Lecture 8: Streaming Algorithms

Lecture Overview

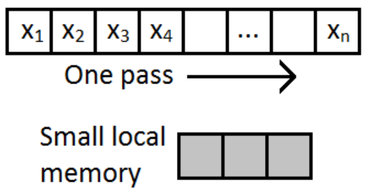
1. Streaming (what and why)
2. Reservoir Sampling
3. Distinct Elements

For most of 6.046 till now, we've been looking at algorithms that can see the entire input and make multiple passes over it if needed. In many cases, however, this may not be feasible. Some algorithms are "online" – they need to make decisions without seeing the whole input (*e.g.*, paging or caching). Today, we're going to look at yet another class of algorithms: streaming algorithms. These are used in scenarios where our memory is extremely limited compared to the size of the input, and where (typically) we can only make one pass over the input – you can't go back and look at something. Examples: routers processing IP packets, video and text streams, gaming.

Note that streaming and online algorithms are not *quite* the same.

- Online algorithms have no space restrictions, but may need to produce partial output without seeing the rest of the input.
- Streaming algorithms have limited space (think $O(\log^t n)$, for some constant t), but only produce output after seeing the entire input.

Of course, these are very general guidelines – there may be algorithms which cross over. Both classes, however, may relax correctness or optimality; we'll see an example of a "probably approximately correct" algorithm today.



We may sometimes relax the "one pass" rule to have $O(1)$ passes. These algorithms are sometimes **exact** (*e.g.*, average, majority), but more often, they are **probably approximately correct** (*e.g.*, # of distinct elements).

Simple Statistics

Average, Max, etc.

We are asked to compute $\frac{\sum x_i}{n}$, or $max(x_i)$.
Solution: Keep a running partial answer (notice how this is basically online).
$O(\log n)$ space, 1 pass.

Majority Element

We are told that there exists an element which appears $> n/2$ times in the input. How can we find it?

Consider the following algorithm:

1. Maintain a "current champion" element x and a counter c. (Initially, x is set to "token" element that is different to any other element and c is equal to 0.)
2. In each step $i = 1, \ldots, n$:
   - If x_i is equal to x, increment the counter c by one.
   - Otherwise (i.e., $x_i \neq x$)
     - If the counter c is equal to 0, set x to be x_i and c equal to 1.
     - Otherwise (i.e., $c > 0$), decrement the counter c by one.
3. Output x.

Note that if there exists a *strict* majority element then it is guaranteed to "prevail" and emerge as the element x at the end. (Think of each instance of a given element to be a member of the corresponding "team". Whenever $x \neq x_i$ we view it as the corresponding team members "taking out" each other. If there is a strict majority element, it will be impossible for all its team members to be taken out, even if all the other teams "collude" against them.)

Importantly though, if there does not exist a majority element there is *no* guarantee on what x will be. It might be even an element that is very far from being the most frequent one. (Think a sequence of $n/2 - 1$ 0s, followed by $n/2 - 1$ 1s and a final two elements being 3—the output element will be 3!)

Reservoir Sampling

Sample a uniformly random element from a stream of (a-priori) unknown size. That is, for each i, we want to output x_i with probability $1/n$. Unfortunately, we have a problem: until the stream ends, <u>we don't know what n is</u>.

At each point, we have to assume that the current element is the last one (the next input we get might be NULL, ending the sequence). So, our probability of keeping elements at each stage looks like this:

1. Keep x_1 w.p. 1. We'll write this as $\{1\}$.
2. Now we have 2 elements. We need to keep x_1 with a reduced probability of 1/2 (from 1), and x_2 w.p. 1/2. We'll write this as $\{\frac{1}{2}, \frac{1}{2}\}$.
3. $\{\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\}$, etc.

So let's think about what we're doing here: at any time i, we have a random sample from $x_1, x_2, \ldots, x_i$, with each element having been kept w.p. $\frac{1}{i}$. So what should we do when we see x_{i+1} in order to get $\frac{1}{i+1}$ for each element?
Answer: Keep each element we already have w.p. $\frac{i}{i+1}$. Since Pr[x_j kept] was $\frac{1}{i}$, after this step, the probability will become $\frac{1}{i} \times \frac{i}{i+1} = \frac{1}{i+1}$, as desired. The new element, x_{i+1}, of course, should be kept w.p. $\frac{1}{i+1}$, so that now, all elements are kept w.p. $\frac{1}{i+1}$.

Generalization to k samples

Now, we have to sample k random elements:

- Keep "reservoir" of k elements at all times
- Include first k
- At each x_{i+1}, keep it w.p. $\frac{k}{i+1}$ and remove a random element from the k in the reservoir.

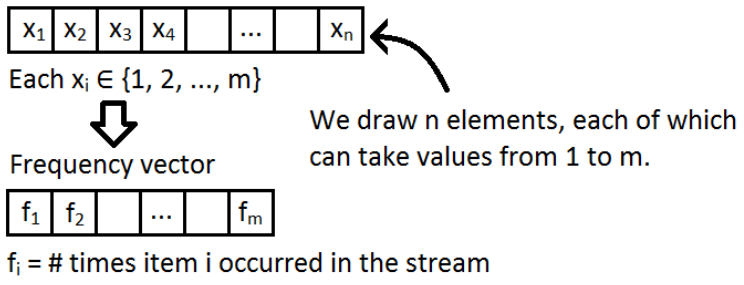
Other generalizations include weighted sampling, where each x_i comes with a weight x_i. That is, we want to sample x_i w.p. $\frac{w_i}{\sum_{j=1}^n w_j}$. Think about how you would need to modify your algorithm in this scenario.

Frequency Moments

We want to find the frequency moments, which are:

$$F_p = \sum_{i=1}^{m} f_i^p$$

.


x1 x2 x3 x4 ... xn
Each xi ∈ {1, 2, ..., m}
We draw n elements, each of which can take values from 1 to m.
Frequency vector
f1 f2 ... fm
fi = # times item i occurred in the stream

- $F_0 = \sum f_i^0 = \#$ of distinct elements (assuming $0^0 = 0$)
- $F_1 = \sum f_i = \#$ of elements $= n$
- $F_2 = \sum f_i^2 =$ size of database join

Probably Approximately Correct

We want to compute an (ϵ, δ)-approximation of F_0. That is, we want an estimate $\hat{F_0}$, such that w.p. $\geq 1 - \delta$, we have that $(1 - \epsilon)F_0 \leq \hat{F_0} \leq (1 + \epsilon)F_0$.

So, for example, for $\epsilon = 0.1$ and $\delta = 0.01$, we could say that our estimate would be between $0.9F_0$ and $1.1F_0$ with 99% probability.

Both ϵ and δ are necessary for a streaming algorithm that finds frequency moments – there are impossibility proofs ruling out both deterministic-approximate and randomized-exact methods. [AMS = Alon, Matias, and Szegedy]

Estimating F_0

Let us define d^* to be the number of distinct elements in the stream. Before we attempt to solve the intended problem, let us solve a simpler version of it first. And then we will turn our attention to the estimation of F_0.

Simpler Problem: Verifying a Guess on d^*

Assume someone gave you a "guess" d on the value of d^*, and your job is to verify if it is within the right ballpark. Specifically, you want to construct an algorithm that:

1. if $d^* \geq 2d$, it outputs Yes;
2. if $d^* < d$, it outputs No;
3. otherwise (i.e., if $d \leq d^* < 2d$), either Yes or No answer it correct.

How to solve this problem? Turns out that one can resort to a (clever) use of hash functions! Specifically, let us set $B := 4d$ and consider a (truly random) hash function $h \in_R \mathcal{H}$, where $\mathcal{H}$ is (for now) the set of all functions that hash elements of the stream into $[B]$, i.e., the set $\{0, 1, \ldots, B-1\}$.

Consider the following algorithm:

1. Take a pass through the stream to check if there exists *at least* one element x_i in it such that $h(x_i) = 0$. (As one will see below, the choice of 0 is arbitrary—any fixed element of $[B]$ would do.)

2. If there exists at least one such element, output Yes; otherwise, output No.

What is the probability that the above algorithm will output the correct answer?

To this end, we need to consider two cases: (a) $d^* < d$, and (b) $d^* \geq 2d$. (We do not care about the third remaining case as either way our algorithm will be correct then.)

Ad Case (a). ($d^* < d$) The correct answer in this case is No, so we want to upper bound the probability that our algorithm says Yes. Observe that, by the union bound, this probability will be at most

$$\begin{aligned} \Pr\left[\text{Algorithm outputs } Yes\right] &= \\ \Pr\left[\cup_{j=1}^{d^*} A_j\right] &\leq \\ \sum_{j=1}^{d^*} \Pr\left[A_j\right] &= \\ \frac{d^*}{B} = \frac{d^*}{4d} &< \frac{1}{4}, \end{aligned}$$

where A_j denotes the event that the j-th distinct element in the stream hashes to 0. So the probability that the algorithm outputs incorrect answer here is at most $\frac{1}{4}$.

Ad Case (b). ($d^* \geq 2d$) In this case, the correct answer is Yes and we want to lower bound the probability of our algorithm outputting this answer.

First of all, notice that that probability is *monotone* in d^*, i.e., the larger d^* is (with respect to d) the more likely the algorithm is to output Yes (as it is more likely that at least one element in the stream hashes to 0). So, without loss of generality, we can assume that $d^* = 2d$.

Now, observe in this case

$$\begin{aligned}
\Pr\left[\text{Algorithm outputs } Yes\right] &= \\
\Pr\left[\cup_{j=1}^{d^*} A_j\right] &\geq \\
\sum_{j=1}^{d^*} \Pr\left[A_j\right] - \sum_{j=1,j'=1,j<j'}^{d^*} \Pr\left[A_j \cap A_{j'}\right] &= \\
\frac{d^*}{B} - \binom{d^*}{2}\frac{1}{B^2} = \frac{d^*}{4d}\left(1 - \frac{d^*-1}{8d}\right) \geq \frac{1}{2}\left(1-\frac{1}{4}\right) &= \frac{3}{8},
\end{aligned}$$

where we we are using the so-called Bonferroni inequalities. Whereas the union-bound truncates the inclusion-exclusion principle at the first level to provide an upper bound to the probability of the union of events, the Bonferroni inequaility provides a *lower bound* by truncating the inclusion-exclusion principle at the second level. Specifically, we have that

$$\Pr\left[\cup_i A_i\right] \geq \sum_i \Pr\left[A_i\right] - \sum_{i<j} \Pr\left[A_i \cup A_j\right].$$

We can thus conclude that our algorithm is more likely ($\frac{3}{8}$ vs. $\frac{1}{4}$) to output Yes when it should than when it should not. But this seems still far from the $1-\delta$ probability of outputting the correct answer that we are looking for. How to get there?

The (standard) idea to apply now is (independent) replication. Specifically, it might be difficult to distinguish between the cases (a) and (b) if we run the above algorithm once. However, the situation changes dramatically if we consider running this algorithm (with independent choices of the hash function) *multiple*, say, k times.

Indeed, in this case, the expected number of Yes answers in case (a) would be at most $\frac{1}{4}k$ while in the case (b) this number would be at least $\frac{3}{8}k$. More importantly, we know that due to their independence, the actual number of Yes answers will concentrate around their expected values.

Specifically, one can show using Chernoff bounds that, for any $\delta > 0$, if one sets $k = C \ln \frac{1}{\delta}$ for a suitably chosen constant C, and decide to output Yes iff the number of observed Yes answers is at least $\frac{1}{2}\left(\frac{3}{8}+\frac{1}{4}\right)k = \frac{5}{16}k$ and output No otherwise; then the resulting algorithm will be correct with probability at least $1-\delta$, as desired.

Exercise: Work out the details. Specifically, pick a constant C such that the above algorithm errs with probability at most δ. *Hint:* Let Y denote the number of Yes answers. Use Chernoff to bound $\Pr(Y \geq \frac{5}{16}k)$ in case (a) and $\Pr(Y \leq \frac{5}{16}k)$ in case (b).

Now, we seem to get the guarantee we wanted to get, but is it really an (efficient) streaming algorithm yet? Unfortunately, not. There are two issues we need to address.

The first one relates to the fact that in the streaming model, unlike the "standard" model of computation, we can't really re-run an algorithm—after all, we can only take a single pass through the stream. This is not really a problem though here, as the runs of the algorithm we need are independent of each other. So we can simply run them all *in parallel*, instead of consecutively. This was we only need to take a single pass through the stream and the sole price we pay is that our space complexity increases by a factor of k, since we need to maintain the computation state for all k executions of the algorithm simultaneously. This is a price that we can easily pay (since k is pretty small to begin with).

The second issue relates to the fact that in our algorithm (and analysis) above we assumed that the hash function h is truly random. That is, that the set $\mathcal{H}$ we are sampling it from contains all the functions that map the elements into $[B]$. Clearly, storing such function would be infeasible (as we would need to remember the mapped value for each of the m potential elements to be mapped). How can we deal with it?

Again, as we have seen before, it turns out that even though thinking of the hash function h at first as truly random is useful, it actually does not need to be such. Specifically, all we need is that h corresponds to a *pairwise independent* family of hash function.

Formally, a family $\mathcal{H} = \{h : X \to Y\}$ of functions is *pairwise independent* iff

$$Pr_{h \in_R \mathcal{H}}[h(x_1) = y_1 \wedge h(x_2) = y_2] = \frac{1}{|Y|^2},$$

for any $x_1 \neq x_2 \in X$ and $y_1, y_2 \in Y$. (Notice that, as usual, the randomness is over the choice of hash function.)

Note that if h is such a function then we have that in our analysis above

$$\Pr[A_j] = \frac{1}{B} \quad \text{and} \quad \Pr[A_j \cap A_{j'}] = \frac{1}{B^2},$$

which is all we needed to have this analysis go through.

Also, once we need h to be pairwise independent, representing such a hash family function can be much more space efficient. For example, if B is a prime then the hash family $\mathcal{H} := \{h_{a,b} | a, b \in [B]\}$, where

$$h_{a,b}(x) = a \cdot x + b \pmod{B},$$

is pairwise independent and can be stored using $O(\log B) = O(\log n)$ bits.

So, this enables us to turn the above procedure into a valid streaming algorithm and its space complexity will be

$$k \cdot O(\log n) = O(\log n \log \frac{1}{\delta}),$$

where, again, $\delta > 0$ is our desired error bound.

Back to Distinct Elements Problem

Ok, so we have solved the above simpler problem. But what does it tell us about the actual problem we wanted to solve, i.e., the distinct element problem? In particular, what should we do if there is no "guess" d to work with?

It turns that the above algorithm is already pretty close to providing us with a factor 2 approximation to distinct element problem. All we have to do is to "fabricate" d. Specifically, instead of trying to "guess" a specific d we will examine a whole spectrum of values of d in parallel. That is, we will look at the values of d being 2^t for $t = 0, \ldots, logm$ and run the above procedure for each such value of d. By doing this and analyzing the (aggregate) Yes/No answers we will get, we can pinpoint the value of d that is within a factor of 2 of d^*, as desired.[1]

Now, how do we get from obtaining such a factor of 2 approximation (which corresponds to an (ε, δ)-approximation to F_0 with $\varepsilon = 1$) into an (ε, δ)-approximation for an arbitrarily small $\varepsilon > 0$?

It turns out that all we have to do is to change the gradation by a factor of 2 we used in the procedure above to a gradation by a factor of $(1 + \varepsilon)$. More precisely, to keep the core algorithm above as is but perform the analysis for the two cases being (a) $d^* < d$, and (b) $d^* \geq (1 + \varepsilon)d$ and then make k be appropriately large (as the gap between the probabilities of outputting Yes in (a) vs. (b) case is appropriately smaller), and adjust the aggregation of the final tally appropriately.

The resulting overall space complexity bound then becomes

$$O\left(\varepsilon^{-3} \cdot \log^2 n \cdot \log \frac{\log n}{\delta}\right)$$

It turns out that the optimal space complexity bound for this problem, due to Kane, Nelson and Woodruff (all MIT alumni!), is equal to

$$O(\varepsilon^{-2} + \log n).$$

[1]Important technical detail is to set the value of δ in each of these procedures to be equal to $\frac{\delta'}{\log m+1}$ so as to be able to use the union bound over all the $\log m + 1$ executions to get the overall desired δ' bound on the probability of outputting the wrong answer.