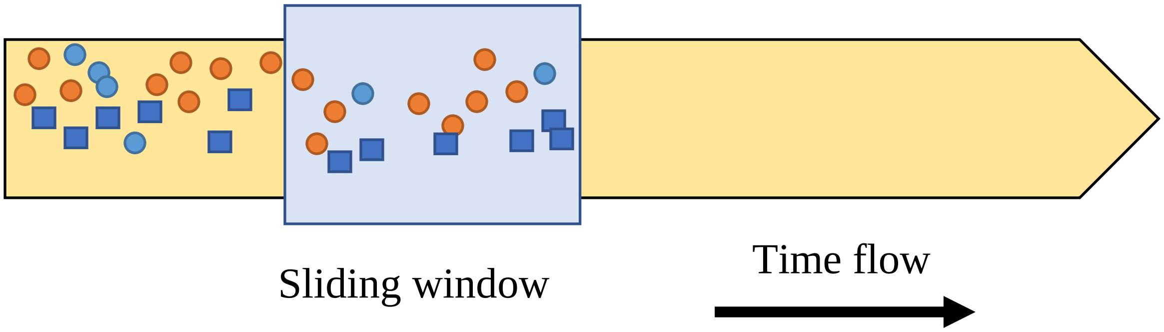
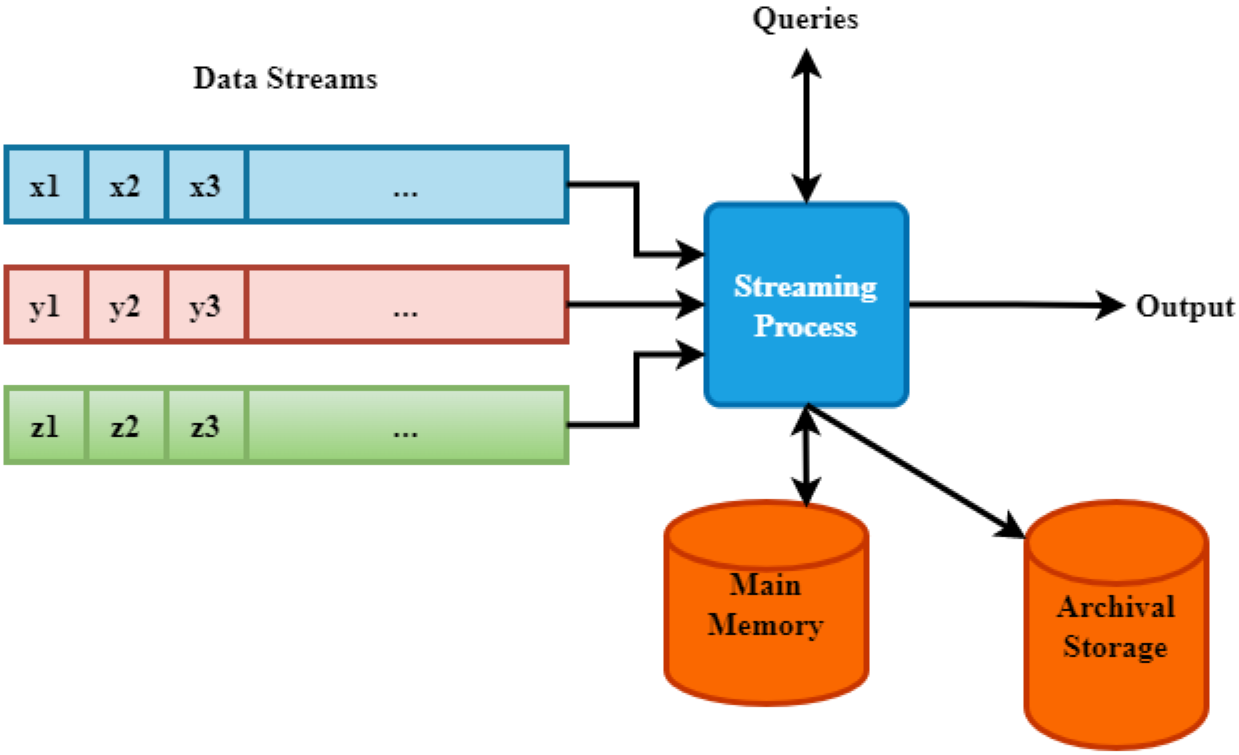
Steaming Data

- **Objectives**
  - Understand the distribution of a data stream
  - Data is continuous and unbound
  - Hard to process with algorithms for batch data
  - Explore stream processing to analyze and process big data in real time to gain current insights to make appropriate decisions.
  - The system cannot store the entire stream
  - How to process the unbound data stream using limited resources?
  - Queries on streams can be very useful: Monitoring, alerts, automated triggering of actions

- **What is a data stream?**
  - Streaming data is used to describe unbounded, time-ordered large sequence data generated continuously at **high velocity**.
  - "A **data stream** is a **real-time**, continuous, ordered (implicitly by arrival time or explicitly by timestamp) **sequence of items**. It is impossible to control the order in which items arrive, nor is it feasible to **locally store** a stream in its entirety." - Golab & Oszu



  - Data streams (also called tuples) are:
    - infinite – one does not know the size of the data
    - non-stationary – the distributions of the data can change over time (seasonally, daily, hourly)



- Applications
    - Counting distinct elements: Number of distinct elements in the last k elements of the stream
    - Sample data from a stream
    - Filtering items: Number of distinct elements in the last k elements of the stream
    - Estimating moments: Estimate avg./std. dev. of last k elements
    - Queries over sliding windows: Number of items of type x in the last k elements of the stream
    - Mining query streams: Google wants to know what queries are most frequent than yesterday
    - Window size = one day and count the frequency of queries
    - Mining click streams: Yahoo wants to know which of its pages are getting an unusual number of hits in the past hour.
    - Mining social networks: Looking for trending topics on twitter, Facebook, etc.

      - Monitor packets at network switch: detect denial of service attaches.

- **Streaming Algorithms**
  - A data stream is a sequence of data

    $S = s_1, s_2, \ldots, s_i, \ldots,$

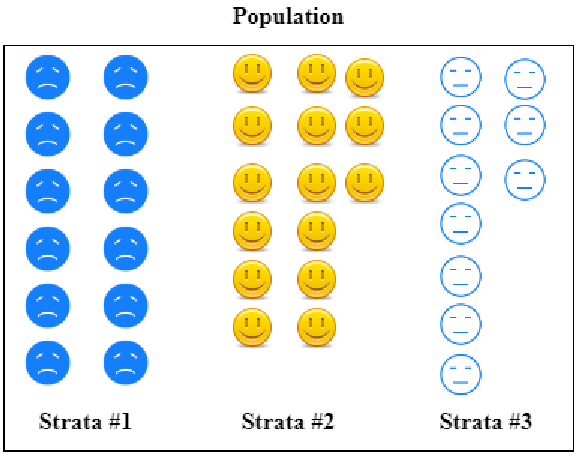
    where each item s_i is an item in the universe U, where $|U| = N$.
  - A streaming algorithm A takes S as input and needs to compute some function f of S.
  - Processing constraints:
    - limited memory
    - limited processing time per item
    - Streaming data can only be read once.
    - Streaming algorithms produces approximate answer due to processing constraints.
    - Streaming algorithms efficiency measurements:
      - How much data you can store at a time
      - Processing time for an input data stream
      - Number of passes to process a data stream
  - **Streaming algorithm approaches:**
    - There are several approaches to process streaming data such sketching, randomized algorithms, etc.
    - We are going to look at two approaches:
      - Random sampling
      - Sliding windows
    - **Window-based streaming:**
      - It is a technique for reducing the complexity of algorithms.
      - Make decisions based only on recent data of sliding window size w
      - An element arriving at time t expires at time t + w
      - Data elements are grouped within a window that slides across the data stream according to a specified interval.

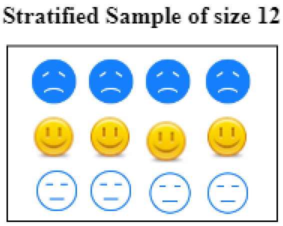
A	B	C	D	E	F	G	H	I

A	B	C	D	E	F	G	H	I

- **Random sampling**
  - It consists of selecting a group from a population to represent the whole population.
  - Sample without knowing the total length in advance
  - Sampling techniques:
    - Probabilistic random sampling:
      - It is a technique in which each member in a population has an equal chance of being selected as a sample(unbiased)
    - Non-probabilistic non-random sampling
      - It uses arbitrary sample selection instead of sampling based on a randomized selection
  - **Probabilistic sampling techniques:**
    - Simple random sampling:
      - It is a random and automated method to select a sample.
      - This sampling method assigns numbers to the individuals and then randomly chooses numbers.
      - The samples are chosen in two ways:
        - Through a lottery system
        - Random number generation software.
    - Systematic sampling:
      - Data elements are selected at regular intervals from the sampling data. The intervals are chosen to ensure an adequate sample size. If you need a sample size s from a population of size n, you should select every $\frac{n}{s}$th data item for the sample.
      - Example:
        - Suppose you want to sample 10 students from a list of 50 students: $\frac{50}{10}$ =5

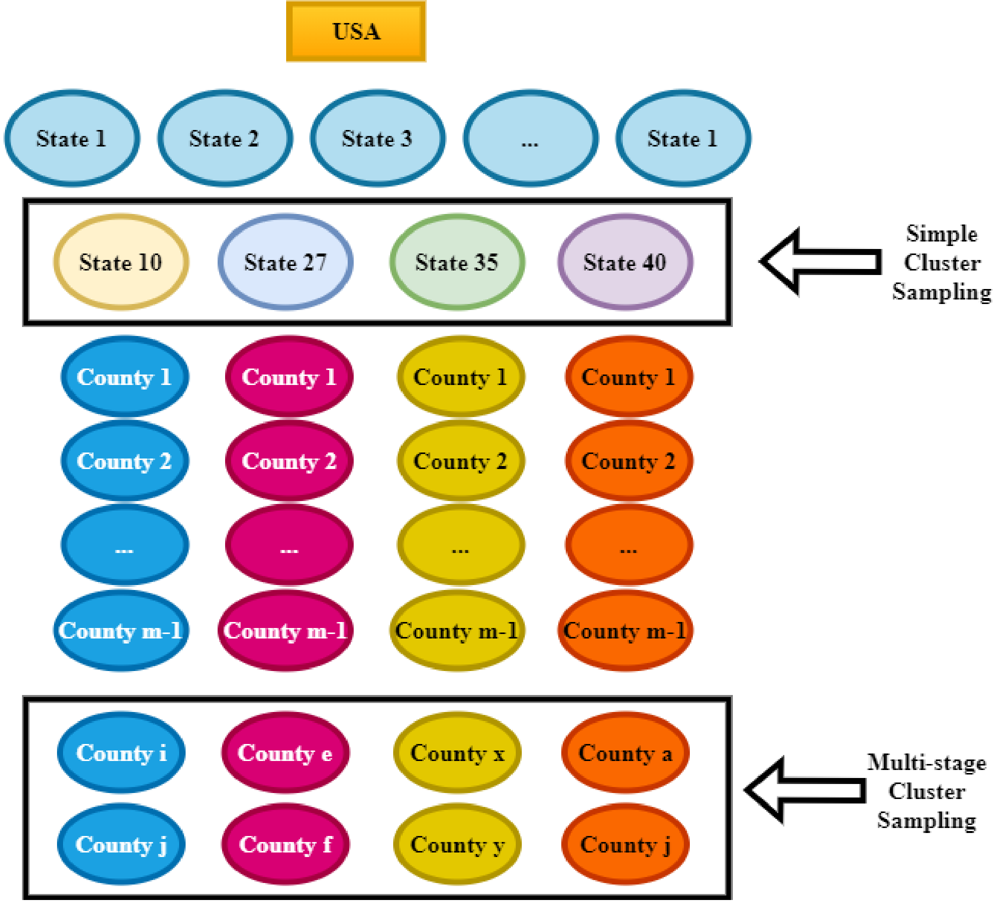
        - So, every 5th student is chosen after a random starting point between 1 and 5.
        - If the ransom number is 4, then the students selected are: 4, 8, 12, 16,18,22,24, 28, 32,36.
- Stratified sampling:
    - It divides the population into smaller groups, or strata, based on shared characteristics-two strata: Male vs. Female.
    - The groups of the population are based on certain criteria, then randomly choose elements from each in proportion to the group's size.





- Clustered sampling:
    - It is also known as area sampling.
    - It is used when the population is very large.
    - It is a probability sampling technique used when different subsets of groups are present in a larger population.
    - Sampling is done in three steps:
        - Step 1: Divide the population into naturally non-overlapping clusters where each cluster is a mini representation of the entire population.

- Step 2: Simple clustered sampling:
    - Randomly choose k clusters to form your sample.
    - Stop if you are happy with your sample. Otherwise, continue to Step 3.
- Step 3: Multi-stage clustered sampling:
    - Further divide each cluster into new clusters and go step 2.



- **Examples of streaming algorithms:**
    - Filtering a data stream:
        - Select elements with property x from the stream
        - **Bloom Filter** algorithm
    - Counting distinct elements:
        - Number of distinct elements in the last k elements of the stream.
        - **Flajolet Martin** (**FM**) Algorithm

- Finding frequent elements:
  - Finding which element is repeatedly coming-which user is repeatedly visiting the site or how many times product x was sold (Amazon)
  - **Datar Gionis Indyk Motwani** (**DGIM**) Algorithm
- Estimating moments:
  - Estimate avg/std. dev. of last k elements.
  - **Alon-Matias-Szegedy** (**AMS**) Algorithm
- Finding data items with certain properties:
  - Queries on google searches in a specific month
  - Products bought at Walmart during the Christmas season
  - **Reservoir Sampling**

- **Distinct element counting problem**
  - Count how many people are visiting a web site.
    - Count how many distinct IP numbers are connecting to the server that hosts the web site.
  - Count the number of distinct products sold last week.
  - Naïve Approach:
    - Clearly, using O(N) memory space, the problem can be solved easily in O(N log N) time by sorting, or O(N) expected time with hashing.
    - With big data: space is limited.
    - We need the following:
      - An unbiased estimator of the counts
      - Ok to have an error in the estimation as trade-off for space.
    - **Flajolet Martin (FM) Algorithm**
      - Flajolet Martin Algorithm, also known as **FM** algorithm, is an **approximation algorithm**.
      - It **approximates** the number of unique elements in a data stream in **one pass** with **less memory** space.
      - If the stream contains n elements with m of them unique, **FM** runs in O(n) times and needs O(log(m)) memory.
      - Algorithm:
        - Assume we have N items in the universe
        - Pick a hash function h mapping the N items to at least $\log_2(N)$ bits
        - for each stream item s,
          - calculate h(s)
          - Convert h(s) to a binary representation
          - Let r(s) be the number of trailing 0s in the bit representation of h(s)
            //for instance, assume h(s) = 12, bit representation 1100
            //r(a) is then equal to 2
          - keep R = max(r(s)) over the entire stream
        - Estimator: the number of distinct items seems thus far is 2^R

- Example:
  - Given a set S={1,3,2,1,2,3,4,3,1,2,3,1} and a hash function:
    - h(x)=(6x+1) mod 5

x	H(x)	Binary	r(a)				x	H(x)	Binary	r(a)
1	2	00010	1				4	0	00000	5
3	4	00100	2				3	4	00100	2
2	3	00011	0				5	1	00001	0
3	4	00100	2				2	3	00011	0
2	3	00011	0				3	4	00100	2
3	4	00100	2				1	2	00010	1

So, R = max(r(a)) = 5
And the number of distinct elements = $N=2^R=2^5=32$

- Consider another has function: h(x)=(x+7) mod 5

x	H(x)	Binary	r(a)				x	H(x)	Binary	r(a)
1	3	00011	0				4	1	00001	2
4	1	00001	0				6	3	00011	0
6	3	00011	0				5	2	00010	1
9	1	00001	0				5	2	00010	1
2	4	00101	0				2	4	00100	2
1	3	00011	0				9	1	00001	0

So, R = max(r(a)) = 2
And the number of distinct elements = $N=2^R=2^2=4$

- **Why FM algorithm works:**
  - The hash function, h(x), maps x with equal probability to any one of the N values
  - Then h(x) is a sequence of $\log_2(N)$ bits.
  - **The probability that h(x) ends r 0's is 2^{-r}.**
    - For r=1
      - $2^{-1}=\frac{1}{2}=50\%$ of the x's hash to **..**0
    - For r=2
      - $2^{-2}=\frac{1}{4}=25\%$ of the x's hash to **..**00
    - If the longest tail of 0's is r=2, item hash ending with **100, then
      - We have probably seen bout $4=2^2$ distinct items so far.
    - For r
      - $\frac{1}{2^r}$ of all hash values have their binary representation end in **r** 0's.
      - if the hash function generated an integer ending in **r** 0's, intuitively, the **number of unique strings** is around $\mathbf{2^r}$
  - So, the probability that a given h(x) ends with r 0's is $\mathbf{2^{-r}}$

➔ And the probability of NOT seeing a tail of r 0's among m elements in the stream:

$$(1-2^{-r})^m$$

<u>The probability of all **m** data items ends in **fewer than r** 0's</u>

- Let us approximate $(1-2^{-r})^m$:

$$(1-2^{-r})^m = e^{\ln(1-2^{-r})^m} = e^{m\ln(1-2^{-r})}$$

- Let us approximate $\ln[(1-\mathbf{2}^{-r})$ using Taylor expansion:

$$\ln[(1+\boldsymbol{x}) = \boldsymbol{x} - \frac{\boldsymbol{x}^2}{\mathbf{2}} + \frac{\boldsymbol{x}^3}{\mathbf{3}} + \ldots$$

$$\ln[\left(1-\mathbf{2}^{-r}\right) \approx -\mathbf{2}^{-r}$$

So,

$$(\mathbf{1}-\mathbf{2}^{-r})^{m} = e^{\mathbf{m}\ln(\mathbf{1}-\mathbf{2}^{-r})} \approx e^{-\,\mathbf{m}\,\mathbf{2}^{-r}}$$

- So, the probability of NOT finding a tail of r 0's is:

  - If $\mathbf{m} \ll \mathbf{2}^{r}$ then the probability tends to **1**

$$(\mathbf{1}-\mathbf{2}^{-r})^{m} \approx e^{-\,\mathbf{m}\,\mathbf{2}^{-r}} = \mathbf{1} \ \ \textbf{since} \ \ \frac{m}{\mathbf{2}^{r}} \to \mathbf{0}$$

The probability of finding a tail of length r 0's tends to 0

  - If $\mathbf{m} \gg \mathbf{2}^{r}$ then the probability tends to **0**

$$(\mathbf{1}-\mathbf{2}^{-r})^{m} \approx e^{-\,\mathbf{m}\,\mathbf{2}^{-r}} = \mathbf{0} \ \ \textbf{since} \ \ \frac{m}{\mathbf{2}^{r}} \to \infty$$

The probability of finding a tail of length r 0's tends to 1

- In summary:
  - Let m be the number of distinct elements seen so far in the stream (Our objective is to estimate m)
  - We have shown that the probability of finding a tail of r 0's is:
    - 1 if $\mathbf{m} \gg \mathbf{2}^{r}$
    - 0 if $\mathbf{m} \ll \mathbf{2}^{r}$

- In practice the probability of seeing a tail of r 0’s is neither 1 or 0

➔ **2^r will always be around m**

- **Bloom Filters: Filtering Data Stream Algorithm**
  - It has been around for over 50 years.
  - Check if some data item is NOT present in a very big list
  - Check if a username exists without hitting performing a full database search – especially for large databases
  - How to save time, space, and disk I/Os in checking if a data element exists?
    - Filter the non-existence of a username without a full search: **Constant TIME and SPACE**.
  - Applications:
    - Google Chrome used to use Bloom filters to detect malicious URLs
    - Facebook and Gmail use Bloom Filter to check if a user exists.
  - What is a Bloom Filter?
    - It is a space efficient probabilistic data structure developed by Burton Howard Bloom back in 1970.
    - It used to test whether an item is a member of a set.
    - It never generates a FALSE NEGATIVE: 0%
    - It has some FALSE POSITIVE: It confirms that an item exists while it does not.

	Actual Positive	**Actual Negative**
Predicted Positive “A maybe answer”	True Positive: 1-p	False Positive: p* **The item has never been inserted, yet we are returning TRUE.**
Predicted Negative	False Negative: 0%	**True Negative:100%**
*** Minimize the probability p**		

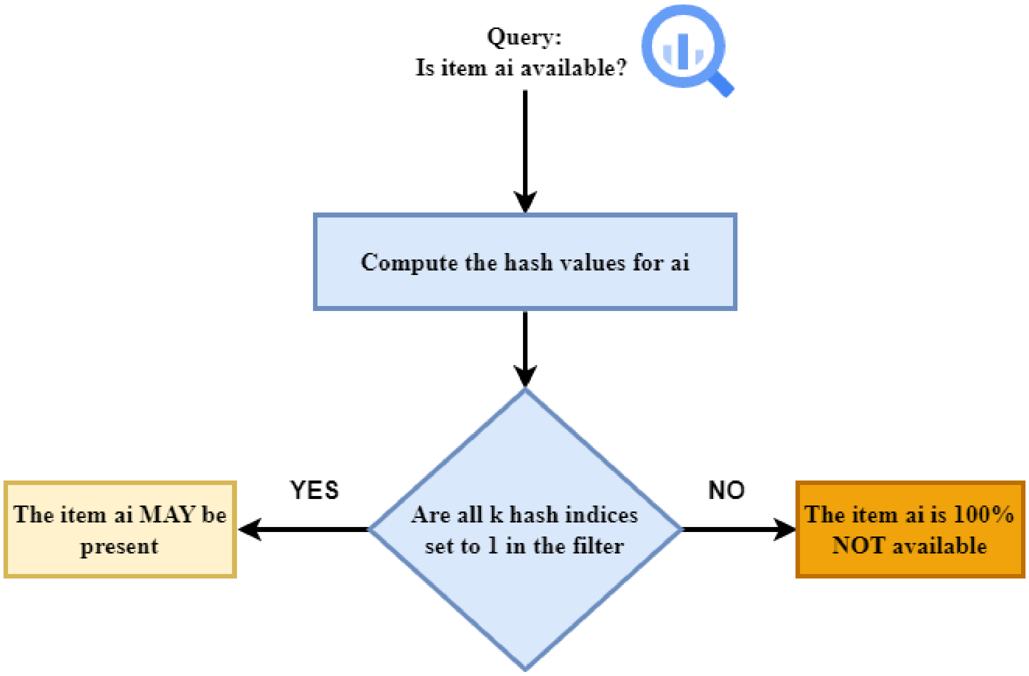
    - No deletion: Cannot delete an item from the filter

  - Cannot list the inserted items in the filter.
- **How does a Bloom Filter work?**
  - Given a set S of m items.
  - A Bloom filter is a n-bit array initialized to 0's:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

  - It uses a collection of k hash function $h_1, h_2, h_3, \dots, h_k$
  - **Insertions:**
    - Step1: Calculate indices using k hash functions
      - Each of the k hash functions maps an item from S to one of the n-bit array
    - Step 2: Set bits to 1 at indices calculated in step 1

  - **Query: Bloom Filter Lookup**
    - Suppose an item ai appears in the data stream and we want to know if it has been seen before.



  - **Example:**

- Given a set S of string characters (usernames):
  S={cat, dog, bird, lion. Frog}
- And the following two hash functions h_1 and h_2.

H_1(word) = (ASCII(first char)+ASCII(second char)+ASCII(last)) mod 16
H_2(word) = ((ASCII(first char))^2+ASCII(second char)+ASCII(last)- ASCII(first char
mod 16

Stream Item	H1	H2
Cat	8	9
Dog	10	0
Bird	15	4
Lion	3	4
Frog	15	4

- Insertions:
  - cat: set bit 8 and 9

0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

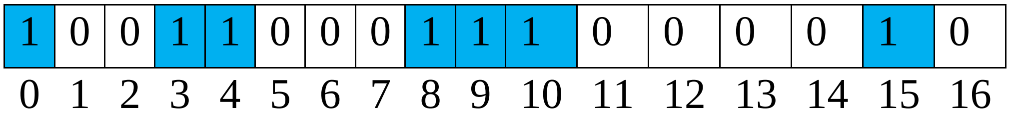
  - dog: set bit 0 and 10

1	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

  - bird: set bit 4 and 15

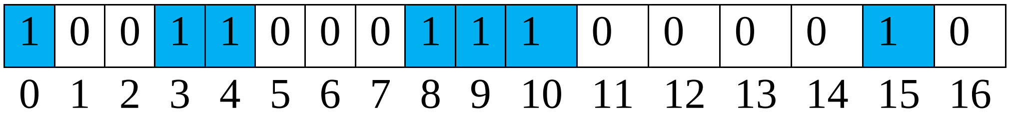
1	0	0	0	1	0	0	0	1	1	1	0	0	0	0	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

  - lion: set bit 3 and 4



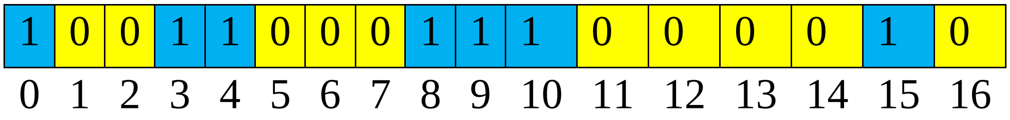
1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

- frog: set bit 4 and 15



1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

- Bloom Filter Final State:



1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

- Check if an item in available:
  - Use the following new items (usernames)

Stream Item	H1	H2
ant	3	0
tiger	15	0
monkey	5	6
snake	6	9

- Query: ant
  - Hash values are 3 and 0
  - All the bits in the filter are set to 1 ➔ False Positive

- Query: Tiger
  - Hash values are 0 and 15
  - All the bits in the filter are set to 0 ➔ True Negative

- Query: Monkey
    - Hash values are 5 and 6
    - All the bits in the filter are set to 0 ➔ True Negative

- Query: Snake
    - Hash values are 6 and 9
    - NOT all the bits in the filter are set to 1 ➔ True Negative

- **Bloom Filter – Analysis**
    - Given a Bloom Filter with **n bits** and uses k hash functions that are uniform and independent
    - What is the probability that a bit in the filter is 1, assuming one hash function?

      Probability is $\frac{\mathbf{1}}{\boldsymbol{n}}$
    - What is the probability that a bit in the filter is 0, assuming one hash function?

      Probability is $1 - \frac{\mathbf{1}}{\boldsymbol{n}}$

    - What is the probability that a bit in the filter is 0, after **m items** have been inserted using all k hash functions?

$$p_0 = (1 - \frac{\mathbf{1}}{\boldsymbol{n}})^{\boldsymbol{km}}$$

    - What is the probability that a bit in the filter is 1, after m items have been inserted using all k hash functions?

$$p_1 = 1 - p_0$$

- The probability of FALSE POSITIVE then is:

$$p_1{}^k = (1 - p_0)^k$$

Let us call the FALSE POSITIVE probability, FP

$$FP = p_1{}^k = (1 - (1 - \frac{\mathbf{1}}{\boldsymbol{n}})^{\boldsymbol{km}})^k$$

- Let us rewrite the probability that a bit in the filter is 0 using e

$$\left(1 - \frac{\mathbf{1}}{\boldsymbol{n}}\right)^{\boldsymbol{km}} = e^{\ln[(1 - \frac{\mathbf{1}}{\boldsymbol{n}})^{\boldsymbol{km}}]}$$

$$\left(1 - \frac{\mathbf{1}}{\boldsymbol{n}}\right)^{\boldsymbol{km}} = e^{km \ln[(1 - \frac{\mathbf{1}}{\boldsymbol{n}})}$$

- Let us approximate $\ln[(1 - \frac{\mathbf{1}}{\boldsymbol{n}})$ using Taylor expansion:

$$\ln[(1 + \boldsymbol{x}) = \boldsymbol{x} - \frac{\boldsymbol{x}^{\mathbf{2}}}{\mathbf{2}} + \frac{\boldsymbol{x}^{\mathbf{3}}}{\mathbf{3}} + \ldots$$

If x is very small, them the terms after x are much smaller

Then, $\ln[(1 - \frac{\mathbf{1}}{\boldsymbol{n}}) \approx -\frac{1}{n}$

And,

$$e^{km\ln[(1-\frac{\mathbf{1}}{\boldsymbol{n}})} \approx \; e^{-\frac{mk}{n}}$$

- So, the probability that a bit in the filter is 0:

$$p_0 = (1-\frac{\mathbf{1}}{\boldsymbol{n}})^{\boldsymbol{km}} \quad \approx e^{-\frac{mk}{n}} = \tilde{p}_0$$

- Let us approximate the FALSE POSITIVE probability, FP

$$\boldsymbol{FP} = \boldsymbol{p_1}^{\boldsymbol{k}} = (\mathbf{1}-(\mathbf{1}-\frac{\mathbf{1}}{\boldsymbol{n}})^{\boldsymbol{km}}\;)^{\boldsymbol{k}} \approx \left(\mathbf{1}-\boldsymbol{e}^{-\frac{\boldsymbol{mk}}{\boldsymbol{n}}}\right)^{\boldsymbol{k}}$$
$$\approx (\mathbf{1}-\tilde{\boldsymbol{p}}_{\mathbf{0}})^{\boldsymbol{k}}$$

Note that $\frac{\boldsymbol{m}}{\boldsymbol{n}}$is the number of items per slot (m is the number of items and n is the number of bits in the filter)

- How so we choose the number of hash function **k**?
  - If K is large, then
    - The filter will clog with 1's
  - If K is too small then,
    - The error does not decrease.
  - If you plot FP, the function shows a minimum
  - Compute the best k for a given m and n:
    - Take the derivative of FA (it is tricky)

$$\frac{d}{dk}(FA) = \; \frac{d}{dk}\left(\mathbf{1}-\boldsymbol{e}^{-\frac{\boldsymbol{mk}}{\boldsymbol{n}}}\right)^{\boldsymbol{k}}$$

➔ Take the log of FA: yield the same minimum

$$\frac{d}{dk}\ln(FA) = \frac{d}{dk}\ln\left(\mathbf{1} - \boldsymbol{e}^{-\frac{\boldsymbol{mk}}{\boldsymbol{n}}}\right)^{\boldsymbol{k}}$$
$$\frac{d}{dk}\ln(FA) = \frac{d}{dk}k\text{In}\left(\mathbf{1} - \boldsymbol{e}^{-\frac{\boldsymbol{mk}}{\boldsymbol{n}}}\right)$$

- Derivative is zero when

$$\boldsymbol{k} = \mathbf{ln\,2}\,.\frac{\boldsymbol{n}}{\boldsymbol{m}}$$

- Therefore, the best value for k is **best choice** of k:

$$\boldsymbol{k} = \mathbf{ln\,2}\,.\frac{\boldsymbol{n}}{\boldsymbol{m}}$$

If we pick ideal (# hashes) for fixed m and n, what fraction of the filter do we expect to be set bits?

What is the optimal value for $\widetilde{\boldsymbol{p}}_{\mathbf{0}}$?

$$\boldsymbol{e}^{-\frac{\boldsymbol{mk}}{\boldsymbol{n}}} = \widetilde{\boldsymbol{p}}_{\mathbf{0}}$$

➔ $\mathbf{ln}\,\boldsymbol{e}^{-\frac{\boldsymbol{mk}}{\boldsymbol{n}}} = \mathbf{ln}\,\widetilde{\boldsymbol{p}}_{\mathbf{0}}$

➔ $-\frac{\boldsymbol{mk}}{\boldsymbol{n}} = \mathbf{ln}\,\widetilde{\boldsymbol{p}}_{\mathbf{0}}$

➔ $\boldsymbol{k} = -\frac{\boldsymbol{n}}{\boldsymbol{m}}\mathbf{ln}\,\widetilde{\boldsymbol{p}}_{\mathbf{0}}$

So, the best choice of k

$$\boldsymbol{k} = \mathbf{ln\,2}\,.\frac{\boldsymbol{n}}{\boldsymbol{m}} = -\frac{\boldsymbol{n}}{\boldsymbol{m}}\mathbf{ln}\,\widetilde{\boldsymbol{p}}_{\mathbf{0}}$$

➔ $-\mathbf{ln\,2} = \mathbf{ln}\,\widetilde{\boldsymbol{p}}_{\mathbf{0}}$

➔ $\ln \tilde{p}_0 = -\ln 2 = \ln 2^{-1} = \ln \frac{1}{2}$

➔ $\tilde{p}_0 = \frac{1}{2}$

The filter is 50% set to 1.

- **Reservoir Sampling**
  - Reservoir sampling is a **fixed-size randomized algorithm** that chooses a data item without replacement, of **s** items from a population of unknown size n in a single pass over the items.
  - It maintains a set **s** of random samples seen so far in the stream.
  - New item has a certain probability $\frac{\mathbf{s}}{\mathbf{n}}$ of replacing an old element in the reservoir.
  - Apache Spark uses reservoir sampling during the generation of values for range partitioning.
  - **Problem Definition:**
    - Given a stream of n items, we want to sample **s** random items, without replacement and by using uniform probabilities.
    - n is unknown and too large for all n items to fit into main memory.
    - Data items are revealed to the algorithm over time, and the algorithm cannot look back at previous items.
  - **Algorithm:**
    - Store all the first **s** items of the stream to a set **S**
    - Suppose we have seen **n-1** items, and now the $\mathbf{n}^{\mathbf{th}}$ item arrives (**n**>**s**)
      - With probability $\frac{\mathbf{s}}{\mathbf{n}}$, keep the $\mathbf{n}^{\mathbf{th}}$ item, else discard it
      - If we picked the $\mathbf{n}^{\mathbf{th}}$ item, then it replaces one of the s items in the sample S, picked uniformly at random
  - **Claim:**
    - The algorithm maintains a sample S with the desired property: After n items, the sample contains each item seen so far with probability $\frac{\mathbf{s}}{\mathbf{n}}$.
    - Proof By Induction:
      - We assume that after n items, the sample contains each item see so far with probability $\frac{\mathbf{s}}{\mathbf{n}}$
      - We need to show that after seeing the item n+1 the sample maintains the property that

        - Sample contains each element seen so far with probability $\frac{\mathbf{s}}{\mathbf{n+1}}$

- Base case:
    - After we see n=s items, the sample has the desired property
        - Each one of the n=s items is included in the sample with probability $\frac{\mathbf{s}}{\mathbf{s}} = \mathbf{1}$
- Inductive hypothesis:
    - After n items, the sample S contains each item seen so far with probability $\frac{\mathbf{s}}{\mathbf{n}}$
    - Let us now process the new item n+1
- Inductive Step:
    - For items already in S, the probability that the algorithm keeps it in S is:

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

$1 - \frac{\mathrm{s}}{\mathrm{n+1}}$ ➔Estimate n+1 discarded

$\frac{\mathrm{s}}{\mathrm{n+1}}$ ➔Estimate n+1 NOT discarded

$\frac{\mathrm{s-1}}{\mathrm{s}}$ ➔Old elements in the sample NOT picked

- So, at time n, items in S were there with probability $\frac{\mathbf{s}}{\mathbf{n}}$

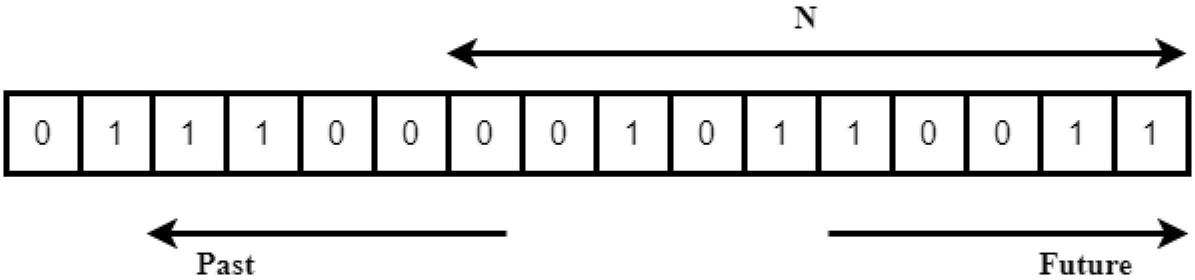
$$\left(1 - \frac{s}{n+1}\right) + \left(\frac{s}{n+1}\right)\left(\frac{s-1}{s}\right)$$
$$= \left(1 - \frac{s}{n+1}\right) + \left(\frac{s-1}{n+1}\right)$$

$$= 1 - \frac{s}{n+1} + \frac{s}{n+1} - \frac{1}{n+1}$$

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

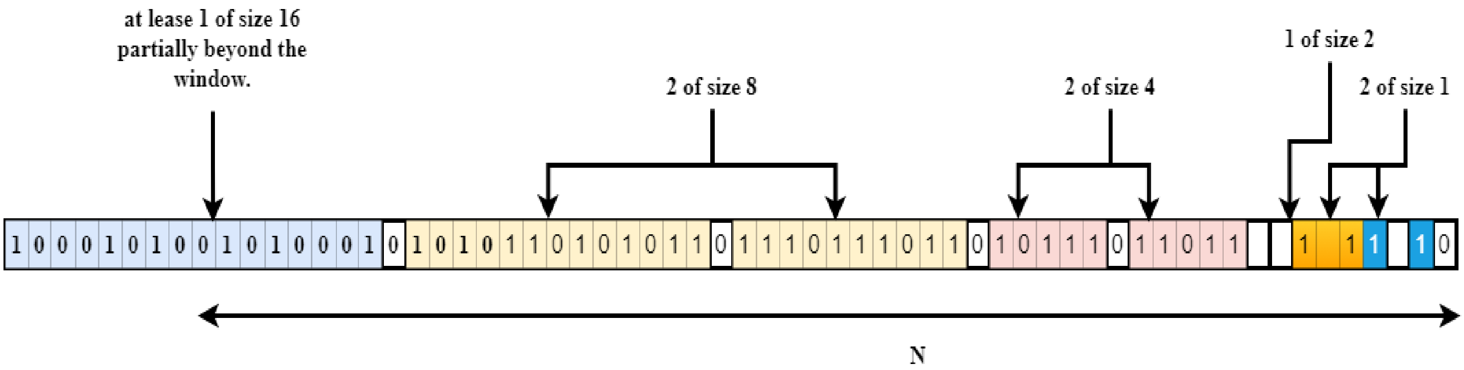
- Time n ➔ n+1, item stayed in S with probability $\mathbf{\frac{n}{n+1}}$

- **Counting Bits Using DGIM Algorithm**
  - For every product **x** we keep 0/1 stream of whether that product was sold in a given transaction
    - How many times have we sold **x** in the last **k** sales?
  - Given is a binary stream with a sliding window of length N
    - How many 1's are in the last N bits?

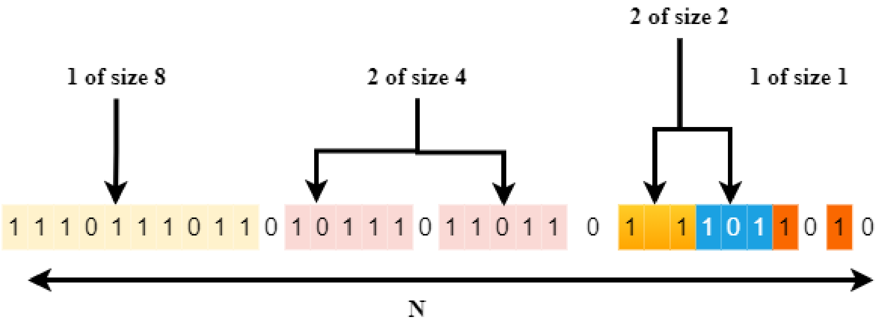


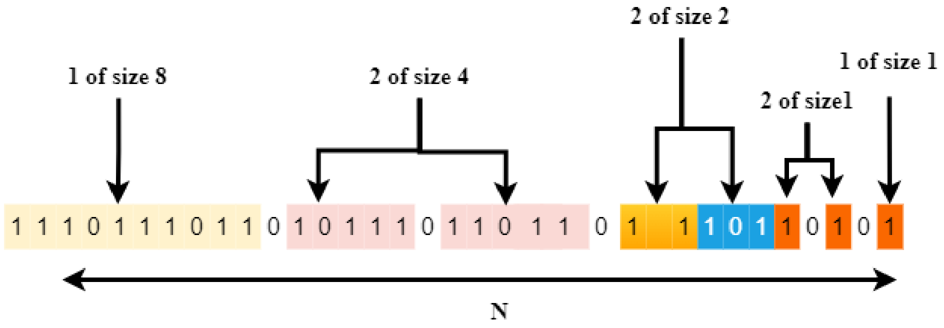
  - **Datar-Gionis-Indyk-Motwani Algorithm (DGIM)**
    - The algorithm only stores $\mathbf{O}(\mathbf{log}^{2}(\mathbf{N}))$
    - The approximate solution is never off by more than 50%
    - The error factor can be further reduced by more complicated algorithm and more stored bits.
    - Allow to estimate the number of 1's in the window with an error of no more than 50%
    - Each bit arrives has a timestamp
    - The window is divided into **buckets** of 1's and 0's
    - Rules for forming the buckets:
      - All buckets should be on **power of 2**: 2^0, 2^1, 2^2,…
      - The **right side** of the bucket **should** always **start with 1** on its right end.
      - Every bucket **should have at least one 1**, otherwise no bucket can be formed.
      - The buckets **cannot be decreased** in size as we read new elements.

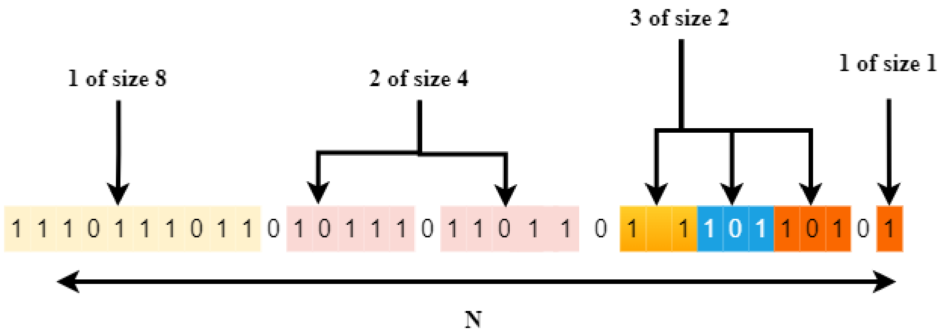
- There are **one or two buckets of the same size up** to some maximum size
- Buckets are **sorted by size**. Earlier buckets are not smaller than later buckets
- Buckets start disappearing when their end time is >N time units in the past.

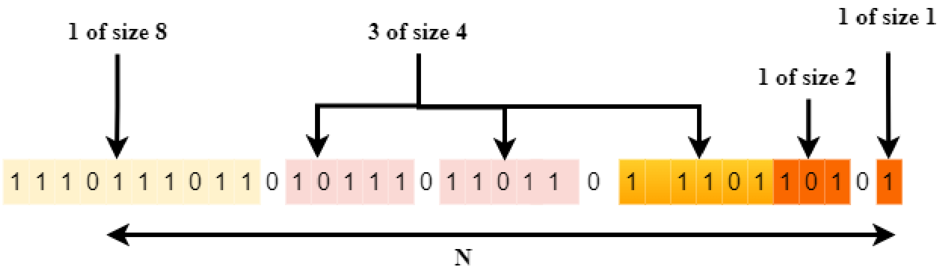


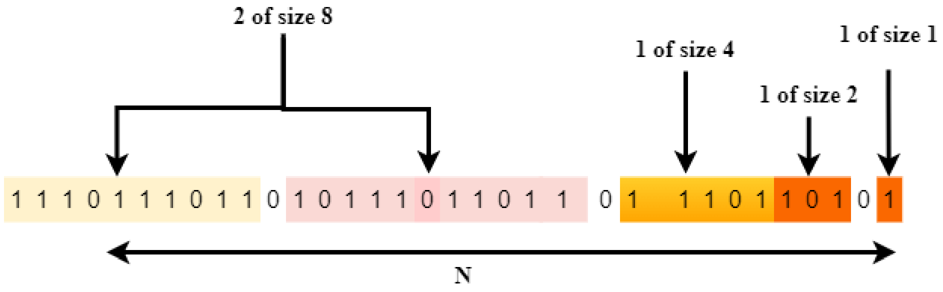
- When a new bit comes in, drop the last bit from the old bucket if its time-end is prior to N time units before the current time. Two cases:
  - If the current bit is 0: No other changes are needed.
  - If the current bit is 1:
    - Create a new bucket of size 1 for just this bit
    - If there are now three buckets of size 1:
      - Combine the oldest buckets of size 1 into a new bucket of size 2.
    - If there are three buckets of size 2:
      - Combine the oldest buckets of size 2 into a bucket of size 4
    - Etc.


2 of size 2
1 of size 8
2 of size 4
1 of size 1
1 1 1 0 1 1 1 0 1 1 0 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 0
N


2 of size 2
1 of size 1
1 of size 8
2 of size 4
2 of size1
1 1 1 0 1 1 1 0 1 1 0 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 0 1
N


3 of size 2
1 of size 8
2 of size 4
1 of size 1
1 1 1 0 1 1 1 0 1 1 0 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 0 1
N


1 of size 1
1 of size 8
3 of size 4
1 of size 2
1 1 1 0 1 1 1 0 1 1 0 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 0 1
N


2 of size 8
1 of size 1
1 of size 4
1 of size 2
1 1 1 0 1 1 1 0 1 1 0 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 0 1
N

- **Why the error is 50%?**
    - Suppose the last bucket has size $\mathbf{2^r}$.
    - If we assume that half of the total number of bits of this bucket are still in the window, we are making an error of at most $\mathbf{2^{r-1}}$
    - In the sliding window, we have at least one bucket of each of the sizes less than $\mathbf{2^r}$, then the total number of bits is:

$$2^0 + 2^1 + 2^2 + \ldots + 2^{r-1} = 2^r - 1$$

⇨ The error is $\frac{2^{r-1}}{2^r-1} = \frac{1}{2}\frac{2^r}{2^r-1} \approx \frac{1}{2} = 50\%$

- **How to reduce the error?**
    - Instead of maintaining 1 or 2 of each bucket, we allow either r-1 or r buckets where r>2.
    - Except of the largest bucket, we can have any number of r-1or r buckets.
    - In the sliding window, we can have up r buckets of each of the sizes less than $\mathbf{2^r}$, then the total number of bits is:

$$r2^0 + r2^1 + r2^2 + \ldots + r2^{r-1} = r(2^r - 1)$$

⇨ The error is $\frac{2^{r-1}}{r(2^r-1)} = \frac{1}{2r}\frac{2^r}{2^r-1} \approx \frac{1}{2r}$

⇨ So, the error is at most $O(\frac{1}{r})$

- By picking r approximately, we can tradeoff between number of bits we store and the error.

- **Finding frequent elements**
  - Applications:
    - High-speed network switch: tokens are packets with source,
    - destination IP addresses and message contents.
    - Each token is an edge in graph (graph streams)
    - Each token in a point in some feature space
    - Each token is a row/column of a matrix
  - Problem Description:
    - The input consists of m objects/items/tokens $S=e_1, e_2,..., e_s$ that are seen one by one by the algorithm where e_i is an element of a universal set U of size n=|U|
    - Let $\mathbf{f_i}$ denote the frequency of an item **i** or number of times **i** is seen in the stream S

Consider the frequency vector:

$$f=(f_1, f_2, \dots, f_n) \text{ where } n = |U|$$

For k>=0 the k'th frequency moment

$$F_k = \sum_{i=1}^{n} f_i^k$$

- Special cases:
  - k=0:

$$F_0 = \sum_{i=1}^{n} f_i^0$$

- F_0 is simply the **number of distinct elements** in stream (**Flajolet-Martin**(**FM)** algorithm)

  - k=1:

$$F_1 = \sum_{i=1}^{n} f_i^1$$

- F_1 is the **length of stream** which is easy

- k=2:

$$F_2 = \sum_{i=1}^{n} f_i^2$$

  - F_2 is **surprise number** is a measure of how **uneven the distribution** is.
  - Example:

    Consider the following set U=(1,2,3,4,5,6,7,8,9,..1000}
    And a stream S of 10 values

    Case 1: S={200, 1,1,1,1,1,1,1,1,1}

    $F_2 = 200^2 + 9x1^2 = 40009$

    Case 2: S={10,10,10,10,10,10, 8, 8, 8, 8}
    $F_2 = 6x10^2+4x8^2=356$

- k=infinity
  - F_∞ is the maximum frequency (heavy hitters prob)

- Direct Method
  - It requires memory of the order $\Omega(N)$ to store m_i for all distinct elements.
  - But we have memory limitations, and requires an algorithm to compute in much lower memory

- **Alon-Matias-Szegedy (AMS) Algorithm (Works on all moments)**
  - AMS works for all moments
  - It gives an unbiased estimate.
  - Let us consider the 2nd moment for now.
  - We pick and keep track of many variables X:
  - For each variable X, we form a key-value pair:

- X.key: The data element i
- X.val: The count of item i

- Note this requires a count in main memory, so the number of Xs is limited
- The objective is to compute:

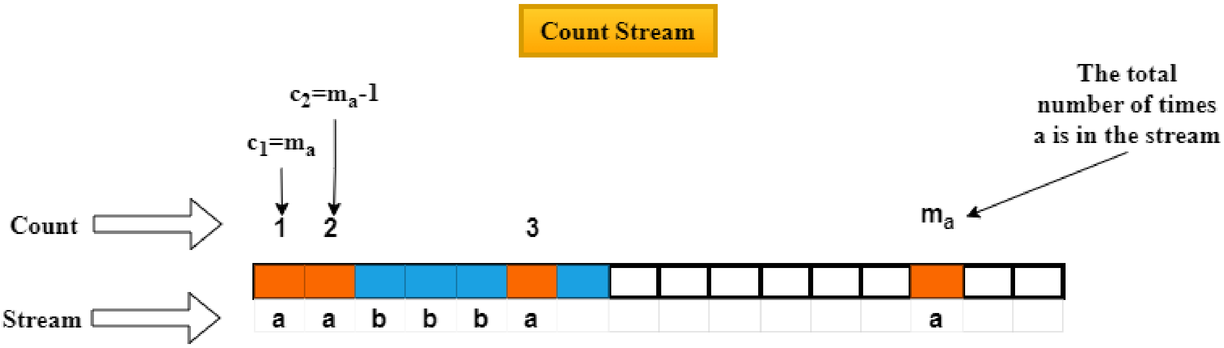
$$S = \sum_i m_i^2$$

  where **mi is the number of times value i occurs** in the stream and **i is the number of distinct elements** in the stream

- Expectation Analysis
  - The second moment is $S = \sum_i m_i^2$
  - Our estimate is
    - **S=f(X)=n(2c-1)**

  - **Let us computer the expectation of our estimate:**
    - **c_t** is the number of times **<u>item at time</u>** t appears from time t onwards (**$c_1=m_a$, $c_2=m_a-1$, $c_3=m_b$**)

      If **m_a** is the total count of a in the stream



mi is the total count of item i in the stream, assuming a stream of length **n**

$$\mathbf{E}[\mathbf{f}(\mathbf{X})] = \frac{\mathbf{1}}{\mathbf{n}}\sum_{\mathbf{t=1}}^{\mathbf{n}} \mathbf{n}(\mathbf{2c}_t - \mathbf{1})$$

Where

Time t when last item i is seen $c_t = 1$
Time t when second last item i is seen $c_t = 2$
…
Time t when the first item i is seen $c_t = m_i$

- **Sum the times by the value seen (By distinct item)**

$$\mathbf{E}[\mathbf{f}(\mathbf{X})] = \frac{\mathbf{1}}{\mathbf{n}}\sum_i \sum_{i=1}^{m_i} n\,(1 + 3 + 5 + \cdots + 2m_i - 1)$$

$$= \frac{\mathbf{1}}{\mathbf{n}}\sum_i n\sum_{i=1}^{m_i}(2i - 1) = \frac{\mathbf{1}}{\mathbf{n}}\sum_i n(\sum_{i=1}^{m_i} 2i - \sum_{i=1}^{m_i} 1)$$

$$= \sum_i 2\frac{m_i(m_i + 1)}{2} - m_i = \sum_i m_i^2 + m_i - m_i = \sum_i m_i^2$$

$$\Rightarrow \mathbf{E}[\mathbf{f}(\mathbf{X})] = \sum_i m_i^2 = S$$

This is the second moment

- High order moments
    - To estimate the kth moment, we use the same algorithm but change the estimate:
    - For k=2, we used n(2.c-1)
    - For k=3, we use: n($3c^2$+-3c+1) where c=X.val
    - Explanation:

- For k=2:
    - We used the following estimate function:
        - **S=f(X) = n(2c-1)**

      And we have shown that $\mathbf{E}[\mathbf{f}(\mathbf{X})] = \sum_i \boldsymbol{m_i}^2 = \boldsymbol{S}$

    - Note that the estimate function:

        - **S=f(X) = n(2c-1) = n(c^2-$(c-1)^2$)**

    - For k = 3:

        - **S=f(X) = n(c^3-$(c-1)^3$)=n($3c^2$-3c+1)**

    - For any k:

        - **S=f(X) = n(c^k-$(c-1)^k$)**

- How do we handle never ending stream?
    - The estimate function we used assume a stream of n items:
      S=f(X) = n(2c-1)
    - Assume we can only store k counts. We must ignore some X's out as time goes on.
    - Objective:
        - Each starting time t is selected with probability $\mathbf{\frac{k}{n}}$
    - Solution:
        - Use fixed-size sampling – Reservoir Sampling
        - Choose the first k times for k variables
        - When the nth element arrives (n>k), choose it with probability$\frac{k}{n}$
        - If you choose it, throw one of the previous stored variable X out with equal probability.