Note to other teachers and users of these slides: We would be delighted if you found this our material useful in giving your own lectures. Feel free to use these slides verbatim, or to modify them to fit your own needs. If you make use of a significant portion of these slides in your own lecture, please include this message, or a link to our web site: http://www.mmds.org

Mining Data Streams (Part 2)

Lecturer: Andrea Clementi

University of Rome *Tor Vergata*

Note: These slides are an adaptation, with more details, of the slides by Jure Leskovec, Anand Rajaraman, Jeff Ullman available at http://www.mmds.org

Main Topics

- **Further Tasks on Data Streams:**
  - **(1) *Filtering* a data stream: *Bloom* filters**
    - Select elements with property **x** from Stream
  - **(2) *Counting* distinct elements: *Flajolet-Martin* Algorithm**
    - Number of distinct elements in the last **k** elements of the Stream
  - **(3) *Estimating* moments: *AMS* Algorithm**
    - Estimate Standard Deviation of last ***k*** elements of the Stream

(1) Filtering Data Streams

Filtering Data Streams

- **Stream Model:** Each element of data *Stream* ***X*** is a *tuple* **x**=<key_1,...,key_k>
- **Input:** Given a list of ``good'' key-values (items) ***S*** (*e.g.* a subset of good values for key_1)
- **Task:** Determine and filter those tuples **x** of **X** having its component key_1(x) in ***S***

- **Trivial solution:** Store all **S** in a Hash_Table **T** and, for each element **x, do**: *hash* key_1(x) and verify in **T**.
  - **Big Data:** Suppose there is no enough memory to store all of items in ***S*** in a Hash Table
    - E.g. we might be processing *millions* of filters on the same Stream

Applications

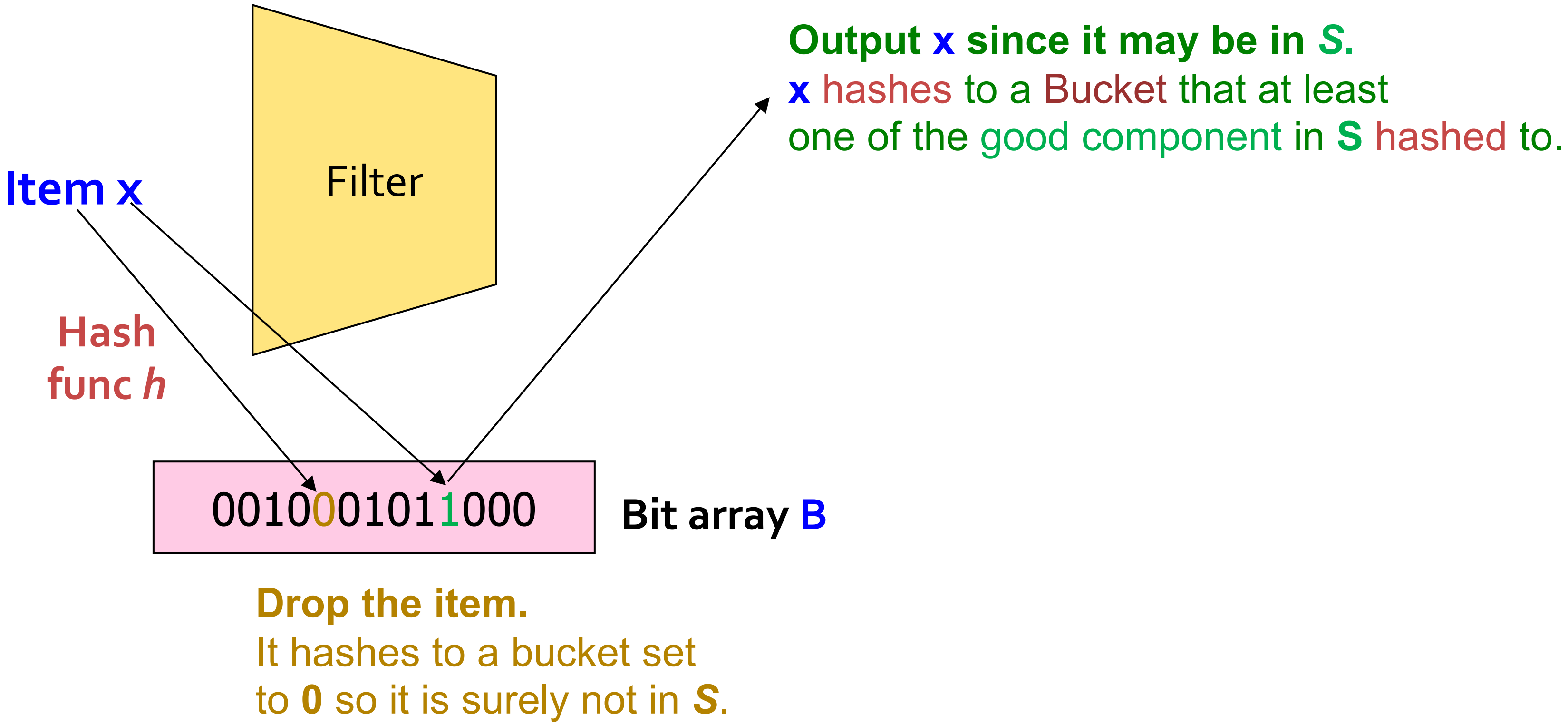
- **Example: Email spam filtering**
  - We know 1 billion "good" email addresses (key_1)
  - If an email comes from one of these, it is **NOT** *spam*

- **Publish-subscribe systems**
  - You are collecting lots of messages (news articles)
  - People express interest (only) in certain sets of keywords
  - Determine whether each message matches user's interest

First-Cut Algorithm

- **U** = {*all* values for key_1}; ***S*** = {good values for key_1}
- **Phase 1: Pre-processing**
  - **Create** an array **B[1...n]** of ***n*** bits, initially all *0*s
  - **Choose** a hash function ***h:*** **U → [n]**
  - **For each** component $s \in S$ **compute h**(s)
  - **Set B[h(s)] = 1**
- **Phase 2: Online** (Let **x**=<x_1,...,x_k> the new stream element)
  - **Compute h**(x_1)
  - **If B[h(**x_1**)] = 1** then **accept** stream element **x**

First-Cut Algorithm: Analysis



- Creates false positives but no false negatives
  - If the key-component of **x** is in ***S*** it surely accept it, if not we may still output it

First-Cut Algorithm: Analysis

- **|S| = m (# of good email addresses) e.g. 1 billion |B|= n (size of the hash-array) e.g. 1GB = 8 billion bits**

- **THM.** If the email address is in ***S*** ***(i.e.*** x_1 ∈ **S**), then it surely hashes to a bucket that has the bit set to **1**, so it always gets through (***no false negatives***)

- Approximately m/n = **1/8** of the bits are set to **1**, so about **1/8** (= Prob[collision]) of the addresses not in ***S*** get through to the output (***false positives***)
  - Actually, less than **1/8**, because more than one address might hash to the same bit

Analysis: Throwing Balls (1)

More accurate analysis for the number of false positives.

Assume |S|= m

The size of the Bucket B is |B|=n

- **OBS:** For some item **u** from **U**, the probability to hit a *full* bucket is exactly the fraction *ERR* of **full** buckets in **B.** To estimate *ERR*, we play a *balls-into-bins* process:

- If we throw ***m*** balls into ***n*** equally likely bins, **what is the probability that a bin gets at least one ball?**

- **In our case:**
  - **Bin** = buckets: 0,1,....,n-1
  - **Balls** = random hash values **h(Key_1(x))** of items in **S**: 1,2,...,m
  - **Pr[one ball hits one bin] = 1/n** (Property of Hash Function **h(..)**)

then.....

Analysis: Throwing Balls (2)

- We have *m* balls into *n* bins
- What is the probability that a bin gets **at least one** ball?
- **Pr[Err] = 1 - (1 - 1/n)^m ≃ 1 – e$^{-m/n}$** (using apx: $(1-x) \simeq e^{x}$)

- **Note.** for m << n , **Pr[Err] ≃ m/n**

Analysis: Throwing Balls (3)

- **Fraction of 1s (full Bins) in the array B =**
  = probability of false positive = $1 - e^{-m/n}$

- **Example:** $\mathbf{10^9}$ balls (|S|), $\mathbf{8 \cdot 10^9}$ Bins
  - Fraction of full bins in **B = $1 - e^{-1/8}$ = 0.1175**
    - Compare with our earlier estimate: **1/8 = 0.125**

Bloom Filter

- Consider: $|S| = m$, $|B| = n$
- Use k independent hash functions $h_1, \dots, h_k$
- **Initialization:**
  - Set **B** to all **0**s
  - Hash each element $s \in S$ using each hash function h_i, set $B[h_i(s)] = 1$ (for each $i = 1, .., k$)

(**note:** we have a single array B!)

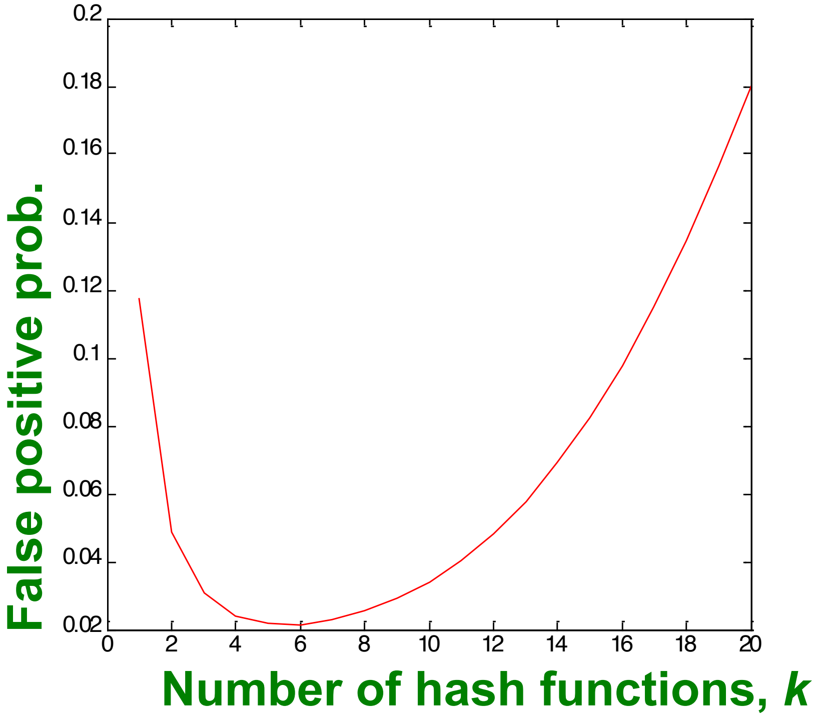
- **Run-time:**
  - When a stream element with key x arrives
    - If $B[h_i(x)] = 1$ for all $i = 1, \dots, k$ then declare that x is in S
      - That is, x hashes to a bucket set to **1** for every hash function $h_i(x)$
    - Otherwise discard the element x

Bloom Filter -- Analysis

- **What fraction of the bit vector B are 1s?**
  - Throwing $k \cdot m$ balls to n bins
  - So fraction of 1s in B is $(1 - e^{-km/n}) \simeq km/n$
- But we have k <u>mutually-independent</u> hash functions and we only let the element x through **if all** k hash element x to a bucket of value **1**
- So, false **positive probability** $= (1 - e^{-km/n})^k$

Bloom Filter – Analysis (2)

- ***m* = 1 billion, *n* = 8 billion**
  - **k = 1**: $(1 - e^{-1/8})$ = **0.1175**
  - **k = 2**: $(1 - e^{-1/4})^2$ = **0.0493**



- **What happens as we keep increasing *k*?**

- "Optimal" value of ***k***: ***n/m* ln(2)**
  - **In our case:** Optimal **k = 8 ln(2) = 5.54 ≈ 6**
    - **Error at k = 6**: $(1 - e^{-1/6})^2$ = **0.0235**

Bloom Filter: Wrap-up

- **Bloom filters guarantee no false negatives, and use limited memory**
  - Great for pre-processing before more expensive checks
- **Suitable for hardware implementation**
  - Hash function computations can be *parallelized*

- Is it better to have **1** big **B** or ***k*** small **B**s?
  - **It is "the same":** $(1 - e^{-km/n})^k$ vs. $(1 - e^{-m/(n/k)})^k$
  - But keeping **1 big B** is simpler

(2) Counting Distinct Elements

Counting Distinct Elements

- **Problem:**
  - Data stream consists of a sequence of elements chosen from a universe set **U** of size ***N***
  - Maintain a count **d** of the *number* of *distinct* elements seen so far in the STREA
- **Obvious approach:** $S \subseteq U \quad (|S| << |U|)$
  Maintain the *set* of elements seen so far
  - That is, keep a ***hash table*** of all the *distinct* elements seen so far

Applications

- **How many <u>different</u> words are found among the Web pages being crawled at a site?**
  - Unusually low or high numbers could indicate artificial pages (spam?)
- **How many <u>different</u> Web pages does each customer request in a week?**
- **How many <u>distinct</u> products have we sold in the last week?**

Using Small Storage

- **Real problem:** What if we do not have *space* (i.e. energy) to maintain the **set** of elements seen so far?
- **d = # of distinct elements seen so far** we only need the number!

1. **Goal:** Compute **d**
2. **Relaxation:** Accept that our **answer** for **d** may have a *little error*, but <u>limit</u> the <u>probability</u> that the error is <u>large</u>

(1)+(2) = Probabilistic Approximation Algorithms

Flajolet-Martin "*Magic*" Approach

- **(Pre-processing) Pick** a hash function ***h*** that maps each of the ***N*** elements of **U** to *at least* $\log_2 N$ bits: $U = [N]$
  $h: [N] \rightarrow \{0,1\}^s$, $s \geq \log N$ (bits)

- **For each** *stream element* ***a*****, Let** (compute h(a)) and:
  - **r(a)** = position of first **1** counting from the right in **h(a)**
    - E.g., say ***h(a) = 01100*** in binary, so ***r(a) = 3***

- **Store** of the STREAM
  - **Sketch R = max { r(a)**, over all the items ***a*** seen so far **}**

- **Magic THM: Return** $\boxed{m = 2^R}$ APX ***as the Estimated Value for*** **d** ***(# of distinct elements seen so far)***

Flajolet-Martin "*Magic*" Approach

Nice **Properties** of the **FM-Algorithm**:

- Repeated occurrences of the same element do not affect the value of **R:** element a will always hash to the same value **h(a)** (so gets the same r(a)).

- Sketch **R** can be easily combined with others: if we have a collection of Sketches R_1, R_2,..., R_k from different Streams and wish to compute **d** in the combined stream, we can simply take
  - $\text{Max}\{R_1, R_2, \ldots, R_k\}$ = MAX { max { r(a) }, — , }

Why It Works: Intuition

- Very very rough and heuristic intuition why **Flajolet-Martin Algorithm** works:
  - ***h(a)*** hashes ***a*** with **equal prob.** to any of the ***N*** slots
  - Then ***h(a)*** is a sequence of *rnd* **$\log_2 N$** bits, where 2^{-r} fraction of all ***a***s have a tail in *h(a)* of ***r*** zeros $= \frac{2^{5-r}}{2^{5}}$
    - About 50% of ***a***s hash to *****0**
    - About 25% of ***a***s hash to ****00**
    - So, if we saw the *longest* tail of ***R=2*** (i.e., item hash ending ***100**) then we have probably seen **about *4*** distinct items so far $4 = 2^R$
  - So, *in expectation*, it takes to hash about **d** = 2^R distinct items before we see one hash with zero-suffix of length ***R, i.e.,***
  - 2^R is the *expected number* of "trials" necessary and sufficient to see a sequence with zero-suffix of length ***R***

Flajolet-Martin "*Magic*" : Formal Analysis

- Assume **h**: $[N] \rightarrow \{0,1\}^s$, where $s \geq \log N$ (bits)
- Since **h** is a perfect hash function:
- **Fact 1.** For any a, it holds $\Pr(r(a) \geq r) = 1/2^r \Leftarrow \frac{2^{s-r}}{2^s}$
- Define *binary r.v.* $\mathbf{X_r = 1}$ *iff* $\exists$ a in Stream s.t. $r(a) \geq r$
- **Fact 2.** $\Pr(X_r = 1) = 1 - (1-2^{-r})^d$ and $\Pr[X_r = 0] = (1-2^{-r})^d$
- **Remark:** d = real # of distinct elements seen so far — OUR TARGET

Now, for the apx output $\mathbf{m = 2^R}$, we can compute the Error Probability: for any $c>0$,

- $\Pr(m > 2^c \cdot d) = \Pr(R > \overbrace{\log d + c}^{\to r}) = \Pr(X_{\overbrace{\log d + c}^{= r}} = 1) \leq d / 2^{\log d + c} = 2^{-c}$

  (we used ineq. $(1-x)^d \geq 1 - xd$ for "small" $|xd| < 1$)

- $\Pr(m < 2^c \cdot d) = \Pr(X_{\log d + c} = 0) \leq \exp(-2^{c-1})$

  (we used ineq. $(1-x) \leq \exp(-x)$ for "small" $|x| < 1$)

Example: for c=2, we get a *8-apx* with Prob $\geq 2/3$

For more detail, see file: Flaj_Martin_Algo_Analysis.pdf

Flajolet-Martin "*Magic*" : Space Complexity

- **Fact.** Space[F-M Algorithm] = O(loglog d), whp.

Proof. It is required to compute and store the quantity R.
From the previous analysis, it holds, whp,

$$\boxed{R \leq \log d + c}$$

Large Deviation and Amplification: Informal

- $E[2^R]$ is actually **infinite**
  - Probability halves when $R \rightarrow R+1$, but value doubles
- Workaround involves using *many* hash functions h_i and getting many sketches R_i
- How are samples R_i **combined**?

SAMPLES = sketch

  - **Average?** What if one very large value 2^{R_i}?
  - **Median?** All estimates are a *power* of **2**
  - **Solution in Pratice:**
    - Partition your samples into small groups
    - Take the **median** of each group
    - Then take the *average* of the **medians**

or VICEVERSA?

Improvements of FM approach

- In [Bar-Yossef, Jayram, Kumar, Sivakumar, Trevisan `04], the authors refine the **FM algorithm** to give:
- a **rnd (1- eps)-apx algorithm** for **d** using overall memory space:

$$\left[O(eps^{-2} * \log(1/eps)) + 2 \lg \lg n \right]$$

- See the file: Flaj_Martin_Algo_Analysis.pdf

(3) Computing Moments

Generalization: Moments

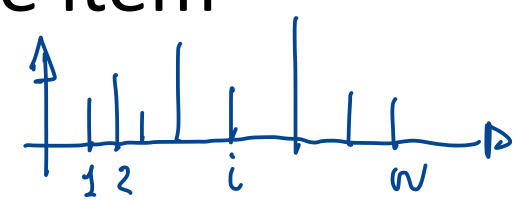
- Suppose a stream I has elements chosen from a set A of N values $A = \{1, 2, \ldots, i, \ldots, N\}$
- Let m_i be the number of times value i occurs in I FREQUENCE
- The k^{th} *moment* is

$$\sum_{i \in A} (m_i)^k$$

Special Cases

$$\sum_{i \in A} (m_i)^k$$

- **0th moment =** number of distinct elements
  - The problem just considered
- **1st moment = |*I*|** $\equiv$ n. of elements in I
  - Easy to compute
- **2nd moment =** ***surprise number S*** **=** a measure of how *uneven* the item **distribution** is



Example: Surprise Number

- **Stream of length 100**
- **11 distinct values**

- Item counts: **10, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9**
  Surprise *S* = 910 → LOWER VALUE
  FLAT DISTRIB.

- Item counts: **90, 1, 1, 1, 1, 1, 1, 1 ,1, 1, 1**
  Surprise *S* = 8,110 → HIGHER VALUE
  UNBIASED DISTRIB.

[Alon, Matias, and Szegedy]

The *AMS* Method: Informal

- *AMS* method works for *all* moments
- Gives an *unbiased* estimate
- **Today: evaluate the 2nd moment *S*** ≡ SURPRISING NUMBER
- **1st Step. pick** and **update** a sample of *i.i.d random*
- variables ***{X_j : j = 1...k}*** *defined as follows:*
  - For each variable ***X*** we store ***X.el*** and ***X.val***
    - ***X.el*** corresponds to some item ***i of I*** *(see later...)*
    - ***X.val*** corresponds to the **count** of *future* occurrences of ***i***
  - Note this requires a count in main memory, so number k of ***X***s should be limited

↳ # of rnd. VARIABLES

The AMS Algorithm: Details

- How to set ***X.val*** and ***X.el***? (valid for all **X**s)
  - **Input:** stream ***I = I[1,...,L]*** of length ***L*** (we relax this later)
  - **Pick** some **time step *t u.a.r.*** in $1 \le t < L$ → $t \in_U [L]$
  - ***Set X.el : = i, where i = I[t]*** ≡ i-th element in I
  - **Compute** counter ***X.val : = c*** as the number of ***i'***s occurrences in the *sub-stream* ***I[t,...,L]*** → FUTURE OCCS!
- **Output:** the estimate of the **2nd moment** $\sum_i m_i^2$ is

$$S \triangleq f(X) = L \cdot (2 \cdot \underbrace{c - 1}_{\text{counter!}})$$

Amplify Confid.

  - **Note**: The algorithm computes *multiple* **X**s: ($X_1, X_2, \ldots X_k$) and the final estimate will be $S = 1/k \sum_j^k f(X_j)$

The AMS Algorithm: Analysis

- Let *r.v.s* step **t**, element **i**, count **c**, and stream length ***L*** defined in the ***AMS*** Algorithm.
- $E(L \cdot (2c - 1)) = \sum_{i=1}^{L} L \cdot (2c(i) - 1) \cdot (1/L)$ $\left\{ \text{since } i = i[t] \text{ and } t \in_U [L] \right.$
  $= \sum_{i=1}^{L} \cancel{L \cdot} (2c(i) - 1)$ (*)

To evaluate (*), for any fixed $a \in A$, let $j(1),\ldots j(m_a)$ the occurrences of a in the stream ***I=I[1,…,L]**, then* :

$c(j(1)) = m_a$, $c(j(2)) = m_a - 1$, …. , $c(j(m_a)) = 1$. (**)

Rewriting (*) using (**) and grouping w.r.t. each $a \in A$:

$(*) = \sum_{a \in A} \sum_{z=1}^{m\_a} (2\underset{z}{\cancel{j}}-1)$ and $\sum_{z=1}^{m\_a} (2\underset{z}{\cancel{j}}-1) = (m_a)^2$ ✱

So, $\boldsymbol{E(L \cdot (2c-1)) = E(S) = \sum_i m_i^2}$ $\square$

$$\sum_{i=1}^{\ell} (2i-1) = \ell^2$$

PROOF (by ind. on ℓ). $\ell=1 \to 1=1$ ok! ; $\boxed{\displaystyle\sum_{i=1}^{\ell-1} (2i-1) \overset{\bullet}{=} (\ell-1)^2}$ Hynd. Hyp.

Now: $$\sum_{i=1}^{\ell-1} (2i-1) + 2\ell - 1 = (\ell-1)^2 + 2\ell - 1 =$$

$$= \ell^2 + \cancel{1} - \cancel{2\ell} + \cancel{2\ell} - \cancel{1} = \ell^2 \qquad \square$$

The AMS Algorithm: Analysis

EXAMPLE

Count: 1 2 $c_3 = m_b$ 3 m_a

Stream: a a b b b a b a

$n = L = |I|$

- **2nd moment is** $S = \sum_i m_i^2$
- c_t = occurrences of item a at time t appears from time **t** onwards ($c_1 = m_a$, $c_2 = m_a - 1$, $c_3 = m_b$)
- $E(S) = \frac{1}{n} \sum_{t=1}^{n} \boxed{n(2c_t - 1)}$

Pr = $\frac{1}{n}$

item at time t

$= \frac{1}{n} \sum_i n\,(1 + 3 + 5 + \cdots + 2m_i - 1)$

EXERCISE

m_i … total count of item i in the stream (we are assuming stream has length **n**)

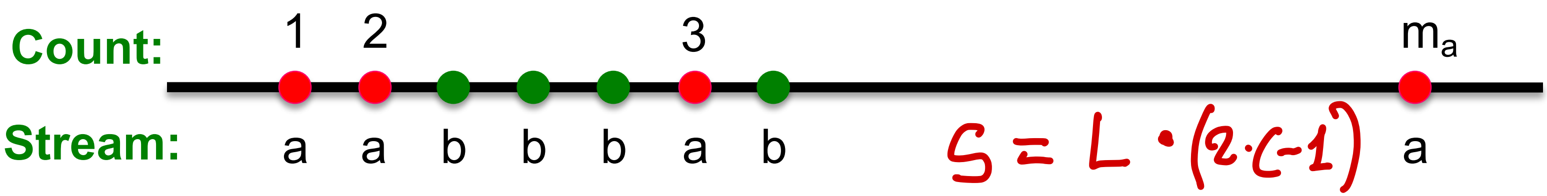
Group times by the value seen

Time t when the last i is seen (c_t=1)

Time t when the penultimate i is seen (c_t=2)

Time t when the first i is seen ($c_t = m_i$)

The AMS Algorithm: Analysis



- $E(S) = \frac{1}{n} \sum_i n \, (1 + 3 + 5 + \cdots + 2m_i - 1)$
  - Little side calculation: $(1 + 3 + 5 + \cdots + 2m_i - 1) = \sum_{i=1}^{m_i} (2i - 1) = 2\frac{m_i(m_i+1)}{2} - m_i = (m\_i)^2$
- **Then** $E(S = f(X)) = \frac{1}{n} \sum_i n \, (m_i)^2$
- **So,** $E(S) = \sum_i (m_i)^2$
- **We have the second moment (in expectation)!**

Higher-Order Moments

- **For estimating k^{th} moment we essentially use the same algorithm but change the estimate:**
  - For **k=2** we used ***n* (2·c – 1)** $\to f(c) \to E(f(c))$
  - For **k=3** we use: $\boxed{n\,(3 \cdot c^2 - 3c + 1)}$ (where **c=X.val**)
- **Why?**
  - **For k=2:** Remember we had $(1 + 3 + 5 + \cdots + 2m_i - 1)$ and we showed terms ***2c-1*** (for **c=1,…,m**) sum to m^2
    - $\sum_{c=1}^{m} 2c - 1 = \sum_{c=1}^{m} c^2 - \sum_{c=1}^{m} (c-1)^2 = m^2$
    - **So:** $\mathbf{2c - 1 = c^2 - (c-1)^2}$
  - **For k=3:** $\mathbf{c^3 - (c-1)^3 = 3c^2 - 3c + 1}$ $\to \sum_{c=1}^{m} 3c^2 - c + 1 = m^3$
- **Generally:** Estimate $= n\,(c^k - (c-1)^k)$

Combining Samples

- **In practice (to increase confidence):**
  - Compute $f(X) = n(2c - 1)$ for as many *mutually-indep.* variables X as you can fit in memory
  - *Average* them in groups
  - Take *Median* of Averages

USUAL AVERAGING

$\frac{\sum^{t} X_i}{t} \xrightarrow{t \to \infty} E(X)$

- **Problem: Streams never end**
  - We assumed input Stream ***I*** has finite size ***L***:

    I = I[1,…,L]

  - But real streams go on forever, so ***L*** is a variable – the number of inputs seen so far

Streams Never End: Fixups

- **(1)** The variables ***X*** have ***L*** as a factor – keep ***L*** separately; just hold the count in ***X***

$X \equiv L \cdot (2c - s)$

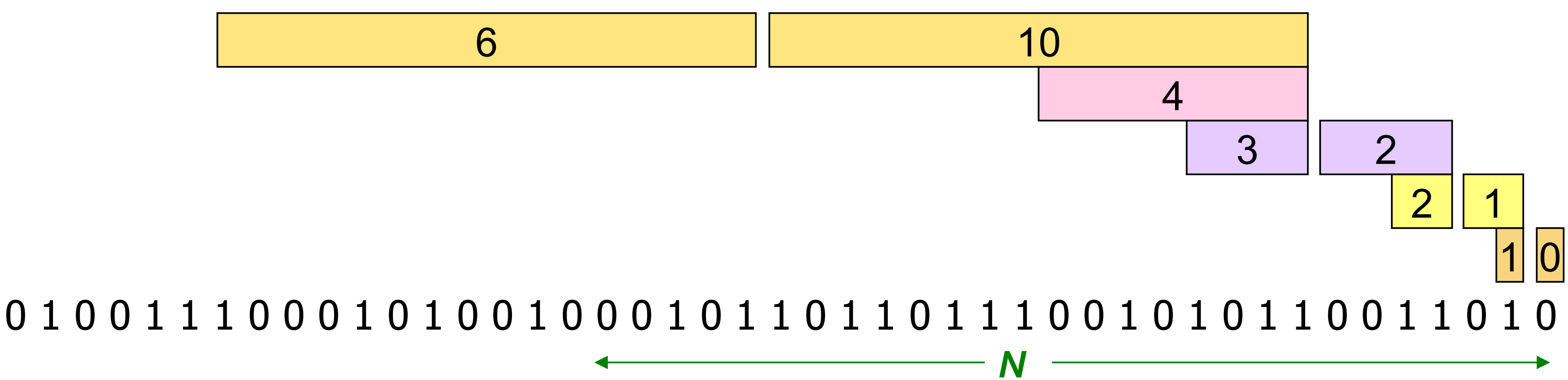
- **(2)** Suppose we can only store ***k*** counts. We must throw some ***X*s** out as time goes on:
  - **Objective:** Each starting time ***t*** is selected with probability ***k/L***
  - **Solution: (fixed-size sampling!** see previous slides**)** !!
    - Choose the first ***k*** time (i.e. elements) for ***k*** variables (deterministic)
    - When the ***L*** th element arrives (***L*** > ***k***), choose it with Prob = ***k/L***
    - If you choose it, throw one of the previously stored variables **X** out, ***u.a.r.***

RESERVOIR ALGO

Counting Itemsets

Counting Itemsets

- **New Problem:** **Given a stream, which items appear more than *s* times in the window?**
- **Possible solution:** Think of the stream of baskets as one binary stream per item
  - **1** = item present; **0** = not present
  - Use **DGIM** to estimate counts of **1**s for all items



Extensions

- **In principle, you could count frequent pairs or even larger sets the same way**
  - **One stream per itemset**

for example: for each pair of items (x,y), define the Binary Stream:
I(x,y) = 010...

- **Drawbacks:**
  - Only approximate
  - **Number of itemsets is way too big**

↳ it increases exponentially with itemset sizes

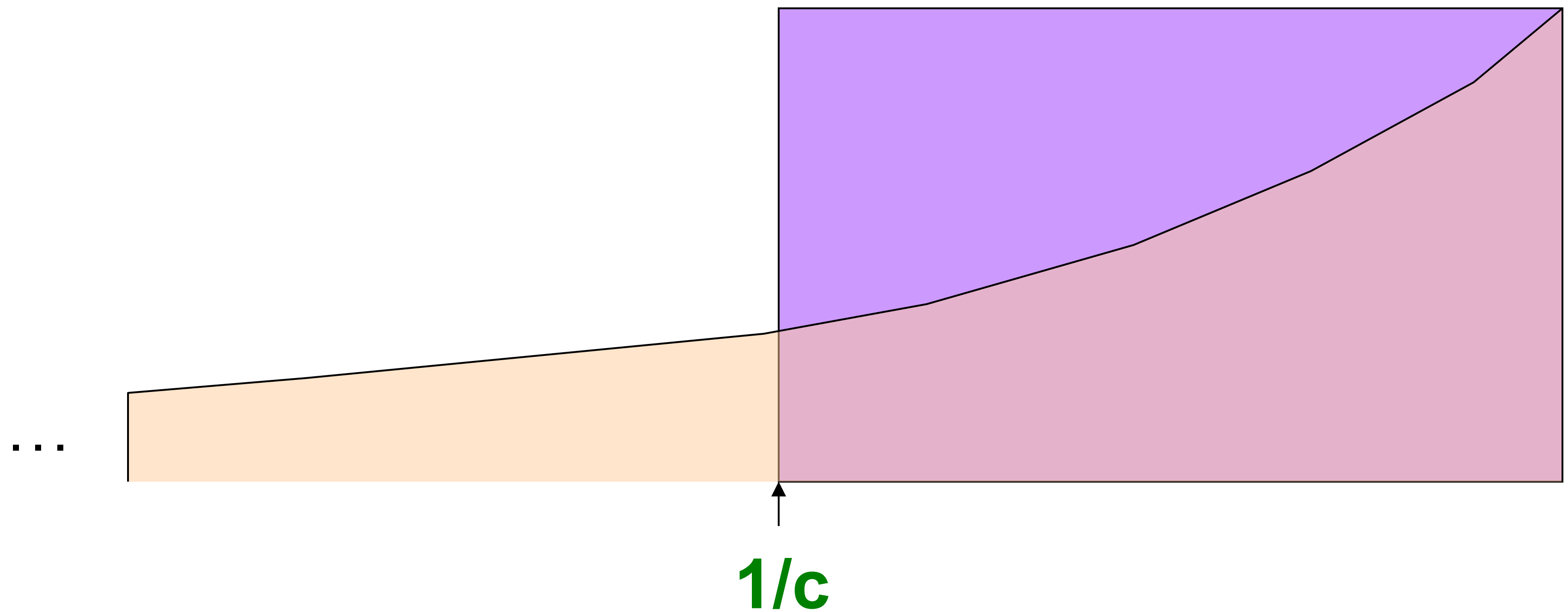
Exponentially Decaying Windows

- **Exponentially decaying windows: A heuristic for selecting likely frequent item(sets)**
  - **What are "currently" most popular movies?**
    - Instead of computing the raw count in last ***N*** elements
    - Compute a **smooth aggregation** over the whole stream
- If stream is $\boldsymbol{a_1, a_2, \ldots}$ and we are taking the sum of the stream, take the answer at time ***t*** to be: $= \sum_{i=1}^{t} a_i (1-c)^{t-i}$
  - **c** is a constant, presumably tiny, like $\mathbf{10^{-6}}$ or $\mathbf{10^{-9}}$
- **When new a_{t+1} arrives:** Multiply current sum by **(1-c)** and add $\mathbf{a_{t+1}}$

Example: Counting Items

- If each ***a*** is an "item" we can compute the **characteristic function** of each possible item ***x*** as an Exponentially Decaying Window
  - That is: $\sum_{i=1}^{t} \delta_i \cdot (1-c)^{t-i}$ where $\boldsymbol{\delta_i=1}$ if $\mathbf{a_i=x}$, and **0** otherwise
  - Imagine that for each item ***x*** we have a binary stream (**1** if ***x*** appears, **0** if ***x*** does not appear)
  - New item ***x*** arrives:
    - Multiply all counts by **(1-c)**
    - Add **+1** to count for element ***x***
- **Call this sum the "weight" of item *x***

Sliding Versus Decaying Windows



- **Important property:** Sum over all weights $\sum_t (1-c)^t$ is 1/[1 – (1 – c)] = 1/c

Example: Counting Items

- **What are "currently" most popular movies?**
- **Suppose we want to find movies of weight > ½**
  - **Important property:** Sum over all weights $\sum_t (1-c)^t$ is 1/[1 – (1 – c)] = **1/c**
- **Thus:**
  - There cannot be more than **2/c** movies with weight of ½ or more
- So, **2/c** is a limit on the number of movies being counted at any time

Extension to Itemsets

- **Count (some) itemsets in an E.D.W.**
  - **What are currently "hot" itemsets?**
    - **Problem:** Too many itemsets to keep counts of all of them in memory
- **When a basket B comes in:**
  - Multiply all counts by **(1-c)**
  - For uncounted items in **B**, create new count
  - Add **1** to count of any item in **B** and to any **itemset** contained in **B** that is already being counted
  - **Drop counts < ½**
  - Initiate new counts (next slide)

Initiation of New Counts

- Start a count for an itemset $S \subseteq B$ if every proper subset of ***S*** had a count prior to arrival of basket ***B***
  - **Intuitively:** If all subsets of **S** are being counted this means they are "**frequent/hot**" and thus ***S*** has a potential to be "**hot**"
- **Example:**
  - Start counting ***S*={i, j}** iff both **i** and **j** were counted prior to seeing ***B***
  - Start counting ***S*={i, j, k}** iff **{i, j}**, **{i, k}**, and **{j, k}** were all counted prior to seeing ***B***

How many counts do we need?

- Counts for single items **< (2/c)·(avg. number of items in a basket)**
- **Counts for larger itemsets = ??**
- **But we are conservative about starting counts of large sets**
  - If we counted every set we saw, one basket of **20** items would initiate **1M** counts