Note: Your TA may not get to all the problems. This is totally fine, the discussion worksheets are not designed to be finished in an hour. The discussion worksheet is also a resource you can use to practice, reinforce, and build upon concepts discussed in lecture, readings, and the homework.

1 Estimating Votes

Suppose we have a stream of votes of form (Id, Yes) or (Id, No) which has person's Id (that is unique to them) and whether they voted Yes/No. We would like to estimate the fraction of Yes votes. Unfortunately, many people have voted multiple times. People who voted multiple times voted for the same option each time.

The Distinct Elements algorithm takes a stream as inputs and outputs $\tilde{n} \in [(1-\epsilon)n, (1+\epsilon)n]$ with probability $1-\delta$, where n the number of distinct elements seen in the stream, using small memory. Let $S(n, \epsilon, \delta)$ be the space complexity Distinct Elements uses in terms of n, ϵ, δ.

Using the Distinct Elements algorithm as a black box, provide an algorithm for estimating the fraction of "Yes" votes within a factor of, say, $(1+3\epsilon)$. You should count the vote given by each Id only once. State its space complexity in terms of S.

Challenge: Justify the error bound on your algorithm (you may assume $\epsilon < 1/3$ for simplicity).

Solution: The algorithm is just to run two copies of Distinct Elements. Pass the ID of "Yes" votes to the first, and pass the ID of all votes to the second. When queried for the fraction, we return the ratio of the outputs of the two data structures.

The space complexity is at most $2 \cdot S(n, \epsilon, \delta)$ if n is the number of distinct voters.

By a union bound, if the number of Yes votes is m, then the first Distinct Elements data structures returns a value in $[(1-\epsilon)m, (1+\epsilon)m]$ with probability $1-\delta$, and the second returns a value in $[(1-\epsilon)n, (1+\epsilon)n]$ with probability $1-\delta$. So our algorithm returns a value in $[\frac{1-\epsilon}{1+\epsilon}m, \frac{1+\epsilon}{1-\epsilon}m]$ with probability $(1-\delta)^2 \geq 1-2\delta$. For $\epsilon \leq 1/3$, we have $(1+\epsilon)/(1-\epsilon) \leq 1+3\epsilon$, and $(1-\epsilon)/(1+\epsilon) \geq 1-3\epsilon$. So our algorithm's output is in $[(1-3\epsilon)m/n, (1+3\epsilon)m/n]$ with probability at least $1-2\delta$. Put another way, using $O(1)$ copies of this data structure only worsens their collective error guarantee by $O(1)$.

2 Streaming an Approximate Median

In this question, we consider the following setting: We want to implement a data structure that supports the following operations using as little space as possible:

- *add*(x): Given $x \in \{1, 2, \ldots, m\}$, add x to the list of numbers seen so far.
- *median*(): Return the median of the list of numbers seen so far.

Let n be the number of elements seen so far. Naively, we can do this using $O(n \log m)$ bits of memory by just storing all numbers seen so far (each number requires $\log m$ bits to store). It turns out any data structure that supports these queries must use $\Omega(n \log m)$ bits of memory. In this problem, we will show that using approximation, we can do much better than this lower bound.

For simplicity, assume in all parts of this problem that *median*() is only called after *add*(x) has been called an odd number of times.

(a) We'll first start with a simple version of this data structure.

Describe how to implement a data structure using $O(\log n)$ bits of memory that for a predetermined $k, \epsilon > 0$ supports the following operations:

- *add*(x): Given $x \in \{1, 2, \ldots, m\}$, add x to the list of numbers seen so far.
- *median*(): Return "True" if the median of the list of numbers seen so far is in the range $[k, (1+\epsilon)k)$ and "False" otherwise.

Just a description of the data structure and a justification for the space complexity is needed.

(b) Based on the previous part, given a predetermined $\epsilon > 0$, describe how to implement a data structure using at most $O(\log_{1+\epsilon} m \log n) = O(\frac{\log m \log n}{\epsilon})$ bits of memory that supports the following operations:

- *add*(x): Given $x \in \{1, 2, \ldots, m\}$, add x to the list of numbers seen so far.
- *median*$()$: Letting M be the median of the list of numbers seen so far, return a value in the range $[\frac{M}{1+\epsilon}, (1+\epsilon)M]$.

Just a description of the data structure and a justification for the space complexity is needed.

Solution:

(a) Our data structure stores three counters a, b, c: the number of times *add*(x) was called on x in the ranges $[1, k), [k, (1+\epsilon)k), [(1+\epsilon)k, m]$ respectively. These three counters never exceed n, so we need $O(\log n)$ bits to store them. *median*$()$ returns "False" if $a \geq b + c$ or $c \geq a + b$ (as this implies the median is in either the first or third range), and "True" otherwise.

(b) We use $O(\log_{1+\epsilon} m)$ copies of the data structure from the previous part, one for each of $k = 1, (1+\epsilon), (1+\epsilon)^2, \ldots (1+\epsilon)^{\lfloor \log_{1+\epsilon} m \rfloor}$. When *add*$(x)$ is called, we call *add*(x) on each of these copies. When *median*$()$ is called, we call *median*$()$ on each of these copies, and return the k-value for whichever data structure outputs "True" (one must output "True", since the union of the ranges considered by the data structures contains $[1, m]$).

Since we use $O(\log_{1+\epsilon} m)$ copies of a data structure using $O(\log n)$ bits of memory, our total space complexity is $O(\log_{1+\epsilon} m \log n)$.