Lecture 14: Streaming Algorithms

Course Overview

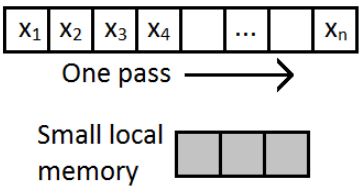
1. Streaming (what and why)
2. Reservoir Sampling
3. Distinct Elements

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 ($O(n)$ or even $O(\log n)$), 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? (Deceptively simple algorithm – try doing an example at home!)

```
counter = 0
majorityElement = 0
for each item in list:
    if item == majorityElement
        counter++
    else
        if counter == 0
            majorityElement = item
            counter = 1
        else
            counter--
return majorityElement
```

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, we don't know what n is.

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 $\Pr[x_j \text{ kept}]$ 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. $\frac{1}{i+1}$.

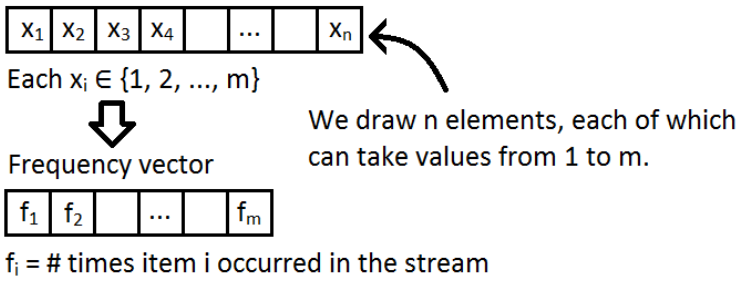
Generalization to k samples

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 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:

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

.

- $F_0 = \sum f_i^0 = \#$ 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]

Pairwise-Independent Hash Function Families

For our discussion, we'll draw upon AMS and another paper by Philippe Flajolet and G. Nigel Martin. Our key tool in this matter will be pariwise-independent hash functions. These are hash functions drawn from a family $\mathcal{H} = \{h : X \to Y\}$ such that the random variables $h(x_1)$ and $h(x_2)$ are independent for all $x_1 \neq x_2$. (Notice that the randomness is over the choice of hash function.)

Another way to (slightly more formally) define such hash function families is to say that for every $x_1 \neq x_2 \in X$ and $y_1, y_2 \in Y$, we can say:

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

There are many such families, *e.g.*, $h_{A,B}(x) = Ax + b \mod 2$, where A is a $k \times k$ matrix, and b is a size-k vector.

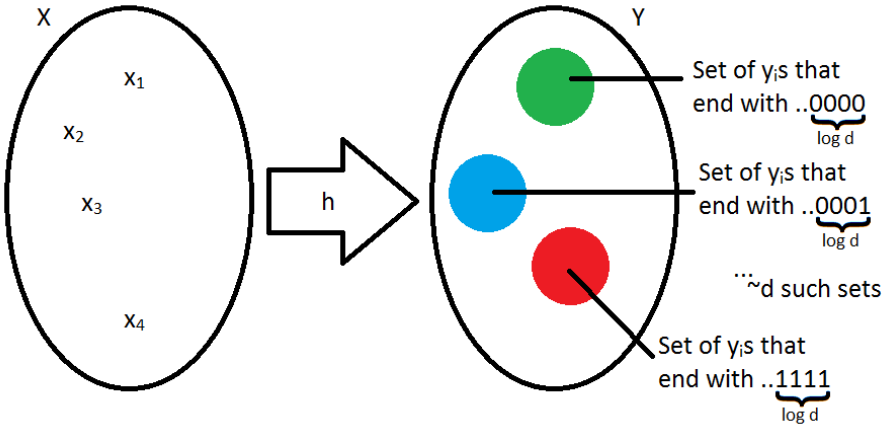
Estimating F_0

Imagine a list of d distinct elements, and each time we take out an element, we toss a coin multiple times until we get tails. What do you think will be the longest streak of heads we shall have if we do this? As you will see in recitation (and I'll give you some quick intuition after this) this will be $\log n$. Keep this idea in mind as you look at the following algorithm.

First, we let $\mathcal{H}$ be a pairwise-independent hash function family, from which we draw our hash function h. We also define a function $zeroes(x)$ which simply counts the number of trailing zeroes in the binary representation of x.

```
z = 0
For each item j in list
    if zeroes(h(j)) > z
        z = zeroes(h(j))
return 2^z
```

So we're just finding the longest trailing sequence of zeroes in our sequence of hashes. Recall that we have n elements in total, and d distinct elements. If we can show that z is around $\log d$, we're done.



Intuition: Since there are d distinct elemetns in the stream, there is a "good chance" that one will hash to the first (green) bin, which will give us *at least* $2^{\log d} = d$. What about overshooting? If we look at bins for elements ending with, say "$\ldots 0000000$", where the number of digits is $\gg \log d = \log cd$, then we find the the number of such bins is much greater than d, so we again have a "good chance" that none of the d elements will hash to the trailing-zeroes bin. So, output is probably $< 2^{\log cd} = cd$.

Proof

There's a bunch of algebra we'll skip here. You'll cover the theorems involved in greater detail in recitation.

Note 1: $\forall j \in X, r \in Y$

$$Pr[zeroes(h(j)) \geq r] = \frac{1}{2^r}$$

Note 2: $\forall j, k \in X, j \neq k, r \in Y$

$$Pr[zeroes(h(j)) \geq r \wedge zeroes(h(k)) \geq r] = \frac{1}{2^r} \times \frac{1}{2^r} = \frac{1}{2^{2r}}$$

Once again, we'll use an indicator random variable. We're going to fix an r and see what we can say about that situation. $W_x = 1$ if $zeroes(h(x)) \geq r$, and 0 otherwise.

Let $Z = Z_r = \sum_{x \in \text{stream}} W_x$. This Z will exceed 0 if and only if there's ≥ 1 hash with $\geq r$ trailing zeroes. Also note that $Z_1 \geq Z_2 \geq \ldots$, and we output 2^r on the first $Z_{r+1} = 0$.

Note that we can assume that we're only looking at a list of d distinct variables becasue we're not actually trying to find or count them; we're just proving some things about them. Now, let's see some characteristics of these variables:

$$\mathbb{E}[Z] = \sum \mathbb{E}[W_x] = d \times \frac{1}{2^r} = \frac{d}{2^r}$$

$$Var[Z] = \sum Var[W_x] = d \times \frac{1}{2^r}(1 - \frac{1}{2^r}) < \frac{d}{2^r}$$

Now, let's see what happens if r is too large, *i.e.*, what is the probability of getting a positive Z if we set r too high. Let our algorithm overshoot r and output $2^r > cd$ (c is some constant). Then, using Markov, we can say:

$$Pr[Z_r > 0] \leq E[Z_r] = \frac{d}{2^r} < \frac{d}{cd} = \frac{1}{c}$$

What about a low r? Let $2^r < \frac{1}{c} \cdot d$ or $c \cdot 2^r < d$. Note that Z_r cannot be negative, so we'll be using $Z_r = 0$ instead of $Z_r \leq 0$ in some places. Then, using Chebyshev:

$$Pr[Z_r = 0] \leq \frac{Var[Z]}{\mathbb{E}[Z]^2} < \frac{d/2^r}{(d/2^r)^2} = \frac{2^r}{d} < \frac{2^r}{c \cdot 2^r} = \frac{1}{c}$$

Combining the two inequalities derived above, we get:

$$Pr[(\hat{F_0} > c \cdot d) \vee (\hat{F_0} < \frac{1}{c} \cdot d)] < \frac{1}{c} + \frac{1}{c}$$

$$\text{negating, } Pr[\frac{d}{c} \leq \hat{F_0} \leq cd] \geq 1 - \frac{2}{c}$$

This uses $O(\log n)$ space to store the hash function. It's pretty good – $c = 5$ would bound the probabilty to be ≥ 0.6. Of course, we can do much better - there's an algorithm by [Bar-Yossef, Jayram, Kumar, Sivakumar, Trevisan] which does (ϵ, δ)-approximation with space $O(\frac{1}{\epsilon^2} \log(\frac{1}{\delta}) \log n)$.

But we can easily do a little bit better than before by having multiple ($l = O(\log \frac{1}{\delta})$) hash functions, running the algorithm as many times, and then taking the median of the l estimates. This would give us (using Chernoff) $Pr[\frac{d}{c} \leq \hat{F_0} \leq cd] \geq 1 - Pr[\hat{F_0} \notin (\frac{d}{c}, cd)] \geq 1 - Pr[\text{majority of outputs } \notin (\frac{d}{c}, cd)] \geq 1 - (c')^{O(\log \frac{1}{\delta})} \geq 1 - \delta$.

Recall: Markov $Pr[Z \geq c\mathbb{E}[Z]] \leq \frac{1}{c} \implies Pr[Z \geq 1] \leq \mathbb{E}[Z]$.
Chebyshev $Pr[|Z - \mathbb{E}[Z]| \geq k \cdot stdDev[Z]] \leq \frac{1}{k^2} \implies Pr[|Z - \mathbb{E}[Z]| \geq \mathbb{E}[Z]] \leq \frac{Var[Z]}{\mathbb{E}[Z]^2}$.