L2: Approximate Algorithms

Wei Wang @ HKUST(GZ)

March 25, 2025

Outline

1. Counting Items: Morris Counter
2. Sketching: Count-Min Sketch
   - Problem Definition: Frequency Estimators
   - Preliminary: Hash Function
   - Preliminary: What does it mean for an approximation to be "good"?
   - Count-Min Sketch

Counting Items

How many items have we read so far?

To count up to t elements **exactly**, $\log t$ bits are **necessary**

Morris's counter: Count approximately using $\log \log t$ bits

Can count up to 1 billion with $\log \log 10^9 = 5$ bits

Approximate counting, v1

Init: $c \leftarrow 0$

Update:

draw a random number $x \in [0, 1]$

if $(x \leq 1/2)$

$c \leftarrow c + 1$

Query: return $2c$

$E[2c] = t, \quad \sigma = \sqrt{t}/2$ **Why? Proof?**

Space $\log(t/2) = \log t - 1 \Rightarrow$ we saved 1 bit!

Any problem?

Approximate counting, v1

Init: $c \leftarrow 0$

Update:

draw a random number $x \in [0, 1]$

if $(x \leq 1/2)$

$c \leftarrow c + 1$

Query: return $2c$

$E[2c] = t, \quad \sigma = \sqrt{t}/2$ **Why? Proof?**

Space $\log(t/2) = \log t - 1 \Rightarrow$ we saved 1 bit!

Any problem?

Proof: Let the returned value be r. Let I_i be the indicator variable when the i-item comes. Then $E[r] = E[\sum I_i \times 1] = \sum E[I_i] = t/2$.

Approximate counting, v2

Init: $c \leftarrow 0$

Update:

  draw a random number $x \in [0, 1]$

  if $(x \leq 2^{-k})$

    $c \leftarrow c + 1$

Query: return $2^k c$

$E[c] = t/2^k, \quad \sigma \simeq \sqrt{t/2^k}$

Memory $\log t - k \Rightarrow$ we saved k bits!

$x \leq 2^{-k}$: AND of k random bits, $\log k$ memory

Morris' counter [Morris77]

Init: $c \leftarrow 0$

Update:

draw a random number $x \in [0, 1]$

if $(x \leq 2^{-c})$ $c \leftarrow c + 1$

Query: return $2^c - 1$

$E[c] \simeq \log t, \quad E[2^c - 1] = t, \quad \sigma \simeq t/\sqrt{2}$

Memory = bits used to hold $c = \log c = \log \log t$ bits

Proof

The complete proof is fairly lengthy. The gist is to show $E[2^c] = t + 1$

Morris' approximate counter

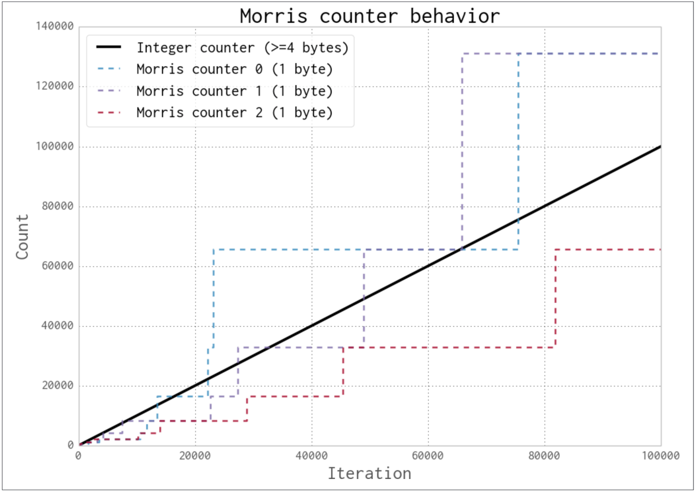


Figure 11-3. Three 1-byte Morris counters vs. an 8-byte integer

Figure: From *High Performance Python*, M. Gorelick & I. Oswald. O'Reilly 2014

Problem: large variance, $\sigma \approx 0.7t$

Reducing the variance, method I

- Run r parallel, independent copies of the algorithm (**boosting**)
- On Query, average their estimates
- $E[\text{Query}] \approx t, \quad \sigma \approx t/\sqrt{2r}$
- Space $r \log\log t$ bits
- Time per item multiplied by r

Reducing the variance, method II

Use basis $b < 2$ instead of basis 2:

- Places t in the series $1, b, b^2, \ldots, b^i, \ldots$ ("resolution" b)
- $E[b^c] \approx t, \quad \sigma \approx \sqrt{(b-1)/2} \cdot t$
- Space $\log \log t - \log \log b$ bits $\quad (> \log \log t,$ because $b < 2)$
- For $b = 1.08$, 3 extra bits, $\sigma \approx 0.2t$

Count-Min Sketches

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

[Cormode-Muthukrishnan 04]

Like Space Saving:

- Provides an approximation f_x^r to f_x, for every x
- Can be used (less directly) to find θ-heavy hitters
- Uses memory $O(1/\theta)$

Unlike Space Saving:

- It is randomized - hash functions instead of counters
- Supports additions **and deletions**
- Can be used as basis for several other queries

What is the meaning of f_x^r and f_x?

How to Build an Estimator

1. Design a simple data structure that, intuitively, gives you a good estimate.
2. Use a **sum of indicator variables** and **linearity of expectation** to prove that, on expectation, the data structure is pretty close to correct.
3. Use a **concentration inequality** to show that, with decent probability, the data structure's output is close to its expectation.
4. Run multiple copies of the data structure in parallel to amplify the success probability.

Outline

- **Hash Functions**
  - Understanding our basic building blocks.
- **Quality of Approximation**
- **Count-Min Sketches**
  - Estimating how many times we've seen something.
- **Concentration Inequalities**
  - "Correct on expectation" versus "correct with high probability."
- **Probability Amplification**
  - Increasing our confidence in our answers.

Preliminaries: Hash Functions

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

$$h : \mathcal{U} \to [m]$$

- **Intuition:** No matter how clever you are with designing a 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?

The Principles of Hash Functions

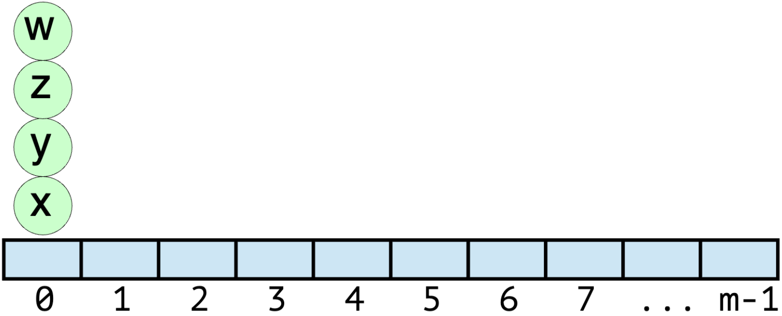
- **Goal:** If we pick $h \in \mathcal{H}$ uniformly at random, then h should distribute elements uniformly randomly.
- **Problem:** A hash function that distributes n elements uniformly at random over $[m]$ requires $\Omega(n \log m)$ space in the worst case.
- **Question:** Do we actually need true randomness? Or can we get away with something weaker?

The Principles of Hash Functions

- **Distribution Property: Each element should have an equal probability of being placed in each slot.**
- For any $x \in \mathcal{U}$ and random $h \in \mathcal{H}$, the value of $h(x)$ is uniform over $[m]$.
- Some "obviously bad" hash functions obey this rule. How is this possible?

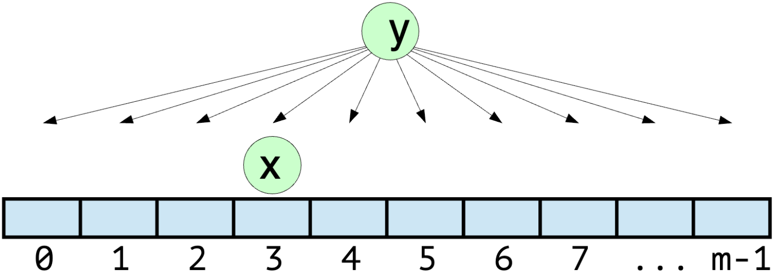
The Principles of Hash Functions

- **Distribution Property: Each element should have an equal probability of being placed in each slot.**
- For any $x \in \mathcal{U}$ and random $h \in \mathcal{H}$, the value of $h(x)$ is uniform over $[m]$.
- Some "obviously bad" hash functions obey this rule. How is this possible?



The Principles of Hash Functions

- **Distribution Property: Each element should have an equal probability of being placed in each slot.**
- For any $x \in \mathcal{U}$ and random $h \in \mathcal{H}$, the value of $h(x)$ is uniform over $[m]$.
- Problem: This rule doesn't guarantee that elements are spread out.



The Principles of Hash Functions

- **Distribution Property: Each element should have an equal probability of being placed in each slot.**
  - For any $x \in \mathcal{U}$ and random $h \in \mathcal{H}$, the value of $h(x)$ is uniform over $[m]$.
- **Independence Property: Where one element is placed shouldn't impact where a second goes.**
  - For any distinct $x, y \in \mathcal{U}$ and random $h \in \mathcal{H}$, $h(x)$ and $h(y)$ are independent random variables.

The Principles of Hash Functions

- **Distribution Property: Each element should have an equal probability of being placed in each slot.**
  - For any $x \in \mathcal{U}$ and random $h \in \mathcal{H}$, the value of $h(x)$ is uniform over $[m]$.
- **Independence Property: Where one element is placed shouldn't impact where a second goes.**
  - 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.

The Principles of Hash Functions

- For any $x \in \mathcal{U}$ and random $h \in \mathcal{H}$, the value of $h(x)$ is uniform over $[m]$.
- For any distinct $x, y \in \mathcal{U}$ 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.**
- Proof:

- For any $x \in \mathcal{U}$ and random $h \in \mathcal{H}$, the value of $h(x)$ is uniform over $[m]$.
- For any distinct $x, y \in \mathcal{U}$ 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.**
- Proof:

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

The Principles of Hash Functions

- For any $x \in \mathcal{U}$ and random $h \in \mathcal{H}$, the value of $h(x)$ is uniform over $[m]$.
- For any distinct $x, y \in \mathcal{U}$ 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.**
- Proof:

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

- For any $x \in \mathcal{U}$ and random $h \in \mathcal{H}$, the value of $h(x)$ is uniform over $[m]$.
- For any distinct $x, y \in \mathcal{U}$ 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.**
- Proof:

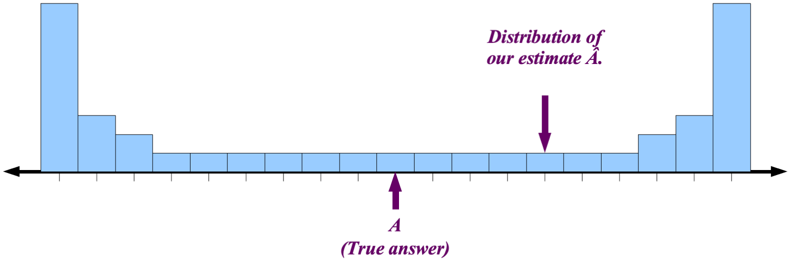
$$\begin{aligned}
\Pr[h(x) = h(y)] &= \sum_{i=0}^{m-1} \Pr[h(x) = i \wedge h(y) = i] \\
&= \sum_{i=0}^{m-1} \Pr[h(x) = i] \cdot \Pr[h(y) = i] \\
&= \sum_{i=0}^{m-1} \frac{1}{m^2} = \frac{1}{m}
\end{aligned}$$

Outline

- **Hash Functions**
  - Understanding our basic building blocks.
- **Quality of Approximation**
- **Count-Min Sketches**
  - Estimating how many times we've seen something.
- **Concentration Inequalities**
  - "Correct on expectation" versus "correct with high probability."
- **Probability Amplification**
  - Increasing our confidence in our answers.

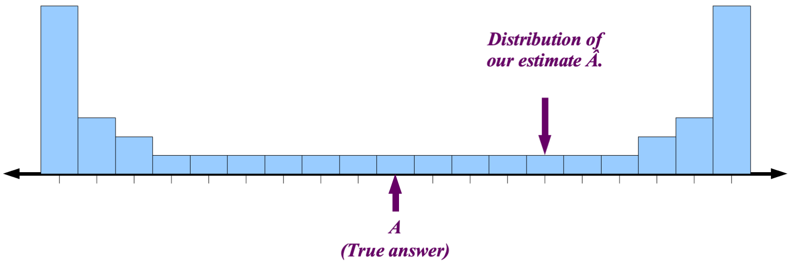
What does it mean for an approximation to be "good"?

- Let A be the true answer.
- Let $\hat{A}$ be a random variable denoting our estimate.
- This would not make for a good estimate. However, we have $\mathbb{E}[\hat{A}] = A$



What does it mean for an approximation to be "good"?

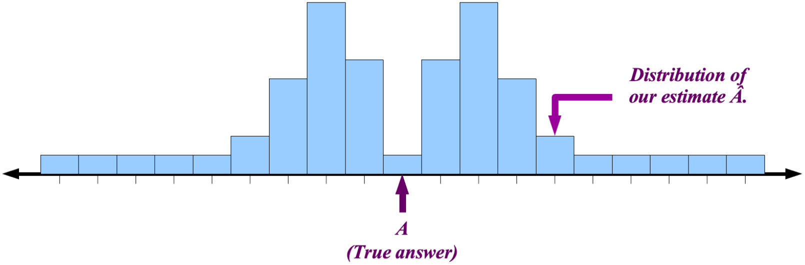
- Let A be the true answer.
- Let $\hat{A}$ be a random variable denoting our estimate.
- This would not make for a good estimate. However, we have $\mathbb{E}[\hat{A}] = A$



Observation 1: **Being correct in expectation isn't sufficient.**

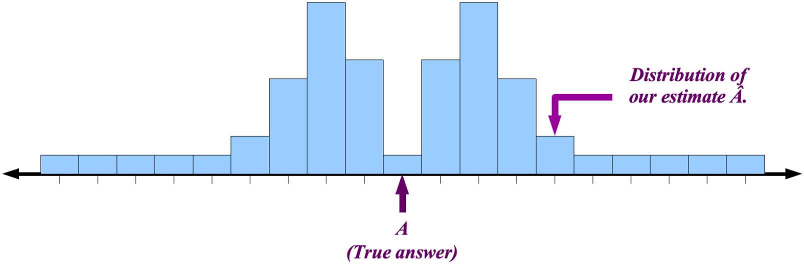
What does it mean for an approximation to be “good”?

- Let A be the true answer.
- Let $\hat{A}$ be a random variable denoting our estimate.
- It’s unlikely that we’ll get the right answer, but we’re probably going to be close.



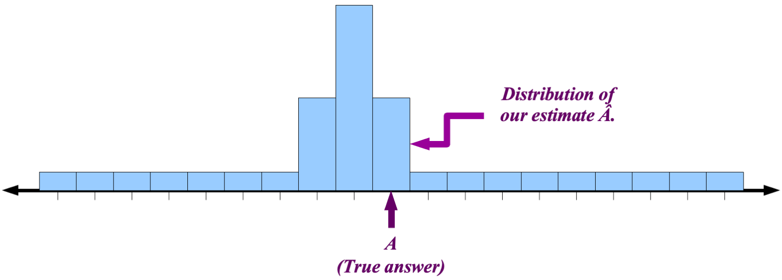
What does it mean for an approximation to be "good"?

- Let A be the true answer.
- Let $\hat{A}$ be a random variable denoting our estimate.
- It's unlikely that we'll get the right answer, but we're probably going to be close.

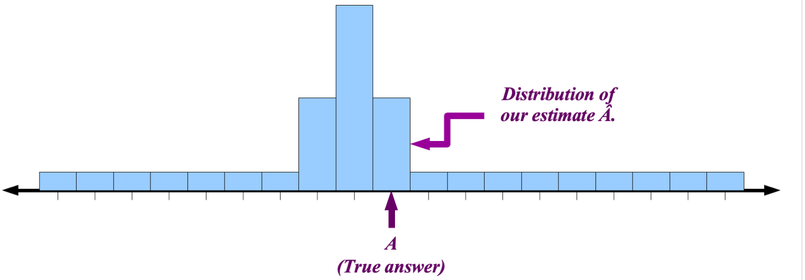


Observation 2: **The difference $|\hat{A} - A|$ between our estimate and the truth should ideally be small.**

- Let A be the true answer.
- Let $\hat{A}$ be a random variable denoting our estimate.
- This estimate skews low, but it's very close to the true value.



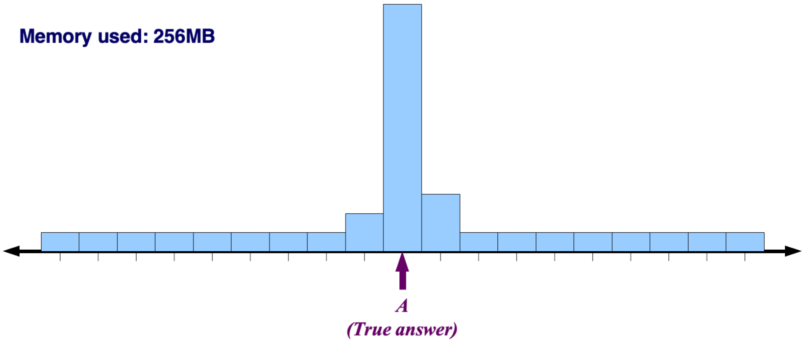
- Let A be the true answer.
- Let $\hat{A}$ be a random variable denoting our estimate.
- This estimate skews low, but it's very close to the true value.



Observation 3: **An estimate doesn't have to be unbiased to be useful.**

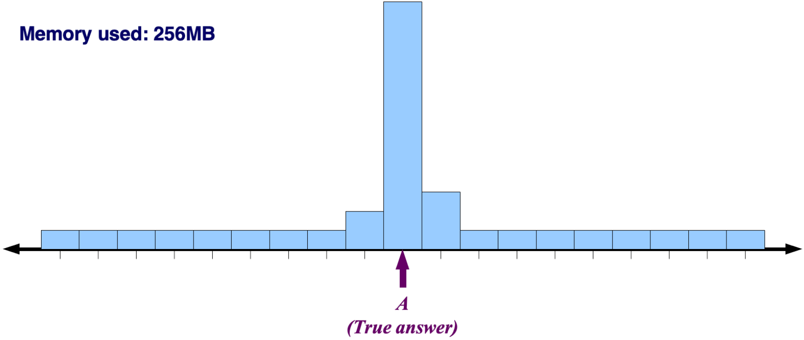
What does it mean for an approximation to be "good"?

- Let A be the true answer.
- Let $\hat{A}$ be a random variable denoting our estimate.
- The more resources we allocate, the better our estimate should be.



What does it mean for an approximation to be "good"?

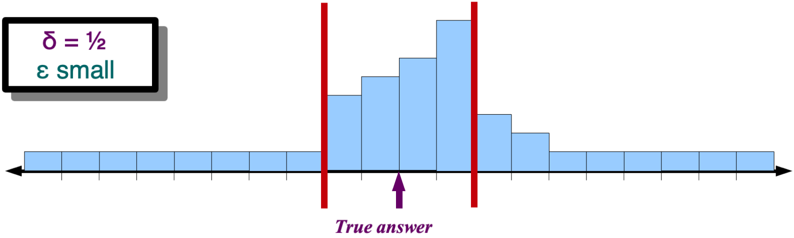
- Let A be the true answer.
- Let $\hat{A}$ be a random variable denoting our estimate.
- The more resources we allocate, the better our estimate should be.



Observation 4: **A good approximation should be tunable.**

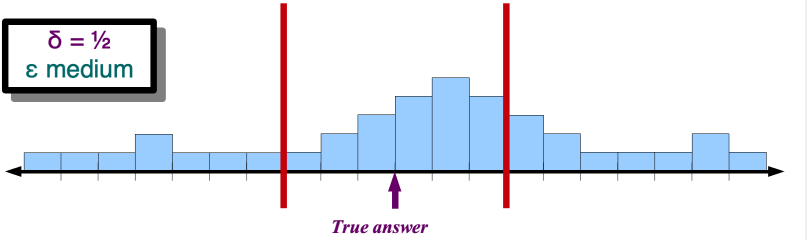
What does it mean for an approximation to be "good"?

- Suppose there are two tunable values:
  - $\varepsilon \in (0, 1]$ represents **accuracy**
  - $\delta \in (0, 1]$ represents **confidence**
- Goal: Make an estimator $\hat{A}$ for some quantity A where
  - With probability at least $1 - \delta$ (Probably)
  - $|\hat{A} - A| \leq \varepsilon \cdot \text{size}(\text{input})$ (Approximately Correct)
  - for some measure of the size of the input



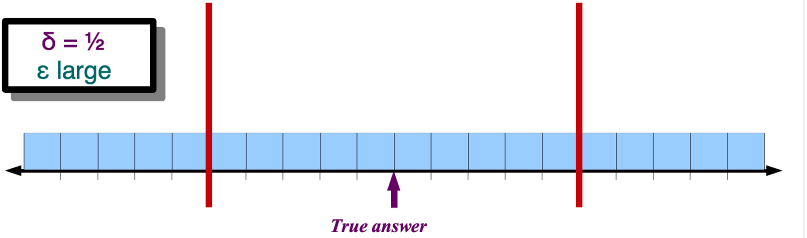
What does it mean for an approximation to be "good"?

- Suppose there are two tunable values:
  - $\varepsilon \in (0, 1]$ represents **accuracy**
  - $\delta \in (0, 1]$ represents **confidence**
- Goal: Make an estimator $\hat{A}$ for some quantity A where
  - With probability at least $1 - \delta$ (Probably)
  - $|\hat{A} - A| \leq \varepsilon \cdot \text{size(input)}$ (Approximately Correct)
  - for some measure of the size of the input



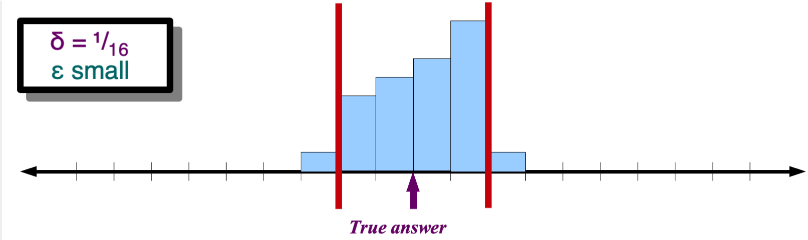
What does it mean for an approximation to be "good"?

- Suppose there are two tunable values:
  - $\varepsilon \in (0, 1]$ represents **accuracy**
  - $\delta \in (0, 1]$ represents **confidence**
- Goal: Make an estimator $\hat{A}$ for some quantity A where
  - With probability at least $1 - \delta$ (Probably)
  - $|\hat{A} - A| \leq \varepsilon \cdot \text{size(input)}$ (Approximately Correct)
  - for some measure of the size of the input



What does it mean for an approximation to be "good"?

- Suppose there are two tunable values:
  - $\varepsilon \in (0, 1]$ represents **accuracy**
  - $\delta \in (0, 1]$ represents **confidence**
- Goal: Make an estimator $\hat{A}$ for some quantity A where
  - With probability at least $1 - \delta$ (Probably)
  - $|\hat{A} - A| \leq \varepsilon \cdot \text{size(input)}$ (Approximately Correct)
  - for some measure of the size of the input



Outline

- **Hash Functions**
  - Understanding our basic building blocks.
- **Quality of Approximation**
- **Count-Min Sketches**
  - Estimating how many times we've seen something.
- **Concentration Inequalities**
  - "Correct on expectation" versus "correct with high probability."
- **Probability Amplification**
  - Increasing our confidence in our answers.

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 $x_i \in \mathcal{U}$. Multiple x_i's 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

- We can model "assigning each x_i to a counter" by using hash functions.
- Choose, from a family of 2-independent hash functions $\mathcal{H}$, a uniformly-random hash function $h : \mathcal{U} \to [w]$.
- Create an array `count` of w counters, each initially zero.
  - We'll choose w later on.
- To `increment`(x), increment `count`[h(x)]. To
- `estimate`(x), return `count`[h(x)].

For each $x_i \in \mathcal{U}$, let a_i denote the number of times we've seen x_i.

Similarly, let $\hat{a}_i$ denote our estimated value of the frequency of x_i.

Goal: Bound the probability that the error $(\hat{a}_i - a_i)$ is too high.

- Let's look at $\hat{a}_i = \texttt{count}[h(x_i)]$ for some choice of x_i. For
- each element x_j:
  - If $\mathrm{h}(x_i) = \mathrm{h}(x_j)$, then x_j contributes a_j to $\texttt{count}[\mathrm{h}(x_i)]$. If
  - $\mathrm{h}(x_i) \neq \mathrm{h}(x_j)$, then x_j contributes 0 to $\texttt{count}[\mathrm{h}(x_i)]$.
- To pin this down precisely, let's define a set of random variables $X_1, X_2, \ldots$, as follows:

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

Each of these variables are called an **indicator random variable**, since it "indicates" whether some event occurs.

Analyzing this Structure

- Let's look at $\hat{a}_i = \texttt{count}[h(x_i)]$ for some choice of h.
- For each element x_j:
  - If $h(x_i) = h(x_j)$, then x_j contributes a_j to $\texttt{count}[\mathrm{h}(x_i)]$.
  - If $h(x_i) \neq h(x_j)$, then x_j contributes 0 to $\texttt{count}[\mathrm{h}(x_i)]$.
- To pin this down precisely, let's define a set of random variables $X_1, X_2, \ldots,$ as follows:

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

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

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

Analyzing this Structure

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

Analyzing this Structure

$$\mathbb{E}[\hat{a}_i - a_i] = \mathbb{E}[\sum_{j \neq i} a_j X_j]$$
$$= \sum_{j \neq i} \mathbb{E}[a_j X_j] \text{ (linearity of expectation)}$$

Analyzing this Structure

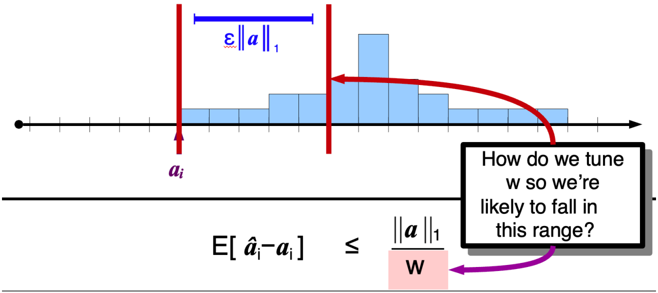
$$
\begin{aligned}
\mathbb{E}[\hat{a}_i - a_i] &= \mathbb{E}[\sum_{j \neq i} a_j X_j] \\
&= \sum_{j \neq i} \mathbb{E}[a_j X_j] \text{ (linearity of expectation)} \\
&= \sum_{j \neq i} a_j \mathbb{E}[X_j] \left(\begin{array}{c} \text{the randomness comes from} \\ \textbf{the choice of hash function} \end{array} \right)
\end{aligned}
$$

Analyzing this Structure

$$
\begin{aligned}
\mathbb{E}[\hat{a}_i - a_i] &= \mathbb{E}[\sum_{j \neq i} a_j X_j] \\
&= \sum_{j \neq i} \mathbb{E}[a_j X_j] \text{ (linearity of expectation)} \\
&= \sum_{j \neq i} a_j \mathbb{E}[X_j] \left(\begin{array}{c} \text{the randomness comes from} \\ \textbf{the choice of hash function} \end{array} \right) \\
&= \sum_{j \neq i} \frac{a_j}{w} \left(\mathbb{E}[X_j] = 1 \cdot \Pr[h(x_i) = h(x_j)] = \frac{1}{w} \right)
\end{aligned}
$$

Analyzing this Structure

$$\begin{aligned}
\mathbb{E}[\hat{a}_i - a_i] &= \mathbb{E}[\sum_{j\neq i} a_j X_j] \\
&= \sum_{j\neq i} \mathbb{E}[a_j X_j] \text{ (linearity of expectation)} \\
&= \sum_{j\neq i} a_j \mathbb{E}[X_j] \left(\begin{array}{c} \text{the randomness comes from} \\ \textbf{the choice of hash function} \end{array} \right) \\
&= \sum_{j\neq i} \frac{a_j}{w} \left(\mathbb{E}[X_j] = 1 \cdot \Pr[h(x_i) = h(x_j)] = \frac{1}{w} \right) \\
&\leq \frac{\|a\|_1}{w}
\end{aligned}$$



- 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 $\Pr[X \geq c] \leq \frac{\mathbb{E}[X]}{c}$

$$\Pr[\hat{a}_i - a_i > \varepsilon \|a\|_1] \leq \frac{\mathbb{E}[\hat{a}_i - a_i]}{\varepsilon \|a\|_1}$$

Analyzing this Structure

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

$$\begin{aligned}
\Pr[\hat{a}_i - a_i > \varepsilon\|a\|_1] &\leq \frac{\mathbb{E}[\hat{a}_i - a_i]}{\varepsilon\|a\|_1} \\
&\leq \frac{\|a\|_1}{w} \cdot \frac{1}{\varepsilon\|a\|_1} \\
&= \frac{1}{\varepsilon w}
\end{aligned}$$

Analyzing this Structure

- Goal: Make an estimator $\hat{A}$ for some quantity A where
  - With probability at least $1-\delta$ (Probably)
  - $|\hat{A} - A| \leq \varepsilon \cdot \text{size(input)}$ (Approximately Correct)
  - for some measure of the size of the input
- $\Pr[\hat{a}_i - a_i > \varepsilon \|a\|_1] \leq \frac{1}{\varepsilon w}$
- Initial Idea: Pick $w = \varepsilon^{-1}\delta^{-1}$. Then, $\Pr[\hat{a}_i - a_i > \varepsilon \|a\|_1] \leq \delta$

Question

Suppose we're counting 1,000 distinct items. If we want our estimate to be within $\varepsilon\|a\|_1$ of the true value with 99.9% probability, how much memory do we need?

Answer: The memory requirement is $1,000\varepsilon^{-1}$.

Can we do better?

Analyzing this Structure

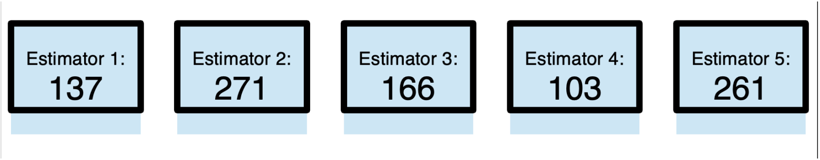
- Goal: Make an estimator $\hat{A}$ for some quantity A where
  - With probability at least $1-\delta$ (Probably)
  - $|\hat{A} - A| \leq \varepsilon \cdot \text{size(input)}$ (Approximately Correct)
  - for some measure of the size of the input
- $\Pr[\hat{a}_i - a_i > \varepsilon \|a\|_1] \leq \frac{1}{\varepsilon w}$
- Revised Idea: Pick $w = e\varepsilon^{-1}$. Then, $\Pr[\hat{a}_i - a_i > \varepsilon \|a\|_1] \leq e^{-1}$

Question

This simple data structure, by itself is likely to be wrong. What happens if we run several of its copies in parallel?

Running in Parallel

- Let's suppose that we run *d* independent copies of this data structure. Each has its own independently randomly chosen hash function.
- To `increment`(x) in the overall structure, we call `increment`(x) on each of the underlying data structures.
- The probability that at least one of them provides a good estimate is quite high.

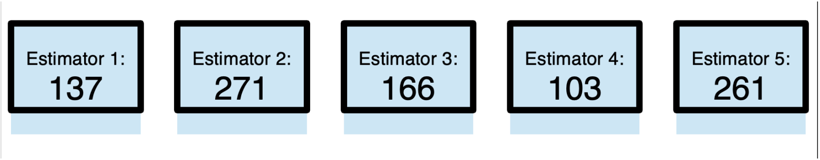


Question

How do you know which estimator is correct?

Running in Parallel

- Let's suppose that we run *d* independent copies of this data structure. Each has its own independently randomly chosen hash function.
- To `increment`(x) in the overall structure, we call `increment`(x) on each of the underlying data structures.
- The probability that at least one of them provides a good estimate is quite high.



Question

How do you know which estimator is correct?

- **The smallest estimate returned has the least 'noise' and that's the best guess for the frequency.**

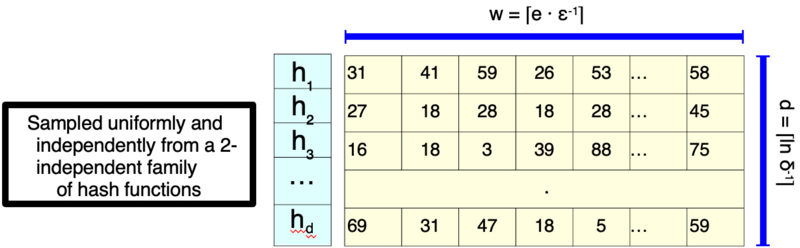
Running in Parallel: Error Bound

- Revised Idea: Pick $w = e\varepsilon^{-1}$. Then, $\Pr[\hat{a}_i - a_i > \varepsilon\|a\|_1] \leq e^{-1}$.
- Let $\hat{a}_{ij}$ be the estimate from the jth copy of the data structure.
- Our final estimate is defined as $\min\{\hat{a}_{ij}\}$.

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

Running in Parallel

- Goal: Make an estimator $\hat{A}$ for some quantity A where
  - With probability at least $1 - \delta$ (Probably)
  - $|\hat{A} - A| \le \varepsilon \cdot \text{size(input)}$ (Approximately Correct)
  - for some measure of the size of the input
- $\Pr[\min\{\hat{a}_{ij}\} - a_i > \varepsilon \|a\|_1] \le e^{-d}$
- Idea: Pick $d = -\ln \delta = \ln \delta^{-1}$. Then,
  $\Pr[\min\{\hat{a}_{ij}\} - a_i > \varepsilon \|a\|_1] \le \delta$



- Update and query times are $\Theta(d)$, which is $\Theta(\log \delta^{-1})$.
- Space usage: $\Theta(\varepsilon^{-1} \cdot \log \delta^{-1})$ counters.
  - This can be significantly better than just storing a raw frequency count!
- Provides an estimate to within $\varepsilon \|\mathbf{a}\|_1$ with probability at least $1 - \delta$.

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.**
- Randomization opens up new routes for tradeoffs in data structures:
  - Trade worst-case guarantees for average-case guarantees.
  - Trade exact answers for approximate answers.
- These data structures are used extensively in practice.