Big-Data Algorithms:
Counting Distinct Elements in a Stream

Reference: http://www.sketchingbigdata.org/fall17/lec/lec2.pdf

Problem Description

- Input: Given an integer n, along with a stream of integers $i_1, i_2, \ldots, i_m \in \{1, \ldots, n\}$.
- Output: The number of distinct integers in the stream.

So want to write a function `query()` that will return same.

Problem Description

- Input: Given an integer n, along with a stream of integers $i_1, i_2, \ldots, i_m \in \{1, \ldots, n\}$.
- Output: The number of distinct integers in the stream.

So want to write a function `query()` that will return same.

Trivial algorithms:

- Remember the whole stream!

Problem Description

- Input: Given an integer n, along with a stream of integers $i_1, i_2, \ldots, i_m \in \{1, \ldots, n\}$.
- Output: The number of distinct integers in the stream.

So want to write a function `query()` that will return same.

Trivial algorithms:

- Remember the whole stream!
  Cost? $\min\{m, n\}$ log n bits

Problem Description

- Input: Given an integer n, along with a stream of integers $i_1, i_2, \ldots, i_m \in \{1, \ldots, n\}$.
- Output: The number of distinct integers in the stream.

So want to write a function `query()` that will return same.

Trivial algorithms:

- Remember the whole stream!
  Cost? $\min\{m, n\} \log n$ bits
- Use a bit vector of length n.

Need $\Omega(n)$ bits of memory in worst case setting.

Need $\Omega(n)$ bits of memory in worst case setting.

Can be done using $\Theta(\min\{m \log n, n\})$ bits of memory if we abandon worst case setting.

Need $\Omega(n)$ bits of memory in worst case setting.

Can be done using $\Theta(\min\{m \log n, n\})$ bits of memory if we abandon worst case setting.

If A is exact answer, seek approximation $\tilde{A}$ such that

$$\mathbb{P}\left(|\tilde{A} - A| > \varepsilon \cdot A\right) < \delta,$$

where

- ε: approximation factor
- δ: failure probability

Universal Hashing

Motivation

We will give a short "nickname" to each of the 2^{32} possible IP addresses.

You can think of this short name as just a number between 1 and 250 (we will later adjust this range very slightly).

Thus many IP addresses will inevitably have the same nickname; however, we hope that most of the 250 IP addresses of our particular customers are assigned distinct names, and we will store their records in an array of size 250 indexed by these names.

What if there is more than one record associated with the same name?

Easy: each entry of the array points to a linked list containing all records with that name. So the total amount of storage is proportional to 250, the number of customers, and is independent of the total number of possible IP addresses.

Moreover, if not too many customer IP addresses are assigned the same name, lookup is fast, because the average size of the linked list we have to scan through is small.

Hash tables

How do we assign a short name to each IP address?

This is the role of a hash function: A function h that maps IP addresses to positions in a table of length about 250 (the expected number of data items).

The name assigned to an IP address x is thus $h(x)$, and the record for x is stored in position $h(x)$ of the table.

Each position of the table is in fact a *bucket*, a linked list that contains all current IP addresses that map to it.
Hopefully, there will be very few buckets that contain more than a handful of IP addresses.

How to choose a hash function?

In our example, one possible hash function would map an IP address to the 8-bit number that is its last segment:

$$h(128.32.168.80) = 80.$$

A table of $n = 256$ buckets would then be required.

But is this a good hash function?

Not if, for example, the last segment of an IP address tends to be a small (single- or double-digit) number; then low-numbered buckets would be crowded.

Taking the first segment of the IP address also invites disaster, for example, if *most of our customers come from Asia*.

How to choose a hash function? (cont'd)

- There is nothing *inherently wrong* with these two functions. If our 250 IP addresses were uniformly drawn from among all $N = 2^{32}$ possibilities, then these functions would behave well.
  The problem is we have no guarantee that the distribution of IP addresses is *uniform*.
- Conversely, there is no *single hash function*, no matter how sophisticated, that behaves well on all sets of data.
  Since a hash function maps 2^{32} IP addresses to just 250 names, there must be a collection of at least

$$2^{32}/250 \approx 2^{24} \approx 16,000,000$$

  IP addresses that are assigned the same name (or, in hashing terminology, **collide**).

Solution: *let us pick a hash function at random from some class of functions.*

Families of hash functions

Let us take the number of buckets to be not 250 but $n = 257$. a prime number!
We consider every IP address x as a quadruple $x =$

$$(x_1, x_2, x_3, x_4)$$

of integers modulo n.

We can define a function h from IP addresses to a number mod n as follows:

Fix any four numbers mod $n = 257$, say 87, 23, 125, and 4. Now map the IP address $(x_1, \ldots, x_4)$ to $h(x_1, \ldots, x_4) = (87x_1 + 23x_2 + 125x_3 + 4x_4) \mod 257$.

In general for any four coefficients $a_1, \ldots, a_4 \in \{0, 1, \ldots, n-1\}$ write $a = (a_1, a_2, a_3, a_4)$ and define h_a to be the following hash function:

$$h_a(x_1, \ldots, x_4) = (a_1 \cdot x_1 + a_2 \cdot x_2 + a_3 \cdot x_3 + a_4 \cdot x_4) \mod n.$$

Property

Consider any pair of distinct IP addresses $x = (x_1, \ldots, x_4)$ and $y = (y_1, \ldots, y_4)$. If the coefficients $a = (a_1, \ldots, a_4)$ are chosen uniformly at random from $\{0, 1, \ldots, n-1\}$, then

$$\Pr\left[h_a(x_1, \ldots, x_4) = h_a(y_1, \ldots, y_4)\right] = \frac{1}{n}.$$

Universal families of hash functions

Let

$$\mathcal{H} = \left\{ h_a \mid a \in \{0, 1, \ldots, n-1\}^4 \right\}.$$

It is **universal**:

> *For any two distinct data items x and y, exactly $|\mathcal{H}|/n$ of all the hash functions in $\mathcal{H}$ map x and y to the same bucket, where n is the number of buckets.*

An Intuitive Approach

Reference: Ravi Bhide's "Theory behind the technology" blog

Suppose a stream has size n, with m unique elements.
FM approximates m using time $\Theta(n)$ and memory $\Theta(\log m)$, along with estimate of standard deviation σ.

An Intuitive Approach

Reference: Ravi Bhide's "Theory behind the technology" blog

Suppose a stream has size n, with m unique elements.
FM approximates m using time $\Theta(n)$ and memory $\Theta(\log m)$, along with estimate of standard deviation σ.

Intuition: Suppose we have good random hash function
h: strings $\to \mathbb{N}_0$.
Since generated integers are random, $1/2^n$ of them have binary representation ending in 0^n.
IOW, if h generated an integer ending in 0^j for $j \in \{0, \ldots, m\}$, then number of unique strings is around 2^m.

An Intuitive Approach

Reference: Ravi Bhide's "Theory behind the technology" blog

Suppose a stream has size n, with m unique elements.
FM approximates m using time $\Theta(n)$ and memory $\Theta(\log m)$, along with estimate of standard deviation σ.

Intuition: Suppose we have good random hash function h: strings $\to \mathbb{N}_0$.
Since generated integers are random, $1/2^n$ of them have binary representation ending in 0^n.
IOW, if h generated an integer ending in 0^j for $j \in \{0, \ldots, m\}$, then number of unique strings is around 2^m.

FM maintains 1 bit per 0^i seen.
Output based on number of consecutive 0^i seen.

Informal description of algorithm:

1. Create bit vector v of length $L > \log n$.
   ($v[i]$ represents whether we've seen hash function value whose binary representation ends in 0^i.)
2. Initialize $v \to 0$.
3. Generate good random hash function.
4. For each word in input:
   - Hash it, let k be number of trailing zeros.
   - Set $v[k] = 1$.
5. Let $R = \min\{\, i : v[i] = 0 \,\}$.
   Note that R is number of consecutive ones, plus 1.
6. Calculate number of unique words as $2^R/\phi$, where $\phi = 0.77351$.
7. $\sigma(R) = 1.12$. Hence our count can be off by
   - factor of 2: about 32% of observations
   - factor of 4: about 5% of observations
   - factor of 8: about 0.3% of observations

For the record,

$$\phi = \frac{2e^{\gamma}}{3\sqrt{2}} \prod_{p=1}^{\infty} \left[\frac{(4p+1)(4p+2)}{(4p)(4p+3)} \right]^{(-1)^{\nu(p)}},$$

where $\nu(p)$ is the number of ones in the binary representation of p.

For the record,

$$\phi = \frac{2e^{\gamma}}{3\sqrt{2}} \prod_{p=1}^{\infty} \left[\frac{(4p+1)(4p+2)}{(4p)(4p+3)} \right]^{(-1)^{\nu(p)}},$$

where $\nu(p)$ is the number of ones in the binary representation of p.

Improving the accuracy:

- Averaging: Use multiple hash functions, and use average R.
- Bucketing: Averages are susceptible to large fluctuations. So use multiple buckets of hash functions, and use median of the average R values.
- Fine-tuning: Adjust number of hash functions in averaging and bucketing steps. (But higher computation cost.)

Results using Bhide's Java implementation:

- Wikipedia article on "United States Constitution" had 3978 unique words. When run ten times, Flajolet-Martin algorithmic reported values of 4902, 4202, 4202, 4044, 4367, 3602, 4367, 4202, 4202 and 3891 for an average of 4198. As can be seen, the average is about right, but the deviation is between -400 to 1000.
- Wikipedia article on "George Washington" had 3252 unique words. When run ten times, the reported values were 4044, 3466, 3466, 3466, 3744, 3209, 3335, 3209, 3891 and 3088, for an average of 3492.

Some Analysis: Idealized Solution

. . . uses real numbers!

Some Analysis: Idealized Solution

. . . uses real numbers!

Flagolet-Martin Algorithm (FM): Let $[n] = \{0, 1, \ldots, n\}$.

1. Pick random hash function $h\colon [n] \to [0, 1]$.
2. Maintain $X = \min\{\, h(i) : i \in \text{stream} \,\}$, smallest hash we've seen so far
3. `query()`: Output $1/X - 1$.

Intuition: Partitioning $[0, 1]$ into bins of size $1/(t + 1)$, where t distinct elements.

Claim: For the expected value, we have

$$\mathbb{E}[X] = \frac{1}{t+1}.$$

Claim: For the expected value, we have

$$\mathbb{E}[X] = \frac{1}{t+1}.$$

Proof:

$$\begin{aligned}
\mathbb{E}[X] &= \int_0^\infty \mathbb{P}(X > \lambda)\, d\lambda \\
&= \int_0^\infty \mathbb{P}(\forall i \in \text{stream}, h(i) > \lambda)\, d\lambda \\
&= \int_0^\infty \prod_{i \in \text{stream}} \mathbb{P}(h(i) > \lambda)\, d\lambda \\
&= \int_0^1 (1-\lambda)^t\, d\lambda \\
&= \frac{1}{t+1} \qquad \square
\end{aligned}$$

Claim: The variance satisfies

$$\mathbb{E}[X^2] = \frac{2}{(t+1)(t+2)}.$$

Proof:

$$\begin{aligned}
\mathbb{E}[X^2] &= \int_0^\infty \mathbb{P}(X^2 > \lambda)\, d\lambda = \int_0^\infty \mathbb{P}(X > \sqrt{\lambda})\, d\lambda \\
&= \int_0^1 (1-\sqrt{\lambda})^t\, d\lambda = 2\int_0^1 u^t(1-u)\, du \\
&= \frac{2}{(t+1)(t+2)} \qquad \square
\end{aligned}$$

Claim: The variance satisfies

$$\mathbb{E}[X^2] = \frac{2}{(t+1)(t+2)}.$$

Proof:

$$\begin{aligned}\mathbb{E}[X^2] &= \int_0^\infty \mathbb{P}(X^2 > \lambda)\, d\lambda = \int_0^\infty \mathbb{P}(X > \sqrt{\lambda})\, d\lambda \\ &= \int_0^1 (1-\sqrt{\lambda})^t\, d\lambda = 2\int_0^1 u^t(1-u)\, du \\ &= \frac{2}{(t+1)(t+2)} \quad \square\end{aligned}$$

Note that

$$\mathrm{Var}[X] = \frac{2}{(t+1)(t+2)} - \frac{1}{(t+1)^2} = \frac{t}{(t+1)^2(t+2)} < (\mathbb{E}[X])^2.$$

FM+: Given $\varepsilon > 0$ and $\eta \in (0, 1)$, run the FM algorithm $q = 1/(\varepsilon^2 \eta)$ times in parallel, obtaining $X_1, \ldots, X_q$. Then `query()` outputs

$$\frac{q}{\sum_{i=1}^{q} X_i} - 1.$$

FM+: Given $\varepsilon > 0$ and $\eta \in (0, 1)$, run the FM algorithm $q = 1/(\varepsilon^2 \eta)$ times in parallel, obtaining $X_1, \ldots, X_q$. Then `query()` outputs

$$\frac{q}{\sum_{i=1}^{q} X_i} - 1.$$

Claim: For any ε and η, the failure probability is given by

$$\mathbb{P}\left(\left| \frac{1}{q} \sum_{i=1}^{q} X_i - \frac{1}{t+1} \right| > \frac{\varepsilon}{t+1} \right) < \eta.$$

FM+: Given $\varepsilon > 0$ and $\eta \in (0,1)$, run the FM algorithm $q = 1/(\varepsilon^2 \eta)$ times in parallel, obtaining $X_1, \ldots, X_q$. Then `query()` outputs

$$\frac{q}{\sum_{i=1}^{q} X_i} - 1.$$

Claim: For any ε and η, the failure probability is given by

$$\mathbb{P}\left(\left| \frac{1}{q} \sum_{i=1}^{q} X_i - \frac{1}{t+1} \right| > \frac{\varepsilon}{t+1} \right) < \eta.$$

Proof: By Chebyshev's inequality, we have

$$\mathbb{P}\left(\left| \frac{1}{q} \sum_{i=1}^{q} X_i - \frac{1}{t+1} \right| > \frac{\varepsilon}{t+1} \right) < \frac{\mathrm{Var}\left[q^{-1} \sum_{i=1}^{q} X_i \right]}{\varepsilon^2/(t+1)^2} < \frac{1}{\varepsilon^2 q} = \eta,$$

as required. □

FM gives linear dependence on failure probability.
We want logarithmic dependence.

FM gives linear dependence on failure probability.
We want logarithmic dependence.

FM++: Given $\varepsilon > 0$ and $\delta \in (0, 1)$.
Let $t = \Theta(\log 1/\delta)$.
Run t copies of FM+ with $\eta = \frac{1}{3}$.
Then `query()` outputs the median of the FM+ estimates.

FM gives linear dependence on failure probability.
We want logarithmic dependence.

FM++: Given $\varepsilon > 0$ and $\delta \in (0, 1)$.
Let $t = \Theta(\log 1/\delta)$.
Run t copies of FM+ with $\eta = \frac{1}{3}$.
Then `query()` outputs the median of the FM+ estimates.

Claim:

$$\mathbb{P}\left(\left|\text{FM++} - \frac{1}{t+1}\right| < \frac{\varepsilon}{t+1}\right) < \Theta(\log 1/\delta).$$

FM gives linear dependence on failure probability.
We want logarithmic dependence.

FM++: Given $\varepsilon > 0$ and $\delta \in (0, 1)$.
Let $t = \Theta(\log 1/\delta)$.
Run t copies of FM+ with $\eta = \frac{1}{3}$.
Then `query()` outputs the median of the FM+ estimates.

Claim:

$$\mathbb{P}\left(\left|\text{FM++} - \frac{1}{t+1}\right| < \frac{\varepsilon}{t+1}\right) < \Theta(\log 1/\delta).$$

Reasoning: About the same as transition Morris+$\rightarrow$ Morris++.
Use indicator random variables $Y_1, \ldots, Y_n$, where

$$Y_i = \begin{cases} 1 & \text{if } i\text{th copy of FM+ doesn't give } (1+\varepsilon)\text{-approximation,} \\ 0 & \text{otherwise.} \end{cases}$$

Some Analysis: Non-idealized Solution

Need a pseudorandom hash function h.

Some Analysis: Non-idealized Solution

Need a pseudorandom hash function h.

Definition: A family $\mathcal{H}$ of functions mapping $[a]$ into $[b]$ is *k-wise independent* iff for all distinct $i_1, \ldots, i_k \in [a]$ and for all $j_1, \ldots, j_k \in [b]$, we have

$$\mathbb{P}_{h \in \mathcal{H}}(h(i_1) = j_1 \wedge \cdots \wedge h(i_k) = j_k) = \frac{1}{b^k}.$$

Can store $h \in \mathcal{H}$ in memory with $\log |\mathcal{H}|$ bits.

Some Analysis: Non-idealized Solution

Need a pseudorandom hash function h.

Definition: A family $\mathcal{H}$ of functions mapping $[a]$ into $[b]$ is *k-wise independent* iff for all distinct $i_1, \ldots, i_k \in [a]$ and for all $j_1, \ldots, j_k \in [b]$, we have

$$\mathbb{P}_{h \in \mathcal{H}}(h(i_1) = j_1 \wedge \cdots \wedge h(i_k) = j_k) = \frac{1}{b^k}.$$

Can store $h \in \mathcal{H}$ in memory with $\log |\mathcal{H}|$ bits.

Example Let $\mathcal{H} = \{ f : [a] \to [b] \}$ Then $|\mathcal{H}| = b^a$, and so $\log |\mathcal{H}| = a \lg b$. □

Some Analysis: Non-idealized Solution

Need a pseudorandom hash function h.

Definition: A family $\mathcal{H}$ of functions mapping $[a]$ into $[b]$ is *k-wise independent* iff for all distinct $i_1, \ldots, i_k \in [a]$ and for all $j_1, \ldots, j_k \in [b]$, we have

$$\mathbb{P}_{h \in \mathcal{H}}(h(i_1) = j_1 \wedge \cdots \wedge h(i_k) = j_k) = \frac{1}{b^k}.$$

Can store $h \in \mathcal{H}$ in memory with $\log |\mathcal{H}|$ bits.

Example Let $\mathcal{H} = \{ f : [a] \to [b] \}$ Then $|\mathcal{H}| = b^a$, and so $\log |\mathcal{H}| = a \lg b$. □

Less trivial examples exist.

Some Analysis: Non-idealized Solution

Need a pseudorandom hash function h.

Definition: A family $\mathcal{H}$ of functions mapping $[a]$ into $[b]$ is *k-wise independent* iff for all distinct $i_1, \ldots, i_k \in [a]$ and for all $j_1, \ldots, j_k \in [b]$, we have

$$\mathbb{P}_{h \in \mathcal{H}}(h(i_1) = j_1 \wedge \cdots \wedge h(i_k) = j_k) = \frac{1}{b^k}.$$

Can store $h \in \mathcal{H}$ in memory with $\log |\mathcal{H}|$ bits.

Example Let $\mathcal{H} = \{ f : [a] \to [b] \}$ Then $|\mathcal{H}| = b^a$, and so $\log |\mathcal{H}| = a \lg b$. □

Less trivial examples exist.

Assume: Access to some pairwise independent hash families. Can store in $\log n$ bits.

Common Strategy: Geometric Sampling of Streams

Let $\tilde{t}$ be a 32-approximation to t. Want a $(1+\varepsilon)$-approximation.

Common Strategy: Geometric Sampling of Streams

Let $\tilde{t}$ be a 32-approximation to t. Want a $(1+\varepsilon)$-approximation.

Trivial solution (TS): Let $K = c/\varepsilon^2$ and remember first K distinct elements in stream.

Common Strategy: Geometric Sampling of Streams

Let $\tilde{t}$ be a 32-approximation to t. Want a $(1+\varepsilon)$-approximation.

Trivial solution (TS): Let $K = c/\varepsilon^2$ and remember first K distinct elements in stream.

Our algorithm:

1. Assume $n = 2^K$ for some $K \in \mathbb{N}$.
2. Pick $g : [n] \to]n]$ from pairwise family.
3. `init()`: Create $\log n + 1$ trivial solutions $\mathrm{TS}_0, \ldots, \mathrm{TS}_K$.
4. `update()`: Run $\mathrm{TS}_{\mathrm{LSB}(g(i))}$ on the input i.
5. `query()`: Choose $j \approx \log(\tilde{t}\varepsilon^2) - 1$.
6. Output $\mathrm{TS}_{j \cdot \texttt{query}() \cdot 2^{+1}}$.

Explanation: LSB is “least significant bit”.

Explanation: LSB is "least significant bit".

For example, suppose $g : [16] \to [16]$ with $g(i) = 1100$; then $\mathrm{LSB}(g(i)) = 2$.

Explanation: LSB is "least significant bit".

For example, suppose $g : [16] \to [16]$ with $g(i) = 1100$; then $\mathrm{LSB}(g(i)) = 2$.

But if $g(i) = 1001$, then $\mathrm{LSB}(g(i)) = 0$.
This explains the "$+1$" in step 3.

Define a set of virtual streams wrapping the trivial solutions:

$$\begin{array}{rcccc} \mathrm{VS}_0 & 0 & \longrightarrow & \mathrm{TS}_0 \\ \vdots & \vdots & \vdots & \vdots \\ \mathrm{VS}_{\log n} & \log n & \longrightarrow & \mathrm{TS}_{\log n} \end{array}$$

Now choose the highest nonempty virtual stream.