E0 206: Theorist's Toolkit November 10, 2020

Lecture 12-13: Streaming Algorithms

Instructor: Prof. Arindam Khan *Scribes: Sreenivas KVN & Prateek Kumar Sinha*

1 Motivation

In the field of Big Data, the data is often too large to be stored entirely. Moreover, the data arrives very frequently. Examples of such situations include data from large sensor networks, genome sequencing and mining text messages.

Streaming algorithms are the algorithms designed for such situations where huge amount of data arrives in a stream and we need to quickly calculate some function of the data. Consider the following setup in IP traffic network analysis. Popular web pages receive millions of hits per minute. Some of these requests are automated by hackers and hence, we would like to detect and restrict such users. One can think of this data as a stream of requests which are typically made up of IP addresses, data required etc. We would ideally want to filter out the IP addresses with abnormal number of requests. But since our memory is limited, we can't store the entire stream. In fact, the amount of memory at our disposal is often far less than the size of the stream. How can we cleverly use the limited amount of memory and yet, efficiently perform the task at hand? Answering this question is the main objective of the field of streaming algorithms.

2 Formal Definition

We have an input sequence $\sigma = \langle a_1, \dots, a_m \rangle$ where $m \in \mathbb{N}$ and is called the *stream length*. The elements of the stream (a_i's) are taken from the universe $[n]$ where $n \in \mathbb{N}$ and is called the *universe size*. We have a small working memory of size s ($s \ll m, n$).

The input stream is accessed in a streaming fashion i.e., one element arrives at a time. Random access is not allowed i.e., we can't access the i^{th} element whenever we like as in an array. We can look at an element only when it arrives. One scan of the whole stream is called a *pass*.

Following are the main goals of a streaming algorithm.

- Small Memory Requirement: Typically, we want the space requirement of the algorithm to be $o(\min(m, n))$ e.g. $O(\text{polylog}(\min(m, n)))$. However, it is ideal if it is $O(\log m + \log n)$.
- Small Number of Passes: The algorithm should use very few number of passes, typically not more than a specified constant. At times, we allow $\omega(1)$ number of passes too but the ideal case would be when the algorithm requires only one pass.
- Small Update Time: Whenever a new element arrives, the algorithm should be able to update its working memory in a very small amount of time i.e., the computations should be light.
- Good Quality of Solution: In the upcoming lectures, we will see some hardness results that hinder us from computing the exact function that we are required to. Hence, the algorithm is allowed to be a randomized approximation algorithm i.e., it outputs a close enough solution with high probability. There are two prevalently used notions of quality of a randomized approximation algorithm:

– *Multiplicative Approximation*: An algorithm $\mathcal{A}$ (ε, δ)-approximates a function ϕ iff for any input stream σ,

$$\Pr\left[\left|\frac{\mathcal{A}(\sigma)}{\phi(\sigma)} - 1\right| > \varepsilon\right] \leq \delta$$

– *Additive Approximation*: An algorithm $\mathcal{A}$ $(\varepsilon, \delta)^+$-approximates a function ϕ iff for any input stream σ,

$$\Pr\left[\left|\mathcal{A}(\sigma) - \phi(\sigma)\right| > \varepsilon\right] \leq \delta$$

2.1 Different Streaming Models

It often helps to think of the input stream $\sigma = \langle a_1, \ldots, a_m \rangle$ as a multi set of elements and represent it using a *frequency vector* which is denoted as $f = (f_1, \ldots, f_n)$ where f_j $(j \in [n])$ is the number of occurrences of the element j in σ. Whenever a new element, say j, arrives, we increment the value of f_j by 1.

As an example, assume $m = 11, n = 5$ and 10 elements – 1,4,1,4,1,1,3,3,1,4 – have arrived. Then $f = (5, 0, 2, 3, 0)$. Now, if 2 arrives as the eleventh element, the updated frequency vector would be $f = (5, 1, 2, 3, 0)$.

Till now, we assumed that only one element can arrive at a time. But in a more general setting, multiple copies of the same element can arrive at once and moreover, besides arriving elements, there can be elements that depart from the stream. In this setting, we change the definition of m to be the maximum number of elements in the multi set at any point of time i.e., we require the following condition to hold true.

$$|f_1| + \cdots + |f_n| \leq m$$

When elements can arrive or depart, the model is called the *Turnstile* model. In this model, a token in the stream can be thought of as a pair consisting of an element from the universe and the number of copies of it arriving (If this number is negative, we say that it is departing). Formally, any token in the stream $a_i \in [n] \times \{-L, \ldots, L\}$. Here, L is the upper bound on the number of copies of an element that can arrive or depart at once. Whenever a token $a_i = (j, c)$ arrives, we update $f_j \leftarrow f_j + c$.

Remark: Whether any f_j can go below zero depends on the specific setting, but generally we impose the constraint that f_j must always be nonnegative.

When elements can only arrive, the model is called the *Cash Register* model. In this case a token is of the form $a_i = (j, c)$ where $c > 0$.

3 Basic Techniques

There are two most commonly used techniques to design streaming algorithms.

One such technique is called *Sampling.* Since, we can't store the entire data, we cleverly choose a subset of data that will be stored. We do this by retaining each element with some probability. The set of elements retained acts as a proxy to the whole stream. We compute the required function on this proxy data and it hopefully gives a close estimate. The technique of sampling is widely used in practice. One such practical example is the exit polls: We cleverly sample a subset of the population and calculate the statistics to determine who the winner would be. In the current pressing scenario, one would like to estimate the degree of spread of Covid-19 in a particular village: Similar to the exit polls, we pick a subset of the village residents and estimate the number of people affected.

Another useful technique is *Sketching.* In this approach, we create a synopsis of the data seen so far and try to estimate the required function using this data. For example, we can find the genre of a particular movie by reading a short overview instead of watching the entire movie.

Now that we have the fundamentals covered, we will see a few streaming algorithms to estimate important statistics like number of distinct elements in a stream etc.

4 Finding the Majority Element

In the *Majority Problem*, we would like to find if there is an element in the stream which has frequency more than $m/2$ where m is the stream length. Formally, the majority problem is as follows: Let $\sigma = \langle a_1, \ldots, a_m \rangle$ be a stream of elements from the universe $[n]$. Let $f = (f_1, \ldots, f_n)$ be the frequency vector of σ (Hence, $\sum_{i=1}^{n} f_i = m$). Then if there exists j such that $f_j > m/2$, then output j; else output ϕ (the empty set). Note that there can only be one majority element if there exists any.

Although this problem seems very straightforward, we need a lot of space if we want to determine the answer without erring. We will provide the hardness result; a formal proof will be given in the upcoming lectures using results from the field of communication complexity.

Lemma 1. *Any single pass deterministic algorithm that finds the majority element or deduces that no majority element exists in the stream requires a space of* $\Omega(\min(m, n))$, *where* m *is the stream length and* n *is the universe size.*

Lemma 1 entails that we have no choice but to err. The algorithm we will design requires one pass and outputs the majority element if one exists; however it still outputs some element if there is no majority element i.e. it may output a false positive. An informal description of the algorithm is as follows. Think of a fortress which is initially empty. People belonging to multiple groups are standing in a queue who enter the fortress one by one. If a person enters the fortress and finds it to be empty, he/she occupies the fortress. On the other hand, if there are people in the fortress belonging to his/her own group, he/she joins the group. In the last possible scenario where the people in the fortress are of a different group, he/she gets into a fight with one of them and both die. Now, we claim that after the process, the group that remains in the fortress is the majority group (if there was one). This simple majority finding algorithm is called the *Boyer-Moore Majority Voting Algorithm.* The pseudo code is as follows.

Algorithm 1: Boyer-Moore Majority Voting Algorithm

```
cnt ← 0, maj ← null
for $i$ in $[m]$ do
    if cnt = 0 then
        maj ← $a_i$
    end
    if maj = $a_i$ then
        cnt ← cnt + 1
    end
    if maj ≠ $a_i$ then
        cnt ← cnt − 1
    end
end
return maj
```

stream:	a	b	b	c	c	a	a	b	a	a	a	
`maj:`	a	a	b	b	c	c	a	a	a	a	ⓐ	(majority)
`cnt:`	1	0	1	0	1	0	1	0	1	2	3	

stream:	a	a	b	b	c	c	
`maj:`	a	a	a	a	c	ⓒ	(false positive)
`cnt:`	1	2	1	0	1	2	

We will now prove that Algorithm 1 correctly returns the majority element if one exists. Recall the setting of groups and the fortress. Whenever a person enters the fortress, either he finds that the fortress is occupied by the people from his group and so he joins the group; or he finds that the fortress has people from some other group and hence he gets into a fight with one of them and both die in the fight. If at the end of the whole process, the fortress is occupied by a group other than the majority group, then it means that the whole population from the majority group died. But this can't be the case since the majority group constitutes for more than 50% of the entire population and two people die in each fight. This is exactly what happens in the algorithm. The decrement of `cnt` is same as a fight and the increment of `cnt` is same as a person joining his own group. The variable `maj` denotes the current group in the fortress. Hence, at the end of the algorithm, `maj` must contain the majority element if it exists.

Clearly there are only two variables that the algorithm maintains. The variable `maj` contains an element from the universe. Hence it needs $O(\log n)$ bits of memory. One the other hand, `cnt` is at most m. Hence, it needs $O(\log m)$ bits of memory. Hence the overall space complexity of the algorithm is $O(\log m + \log n)$.

One can find the majority element and avoid false positives as well using two passes as follows: Just run the Boyer-Moore majority voting algorithm. Let the returned value be `maj`. In the second pass, check if the frequency of `maj` is greater than half the stream length.

5 Finding Frequently Occurring Elements

Let $k > 2$ be a given constant. Let $\sigma = \langle a_1, \ldots, a_m \rangle$ be input stream. We would like to find all such elements from the universe whose frequency in the stream is more than $m/(k+1)$. Let us call such elements as *frequent* elements. Note that there can be multiple frequent elements in the stream but they can't be more than k in number. The formal objective is, if f is the frequency vector of σ, output all such j in the universe for which $f_j > m/(k+1)$.

We will study an algorithm given by Misra and Gries to find all the frequent elements. In short, let us call this MG-Algorithm. As in the previous case, the MG-Algorithm may output false positives i.e., it may output elements that are not frequent; but it certainly finds all the frequent elements.

Observe that when $k = 1$, this problem is exactly same as the majority problem; hence this problem can be viewed as a generalization of the majority problem. Unsurprisingly, MG-Algorithm is a generalization of the Boyer-Moore majority voting algorithm.

Algorithm 2: Misra-Gries Algorithm to find frequent elements

```
Initialize an empty list $L$
Initialize an empty map cnt (default value of a key is 0)
for $i$ in $[m]$ do
    if $a_i$ is already in the list $L$ then
        cnt$[a_i] \leftarrow$ cnt$[a_i] + 1$
    end
    if $a_i$ is not in $L$ and there are less than $k$ elements in $L$ then
        cnt$[a_i] \leftarrow 1$
        Add $a_i$ to the list $L$
    end
    if $a_i$ is not in $L$ and there are $k$ elements in $L$ already then
        foreach element $e$ in $L$ do
            cnt$[e] \leftarrow$ cnt$[e] - 1$
            if cnt$[e] = 0$ then remove $e$ from $L$
        end
    end
end
return list $L$
```

The following is an example run of the algorithm with $k = 2$.

σ	1	1	2	1	3	3	1	2	1	3	1	2	3	
L	1	1	1, 2	1,2	1	1,3	1,3	1	1	1,3	1,3	1	1,3	(final output)
`cnt`	1	2	2,1	3,1	2	1,1	2,1	1	2	2,1	3,1	2	2,1	

Figure 1: A run of MG-Algorithm on σ when scanned from left to right. Here $k = 2$. If in a particular column, $L = i, j$ and `cnt` $= p, q$, then it means `cnt`$[i] = p$ and `cnt`$[j] = q$. The only frequent element of the stream is 1. The algorithm returns 1 as promised, but it also outputs a false positive in form of 3.

The algorithm provides the following guarantees.

Lemma 2. *Say we run Algorithm 2 on the input stream $\sigma = \langle a_1, \dots, a_m \rangle$ with some arbitrary integer constant $k > 1$. Then*

1. *For all $s \in [n]$, if s is in the list L, then $\widetilde{f_s} :=$ `cnt`$[s] \in [f_s - m/(k+1), f_s]$ where f_s denotes the frequency of s. In other words, the algorithm estimates the frequency of an element within an additive term of $m/(k+1)$.*
2. *If an element s is not in the list L, then $f_s \le m/(k+1)$.*

Proof. First we will prove property 1. It is easy to see that $f_s \ge \widetilde{f_s}$ because we increment `cnt`$[s]$ (if s is in L) only when the element s arrives. To prove $\widetilde{f_s} \ge f_s - m/(k+1)$ we use similar arguments that we used in the case of the Boyer-Moore majority voting algorithm. When an element s arrives and there are k elements in L already, then a total of $(k+1)$ copies of each element are eliminated in the round (k for each element on the list and 1 for s itself). So, for any element s, at most $m/(k+1)$ copies of s are eliminated. Therefore, `cnt`$[s] \ge f_s - m/(k+1)$.

We will prove property 2 by contradiction. Assume that an element s is not in the final list. As proved in the above paragraph, at most $m/(k+1)$ copies of s will be eliminated. So, if $f_s > m/(k+1)$, then s must be on the list, which isn't the case. Hence $f_s \le m/(k+1)$. $\square$

Let's analyze the space complexity. The map `cnt` holds at most k elements each having a value at most m. The list L contains at most k elements from the universe. Hence the overall space occupied is $O(k \log m + k \log n)$.

6 Frequency Moment of Data Streams

Assume $\sigma = \langle a_1, a_2, ...a_m \rangle$ represents the m elements of the stream where $a_i \in U$ and $n = |U|$ represents the size of universe and $f = \langle f_1, f_2, ..., f_n \rangle$ is the frequency vector which represents the frequency of each symbol of the universe with $\sum_{i=1}^{n} f_i = m$
The k^{th} frequent moment of the stream is represented as:

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

The different k^{th} frequent moments which are frequently used and are interpreted as follows:
F_0 : Represents the number of distinct elements in the stream
F_1 : Represents the total number of elements in the stream
F_2 : Represents the non-uniformity in the stream

Example : If all n elements occur with equal frequency m/n, then $F_2 = m^2/n$. If only one element is there in the stream then $F_2 = m^2$

Also, as p becomes very large , ${F_p}^{1/p}$ indicates the frequency of the most frequent element in σ.
Flajolet-Martin gave and algorithm for counting number of distinct elements of the stream (F_0) with two cool tricks : universal hashing and median trick. We will first learn about universal hashing and k-wise independence before estimating F_0.

6.1 Hash Functions

A family of functions $H = \{h | h : N \mapsto M\}$ are called Universal Hash functions such that:

- Every function $h \in H$ is easy to represent.
- $\forall x \in N, h(x)$ is easy to evaluate.
- $\forall S \subseteq N$ with small cardinality, hashed values of items in S have small collisions

6.2 k-wise Independence

Sometimes mutual independence is too much to ask for, and we need to deal with limited dependence.
A set of random variables $X_1, ...X_n$ are said to be k-wise independent if, for any index set $J \subset [n]$ with $|J| \leq k$, and for any values $x_i, i \in J$,

$$Pr\left[\bigwedge_{i \in J} X_i = x_i\right] = \prod_{i \in J} Pr\left[X_i = x_i\right]$$

In particular, X_i's are pairwise independent if they are 2-wise independent, i.e. $\forall i, j, x, y$

$$Pr\left[(X_i = x) \wedge (X_j = y)\right] = Pr\left[(X_i = x)\right].Pr\left[(X_j = y)\right]$$

6.3 Pairwise independent hash functions

A family of functions $H = \{h | h : N \mapsto M\}$ is pairwise independent if :

1. $\forall x \in N$, the random variable h(x) is uniformly distributed in M
2. $\forall x_1 \neq x_2 \in N$, random variables $h_1(x)$ and $h_2(x)$ are independent, i.e. $\forall x_1 \neq x_2 \in N$ and $y_1, y_2 \in M$ and $h \in H$
$$Pr[h(x_1) = y_1 \wedge h(x_2) = y_2] = \frac{1}{|M|^2}$$

Note that Property 2 already implies property 1. This is because for any $x_1, y_1, x_2 (\neq x_1)$, $Pr[h(x_1) = y_1]$ can be written as $\sum_{y_2} Pr[h(x_1) = y_1 \wedge h(x_2) = y_2]$ which is in turn equal to $1/|M|$.

Proposition 3. *Let p be a prime and $h_{ab} = (ax + b) \mod p$. We define $H = \{h_{a,b} | 0 \leq a, b \leq (p-1)\}$. Then H is the family of pairwise independent hash functions.*

Proof. Observe that for all a, b if $h_{a,b}(x_1) = y_1, h_{a,b}(x_2) = y_2$, then

$$ax_1 + b = y_1 \mod p$$
$$ax_2 + b = y_2 \mod p$$

This is the system of two equations and two unknowns has just one solution $(x_1 \neq x_2)$

$$a = (y_2 - y_1)/(x_2 - x_1) \mod p$$
$$b = y_1 - ax_1 \mod p$$

Hence, out of p^2 possible (a, b)'s only one choice results in hashing of x_1 and x_2 into y_1 and y_2. Thus,

$$Pr[h_{a,b}(x_1) = y_1 \wedge h_{a,b}(x_2) = y_2] = 1/p^2$$

□

7 Estimating F_0

Suppose the set D of distinct elements is uniformly distributed among n members of U. Let $p(x) = \max_i\{i : 2^i \text{ divides } x (\neq 0)\}$. Basically it is the number of 0's at the end of binary representation of x (Note : If $\max_x p(x) = r$, then it is likely that the number of distinct integers is close to 2^r). However, the numbers may not be uniformly distributed. Hash functions help in getting the numbers distributed uniformly.
After applying h, all items in D will be uniformly distributed and on average one out of F_0 distinct numbers hit $p(h(x)) \geq \log(F_0)$. So, $\max_x p(h(x)) \approx \log(\text{number of distinct elements})$.

Algorithm 3: Algorithm to estimate F_0

```
Pick a random function $h : [n] \mapsto [n]$ from a family of pairwise independent hash functions;
$z \leftarrow 0$;
while (an element $a_i$ arrives) do
    if $\{p(h(a_i)) > z\}$ then
        $z \leftarrow p(h(a_i))$;
    end
end
return $2^{z+c}$
```

Analysis: Let $X_{r,j}$ be indicator random variable such that $X_{r,j} = 1$ if and only if $p(h(j)) \geq r$. $Y_r = \sum_{j \in U} X_{r,j}$ be the total number of such items. z^* be the value of z when the algorithm terminates. Hence, $Y_r > 0$ if and only if $z^* \geq r$.
Since, h is pairwise independent, $h(j)$ is uniformly distributed, and

$$\begin{aligned} E[X_{r,j}] &= Pr[p(h(j)) \geq r] \\ &= Pr[2^r \text{ divides } h(x)] \\ &= 1/2^n \end{aligned}$$

By Linearity of Expectation,

$$\begin{aligned} E[Y_r] &= \sum_{j \in U} E[X_{r,j}] \\ &= F_0/2^r \end{aligned}$$

Now we will upper bound the variance.

$$\begin{aligned} Var[Y_r] &= \sum_{j \in U} Var[X_{r,j}] \\ &\leq \sum_{j \in U} X^2{}_{r,j} \\ &= \sum_{j \in U} E[X_{r,j}] \\ &= F_0/2^r \end{aligned}$$

Here, the first equality holds as the random variables are pair wise independent and the second equality holds as $X_{r,j}$'s are $0/1$ random variables.
Now, we are interested in the concentration. So, by Markov inequality,

$$\begin{aligned} Pr[Y_r > 0] &= Pr[Y_r \geq 1] \\ &\leq E[y_r]/1 \\ &= F_0/2^r \end{aligned}$$

Also by Chebyshev inequality we get,

$$\begin{aligned} Pr[Y_r = 0] &\leq Pr\left[|Y_r - E[Y_r]| \geq \frac{F_0}{2^r}\right] \\ &\leq \frac{Var[Y_r]}{(F_0/2^r)^2} \\ &\leq \frac{2^r}{F_0} \end{aligned}$$

Let the final output be $F^* = 2^{z^*+c}$. Also, let a be the smallest integer such that $2^{a+c} \geq 3F_0$. Then,

$$\begin{aligned}
Pr[F^* \geq 3F_0] &= Pr[2^{z^*+c} \geq 2^{a+c}] \\
&= Pr[z^* \geq a] \\
&= Pr[Y_a > 0] \\
&\leq \frac{F_0}{2^a} \\
&\leq \frac{2^{a+c}}{3.2^a} \\
&= \frac{2^c}{3}
\end{aligned}$$

Similarly, let b be the largest integer such that $2^{b+C} \leq F_0/3$, we have

$$\begin{aligned}
Pr[F^* \leq F_0/3] &= Pr[2^{z^*+c} \leq 2^{b+c}] \\
&= Pr[z^* \leq b] \\
&= 1 - Pr[z^* \geq b+1] \\
&= 1 - Pr[Y_{b+1} = 0] \\
&\leq \frac{2^{b+1}}{F_0} \\
&\leq \frac{2^{1-c}F_0}{3F_0} \\
&\leq \frac{2^{1-c}}{3}
\end{aligned}$$

We choose $c = 1/2$ to minimize $(2^c + 2^{1-c})/3$.
The guarantees here are weak

- F^* is not arbitrary good approximation of F_0 (it lies in a huge range).
- Success of probability is rather small (One can possibly use larger constant instead of 3 to improve slightly, but the range widens).

So, here we use a much useful median trick.

7.1 The Median Trick

In the median trick, we run k copies of the algorithm in parallel and output the median of k runs. If median exceeds $3F_0$, then atleast $k/2$ of the individual answers must exceed $3f_0$. Let $Z_1, Z_2, ...Z_k$ denote random variables corresponding to the event that i^{th} run answer is $\geq 3F_0$. These are independent trials. So,

$$Pr[Z_i = 1] \leq \frac{\sqrt{2}}{3}$$

$$Let, Z = \textstyle\sum_{i=1}^{k} Z_i, \text{ then } E[Z] \leq k\sqrt{2}/3$$

But from standard Chernoff bounds, we get

$$\begin{aligned}
Pr[Z \geq k/2] &= Pr[Z \geq (3/(2\sqrt{2}).E[Z]] \\
&\leq \left(\frac{e^{(3/(2\sqrt{2})-1)}}{(3/2\sqrt{2})^{(3/2\sqrt{2})}} \right)^{k\sqrt{2}/3} \\
&\approx 2^{-\Omega(k)}
\end{aligned}$$

Similarly, the probability that median is below $F_0/3$ is also $2^{-\Omega(k)}$. Choosing $k = \Theta(\log(1/\delta))$, we can make the sum of two probabilities $\leq \delta$. This gives $(O(1), \delta)$ estimate for F_0.

8 Finding frequent items via Sketching

Let $\sigma = \langle a_1, a_2, ...a_m \rangle$ be the stream which comes , where the universe set U has size n. We will learn about an algorithm to find the minimum size sketch.

8.1 Count-Min Sketch by Cormode and Muthukrishnan

Let us consider a fixed array of counters of width w and depth d. The array is initialized to 0. Each row corresponds to a d-wise hash function $h_i : U \mapsto \{1, 2, ...w\}$. Figure 8.1 represents the idea of this array.

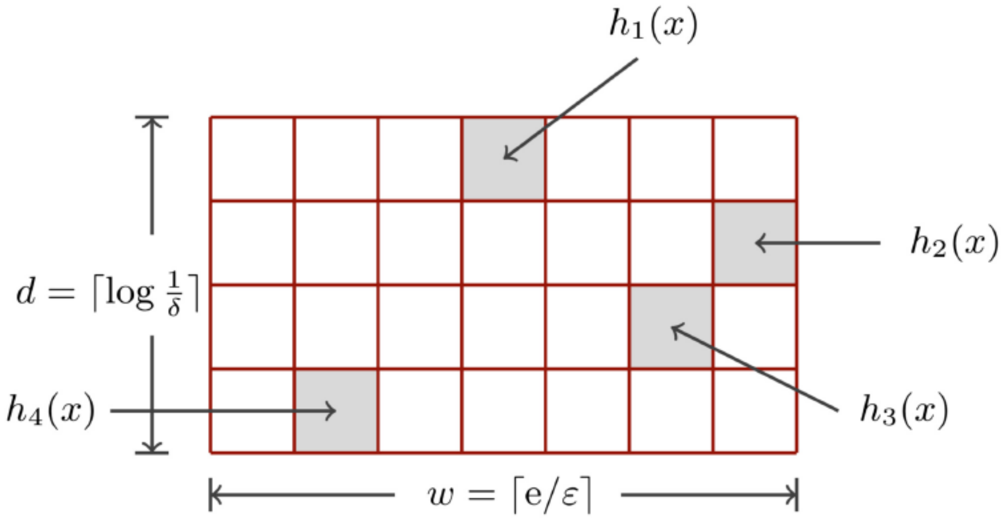
$h_1(x)$
$h_2(x)$
$d = \lceil \log \frac{1}{\delta} \rceil$
$h_3(x)$
$h_4(x)$
$w = \lceil e/\varepsilon \rceil$

Figure 2: Sketch of the Items in the Stream

Algorithm 4: Count Min-Sketch Algorithm

```
$d = \lceil \log(1/\delta) \rceil$, $\omega = \lceil e/\varepsilon \rceil$, $c[1...d][1...\omega] \leftarrow 0$;
Choose d independent hash functions $h_i : [n] \mapsto \omega$ from a d-universal family;
for $i$ = 1 to $m$ do
    if {$a_i$ arrives} then
        for $j$ = 1 to $d$ do
            $c[j, h_j(a_i)] ++$
        end
    end
end
for $k$ = 1 to $n$ do
    return $\widetilde{f_k} = \min_{j=1}^{d} c[j, h_j(k)]$ as the number of occurrences of k
end
```

Note : The size of count-min sketch $o(1/\varepsilon . \log(1/\varepsilon)).(\log(m) + \log(n))$ only depends on $\varepsilon and \delta$ and is independent of the number of symbols in the universe.

Theorem : For any item $i \in U$ with frequency f_i the estimator $\widetilde{f} \in [f_i, f_i + \varepsilon.F_1]$ with probability $\geq (1-\delta)$, where F_1 is the first moment of σ.

Proof: Foe each of f_i instances of i, we incremented $c[j, h_j(i)] \implies \widetilde{f_i} \geq f_i$. Now, we will show that for a fixed i,

$$\widetilde{f_i} \leq f_i + \varepsilon.F_1 \text{ with probability } \geq (1-\delta)$$

So, we have to show that there are not many false positives, i.e. $k \in [n]$
i for which $h_j(k) = h_j(i)$.
Let, Y_{jk} be the indicator random variable for the event "$h_j(k) = h_j(i)$". Now if $Y_{jk} = 1$, it adds f_k to $c[j, h_j(i)]$. So, the total error for $c[j, h_j(i)$ is

$$X_j = \textstyle\sum_{k \in [n] \setminus i} f_k . Y_{jk}$$

As h_j is chosen from d-universal family,

$$\begin{aligned} E[Y_{jk}] &= 1/\omega \\ &\leq \varepsilon / e \end{aligned}$$

Hence, by linearity of expectations,

$$\begin{aligned} E[X_j] &= \sum_{k \in [n] \setminus i} f_k . E[Y_{jk}] \\ &\leq \frac{\varepsilon}{e} \sum_{k \in [n] \setminus i} f_k \\ &\leq \frac{F_1 \varepsilon}{e} \end{aligned}$$

This shows that the expected error is as small as

$$\begin{aligned} E[c[j, h_j(i)]] &= E[f_i + X_j] \\ &= E[f_i] + E[X_j] \\ &\leq f_i + \frac{\varepsilon F_1}{e} \end{aligned}$$

Now, we will show the concentration. As each $f_k \geq 0$, we obtain

$$\begin{aligned} Pr[\widetilde{f}_i > f_i + \varepsilon.F_1] &= Pr[\forall j : c[j, h_j(i)] > f_i + \varepsilon.F_i] \\ &= Pr[\forall j : f_i + X_j > f_i + \varepsilon.F_i] \\ &= Pr[\forall j : X_j > \varepsilon.F_j] \\ &\leq Pr[\forall j : X_j > e.E[X_j]] \\ &\leq \prod_{j=1}^{d} Pr[X_j > e.E[X_j]] \\ &\leq (1/e)^d \\ &\leq \delta \end{aligned}$$

where the third last and second last inequality hold by the property of d-wise independent hash and Markov's inequality respectively. Also the last last inequality holds as $d = \log(1/\delta)$