search.
- Store each weight vector $w_i$ (corresponding to each node) in a set of hash tables and check inputs $x$ for similarity to these stored vectors.
Questions on MinHash and Locality Sensitive Hashing?
**THE FREQUENT ITEMS PROBLEMS**

\textbf{*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.
**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.

\[ k = 3 \]

<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>
**$k$-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.

<p>| | | | | | | | | |</p>
<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>$x_1$</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>
</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>
</tr>
</tbody>
</table>
**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.

\[
\frac{n}{k} = 3
\]

<p>| | | | | | | | | |</p>
<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>( 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>
</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>
</tr>
</tbody>
</table>

- What is the maximum number of items that must be returned?
**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.

<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 $\geq \frac{n}{k}$. 
**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.

<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 \( \geq \frac{n}{k} \).
- Trivial with \( O(n) \) space – store the count for each item and return the one that appears \( \geq n/k \) times.
**THE FREQUENT ITEMS PROBLEMS**

\(k\)-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.

<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>(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>

- What is the maximum number of items that must be returned? At most \(k\) items with frequency \(\geq \frac{n}{k}\).
- Trivial 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?
**The Frequent Items Problems**

**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.

<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 \( \geq \frac{n}{k} \).
- Trivial 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:
Applications of Frequent Items:

- Finding top/viral items (i.e., products on Amazon, videos watched on Youtube, Google searches, etc.)
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).
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.
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.
**Association rule learning:** A very common task in data mining is to identify common associations between different events.
Association rule learning: A very common task in data mining is to identify common associations between different events.
**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.
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.
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.
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>

\( \downarrow \ \downarrow \ \downarrow \ \downarrow \)
**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 $m := \bot$
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 $c := c - 1$. 
BOYER-MOORE ALGORITHM

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 $m := \bot$
- 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 \( m := \bot \)
- 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 $m := \bot$
- 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 $m := \bot$
- 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>
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 $m := \bot$
- 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>
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>
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 (if it is a strict majority).
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.

\[
\begin{array}{c}
5 & 5 & 5 & 4 & 4 & 5 & 5
\end{array}
\]

\[
\begin{array}{c}
c = 0 \\
m = 5 \\
m = 5 \\
c = 1 \\
m = 5
\end{array}
\]
**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.
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).
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).
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).
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).

\[ S = 1 \]
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 $m = M$. 
**Next Time:** Will see a variant on the Boyer-Moore algorithm – the Misra-Greis summary.

- Stores $k$ top items at once and solves the Frequent Items problem.