Hashing and Sketching
Part One
Randomized Data Structures

- Randomization is a powerful tool for improving efficiency and solving problems under seemingly impossible constraints.
- Over the next three lectures, we’ll explore a sampler of data structures that give a feel for the breadth of what’s out there.
- You can easily spend an entire academic career just exploring this space; take CS265 for more on randomized algorithms!
Where We’re Going

- **Hashing and Sketching (Thursday / Tuesday)**
  - Using hash functions to count without counting.

- **Cuckoo Hashing (Next Thursday)**
  - Hashing with *worst-case* O(1) lookups, along with a splash of random hypergraph theory.
Outline for Today

• **Hash Functions**
  • Understanding our basic building blocks.

• **Frequency Estimation**
  • Estimating how many times we’ve seen something.

• **Probabilistic Techniques**
  • Standard but powerful tools for reasoning about randomized data structures.
Preliminaries: *Hash Functions*
Hashing in Practice

• Hash functions are used extensively in programming and software engineering:
  • They make hash tables possible: think C++ std::hash, Python’s __hash__, or Java’s Object.hashCode().
  • They’re used in cryptography: SHA-256, HMAC, etc.

• **Question:** When we’re in Theoryland, what do we mean when we say “hash function?”
Hashing in Theoryland

- In Theoryland, a hash function is a function from some domain called the \textit{universe} (typically denoted $\mathcal{U}$) to some codomain.

- The codomain is usually a set of the form $[m] = \{0, 1, 2, 3, \ldots, m - 1\}$

\[
h : \mathcal{U} \rightarrow [m]\]
Hashing in Theoryland

- **Intuition:** No matter how clever you are with designing a specific hash function, that hash function isn’t random, and so there will be pathological inputs.
  - You can formalize this with the pigeonhole principle.
- **Idea:** Rather than finding the One True Hash Function, we’ll assume we have a collection of hash functions to pick from, and we’ll choose which one to use randomly.
Families of Hash Functions

- A *family* of hash functions is a set $\mathcal{H}$ of hash functions with the same domain and codomain.
- We can then introduce randomness into our data structures by sampling a random hash function from $\mathcal{H}$.
- **Key Point:** The randomness in our data structures almost always derives from the random choice of hash functions, not from the data.

*Data is adversarial.*

*Hash function selection is random.*

- **Question:** What makes a family of hash functions $\mathcal{H}$ a “good” family of hash functions?
**Goal:** If we pick \( h \in \mathcal{H} \) uniformly at random, then \( h \) should distribute elements uniformly randomly.

**Problem:** A hash function that distributes \( n \) elements uniformly at random over \([m]\) requires \( \Omega(n \log m) \) space in the worst case.

**Question:** Do we actually need true randomness? Or can we get away with something weaker?
Distribution Property: Each element should have an equal probability of being placed in each slot.

Independence Property: Where one element is placed shouldn’t impact where a second goes.

For any $x \in \mathcal{U}$ and random $h \in \mathcal{H}$, the value of $h(x)$ is uniform over its codomain.

For any distinct $x, y \in \mathcal{U}$ and random $h \in \mathcal{H}$, $h(x)$ and $h(y)$ are independent random variables.

A family of hash functions $\mathcal{H}$ is called 2-independent (or pairwise independent) if it satisfies the distribution and independence properties.
For any $x \in \mathcal{U}$ and random $h \in \mathcal{H}$, the value of $h(x)$ is uniform over its codomain.

**Intuition:**
2-independence means any pair of elements is unlikely to collide.

For any distinct $x, y \in \mathcal{U}$ and random $h \in \mathcal{H}$, $h(x)$ and $h(y)$ are independent random variables.

$$
\Pr[h(x) = h(y)] = \sum_{i=0}^{m-1} \Pr[h(x) = i \land h(y) = i]
$$

**Question:** Where did these elements collide with one another?
For any \( x \in \mathcal{U} \) and random \( h \in \mathcal{H} \), the value of \( h(x) \) is uniform over its codomain.

**Intuition:**
2-independence means any pair of elements is unlikely to collide.

\[
\Pr[h(x) = h(y)] = \sum_{i=0}^{m-1} \Pr[h(x) = i \land h(y) = i] = \sum_{i=0}^{m-1} \Pr[h(x) = i] \cdot \Pr[h(y) = i]
\]
For any $x \in \mathcal{U}$ and random $h \in \mathcal{H}$, the value of $h(x)$ is uniform over its codomain.

**Intuition:**
2-independence means any pair of elements is unlikely to collide.

For any distinct $x, y \in \mathcal{U}$ and random $h \in \mathcal{H}$, $h(x)$ and $h(y)$ are independent random variables.

\[
\Pr[h(x) = h(y)]
= \sum_{i=0}^{m-1} \Pr[h(x) = i \land h(y) = i]
= \sum_{i=0}^{m-1} \Pr[h(x) = i] \cdot \Pr[h(y) = i]
= \sum_{i=0}^{m-1} \frac{1}{m^2}
\]
For any $x \in \mathcal{U}$ and random $h \in \mathcal{H}$, the value of $h(x)$ is uniform over its codomain.

**Intuition:**
2-independence means any pair of elements is unlikely to collide.

$$
\Pr[h(x) = h(y)] \\
= \sum_{i=0}^{m-1} \Pr[h(x) = i \land h(y) = i] \\
= \sum_{i=0}^{m-1} \Pr[h(x) = i] \cdot \Pr[h(y) = i] \\
= \sum_{i=0}^{m-1} \frac{1}{m^2} \\
= \frac{1}{m}
$$

This is the same as if $h$ were a truly random function.
For more on hashing outside of Theoryland, check out *this Stack Exchange post*. 
Frequency Estimation
**Frequency Estimators**

- A *frequency estimator* is a data structure supporting the following operations:
  - `increment(x)`, which increments the number of times that $x$ has been seen, and
  - `estimate(x)`, which returns an estimate of the frequency of $x$.

- Using BSTs, we can solve this in space $\Theta(n)$ with worst-case $O(\log n)$ costs on the operations.

- Using hash tables, we can solve this in space $\Theta(n)$ with expected $O(1)$ costs on the operations.
Frequency Estimators

• Frequency estimation has many applications:
  • Search engines: Finding frequent search queries.
  • Network routing: Finding common source and destination addresses.
• In these applications, $\Theta(n)$ memory can be impractical.
• **Goal:** Get *approximate* answers to these queries in sublinear space.
The Count-Min Sketch
# How to Build an Estimator

<table>
<thead>
<tr>
<th>Step One: Build a Simple Estimator</th>
<th>Count-Min Sketch</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Revisiting the Exact Solution

• In the exact solution to the frequency estimation problem, we maintained a single counter for each distinct element. This is too space-inefficient.

• **Idea:** Store a fixed number of counters and assign a counter to each \( x \in \mathcal{U} \). Multiple objects might be assigned to the same counter.

• To \textit{increment}(x), increment the counter for \( x \).

• To \textit{estimate}(x), read the value of the counter for \( x \).
Our Initial Structure

- Create an array of counters, all initially 0, called \texttt{count}. It will have \( w \) elements for some \( w \) we choose later.
- Choose, from a family of 2-independent hash functions \( \mathcal{H} \), a uniformly-random hash function \( h : \mathcal{U} \rightarrow [w] \).
- To \texttt{increment}(x), increment \texttt{count}[h(x)].
- To \texttt{estimate}(x), return \texttt{count}[h(x)].
# How to Build an Estimator

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
<th>Count-Min Sketch</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Step One:</strong> Build a Simple Estimator</td>
<td>Hash items to counters; add +1 when item seen.</td>
<td></td>
</tr>
<tr>
<td><strong>Step Two:</strong> Compute Expected Value of Estimator</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Count-Min Sketch*
Some Notation

- Let $x_1, x_2, x_3, \ldots$ denote the list of distinct items whose frequencies are being stored.
- Let $a_1, a_2, a_3, \ldots$ denote the frequencies of those items.
  - e.g. $a_i$ is the true number of times $x_i$ is seen.
- Let $\hat{a}_1, \hat{a}_2, \hat{a}_3, \ldots$ denote the estimate our data structure gives for the frequency of each item.
  - e.g. $\hat{a}_i$ is our estimate for how many times $x_i$ has been seen.
  - **Important detail:** the $a_i$ values are not random variables (data are chosen adversarially), while the $\hat{a}_i$ values are random variables (they depend on a randomly-sampled hash function).
- In what follows, imagine we’re querying the frequency of some specific element $x_i$. We want to analyze $\hat{a}_i$. 
Analyzing our Estimator

• We’re interested in learning more about $\hat{a}_i$. A good first step is to work out $E[\hat{a}_i]$.

• $\hat{a}_i$ will be equal to $a_i$, plus some “noise” terms from colliding elements.

• Each of those elements is very unlikely to collide with us, though. (There’s a $\frac{1}{w}$ chance of a collision for any one other element.)

• **Reasonable guess:** $E[\hat{a}_i] = a_i + \sum_{j \neq i} \frac{a_j}{w}$

Frequency of each other item, scaled to account for chance of a collision.
Making Things Formal

- Let’s make this more rigorous.
- For each element $x_j$:
  - If $h(x_i) = h(x_j)$, then $x_j$ contributes $a_j$ to $\text{count}[h(x_i)]$.
  - If $h(x_i) \neq h(x_j)$, then $x_j$ contributes 0 to $\text{count}[h(x_i)]$.
- To pin this down precisely, let’s define a set of random variables $X_1, X_2, \ldots$, as follows:

$$X_j = \begin{cases} 
1 & \text{if } h(x_i) = h(x_j) \\
0 & \text{otherwise}
\end{cases}$$

Each of these variables is called an *indicator random variable*, since it “indicates” whether some event occurs.
Making Things Formal

Let’s make this more rigorous.

For each element $x_j$:
- If $h(x_i) = h(x_j)$, then $x_j$ contributes $a_j$ to $\text{count}[h(x_i)]$.
- If $h(x_i) \neq h(x_j)$, then $x_j$ contributes 0 to $\text{count}[h(x_i)]$.

To pin this down precisely, let’s define a set of random variables $X_1, X_2, \ldots$, as follows:

$$X_j = \begin{cases} 
1 & \text{if } h(x_i) = h(x_j) \\
0 & \text{otherwise}
\end{cases}$$

The value of $\hat{a}_i - a_i$ is then given by

$$\hat{a}_i - a_i = \sum_{j \neq i} a_j X_j$$
$$E[\hat{a}_i - a_i] = E[\sum_{j \neq i} a_j X_j]$$

$$= \sum_{j \neq i} E[a_j X_j]$$

This follows from **linearity of expectation**. We’ll use this property extensively over the next few days.
\[ E[\hat{a}_i - a_i] = \mathbb{E}[\sum_{j \neq i} a_j X_j] \]
\[ = \sum_{j \neq i} \mathbb{E}[a_j X_j] \]
\[ = \sum_{j \neq i} a_j \mathbb{E}[X_j] \]

The values of \( a_j \) are not random. *The randomness comes from our choice of hash function.*
\[ E[\hat{a}_i - a_i] = E\left[ \sum_{j \neq i} a_j X_j \right] \]
\[ = \sum_{j \neq i} E[a_j X_j] \]
\[ = \sum_{j \neq i} a_j E[X_j] \]

\[ E[X_j] = 1 \cdot \Pr[h(x_i) = h(x_j)] + 0 \cdot \Pr[h(x_i) \neq h(x_j)] \]

\[ X_j = \begin{cases} 1 & \text{if } h(x_i) = h(x_j) \\ 0 & \text{otherwise} \end{cases} \]
\[ E[\hat{a}_i - a_i] = E[\sum_{j \neq i} a_j X_j] \]

\[ = \sum_{j \neq i} E[a_j X_j] \]

\[ = \sum_{j \neq i} a_j E[X_j] \]

---

\[ E[X_j] = 1 \cdot \Pr[h(x_i) = h(x_j)] + 0 \cdot \Pr[h(x_i) \neq h(x_j)] \]

\[ = \Pr[h(x_i) = h(x_j)] \]

If \( X \) is an indicator variable for some event \( \mathcal{E} \), then \( E[X] = \Pr[\mathcal{E}] \). This is really useful when using linearity of expectation!
\[
\begin{align*}
E[\hat{a}_i - a_i] &= E\left[ \sum_{j \neq i} a_j X_j \right] \\
&= \sum_{j \neq i} E[a_j X_j] \\
&= \sum_{j \neq i} a_j E[X_j] \\
&= \sum_{j \neq i} \frac{a_j}{w}
\end{align*}
\]

\[
\begin{align*}
E[X_j] &= 1 \cdot \Pr[h(x_i) = h(x_j)] + 0 \cdot \Pr[h(x_i) \neq h(x_j)] \\
&= \Pr[h(x_i) = h(x_j)] \\
&= \frac{1}{w}
\end{align*}
\]

Hey, we saw this earlier!
\[ E[ \hat{a}_i - a_i ] = E[ \sum_{j \neq i} a_j X_j ] \]

\[ = \sum_{j \neq i} E[ a_j X_j ] \]

\[ = \sum_{j \neq i} a_j E[ X_j ] \]

\[ \leq \frac{\|a\|_1}{w} \]

**Idea:** Think of our element frequencies \( a_1, a_2, a_3, \ldots \) as a vector \( a = [a_1, a_2, a_3, \ldots] \)

The total number of objects is the sum of the vector entries.

This is called the **L₁ norm** of \( a \), and is denoted \( \|a\|_1 \):

\[ \|a\|_1 = \sum_i |a_i| \]
\[ E[\hat{a}_i - a_i] = E[\sum_{j \neq i} a_j X_j] \]

\[ = \sum_{j \neq i} E[a_j X_j] \]

\[ = \sum_{j \neq i} a_j E[X_j] \]

\[ = \sum_{j \neq i} \frac{a_j}{w} \]

\[ \leq \frac{||a||_1}{w} \]
# How to Build an Estimator

<table>
<thead>
<tr>
<th><strong>Step One:</strong> Build a Simple Estimator</th>
<th>Count-Min Sketch</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Step Two:</strong> Compute Expected Value of Estimator</td>
<td>Hash items to counters; add +1 when item seen.</td>
</tr>
<tr>
<td><strong>Step Three:</strong> Apply Concentration Inequality</td>
<td>Sum of indicators; 2-independent hashes have low collision rate.</td>
</tr>
</tbody>
</table>

- Hash items to counters; add +1 when item seen.
- Sum of indicators; 2-independent hashes have low collision rate.
On Expected Values

- We know that \( E[\hat{a}_i - a_i] \leq ||a||_1 / w \). This means that the expected overestimate is low.

- **Claim:** This fact, in isolation, is not very useful.

- Below is a probability distribution for a random variable whose expected value is 9 that never takes values near 9.

- If this is the sort of distribution we get for \( \hat{a}_i \), then our estimator is not very useful!
On Expected Values

• We’re looking for a way to say something like the following:

  “Not only is our estimate’s expected value pretty close to the real value, our estimate has a high probability of being close to the real value.”

• In other words, if the true frequency is 9, we want the distribution of our estimate to kinda sorta look like this:

If the true frequency is 9, why isn’t there any probability mass below 9?

Answer at https://pollev.com/cs166spr23
On Expected Values

- We’re looking for a way to say something like the following:

  “Not only is our estimate’s expected value pretty close to the real value, our estimate has a high probability of being close to the real value.”

- In other words, if the true frequency is 9, we want the distribution of our estimate to kinda sorta look like this:
How Close is Close?

• In some applications, we might be okay overshooting by a larger amount (e.g. roughly estimating which restaurants people are visiting).

• In others, it’s really bad if we overestimate by too much (e.g. polling for an election).

• **Idea:** Allow the client of the estimator to pick some value ε between 0 and 1 indicating how close they want to be to the true value. The closer ε is to 0, the better the approximation we want.
How Close is Close?

- Our overestimate is related to $\|a\|_1$.
- We’ll formalize how $\varepsilon$ works as follows: we’ll say we’re okay with any estimate that’s within $\varepsilon \|a\|_1$ of the true value.
- This is okay for high-frequency elements, but not so great for low-frequency elements. (Why?)
- But that’s okay. In practice, we are most interested in finding the high-frequency items.
Making Things Formal

- We know that
  \[ \mathbb{E} [ \hat{a}_i - a_i ] \leq \frac{\|a\|_1}{w} \]
- We want to bound this quantity:
  \[ \Pr [ \hat{a}_i - a_i > \varepsilon \|a\|_1 ] \]
- Let’s run the numbers!
Pr \left[ \hat{a}_i - a_i > \varepsilon \|a\|_1 \right] 

\leq \frac{\mathbb{E} \left[ \hat{a}_i - a_i \right]}{\varepsilon \|a\|_1}

We don’t know the exact distribution of this random variable.

However, we have a one-sided error: our estimate can never be lower than the true value. This means that \( \hat{a}_i - a_i \geq 0 \).

Markov’s inequality says that if \( X \) is a nonnegative random variable, then

\[ \Pr[ X \geq c ] \leq \frac{\mathbb{E}[X]}{c}. \]
\[
\begin{align*}
\text{Pr} [ \hat{a}_i - a_i > \varepsilon \|a\|_1 ] & \\
\leq & \frac{\mathbb{E} [ \hat{a}_i - a_i ]}{\varepsilon \|a\|_1} \\
\leq & \frac{\|a\|_1}{w} \cdot \frac{1}{\varepsilon \|a\|_1}
\end{align*}
\]
\[
\Pr[\hat{a}_i - a_i > \varepsilon \|a\|_1] \\
\leq \frac{\mathbb{E}[\hat{a}_i - a_i]}{\varepsilon \|a\|_1} \\
\leq \frac{\|a\|_1}{w} \cdot \frac{1}{\varepsilon \|a\|_1} \\
= \frac{1}{\varepsilon w}
\]
Interpreting this Result

• Here’s what we just proved:

\[ \Pr \left[ \hat{a}_i - a_i > \varepsilon \|a\|_1 \right] \leq \frac{1}{\varepsilon \ w} \]

• What does this tell us?
  
  • Increasing \( w \) decreases the chance of an overestimate. Decreasing \( w \) increases the chance of an overestimate.

  • As the user decreases \( \varepsilon \), we have to proportionally increase \( w \) for this bound to tell us anything useful.

• Idea: Choose \( w = e \cdot \varepsilon^{-1} \).

  • The choice of \( e \) is “somewhat” arbitrary in that any constant will work – but I peeked ahead and there’s a good reason to choose \( e \) here.
Interpreting this Result

• Here’s what we just proved:

\[ \Pr \left( \hat{a}_i - a_i > \varepsilon \|a\|_1 \right) \leq e^{-1} \]

• What does this tell us?
  
  • Increasing \( w \) decreases the chance of an overestimate. Decreasing \( w \) increases the chance of an overestimate.
  
  • As the user decreases \( \varepsilon \), we have to proportionally increase \( w \) for this bound to tell us anything useful.

• **Idea:** Choose \( w = e \cdot \varepsilon^{-1} \).
  
  • The choice of \( e \) is “somewhat” arbitrary in that any constant will work – but I peeked ahead and there’s a good reason to choose \( e \) here.
The user chooses a value \( \varepsilon \in (0, 1) \). We pick \( w = e \cdot \varepsilon^{-1} \).

- Create an array `count` of \( w \) counters, each initially zero.
- Choose, from a family of 2-independent hash functions \( \mathcal{H} \), a uniformly-random hash function \( h : U \rightarrow [w] \).
- To `increment` \((x)\), increment `count[\(h(x)\)]`.
- To `estimate` \((x)\), return `count[\(h(x)\)]`.
- With probability at least \( 1 - \frac{1}{e} \), the estimate for the frequency of item \( x_i \) is within \( \varepsilon \cdot \|a\|_1 \) of the true frequency.

\[
w = O(\varepsilon^{-1}) \text{ counters}
\]
## How to Build an Estimator

<table>
<thead>
<tr>
<th>Step One: Build a Simple Estimator</th>
<th>Hash items to counters; add +1 when item seen.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step Two: Compute Expected Value of Estimator</td>
<td>Sum of indicators; 2-independent hashes have low collision rate.</td>
</tr>
<tr>
<td>Step Three: Apply Concentration Inequality</td>
<td>One-sided error; use expected value and Markov’s inequality.</td>
</tr>
<tr>
<td>Step Four: Boost Confidence</td>
<td>Count-Min Sketch</td>
</tr>
</tbody>
</table>
The Story So Far

• We now have a simple estimator where

\[ \Pr \left[ \hat{a}_i - a_i > \varepsilon \|a\|_1 \right] \leq e^{-1} \]

• This means we have a decent chance of getting an estimate we’re happy with.

• **Problem:** We probably want to be more confident than this.
  
  • In some applications, maybe it’s okay to have a 63% success rate.
  
  • In others (say, election polling) we’ll need to be a lot more confident than this.

• **Question:** How do you define “confident enough”?
The Parameter $\delta$

- The user already can select a parameter $\varepsilon$ tuning the *accuracy* of the estimator: how close we want to be to the true value.

- Let’s have them also select a parameter $\delta$ tuning the *confidence* of the estimator: how likely it is that we achieve this goal.

- $\delta$ ranges from 0 to 1. Lower $\delta$ means a higher chance of getting a good estimate.
Our Goal

• Right now, we have this statement:
  \[ \Pr [\hat{a}_i - a_i > \varepsilon \|a\|_1] \leq e^{-1} \]

• We want to get to this one:
  \[ \Pr [\hat{a}_i - a_i > \varepsilon \|a\|_1] \leq \delta \]

• How might we achieve this?
A Key Technique
It’s super unlikely that you’ll miss the center of the target every single time!
Running in Parallel

• Let’s run \( d \) copies of our data structure in parallel with one another.

• Each row has its hash function sampled uniformly at random from our hash family.

• Each time we \textit{increment} an item, we perform the corresponding \textit{increment} operation on each row.

\[
w = \lceil e \cdot \varepsilon^{-1} \rceil
\]

\[
d = ??
\]

\[
\begin{array}{cccccccc}
  h_1 & 31 & 41 & 59 & 26 & 53 & \ldots & 58 \\
  h_2 & 27 & 18 & 28 & 18 & 28 & \ldots & 45 \\
  h_3 & 16 & 18 & 3 & 39 & 88 & \ldots & 75 \\
  \ldots & & & & & \ldots & & \\
  h_d & 69 & 31 & 47 & 18 & 5 & \ldots & 59 \\
\end{array}
\]
Running in Parallel

- Imagine we call \textit{estimate}(x) on each of our estimators and get back these estimates.
- We need to give back a single number.
- \textbf{Question:} How should we aggregate these numbers into a single estimate?

\begin{align*}
\text{Estimator 1:} & \quad 137 \\
\text{Estimator 2:} & \quad 271 \\
\text{Estimator 3:} & \quad 166 \\
\text{Estimator 4:} & \quad 103 \\
\text{Estimator 5:} & \quad 261 
\end{align*}
Running in Parallel

- Imagine we call \textit{estimate}(x) on each of our estimators and get back these estimates.
- We need to give back a single number.
- \textbf{Question:} How should we aggregate these numbers into a single estimate?

\begin{itemize}
  \item \textit{Estimator 1:} 137
  \item \textit{Estimator 2:} 271
  \item \textit{Estimator 3:} 166
  \item \textit{Estimator 4:} 103
  \item \textit{Estimator 5:} 261
\end{itemize}

\textbf{Intuition:} The smallest estimate returned has the least “noise,” and that’s the best guess for the frequency.
Pr \left[ \min \{ \hat{a}_{ij} \} - a_i > \varepsilon \|a\|_1 \right]

The only way the minimum estimate is inaccurate is if \textit{every} estimate is inaccurate.

Let \( \hat{a}_{ij} \) be the estimate from the \( j \)th copy of the data structure.

Our final estimate is \( \min \{ \hat{a}_{ij} \} \).
\[
\Pr \left[ \min \{ \hat{a}_{ij} \} - a_i > \varepsilon \|a\|_1 \right] = \Pr \left[ \bigwedge_{j=1}^d \left( \hat{a}_{ij} - a_i > \varepsilon \|a\|_1 \right) \right]
\]

The only way the minimum estimate is inaccurate is if every estimate is inaccurate.

Let \(\hat{a}_{ij}\) be the estimate from the \(j\)th copy of the data structure.

Our final estimate is \(\min \{ \hat{a}_{ij} \}\).
\[
\Pr \left[ \min \{ \hat{a}_{ij} \} - a_i > \varepsilon \|a\|_1 \right]
= \Pr \left[ \bigwedge_{j=1}^{d} \left( \hat{a}_{ij} - a_i > \varepsilon \|a\|_1 \right) \right]
= \prod_{j=1}^{d} \Pr \left[ \hat{a}_{ij} - a_i > \varepsilon \|a\|_1 \right]
\]

Each copy of the data structure is independent of the others.

Let \( \hat{a}_{ij} \) be the estimate from the \( j \)th copy of the data structure.

Our final estimate is \( \min \{ \hat{a}_{ij} \} \)
\[
\Pr \left[ \min \{ \hat{a}_{ij} \} - a_i > \varepsilon \|a\|_1 \right] = \Pr \left[ \bigwedge_{j=1}^{d} \left( \hat{a}_{ij} - a_i > \varepsilon \|a\|_1 \right) \right] = \prod_{j=1}^{d} \Pr \left[ \hat{a}_{ij} - a_i > \varepsilon \|a\|_1 \right] \leq \prod_{j=1}^{d} e^{-1}
\]

Let \( \hat{a}_{ij} \) be the estimate from the \( j \)th copy of the data structure.

Our final estimate is \( \min \{ \hat{a}_{ij} \} \).
Let $\hat{a}_{ij}$ be the estimate from the $j$th copy of the data structure.

Our final estimate is $\min \{ \hat{a}_{ij} \}$
Finishing Touches

• We now see that
  \[ \Pr \left[ \hat{a}_i - a_i > \varepsilon \|a\|_1 \right] \leq e^{-d} \]

• We want to reach this goal:
  \[ \Pr \left[ \hat{a}_i - a_i > \varepsilon \|a\|_1 \right] \leq \delta \]

• So set \( d = \ln \delta^{-1} \).
The Count-Min Sketch

\[ \text{increment}(x): \]
\[ \quad \text{for } i = 1 \ldots d: \]
\[ \quad \text{count}[i][h_i(x)]++ \]

<table>
<thead>
<tr>
<th>( h_1 )</th>
<th>32</th>
<th>41</th>
<th>59</th>
<th>26</th>
<th>53</th>
<th>...</th>
<th>58</th>
</tr>
</thead>
<tbody>
<tr>
<td>( h_2 )</td>
<td>27</td>
<td>18</td>
<td>28</td>
<td>19</td>
<td>28</td>
<td>...</td>
<td>45</td>
</tr>
<tr>
<td>( h_3 )</td>
<td>16</td>
<td>19</td>
<td>3</td>
<td>39</td>
<td>88</td>
<td>...</td>
<td>75</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( h_d )</td>
<td>69</td>
<td>31</td>
<td>47</td>
<td>18</td>
<td>5</td>
<td>...</td>
<td>60</td>
</tr>
</tbody>
</table>
The Count-Min Sketch

increment(x):
for i = 1 ... d:
    count[i][h_i(x)]++

estimate(x):
    result = \infty
    for i = 1 ... d:
        result = \min(result, count[i][h_i(x)])
    return result
The Count-Min Sketch

- Update and query times are $\Theta(\log \delta^{-1})$.
  - That’s the number of replicated copies, and we do $O(1)$ work at each.

- Space usage: $\Theta(\varepsilon^{-1} \cdot \log \delta^{-1})$ counters.
  - Each individual estimator has $\Theta(\varepsilon^{-1})$ counters, and we run $\Theta(\log \delta^{-1})$ copies in parallel.

- Provides an estimate to within $\varepsilon \| a \|_1$ with probability at least $1 - \delta$.

- This can be *significantly* better than just storing a raw frequency count – especially if your goal is to find items that appear very frequently.
# How to Build an Estimator

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
<th>Count-Min Sketch</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Step One:</strong> Build a Simple Estimator</td>
<td>Hash items to counters; add +1 when item seen.</td>
<td></td>
</tr>
<tr>
<td><strong>Step Two:</strong> Compute Expected Value of Estimator</td>
<td>Sum of indicators; 2-independent hashes have low collision rate.</td>
<td></td>
</tr>
<tr>
<td><strong>Step Three:</strong> Apply Concentration Inequality</td>
<td>One-sided error; use expected value and Markov’s inequality.</td>
<td></td>
</tr>
<tr>
<td><strong>Step Four:</strong> Replicate to Boost Confidence</td>
<td>Take min; only fails if all estimates are bad.</td>
<td></td>
</tr>
</tbody>
</table>
Major Ideas From Today

- **2-independent hash families** are useful when we want to keep collisions low.
- A “good” approximation of some quantity should have tunable *confidence* and *accuracy* parameters.
- **Sums of indicator variables** are useful for deriving expected values of estimators.
- **Concentration inequalities** like *Markov’s inequality* are useful for showing estimators don’t stay too much from their expected values.
- Good estimators can be built from *multiple parallel copies* of weaker estimators.
Next Time

- **Count Sketches**
  - An alternative frequency estimator with different time/space bounds.

- **Cardinality Estimation**
  - Estimating how many different items you’ve seen in a data stream.