ECS 189A Sublinear Algorithms **Fall 2024**

Lecture 14: Introduction to Streaming Algorithms

Lecturer: Jasper Lee *Scribe: Aadithyaa Sridharbaskari*

Introducing the Streaming Model

In this course, we always want to ask which computational resource we wish to optimize. Previously, we examined algorithms that were "sublinear in time." For example, algorithms that have a sublinear sample complexity or query complexity. Going forward, we will focus on algorithms that are sublinear in space complexity. We'll begin by defining the streaming model. One should note that there are many variants of this model.

Streaming model setup

Let $[n]$ be the universe or domain of our stream. The input to the algorithm in this model is an ordered stream of length m: $(\sigma_1, \sigma_2, \ldots, \sigma_m) \in [n]^m$. The algorithm receives this stream in an adversarial order. We want an algorithm that computes something about the stream such that the algorithm updates its internal state after reading each σ_i.

Goals of the Streaming Model

The main goal of the streaming model is to minimize the memory needed at any point during the execution of an algorithm. To accomplish this, we need to compute some "sketch" of the elements we've seen so far rather than storing all of them.

What benchmark do we compare our memory usage against? Let s denote the maximum number of bits required by our algorithm. The two quantities of interest are the stream length m and the support size n. We consider memory usage to be sub-linear if

$$s = o(\min(n, m)).$$

In other words, we aim to find algorithms whose memory complexity grows slower than the minimum of n and m.

In particular, the holy grail of streaming algorithm memory complexity is

$$s = O(\log n + \log m).$$

This means the algorithm has enough memory to store the length of the stream and the identifier of a constant number of elements from the support at any time. Realistically, we'll be happy with $O(\text{polylog}(n) + \text{polylog}(m))$ bits. In this class, we'll mainly be concerned with *single-pass* streaming algorithms. For a motivating example of this setup, consider a router that continuously handles internet traffic with a fixed-size memory buffer. Once it evicts a piece of data, it's gone forever. On the other hand, we can also consider the *multi-pass* streaming model. For example, think of a magnetic tape that we can feed back into the read/write head in a loop manner.

Majority Element Problem

Suppose that the stream guarantees the existence of a $k \in [n]$ that appears strictly more than $m/2$ times. The task is to find this k using minimal memory complexity. Clearly, if such a k exists, it must be unique. We will analyze a simple deterministic algorithm that solves this problem.

Algorithm 14.1 Majority element algorithm

```
1: Initialize count ← 0
2: Initialize majority ← NULL
3: while not end of stream do
4:     Read the next stream element σ_i
5:     if count = 0 then
6:         majority ← σ_i
7:         count ← count +1
8:     else if majority = σ_i then
9:         count ← count +1
10:    else
11:        count ← count −1
12:    end if
13: end while
14: return majority
```

Note that this algorithm is particularly good in that it doesn't even need to know the length of the stream as long as it has $> \log 2m$ bits of memory. Why is this algorithm correct? Intuitively, each time we increment an element, we raise our confidence that it is the majority element. In particular, we need to see other elements just as frequently in order to convince ourselves otherwise. Since there are fewer than $m/2$ non-majority elements, there are simply not enough to offset the contribution to count from the majority element.

Theorem 14.2 (Correctness and efficiency of Algorithm 14.1)**.** *Algorithm 14.1 will output the correct majority length element on a stream of any length provided that one exists. Additionally, it uses no more than* $O(\log n + \log m)$ *bits of memory.*

Proof. Let k be the majority element. We claim that Algorithm 14.1 outputs k at the end. For convenience of analysis, we will define `count'` to be equal to `count` whenever `majority` $= k$ and $-$`count` otherwise. Intuitively, count$'$ measures how close the algorithm is to thinking that the answer is k. Clearly, if `count'` is positive at the end of Algorithm 14.1, then it will output the correct majority element. Notice that `count`$'$ increments whenever `majority` $= k$ and decrements otherwise. Since `count'` increments more than it decrements and begins at zero, it will be positive at the end of Algorithm 14.1. □

Variants of the streaming model

Often, the task will not depend on the stream ordering, just on the *frequency vector*. Denote this by $\mathbf{f} = (f_1, f_2, \ldots, f_n) \in [m]^n$ where f_i is the number of occurrences of element i in the stream. This motivates a slightly different setup, where each stream token gives you a pair

(i, c) of support element $i \in [n]$ and element count $c \in [m]$. Concretely, each stream token will increment f_i by c. Of course we will have the promise that $\|\mathbf{f}\|_1 \leq m$ to be consistent with the stream length. Depending on how you constrain the element count updates c, we can get many different interesting models. In particular

1. $c > 0$ corresponds to the "cash register model"
2. No restrictions on c corresponds to the "turnstile model"
3. No restrictions on c except that $f_i \geq 0$ at all times corresponds to the "strict turnstile model"

Reservoir sampling

Our task is to sample an element from the stream uniformly at random. In particular, we want to explicitly treat the stream as a multiset, meaning that multiple occurrences of an element will give it commensurately higher probability. One can motivate an algorithm for this problem with an inductive argument. Suppose we have already seen j elements of the stream. We can ask how does our sampling procedure change after seeing the next element from the stream σ_{i+1}? If we assume that everything our algorithm did before seeing σ_{j+1} was correct, then the sample our algorithm would output at stream index j will be a uniform sample from the first j stream elements. Having seen σ_{j+1}, we want to output that element with probability $\frac{1}{j+1}$. Thus, we can simply replace our current output with σ_{j+1} with probability $\frac{1}{j+1}$. Our algorithm will simply do this at every time step j.

Algorithm 14.3 Reservoir sampling algorithm

```
Initialize sample ← NULL
for i ∈ {1, 2, ..., m} do
    Read the next stream element σ_i
    Replace sample with σ_i with probability 1/i
end for
return sample
```

Theorem 14.4 (Correctness and efficiency of Algorithm 14.3)**.** *Algorithm 14.3 uniformly samples an element from the stream with multiplicity, and uses no more than* $O(\log n + \log m)$ *bits of memory.*

Proof. The only memory we store is an index that goes up to the stream length and a sample that can take values from the support, which requires no more than $O(\log n + \log m)$ bits of memory. As for correctness, we will proceed via induction as suggested earlier. Let $E_{i,j}$ denote the event that our current sample is equal to stream element σ_i given that we've only seen the first j elements of the stream for $i \leq j$. Suppose that $\mathbb{P}(E_{i,j}) = \frac{1}{j}$, and that the next stream element we see is σ_{j+1}. If we independently replace the current sample with σ_{j+1} with probability $\frac{1}{j+1}$, then the probability that our sample is equal to σ_i after seeing σ_{j+1} is simply $\mathbb{P}(E_{i,j})$ times the probability that we don't replace the sample with σ_{j+1}, which is

$$\mathbb{P}(E_{i,j+1}) = \mathbb{P}(E_{i,j}) \left(\frac{j}{j+1}\right) = \left(\frac{1}{j}\right) \left(\frac{j}{j+1}\right) = \frac{1}{j+1} \tag{1}$$

□

Thus, $\mathbb{P}(E_{i,j+1}) = \frac{1}{j+1}$. We complete the induction by noting that the base case is trivial since the probability of sampling a single element from a stream uniformly at random is simply 1.