Mining Data Stream: Part 1

Lecturer: Andrea Clementi

University of Rome *Tor Vergata*

clementi@mat.uniroma2.it

These slides are a slight adaptation, with more technical details, of the book and the slides by Jure Leskovec, Anand Rajaraman, Jeff Ullman available at http://www.mmds.org

The Framework: Infinite Data

High dim. data	Graph data	Infinite data	Machine learning	Apps
Locality sensitive hashing	PageRank, SimRank	Filtering data streams	SVM	Recommender systems
Clustering	Community Detection	Queries on Streams	Decision Trees	Association Rules
Dimensionality reduction	Spam Detection	Web advertising	Perceptron, kNN	Duplicate document detection

Data Streams

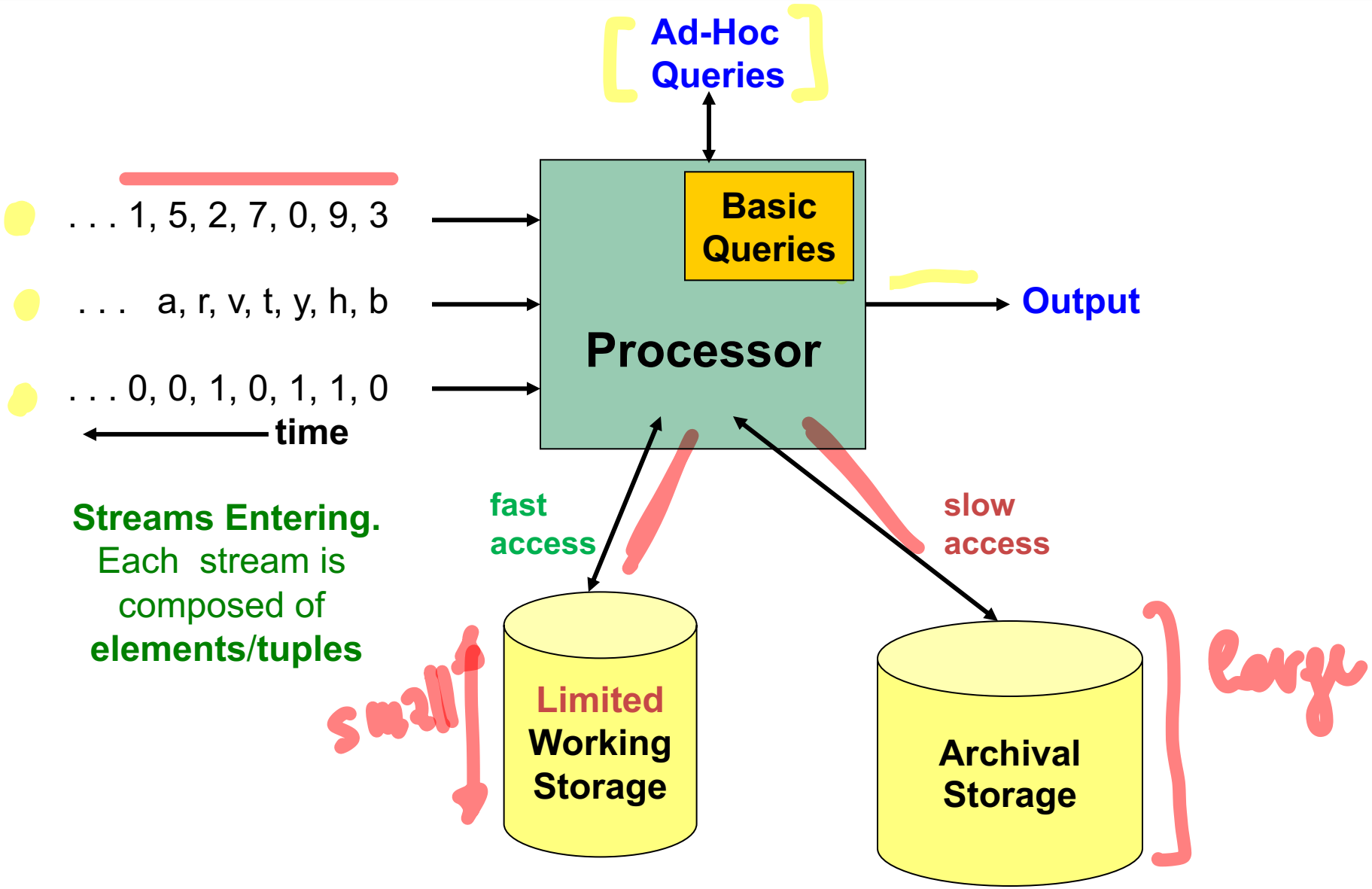
- In many **Data-Mining** situations, we do not know the *entire* Data Set *in advance*

- **Stream Management** is important when the input rate is controlled **externally:**
  - Search-Engines queries
  - Twitter or Facebook status updates
- We can think of the **data** as ***infinite*** and ***non-stationary*** (the distribution changes over time)

The Stream Model

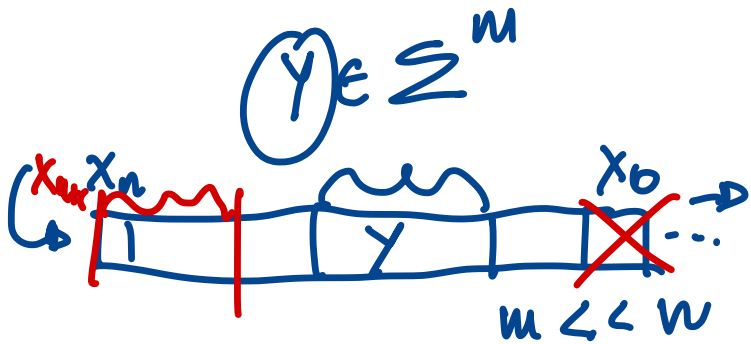
- Input **elements** enter at a rapid rate, at one or more input ports (i.e., **streams**)
  - **elements** of the stream ≡ **tuples** $\langle x_{1,1}, x_{2,1}, \ldots, x_K \rangle$
- The system cannot store the entire stream accessibly.
- Only short *sketches* of the stream can be maintained and updated
- **Q:** How do you make critical calculations about the stream using a *limited amount of (secondary) memory*?

General Stream Processing Model



Problems on Data Streams

- Types of queries one wants to answer on a data stream:
  - **Sampling** data from a Stream
    - Construct a ***random*** sample
  - **Queries** over ***sliding*** windows
    - Pattern Matching
    - Number of items of *type x* in the last *k* elements of the Stream



Problems on Data Streams

- Types of **queries** on a data stream:
  - Filtering a data stream
    - Select elements with property *x* from the stream
  - Counting *distinct* elements
    - Number of distinct elements in the last *k* elements of the stream
  - Statistics: Estimating ***Moments***
    - Estimate *avg./std. dev*. of last *k* elements
  - Finding *frequent* elements

Applications (1)

- Mining query streams
  - Google wants to know what queries are more frequent today than yesterday
- Mining click streams
  - Yahoo wants to know which of its pages are getting an unusual number of hits in the past hour
- Mining social network news feeds
  - E.g., look for trending topics on Twitter, Facebook

Applications (2)

- Sensor Networks
  - Many sensors feeding into a central controller
- Telephone call records
  - Data feeds into customer bills as well as settlements between telephone companies
- IP packets monitored at a switch
  - Gather information for optimal routing
  - Detect denial-of-service attacks

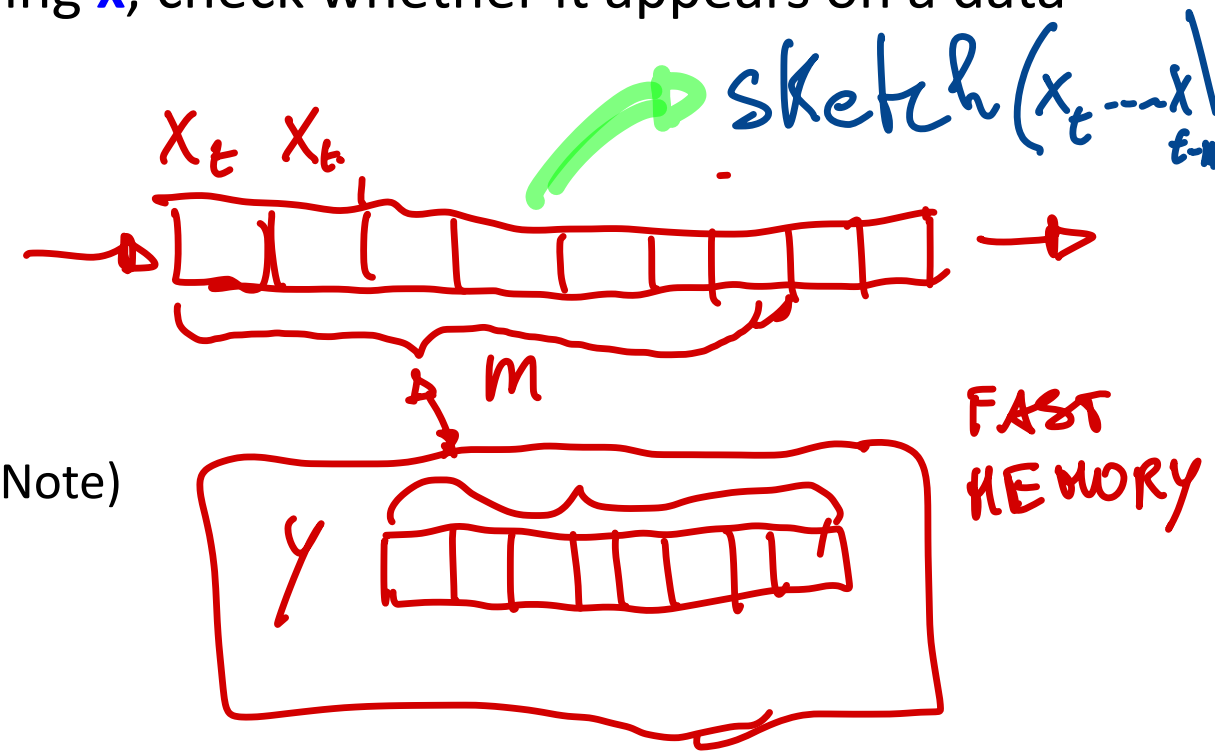
Mining Data Streams: Pattern Matching

Pattern Matching Problem: *Karp-Rabin's Algorithm*

Problem: Given a string **x**, check whether it appears on a data stream ***S***

$x_i \in \Sigma$

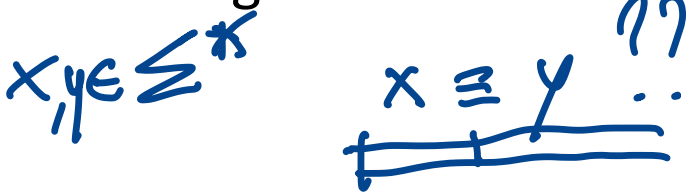
$y \in \Sigma^n \quad n << m$



(See **Section 4 of** Prezza's Note)

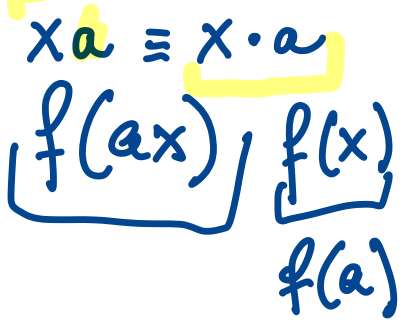
Pattern Matching (PM)

- To solve efficiently ***PM*** on Data Stream we need good *Data Sketches* (for *Identity Test*):



Def. (Data Sketches). Let $x \in \Sigma^*$, a data sketch is a rnd function $f: x \in \Sigma^* \rightarrow \{0,1\}^k$, having the following properties:

- $k << |x|$ (the sketch must be much shorter than the input);
- f must be easy to *update* and to *combine*: $f(xa) = F(f(x), f(a))$
- f must preserve *key properties** (*whp*) of x



Examples of Data Sketches:

* **Key Property:** ***Set Jaccard Similarity*** → **Min-Hashing**

Sketches for Pattern Matching (PM)

The Key Property for ***PM*** is *String Identity*

Given **x,y ∈ Σ*,** we want sketch **f(x)** s.t. **f(x) = f(y) iff x = y (whp)**

Setting: **Σ = [s] = {0,1,...,s-1}** and |x|=|y|=n. So,

x = <x[0],x[1],...,x[n-1]> **∈ Σ^n**

Wrong Idea: consider the string **x** as a *number* of n digits and then use **hashing** f(x) = ((**a**x + **b**) Mod M) mod d (d = size of the Hash table)

This would require M > x (x as a value, not as its length!!), so *mod* arithmetic with a number of n digits → **space = Ω(n)!!!**

Sketches of String Identity: Rabin's Hash Functions

- **Def.** Fix *prime* number $q > s$, and pick *u.a.r.* $z \in [q]$. Let $\mathbf{x} = \langle x[0], x[1], \ldots, x[s-1] \rangle \in [s]^n$. Then, *Rabin Hash function* is:

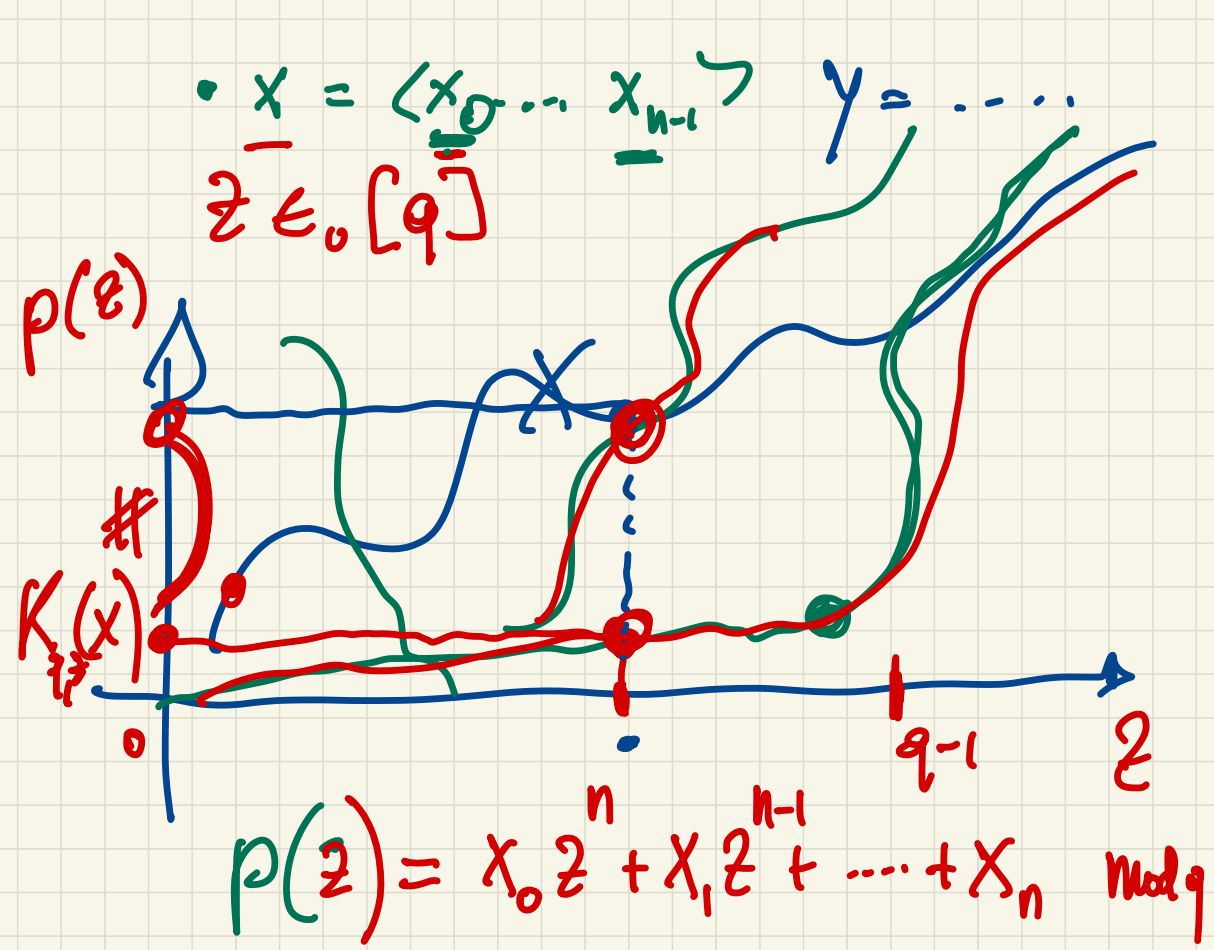
$$K_{q,z}(\mathbf{x}) = \left(\sum_{i=0,\ldots,n-1} x[n-1] * z^i \right) \bmod q$$

Lemma 3.2 (easy to update). Let $c \in \Sigma = [s]$. Then

$$K_{q,z}(x \circ c) := (K_{q,z}(x) * z + c) \bmod q$$

Proof. Simple algebra of polynomial in field Z_q.

- Updating Time/Space = O(log q)
- Homework: How to eff. compute $K_{q,z}(x\ y)$?
  hint: Use ***Exponentiation***



$$p(z) = x_0 z^n + x_1 z^{n-1} + \dots + x_n \mod q$$

Rabin's Hash Function

- **Algorithm *A*** for $K_{q,z}(x) = (\Sigma_{i=0,..,n-1}\ x[n-1] * z^i) \bmod q$
  - **Input:** q prime, z u.a.r. in [q]; $\mathbf{x} = \langle x[0],x[1],...,x[n-1]\rangle \in [s]^n$
  - $K_{q,z}(\varepsilon) := 0$;
  - **For j:=1 to n do** $(K_{q,z}(\langle x[0],x[1],...,x[j-1]\rangle)\ z + x[j-1]) \bmod q$

- **Lemma 3.3. Algorithm *A'*** for $K_{q,z}(x \cdot y)$:
  - $K_{q,z}(x\ y) := (K_{q,z}(x)\ z^{|y|} * + K_{q,z}(y)) \bmod q$

Notes:

- **Space** (*A*(q,z,x)) = O(log q)
- **Space** (*A'*(q,z,x,y)) = O(log q + log|y| log q)
- * $z^{|y|}$ can be *pre-computed,* using exponentiation!

Rabin's Hash Function

- **Lemma 3.4 (Good Sketch for Identity).**

Let $x \neq y$ with $|x| = |y| = n$. Then:

$$\Pr[K_{q,z}(x) = K_{q,z}(y) \bmod q] \leq n/q \quad [1]$$

Proof. Notice that

$$\Pr[K_{q,z}(x) = K_{q,z}(y) \bmod q] = \Pr[K_{q,z}(x) - K_{q,z}(y) \equiv 0 \bmod q],$$

where $K_{q,z}(x) - K_{q,z}(y)$ is a polynomial.

Define $x\text{-}y = \langle x(n-1)-y(n-1),\ldots,x(0)-y(0)\rangle$, then it holds that

$K_{q,z}(x) - K_{q,z}(y) \equiv K_{q,z}(x - y) \bmod q$, and $K_{q,z}(x - y)$ has *degree* $\leq n$.

Since the number of **roots of** $K_{q,z}(x - y)$ is at most n, we have [1]

Rabin's Hash Function

- **Corollary 3.4.1 [Error probability].** Let q a *prime* such that $n^{b+1} \le q \le 2n^{b+1}$ for any constant $b>0$. Then, for any $x \neq y$,

w.hp. $\Pr[K_{q,z}(x) = K_{q,z}(y) \bmod q] \le 1/n^b$

$q = \Theta(u^b)$

Note. $\text{Space}(K_{q,z}(x)) = O(\log n)$, So the **sketch** compresses **from n to log n !!!**

$\lg q = \Theta(\lg u)$

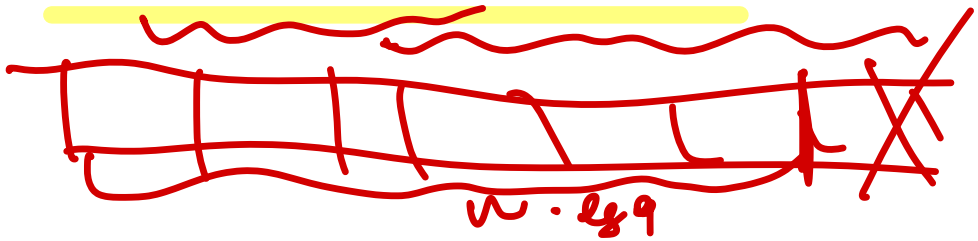
Mining Data Streams: Pattern Matching

Pattern Matching Problem: *Karp-Rabin's Algorithm*

Problem: Given a pattern (i.e. string) **y**, count *how many times* appears on the Data Stream (with alphabet $\Sigma = [s] = \{0,1,\ldots,s-1\}$).

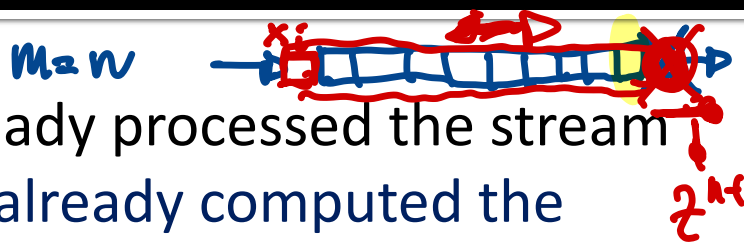
The (current) Stream is a string $\mathbf{x} = \langle x[1],x[1],\ldots,x[m] \rangle \in [s]^m$ and the pattern is $\mathbf{y} = \langle y[1],x[2],\ldots,x[n] \rangle \in [s]^n$ with $\mathbf{m >> n}$

Karp-Rabin's Algorithm uses the Rabin's Sketch for Identity →



(See **Section 4 of** Prezza's Note)

Karp-Rabin's Algorithm



Inductive Argument: Assume we've already processed the stream x[1], x[2],..., x[i], with $i \geq n$, and we have already computed the **sketches**:

- **(i)** $K_{q,z}(\langle x[i-n+1], x[i-n+2], \ldots, x[i] \rangle)$ and **(ii)** $K_{q,z}(y)$

1). compare them and update the output ***counter***.

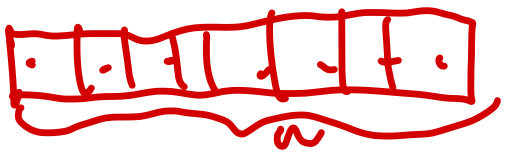
2). *A new Stream element* x[i] arrives, then update:

- $K_{q,z}(\langle x[i-n+1], x[i-n+2], \ldots, x[i] \rangle)$ as follows:

$K_{q,z}(\langle x[i-n+2], x[i-n+2], \ldots, x[i+1] \rangle) :=$

$((K_{q,z}(\langle x[i-n+1], x[i-n+2], \ldots, x[i] \rangle) - x[i-n+1] \cdot z^{n-1}) \cdot z + x[i+1]) \bmod q$

3). Go to Step 1....



Karp-Rabin's Algorithm

Performance analysis (over the Stream).

Lemma 1 (Space). Stream Length m ; Pattern Length n. Then

- **Space(KA(m,n) = $\Theta(n)$)** ($\Theta(\lg q) \rightarrow \Theta(1)$)

Proof. To apply **Step 2**, we need to keep the last **n** items of the Stream

Lemma 2 (Time). The number of operations per item is:

Time (KA(m,n) = $\Theta(1)$)

Proof. To compute **Step 2 in constant time**, it suffices to *pre-compute* the quantity $z^{n-1} \bmod q$

non dip da m !!

Improvement of KR"s Algorithm

- but also **n (<<m)** **could be very large!....**
- **Porat-Porat's Algorithm** (2009) solves the ***PM*** problem on the Stream Model with O(log n) space !!
- Very complex Algorithm and Analysis
  (for Laurea Magistrale and PhD Students)!!!

See Section 4.1.2 of Prezza's Note.

Sampling from a Data Stream: Sampling a fixed proportion

Note: As the stream grows the sample also gets bigger

Sampling from a Data Stream: (FAI: Secs 4.2.1-2-3)

- Since **we cannot store the entire stream**, one obvious approach is to store a ***sample***
- **Two different problems:**
  - **(1)** Sample a fixed proportion of elements in the stream (say 1 in 10)
  - **(2)** Maintain a *random* **sample** *of fixed size* over a potentially *infinite* stream
    - At any time step ***k,*** we want a random **sample** of ***s*** elements
      - **What is the property of the sample we want to maintain?** For every time step ***k***, each of the ***k*** elements seen so far has *equal probability* of being sampled

Sampling from a Data Stream: Statistics + Computer Science

1. **Target Dynamic Property of the sample:** For *every* time steps ***k***, each of the ***k*** elements seen so far has *equal probability* of being sampled.
2. **Efficiency:** fast updating, small memory space.

Note:

(1) AND (2) form the typical scenario of ***Data Science***

Sampling a Fixed Proportion

- **Problem 1: Sampling fixed proportion**
- **App Scenario:** *Search Engine Query Stream*
  - **Input Stream U** of tuples: (user ID, **query**, time)
  - **Typical (informal) Question:** How often did a user run the **same query** in a single day (can store only 10% of the full daily query stream).
  - **Algorithmic Task:** Find a Sample $S \subseteq U$ that, for the *typical* (i.e. *average*) user ***u***, for each query ***q***, well approximates (in expectation), the *fraction* of ***q-occurrences*** in **U** made by ***u***
  - **Def.** ***q-occurrence*** = *any* tuple in **U** having the component **query** = **q**

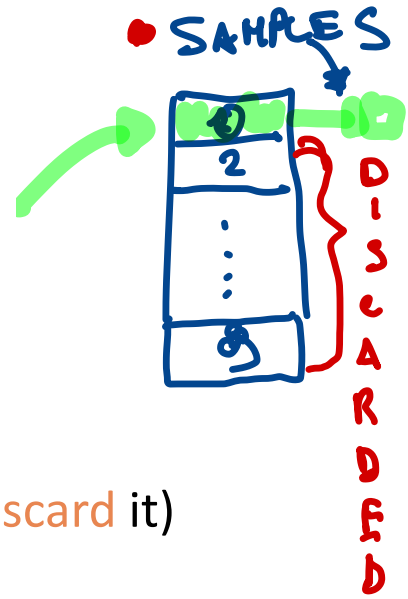
RND VAR. ✗ !

→ deterministic query

→ <User ID, q, time>

Problem 1: Sampling fixed proportion

- ***Naïve* rnd algorithm** N-Algo: (WRONG)
  - **S = Ø**
  - For each **tuple =** (user ID, query, time) **∈ U**
    - Pick (hash to) *i.u.a.r.* **z ∈ [10]** ** (10 Buckets)
    - Store (user, query, time) in ***S*** if **z =0** ** (o.w. discard it)
  - For each **query q**, return (average) values of the q-**occurrence** fractions computed (only) over **S**



Recall: [n] ≡ {0,1,2,…,n-1}

Analysis of Naïve Algorithm

- **Simple Homogeneous Case:** Suppose Stream **U** is such that *each* user issues ***m*** different **queries** once and ***d*** different **queries** twice (so in **U** there are ***m+2d*** **q/q-occurrences** *per* user)

- The N-Algo-sample **S** should (also) *well-apx* the fraction of those *average-user* queries having two occurrences in **U**
  - **Correct Answer:** ***d/(m+d)***
  - **Fact 1:** **E(|S|) = (1/10) · |U|** **(ok!)** but....
  - **S** will contain ***m*/10** of the *singleton* **queries** (ok!) and ***(19*/100)· *d*** of the *duplicate* **queries** at least *once (in expectation)*
  - But only ***d*/100 pairs** of the *duplicates queries*: **1/10 · 1/10 · d**
  - Of the ***d*** *duplicates*, ***18d/100*** appear exactly *once in* ***S***:
    - **((1/10 · 9/10)+(9/10 · 1/10)) · d**
  - N-Algo-sample **S average apx is:** $\frac{\frac{d}{100}}{\frac{m}{10}+\frac{d}{100}+\frac{18d}{100}} = \frac{d}{10m+19d}$

Exp # of **distinct** items in **S** = m/10 + d/100 + 18d/100 ; E(|**S**|) = m/10 + **2** (d/100) + 18d/100

Ex // STREAM = $\langle q_1, q_3, d_2^1, q_4, d_1^1, q_2, q_5, q_6, d_2^2, d_1^2 \rangle$ (1 USER)

Queries $q_1, \dots, q_m$, $m = 6$

queries twice $d_1, \dots, d_d$, $d = 2$

- Exact VALUE of the fraction of queries $d_1 \cup d_2$: is $\frac{2}{6+2} = \frac{m}{m+d}$

- Apx value of the fraction of $\cup$ $\cup$ computed in sample S (in Expectation) is

$$\frac{2/100}{\frac{6}{10} + \frac{2}{100} + \frac{18 \cdot 2}{100}}$$

we have 2 occ.

$$E(|S|) = \frac{m+2d}{10} \Leftarrow \frac{m}{10} + \frac{2}{100} \cdot d + \frac{18}{100} \cdot d =$$

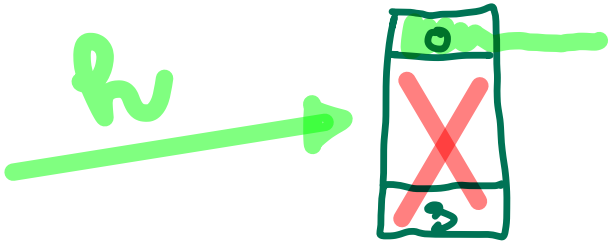
$$= \frac{m}{10} + \frac{20}{100} \cdot d$$

The Right Solution: Do Sample Users

User-Sample Algo: (***I*** ≐ Set of all User IDs)

Hyp: Known in adv.

- Pick **1/10th** of **users** and take ***all*** their **q-occurences** in the sample ***S:***
  - Use a hash function that hashes the *user IDs* into 10 Buckets **i.u.a.r.**
  - Select all **q-occurrences** from IDs hashed to Bucket n. **1** and store them into **S**
- Make average computations (statistics) on ***S***



- **Excercise:** What if ***I*** is not known *in advance*?

(Hints: do user-ID hashing *on-line*, during streaming phase)

User-Sample Algo: Analysis

- Q ≐ **Query** Set; *I* ≐ **User** ID Set
- x(v,q) ≐ # of q-occurencies by ID v in the Input Stream
- $X_S(V,q)$ ≐ # of q-occurencies by rnd ID *V* in **S** **(this is a r.v.!)**
  - ↳ RND VAR $V \in_U I$
- In **User-Sample Algo**, the ID *V's* are chosen *i.u.a.r.* →

- (*) $AVG_{v \in I}(x(v,q)) = \mathbf{E}_{V \in S}(X_S(V,q))$, *for each fixed* ***q***
- ***Note: (*) is the Target!***

- **Hence:**
- $AVG_{V \in S}(X_S(V,q))$ → $AVG_{v \in I}(x(v,q))$ for $|\{\#users\ in\ S\}| \to |I|$

Thm (lim)

U-S Algo Output — **Target Value**

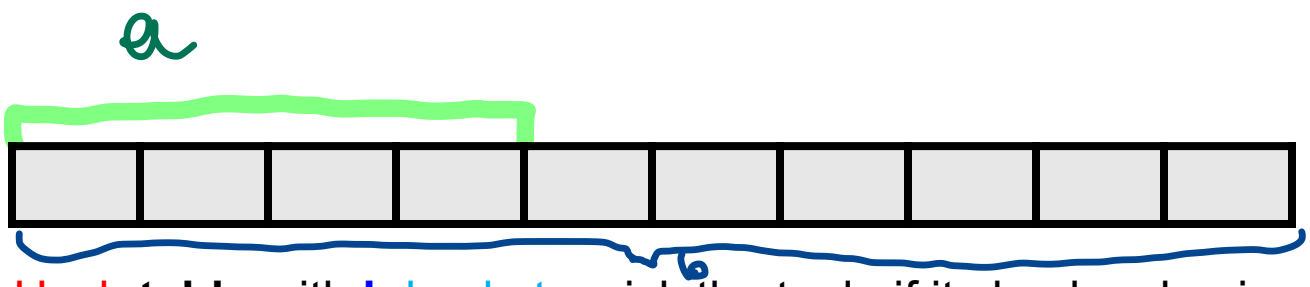
User-Sample Algo: Analysis

- **Remark:** The *expected* size of ***S*** is **1/10** of the Input Stream **U** *of all items*
  - Let $C_j = 1$ if user j is hashed to 1 (thus selected in ***S***) and $= 0$ *o.w.* Let x_j = # items of user j in **U**. Then
  - $\mathbf{E}(|S|) = \sum_j x_j \cdot \Pr(C_j = 1) = (1/10) \cdot \sum_j x_j = |U|/10$
- But **variance** can be very large!!!
  - Apply **User-Sample Algo** to any input stream **U** with one user j_1 having **90%** of items and another user j_2 having the other items.

Generalized Solution

- **Stream** of **tuples** ($X_1, X_2, \ldots, X_k$)
  - A **Key** is some *subset* ($X_{i1}, X_{i2}, \ldots, X_{ij}$) of each tuple's components
    - e.g., tuple is (**user**, search, time); **Key** is **user**
  - Choice of **key** depends on application

- To get a sample ***S*** of ***a/b*** fraction of the (Stream) User:
  - Hash each **tuple's key** i.u.a.r. into ***b* Buckets**
  - Insert in ***S*** the **tuple** iff its hash value (**Bucket**) is at most ***a***



Ex. Hash **table** with **b** buckets, pick the tuple if its hash value is at most **a.**
How to generate a 30% user sample?
Hash into b=10 buckets, take the tuple if it hashes to one of the first 3 buckets

Dynamic reduction of the Sample Size

- If the Input Stream **U** increases, it might not be possible to store 10% of **|U|**.

Practical Solution:

1. Hash each user ID to a large number **b** of Buckets
2. Set an initial *threshold* **t** ≤ b (all data of IDs hashed to Buckets ≤ **t** will be included in **S**).
3. If |S| gets *too* large, reduce **t→t'** (with **t' << t**) and remove *all* items of users hashed to Buckets > **t'**
4. Repeat Step **3** if necessary

Sampling from a Data Stream: Sampling a fixed-size sample

Note: As the stream grows, the sample is of fixed size

Maintaining a fixed-size sample: (See Sec. 4.5.5)

- **Problem 2: Fixed-size Sample (See Sec. 4.5.5)**
- Suppose we need to maintain a *random* sample ***S*** of size exactly ***s*** items(tuples)
  - E.g. constraint on the main-memory size
- Why? Don't know length of stream in advance: *Infinite Streams*
- Suppose at time ***n*** we have seen ***n*** items
  - Each item (tuples) must be in the current **sample *S*** with equal probability ***s/n***

How to think about the problem: say s = 2

Stream: a x c y z k c d e g…

At time **n= 5,** each of the first 5 tuples is included in ***S*** with equal prob.

At time **n= 7,** each of the first 7 tuples is included in ***S*** with equal prob.

Note: Impractical trivial solution = store all the ***n*** tuples seen so far and out of them pick *s* ***i.u.a.r***

Solution: Fixed Size Sample

- **Algorithm** ***Reservoir Sampling RS(s)***

1) Store all the first ***s*** items of the stream to ***S***

2) Suppose we have seen ***n-1*** items, and at time step **n,** the ***n^th^*** item arrives (***n > s***):

   1. With probability ***s/n***, ***n^th^*** item → ***S*** , else discard it
   2. If we picked the ***n^th^*** element, then it replaces one of the ***s*** items in the sample ***S***, picked ***u.a.r.*** (i.e. Pr = 1/s)

- **Claim:** For any s≥ 1, for any time n ≥ s , the ***RS(s)***-sample ***S*** has the desired property:

  - After ***n*** time steps, ***S*** contains each of the ***n*** elements *so far* with probability ***s/n***

Proof of Claim I

- **Base case:**
  - After we see **n=s** items the sample ***S*** has the desired property
    - Each out of **n=s** elements is in ***S*** with probability ***s/s = 1***
- **Inductive Step:**
  - Assume that after ***n*** time steps, ***S*** contains each item seen so far with probability ***s/n***
  - **Claim I**: After seeing item ***n+1***, ***S*** maintains the property

    i.e. ***S*** contains each element seen so far with probability ***s/(n+1)***

INITIAL DET. PHASE

Proof: By Induction

- **Inductive hypothesis:** After ***n*** items, sample ***S*** contains each item *seen so far* with Prob = ***s/n***
- **Now element *n+1* arrives:** it will be in ***S*** with prob. = ***s/(n+1)***
- For each *old* item **x** *already* in ***S***, Prob that the algorithm keeps it in ***S*** is:

$$\left(1-\frac{s}{n+1}\right)+\left(\frac{s}{n+1}\right)\left(\frac{s-1}{s}\right)=\frac{n}{n+1}$$

$\mathcal{E}_1$ ≡ "Item **n+1** discarded"

$\mathcal{E}_{21}$: Item **n+1** not discarded

$\mathcal{E}_{22}$: Item x in the sample not removed

- So, at time ***n***, every item in ***S*** were there with prob. = **s/n**
- At time ***n+1***, *each* *old* item **x** stays in ***S*** with prob. **n/(n+1)**
- So: Prob{ old item x is in ***S*** at time ***n+1***} $= \frac{s}{n} \cdot \frac{n}{n+1} = \frac{s}{n+1}$

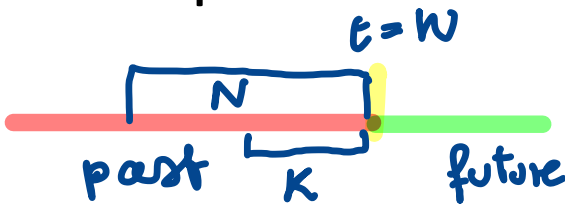
was there before n+1 & keep it at time n+1

STEP ↓.1

Queries over a (long) Sliding Window

Sliding Windows

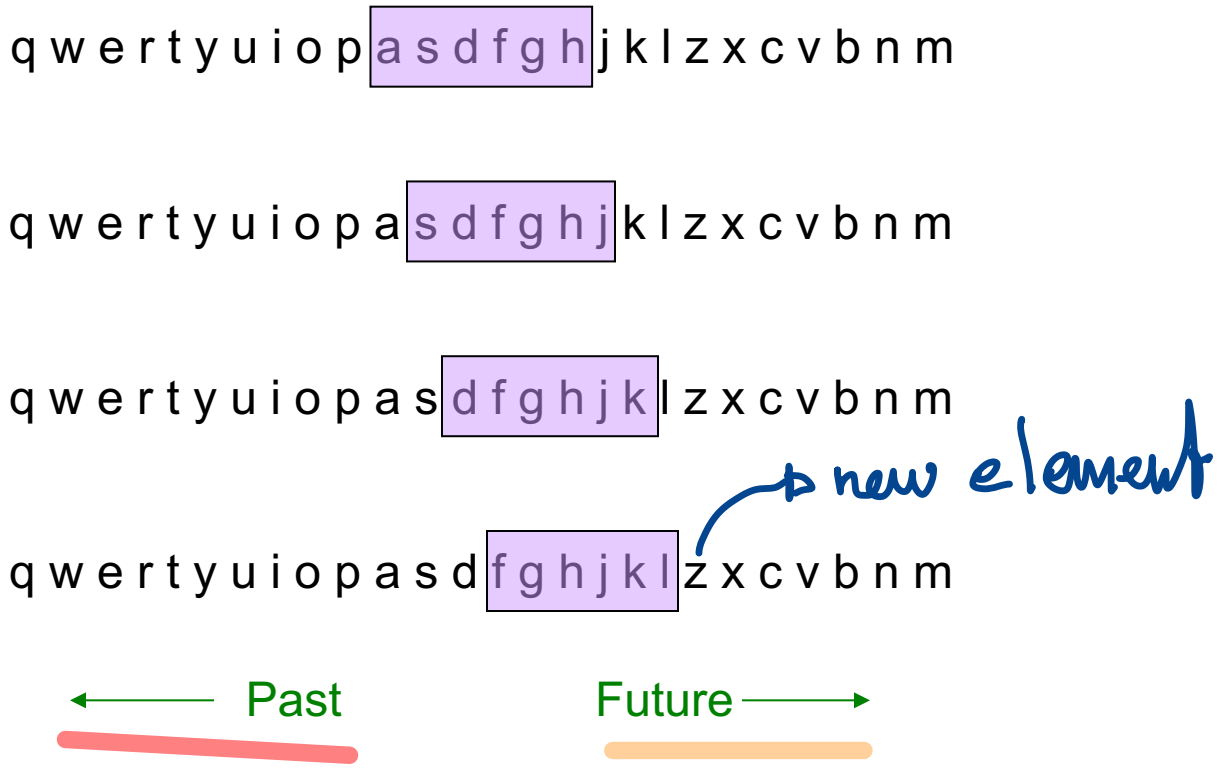
- A useful model of *stream* processing is that queries are about a ***window*** of length ***N***, *i.e.*, the ***N*** most recent received elements



- **Interesting case (BigData): *N*** is so large that the data cannot be stored in memory, or even on disk
  - Or, there are so many streams that windows for all cannot be stored
- **Application:** A stream element is a money *transaction* of a big company.
  - For every product ***x*** we keep a 0/1 ***stream*** of whether **x** was sold in the ***j-th*** transaction (**j = 1, …, N**)
  - **Typical query:** With input **k ≤ N,** return how many times have we sold **x** in the last **k** transactions

Sliding Window: 1 Stream

- **Sliding window on a single stream:** N = 6



Counting Bits

- **Problem**
- **Input:** An infinite on-line binary stream ***I***; Window Length **N**;
  - Compute function ***#1(I,N,k) = # 1's in the last k bits, for any k ≤ N***

- **Trivial Solution:**

  Store the most recent ***N*** bits
  - When new bit comes in, discard the ***N+1st*** bit

0 1 0 0 1 1 0 1 1 1 0 1 0 1 0 1 1 0 1 1 0 1 1 0 Ex. N=k=6

Past ← → Future

Counting Bits

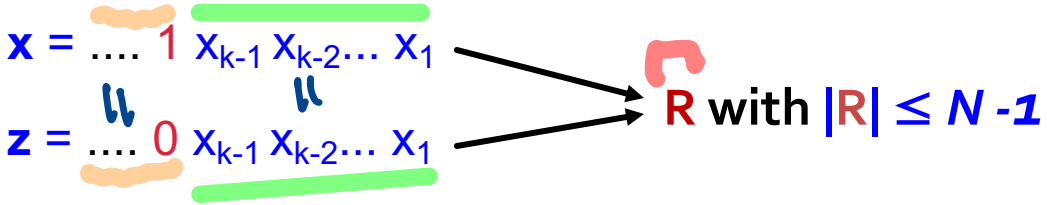
THM (Worst-Case Lower Bound). For arbitrary input-stream distributions, function ***#1(I,N,k)*** **requires**

Memory-Space(N,k) ≥ N

DETERMINISTIC EXACT SOLUTIONS!

sketch

Proof : Suppose it is possible to use a representation scheme **R(N,k)** with $|R(N,k)| \leq N - 1$. Then two N-bits windows **w # x** exist that must have the same representation **R**. Then, wlog, assume, for some k,



Now, any algorithm for ***#1(I,N,k), having access only to R,*** ***cannot distinguish between x and z***, so it **fails**

Counting Bits

- **"Big Data" Scenario:**

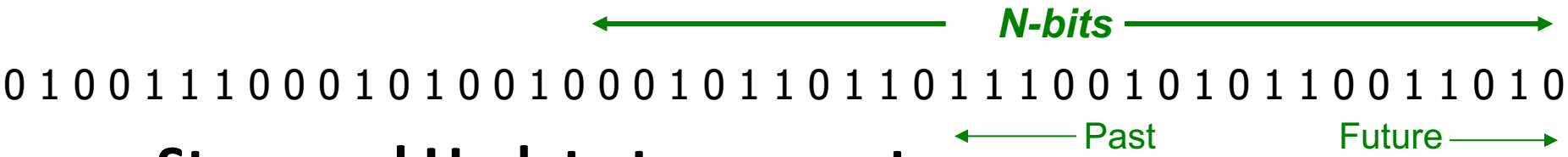
What if we cannot afford to store *N* bits?

- **E.g.**, we're processing **1 billion** streams and ***N* = 1 billion**

But we are happy with an ***approximate*** answer

Approximating Function #1(I,N,k)

- A simple solution that does not really solve our problem: **Uniformity Assumption on Input Stream *I***



- **Store and Update two counters:**
  - ***S***: #1s in the N-bits window **(Θ(log N) space)**
  - (***Z***: #0s in the N-bits window) (*for free:* S+Z = N)
- **Output:** Function ***#1(I,N,k)*** ≈ $k \cdot \frac{S}{S+Z}$

correct solution only for K = N

- What if the input distribution is non-uniform?
- What if it changes over time?

DGIM Method

[Datar, Gionis, Indyk, Motwani-*SICOMP* 2002]

- ***DGIM*** Algorithm for ***#1(I,N,k)*** works for *any* input distribution.
- ***Memory-Space(N,k)*** $= O(\log^2 N)$ bits per stream

- It gives **approximate** output, never off (i.e. **worst-case**) by more than **50%**
  - Error factor can be reduced to any constant value > 0, with more complicated algorithms and proportionally more stored bits

WORST-CASE APX RATIO : $\frac{1}{2} \leq \frac{DGIM(I,N,k)}{\#(I,N,k)} \leq 1+\frac{1}{2}$

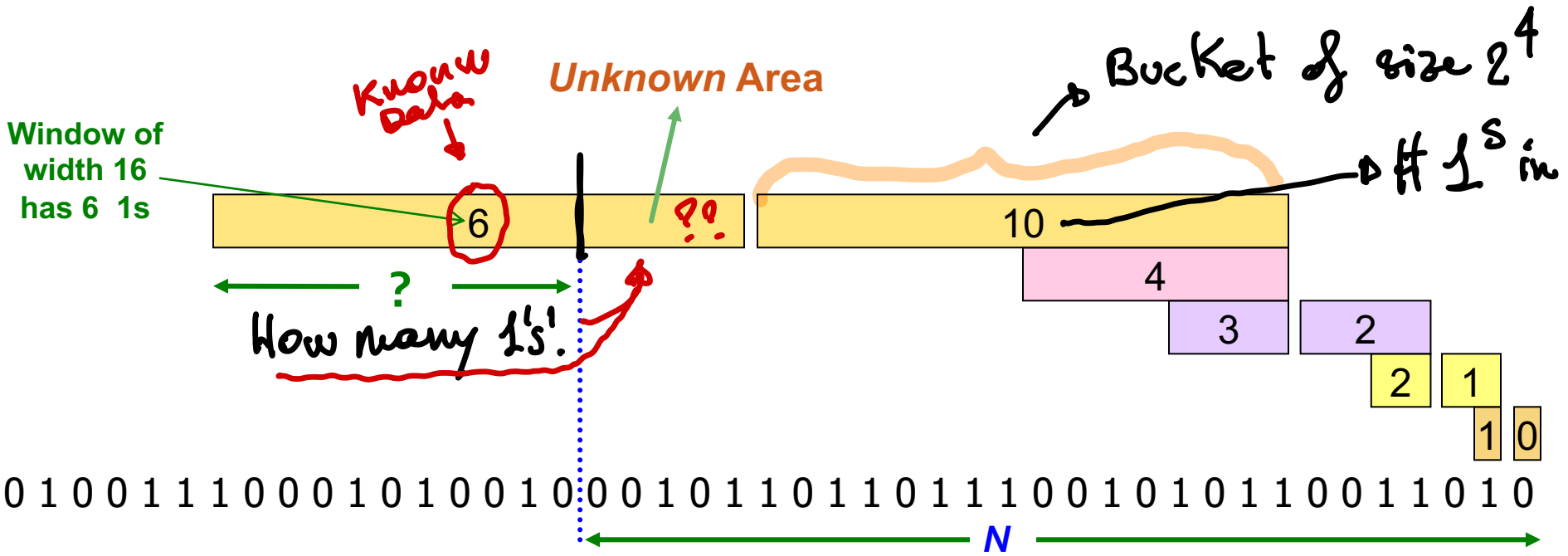
APX

TARGET

Idea: Storing Exponential Sub-Windows

WARM-UP

- Algorithm **Exp-Buckets** that **doesn't (quite) work** (it is not in the book):
  - Give *Sketches* (keep **#1s** and a *starting pointer*) for **exponentially increasing** regions (Buckets) of the Stream (mod N), looking backward
  - Drop any small Bucket if it begins at the same point as a larger Bucket



Important Remark: can reconstruct the value for *#1(I,N,k)* of the last ***N*** bits, **except** we *are not sure* how many of the last **six 1s** are included in the ***last N bits***

Positive Aspects of Exp-Buckets

- **Memory-Space(N,k) = O($\log^2 N$) bits**
  - $O(\log N)$ Buckets of size $\log_2 N$ bits each
- ***Easy* update** as more stream bits enter
- Worst-case Error for ***#1(I,N,k)*** no greater than the number of **1s** in the ***Unknown* Area**

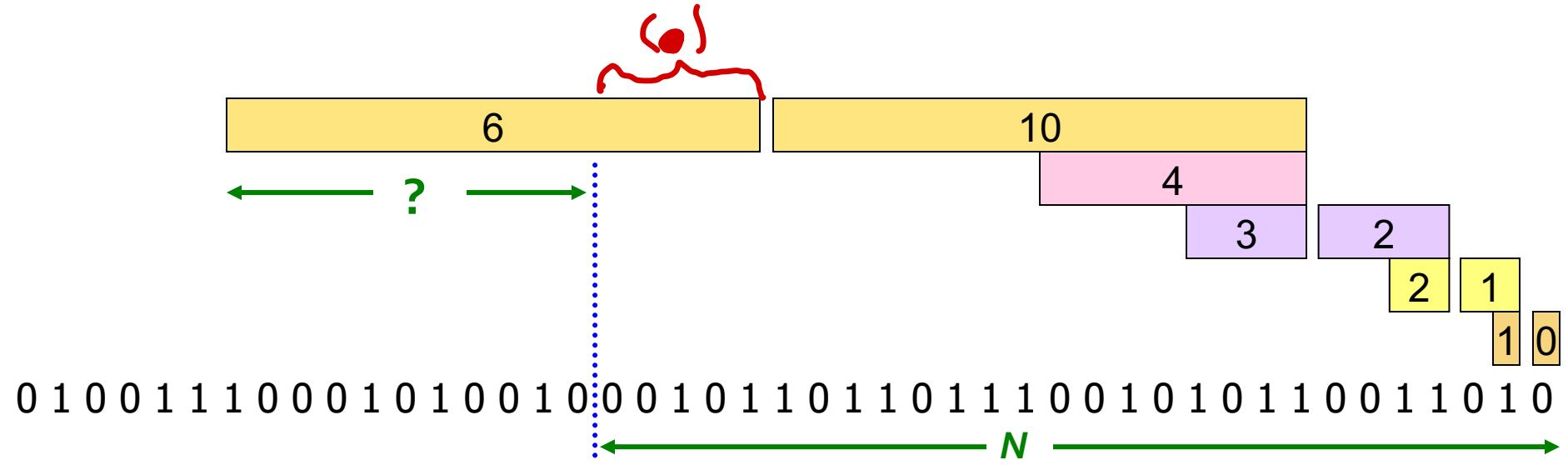
(•) why ?? for a fixed length 2^t, only 2 Buckets exist!

(••) why ?? length and starting points are Indices $\leq N$

Negative Aspects of Exp-Buckets

- As long as the **1s** are *fairly evenly* distributed, the apx-error due to the ***Unknown Area*** is small – BOUNDED
- But it could be that **all the 1s** are in the ***Unknown Area*** (at the end): In that case, **the apx-error is unbounded!**

The #1's in (●) can be of SIZE $(1-o(1)) \cdot \#1(I,N,N)$



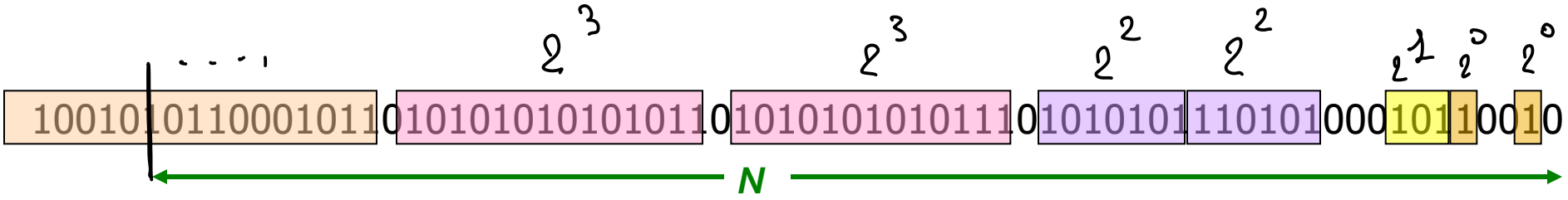
[Datar, Gionis, Indyk, Motwani]

Fixup: *DGIM* Algorithm

- ***DGIM* Key-Idea:** Instead of summarizing *fixed-length* blocks, summarize blocks with *specific* number of **1s**:
  - Let the block ***sizes*** (= number of **1s**) increase *exponentially*

Key Idea!

- When there are ***few*** 1s in the window, block sizes stay *small*, so apx-errors are *small*

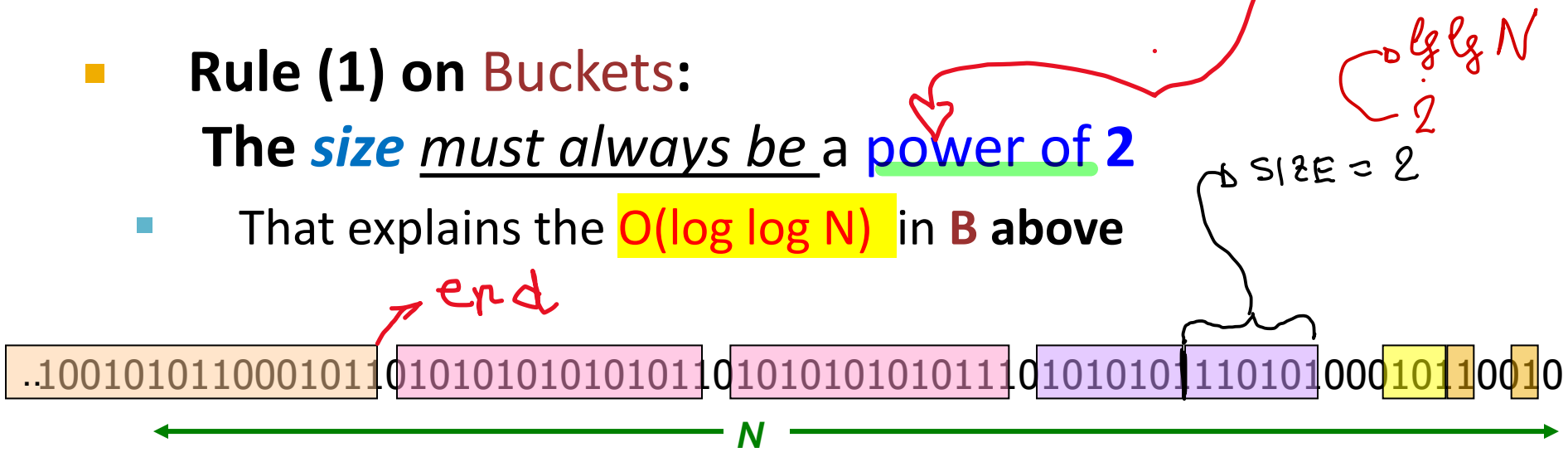


DGIM: Timestamps

- Each bit in the stream ***I*** has a ***time-stamp***, starting **1**, **2**, …
- Record **time-stamps** mod ***N***, so we can **represent** any **relevant** time-stamp with overall $\boldsymbol{O(log_2 N)}$ bits of memory-space

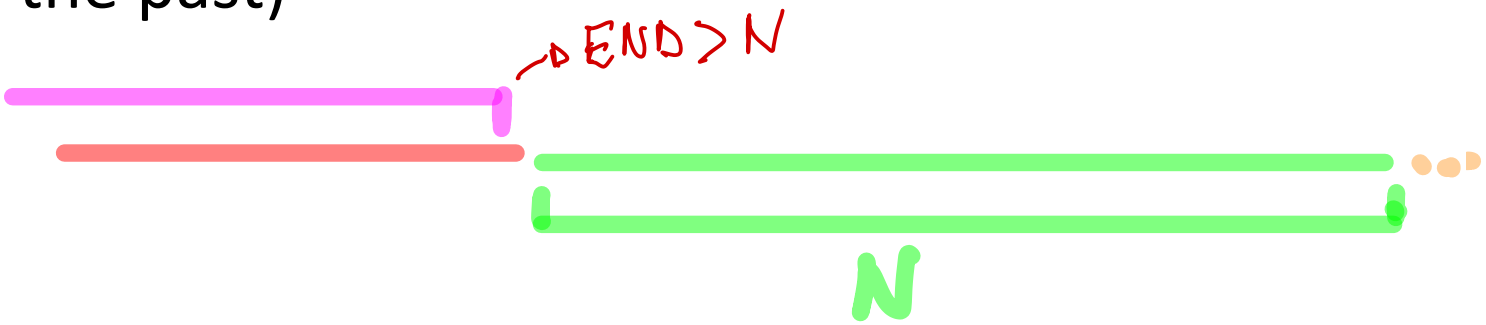
DGIM: Buckets

- A *Bucket* in the *DGIM* Algo is a record <A,B> consisting of:
  - **A =** The time-stamp of its ***end*** [O(log *N*) bits]
  - **B = The *size* of the** Bucket **=** The **#** of **1s** *between* its ***beginning*** and ***end*** [O(log log *N*) bits] (?)
- **Rule (1) on** Buckets**:**
  The *size* *must always be* a power of **2**
  - That explains the O(log log N) in **B above**

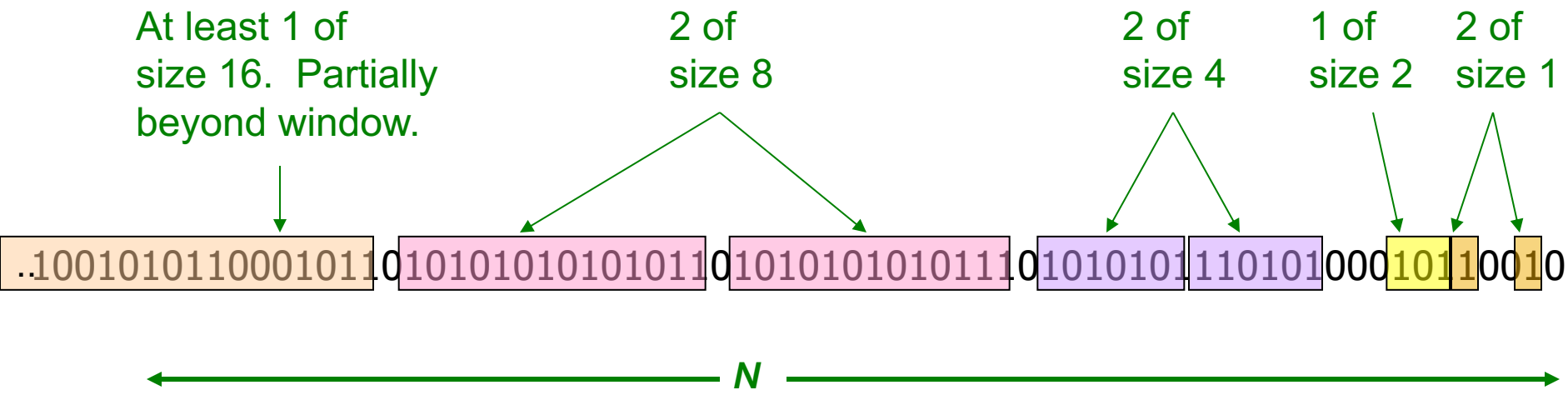


DGIM : Further Buckets Rules

- **(2)** Either **one** or **two** Buckets may exist with the same *size* (i.e. same **power-of-2 number** of **1s**)
- **(3)** Buckets do not **overlap** in time-stamps
- **(4)** Buckets are sorted by *size*
  - *Earlier* Buckets are not smaller than *later* Buckets
- **(5)** Buckets disappear when their *end* is **> *N*** (time units in the past)



Example: *DGIM* Bucketized Stream



Three key-properties of Buckets that are *dynamically* maintained:

- (2) Either **one** or **two** Buckets with the same *size*
- (3) Buckets do not overlap in time-stamps
- (4) Buckets are sorted by *size* (decreasing w.r.t. TIME)

DGIM : Updating Buckets (I)

RULES :

- When a *new* bit comes in, *drop* the last (oldest) Bucket if its ***end*** is prior to ***N*** time units before the current time → check is easy!
- **Two cases may arise:** current bit x is **0** or **1**
- **IF x = 0 THEN no** other changes are needed

DGIM : Updating Buckets (II)

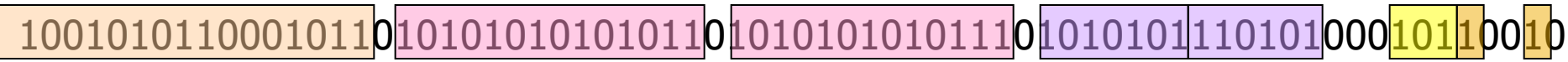
- **IF x = 1 THEN**
  - **(i)** Create a new Bucket of size **1**, for just this bit
    - **End time-stamp** := *current* time
  - **(ii)** If there are now three Buckets of size 1, combine the *oldest* two into a Bucket of size 2
  - **(iii)** If there are now three Buckets of size 2, combine the oldest two into a Bucket of size 4
  - **(iv)** *And so on …*

Q: How many ***merging*** steps does *DGIM* need *for each new bit* (Worst-Case) ?

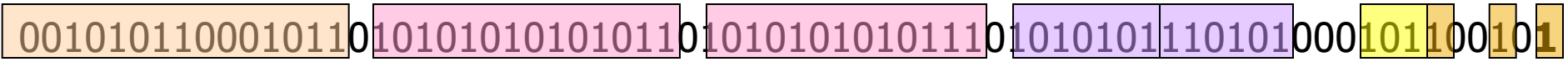
Example: Updating Buckets

Buckets size increases exponentially ⇒ # of MERGING is $\Theta(\log N)$!

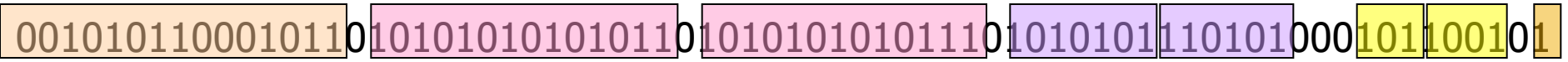
Current state of the stream:



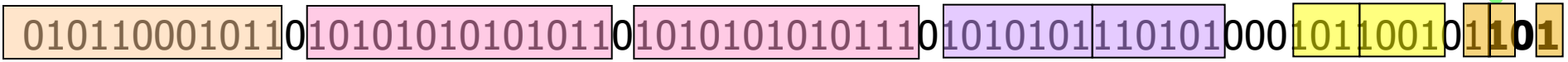
Bit of value 1 arrives



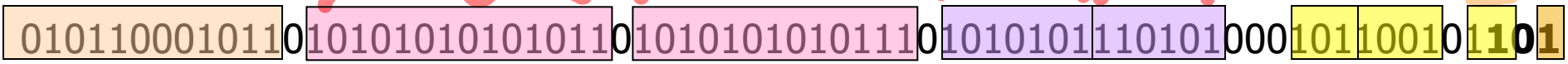
Two orange buckets get merged into a yellow bucket



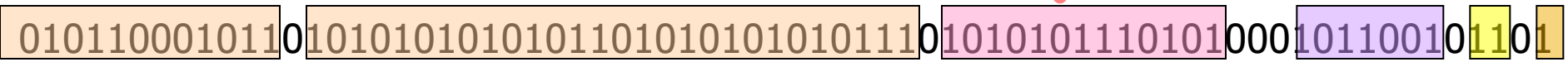
Next bit 1 arrives, new orange bucket is created, then 0 comes, then 1:



Buckets get merged…



State of the buckets after merging



DGIM : The Ouput

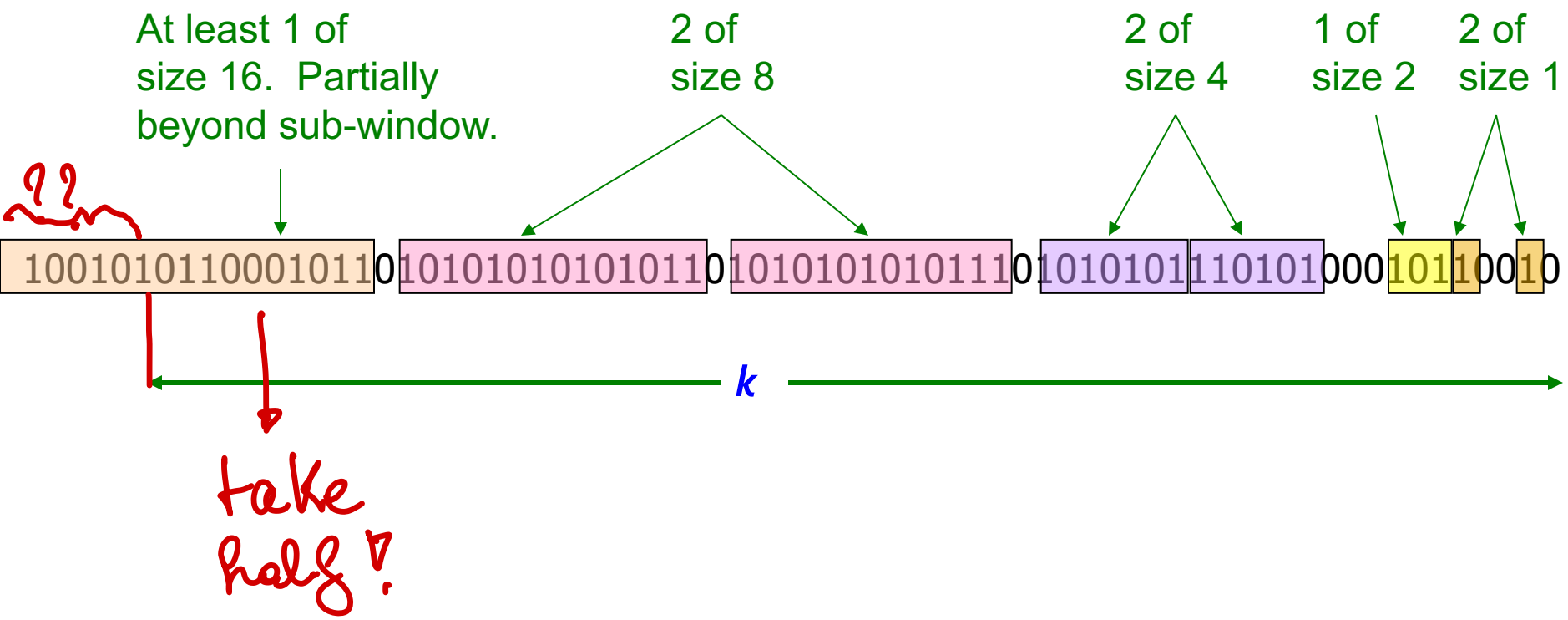
- The *DGIM* approximation ***Y*** for *Function* ***#1(I,N,k)*** is computed as follows:

ALGO OUTPUT

1. Sum the *sizes* of all living Buckets having end time-stamps **≤ k** but the oldest (Recall: "*size*" means the number of 1s in the Bucket)
2. Add half the *size* of the oldest Bucket having end time-stamp **≤ k**

- **Recall:** We do not know how many **1s** of the oldest Bucket are still within the target sub-window

Example: Bucketized Stream



DGIM Apx-Error Bound

ANALYSIS of APX

- **THM.** **Worst-Case *DGIM* Apx Error for *#1(I,N,k)* is at most *1.5*.**
- **Proof.** Assume: **(i)** the oldest Bucket has *size* 2^r and, **(ii)** $\#1(I,N,k) < Y$ → this yields Error
- *DGIM* Algo assumes that 2^{r-1} (i.e., half) of its **1s** are still within the k-sub-window → *DGIM* output is

$$Y = \sum_{j=0\ldots r-1} a_j \cdot 2^j + (1/2) \cdot 2^r = \sum + 2^{r-1} \quad \text{with} \quad a_j \in \{1,2\}$$

- while, in the worst case, $\#1(I,N,k) \geq \sum + 1$. So, the **Apx** ratio (WORST-CASE)

$$Y / \#1(I,N,k) \leq (\sum + 2^{r-1}) / (\sum + 1) \quad (*)$$

but, since there is *at least one* Bucket of *each* of the *sizes less* than 2^r, $\sum \geq 2^r - 1$, *

$$(*) \rightarrow Y / \#1(I,N,k) \leq (\sum / \sum) + (2^{r-1} / 2^r - 1) \approxeq 1 + 0.5$$

The case $\#1(I,N,k) > Y$ is similar.

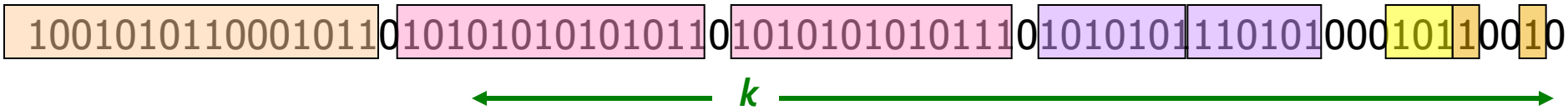
$$* \quad \leq \sum_{j=0}^{r} 2^j = 2^r - 1$$

Further Reducing the Apx-Error: Informal

- **Key-Idea:** Instead of maintaining **1** or **2** Buckets of *each size*, we allow either ***b*-1** or ***b*** Buckets (***b* > 2**)
  - Except for the largest size buckets; we can have ***any number*** between **1** and ***r*** of those
- **Good News: Error is at most *O(1/r)***
- By picking ***r*** appropriately, we can *tradeoff* between space complexity and the apx-error

Extensions: (Informal)

- Can we handle the case where the stream is not bits, but integers, and we want the **sum** of the last *k* elements?

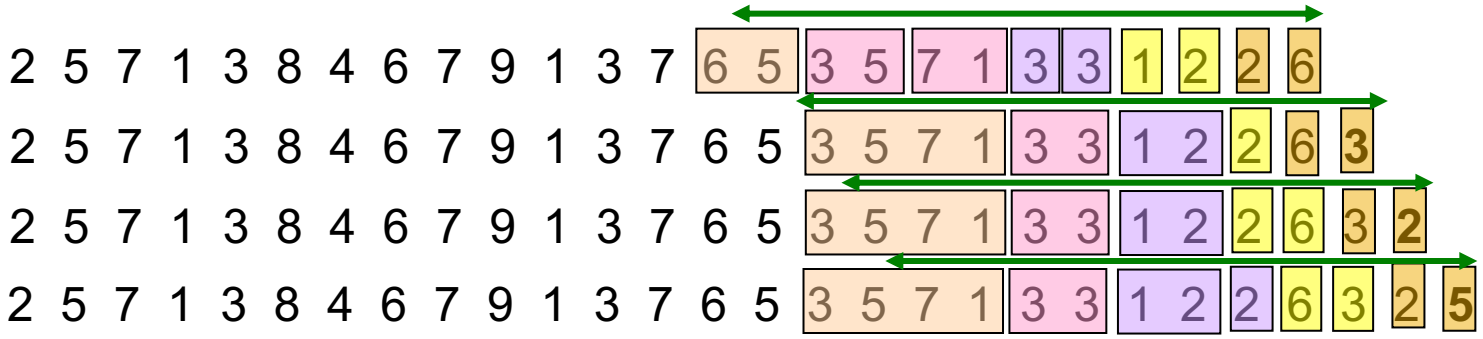


Extensions (Informal)

- **Stream** of positive integers
- **Goal: Estimate the Sum of the last *k* elements**
  - **E.g.:** *Avg. price* of last **k** sales
- **Solutions**
  - **(1) If you know all have at most *m* bits**
    - Treat ***m*** bits of each integer as a *separate* stream
    - Use DGIM to count **1s** in each integer
    - The sum is $= \sum_{i=0}^{m-1} c_i 2^i$
  - **(2) Use buckets to keep partial sums**
    - **Sum of elements in size *b* bucket is at most 2^b**

c_i …estimated count for **i-th** bit

: BINARY REPR → DECIMAL REPR.



Idea: Sum in each bucket is at most 2^b (unless bucket has only **1** integer)

Bucket sizes:

16	8	4	2	1

Summary

- **Sampling a fixed proportion of a stream**
  - Sample size grows as the stream grows
- **Sampling a fixed-size sample**
  - *Reservoir* Sampling
- **Counting the number of 1s in the last k bits of an N-size window of the Stream**
  - Exponentially increasing windows (*DGIM* Algo)
  - **Extension**:
    - Sums of integers in the last N elements

The End