CS 170 Final Review: Streaming

1 Streaming for Voting

Consider the following scenario. Votes are being cast for a major election, but due to a lack of resources, only one computer is available to count the votes. Furthermore, this computer only has enough space to store one vote at a time, plus a single extra integer. Each vote is a single integer 0 or 1, denoting a vote for Candidate A and Candidate B respectively.

(a) Come up with an algorithm to determine whether candidate A or B won, or if there was a tie.

(b) Consider now an election with 3 candidates. Say there is a winner only if a candidate receives more than 50 percent of the vote, otherwise there is no winner. If we're given another integer's worth of storage, come up with an algorithm to determine the winner if there is one. For simplicity, your algorithm can output any of the candidates in the case that there is no winner (not necessarily the one with the most votes). Votes are now numbered 0, 1, 2.

Solution:

(a) Initialize one of the integers i to 0. Use the other to store the incoming votes. For every vote for Candidate A, decrement i by one. For every vote to candidate B, increment i by 1.

If at the end i is negative, Candidate A won. If at the end i is positive, Candidate B won. If i is 0, there was a tie.

(b) Let m be a variable which will hold either 0, 1, or 2. Let i be a counter similar to in the previous part. For each element in the stream, do the following. If i is 0, set m equal to the value of the current vote, and set i to 1. Else if $i > 0$, and m is equal to the current vote in the stream, increment i. Else decrement i.

At the end, if there is a majority vote, m will contain it. However, if there is no majority vote, m will still contain some element of the stream.

We can see that this is the case with the following short inductive proof:

Base case: Our algorithm will definitely detect the majority element of a 1 element stream.

Inductive Hypothesis: Assume our algorithm works for a stream of n or fewer elements.

We have two cases. Either the first candidate's count drops to zero at some point during the streaming, or it doesn't.If the first candidate's count never drops to zero, then the first candidate must be the majority candidate.

If the first candidate's count does drop to zero (let's say after the xth vote), then we can say the following. Suppose we have not yet encountered the true majority element. Then it must be the majority of the rest of the stream. By the inductive hypothesis we will find it by the end of the algorithm. If we have encountered the true majority element already, we can say the following. This element could have appeared only up to $\frac{x}{2}$ times so far. Of the remaining $n - x$ elements, at least $(\frac{n}{2} + 1 - \frac{x}{2}) = \frac{n-x}{2} + 1$ must be the majority element. Thus the true majority element will also be the majority of the rest of the stream. We can then apply the inductive hypothesis again, completing the proof.

2 Streaming [Final Fall 2016]

(a) Show that, for every $\epsilon > 0$, you can use the count-min data structure to determine the frequency of the most frequent element within an *additive* error of $\epsilon \cdot n$, where n is the number of items in the stream, with the same space complexity $O(\epsilon^{-1}(\log n)(\log n + \log \Sigma))$ of count-min. (You do not need to rederive the space complexity analysis and correctness analysis of count-min.)

Solution: Simply reduce this problem to the heavy hitters problem: We know that using $\log^2 n$ memory we can with high probability $(1 - 1/n)$ for all the elements correctly determine their frequency within an additive error of $\epsilon \cdot n$. After each element is seen, check its frequency and return the maximum frequency found. With high probability, none of the estimates are overestimating by more than $\epsilon \cdot n$, and they cannot be underestimates, so our estimate for the maximum frequency is likely within the desired bound. (Also, note that for each element we check its frequency right after its last appearance).

Common mistakes:

(a) Simply explaining the heavy-hitters algorithm, hash structure of the count-min sketch, and/or justifying the memory usage that leads to the additive error ϵ guarantee. The point was to take this as a given and explain how this would be used for finding the frequency of the most frequent element.

(b) Thinking that count-min finds the element with the minimum frequency, and then either somehow "inverting" it to find the max frequency, or running it n times (which is impossible in the streaming model, where you only see each item once).

(c) Directly using the hash values at the end of the algorithm to divine the most frequent element. This doesn't work because there's no way to link all the buckets an element would hash to without knowing the element.

(b) If ϵ is a constant and $|\Sigma|$ is polynomial in n, then the above algorithm uses space polynomial in $\log n$, but it only guarantees an additive, not multiplicative, approximation. Prove that deterministically estimating the frequency of the most frequent element within a multiplicative factor smaller than two requires linear space: specifically, the algorithm would never underestimate the frequency of an element, and would always predict less than twice the true frequency. *Hint: Recall that it takes linear space to tell if all elements are distinct.*

Solution: In the vein of the last discussion and lecture notes, let's show that we could compress a string if we had such an algorithm that gave multiplicative guarantees on the frequency of the most frequent element. Simply use the same construction: A creates the stream containing the number i iff the ith element of the bitstring to be compressed is a 1. Then, A sends the state of the stream s to B.

B can recreate the bitstring as follows. For each index i, reset the stream state to s, same as when sent by A, pass i though the stream, and check the frequency of the most frequent element. If the frequency is 2, B knows that i was in the original stream, and thus is a 1 in the bitstring. If the frequency is 1, it wasn't in the stream, and thus is a 0 in the bitstring. We know these numbers are accurate since we are guaranteed accuracy to within a factor of less than 2. Thus, B can recreate the original bitstring, so we know that it takes linear space to store the space for such an algorithm.

Alternate solution: We could use this algorithm to determine whether all the elements of a list are distinct, by taking a stream where all elements may or may not be distinct, running it through the algorithm, and after each element checking for its frequency. If this algorithm is accurate, we can return that they're all distinct if the frequency is always 1, or not all distinct if it ever says 2. But we know this takes linear space to do, so this algorithm must take linear space as well.

Common mistakes:

(a) Asserting that "finding whether all elements are distinct reduces to finding the most frequent element" or similar. This is what you're being asked to explain, so you can't simply assert it without justification.

(b) Giving a linear-space algorithm; this doesn't show that is required.

(c) Arguing that to achieve high enough multiplicative accuracy with this algorithm the parameters of the same heavy hitter algorithms would take linear space; again, this doesn't mean some other algorithm couldn't use less space.

(d) Saying something general like, "we cannot do this with hashing so we need linear space."

3 Reservoir Sampling

(a) You have a length n stream of numbers, each of which is at most n. You want to randomly select one number fairly (all numbers should have equal probability of being selected). However, you don't know what n is beforehand, and you can only parse through the stream once! Design a $O(\log n)$-space algorithm that will randomly choose one element fairly.

Solution:

This is exactly the problem statement seen in the notes. Our algorithm is as follows:

- Keep the first item in memory
- When the i^{th} item arrives ($i > 1$),
  1) With probability $\frac{1}{i}$, discard the old item, and keep the new one in memory
  2) With probability $1 - \frac{1}{i}$, keep the old item and ignore the new one

This will give each item a probability $\frac{1}{n}$ of being chosen. This can be seen intuitively for $n = 1, 2$, and can be shown in general using induction (if this property holds for size n, if we were to increase the size to $n + 1$, the property will hold by definition of what to do on the $(n + 1)^{th}$ element).

(b) Now suppose that instead of wanting each element to have an equal chance of being chosen, you want later elements in the stream to be more likely. Specifically, we wish the probability of choosing element $i + 1$ to be twice the probability of choosing element i for all $i > 1$ and the probability of choosing the first element to be same as that for the second. Design an $O(\log n)$-space algorithm to implement this sampling method.

Solution: We can take a similar approach as the solution for part (a). However, instead of replacing the old item with probability $\frac{1}{i}$, we replace it with probability $\frac{1}{2}$. We note that for $k \geq i > 1$, the probability that item i is in the reservoir after item k arrives is just the probability that we replaced at round i but not any any round between $i + 1$ and k. Since there are $k - i$ rounds between $i + 1$ and k, this probability is just $\frac{1}{2} \cdot \left(\frac{1}{2}\right)^{k-i} = \left(\frac{1}{2}\right)^{k-i+1}$. Thus, after all n items have arrived, the probability of choosing element $i + 1$ is $(\frac{1}{2})^{n-i} = 2 \cdot (\frac{1}{2})^{n-i+1}$, ie twice the probability of choosing element i. We additionally have that the probability of sampling the first element is the probability of not replacing in any of the remaining $n - 1$ rounds, which is just $(\frac{1}{2})^{n-1}$; this is indeed equal to the probability of sampling element two as desired.