CSCI 1951-W Sublinear Algorithms for Big Data **Fall 2020**

Lecture 16: Introduction to Streaming Algorithms

Lecturer: Jasper Lee *Scribe: Thomas Ottaway*

1 Motivation

So far in the course we have been concerning ourselves with algorithms which have sublinear time or query complexity. Now we are going to look at algorithms which are sublinear in space. One common situation where we want to be able to have sublinear space complexity is when data is coming from a (very long) stream, where we cannot afford to keep all the data from the stream. Streams are of great interest both in theory and practice, since they can be used to describe a wide range of situations such as information coming from a sensor or the internet.

2 Problem Setting and Typical Runtimes

A typical model for a streaming algorithm is as follows: Consider some universe $[n]$ and some length-m stream input $\sigma = (\sigma_1, \ldots, \sigma_m) \in [n]^m$. Our goal is to compute some function of this stream.

The order of these elements is not known to the algorithm and is often assumed to be chosen in an adversarial manner. If we were allowed to assume some fixed ordering, or even a random ordering, then many problems would become much simpler.

When developing streaming algorithms, we are allowed to look at the elements in order, and after looking at each element, we are allowed to update some internal state. Algorithms are typically judged based on the maximum size of this internal state with less memory usage being better.

We call this the space complexity, s, of an algorithm, measured in bits. We want all of our algorithms to be sublinear (i.e. $s = o(\min(n, m))$). However, the holy grail is for an algorithm to run in $s = O(\log n + \log m)$ space. This is just enough memory to store a constant number of stream elements (each one takes $\log_2 n$ bits) and a constant number of counters (where each of these counters could potentially go up to m which would take $\log_2 m$ bits). Sometimes this is not possible, so in general we can settle for $s = \text{polylog}(n) + \text{polylog}(m)$.

Side Note: It is possible to keep a approximately correct counter using only $\log \log m$ space. One example is the *Morris Counter*. The basic idea is to store only the exponent of the counter and then instead of incrementing the counter every time an event occurs you increment it will probability equal to $\frac{1}{2}^c$ where c is the current value of the counter. You can read more about this on the Wikipedia page, or on various lecture note online.

We typically allow algorithms to take just a single pass through the stream. This requirement is natural in situations like sensor data where once you've looks at the measurement and chosen not to store it, it is gone forever. However, there are interesting multi-pass streaming algorithms in the literature, although out of the scope of this course.

3 Majority Element

Problem statement: Suppose we know that a given stream contains some element $k \in [n]$ such that k appears in the stream $> \frac{m}{2}$ times. We want to determine the identity of k.

Algorithm 16.1

1. Initialize `count` to 0
2. Initialize `Maj` to `Null`
3. Repeat m times:
   (a) Read new $\sigma_i \in [n]$
   (b) If `count` $= 0$, set the `Maj` equal to σ_i and increment `count`
   (c) If `count` > 0, if `Maj` $= \sigma_i$ then increment `count`, otherwise decrement `count`
4. Return `Maj`

Note that the algorithm does not need to know n or m (as long as it has at least $\log_2(n) + \log_2(m)$ memory allocated).

Theorem 16.2 *Algorithm 16.1, on input of length m stream over $[n]$ with a majority element, outputs this majority element. Furthermore, Algorithm 16.1 runs in $\Theta(\log n + \log m)$ space, $\Theta(m)$ time.*

Proof. Correctness: Let k be the majority element. Consider `count'` which is defined as being equal to `count` when `Maj` is k and equal to `-count` otherwise. Note that `count'` always increments when $\sigma_i = k$. Therefore, by the definition of majority element, `count'` will be positive at the end of the stream, meaning that `Maj` must be equal to k.

The complexity claims are proven by construction (`Maj` take $\log_2 n$ bits to store, and `count` take $\log_2 m$ bits to store). □

4 Variant of streaming model

Often, the function of the stream does not depend on the stream order. Instead, it depends only on the *frequency vector*.

$$\mathbf{f} = (f_1, \ldots, f_n) \in [m]^n$$

Here, f_i is the number of occurrences of element $i \in [n]$.

For these problems, we can use a slightly different streaming model. In this new model, every stream token contains a value i and a frequency c. Seeing the token (i, c) means to "add c copies of element i," or to update $f_i \leftarrow f_i + c$. Note that this means that it no longer makes sense to talk about m being the total number of tokens. Instead, we analogously assume that m is an upper bound to $||f||_1 = \sum_i |f_i|$.

There are a number of different models which have different requirements on c.

- *Cash Register Model*: requires $c > 0$.
- *Turnstile Model*: no requirements.
- *Strict Turnstile Model*: no restrictions on c, but we are guaranteed that $\mathbf{f} > 0$.

Note that Algorithm 16.1 can be easily adapted to work in the *Cash Register Model*.

5 Reservoir Sampling

Problem: Sample an element from a stream uniformly (in the original stream model). Here we treat the stream as a multiset, so if i occurs twice it shows up twice as often. This would be trivial if m were known before runtime, however, so we want to do this without knowing m in advance.

This problem is also known as ℓ_1-sampling since we are sampling i with probability $\frac{|f_i|}{||f||_1}$.

Algorithm 16.3

1. Initialize `currentSample` to be null
2. Repeat m times:
   (a) Read new σ_i
   (b) With probability $\frac{1}{i}$ assign `currentSample` to be σ_i otherwise do nothing.

Theorem 16.4 *Algorithm 16.3 samples between the elements of the stream uniformly at random with space complexity* $= O(\log n + \log m)$ *space.*

Proof. We prove this by induction, claiming that after the k^{th} iteration, the distribution of `currentSample` is uniform over the elements $[k]$ is uniform. The base case holds since in the first iteration we select σ_1 with probability 1. Now we show that if, at step $k-1$, the distribution was uniform over $[k-1]$, then the distribution at step k will be uniform over $[k]$.

Let $\texttt{currentSample}_i$ be the value of `currentSample` after seeing the σ_i.

- $\mathbb{P}(\texttt{currentSample}_k = \sigma_k)$ is equal to σ_k is $\frac{1}{k}$ by the construction of Algorithm 16.3.
- If $\texttt{currentSample}_k$ is not equal to σ_k then it must be equal to $\texttt{currentSample}_{k-1}$ by the construction of our algorithm. Therefore, for any $i \leq k-1$ we know that:

$$\begin{aligned}\mathbb{P}(\texttt{currentSample}_k = \sigma_i) = &\mathbb{P}(\texttt{currentSample}_{k-1} = \sigma_i) \\ &\cdot \mathbb{P}(\texttt{currentSample}_k \text{ was not replaced by } \sigma_k)\end{aligned}$$

Then by our inductive hypothesis and the construction of the algorithm we get that

$$\mathbb{P}(\texttt{currentSample}_k = \sigma_i) = \frac{1}{k-1}\left(1 - \frac{1}{k}\right) = \frac{1}{k}$$

Therefore, at every step k, `currentSample` is uniformly distributed over $[k]$.

The space complexity bounds are clear from construction. We store one element at a time which takes $\log_2 n$ bits. We also need to store i, which ranges from 1 to m, which means it will take $O(\log m)$ bits to store. □