New York University Tandon School of Engineering
Computer Science and Engineering

CS-GY 6763: Homework 1.
Due Monday, February 14th, 2021, 11:59pm ET.

Collaboration is allowed on this problem set, but solutions must be written-up individually. Please list collaborators for each problem separately, or write "No Collaborators" if you worked alone.

For just this first problem set, 10% extra credit will be given if solutions are typewritten in LaTeX.

Problem 1: Collision free hashing.

(15 pts) Consider inserting m unique keys into a hash table of size $n = m^2$ using a uniformly random hash function. Using the mark-and-recapture analysis from class, we know that the expected number of collisions in the table is $\frac{m\cdot(m-1)}{2m^2} < 1/2$. So, by Markov's inequality, with probability $> 1/2$, the table has no collisions (strictly < 1 collisions). Thus, we can look up items from the table in worst-case $O(1)$ time.

1. Give an alternative proof of the fact that we have no collisions with $> 1/2$ probability in a table of size cm^2 for some sufficiently large constant c. Specifically, to have no collisions, we must have the following events all happen in sequence: the second item inserted into the hash table doesn't collide with an existing item, the third item inserted doesn't collide with an existing item, ..., the m^{th} item inserted doesn't collide with an existing item. Analyze the probability these events all happen.

   Hint: You might want to use the standard fact that $\frac{1}{2e} \leq (1 - \frac{1}{n})^n \leq \frac{1}{e}$ for any positive integer $n \geq 2$.

2. Consider the following alternative scheme: build two tables, each of size $O(m^{1.5})$ and choose a separate random hash function (independently at random) for each table. To insert an item, hash it to one bucket in each table and place it in the emptier bucket. Show that, if we're hashing m items, then with probability $\geq 1/2$, there will be no collisions in either table. You may assume uniformly random hash functions.

3. Modify the above scheme to use $O(\log m)$ tables. Prove that this approach yields a hashing scheme with space $O(m \log m)$ and $O(\log m)$ worst-case look-up time.

Problem 2: Why does Count-Min work well in practice?

(15 pts) We showed that, for any $\epsilon, \delta \in (0, 1)$, Count-Min can estimate the frequency of any coordinate f_i in the frequency vector $f \in \mathbb{R}^n$ up to additive error $\epsilon\|f\|_1 = \epsilon \sum_{i\in[n]} |f_i|$ with probability $1 - \delta$ using $O(\frac{1}{\epsilon}\log(\frac{1}{\delta}))$ words of space. In practice, it is often observed that this bound is pessimistic: the algorithm performs better than expected. In this problem, you will establish one reason why.

Given any vector $x \in \mathbb{R}^n$ and integer $1 \leq k \leq n$, we define the *k-tail* of x, denoted x_{-k}, to be the vector x with the k largest (in absolute value) coordinates set equal to zero. In other words, if $S \subset [n]$ are the k largest coordinates of x, we have

$$(x_{-k})_i = \begin{cases} x_i & \text{if } i \notin S \\ 0 & \text{otherwise} \end{cases}$$

For instance, if $x = [8, 2, 4, -5, -7]$, then $x_{-3} = [0, 2, 4, 0, 0]$. Note that, in general, and especially if most of the updates were made to a small number of $\leq k$ coordinates, we can have $\|f_{-k}\|_1 = \sum_{i\notin S} |f_i| \ll \|f\|_1$. For example, it has been reported that up to 95% of YouTube video views come from just 1% of videos. Prove that for any $\epsilon \in (0, .5)$, Count-Min can actually return an estimate $\tilde{f}_i$ to f_i for any individual item $i \in [n]$ satisfying:

$$f_i \leq \tilde{f}_i \leq f_i + \epsilon\|f_{-(1/\epsilon)}\|_1$$

with 9/10 probability, using at most $O(\frac{1}{\epsilon})$ space. This is known as the ℓ_1-*tail error guarantee*, and is strictly better than the $\epsilon\|f\|_1$ error bound shown in class.

Problem 3: Distinct Elements Estimation in a Stream

(15 pts) In class we saw how to use Count-Min sketch to solve the k-frequent items problem in the streaming model. Besides frequent items, another important statistic one may want to estimate from streaming data is the number of non-zero coordinates in f — this is known as *distinct elements estimation*. In particular, the goal is to output an estimate R with

$$(1-\epsilon)\|f\|_0 \leq R \leq (1+\epsilon)\|f\|_0$$

Where $\|f\|_0 = |\{i \mid f_i \neq 0\}|$ is the "zero-th moment" of f (i.e., number of non-zero coordinates). In this problem, we will analyze a important streaming algorithm for this problem. The algorithm is as follows:

Algorithm for Distinct Elements Estimation

1. Select a uniformly random hash function $h : [n] \to [0,1]$.[a]
2. For each update $a_i \in [n]$ in the stream (causing the change $f_{a_i} \leftarrow f_{a_i} + 1$), compute $h(a_i) \in [0,1]$, and store it if it is one of the the k-smallest unique hash values seen so far.
3. At the end of the stream, let $h_1 < h_2 < \cdots < h_k \in [0,1]$ be the k-smallest hash values seen during the stream (i.e., the k-smallest real numbers in the set $\{h(i) : i \in [n],\ f_i \neq 0\}$), which the algorithm now has stored. Output the estimator $R = \frac{1}{h_k} \cdot k$.

[a] For now, don't worry about what it means for a hash function to output a real number, we can always discretize the output to be represented in a finite number of bits.

You may assume that all hash values in the set $\{h(i) \mid i \in [n]\}$ are unique.[1] Then note that, if $\|f\|_0 \leq k$, then the unique hashes for all non-zero f_i will be stored by the algorithm, and we will know $\|f\|_0$ exactly. Thus, for the following problems, we may assume that $\|f\|_0 \geq ck$ for any arbitrarily large constant c. In the following, fix any $\epsilon \in (0, \frac{1}{2})$.

1. Show that by setting $k = C \cdot \frac{1}{\epsilon^2}$ for a big enough constant C, we have $\Pr[h_k < (1-\epsilon)\frac{k}{\|f\|_0}] < 1/10$.
2. Similarly, show, for the same value of k, that $\Pr[h_k > (1+\epsilon)\frac{k}{\|f\|_0}] < 1/10$. Conclude that $R = (1 \pm 2\epsilon)\|f\|_0$ with probability at least $1/5$.
3. **Boosting success probability:** Describe how to modify the algorithm to be correct with probability $1-\delta$, using at most $O(k \log(1/\delta))$ space. You may ignore the space required to store random hash functions h for the purpose of this exercise.

Problem 4: Randomized methods for COVID-19 group testing.

(10 pts) One of the most important factors in controlling diseases like COVID-19 is testing. Unfortunately, testing can be expensive and slow. One way to make it cheaper is by testing patients in *groups*. The biological samples from multiple patients (e.g., multiple nose swabs) are combined into single test tube and tested for COVID-19 all at once. If the test comes back negative, we know everyone in the group is negative. If the test comes back positive, we do not know which patients in the group actually had COVID-19, so further testing would be necessary. There's a trade-off here, but it turns out that, overall, group testing can save on the total number of tests run.

1. Consider the following deterministic "two-level" testing scheme. We divide a population of n individuals to be tested into C arbitrary groups of the same size. We then test each of these groups in aggregate. For any group that comes back positive, we retest all members of the group individually. Show that there is a choice for C such that, if k individuals in the population have COVID-19, we can find all of those individuals with $\leq 2\sqrt{nk}$ tests. You can assume k is known in advance (often it can be estimated

[1] Note that this will occur with good probability if we discretize the output of h so that it maps to the set of values $\left\{\frac{1}{cn^2}, \frac{2}{cn^2}, \frac{3}{cn^2}, \ldots \frac{cn^2-1}{cn^2}, 1\right\}$ for a large enough constant c, can you see why?

accurately from the positive rate of prior tests). This is already an improvement on the naive n tests when $k < 25\% \cdot n$.

2. We can use randomness to do better. Consider the following scheme: Collect $q = O(\log n)$ nose swabs from each individual (I know... not pleasant). Then, repeat the following process q times: randomly partition our set of n individuals into C groups, and test each group in aggregate. Once this process is complete, report that an individual "has COVID" if the group they were part of tested positive all q times. Report that an individual "is clear" if any of the groups they were part of tested negative. Show that for $C = O(k)$, with probability $9/10$, this scheme finds all truly positive patients and reports no false positives. Thus, we only require $O(k \log n)$ tests!

3. **Extra Credit – optional.** Show that no scheme can use $o(k \log(n/k))$ tests and succeed with probability $> 2/3$. So, for small k, the approach above is essentially optimal up to constant factors!

Problem 5: Hashing around the clock.

(15 pts) In modern systems, hashing is often used to distribute data items or computational tasks to a collection of servers. What happens when a server is added or removed from a system? Most hash functions, including those discussed in class, are tailored to the number of servers, n, and would change completely if n changes. This would require rehashing and moving all of our m data items, an expensive operation.

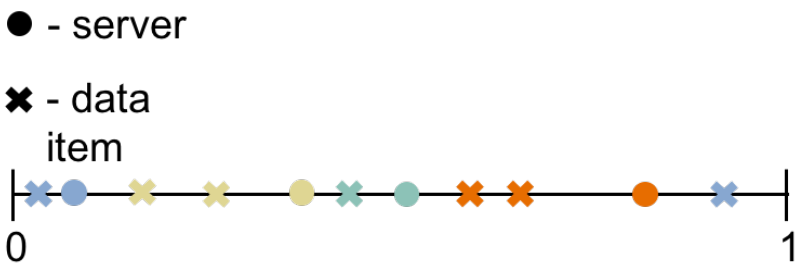

Figure 1: Each data item is stored on the server with matching color.

Here we consider an approach to avoid this problem. Assume we have access to a completely random hash function that maps any value x to a real value $h(x) \in [0, 1]$. Use the hash function to map *both* data items and servers randomly to $[0, 1]$. Each data item is stored on the first server to its right on the number line (with wrap around – i.e. a job hashed below 1 but above all serves is assigned to the first server after 0). When a new server is added to the system, we hash it to $[0, 1]$ and move data items accordingly.

1. Suppose we have n servers initially. When a new server is added to the system, what is the expected number of data items that need to be relocated?

2. Show that, with probability $> 9/10$, no server "owns" more than an $O(\log n/n)$ fraction of the interval $[0, 1]$. **Hint:** This can be proven without a concentration bound.

3. Show that if we have n servers and m items and $m > n$, the maximum load on any server is no more than $O(\frac{m}{n} \log n)$ with probability $> 9/10$.