CS 170: Efficient Algorithms and Intractable Problems Spring 2020

Lecture Notes — April 21, 2020

Profs. Alessandro Chiesa and Jelani Nelson

Streaming Algorithms

Recall that a *data structure* is simply a method for laying out data in memory to support efficiently answering (certain types of) queries, as well as possibly updates. For example, for the union-find problem the disjoint forest data structure laid out the "data", i.e. the set of unions that had been done, in such a way as to allow for efficient answering of Find queries and Union updates. A data structure is called *dynamic* if it supports both updates and queries, and *static* if it only supports queries.

The term *streaming algorithm* simply refers to a dynamic data structure (data is updated and queried) with the connotation that the data structure should only use *sublinear* memory. That is, if it takes some number b bits of memory to represent the data, we would like our data structure to still allow answering the desired queries while only using $o(b)$ bits.

As a simple example, consider the case that some online vendor performs a sequence of sales and wants to estimate gross revenue. That is, we would like to allow for inserting new sale transactions into a database while supporting a single query: return the total gross sales so far. Even if there are n transactions, there is a clear solution using $o(n)$ memory: initialize a counter C to zero, and every time there is a sale for some price p, perform the update $C \leftarrow C + p$. Thus our total memory usage is only a single counter, which does not depend on n! This is a very simple example, but as we shall soon see, there are other examples for which non-trivial solutions exist. In this lecture, we focus on the problems of *approximate counting* and estimating the number of *distinct elements*[1].

1 Approximate Counting

Problem. Our algorithm must monitor a sequence of events, then at any given time output (an estimate of) the number of events thus far. More formally, this is a data structure maintaining a single integer n (initially set to zero) and supporting the following two operations:

- **update():** increments n by 1
- **query():** must output (an estimate of) n

Thus n is just the number updates so far. Before any operations are performed, it is assumed that n starts at 0. Of course a trivial algorithm maintains n using $\lceil \log n \rceil$ bits of memory (a counter). Our goal is to use much less space than this. It is not too hard to prove that it is impossible to solve this problem *exactly* using $o(\log n)$ bits of space. More specifically, if the query algorithm just

[1]The exposition below makes heavy usage of previous notes on the matter written by Prof. Nelson at `http://people.eecs.berkeley.edu/~minilek/tum2016/index.html`

```
// increment counter only every other update
```
1. **init():** $X \leftarrow 1$, parity$\leftarrow 0$
2. **update():**
   (a) **if** parity$= 1$: $X \leftarrow X + 1$
   (b) parity $\leftarrow 1 -$ parity
3. **query(): return** $2X$

Figure 1: Incrementing every other update.

knows that the answer is some integer between 0 and N, then since each memory configuration corresponds to a different answer and as there are only 2^S distinct memory configurations with S bits, we must have $2^S \geq N$, i.e. $S \geq \lceil \log N \rceil$. Thus we would like to answer **query()** with some *estimate* $\tilde{n}$ of n satisfying

$$\mathbb{P}(|\tilde{n} - n| > \varepsilon n) < \delta, \tag{1}$$

for some $0 < \varepsilon, \delta < 1$ that are given to the algorithm up front. That is, we relax the requirement for exact maintenance of n to only approximate maintenance. Here the probability is taken over the randomness of the algorithm, so that we are trying to develop a randomized Monte Carlo solution that has some probability of failure.

How might we do better though than using $\lceil \log n \rceil$ bits to store n in memory? Imagine the following idea: what if we did not store n, but rather stored a counter $X = \lfloor n/2 \rfloor$. Then to answer queries, we simply return $\tilde{n} = 2X$. Since $n/2$ takes one less bit to store than n, we have saved a bit! Plus, our answer is only ever off by at most an additive 1 (when n is odd). This idea, of course, has a major problem: to implement this, we need to increment X every *other* update; say, we do so on the even updates but not the odd ones. But how can we know the parity of the number of updates so far? Well, we could store that in an extra bit (see Figure 1! But wait, of course this is quite silly since know we're back to using $\lceil \log n \rceil$ bits again. An idea around this is to use randomness: during each update we simply *flip a coin.* If heads, we increment X; else if tails, we do nothing. Then X is a random variable with $\mathbb{E}\, X = n/2$, so $\tilde{n} = 2X$ is equal to n in expectation.

The above idea is a good starting point, but has two main issues which we address shortly. The first issue is that we only were able to save one bit of memory! However we want to use truly sublinear memory, i.e. $o(\log n)$ bits. The second issue is that recall we want $|n - \tilde{n}| \leq \varepsilon n$. In other words, we want $\tilde{n}$ to be in the range $[(1 - \varepsilon)n, (1 + \varepsilon)n]$. Think of ε being say 0.01, so that we want 1% relative error. The described method cannot possibly achieve this when n is small. For example, consider what happens after a single update, so that $n = 1$. Either the first coin flip will be heads, in which case we have $X = 1$, or it will be tails, in which case $X = 0$. Thus we will either have $\tilde{n} = 2$ or $\tilde{n} = 0$, neither of which is within 1% of the true value $n = 1$.

Both these issues were remedied by an algorithm of Robert Morris [2], provides an estimate $\tilde{n}$ satisfying (1) for some ε, δ that we will analyze shortly. The algorithm works as follows:

1. Initialize $X \leftarrow 0$.
2. For each update, increment X with probability $\frac{1}{2^X}$.

3. For a query, output $\tilde{n} = 2^X - 1$.

Intuitively, the variable X is attempting to store a value that is $\approx \log_2 n$.

1.1 Analysis of Morris' algorithm

Let X_n denote X in Morris' algorithm after n updates.

Claim 1.

$$\mathbb{E}\, 2^{X_n} = n + 1.$$

Proof. We prove by induction on n. The base case is clear, so we now show the inductive step.

We have

$$\begin{aligned}
\mathbb{E}\, 2^{X_{n+1}} &= \sum_{j=0}^{\infty} \mathbb{P}(X_n = j) \cdot \mathbb{E}(2^{X_{n+1}} | X_n = j) \\
&= \sum_{j=0}^{\infty} \mathbb{P}(X_n = j) \cdot \left(\overbrace{2^j(1 - \frac{1}{2^j})}^{\text{don't increment } X} + \overbrace{\frac{1}{2^j} \cdot 2^{j+1}}^{\text{increment } X} \right) \\
&= \sum_{j=0}^{\infty} \mathbb{P}(X_n = j) 2^j + \sum_{j=0}^{\infty} \mathbb{P}(X_n = j) \\
&= \mathbb{E}\, 2^{X_n} + 1 \\
&= (n+1) + 1
\end{aligned} \tag{2}$$

□

It is now clear why we output our estimate of n as $\tilde{n} = 2^X - 1$: it is an unbiased estimator of n. That is, $\mathbb{E}\, \tilde{n} = n$. However we do not only want to know that our answer is correct in expectation; we want to say also that it is close to the true answer with good probability. For this, we use Chebyshev's inequality:

Lemma 2 (Chebyshev)**.**

$$\forall \lambda > 0, \mathbb{P}(|X - \mathbb{E}\, X| > \lambda) < \frac{\mathbb{E}(X - \mathbb{E}\, X)^2}{\lambda^2} := \frac{\mathrm{Var}[X]}{\lambda^2} \tag{3}$$

Proof. $\mathbb{P}(|X - \mathbb{E}\, X| > \lambda) = \mathbb{P}((X - \mathbb{E}\, X)^2 > \lambda^2)$, and thus the claim follows by Markov's inequality. □

Recall that the value $\mathbb{E}(X - \mathbb{E}\, X)^2$ is called the *variance* of X, $\mathrm{Var}[X]$. Now in order to show (1), we will also control on the variance of Morris' estimator. This is because, by Chebyshev's inequality,

$$\mathbb{P}(|\tilde{n} - n| > \varepsilon n) < \frac{1}{\varepsilon^2 n^2} \cdot \mathbb{E}(\tilde{n} - n)^2 = \frac{1}{\varepsilon^2 n^2}\, \mathbb{E}(2^X - 1 - n)^2. \tag{4}$$

When we expand the above square, we find that we need to control $\mathbb{E}\, 2^{2X_n}$. The proof of the following claim is by induction, similar to that of Claim 1.

Claim 3.

$$\mathbb{E}(2^{2X_n}) = \frac{3}{2}n^2 + \frac{3}{2}n + 1. \tag{5}$$

Proof. We again prove this by induction. It is clearly true for $n = 0$. Then

$$\begin{aligned}
\mathbb{E}2^{2X_{n+1}} &= \sum_j \mathbb{P}(2^{X_n} = j) \cdot \mathbb{E}(2^{2X_{n+1}} \mid 2^{X_n} = j) \\
&= \sum_j \mathbb{P}(2^{X_n} = j) \cdot \left(\frac{1}{j} \cdot 4j^2 + \left(1 - \frac{1}{j}\right) \cdot j^2\right) \\
&= \sum_j \mathbb{P}(2^{X_n} = j) \cdot (j^2 + 3j) \\
&= \mathbb{E}2^{2X_n} + 3 \cdot \mathbb{E}2^{X_n} \\
&= \left(\frac{3}{2}n^2 + \frac{3}{2}n + 1\right) + (3n + 3) \\
&= \frac{3}{2}(n+1)^2 + \frac{3}{2}(n+1) + 1
\end{aligned}$$

□

This implies $\mathbb{E}(\tilde{n} - n)^2 = (\mathbb{E}\,\tilde{n}^2) - (\mathbb{E}\,\tilde{n})^2 = (1/2)n^2 - (1/2)n - 1 < (1/2)n^2$, and thus

$$\mathbb{P}(|\tilde{n} - n| > \varepsilon n) < \frac{1}{\varepsilon^2 n^2} \cdot \frac{n^2}{2} = \frac{1}{2\varepsilon^2}, \tag{6}$$

which is not particularly meaningful since the right hand side is only better than $1/2$ failure probability when $\varepsilon \geq 1$ (which means the estimator may very well always be 0!).

1.2 Morris+

To decrease the failure probability of Morris' basic algorithm, we instantiate s independent copies of Morris' algorithm and average their outputs. That is, we obtain independent estimators $\tilde{n}_1, \ldots, \tilde{n}_s$ from independent instantiations of Morris' algorithm, and our response to a query is

$$\tilde{n} = \frac{1}{s} \sum_{i=1}^{s} \tilde{n}_i$$

By linearity of expectation,

$$\begin{aligned}\mathbb{E}\,\tilde{n} &= \mathbb{E}\left[\frac{1}{s}\sum_{i=1}^{s}\tilde{n}_i\right] \\ &= \frac{1}{s}\sum_{i=1}^{s}\mathbb{E}\,\tilde{n}_i \\ &= \frac{1}{s}\cdot s\cdot n \\ &= n \end{aligned} \tag{7}$$

Also for *independent* random variables X, Y and scalars a, b it holds that $\mathrm{Var}[aX + bY] = a^2 \cdot \mathrm{Var}[X] + b^2 \cdot \mathrm{Var}[Y]$. Thus

$$\begin{aligned}\mathrm{Var}[\tilde{n}] &= \mathrm{Var}\left[\frac{1}{s}\sum_{i=1}^{s}\tilde{n}_i\right] \\ &= \frac{1}{s^2}\sum_{i=1}^{s}\mathrm{Var}[\tilde{n}_i] \\ &= \frac{1}{s^2}\cdot s\cdot\left(\frac{1}{2}n^2 - \frac{1}{2}n - 1\right) \\ &< \frac{1}{s}\cdot\frac{1}{2}n^2. \end{aligned} \tag{8}$$

Thus the right hand side of (6) becomes

$$\mathbb{P}(|\tilde{n} - n| > \varepsilon n) < \frac{1}{2s\varepsilon^2} < \delta$$

for $s > 1/(2\varepsilon^2\delta) = \Theta(1/(\varepsilon^2\delta))$.

Overall space complexity. When we unravel Morris+, it is running a total of $s = \Theta(1/(\varepsilon^2\delta))$ instantiations of the basic Morris algorithm. Now note that once any given Morris counter X reaches the value $\lg(sn/\delta^2)$, the probability that it is incremented at any given moment is at most $\delta^2/(ns)$. Thus the probability that it is incremented at all in the next n increments is at most δ^2/s by a union bound (recall that the union bound states that for any collection of events $\mathcal{E}_1, \ldots, \mathcal{E}_r$, $\mathbb{P}(\cup_i \mathcal{E}_i) \leq \sum_i \mathbb{P}(\mathcal{E}_i)$). Thus by another union bound, with probability $1 - \delta$ none of the s basic Morris instantiations ever stores a value larger than $\lg(sn/\delta^2)$, which takes $O(\log\log(sn/\delta^2)) = O(\log\log(sn/\delta))$ bits. Thus the total space complexity is, with probability $1 - \delta$, at most

$$O(\varepsilon^{-2}\delta^{-1}(\log\log(n/(\varepsilon\delta))))$$

bits. In particular, for constant ε, δ (say each $1/100$), the total space complexity is $O(\log\log n)$ with constant probability. This is exponentially better than the $\log n$ space achieved by storing a counter!

2 Distinct elements.

In this section we will consider the *distinct elements problem*, also known as the F_0 problem, defined as follows.

Problem. Our sequence of updates is a stream of integers $i_1, \ldots, i_m \in [n]$ where $[n]$ denotes the set $\{1, 2, \ldots, n\}$. We would like to output the number of *distinct* elements seen in the stream.

As with Morris' approximate counting algorithm, our goal will be to minimize our space consumption. There are two straightforward solutions as follows:

1. **Solution 1:** keep a bit array of length n, initialized to all zeroes. Set the ith bit to 1 whenever i is seen in the stream (n bits of memory).
2. **Solution 2:** Store the whole stream in memory explicitly ($\lceil m \log_2 n \rceil$ bits of memory).

We can thus solve the problem exactly using $\min\{n, \lceil m \log_2 n \rceil\}$ bits of memory.

Like with Morris' algorithm, we will instead settle for computing some value $\tilde{t}$ s.t. $\mathbb{P}(|t - \tilde{t}| > \varepsilon t) < \delta$, where t denotes the number of distinct elements in the stream. The first work to show that this is possible using small memory, i.e. $o(n)$ bits (assuming access to a random function that is free to store), is due to Flajolet and Martin (FM) [1].

2.1 Idealized FM algorithm

1. Pick a random function $h : [n] \to [0, 1]$
2. Maintain counter $X = \min_{i \in stream} h(i)$
3. Output $1/X - 1$

Note this algorithm really is idealized, since we cannot afford to store a truly random such function h in $o(n)$ bits (first, because there are n independent random variables $(h(i))_{i=1}^n$, and second because its outputs are real numbers).

Intuition. Note that the value X stored by the algorithm is a random variable that is the minimum of t i.i.d. $Unif(0, 1)$ random variables. The key claim is then the following, in which we suppose that the unique integers in the stream are $i_1, \ldots, i_t$. If X behaves as we expect, then since $X \approx 1/(t + 1)$, we naturally obtain the estimator above as $\tilde{t} = 1/X - 1 \approx 1/(1/(t + 1)) - 1 = t$.

Claim 4. $\mathbb{E}\, X = \frac{1}{t+1}$.

Proof.

$$\begin{aligned}
\mathbb{E}\, X &= \int_0^\infty \mathbb{P}(X > \lambda) d\lambda \\
&= \int_0^\infty \mathbb{P}(\forall i \in stream, h(i) > \lambda) d\lambda \\
&= \int_0^\infty \prod_{r=1}^{t} \mathbb{P}(h(i_r) > \lambda) d\lambda \\
&= \int_0^1 (1 - \lambda)^t d\lambda \\
&= \frac{1}{t+1}
\end{aligned}$$

□

We will also need the following claim in order to execute Chebyshev's inequality to bound the failure probability in our final algorithm.

Claim 5. $\mathbb{E}\, X^2 = \frac{2}{(t+1)(t+2)}$

Proof.

$$\begin{aligned}
\mathbb{E}\, X^2 &= \int_0^1 \mathbb{P}(X^2 > \lambda) d\lambda \\
&= \int_0^1 (\mathbb{P}(X > \sqrt{\lambda}) + \underbrace{\mathbb{P}(X < -\sqrt{\lambda})}_{0}) d\lambda && \text{(since } X \text{ is nonnegative)} \\
&= \int_0^1 (1 - \sqrt{\lambda})^t d\lambda \\
&= 2\int_0^1 u^t (1-u) du && \text{(substitution } u = 1 - \sqrt{\lambda}) \\
&= \frac{2}{(t+1)(t+2)}
\end{aligned}$$

□

This gives $\mathrm{Var}[X] = \mathbb{E}\, X^2 - (\mathbb{E}\, X)^2 = \frac{t}{(t+1)^2(t+2)}$, or the simpler $\mathrm{Var}[X] < (\mathbb{E}\, X)^2 = \frac{1}{(t+1)^2}$.

2.2 FM+

To obtain an algorithm providing a randomized approximate guarantee, just as with Morris+ we form an algorithm FM+ which averages together the outputs from s independent instantiations of the basic FM algorithm. Averaging many copies reduces variance, which let us argue that our estimate will be closer to the expectation with good probability.

1. Instantiate $s = \lceil 1/(\varepsilon^2 \delta) \rceil$ FMs independently, $\mathrm{FM}_1, \ldots, \mathrm{FM}_s$.
2. Let X_i be the output of FM_i.
3. Upon a query, output $1/Z - 1$, where $Z = \frac{1}{s}\sum_i X_i$.

We have that $\mathbb{E}\, Z = \frac{1}{t+1}$, and $\mathrm{Var}[Z] = \frac{1}{s}\frac{t}{(t+1)^2(t+2)} < \frac{1}{s(t+1)^2}$ via similar computations as (7) and (8).

Claim 6. $\mathbb{P}(|Z - \frac{1}{t+1}| > \frac{\varepsilon}{t+1}) < \delta$

Proof. We apply Chebyshev's inequality.

$$\mathbb{P}(|Z - \frac{1}{t+1}| > \frac{\varepsilon}{t+1}) < \frac{(t+1)^2}{\varepsilon^2}\frac{1}{s(t+1)^2} \leq \delta$$

□

Claim 7. $\mathbb{P}(|(\frac{1}{Z} - 1) - t| > O(\varepsilon)t) \leq \delta$

Proof. By the previous claim, with probability at least $1 - \delta$

$$\frac{1}{(1 \pm \varepsilon)\frac{1}{t+1}} - 1 = (1 \pm O(\varepsilon))(t+1) - 1 = (1 \pm O(\varepsilon))t \pm O(\varepsilon),$$

where we use $a \pm b$ to denote a value that is in the interval $[a - b, a + b]$.

□

Note that ignoring the space to store h, and with the (unrealistic) assumption that real numbers fit into a single machine word, the total space is $O(1/(\varepsilon^2\delta))$.

2.3 Removing the idealized assumptions

We will not show proofs here, but it is possible to show that both idealized assumptions can be removed. First, one can naturally not use continuous random variables but rather discretize $[0, 1]$ into integer multiples of some small value $1/B$. That is, if we have a random function $g : [n] \to \{0, 1, \ldots, B\}$, we can approximate $h(i)$ as $g(i)/B$. Next, let's think about what it means that g is a random function with the specified domain and range. Let $\mathcal{H}$ be the set of all $(B+1)^n$ functions mapping $[n]$ to $\{0, 1, \ldots, B\}$. Then we are picking a uniformly random element of $\mathcal{H}$, which requires $\log |\mathcal{H}| = n \log(B + 1)$ bits to specify. We will instead not use a uniformly random function chosen from the set of all functions, but rather from a *smaller* set of functions $\mathcal{H}'$ such that $|\mathcal{H}'| \ll |\mathcal{H}|$ (in fact we will have $\log |\mathcal{H}'| = O(\log(nB))$, so that h can be stored using only $O(\log(nB))$ bits). Thus h is not truly uniformly random but rather *pseudorandom*: just random enough to hopefully still be able to use to prove correctness of our algorithm. In fact one notion of pseudorandomness that suffices for this application is for $\mathcal{H}'$ to be what is called a *universal hash family*; we will not discuss this further in CS 170, but you will see more on universal hashing in just a couple more lectures!

References

[1] Philippe Flajolet, G. Nigel Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, volume 31, number 2, pages 182–209, 1985.

[2] Robert Morris. Counting Large Numbers of Events in Small Registers. *Commun. ACM*, volume 21, number 10, pages 840–842, 1978.