1 Reservoir Sampling

Problem definition: Pick a uniformly random item from the stream.

Reservoir Sampling [Vitter, 1985]:
1. Init: $s = \text{null}$
2. Update: When the next item $\sigma_j$ is read, toss a biased coin and with probability $1/j$ let $s = \sigma_j$ in the stream (note we need to maintain $j$)
3. Output: $s$

Lemma: Assuming every $\sigma_j \in [n]$, this algorithm uses storage $O(\log(n + m))$ and its output is a uniform item from the stream, i.e., each item $\sigma_j$ (each position) ends up being output with the same probability $1/m$.

Note that items appearing many times are output with high probability.

Exer: Prove this lemma.

Exer: Design a streaming algorithm that at any point $m$ (not known in advance) receives a query $S \subset [n]$ and outputs and estimate what fraction of items in the stream belong to $S$ within additive error $\epsilon$. Note that $S$ is given only at query time (not in advance).

Hint: Maintain $O(1/\epsilon^2)$ random samples and use them to estimate the fraction in $S$.

Exer: Design an algorithm that samples $s$ items without replacement from an input stream $\sigma = (\sigma_1, \ldots, \sigma_m)$. The algorithm’s memory requirement should be $O(s)$ words ($s$ is a parameter known in advance). Prove that the algorithm’s output has the correct distribution.

Hint: The goal is essentially to sample $s$ distinct indices $(i_1 < \cdots < i_s)$ uniformly at random. In contrast, executing the Reservoir Sampling algorithm $s$ times in parallel gives $k$ samples with...
replacement, i.e., the same \( i \in [m] \) could be reported more than once.

## 2 Frequency-vector model

A famous and common setting for data-stream problems lets the input be a stream of \( m \) items from a universe \( [n] = \{1, \ldots, n\} \); the stream \( \sigma = (\sigma_1, \ldots, \sigma_m) \) implicitly defines a frequency vector \( x \in \mathbb{R}^n \), where coordinate \( x_i \) counts the frequency of item \( i \in [n] \) in the stream.

**Example:** The sequence of IP addresses observed by a router. Here, \( n = 2^{32} \) is huge but the vector \( x \) is sparse (many zeros).

**Remark:** In this setting, it is common to assume \( m = \text{poly}(n) \), hence one machine word can store value in the ranges \([n]\) and \([m]\). The usual goal is to achieve storage requirement \( \text{polylog}(n) \).

**Example Problems:** Two classical computational problems ask for the most frequent item and for the number of distinct items, which can be expressed in terms of the frequency vector \( x \) as \( \|x\|_\infty \) and \( \|x\|_0 \), respectively.

Suppose we are guaranteed that one item appears more than half the time, i.e., there exists (unknown) \( i \in [n] \) such that \( x_i > m/2 \). Design a streaming algorithm with \( O(\log n) \) storage that finds this item \( i \). Hint: Store only two items.

Can you provide a \((1 + \epsilon)\)-approximation to its frequency? Can you extend it to every \( k \) (i.e., frequency \( > m/k \)?)

**Variations and further questions (we will discuss only some of these):**

- \( \|x\|_0 \) (distinct elements)
- heavy hitters (\( \|x\|_\infty \) when it is guarantee to be “large”)
- \( \|x\|_2 \) (reflects the probability that two random items from the stream are equal)
- more generally \( \|x\|_p \)
- \( \ell_p \)-sampling
- item deletions (turnstile updates to \( x \)), now even \( \|x\|_1 \) is interesting
- sliding window (always refer to the \( w \) most recent items, for a parameter \( w \) known in advance)
- multiple passes over the input

## 3 Distinct Elements

**Problem Definition:** Let \( x \in \mathbb{R}^n \) be the frequency vector of the input stream, and let \( \|x\|_0 = |\{i \in [n] : x_i > 0\}| \) be the number of distinct elements in the stream. It’s also called the \( F_0 \)-moment of \( \sigma \).

**Naive algorithms:** Storage \( O(n) \) (a bit for each possible item) or \( O(m \log n) \) (list of seen items) bits.
Algorithm FM [Flajolet and Martin, 1985]:

It employs a “hash” function $h : [n] \rightarrow [0, 1]$ where each $h(i)$ has an independent uniform distribution on $[0, 1]$. (This is an “idealized” description, because even though we can generate $n$ truly random bits, we cannot store and re-use them.)

Idea: We will have exactly $d^* = \|x\|_0$ distinct hashes, and since they are random, by symmetry their minimum should be around $1/(d^* + 1)$.

1. Init: $z = 1$ and a hash function $h$
2. Update: When item $i \in [n]$ is seen, update $z = \min\{z, h(i)\}$
3. Output: $1/z - 1$

Storage requirement: $O(1)$ words (not including randomness); we will discuss implementation issues later.

Denote by $d^* := \|x\|_0$ the true value, and let $Z$ denote the final value of $z$ (to emphasize it is a random variable).

**Lemma 1:** $\mathbb{E}[Z] = 1/(d^* + 1)$.

Note: This is the expectation of $Z$ and not of its inverse $1/Z$ (as used in the output).

**Proof:** We will use a trick to avoid the integral calculation (which is actually straightforward). Choose an additional random value $X$ uniformly from $[0, 1]$ (for sake of analysis only), then by the law of total expectation

$$
\mathbb{E}[Z] = \mathbb{E}\left[\mathbb{E}_X[1_{X < Z} \mid Z]\right] = \mathbb{E}\left[\mathbb{E}_X[1_{X < Z} \mid Z]\right] = 1/(d^* + 1).
$$

**Lemma 2:** $\mathbb{E}[Z^2] = \frac{2}{(d^* + 1)(d^* + 2)}$ and thus $\text{Var}[Z] \leq (\mathbb{E}[Z])^2$.

**Exer:** Prove this lemma using the above trick with two new random values (and/or prove both by calculating the integral).

**Algorithm FM+:**

1. Run $k = O(1/\varepsilon^2)$ independent copies of algorithm FM, keeping in memory $Z_1, \ldots, Z_k$ (and functions $h_1, \ldots, h_k$)
2. Output: $1/\bar{Z} - 1$ where $\bar{Z} = \frac{1}{k} \sum_{i=1}^k Z_i$

As before, averaging reduces the standard deviation by factor $\sqrt{k}$, and then by Chebyshev’s inequality, WHP

$$
\bar{Z} \in [d^* \pm O(d^* / \sqrt{k})] = [d^* \pm \varepsilon d^*].
$$

Storage requirement: $O(k)$ words (not including randomness); we will discuss implementation issues later.

**Remark:** The storage can be improved similarly to the probabilistic counting. It suffices to store a $(1 + \varepsilon)$-approximation of $z$, which can reduce the number of bits from $O(\log n)$ (in a “typical”
implementation of the real-valued hashes) to $O(\log \log n)$. A particularly efficient 2-approximation is to store the number of zeros in the beginning of $z$’s binary representation.

Remark: Notice this algorithm does not work under deletions.

4 Frequency Moments and the AMS algorithm

$p$-norm problem: Let $x \in \mathbb{R}^n$ be the frequency vector of the input stream, and fix a parameter $p > 0$.

Goal: estimate its $\ell_p$-norm $\|x\|_p = (\sum_i |x_i|^p)^{1/p}$. We focus on $p = 2$.

Theorem 1 [Alon, Matthias, and Szegedy, 1996]: One can estimate the $\ell_2$ norm of a frequency vector $x \in \mathbb{R}^n$ within factor $1 + \varepsilon$ [with high constant probability] using storage requirement of $s = O(\varepsilon^{-2})$ words. In fact, the algorithm uses a linear sketch of dimension $s$.

Algorithm AMS (also known as Tug-of-War):

1. Init: choose $r_1, \ldots, r_n$ independently at random from $\{-1, +1\}$
2. Update: maintain $Z = \sum_i r_i x_i$
3. Output: to estimate $\|x\|_2^2$ report $Z^2$

The sketch $Z$ is linear, hence can be updated easily.

Storage requirement: $O(\log(nm))$ bits, not including randomness; we will discuss implementation issues a bit later.

Will be continued next class.