Probabilistic Counting

Partially based on slides from:

http://infolab.stanford.edu/~ullman/mining/2009/index.html

Anand Rajaraman & Jeff Ullman

Counting Distinct Elements

- Problem: a data stream consists of elements chosen from a set of size ***n (n very big!)***. *How to maintain the count of the number of distinct elements seen so far?*

- Obvious approach: maintain the set of elements seen (costs O(n) memory!)

- *Use less memory (and accept loss of accuracy)*

Applications

- How many different URLs have we seen so far?
- How many different words are found among the Web pages being crawled at a site?
  - Unusually low or high numbers could indicate “artificial pages”
- How many different Web pages does each customer requests in a week?
- How many distinct elements in a column of a table?
  (optimization of the join operation of two tables)
- How many distinct <source, destination> pairs through a router?
  (detection of DoS attacks)

Using Small Storage

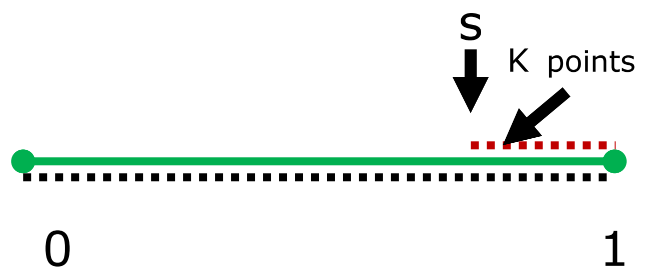
- Real Problem:
  what if we do not have space to store the complete set?

- Estimate the count in an unbiased way.

- Accept that the count may be in error, but limit the probability that the error is large.

A simple idea: MinTopK estimate

- *Hash incoming objects into doubles from the interval [0, 1] and count them shrinking the interval if needed.*
- Due to limited memory, maintain only the K biggest values (“TopK”), say, K=1000.
- Let **s** denote the minimum of our set (MinTopK)
- The number of distinct elements ≈ **K/(1-s)**
- *What about the accuracy? The number of bits?*



Flajolet-Martin Approach

X_1 -> 10001110->1
X_2 -> 01010010->1
X_3 -> 10011011->0
X_4 -> 00101000->3
X_5 -> 01011110->1
... -> .... ->...
#bits #bits
N -> $\log_2(N)$-> $\log_2(\log_2(N))$
eg.: $\log_2(\log_2(1.000.000))<5$

- Key idea:
  - hash passing elements into short bitstrings,
  - store only the length of the longest tail of 0's,
  - "the more distinct elements" the longer the longest tail of 0's.

- Pick a hash function h that maps each of the m elements to $\log_2 m$ bits.
- For each stream element a, let $r(a)$ be the number of trailing 0's in $h(a)$.
- Record R = the maximum $r(a)$ seen.
- Estimate the number of distinct elements as 2^R. ***WHY?***

Why does it Work?

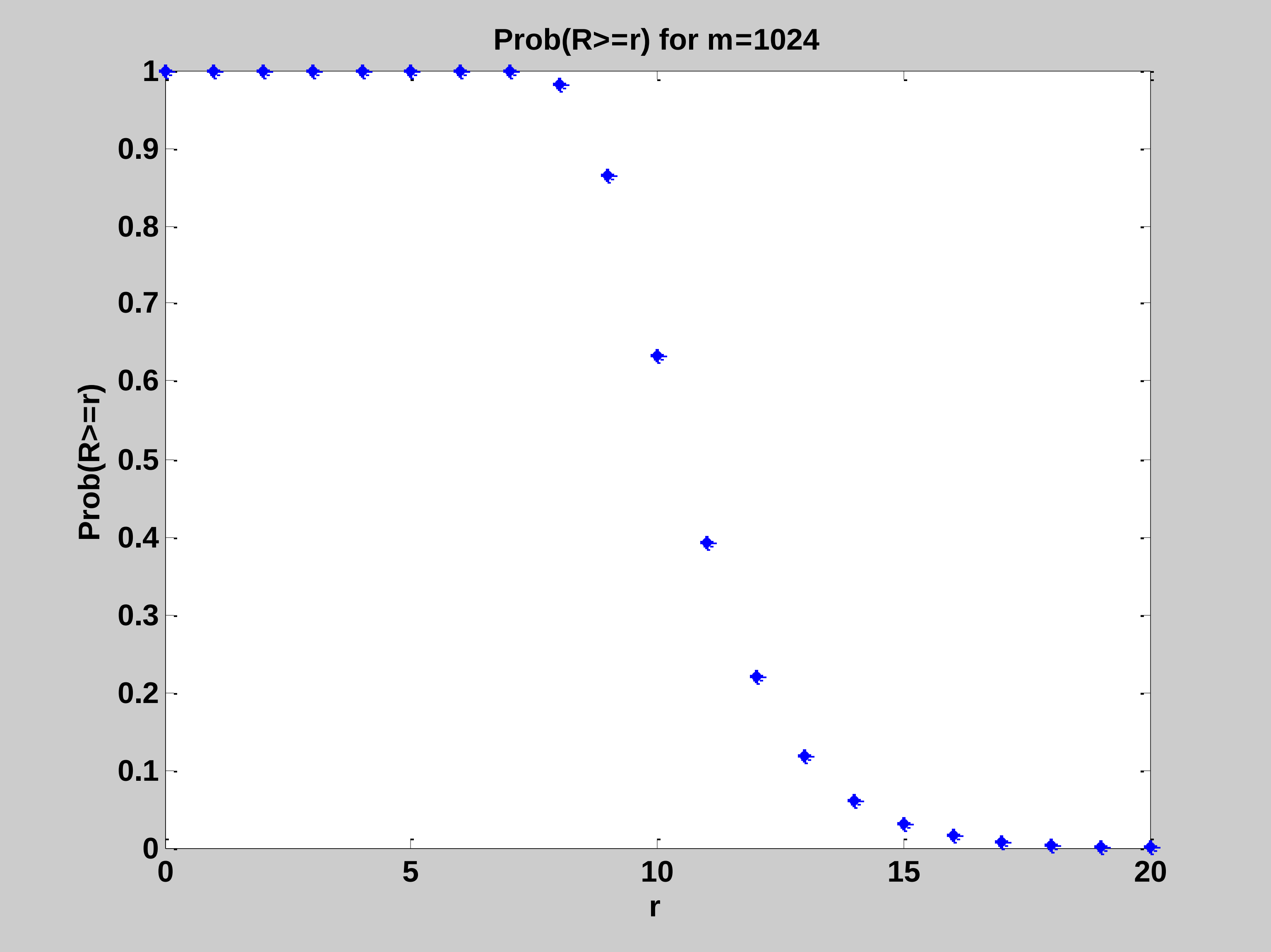
- The probability that a given $h(a)$ ends in at least r 0's is 2^{-r}.
- If there are m different elements, the probability that $R \geq r$ is $1 - \boxed{(\boxed{1 - 2^{-r}})^m}$

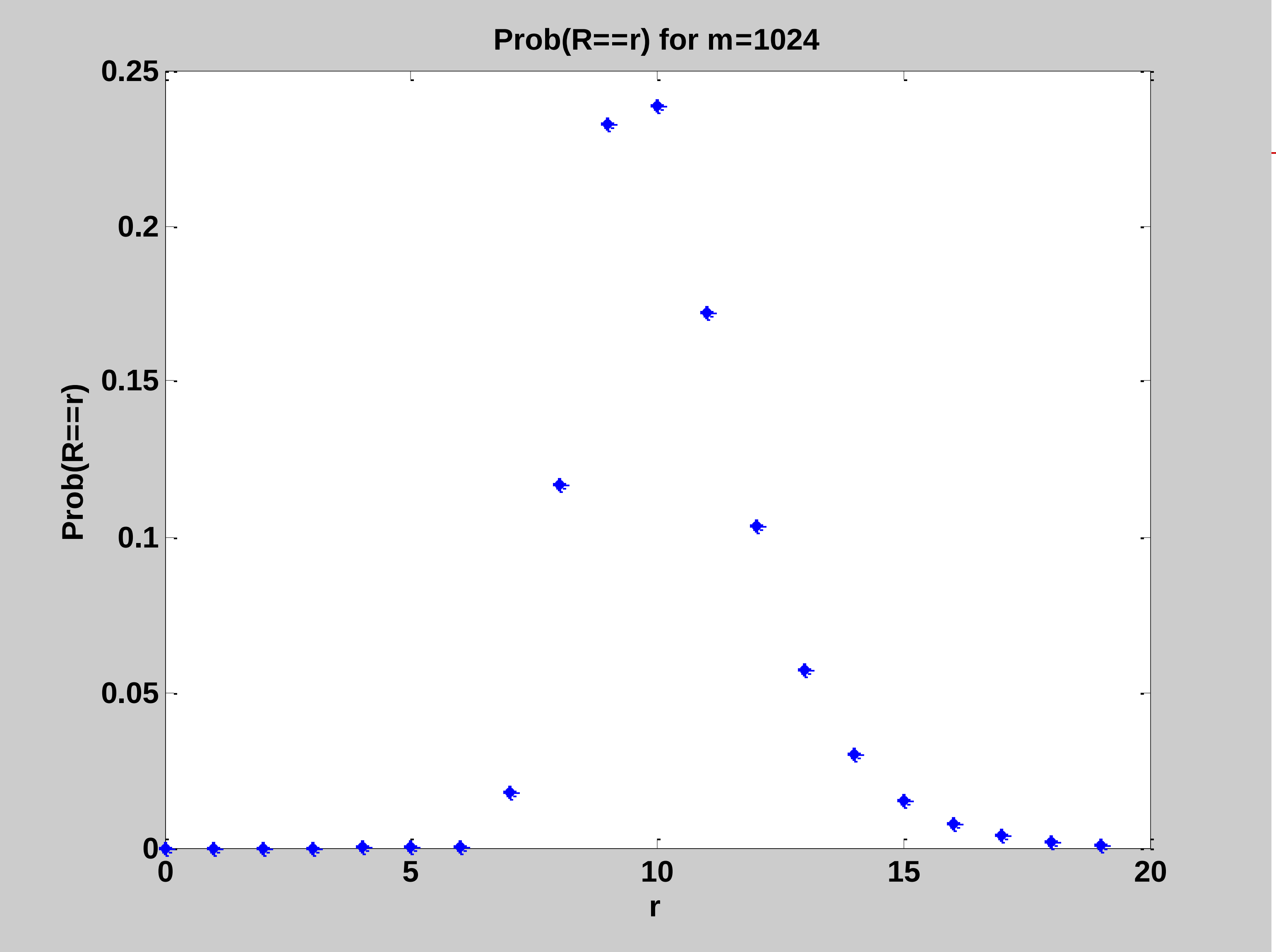
Prob. all h(a)'s end in fewer than r 0's.

Prob. a given h(a) ends in fewer than r 0's.

Why does it Work – (2)

- Since 2^{-r} is small, $1 - (1-2^{-r})^m \approx 1 - e^{-m/2^r}$
- If $2^r >> m$, $1 - (1 - 2^{-r})^m \approx 1 - 1 = 0$
- If $2^r << m$, $1 - (1 - 2^{-r})^m \approx 1 - 0 = 1$
- Thus, 2^R will almost always be around m


Prob(R>=r) for m=1024
Prob(R>=r)
r
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
5
10
15
20


Prob(R==r) for m=1024
Prob(R==r)
r
0.25
0.2
0.15
0.1
0.05
0
0
5
10
15
20

Why It Doesn't Work?

- 2^R *is always a power of 2 ….*
- Bad luck can result in huge errors…
- Workaround: run in parallel several copies of this algorithm, using different hash functions and average the results
- How do we average results?
  - MEAN? What if one very large value?
  - MEDIAN? All values are a power of 2!

Solution

- The Book:
  - Partition your hash functions into several groups.
  - Calculate the average of each group.
  - Then take the median of the averages.

- Does it really work?
- Check the original papers of Fajolet et al.
- Many variants (about 20), 1983-....

Durand, Flajolet: the LogLog algorithm

- Key ideas: *stochastic averaging + calibration*
  - Partition your samples into $n=2^l$ groups, using the first l bits of the hash function as a selector
  - Calculate $R_1, \dots, R_n$, (R_i for group i) and return: $a_n * n * 2^{\text{mean}(R_1, \dots, R_n)}$ where a_n="a correction factor" (pre-computed)
  - n=1024 (*10 extra bits!*) => relative error 3% -4%

Complexity

- Required memory
  - We work with bit strings of **length $\log_2 n$**
  - We only have to maintain
    "**the length of the longest tail of 0's**" - **$\log_2(\log_2 n)$**
    - almost nothing:
    - **8 bits**=>2^256 =10^77 objects
    - **5 bits**=>2^32=10^9 objects
  - Multiply it by the number of hash functions:
    - **1024*5=640 bytes of memory**

- Time
  - processing an element requires computing values of **ONE hash function** (linear in the length of input).

Summary (Durand&Flajolet, 2003):

The basic LogLog counting algorithm makes it possible to estimate cardinalities till ***10^8*** *with a standard error of* ***4%*** *using* ***1024 registers of 5 bits*** *each, that is, a table of* ***640 bytes in total****.*

```
ghfffghfghgghggggghghheehfhfhhgghghghhfgffffhhhiigfhhffgfiihfhhh
igigighfgihfffghigihghigfhhgeegeghgghhhgghhfhidiigihighihehhhfgg
hfgighigffghdieghhhggghhfghhfiiheffghghihifgggffihgihfggighgiiif
fjgfgjhhjiifhjgehgghfhhfhjhigggghghihigghhihihgiighgfhlgjfgjjjmfl
```

The LogLog Algorithm with $m = 256$ condenses the whole of Shakespeare's works to a table of 256 "small bytes" of 4 bits each. The estimate of the number of distinct words is here $n^\circ = 30897$ (true answer: $n = 28239$), i.e., a relative error of $+9.4\%$.

Recommended papers:

- P. Flajolet:
  Counting by Coin Tossings

- M. Durand and P. Flajolet:
  Loglog Counting of Large Cardinalities

- A. Metwally, D. Agrawal and A. El Abbadi:
  Why Go Logarithmic if We Can Go Linear?