DS 503: Advanced Data Analytics

Lecture 8: Streams

Dr. Gagan Raj Gupta

More algorithms for streams:

- **More algorithms for streams:**
  - **(1) DGIM Method**
  - **(2) Itemsets**
  - **(3) Filtering a data stream: Bloom filters**
    - Select elements with *property* ***x*** from stream (approx. set membership)

DGIM Method

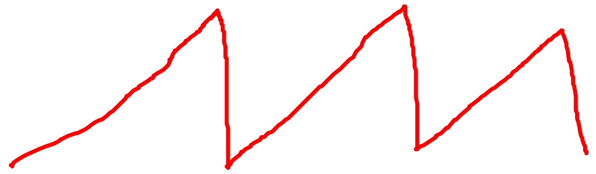
- **DGIM solution that does not assume uniformity**
  - We store $O(\log 2N)$ bits per stream
- **Solution is approximate, never off by more than 50%**
  - Error factor can be reduced to any fraction > 0, with more complicated algorithm and proportionally more stored bits
  - Error example: If we have 10 1s then 50% error means estimate = 10 +/- 5
- Basic Idea:
  - Summarize exponentially increasing regions of the stream, looking backward

DGIM: Terms

- Timestamps:
  - Each bit in the stream has a timestamp, starting 1,2,…
  - Record the timestamps modulo N (the window size), so we can represent any relevant timestamp in N bits

$12 \bmod 7 = 5$

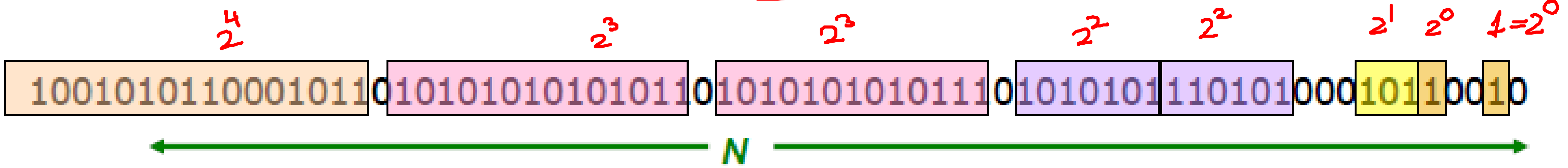
$12000 \bmod 7 = \{0 \ldots 6\}$



- Buckets:
  - A bucket in the DGIM method is a record consisting of
    - The timestamp of its end [O(log N) bits]
    - Number of 1s in each bucket must be a power of 2
    - The number of 1s between its beginning and end [O(log log N) bits]

binary

$011 \rightarrow 1 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 = 3$



Representing a stream by buckets

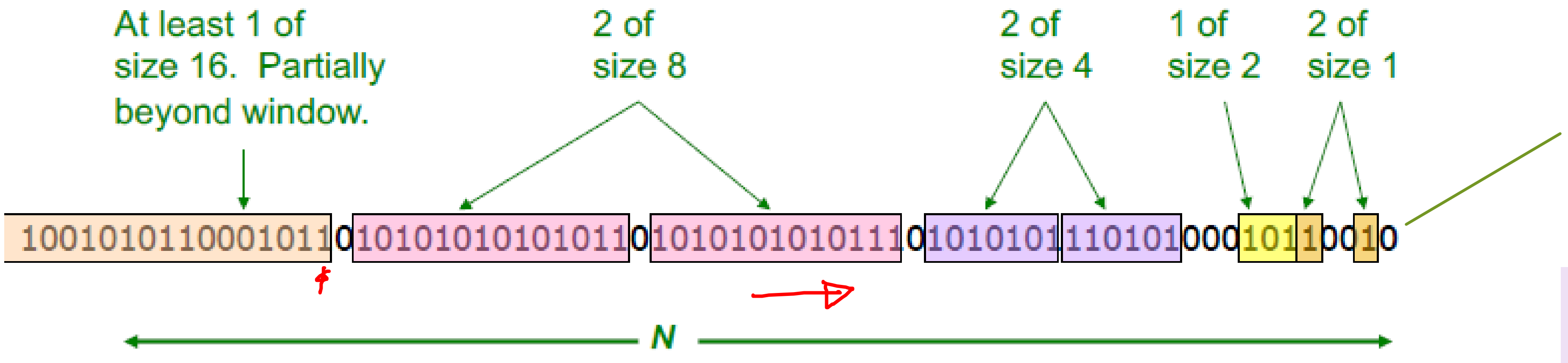
- We can have only one or two buckets with the same number of 1s (power-of 2)
  - Think of this as the binary representation of the number of 1s

5 → 2+2+1

- Buckets do not overlap in timestamps
- Buckets are sorted by size
  - Earlier buckets are not smaller than later buckets

recent

- Buckets disappear when their end-time is >N time units in the past



Most recent bit

Bits are not stored, only buckets are!

Updating Buckets

- When a new bit comes in, drop the last (oldest) bucket if its end-time is prior to N time units before the current time
- Processing the current bit:
  - If the current bit is 0: No other changes needed
  - If current bit is 1:
    - Create a new bucket of size 1, for just this bit. End timestamp = current time
    - If there are now three buckets of size 1, combine the oldest two into a bucket of size 2 with timestamp of older one
    - If there are now three buckets of size 2, combine the oldest two into a bucket of size 4 with timestamp of older one
    - And so on …

Example: Updating Buckets

Current state of the stream:

10010101100010110101010101010101101010101010101110101010111010100010110010

16 8 8 4 4 2 1 1

Bit of value 1 arrives

00101011000101101010101010101011010101010101011101010101110101000101100101

End timestamp

1 1 1

Two orange buckets get merged into a yellow bucket

00101011000101101010101010101011010101010101011101010101110101000101100101

2 2

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

new bits

0101100010110101010101010101011010101010101011101010101110101000101100101101

Buckets get merged…

0101100010110101010101010101011010101010101011101010101110101000101100101101

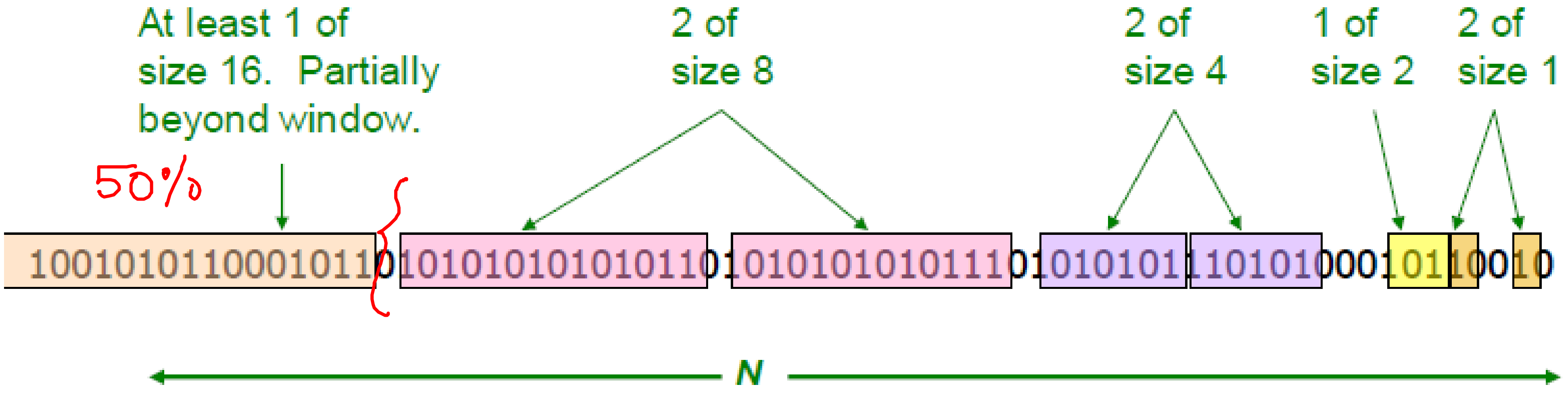
State of the buckets after merging

0101100010110101010101010101011010101010101011101010101110101000101100101101

N

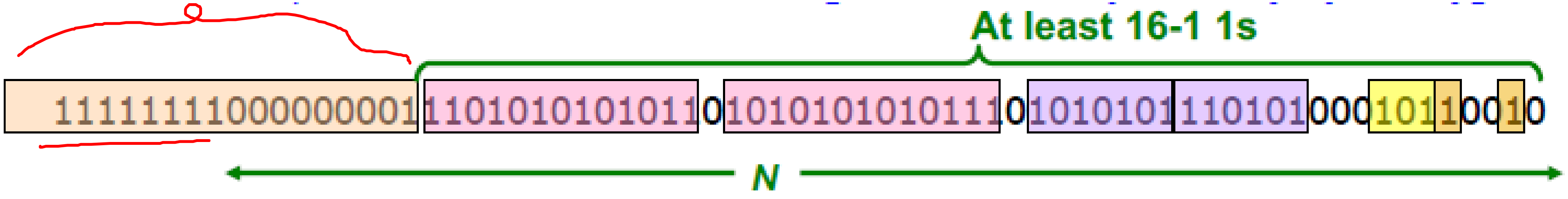
How to Query?

- **To estimate the number of 1s in the most recent *N* bits:**
  - **Sum the sizes of all buckets but the last (note "size" means the number of 1s in the bucket)**
  - **Add half the size of the last bucket (**we can also do other approximations**)**
- **Remember:** We do not know how many **1s** of the last bucket are still within the wanted window



Error Bound: Proof

- **Why is error at most 50%? Let's prove it!**
- Suppose the last bucket has size 2^r
- Then by assuming 2^{r-1} (i.e., half) of its **1s** are within the last bucket (but out of last-N bits), we make an error of at most $\pm 2^{r-1} - 1$
- Since there is at least one bucket of each of the sizes less than 2^r, the true sum is at least **1 + 2 + 4 + .. +** $\mathbf{2^{r-1}} = \mathbf{2^r}$**-1**
- Thus, error at most **50%** **[=**$\mathbf{2^{r-1}/2^r > (2^{r-1}-1)/(2^r-1)}$**]**



Further Reducing the Error

- Instead of maintaining **1** or **2** of each size bucket, we allow either ***r*-1** or ***r*** buckets (***r* > 2**)
  - Except for the largest size buckets; we can have any number between **1** and ***r*** of those
- **Error is at most *O*(1/*r*)**

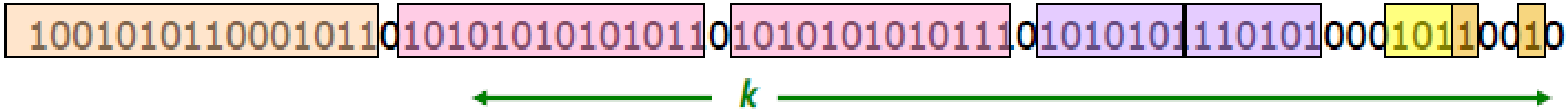
  $\frac{1}{2} \sim 50\%$ 10% → 10
  - see MMDS book for details
- By picking ***r*** appropriately, we can tradeoff between number of bits we store and the error

Extensions (Dynamic Window size)

- Can we use the same trick to answer queries: **How many 1's in the last *k*?** where ***k* < *N***?

  100

  - Find earliest bucket **B** that at overlaps with ***k***.
  - Number of **1s** is the **sum of sizes of more recent buckets + ½ size of B**

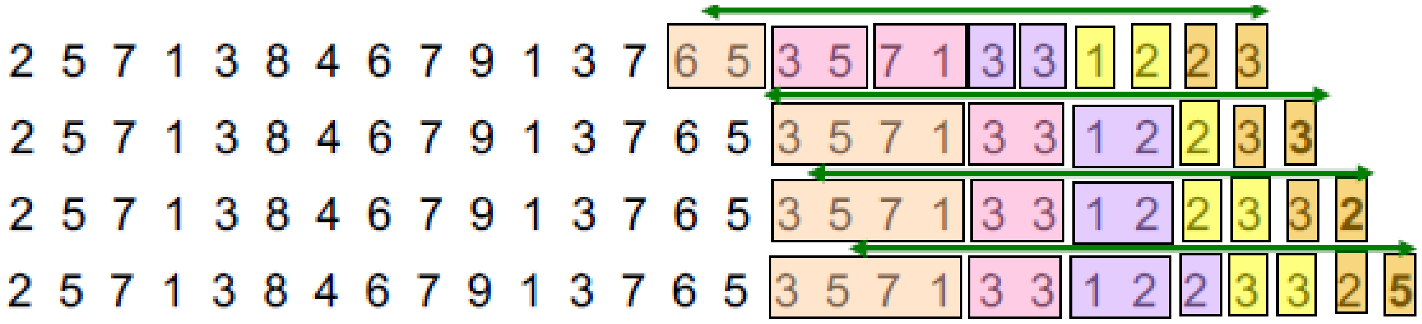


80

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

Extensions: Sum of last k elements

- **Stream of positive integers**
- **We want the sum of the last *k* elements**
  - **Application (Amazon):** Avg. price of last **k** sales
- **Solution:**
  - **(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/stream
    - 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**



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

Max bucket sum:

16	8	4	2	1

Counting Itemsets

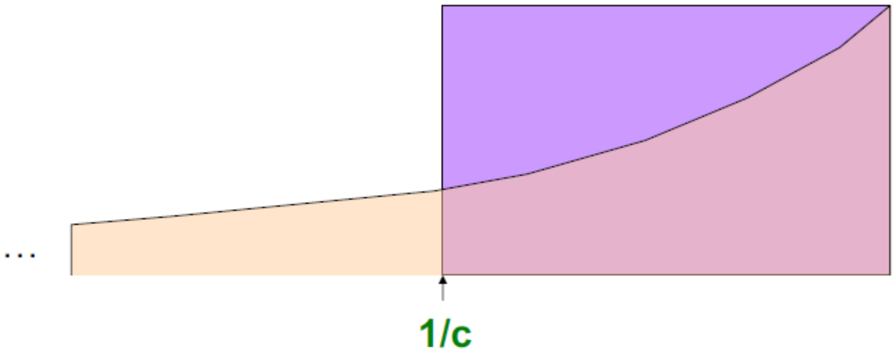
- **New Problem: Given a stream, which items (or itemsets) appear more than *s* times in the window?**
- **Possible solution:** Think of the stream of baskets as one binary stream per item/itemset
  - **1** = item/itemset present; **0** = not present
  - Use **DGIM** to estimate counts of **1**s for all items/itemsets
- **In principle, you could count frequent pairs or even larger sets**
- **Drawbacks:**
  - Only approximate counts are maintained
  - **Number of itemsets (all subsets of baskets/carts) is way too big!**

Exponentially Decaying Windows

- **Exponentially decaying windows: A heuristic for selecting likely frequent item(sets)**
  - **What are "currently" most popular items?**
    - Instead of computing the raw count in last ***N*** elements
    - Compute a **smooth aggregation** over the whole stream
- If stream is $a_1, a_2, \ldots$ and we are taking the sum of the stream, take the answer at time ***t*** to be: $S_1 = a_1$; $S_t = \sum_{i=1}^{t} a_i (1-c)^{t-i}$
  - **c** is a constant, presumably tiny, like 10^{-6} or 10^{-9}
- **When new a_{t+1} arrives:**
  - Multiply current sum by **(1-c)** and add a_{t+1} ; $\boxed{S_{t+1} = S_t(1-c) + a_{t+1}}$

Example: Counting Items

- If each a_i is an "item" we can compute the **characteristic function** of each possible item ***x*** as an Exponentially Decaying Window
  - That is: $\sum_{i=1}^{t} \delta_i (1-c)^{t-1}$ where $\delta_i = 1$ if $a_i = x$ and 0 otherwise
- Imagine that for each item ***x*** we have a binary stream (**1** if ***x*** appears, **0** if ***x*** does not appear)
- ***We maintain the sum of each element***
- 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***



Important property:

$$\sum_t (1-c)^t = \frac{1-(1-c)^t}{1-(1-c)} \sim \frac{1}{c}$$

Example: Counting items

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

Suppose c=0.1
Current Counts: $\{S_x = 5, S_y = 2, S_z = 1, S_a = 0.8, S_b = 0.7, S_c = 0.5\}$
An item d comes next
Updated Counts: $\{S_x = 4.5, S_y = 1.8, S_d = 1, S_z = 0.9, S_a = 0.72, S_b = 0.63\}$
S_c *is dropped!*

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
  - Count every subset: One basket of **20** items would initiate **1M** counts (2^20)
- **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** 2/c
  - Initiate new counts for itemsets **only under certain condition** (next slide)

Initiation of New Counts for itemsets

- 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

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 N elements**
  - Exponentially increasing windows
  - Extensions:
    - Number of 1s in any last k (k < N) elements
    - Sum of integers in the last N elements
- **Task: Which were the most popular recent items/itemsets?**
  - Can keep exponentially decaying counts for items and potentially larger itemsets
  - **Be conservative about starting counts of large sets**

Filtering Data Streams/Approximate Set Membership

- **Each element of data stream is a tuple**
- Given a list of keys **S**
- **Determine which tuples of stream are in *S***
- **Obvious solution: Hash table**
  - But suppose we **do not have enough memory** to store all of ***S*** in a hash table
  - E.g., we might be processing millions of filters on the same stream

Very large prime

$(a \cdot b + c) \bmod p$

$0 \; . \; . \; . \; p-1$

D

$h(x)$

x

0

Pdts

$p-1$

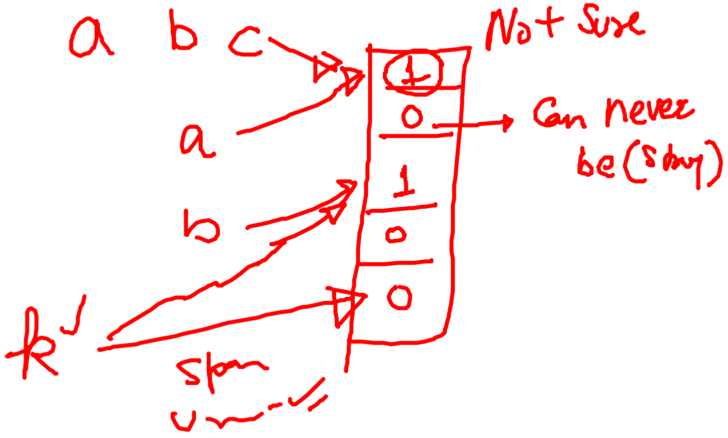
$p(x = k) = \frac{1}{p}$

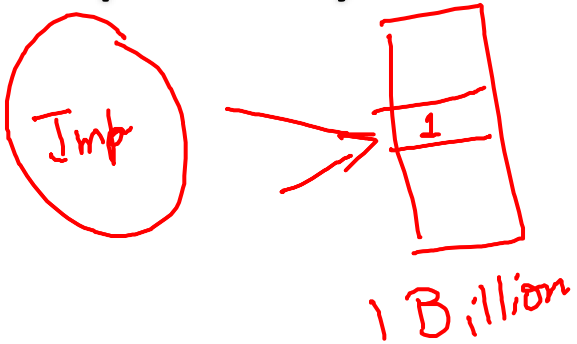
Applications

- **Example: Email spam filtering**
  - We know 1 billion "good" email addresses
    - Or, each user has a list of trusted addresses
  - 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 in certain sets of keywords
  - Determine whether each message matches user's interest
- **Content filtering:**
  - You want to make sure the user does not see the same ad multiple times
- **Web cache filtering:**
  - Has this piece of content been requested before? Then cache it now.

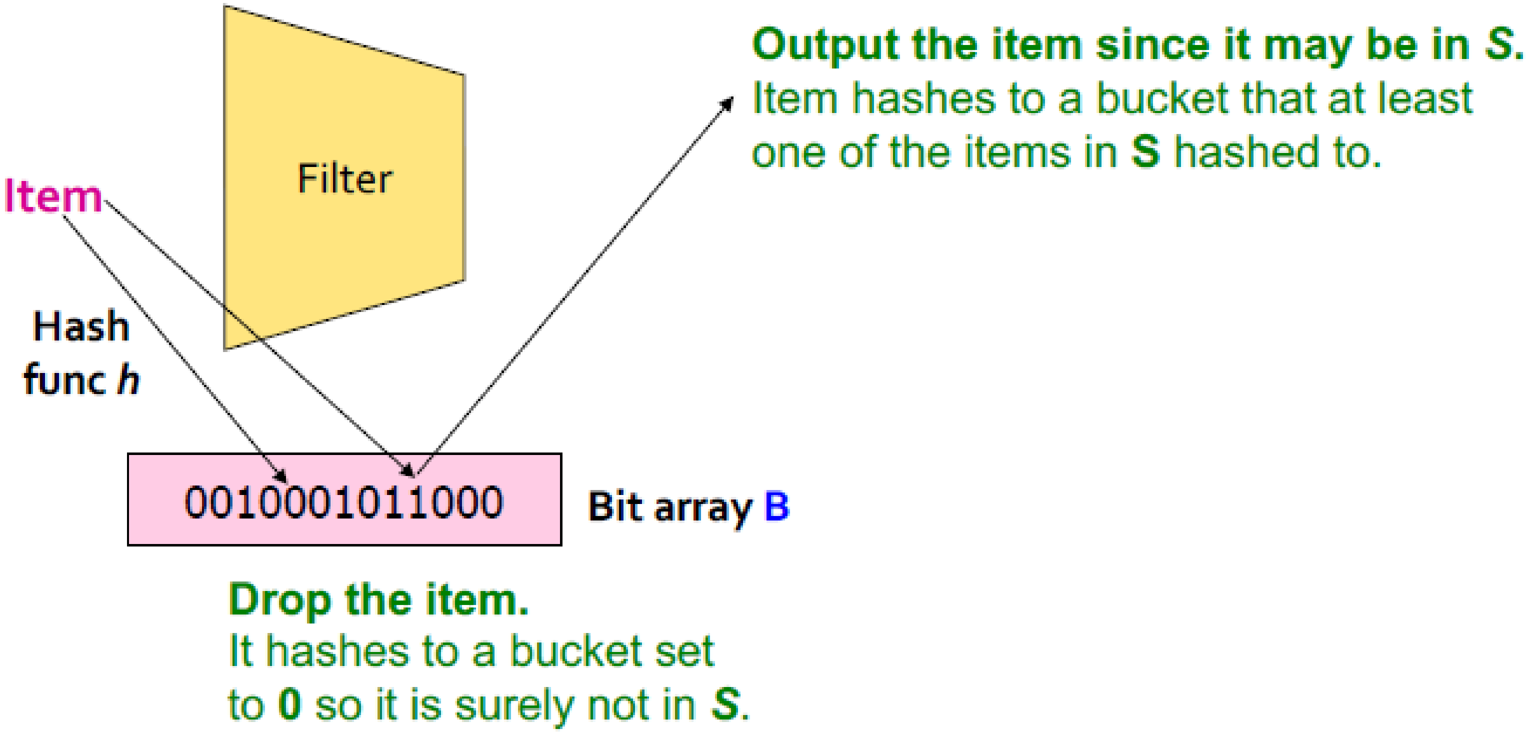
First Cut Solution (1)

- **Given a set of keys *S* that we want to filter**
  - Create a **bit array *B*** of ***n*** bits, initially all ***0s***
  - Choose a **hash function *h*** with range ***[0,n)***
  - Hash each member of $s \in S$ to one of ***n*** buckets, and set that bit to **1**, i.e., ***B[h(s)]=1***
  - Hash each element ***a*** of the stream and output only those that hash to bit that was set to **1**
    - **Output *a* if B[h(a)] == 1**





First Cut Solution (2)



- **Creates false positives but no false negatives**
  - If the item is in ***S*** we surely output it, if not we may still output it (depends on the hash function collisions)

First Cut Solution (3)

- **|S| = 1 billion email addresses** (good)
- **|B|= 1GB = 8 billion bits**
- If the email address is in ***S***, then it surely hashes to a bucket that has the bit set to **1**, so it always gets through (***no false negatives***)
- Approximately **1/8** of the bits are set to **1**, so about **1/8th** of the addresses not in ***S*** get through to the output (***false positives***)
- Actually, less than **1/8th**, because more than one address might hash to the same bit

Analysis: Throwing Balls into Bins

- **More accurate analysis for the number of false positives**
- **Consider:** 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:**
  - **Bins** = bits/buckets, **Balls** = hash values of items
- P[A given bin is empty] = $\left(1-\frac{1}{n}\right)^m$
- P[A given bin has at least one ball] = $1-\left(1-\frac{1}{n}\right)^m \sim 1-e^{-\frac{m}{n}}$
  - As $n \to \infty$, $\left(1-\frac{1}{n}\right)^n = e^{-1} = \frac{1}{e}$

1 ball

$\frac{n-1}{n}$

$\frac{m}{n}$

m/n

BQ

Example:

M=1 Billion

N= 8 Billion

Fraction of 1s in array B

$= 1 - e^{-\frac{1}{8}} = 0.1175$

.125

1 0 0 0 1 0 8

Bloom Filter

- Consider: **|S| = *m*, |B| = *n***
- Use ***k*** independent hash functions $h_1, h_2, \ldots, h_k$
  - Simple hash functions such as H={h=[(ax+b) mod p] mod n} will work
- **Initialization:**
  - Set **B** to all **0s**
- **Update**:
  - Hash each element $s \in S$ using each hash function h_i
    - Set $B[h_i(s)] = 1$ (for each ***i* = 1,.., *k***)
- **Run-time:**
  - When a stream element with key ***x*** arrives
  - If $B[h_i(x)] = 1$ **for all *i* = 1,…, *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***

Note 1
We have the same array B for all hash functions

Note 2
Bloom filters support only insertions, but no deletions to the Set

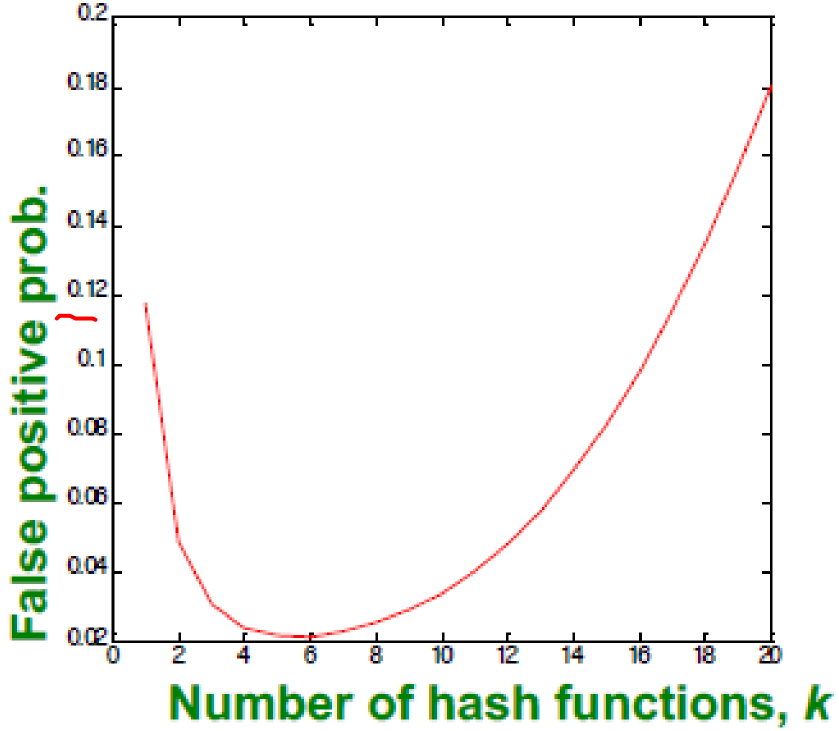
Bloom Filter - Analysis

- **What fraction of the bit vector B are 1s?**
- Throwing ***k·m*** balls into ***n bins***
  - Fraction of **1**s is $1 - e^{-\frac{k.m}{n}}$
- We have ***k*** independent hash functions and we only let the element ***x*** through **if all *k*** hash element ***x*** to buckets of value **1**
- So, **false positive probability =** $\left(1 - e^{-\frac{k.m}{n}}\right)^{k}$
- Question: Given, m and n, what is the optimal value of k?
  - Differentiate w.r.t k and set derivative to 0.
  - Minima at $k = \frac{n}{m}\ln(2)$

BQ

Bloom Filter: Example

- m = 1 billion, n = 8 billion
- k = 1: $\left(1 - e^{-\frac{1}{8}}\right)$ = 0.1175
- k = 2: $\left(1 - e^{-\frac{1}{4}}\right)^2$ = 0.0493
- Optimal k = 8 ln(2) = 5.54 ~ 6
- Error at k=6: $\left(1 - e^{-\frac{3}{4}}\right)^6$ =0.0216



Bloom Filter: Summary

- Bloom filters allow for filtering / set membership
- **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:** $\left(1-e^{-\frac{k.m}{n}}\right)^k$ vs. $\left(1-e^{-\frac{m}{\left(\frac{n}{k}\right)}}\right)^k$
  - But keeping **1 big B** is simpler