Hashing and Sketching

Part One

The Magic of Hash Functions

- Last week, we explored how to reduce the number of bits used by a data structure.
- In many cases, there are hard limits on how space-efficient any deterministic algorithm can be, but *randomized* algorithms can use shockingly few bits.
- This week explores how to use hash functions to seemingly achieve the impossible, as long as we’re okay with answers that are *approximately* correct *most* of the time.

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 ***universe*** (typically denoted $\mathscr{U}$) to some codomain.
- The codomain is usually a set of the form

$$\mathbf{[}\boldsymbol{m}\mathbf{]} = \{0, 1, 2, 3, \dots, m - 1\}$$

$$h : \mathscr{U} \rightarrow [m]$$

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 $\mathscr{H}$ 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 $\mathscr{H}$.
- ***Key Point:*** The randomness in our data structures almost always derives from the random choice of hash functions, not from the data.

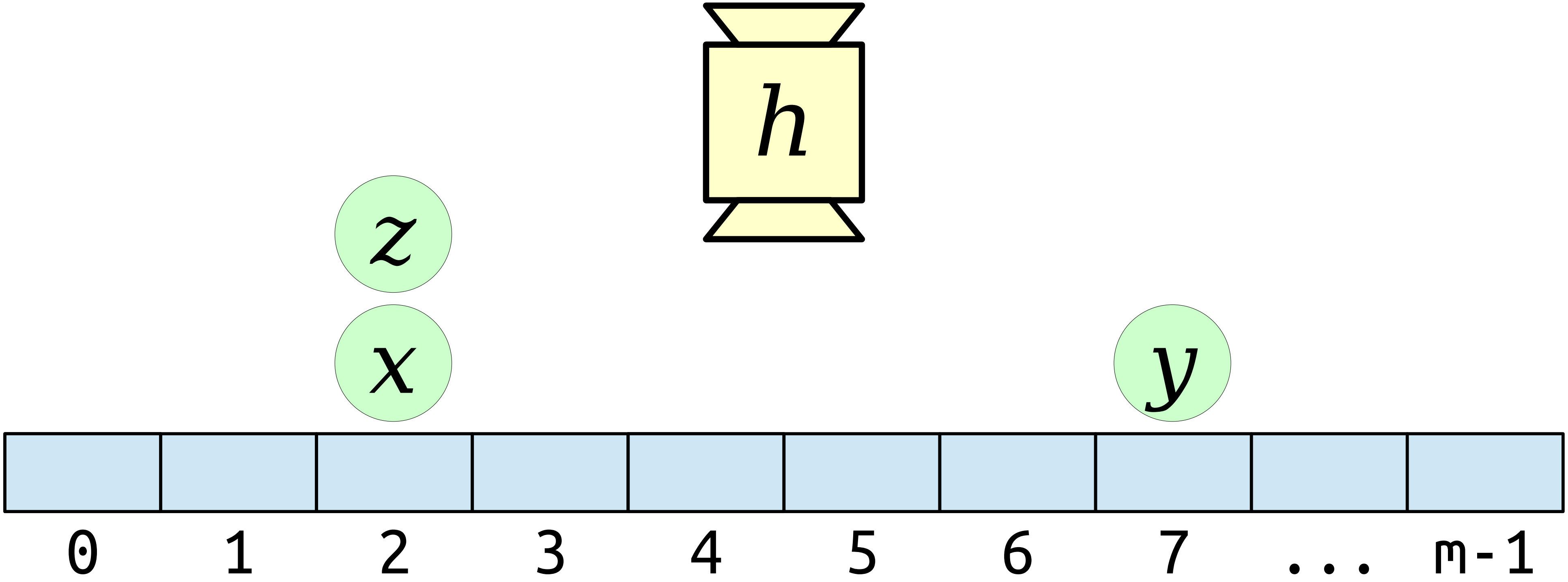
Data is adversarial.
Hash function selection is random.

- ***Question:*** What makes a family of hash functions $\mathscr{H}$ a “good” family of hash functions?

Goal: If we pick $h \in \mathscr{H}$ 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?



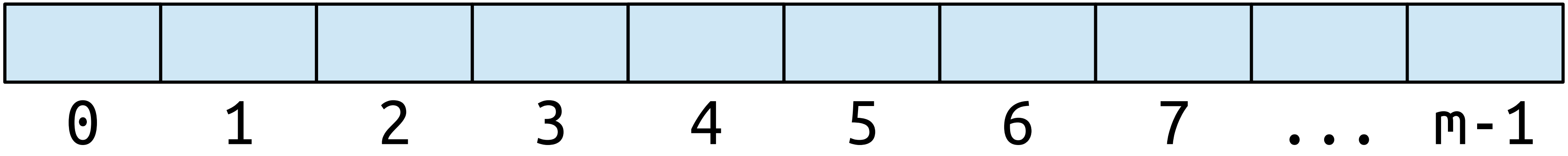
Distribution Property:
Each element should have an equal probability of being placed in each slot.

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

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

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

A family of hash functions $\mathscr{H}$ is called ***2-independent*** (or ***pairwise independent***) if it satisfies the distribution and independence properties.

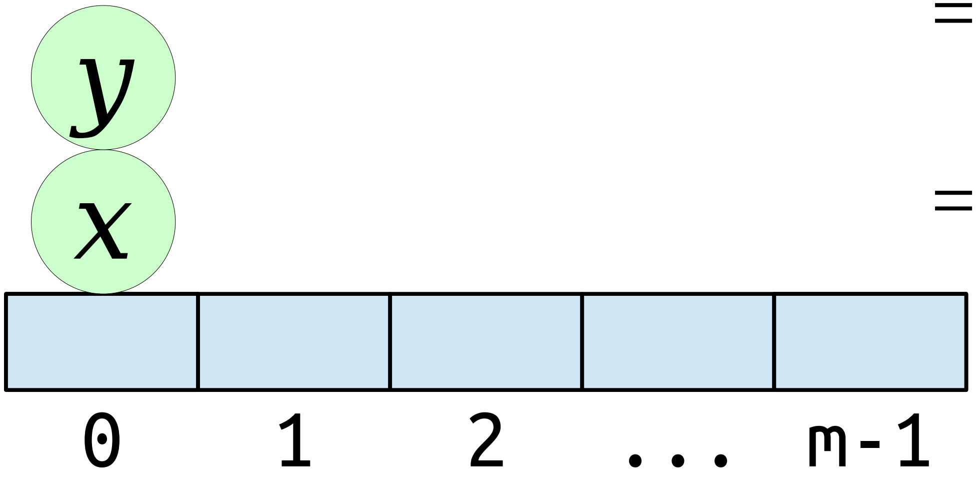


For any $x \in \mathscr{U}$ and random $h \in \mathcal{H}$, the value of $h(x)$ is uniform over its codomain.

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

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

$$
\begin{aligned}
& \Pr[h(x) = h(y)] \\
= & \sum_{i=0}^{m-1} \Pr[h(x) = i \wedge h(y) = i] \\
= & \sum_{i=0}^{m-1} \Pr[h(x) = i] \cdot \Pr[h(y) = i] \\
= & \sum_{i=0}^{m-1} \frac{1}{m^2} \\
= & \frac{1}{m}
\end{aligned}
$$



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***.

Time-Out for Announcements!

Problem Set 2

- Problem Set 1 was due at 1:00PM today.
  - Need more time? You can use up to two late days to submit either 24 or 48 hours late.
- Problem Set 2 (***Succinct Data Structures***) goes out today. It’s due next Thursday at 1:00PM.
  - Dive deeper into succinct rank and select.
  - Probe the limits of how far we can compress data structures.
  - Apply the techniques you’ve learned!
- As always, stop by office hours or ping us on Ed if you have questions!

Back to CS166!

Frequency Estimation

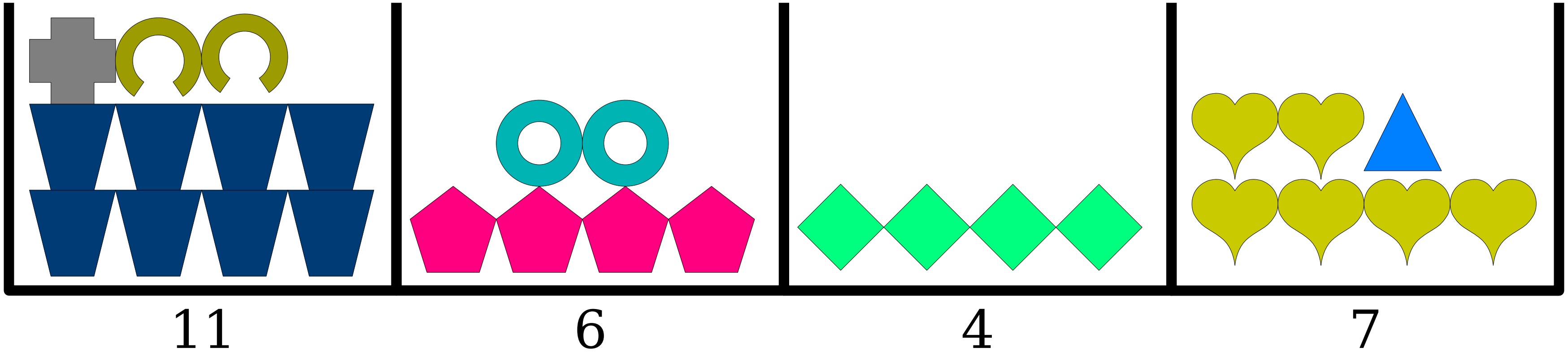
Frequency Estimators

- 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.
- This is easy to solve exactly using BSTs or hash tables, except that we need $\Omega(n)$ space simply to write down everything we've ***increment***ed.
- ***Question:*** Can we solve this problem without using $\Omega(n)$ bits of space?

The Count-Min Sketch

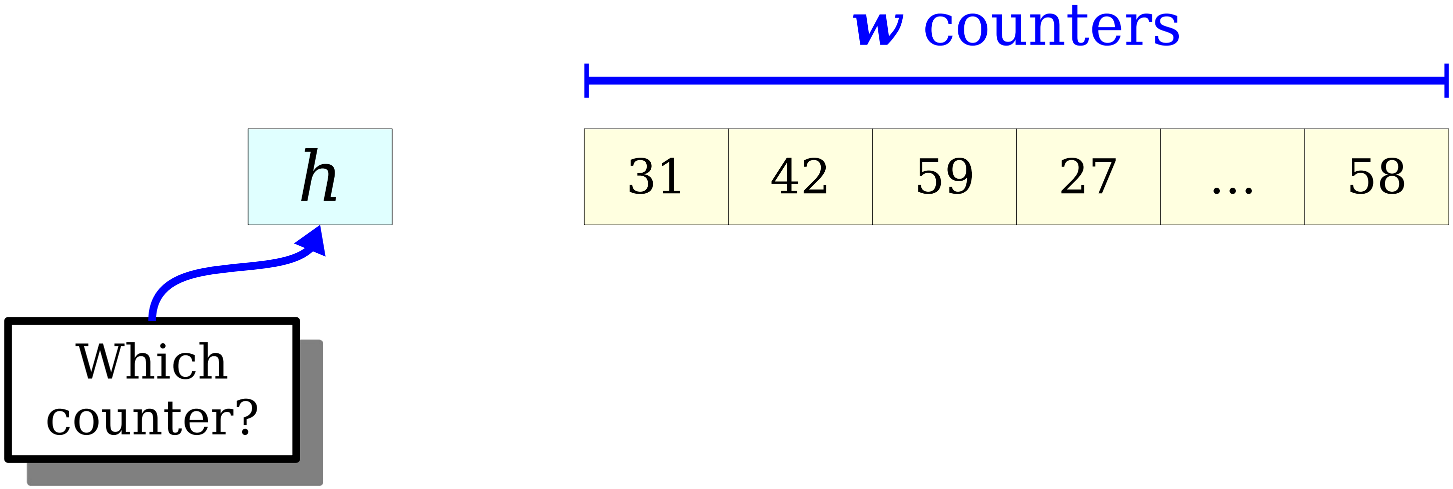
Revisiting the Exact Solution

- 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 $x \in \mathscr{U}$. Multiple objects might be assigned to the same counter.
- To ***increment***(x), increment the counter for x.
- To ***estimate***(x), read the value of the counter for x.



Our Initial Structure

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



31	42	59	27	…	58

Which counter?

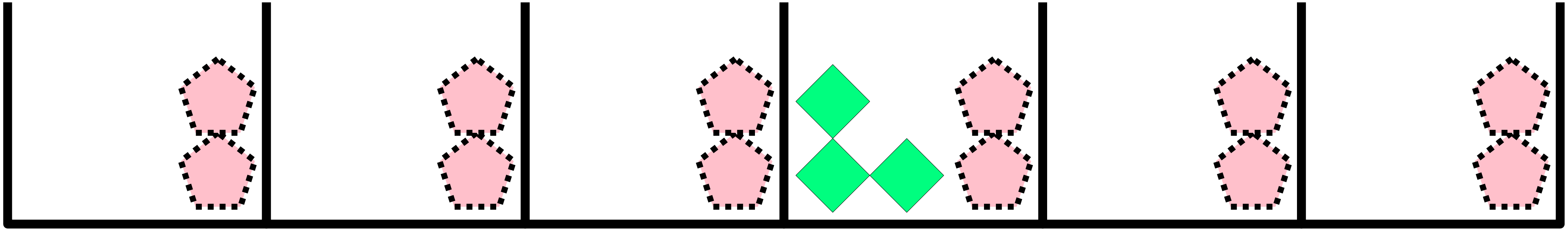
Some Notation

- Let x_1, x_2, x_3, … denote the list of distinct items whose frequencies are being stored.
- Let $\boldsymbol{a}_1$, $\boldsymbol{a}_2$, $\boldsymbol{a}_3$, … denote the frequencies of those items.
    - e.g. $\boldsymbol{a}_i$ is the true number of times x_i is seen.
- Let $\hat{\boldsymbol{a}}_1$, $\hat{\boldsymbol{a}}_2$, $\hat{\boldsymbol{a}}_3$, … denote the estimate our data structure gives for the frequency of each item.
    - e.g. $\hat{\boldsymbol{a}}_i$ is our estimate for how many times x_i has been seen.
    - ***Important detail:*** the $\boldsymbol{a}_i$ values are not random variables (data are chosen adversarially), while the $\hat{\boldsymbol{a}}_i$ 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 $\hat{\boldsymbol{a}}_i$.

Analyzing our Estimator

- We're interested in learning more about $\hat{\boldsymbol{a}}_i$. A good first step is to work out E[$\hat{\boldsymbol{a}}_i$].
- $\hat{\boldsymbol{a}}_i$ will be equal to $\boldsymbol{a}_i$, plus some "noise" terms from colliding elements.
- Each of those elements is very unlikely to collide with us, though. (There's a $^1/_w$ chance of a collision for any one other element.)
- ***Reasonable guess:*** $\mathrm{E}[\hat{\boldsymbol{a}}_i] \;=\; \boldsymbol{a}_i + \sum_{j \neq i} \frac{\boldsymbol{a}_j}{w}$

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 $\boldsymbol{a}_j$ to **count**$[h(x_i)]$.
  - If $h(x_i) \neq h(x_j)$, then x_j contributes 0 to **count**$[h(x_i)]$.
- To pin this down precisely, let's define a set of random variables as follows:

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

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

$$\hat{\boldsymbol{a}}_i - \boldsymbol{a}_i = \sum_{j \neq i} \boldsymbol{a}_j \mathbb{1}_{h(x_i)=h(x_j)}$$

$$\begin{aligned}
\mathrm{E}[\hat{\boldsymbol{a}}_i - \boldsymbol{a}_i] &= \mathrm{E}\Big[\sum_{j \neq i} \boldsymbol{a}_j \mathbb{1}_{h(x_i)=h(x_j)}\Big] \\
&= \sum_{j \neq i} \mathrm{E}[\boldsymbol{a}_j \mathbb{1}_{h(x_i)=h(x_j)}] \\
&= \sum_{j \neq i} \boldsymbol{a}_j \mathrm{E}[\mathbb{1}_{h(x_i)=h(x_j)}] \\
&= \sum_{j \neq i} \frac{\boldsymbol{a}_j}{w} \\
&\leq \frac{\|\boldsymbol{a}\|_1}{w}
\end{aligned}$$

Idea: Think of our element frequencies $\boldsymbol{a}_1$, $\boldsymbol{a}_2$, $\boldsymbol{a}_3$, … as a vector

$$\boldsymbol{a} = [\boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{a}_3, \dots]$$

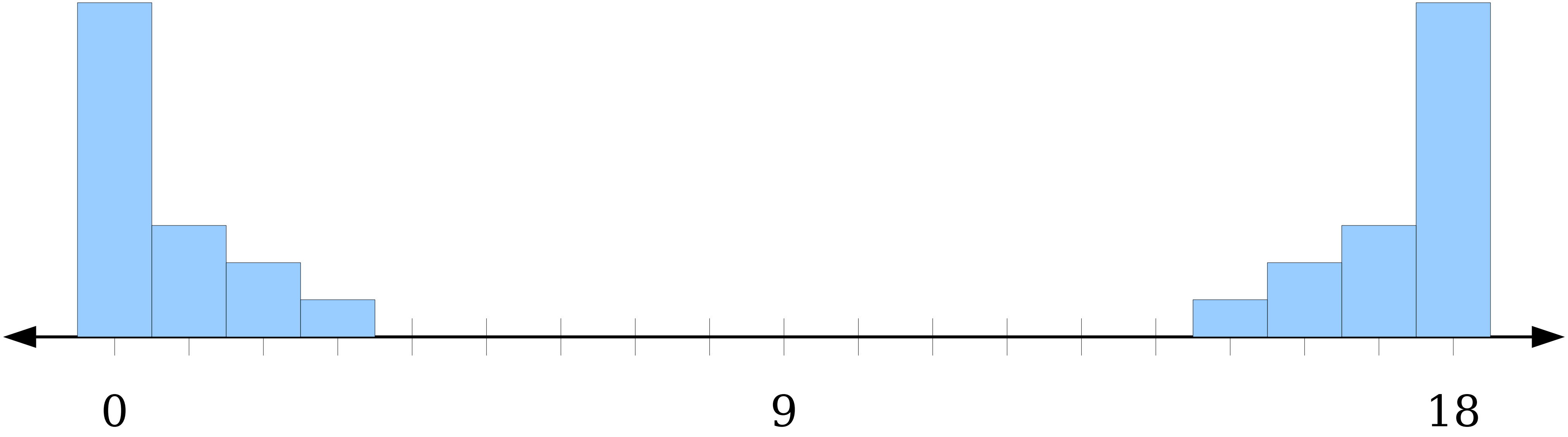
The total number of objects is the sum of the vector entries.

This is called the ***L_1 norm*** of $\boldsymbol{a}$, and is denoted $\|\boldsymbol{a}\|_1$:

$$\|\boldsymbol{a}\|_1 = \sum_i |\boldsymbol{a}_i|$$

On Expected Values

- We know that $E[\hat{\boldsymbol{a}}_i - \boldsymbol{a}_i] \leq ||\boldsymbol{a}||_1 / w$. 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 $\hat{\boldsymbol{a}}_i$, then our estimator is not very useful!

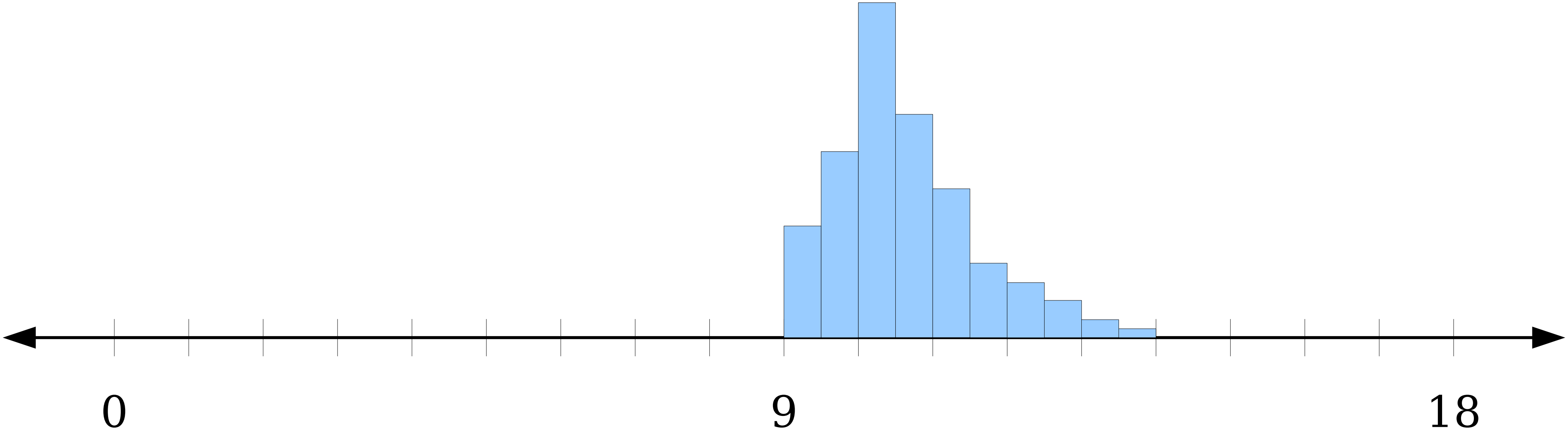


On Expected Values

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

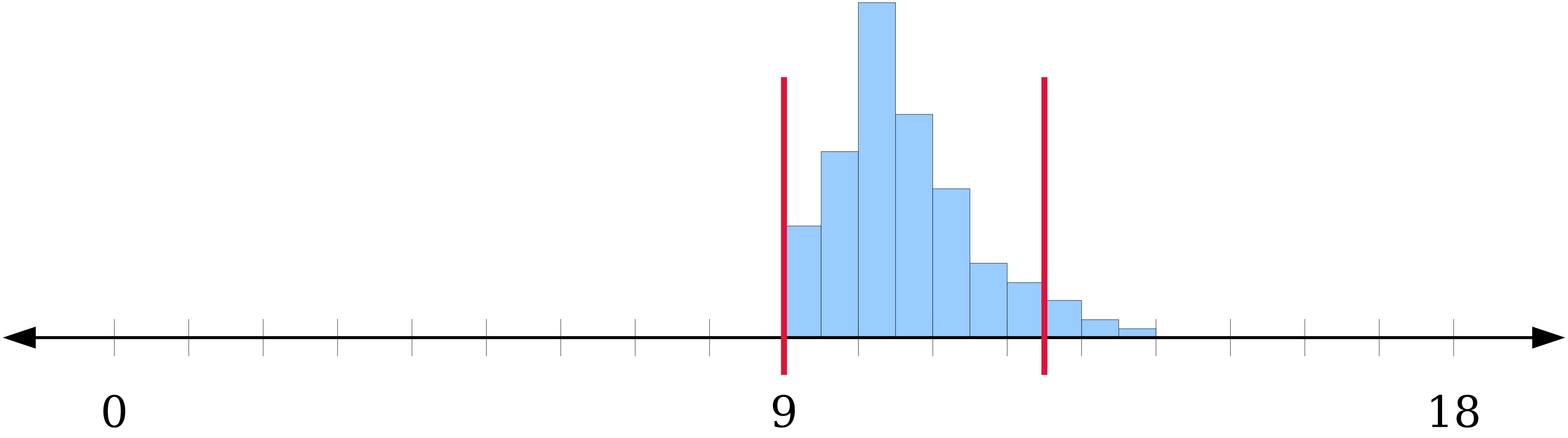
"Not only is our estimate's expected value pretty close to the real value, our estimate has a high probability of being close to the real value."

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



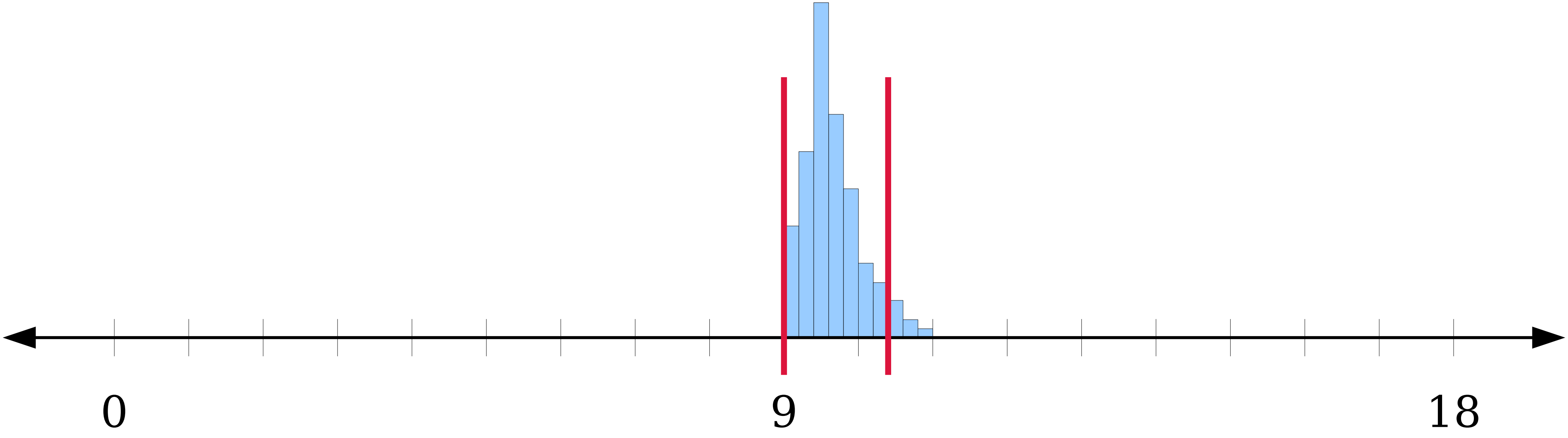
How Close is Close?

- 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.



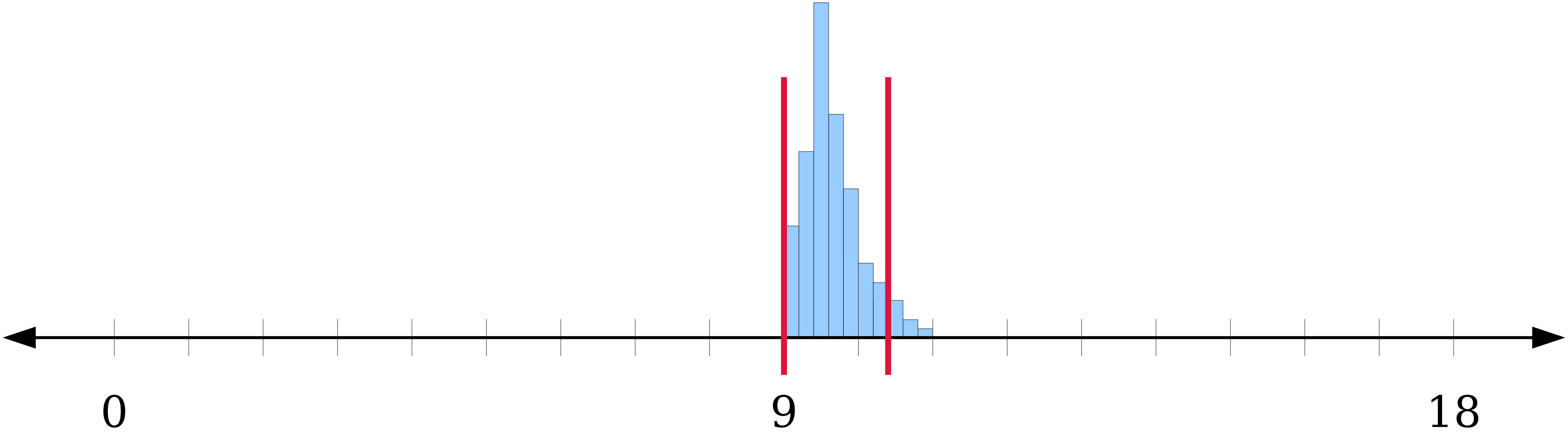
How Close is Close?

- 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 $||\boldsymbol{a}||_1$.
- We'll formalize how ε works as follows: we'll say we're okay with any estimate that's within $\varepsilon||\boldsymbol{a}||_1$ 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

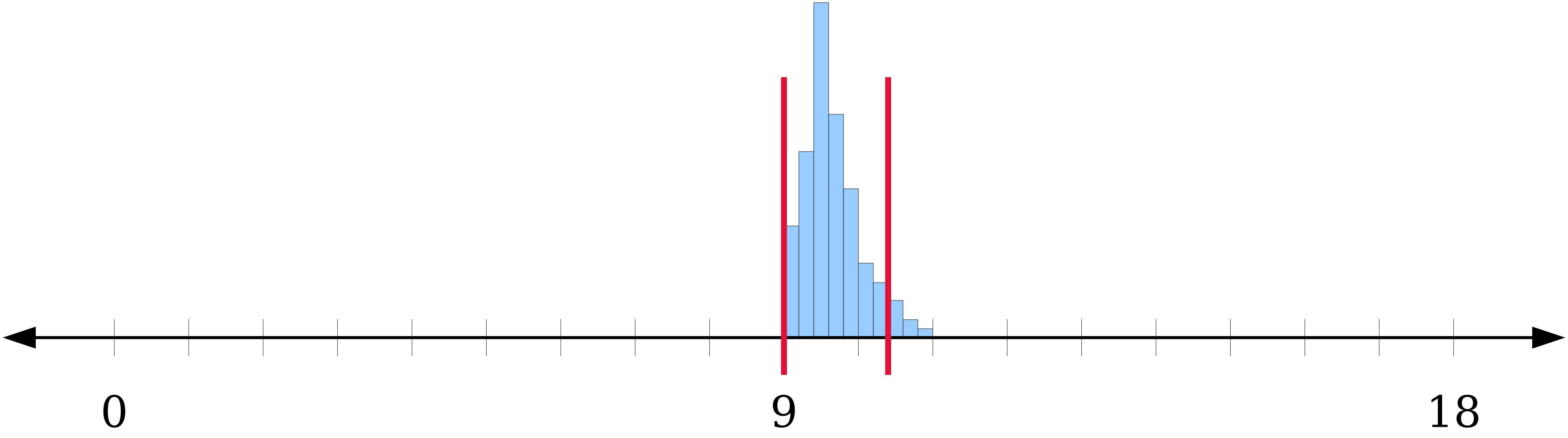
- We know that

$$\mathrm{E}[\hat{\boldsymbol{a}}_i - \boldsymbol{a}_i] \leq \frac{\|\boldsymbol{a}\|_1}{w}$$

- We want to bound this quantity:

$$\Pr[\hat{\boldsymbol{a}}_i - \boldsymbol{a}_i \ \boldsymbol{>}\ \varepsilon\|\boldsymbol{a}\|_1]$$

- Let’s run the numbers!



$$\Pr\left[\hat{\boldsymbol{a}}_i - \boldsymbol{a}_i \;>\; \varepsilon\left\|\boldsymbol{a}\right\|_1\right]$$

$$\leq \quad \frac{\mathrm{E}\left[\hat{\boldsymbol{a}}_i - \boldsymbol{a}_i\right]}{\varepsilon\left\|\boldsymbol{a}\right\|_1}$$

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 $\hat{\boldsymbol{a}}_i - \boldsymbol{a}_i \geq 0$.

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

$$\Pr[X \geq c] \;\leq\; \frac{\mathrm{E}[X]}{c}.$$

$$
\begin{aligned}
& \Pr\left[\hat{\boldsymbol{a}}_i - \boldsymbol{a}_i \;\; > \;\; \varepsilon \left\|\boldsymbol{a}\right\|_1\right] \\
\leq \quad & \frac{\mathrm{E}\left[\hat{\boldsymbol{a}}_i - \boldsymbol{a}_i\right]}{\varepsilon \left\|\boldsymbol{a}\right\|_1} \\
\leq \quad & \frac{\left\|\boldsymbol{a}\right\|_1}{w} \cdot \frac{1}{\varepsilon \left\|\boldsymbol{a}\right\|_1} \\
= \quad & \frac{1}{\varepsilon\, w}
\end{aligned}
$$

Interpreting this Result

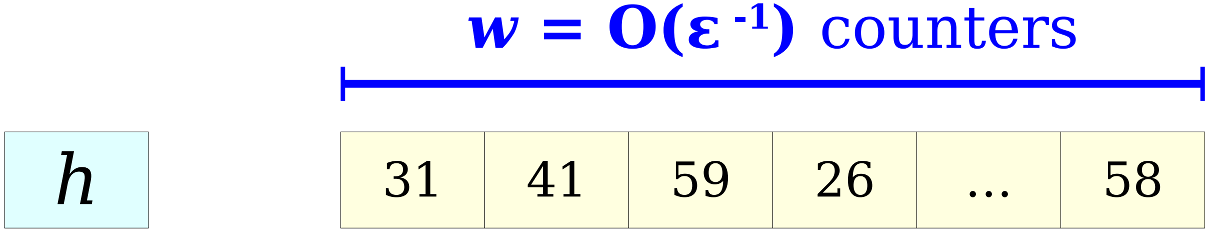
- Here's what we just proved:

$$\Pr\left[\hat{\boldsymbol{a}}_i - \boldsymbol{a}_i > \varepsilon \|\boldsymbol{a}\|_1\right] \le e^{-1}$$

- 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 ε, 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 Story So Far

- 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 $\mathscr{H}$, a uniformly-random hash function $h : \mathscr{U} \to [w]$.
- To ***increment***(x), increment **count**$[h(x)]$.
- To ***estimate***(x), return **count**$[h(x)]$.
- With probability at least $1 - {}^1/_e$, the estimate for the frequency of item x_i is within $\varepsilon \cdot ||\boldsymbol{a}||_1$ of the true frequency.



$\boldsymbol{w = O(\varepsilon^{-1})}$ counters

h

31	41	59	26	…	58

The Story So Far

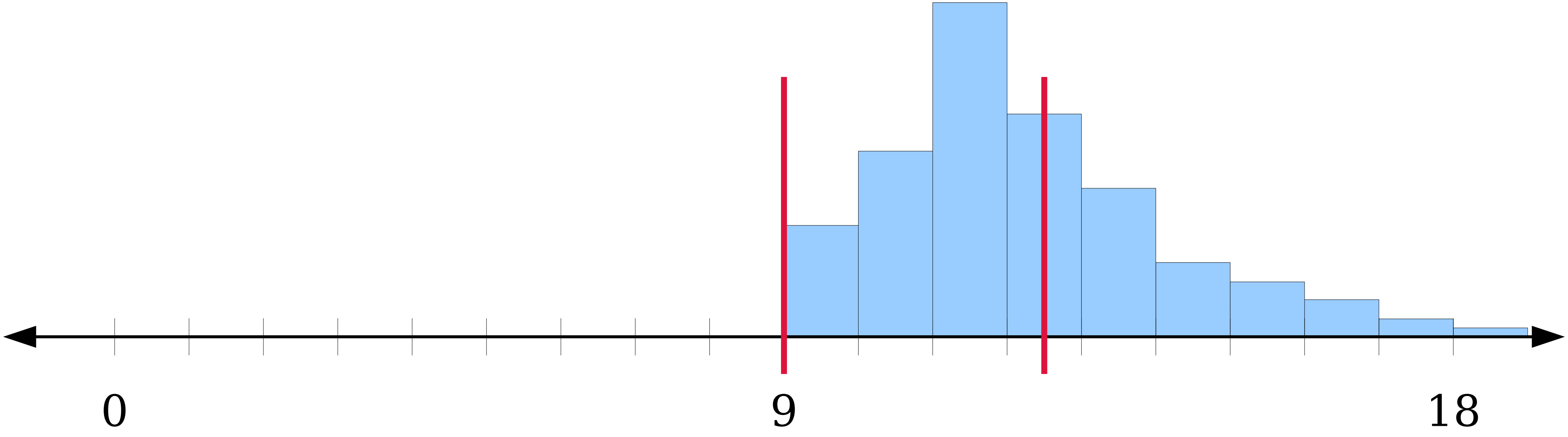
- We now have a simple estimator where

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

- 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 δ

- The user already can select a parameter ε tuning the ***accuracy*** of the estimator: how close we want to be to the true value.
- Let’s have them also select a parameter δ tuning the ***confidence*** of the estimator: how likely it is that we achieve this goal.
- δ ranges from 0 to 1. Lower δ means a higher chance of getting a good estimate.



Our Goal

- Right now, we have this statement:

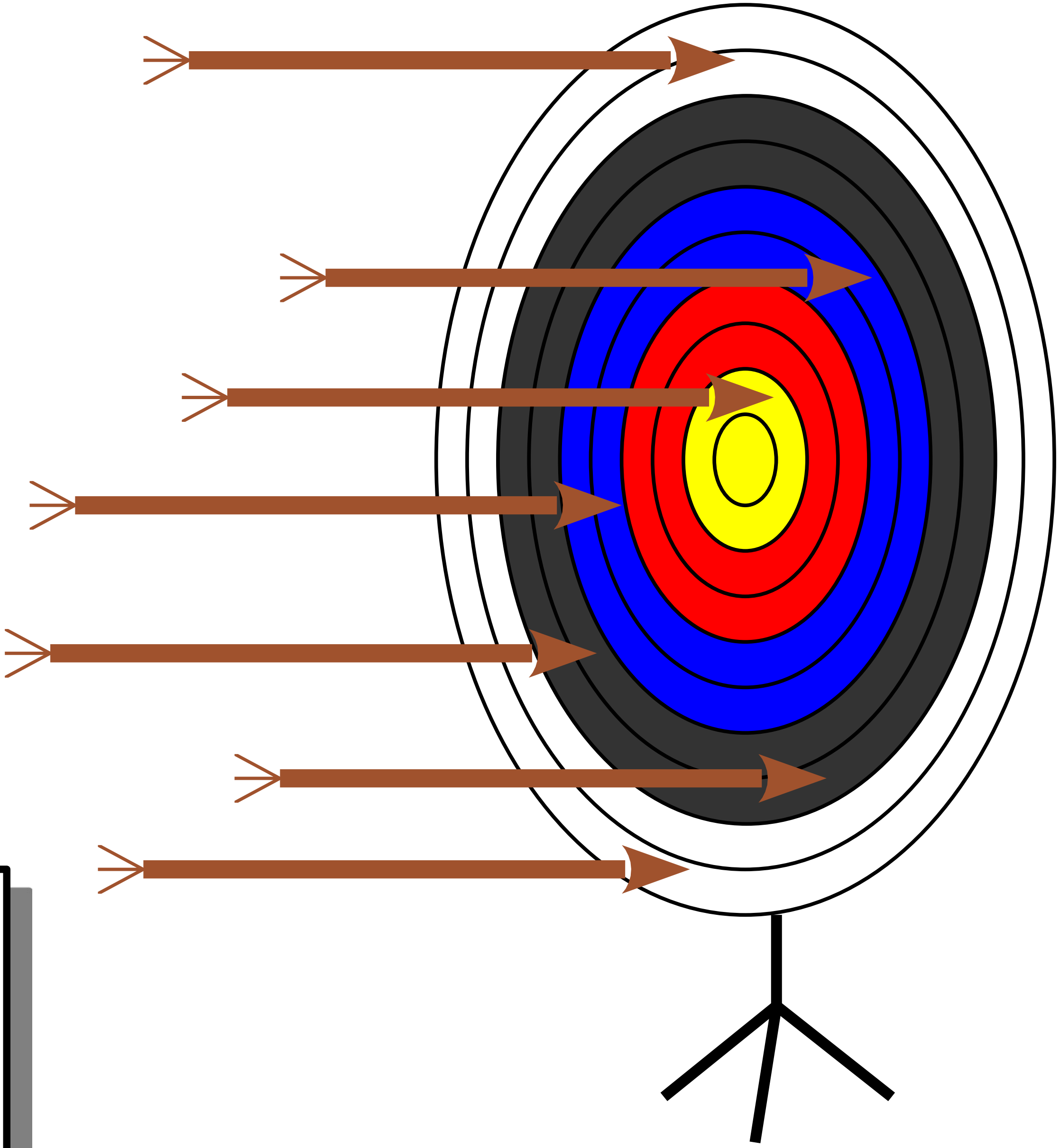
$$\Pr\left[\hat{\boldsymbol{a}}_i - \boldsymbol{a}_i \;\boldsymbol{>}\; \varepsilon\,\|\boldsymbol{a}\|_1\right] \quad \leq \quad e^{-1}$$

- We want to get to this one:

$$\Pr\left[\hat{\boldsymbol{a}}_i - \boldsymbol{a}_i \;\boldsymbol{>}\; \varepsilon\,\|\boldsymbol{a}\|_1\right] \quad \leq \quad \delta$$

- How might we achieve this?

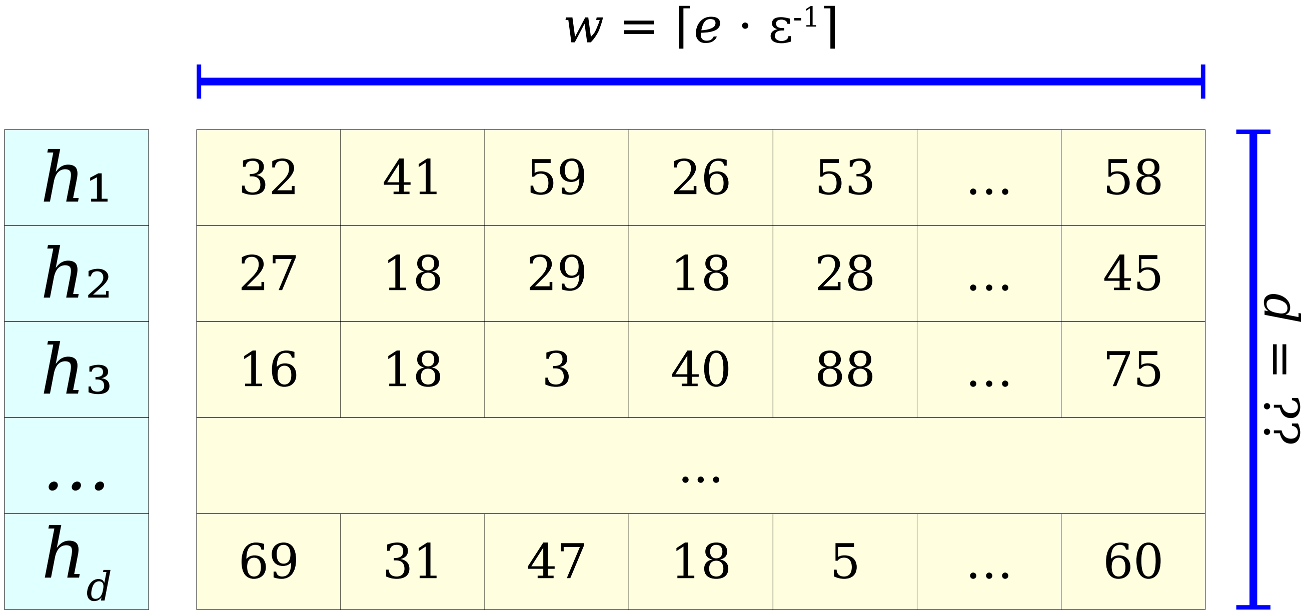
A Key Technique



It’s *super unlikely* that *every* shot will miss the center of the target!

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 ***increment*** an item, we perform the corresponding ***increment*** operation on each row.



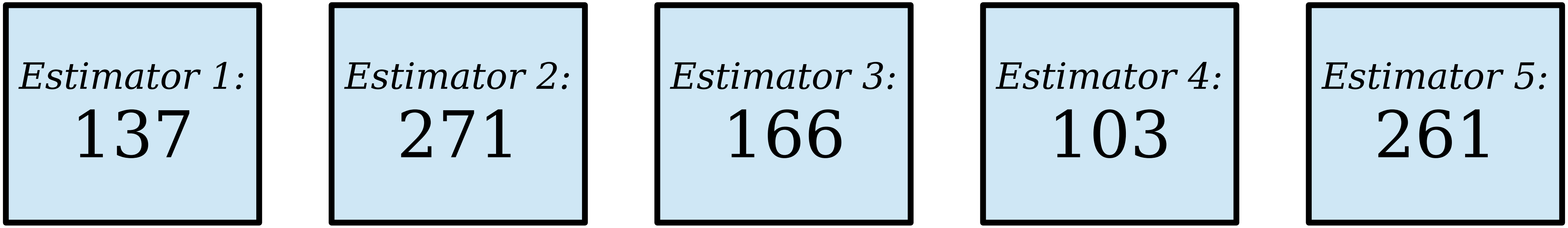
$w = \lceil e \cdot \varepsilon^{-1} \rceil$

h_1	32	41	59	26	53	…	58
h_2	27	18	29	18	28	…	45
h_3	16	18	3	40	88	…	75
…	…						
h_d	69	31	47	18	5	…	60

$d = ??$

Running in Parallel

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



Estimator 1: 137

Estimator 2: 271

Estimator 3: 166

Estimator 4: 103

Estimator 5: 261

Running in Parallel

- Imagine we call ***estimate***(x) on each of our estimators and get back these estimates.
- We need to give back a single number.
- ***Question:*** How should we aggr into a single estimate?

Intuition: The smallest estimate returned has the least “noise,” and that’s the best guess for the frequency.

Estimator 1: 137

Estimator 2: 271

Estimator 3: 166

Estimator 4: 103

Estimator 5: 261

$$
\begin{aligned}
& \Pr\left[\min\left\{\hat{\boldsymbol{a}}_{ij}\right\} - \boldsymbol{a}_i \;>\; \varepsilon \left\|\boldsymbol{a}\right\|_1\right] \\
= \;& \Pr\left[\bigwedge_{j=1}^{d} \left(\hat{\boldsymbol{a}}_{ij} - \boldsymbol{a}_i \;>\; \varepsilon \left\|\boldsymbol{a}\right\|_1\right)\right] \\
= \;& \prod_{j=1}^{d} \Pr\left[\hat{\boldsymbol{a}}_{ij} - \boldsymbol{a}_i \;>\; \varepsilon \left\|\boldsymbol{a}\right\|_1\right] \\
\leq \;& \prod_{j=1}^{d} e^{-1} \\
= \;& e^{-d}
\end{aligned}
$$

Let $\hat{\boldsymbol{a}}_{ij}$ be the estimate from the jth copy of the data structure.

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

Finishing Touches

- We now see that

$$\Pr[\hat{\boldsymbol{a}}_i - \boldsymbol{a}_i \;>\; \varepsilon \|\boldsymbol{a}\|_1] \quad \leq \quad e^{-d}$$

- We want to reach this goal:

$$\Pr[\hat{\boldsymbol{a}}_i - \boldsymbol{a}_i \;>\; \varepsilon \|\boldsymbol{a}\|_1] \quad \leq \quad \delta$$

- So set $\boldsymbol{d = \ln \delta^{-1}}$.

The Count-Min Sketch

h_1	32	41	59	26	53	…	58
h_2	27	18	28	19	28	…	45
h_3	16	19	3	39	88	…	75
…	…						
h_d	69	31	47	18	5	…	60

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

```
estimate(x):
   result = ∞
   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.
  - How many bits do you use per counter? Depends on the particulars of your problem.
- Provides an estimate to within $\varepsilon \|\boldsymbol{a}\|_1$ 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

	Count-Min Sketch
Step One: Build a Simple Estimator	Hash items to counters; add +1 when item seen.
Step Two: Compute Expected Value of Estimator	Sum of indicators; 2-independent hashes have low collision rate.
Step Three: Apply Concentration Inequality	One-sided error; use expected value and Markov's inequality.
Step Four: Replicate to Boost Confidence	Take min; only fails if all estimates are bad.

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.