Deterministic data streaming

Consider a stream of n items, where items can appear more than once in the stream. The problem is to find the most frequently appearing item in the stream (where ties broken arbitrarily if more than one item satisfies the latter). Suppose that only k items can be stored, one item per memory cell, where the available storage is $k + O(1)$ memory cells. Show that the problem cannot be solved deterministically under the following rules: the algorithm can access only $O(log^c n)$ bits for each of the k items that it can store, and can read the next item of the stream; you, the adversary, have access to all the stream, and the content of the k items stored by the algorithm, and can decide what is the next item that the algorithm reads (please note that you cannot change the past, namely, the items already read by the algorithm) . Hint: it is an adversarial argument based on the k items chosen by the hypothetical deterministic streaming algorithm, and the fact that there can be a tie on $> k$ items till the last minute.

Show that the problem cannot be solved deterministically under the following rules: the algorithm can only use b bits, and read the next item of the stream, one item at a time. You, the adversary, have access to all the stream, and the content of the b bits stored by the algorithm: you cannot change those b bits and the past, namely, the items already read by the algorithm, but you can change the future, namely, the next item to be read. Since the algorithm must be correct for any input, you can use any amount of streams to be fed to the algorithm and as many distinct items as you want. [Hint: it is an adversarial argument based on the fact that, for many streams, there can be a tie on the items.]

SOLUTION

Any deterministic algorithm can be described by a deterministic finite state machine, in which every state given an input move deterministically to another state. Suppose to have a streaming of n element, k memory cell to store the frequencies, and θ are the bit necessary to store the algorithm state. Therefore we can have 2^θ possible states.

Let's suppose to input $2^\theta + 1$ times the same element x to the algorithm. Then, it must be the case that the algorithm is going to be in the same state in two different moments (notice that the possible state are 2^θ). Let's call A that particular state, let's use A_i and A_j to denote the two different moments in which we arrive in state A during the execution of the algorithm (without loss of generality we can assume $i < j - 1$, we will later handle the $i = j - 1$ case).

Now let's input to the algorithm an element $y \neq x$ for $(i + 1)$-times, starting from both A_i and A_j. Then let's proof that the algorithm replies in two different ways, depending on whether it starts from A_i or A_j:

Case 1: when we start from A_i, we will end up in this situation:
(for simplicity, we'll use a JSON-like syntax for representing the counters)
Counters: $\{x : i, y : i + 1\} \implies$ the algorithm outputs y as the most frequent element.

Case 2: when we start from A_i, we will end up in this situation:
Counters: $\{x : j, y : i + 1\} \implies$ since $j > i + 1$, the algorithm outputs x.

We've seen how, starting from the same state A and with the same input, the algorithm produces different outputs; therefore it cannot be deterministic.

case $i = j - 1$

If i=j-1, then the algorithm contains the transition $(A_i, x) \to A_j = A_{i+1}$. This means that the algorithms remains in state A after receving x. In particular, $A_{i+2} = \delta(\delta(A_i, x), x) = \delta(A_i, x) = A_i$, and thus we can just take $j = i + 2$ and we fall in the previous case $i < j + 1$.