10^{TH} TUTORIAL ON RANDOMIZED ALGORITHMS

Streaming algorithms

Notation:

- *Input:* sequence of items $a_1, a_2, \ldots, a_m$, where $a_i \in [N]$
- *Goal:* process the stream in *one pass with low memory*, ideally $\text{poly}(\log(N+m))$, and estimate desired statistics of the stream.

Let f_j be the frequency of $j \in [N]$, i.e., the number of times j appears in the streams.

1. *Strict majority in constant space.* Design a streaming algorithm using as small space as possible that finds the *majority element* if there is one, i.e., an element j with frequency $f_j > m/2$. If there is no majority element, the output can be arbitrary. (Alternatively, you may use the second pass over the input to verify that the output element indeed achieves majority.)

2. *Finding heavy hitters deterministically.*

a) Generalize the previous algorithm so that for a parameter $k \in \mathbb{N}$, it returns all elements j with $f_j > m/k$, the so-called “heavy hitters”, and possibly some other elements.

b) The algorithm can also return an estimate of f_j for any given $j \in [N]$. What kind of guarantee do we get from it?

c) The algorithm returns all heavy hitters but possibly also elements that may be far from heavy hitters (in extreme cases occurring just once in the stream). How to modify the algorithm so that all of the returned elements are close to being heavy hitters, say, having frequency at least $m/(2k)$ while we still return all of the true heavy hitters.

3. *Back to randomization: The k Minimum Values (KMV) sketch.* As in the lecture, we would like to count the number of distinct elements, i.e., estimate set cardinality. We look at a different approach which is actually quite popular in practice: For a parameter k and a hash function $h : [N] \to [0,1]$, store the k smallest hash values of the distinct stream elements, i.e., we store k pairs (item j, $h(j)$); note that the sketch is not updated if any stream item arrives again. When queried for cardinality, return $(k-1)/v_k$, where v_k is the k-th smallest hash value (the largest one stored).

Analyze the algorithm and prove that for $k \geq c/\varepsilon^2$ (where c is a large enough constant) the algorithm gives an ε-approximation of $F_0 =$ the number of distinct elements.

a) Show that the probability of $(k-1)/v_k > (1+\varepsilon)\cdot F_0$ is an arbitrarily small constant (depending on c). For now, assume a fully random hash function (i.e., all items hashed into $[0,1]$ uniformly and independently). A similar analysis applies to the other inequality, that is, bounding the probability of $(k-1)/v_k < (1-\varepsilon)\cdot F_0$.

b) What is wrong with h being fully random? What kind of hash functions would be sufficient for the analysis?

c) How to get a $(1+\varepsilon)$-approximation with probability $1-\delta$ for any $\delta > 0$? How does the space complexity depend on δ?

4. *Median estimation.* We would like to find the median, i.e., the $\lceil m/2 \rceil$-th largest element of the stream, assuming items are from a totally ordered domain U (e.g., U can be integers or floating point numbers). This task provably requires $\Omega(m)$ memory (i.e., storing essentially all of the input items). To get a more space-efficient streaming algorithm, we relax the requirement to return the *exact* median. More precisely, for a parameter $\varepsilon > 0$, we would like to get an ε-approximate median with high probability, i.e., an element a of the stream that is the k-th largest for $k = (0.5 \pm \varepsilon) \cdot m$.

Consider the following simple sampling algorithm: Draw s uniform samples from the stream (with replacement).

a) How to implement the sampling as a streaming algorithm? The stream length m should be unknown in advance.

b) Given the s uniform samples, what should the algorithm return?

c) How large s do we need to get an ε-approximate median with probability at least $1-\delta$ for a given $\delta > 0$?

Bonus: try the same algorithm but sampling without replacement.