Problem Set 1: COSC-548 (Streaming Algorithms)

Due September 29th, 2016 by end of class

1. Given two streams $\sigma_1 = \langle a_1, \ldots, a_m \rangle$ and $\sigma_2 = \langle b_1, \ldots, b_m \rangle$ with all a_i, b_i drawn from a data universe $[n]$, let $\sigma := \sigma_1 \circ \sigma_2$ denote their concatenation.

   In class, we saw a randomized streaming algorithm that made a single pass over σ, used space $O(\log(m \cdot n))$, and achieved the following guarantee.

   - If $a_j = b_j$ for all $j \in [m]$, then the algorithm output EQUAL with probability 1.
   - If there exists some j such that $a_j \neq b_j$, then the algorithm output NOTEQUAL with probability at least $1 - 1/m$.

   Give a streaming algorithm with space cost $O(\log(m \cdot n))$ that tests whether σ_1 and σ_2 have the same frequency vectors (instead of testing whether σ_1 and σ_2 are equal as ordered lists, which is what the algorithm given in class achieved). That is, letting f_1 denote the frequency vector of σ_1 and f_2 denote the frequency vector of σ_2, your algorithm should achieve the following guarantee.

   - If $f_1 = f_2$ entry-wise, then the algorithm must output EQUAL with probability 1.
   - If f_1 does not equal f_2 entry-wise, then the algorithm must output NOTEQUAL with probability at least $1 - 1/m$.

   Does your algorithm work in the turnstile streaming model?

   (For reference, frequency vectors and the turnstile streaming model are defined in Lecture 0 of Amit Chakrabarti's notes at `http://www.cs.dartmouth.edu/~ac/Teach/CS49-Fall11/Notes/lecnotes.pdf`).

2. Recall the Misra-Gries algorithm from class in the context of the vanilla streaming model. When run on a data stream $\sigma = \langle a_1, \ldots, a_m \rangle$, the algorithm maintains k (item, count) pairs. Every time the algorithm processes a stream update a_i, the algorithm looks to see if a_i is assigned a counter, and if so it increments the counter. If not, and an unassigned counter exists, the algorithm assigns the counter to a_i and sets the count to 1. If no unassigned counter exists, the algorithm decrements all counters by 1, and marks all counters set to 0 as unassigned. When asked to provide an estimate $\hat{f}_j^{MG,k}$ for the frequency of item j, the algorithm returns 0 if j is not assigned a counter, and returns the value of the counter assigned to j otherwise.

   We proved in class that $0 \leq f_j - \hat{f}_j^{MG,k} \leq m/(k+1)$. Prove that in fact for any $c < k$, the following holds: $0 \leq f_j - \hat{f}_j^{MG,k} \leq F^{\text{res}(c)}/(k+1-c)$. Here, $F^{\text{res}(c)}$ denotes the sum of the frequencies of all but the c most frequent items.

3. In class, we also discussed the SpaceSaving algorithm, which works as follows. When run on a data stream $\sigma = \langle a_1, \ldots, a_m \rangle$, the algorithm maintains k (item, count) pairs. Every time the algorithm processes a stream update a_i, the algorithm looks to see if a_i is assigned a counter, and if so it increments the counter. If not, and an unassigned counter exists, the algorithm assigns the counter to a_i and sets the count to 1. If no unassigned counter exists, the algorithm finds the smallest counter, reassigns it to a_i, and increments the count. When asked to provide an estimate $\hat{f}_j^{SS,k}$ for the frequency of item j, the algorithm returns the minimum counter value if j is not assigned a counter, and returns the value of the counter assigned to j otherwise.

   Prove that SpaceSaving and Misra-Gries are isomorphic in the following sense. After running Misra-Gries with k counters on a stream σ, one can use the resulting k (item, count) pairs to compute $\hat{f}_j^{SS,k+1}$. Similarly, after running SpaceSaving with $k+1$ counters on σ, one can use the resulting $k+1$ (item, count) pairs to compute $\hat{f}_j^{MG,k}$.

   Hint: Let $\min^{SS,k+1}$ denote the minimum counter value in the SpaceSaving data structure, and let $\hat{m}^{MG,k}$ denote the sum of all the counter values in the Misra-Gries data structure. Show that $\hat{f}_j^{SS,k+1} - \hat{f}_j^{MG,k} = \min^{SS,k+1} = (m - \hat{m}^{MG,k})/(k+1)$. Do this by induction on the number of stream updates. That is, show that this holds for streams with just a single update. Then assume that it holds for all streams with at most $m-1$ stream updates, and show that it also holds after the algorithms process the m'th stream update.

4. Suppose we throw m balls into n bins at random, with $m \geq n$.

   (a) Let the random variable B_i denote the number of balls in bin i. What is $\mathbb{E}[B_1]$?

   (b) Suppose $m = 100n \ln n$. Show that the number of balls in bin i does not differ from the expectation by more than (say) $50 \ln n$ with probability at least $1 - 1/n^2$.

   Hint: Use the following form of the Chernoff bound that we covered in class. Let $X_1, \ldots, X_m$ be independent Poisson trials with expectation p (i.e., X_i takes value 1 with probability p, and 0 with probability $1-p$). Let $X = \sum_{i=1}^m X_i$ and $\mu = \mathbb{E}[X] = m \cdot p$. Then the following holds:

$$\text{For } 0 < \delta \leq 1, \Pr(|X - \mu| > \delta\mu) \leq 2 \cdot e^{-\mu\delta^2/3}.$$

   (c) Let $m = n$. Prove that with probability at least $1 - 1/n^2$, the maximum number of balls in any bin is at most $c \cdot \log n / \log\log n$ for some constant $c > 0$.

   Hint: Use the following stronger version of the Chernoff bound. Let $X_1, \ldots, X_m$ be independent Poisson trials with expectation p. Let $X = \sum_{i=1}^m X_i$ and $\mu = \mathbb{E}[X] = m \cdot p$. Then the following holds:

$$\text{For any } \delta > 0, \Pr(X - \mu > \delta\mu) \leq \left(e^{\delta}/(1+\delta)^{(1+\delta)}\right)^{\mu}.$$

   Remark. The $O(\log n/\log\log n)$ upper bound on the maximum load of any bin implies that a simple hash table implementation based on chaining can support any sequence of n lookup, insert, and delete operations in $O(\log n/\log\log n)$ time with high probability, while using $O(n \log |\mathcal{U}|)$ bits of space, where $\mathcal{U}$ is the data universe from which keys are drawn. Specifically, the hash table maintains n "buckets". It hashes each key to a random bucket, and for each bucket it maintains a linked list storing all keys residing in the bucket. The entire linked list for any bucket can be searched in time proportional to the length of the linked list, which is $O(\log n/\log\log n)$ with high probability. This runtime beats that of self-balancing binary search trees by a $\Theta(\log\log n)$ factor. Later in the semester, we will see more sophisticated hash table implementations achieving faster runtimes.