Exercises in Algorithm Design

Issues to Consider

- How many queries are required, and what kind of composition can we use?
  - Is parallel composition possible?
  - Should we use sequential composition, advanced composition, or a variant of differential privacy?
- Can we use the sparse vector technique?
- Can we use the exponential mechanism?
- How should we distribute the privacy budget?
- If there are unbounded sensitivities, how can we bound them?
- Would synthetic data help?
- Would post-processing to "de-noise" help?

1. Generalized Sample and Aggregate

Design a variant of sample and aggregate which does *not* require the analyst to specify the output range of the query function f.

Ideas: use SVT to find good upper and lower bounds on $f(x)$ for the whole dataset first. The result of $clip(f(x), lower, upper)$ has bounded sensitivity, so we can use this query with SVT. Then use sample and aggregate with these upper and lower bounds.

2. Summary Statistics

Design an algorithm to produce differentially private versions of the following statistics:

- Mean: $\mu = \frac{1}{n} \sum_{i=1}^{n} x_i$
- Variance: $var = \frac{1}{n} \sum_{i=1}^{n} (x_i - \mu)^2$
- Standard deviation: $\sigma = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (x_i - \mu)^2}$

Ideas:

Mean

1. Use SVT to find upper and lower clipping bounds
2. Compute noisy sum and count, and derive mean by post-processing

Variance

1. Split it into a count query ($\frac{1}{n}$ - we have the answer from above) and a sum query
2. What's the sensitivity of $\sum_{i=1}^{n} (x_i - \mu)^2$? It's b^2; we can clip and compute $\sum_{i=1}^{n} (x_i - \mu)^2$, then multiply by (1) by post processing

Standard Deviation

1. Just take the square root of variance

Total queries:

- Lower clipping bound (SVT)
- Upper clipping bound (SVT)
- Noisy sum (mean)
- Noisy count
- Noisy sum (variance)

3. Heavy Hitters

Google's RAPPOR system is designed to find the most popular settings for Chrome's home page. Design an algorithm which:

- Given a list of the 10,000 most popular web pages by traffic,
- Determines the top 10 most-popular home pages out of the 10,000 most popular web pages

Ideas: Use parallel composition and take the noisy top 10

4. Hierarchical Queries

Design an algorithm to produce summary statistics for the U.S. Census. Your algorithm should produce total population counts at the following levels:

- Census tract
- City / town
- ZIP Code
- County
- State
- USA

Ideas:

Idea 1: *Only* compute the bottom level (census tract), using parallel composition. Add up all the tract counts to get the city counts, and so on up the hierarchy. Advantage: lowers privacy budget.

Idea 2: Compute counts at all levels, using parallel composition for each level. Tune the budget split using real data; probably we need more accuracy for the smaller levels of the hierarchy.

Idea 3: As (2), but also use post-processing to re-scale lower levels of the hierarchy based on higher ones; truncate counts to whole numbers; move negative counts to 0.

5. Workloads of Range Queries

Design an algorithm to accurately answer a workload of *range queries*. Range queries are queries on a single table of the form: "how many rows have a value for c between a and b?" (i.e. the count of rows which lie in a specific range).

Part 1

The whole workload is pre-specified as a finite sequence of ranges: $\{(a_1, b_1), \ldots, (a_k, b_k)\}$, and

Part 2

The length of the workload k is pre-specified, but queries arrive in a streaming fashion and must be answered as they arrive.

Part 3

The workload may be infinite.

Ideas:

Just run each query with sequential composition.

For part 1, combine them so we can use L2 sensitivity. When k is large, this will work well with Gaussian noise.

Or, build synthetic data:

- For each range $(i, i+1)$, find a count (parallel composition). This is a synthetic data representation! We can answer infinitely many queries by adding up the counts of all the segments in this histogram which are contained in the desired interval.
- For part 2, use SVT

For SVT: for each query in the stream, ask how far the real answer is from the synthetic data answer. If it's far, query the real answer's range (as a histogram, using parallel composition) and update the synthetic data. Otherwise just give the synthetic data answer. This way you *ONLY* pay for updates to the synthetic data.

In [ ]: