# Graph algorithms

# Outline

- Counting triangles
- An interlude: probabilistic counters
- Computing distances [and geometric centralities] in large graphs using HyperBall
- HyperBall on Facebook (a Milgram-like experiment)
- Other applications of distances (in particular: robustness)

# Counting triangles

# Triangles and clustering

One of the distinguishing features of many complex (undirected) networks is their relatively large *clustering coefficient*:

# Triangles and clustering

One of the distinguishing features of many complex (undirected) networks is their relatively large *clustering coefficient*:

- the clustering coefficient of a vertex $x$ is the fraction of its pairs of neighbors that are neighbors of each other

# Triangles and clustering

One of the distinguishing features of many complex (undirected) networks is their relatively large *clustering coefficient*:

- the clustering coefficient of a vertex $x$ is the fraction of its pairs of neighbors that are neighbors of each other
- i.e., the fraction of triples $(y_1, x, y_2)$ formed by two edges that form themselves a triangle.

# Triangles and clustering

One of the distinguishing features of many complex (undirected) networks is their relatively large *clustering coefficient*:

- the clustering coefficient of a vertex $x$ is the fraction of its pairs of neighbors that are neighbors of each other
- i.e., the fraction of triples $(y_1, x, y_2)$ formed by two edges that form themselves a triangle.
- Social networks exhibit a relatively large clustering coefficient, compared to their diameter.

# Local vs. global clustering coefficient

As we said, the *local clustering coefficient* of a vertex $x$ is

$$cc(x) = \frac{|\{\{y,z\}|y,z \in N(x), y \neq z, y \in N(z)\}|s}{\binom{d(x)}{2}}$$

# Local vs. global clustering coefficient

As we said, the *local clustering coefficient* of a vertex $x$ is

$$cc(x) = \frac{|\{\{y,z\}|y,z \in N(x), y \neq z, y \in N(z)\}|s}{\binom{d(x)}{2}}$$

A related notion is that of *global clustering coefficient*

$$cc_G = \frac{\sum_x cc(x)}{n},$$

the average clustering coefficient of its vertices.

# Local vs. global clustering coefficient

As we said, the *local clustering coefficient* of a vertex $x$ is

$$cc(x) = \frac{|\{\{y,z\}|y,z \in N(x), y \neq z, y \in N(z)\}|s}{\binom{d(x)}{2}}$$

A related notion is that of *global clustering coefficient*

$$cc_G = \frac{\sum_x cc(x)}{n},$$

the average clustering coefficient of its vertices.

- How can one efficiently compute or approximate the local/global clustering coefficient?

# Local vs. global clustering coefficient

As we said, the *local clustering coefficient* of a vertex $x$ is

$$cc(x) = \frac{|\{\{y,z\}|y,z \in N(x), y \neq z, y \in N(z)\}|s}{\binom{d(x)}{2}}$$

A related notion is that of *global clustering coefficient*

$$cc_G = \frac{\sum_x cc(x)}{n},$$

the average clustering coefficient of its vertices.

- How can one efficiently compute or approximate the local/global clustering coefficient?
- Here we consider the local case

# Triangles of an edge

Define, for every edge $yz$

$$T(yz) = |N(y) \cap N(z)|.$$

# Triangles of an edge

Define, for every edge $yz$

$$T(yz) = |N(y) \cap N(z)|.$$

This is the number of triangles that the edge $yz$ closes.

# Triangles of an edge

Define, for every edge $yz$

$$T(yz) = |N(y) \cap N(z)|.$$

This is the number of triangles that the edge $yz$ closes.

From this, you can define

$$T(x) = \sum_{y \in N(x)} T(xy),$$

# Triangles of an edge

Define, for every edge $yz$

$$T(yz) = |N(y) \cap N(z)|.$$

This is the number of triangles that the edge $yz$ closes.

From this, you can define

$$T(x) = \sum_{y \in N(x)} T(xy),$$

hence

$$cc(x) = \frac{T(x)}{2\binom{d(x)}{2}}$$

because $T(x)$ counts every triangle twice...

# Jaccard coefficient of an edge

The problem thus can be reduced to computing, for every edge $yz$,

$$T(yz) = |N(y) \cap N(z)|.$$

# Jaccard coefficient of an edge

The problem thus can be reduced to computing, for every edge $yz$,

$$T(yz) = |N(y) \cap N(z)|.$$

Recall the notion of Jaccard coefficient:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

# Jaccard coefficient of an edge

The problem thus can be reduced to computing, for every edge $yz$,

$$T(yz) = |N(y) \cap N(z)|.$$

Recall the notion of Jaccard coefficient:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Equivalently:

$$\frac{1}{J(A, B)} = \frac{|A \cup B|}{|A \cap B|} = \frac{|A| + |B| - |A \cap B|}{|A \cap B|} = \frac{|A| + |B|}{|A \cap B|} - 1.$$

# Jaccard coefficient of an edge

The problem thus can be reduced to computing, for every edge $yz$,

$$T(yz) = |N(y) \cap N(z)|.$$

Recall the notion of Jaccard coefficient:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Equivalently:

$$\frac{1}{J(A, B)} = \frac{|A \cup B|}{|A \cap B|} = \frac{|A| + |B| - |A \cap B|}{|A \cap B|} = \frac{|A| + |B|}{|A \cap B|} - 1.$$

Hence

$$|A \cap B| = \frac{|A| + |B|}{1 + \frac{1}{J(A,B)}}.$$

# Jaccard coefficient through min-wise permutations

So the problem is further reduced to computing, for every edge $yz$,

$$J(yz) = J(N(y), N(z)),$$

after which

$$T(yz) = \frac{d(y) + d(z)}{1 + \frac{1}{J(yz)}}.$$

# Jaccard coefficient through min-wise permutations

So the problem is further reduced to computing, for every edge $yz$,

$$J(yz) = J(N(y), N(z)),$$

after which

$$T(yz) = \frac{d(y) + d(z)}{1 + \frac{1}{J(yz)}}.$$

Recall that

# Jaccard coefficient through min-wise permutations

So the problem is further reduced to computing, for every edge $yz$,

$$J(yz) = J(N(y), N(z)),$$

after which

$$T(yz) = \frac{d(y) + d(z)}{1 + \frac{1}{J(yz)}}.$$

Recall that

**Theorem**

*Let $A, B \subseteq \Omega = \{0, 1, \ldots, M-1\}$, and let $\Pi$ be the set of all $M!$ permutations of $\Omega$. If $\pi$ is drawn uniformly at random from $\Pi$*

$$P[\min(\pi(A)) = \min(\pi(B))] = J(A, B).$$

# The algorithm (outline)

So, the idea to compute $J(N(y), N(z))$ is:

# The algorithm (outline)

So, the idea to compute $J(N(y), N(z))$ is:

- generate a random permutation (i.e., renumbering) $\pi$ of the nodes

# The algorithm (outline)

So, the idea to compute $J(N(y), N(z))$ is:

- generate a random permutation (i.e., renumbering) $\pi$ of the nodes
- compute $\min \pi(N(y))$ and $\min \pi(N(z))$

# The algorithm (outline)

So, the idea to compute $J(N(y), N(z))$ is:

- generate a random permutation (i.e., renumbering) $\pi$ of the nodes
- compute $\min \pi(N(y))$ and $\min \pi(N(z))$
- if the two values coincide, count $+1$

# The algorithm (outline)

So, the idea to compute $J(N(y), N(z))$ is:

- generate a random permutation (i.e., renumbering) $\pi$ of the nodes
- compute $\min \pi(N(y))$ and $\min \pi(N(z))$
- if the two values coincide, count $+1$

Repeat the above procedue many times, and use the fraction of $+1$'s to estimate $J(N(y), N(z))$.

# The algorithm (outline)

Some further notes:

# The algorithm (outline)

Some further notes:

- we have a counter per edge $C[yz]$ (to count the number of $+1$'s): we keep them on external memory

# The algorithm (outline)

Some further notes:

- we have a counter per edge $C[yz]$ (to count the number of $+1$'s): we keep them on external memory
- to know if $\min \pi(N(y)) = \min \pi(N(z))$ we must have computed the minima before: we need two passes

Some further notes:

- we have a counter per edge $C[yz]$ (to count the number of $+1$'s): we keep them on external memory
- to know if $\min \pi(N(y)) = \min \pi(N(z))$ we must have computed the minima before: we need two passes
  - first pass: generate the permutation $\pi$ and compute the minima $\min \pi N(-))$ (kept in central memory)

# The algorithm (outline)

Some further notes:

- we have a counter per edge $C[yz]$ (to count the number of $+1$'s): we keep them on external memory
- to know if $\min \pi(N(y)) = \min \pi(N(z))$ we must have computed the minima before: we need two passes
  - first pass: generate the permutation $\pi$ and compute the minima $\min \pi N(-))$ (kept in central memory)
  - second pass: increment the counters

# The algorithm (outline)

Some further notes:

- we have a counter per edge $C[yz]$ (to count the number of $+1$'s): we keep them on external memory
- to know if $\min \pi(N(y)) = \min \pi(N(z))$ we must have computed the minima before: we need two passes
  - first pass: generate the permutation $\pi$ and compute the minima $\min \pi N(-))$ (kept in central memory)
  - second pass: increment the counters
- we use hashing instead of permutations (equivalent, provided that the probability of collision is negligible).

# The algorithm (1)

# The algorithm (1)

**for** $K$ times **do**

# The algorithm (1)

**for** $K$ times **do**
  generate a hash function $h : V_G \to \mathbf{N}$

# The algorithm (1)

**for** $K$ times **do**
    generate a hash function $h : V_G \to \mathbf{N}$
    **for** each $x \in V_G$ **do**

# The algorithm (1)

**for** $K$ times **do**
  generate a hash function $h : V_G \to \mathbf{N}$
  **for** each $x \in V_G$ **do**
    $M[x] \leftarrow \min_{y \in N(x)} h(y)$
  **end for**

# The algorithm (1)

**for** $K$ times **do**
  generate a hash function $h : V_G \to \mathbf{N}$
  **for** each $x \in V_G$ **do**
    $M[x] \leftarrow \min_{y \in N(x)} h(y)$
  **end for**
  **for** each $x \in V_G$ **do**
    **for** each $y \in N(x)$ **do**

# The algorithm (1)

**for** $K$ times **do**
  generate a hash function $h : V_G \to \mathbf{N}$
  **for** each $x \in V_G$ **do**
    $M[x] \leftarrow \min_{y \in N(x)} h(y)$
  **end for**
  **for** each $x \in V_G$ **do**
    **for** each $y \in N(x)$ **do**
      read $C[xy]$ from disk

# The algorithm (1)

**for** $K$ times **do**
  generate a hash function $h : V_G \to \mathbf{N}$
  **for** each $x \in V_G$ **do**
    $M[x] \leftarrow \min_{y \in N(x)} h(y)$
  **end for**
  **for** each $x \in V_G$ **do**
    **for** each $y \in N(x)$ **do**
      read $C[xy]$ from disk
      **if** $M[x] = M[y]$ **then**
        $C[xy] \leftarrow C[xy] + 1$
      **end if**

# The algorithm (1)

```
for K times do
    generate a hash function h : V_G → N
    for each x ∈ V_G do
        M[x] ← min_{y∈N(x)} h(y)
    end for
    for each x ∈ V_G do
        for each y ∈ N(x) do
            read C[xy] from disk
            if M[x] = M[y] then
                C[xy] ← C[xy] + 1
            end if
            write C[xy] to disk
        end for
    end for
end for
```

# The algorithm (2)

# The algorithm (2)

**for** each $x \in V_G$ **do**
    $T(x) \leftarrow 0$
    **for** each $y \in N(x)$ **do**

# The algorithm (2)

**for** each $x \in V_G$ **do**
  $T(x) \leftarrow 0$
  **for** each $y \in N(x)$ **do**
    read $C[xy]$ from disk

# The algorithm (2)

**for** each $x \in V_G$ **do**
  $T(x) \leftarrow 0$
  **for** each $y \in N(x)$ **do**
    read $C[xy]$ from disk
    $T(xy) \leftarrow (d(x) + d(y))/(1 + K/C[xy])$

# The algorithm (2)

**for** each $x \in V_G$ **do**
  $T(x) \leftarrow 0$
  **for** each $y \in N(x)$ **do**
    read $C[xy]$ from disk
    $T(xy) \leftarrow (d(x) + d(y))/(1 + K/C[xy])$
    $T(x) \leftarrow T(xy)$
  **end for**

# The algorithm (2)

**for** each $x \in V_G$ **do**
  $T(x) \leftarrow 0$
  **for** each $y \in N(x)$ **do**
    read $C[xy]$ from disk
    $T(xy) \leftarrow (d(x) + d(y))/(1 + K/C[xy])$
    $T(x) \leftarrow T(xy)$
  **end for**
  $cc(x) \leftarrow T(x)/(d(x)^2 - d(x))$
**end for**

# An interlude: probabilistic counters

# Probabilistic counter

**Careful**: the name is misleading, and similar to "approximate counter", but the two notions are different:

# Probabilistic counter

**Careful**: the name is misleading, and similar to "approximate counter", but the two notions are different:

- an *approximate counter* is like a counter (with primitives "increment()" and "value()") that uses exponentially less bits than a standard counter and returns only approximate values, with some probabilistic guarantee [pioneer: Morris 1978]

# Probabilistic counter

**Careful**: the name is misleading, and similar to "approximate counter", but the two notions are different:

- an *approximate counter* is like a counter (with primitives "increment()" and "value()") that uses exponentially less bits than a standard counter and returns only approximate values, with some probabilistic guarantee [pioneer: Morris 1978]
- a *probabilistic counter* is like a set (with primitives "add(x)" and "size()"):

# Probabilistic counter

**Careful**: the name is misleading, and similar to "approximate counter", but the two notions are different:

- an *approximate counter* is like a counter (with primitives "increment()" and "value()") that uses exponentially less bits than a standard counter and returns only approximate values, with some probabilistic guarantee [pioneer: Morris 1978]
- a *probabilistic counter* is like a set (with primitives "add(x)" and "size()"): it is called a counter because it can be used to count the number of *distinct* elements in a stream [pioneer: Flajolet 1985].

ADT to represent a subset $A$ of a universe $\Omega$. The ADT has two primitives:

# Probabilistic counters

ADT to represent a subset $A$ of a universe $\Omega$. The ADT has two primitives:

- "add(x)" to add an element $x \in \Omega$ to $A$

# Probabilistic counters

ADT to represent a subset $A$ of a universe $\Omega$. The ADT has two primitives:

- "add(x)" to add an element $x \in \Omega$ to $A$
- "size()" to get the (approximate) $|A|$

# Probabilistic counters

ADT to represent a subset $A$ of a universe $\Omega$. The ADT has two primitives:

- "add(x)" to add an element $x \in \Omega$ to $A$
- "size()" to get the (approximate) $|A|$

With $|\Omega|$ bits you can realize an *exact* (non-approximate) version of this.

# Probabilistic counters

ADT to represent a subset $A$ of a universe $\Omega$. The ADT has two primitives:

- "add(x)" to add an element $x \in \Omega$ to $A$
- "size()" to get the (approximate) $|A|$

With $|\Omega|$ bits you can realize an *exact* (non-approximate) version of this.

Probabilistic counters "in the marketplace" use much less (e.g., $\log |\Omega|$ or $\log \log |\Omega|$ bits), and give only probabilistic guarantees on the value ("the value differs from the real size more than $\epsilon$% with probability not larger than..." )

Many solutions on the market! They differ by:

# Probabilistic counters

Many solutions on the market! They differ by:

- size vs. accuracy tradeoff

# Probabilistic counters

Many solutions on the market! They differ by:

- size vs. accuracy tradeoff
- "quantity" of randomness needed

# Probabilistic counters

Many solutions on the market! They differ by:

- size vs. accuracy tradeoff
- "quantity" of randomness needed
- simplifying assumptions they take for granted.

# FM counters

Flajolet-Martin counters (1985) represent $\Omega$ in $\ell = \log|\Omega|$ bits.

# FM counters

Flajolet-Martin counters (1985) represent $\Omega$ in $\ell = \log|\Omega|$ bits.

- We choose a hash function $h : \Omega \to \{0, \ldots, 2^\ell - 1\}$

# FM counters

Flajolet-Martin counters (1985) represent $\Omega$ in $\ell = \log|\Omega|$ bits.

- We choose a hash function $h : \Omega \to \{0, \ldots, 2^\ell - 1\}$
- Assumption: elements of $h(x)$ are uniformly distributed

# FM counters

Flajolet-Martin counters (1985) represent $\Omega$ in $\ell = \log |\Omega|$ bits.

- We choose a hash function $h : \Omega \to \{0, \ldots, 2^\ell - 1\}$
- Assumption: elements of $h(x)$ are uniformly distributed
- Use a counter with $\ell$ bits (indices: $0, 1, \ldots, \ell - 1$)

# FM counters

Flajolet-Martin counters (1985) represent $\Omega$ in $\ell = \log |\Omega|$ bits.

- We choose a hash function $h : \Omega \to \{0, \ldots, 2^\ell - 1\}$
- Assumption: elements of $h(x)$ are uniformly distributed
- Use a counter with $\ell$ bits (indices: $0, 1, \ldots, \ell - 1$)
- "add(x)": set to 1 the bit of index $k$ where $k$ is the number of trailing zeroes in the binary representation of $h(x)$

# FM counters

Flajolet-Martin counters (1985) represent $\Omega$ in $\ell = \log|\Omega|$ bits.

- We choose a hash function $h : \Omega \rightarrow \{0, \ldots, 2^\ell - 1\}$
- Assumption: elements of $h(x)$ are uniformly distributed
- Use a counter with $\ell$ bits (indices: $0, 1, \ldots, \ell - 1$)
- "add(x)": set to 1 the bit of index $k$ where $k$ is the number of trailing zeroes in the binary representation of $h(x)$
- "size()": see below.

# FM counters (cont'd)

Let $n = |A|$. About $n/2$ of them are odd (i.e., have 0 trailing zeroes), about $n/4$ end with 10, about $n/8$ end with 100...

# FM counters (cont'd)

Let $n = |A|$. About $n/2$ of them are odd (i.e., have 0 trailing zeroes), about $n/4$ end with 10, about $n/8$ end with 100...

- if $n/2^k$ is large enough, probably the $k$-th bit was set.

# FM counters (cont'd)

Let $n = |A|$. About $n/2$ of them are odd (i.e., have 0 trailing zeroes), about $n/4$ end with 10, about $n/8$ end with 100...

- if $n/2^k$ is large enough, probably the $k$-th bit was set.
- in other words, if $k \ll \log_2 n$ it is very likely that $k$ was set...

# FM counters (cont'd)

Let $n = |A|$. About $n/2$ of them are odd (i.e., have 0 trailing zeroes), about $n/4$ end with 10, about $n/8$ end with 100...

- if $n/2^k$ is large enough, probably the $k$-th bit was set.
- in other words, if $k \ll \log_2 n$ it is very likely that $k$ was set...
- ...and, if $k \gg \log_2 n$ it is very likely that $k$ was not set.

# FM counters (cont'd)

Let $n = |A|$. About $n/2$ of them are odd (i.e., have 0 trailing zeroes), about $n/4$ end with 10, about $n/8$ end with 100...

- if $n/2^k$ is large enough, probably the $k$-th bit was set.
- in other words, if $k \ll \log_2 n$ it is very likely that $k$ was set...
- ...and, if $k \gg \log_2 n$ it is very likely that $k$ was not set.

$\Longrightarrow$ "size()": Let $k$ be the number of trailing 1's; return $2^k/0.77351$

# FM counters (cont'd)

Let $n = |A|$. About $n/2$ of them are odd (i.e., have 0 trailing zeroes), about $n/4$ end with 10, about $n/8$ end with 100...

- if $n/2^k$ is large enough, probably the $k$-th bit was set.
- in other words, if $k \ll \log_2 n$ it is very likely that $k$ was set...
- ... and, if $k \gg \log_2 n$ it is very likely that $k$ was not set.

$\Longrightarrow$ "size()": Let $k$ be the number of trailing 1's; return $2^k/0.77351$

... Unbiased estimator.

# Tricks (1): Repetita iuvant

The probabilistic guarantee of the FM counter depends basically on the choice of the hash function $h$: different choices of $h$ produce different estimates!

# Tricks (1): Repetita iuvant

The probabilistic guarantee of the FM counter depends basically on the choice of the hash function $h$: different choices of $h$ produce different estimates!

Basic (standard) solution to improve concentration: make $T$ runs (with different choices of $h$), and average!

# Tricks (1): Repetita iuvant

The probabilistic guarantee of the FM counter depends basically on the choice of the hash function $h$: different choices of $h$ produce different estimates!

Basic (standard) solution to improve concentration: make $T$ runs (with different choices of $h$), and average! Variance reduces from $\sigma^2$ to $\sigma^2/T$

# Tricks (1): Repetita iuvant

The probabilistic guarantee of the FM counter depends basically on the choice of the hash function $h$: different choices of $h$ produce different estimates!

Basic (standard) solution to improve concentration: make $T$ runs (with different choices of $h$), and average! Variance reduces from $\sigma^2$ to $\sigma^2/T$

Accuracy vs. time! [Or space, if you run the $T$ solutions in parallel]

# Tricks (2): Splitting trick

# Tricks (2): Splitting trick

- Fix a splitting (hash) function $s : \Omega \to \{0, \ldots, k-1\}$ that divides the universe into $k$ sub-universes of (approximately) equal size $\Omega_0, \ldots, \Omega_{k-1}$

# Tricks (2): Splitting trick

- Fix a splitting (hash) function $s : \Omega \to \{0, \ldots, k-1\}$ that divides the universe into $k$ sub-universes of (approximately) equal size $\Omega_0, \ldots, \Omega_{k-1}$
- $\sigma$ can just use the first $\log k$ bits of the hash function $h$, leaving the remaining bits for the rest of the processing

# Tricks (2): Splitting trick

- Fix a splitting (hash) function $s : \Omega \to \{0, \ldots, k-1\}$ that divides the universe into $k$ sub-universes of (approximately) equal size $\Omega_0, \ldots, \Omega_{k-1}$
- $\sigma$ can just use the first $\log k$ bits of the hash function $h$, leaving the remaining bits for the rest of the processing
- Use $k$ distinct counters for the $k$ splits, applying the appropriate counter every time a new element comes in

# Tricks (2): Splitting trick

- Fix a splitting (hash) function $s : \Omega \to \{0, \ldots, k-1\}$ that divides the universe into $k$ sub-universes of (approximately) equal size $\Omega_0, \ldots, \Omega_{k-1}$
- $\sigma$ can just use the first $\log k$ bits of the hash function $h$, leaving the remaining bits for the rest of the processing
- Use $k$ distinct counters for the $k$ splits, applying the appropriate counter every time a new element comes in
- Space requirement $k \log(|\Omega|/k)$

# Tricks (2): Splitting trick

- Fix a splitting (hash) function $s : \Omega \to \{0, \ldots, k-1\}$ that divides the universe into $k$ sub-universes of (approximately) equal size $\Omega_0, \ldots, \Omega_{k-1}$
- $\sigma$ can just use the first $\log k$ bits of the hash function $h$, leaving the remaining bits for the rest of the processing
- Use $k$ distinct counters for the $k$ splits, applying the appropriate counter every time a new element comes in
- Space requirement $k \log(|\Omega|/k)$
- Increasing $k$ improves accuracy (in the limit, $k = |\Omega|$ and the counter becomes exact!)

# Tricks (2): Splitting trick

- Fix a splitting (hash) function $s : \Omega \to \{0, \ldots, k-1\}$ that divides the universe into $k$ sub-universes of (approximately) equal size $\Omega_0, \ldots, \Omega_{k-1}$
- $\sigma$ can just use the first $\log k$ bits of the hash function $h$, leaving the remaining bits for the rest of the processing
- Use $k$ distinct counters for the $k$ splits, applying the appropriate counter every time a new element comes in
- Space requirement $k \log(|\Omega|/k)$
- Increasing $k$ improves accuracy (in the limit, $k = |\Omega|$ and the counter becomes exact!)
- Accuracy vs. space

# DF (HyperLogLog) counters

Durand-Flajolet counters (2003) represent $\Omega$ in $\ell = \log\log|\Omega|$ bits.

# DF (HyperLogLog) counters

Durand-Flajolet counters (2003) represent $\Omega$ in $\ell = \log \log |\Omega|$ bits.

- It uses a $\log |\Omega|$ splitting...

# DF (HyperLogLog) counters

Durand-Flajolet counters (2003) represent $\Omega$ in $\ell = \log\log|\Omega|$ bits.

- It uses a $\log|\Omega|$ splitting...
- ...followed by a count of the *leading* zeroes

# DF (HyperLogLog) counters

Durand-Flajolet counters (2003) represent $\Omega$ in $\ell = \log\log|\Omega|$ bits.

- It uses a $\log|\Omega|$ splitting...
- ...followed by a count of the *leading* zeroes
- The maximum (splitwise) of such maxima is stored

# DF (HyperLogLog) counters

Durand-Flajolet counters (2003) represent $\Omega$ in $\ell = \log\log|\Omega|$ bits.

- It uses a $\log|\Omega|$ splitting...
- ...followed by a count of the *leading* zeroes
- The maximum (splitwise) of such maxima is stored
- It is asymptotically almost unbiased with small error:

$$\text{standard error} \approx \frac{1.30}{m}$$

if $\approx m \log\log|\Omega|$ bits are used

# DF (HyperLogLog) counters

Durand-Flajolet counters (2003) represent $\Omega$ in $\ell = \log\log|\Omega|$ bits.

- It uses a $\log|\Omega|$ splitting...
- ...followed by a count of the *leading* zeroes
- The maximum (splitwise) of such maxima is stored
- It is asymptotically almost unbiased with small error:

$$\text{standard error} \approx \frac{1.30}{m}$$

  if $\approx m \log\log|\Omega|$ bits are used
- "cardinalities up to $10^9$ can be approximated with up to 2% of error in 1.5KB of memory!"