Streaming V : Counting Distinct Elements II[1]

- In the previous lecture, we saw that FLAJOLET-MARTIN gives a $O(1)$-approximation to the number of distinct elements, even with the use of pairwise independent hash functions. What if we desire a better approximation? Note that our proof of the $O(1)$ approximation was not via the usual "unbiased-estimate + low-variance + medians-of-means" method. Indeed, the estimate is not unbiased. As I mentioned in the previous lecture, Flajolet and Martin actually proved that a scaled version of their estimator is indeed (close to) an unbiased one and they also bound their variance, and also prove that taking averages tends to give a $(1 \pm \varepsilon)$ approximation to F_0. In this lecture, we will see a different modification of the algorithm which gives an $(1 \pm \varepsilon)$-estimate. This algorithm is one of three F_0-estimation algorithms in a paper by Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, D. Sivakumar, and Luca Trevisan.

- The main idea stems from an observation made last lecture. Let us recall the definitions. For any integer $1 \le r \le L$ (where $L := \lceil \lg n \rceil$) and every element e in the stream, we denote $X_{e,r} = 1$ if $h(e)$ contained $\ge r$ trailing zeros, and $X_{e,r} = 0$ otherwise. We denoted $Y_r := \sum_{e \in D} X_{e,r}$ as the number of distinct elements in the data stream that have $\ge r$ trailing zeros. We observed that (and this needed only pairwise independence in the hash family)

$$\text{For all } 1 \le r \le L,\ \ \mathbf{Exp}[Y_r] = \frac{d}{2^r} \quad \text{and} \quad \mathbf{Var}[Y_r] = \frac{d}{2^r} \cdot \left(1 - \frac{1}{2^r}\right) \tag{1}$$

  And thus, the scaled random variable $Z_r := 2^r Y_r$ is an unbiased estimate of d. Therefore, by Chebyshev, if we want an $(1 \pm \varepsilon)$-estimate to d, we need to focus on the r such that $2^r \le \varepsilon^2 d$. More precisely, Chebyshev gives us

$$\text{For all } 1 \le r \le L,\ \ \mathbf{Pr}\left[\left|2^r Y_r - d\right| \ge \varepsilon d\right] = \mathbf{Pr}\left[\left|Y_r - \frac{d}{2^r}\right| \ge \frac{\varepsilon d}{2^r}\right] \ < \ \frac{2^r}{\varepsilon^2 d} \tag{2}$$

  Why not then just pick an r which is small enough? The flip is the size required to evaluate Y_r. How do we keep track of Y_r? For instance, if $r = 1$, then Y_r is the number of distinct elements which have ≥ 1 leading zeros. But that's going to be $\approx \frac{1}{2}$ the elements. And note that we need to actually *store* the elements to keep track : when the same copy of a previously seen element arrives, we need to make sure not to double count. So, we need to make sure r is "big enough" such that Y_r itself is not too large. In particular, we want to choose r such that $2^r \approx \varepsilon^2 d$; but this can't be a priori fixed. One needs to be more ingenious.

  > **Remark:** *Before moving on, however, note that a "two pass" algorithm leaps out at us. That is, suppose we could make two scans across the whole stream. Then, in the first scan/stream find the estimate r by* FLAJOLET-MARTIN *with the guarantee $\frac{d}{8} \le 2^r \le 4d$. In the second pass, keep track of Y_r. That is, explicitly store all elements with $\ge r$ trailing zeros. We expect this number to be $O(1)$, and so the total storage will be $O(\log n)$ bits. The variance-by-squared-mean is also a constant, which means $O(\frac{1}{\varepsilon^2}\ln(1/\delta))$* parallel *repetitions will suffice to give us an*

[1] *Lecture notes by Deeparnab Chakrabarty. Last modified : 19th April, 2021*
These have not gone through scrutiny and may contain errors. If you find any, or have any other comments, please email me at deeparnab@dartmouth.edu. Highly appreciated!

(ε, δ)*-approximation to* d.

- The idea behind the one-pass algorithm is to start with a "small" r and then keep bumping it up as we go. Since we are only in the insertion-only setting, the number of distinct elements cannot go down. To explain this, let us go back to our "wasteful" description of the FLAJOLET-MARTIN algorithm. Imagine, we have L counters, or rather more precisely, buckets. When an item e arrives, we evaluate $r(e)$ and then plonk it into the $C[r(e)]$th bucket ***if*** it is not already present. The presence check can be done via a linear scan (at the end, these buckets will be small, so no real need to be clever here, but one could use a binary search tree or a hash table). Note that for any r, the sum of the sizes of the buckets with index bigger than equal to r is precisely Y_r. Worth writing this explicitly:

  $$\text{For any } 1 \le r \le L, \;\; Y_r = \sum_{j \ge r} |C[j]|$$

  As described so far, the algorithm stores all the elements. But here is the kicker: at any point of time, the algorithm only stores buckets with index $\ge r$ such that $Y_r \le \frac{c}{\varepsilon^2}$ for some constant c. In doing so, it ensures that the algorithm never stores more than $\frac{c}{\varepsilon^2}$ elements, and thus never uses more than $O(\frac{c}{\varepsilon^2})$-words of memory (or $O(\frac{c \log n}{\varepsilon^2})$ bits).

  More precisely, the algorithm maintains a bucket index rmin initialized to 1 and only stores the buckets $C[j]$ with $j \ge \mathsf{rmin}$. So, when an item e arrives and $r(e) < \mathsf{rmin}$, the algorithm ignores the element. The algorithm also maintains Y_{rmin} as defined above. Initially, this is 0. If $r(e) \ge \mathsf{rmin}$, then it adds e to $C[r(e)]$ if e is not already present. This increases Y_{rmin} by 1. If Y_{rmin} exceeds $\frac{c}{\varepsilon^2}$, then the algorithm deletes $C[\mathsf{rmin}]$ from memory, updates Y_{rmin}, and increments rmin by 1. At the end of the stream, this rmin is the "r" we wanted: the algorithm outputs $\mathsf{est} \leftarrow 2^{\mathsf{rmin}} \cdot Y_{\mathsf{rmin}}$.

1: **procedure** BASIC-BJKST:
2: Choose $h : [n] \to [2^L]$ from a strongly universal hash family as in FLAJOLET-MARTIN.
3: $C[1 : L]$ is a list of buckets.
4: $\mathsf{rmin} \leftarrow 1$, $Y_{\mathsf{rmin}} \leftarrow 0$.
5: **for** arrival of element e **do**:
6: Evaluate $r(e)$: the number of trailing zeros in $h(e)$.
7: **if** $r(e) < \mathsf{rmin}$ **then**:
8: Ignore this element e.
9: **else**:
10: If e is already present in $C[r(e)]$ ignore this element.
11: Otherwise, add e to $C[r(e)]$ and update $Y_{\mathsf{rmin}} \leftarrow Y_{\mathsf{rmin}} + 1$. ▷ *Note that $Y_{\mathsf{rmin}} = \sum_{j \geq \mathsf{rmin}} |C[j]|$ is maintained.*
12: ▷ *This increase in Y_{rmin} may make it $> \frac{c}{\varepsilon^2}$. The next while loop fixes this.*
13: **while** $Y_{\mathsf{rmin}} > \frac{c}{\varepsilon^2}$ **do**:
14: Delete $C[\mathsf{rmin}]$ and update $Y_{\mathsf{rmin}} \leftarrow Y_{\mathsf{rmin}} - |C[\mathsf{rmin}]|$.
15: Update $\mathsf{rmin} \leftarrow \mathsf{rmin} + 1$. ▷ *Note that $Y_{\mathsf{rmin}} = \sum_{j \geq \mathsf{rmin}} |C[j]|$ is maintained.*
16: ▷ *Note: if at the beginning of the while loop, $C[\mathsf{rmin}] > 0$, then it runs only once. But Line 16 may lead rmin to point to a bucket with $C[\mathsf{rmin}] = 0$.*
17: **return** $\mathsf{est} \leftarrow 2^{\mathsf{rmin}} \cdot Y_{\mathsf{rmin}}$.

Observation 1. Since items are only inserted, throughout the algorithm Y_{rmin} equals $\sum_{e \in D} X_{e,\mathsf{rmin}}$, which is the number of elements in D whose hash has $\geq \mathsf{rmin}$ trailing zeros.

- *Analysis of Quality.*

Theorem 1. The estimate est returned by BASIC-BJKST satisfies $(1-\varepsilon)d \leq \mathsf{est} \leq (1+\varepsilon)d$ with probability $\geq \frac{3}{4}$.

Proof. If you have followed the intuition of the algorithm, then you probably see that by design the worry of rmin being "too small" is allayed. The algorithm never stores more than $O(1/\varepsilon^2)$ items. The worry, if any, is whether rmin became too big. Or rather, can $2^{\mathsf{rmin}} \gg \varepsilon^2 d$? The answer is no. Consider the time when rmin is incremented from some k to $k+1$. At that point, we must have $Y_k > \frac{c}{\varepsilon^2}$. Can k be such that $2^k \gg \varepsilon^2 d$? No, because $\mathbf{Exp}[Y_k] = \frac{d}{2^k}$ with variance also of that order, and so whp $2^k \approx \frac{\varepsilon^2 d}{c}$ in this case. So, the final rmin we output, will almost sure be such that $\mathbf{Var}[Y_{\mathsf{rmin}}]/\,\mathbf{Exp}^2[Y_{\mathsf{rmin}}]$ is small, and therefore, the estimate $2^{\mathsf{rmin}} \cdot Y_{\mathsf{rmin}}$ should be a good one.

To make the above proof formal, there is a bit of care needed. Note that we can't simply argue about Y_{rmin} as it is a random variable indicated by a random variable. While, (1) is for random variables with a fixed index. So, a bit more care is needed. To do so, we go over *all* L possibilities of rmin, and argue that the probability of something bad is happening is small. What is bad? Well, the bad event for us is the following: $\mathsf{rmin} = k$ but $2^k Y_k$ is not a good estimate. Let's define this as an event:

$$\text{For } 1 \leq k \leq L, \quad \mathcal{E}_k := \{\mathsf{rmin} = k \;\; \wedge \;\; |2^k Y_k - d| \geq \varepsilon d\}$$

We want to prove, $\mathbf{Pr}[\bigvee_{k=1}^{L} \mathcal{E}_k] < \frac{1}{4}$. We will proceed by union bound. We break the k's into "small" and "large". To define this, let k_* to be the *largest* integer such that $\frac{2^{k_*}}{d} \leq \frac{\varepsilon^2}{16}$. Then, using Equation (2) and the union bound, we get

$$\mathbf{Pr}[\bigvee_{k\leq k_*} \mathcal{E}_k] \underbrace{\leq}_{\mathbf{Pr}[A\wedge B]\leq \mathbf{Pr}[B]} \mathbf{Pr}[\bigvee_{k\leq k_*} \{|2^k Y_k - d| \geq \varepsilon d\}] \underbrace{\leq}_{\text{Union Bound and (2)}} \frac{1}{\varepsilon^2 d}\sum_{k=1}^{k_*} 2^k \leq \frac{1}{8} \tag{3}$$

What about the "large" k's? Well, we bound $\mathbf{Pr}[\bigvee_{k>k_*} \mathcal{E}_k] \leq \mathbf{Pr}[\mathsf{rmin} > k_*]$. Which means that $Y_{k_*} > \frac{c}{\varepsilon^2}$. By our choice of k_*, we know that $\frac{2^{k_*}}{d} > \frac{\varepsilon^2}{32}$. That is, $\mathbf{Exp}[Y_{k^*}] = \frac{d}{2^{k^*}} < \frac{32}{\varepsilon^2}$, and thus just Markov gives us that

$$\mathbf{Pr}[Y_{k_*} > \frac{c}{\varepsilon^2}] \leq \frac{\mathbf{Exp}[Y_{k^*}]}{c/\varepsilon^2} < \frac{32}{c} < \frac{1}{8} \quad \text{if } c \text{ is large enough.}$$

Putting everything together, we get

$$\mathbf{Pr}[\bigvee_{k>k_*} \mathcal{E}_k] \underbrace{\leq}_{\mathbf{Pr}[A\wedge B]\leq \mathbf{Pr}[B]} \mathbf{Pr}[\mathsf{rmin} > k_*] = \mathbf{Pr}[Y_{k^*} > \frac{c}{\varepsilon^2}] \underbrace{\leq}_{\text{if } c \text{ large enough}} \frac{1}{8} \tag{4}$$

Union bounding over Equation (3) and Equation (4) gives that $\mathbf{Pr}[\bigvee_{k=1}^{L} \mathcal{E}_k] < \frac{1}{4}$. □

- *Time and Space: a space saving trick by BJKST.* Per update, the algorithm spends time evaluating $r(e)$. After that, the majority of the time taken is in checking if e is already in $C[r(e)]$. Since $|C[r(e)]| \leq Y_{\mathsf{rmin}} \leq \frac{c}{\varepsilon^2}$, this takes at most $O(\frac{1}{\varepsilon^2})$ time. One can do *much* better though : either $O(\lg \frac{1}{\varepsilon})$ by storing $C[r(e)]$ as a binary search tree, or even $O(1)$ amortized time by just hashing. This forms the lion's share of the update time.

  How about space? There is some space required to store the hash functions. This is $O(\log n)$ bits or $O(1)$ words. The bigger usage of space is the buckets. Note, we only store the buckets for $j \geq \mathsf{rmin}$, and we maintain $|Y_{\mathsf{rmin}}| \leq \frac{c}{\varepsilon^2}$. Therefore, the space required is $O(\frac{\log n}{\varepsilon^2})$. This doesn't sound too bad, till you compare with the space required by FLAJOLET-MARTIN: ignoring the hash function, the space usage of that algorithm was only $\lg\lg n + O(1)$ bits. Next, we discuss a space-saving trick by BJKST which takes motivation from the birthday paradox.

  The main idea is again hashing. Note that we don't really need to store the element e in the bucket. We just need to make sure when another copy of e arrives we don't count it again. So instead of storing e, we just store a hash $g(e)$ for some hash function $g : [n] \to [s]$. The question is how big does s need to be? If we wanted no collisions at all, that is we wanted g to be perfect, then the constructions we studied were of size $s = O(n)$. But this defeats the purpose – storing $g(e)$ would take the same amount of space.

  But then we realize that the elements we desire no collision on are the ones *ever* present in the buckets. So we ask ourselves : how many distinct elements $e \in [n]$ are ever present in the buckets? Crudely, for every r, the size of the bucket $C[r]$ is $\leq \frac{c}{\varepsilon^2}$, and there are $L = O(\lg n)$ possible such r's. Thus, in the run of BASIC-BJKST, if we consider the (random) set $S \subseteq [n]$ that is hashed into the buckets, this size $|S| = O(\frac{\lg n}{\varepsilon^2})$. Therefore, we don't need the range of g to be large. If $g : [n] \to [s]$ where $s = \frac{b\lg^2 n}{\varepsilon^4}$, then by a birthday-paradox style argument, the probability there is a collision among the elements is S is $\leq \frac{1}{12}$ for a large enough b. One can add this to the failure probability, and get the success probability of the full algorithm to be $\geq \frac{2}{3}$.

1: **procedure** BJKST:
2: Choose $h : [n] \to [2^L]$ from a strongly universal hash family as in FLAJOLET-MARTIN.
3: Choose $g : [n] \to [\frac{b \log^2 n}{\varepsilon^4}]$ from a UHF.
4: $C[1 : L]$ is a list of buckets. ▷ *In reality, one uses a data structure which can dynamically store buckets indexed via a key*
5: $\mathsf{rmin} \leftarrow 1$, $Y_{\mathsf{rmin}} \leftarrow 0$.
6: **for** arrival of element e **do**:
7: Evaluate $r(e)$: the number of trailing zeros in $h(e)$.
8: **if** $r(e) < \mathsf{rmin}$ **then**:
9: Ignore this element e.
10: **else**:
11: If $g(e)$ is already present in $C[r(e)]$ ignore this element. ▷ *Use a binary search tree, or another hash table*
12: Otherwise, add $g(e)$ to $C[r(e)]$ and update $Y_{\mathsf{rmin}} \leftarrow Y_{\mathsf{rmin}} + 1$.▷ *Note that* $Y_{\mathsf{rmin}} = \sum_{j \geq \mathsf{rmin}} |C[j]|$ *is maintained.*
13: ▷ *This increase in* Y_{rmin} *may make it* $> \frac{c}{\varepsilon^2}$. *The next while loop fixes this.*
14: **while** $Y_{\mathsf{rmin}} > \frac{c}{\varepsilon^2}$ **do**:
15: Delete $C[\mathsf{rmin}]$ and update $Y_{\mathsf{rmin}} \leftarrow Y_{\mathsf{rmin}} - |C[\mathsf{rmin}]|$.
16: Update $\mathsf{rmin} \leftarrow \mathsf{rmin} + 1$. ▷ *Note that* $Y_{\mathsf{rmin}} = \sum_{j \geq \mathsf{rmin}} |C[j]|$ *is maintained.*
17: ▷ *Note: if at the beginning of the while loop,* $C[\mathsf{rmin}] > 0$, *then it runs only once. But Line 16 may lead* rmin *to point to a bucket with* $C[\mathsf{rmin}] = 0$.
18: **return** $\mathsf{est} \leftarrow 2^{\mathsf{rmin}} \cdot Y_{\mathsf{rmin}}$.

Lemma 1. For *any* subset $S \subseteq [n]$, the probability there exists $e, e' \in S$ such that $g(e) = g(e')$ is at most $\frac{|S|^2 \varepsilon^4}{2b \lg^2 n}$.

Proof. The probability of collision for a fixed pair is $\leq \frac{\varepsilon^4}{b \lg^2 n}$ since g is drawn from a UHF. The lemma follows by a union bound over the $\leq |S|^2/2$ different pairs. □