AGENDA

① LIST - RECOVERY
② APPLICATIONS:
- Sublinear-time Group Testing (HW)
- Applications to list-DECODING...
- Heavy hitters, two ways.
- Cute approach to IP-traceback.

Recall that last time we proved:

THM. For all \( r > 0 \), RS codes of rate \( R \) are \((1 - \sqrt{R(1+4r)}, \sqrt{r}R)\) list-decodable, and the Guruswami-Sudan algorithm can do the list-decoding in time \( \text{poly}(n,r) \).

We did this via the following algorithm:

GURUSWAMI-SUDAN ALGORITHM.

1. INTERPOLATION STEP
   Find a polynomial \( Q(X,Y) \) with \((t,b)\)-degree \( D = \sqrt{kn \cdot r \cdot (r+1)} \) so that \( O(x_i,y_i) = 0 \) with multiplicity \( r \) for \( i = 1, \ldots, n \).

2. ROOT-FINDING STEP.
   Return all \( f \) so that \( Q(x_i,f(x)) = 0 \).
   [Notice that there are \( \leq \deg_y(Q) \leq D/k = \sqrt{kn} \) of these.]

TODAY'S TERMITE TIDBIT

If their nest is under attack, some species of termites will defend tunnels by having the soldiers form a phalanx-type formation around the opening to bite off intruders.
OBSERVATION: There is no reason that the $x_i$’s need to be distinct.

What we actually proved was:

**Thm** Let $\mathcal{S}(x_i, y_i) : i = 1, \ldots, M \subseteq \mathbb{F}_q^2$ be any subset $\mathcal{S}$. Then there is an efficient algorithm which will return all polynomials $f(X)$ of degree $< k$, so that:

$$f(x_i) = y_i \text{ for at least } t \geq \sqrt{Mk(1+\varepsilon)} \text{ } i's.$$

Moreover, there are at most $r \cdot \frac{M}{k}$ such polynomials.

Before $M = n$. But! It might be useful to have $M > n$ ... for example, if the $x_i$’s are not distinct.

**Def.** A code $C \subseteq \mathbb{F}_q^n$ is $(\frac{n}{k}, k, L)$-LIST-RECOVERABLE if:

For all $S_1, S_2, \ldots, S_n \subseteq \mathbb{F}_q$ with $|S_i| \leq k \forall i,$

$$\{ x \in C \mid x_i \in S_i \text{ for at least } t \text{ values of } i \} \subseteq L.$$

**Picture:**

**Adversary:**

This symbol is 3, 5 or 12

This one is 1 or 2

This one is 6, 7, or 0

This one is 17, 2 or 23.

**You:** Aha! There are not too many codewords that meet all of those constraints ... and here are all of them!

And this one is $\{ \} \subseteq L$
In the context of RS codes, the picture is this:

**ADVERSARY:**

I'm thinking of a low-degree polynomial that goes through only "x" pts.

**YOU:**

There are not too many of those, and here they are!

And, the Guruswami-Sudan algorithm precisely solves this problem!
RS_q(n, k) is \((\frac{t}{n}, \lambda, L)\)-list-recoverable as long as
\[ t = \sqrt{n} \ln \lambda \quad \text{and} \quad L \geq 2(n \cdot \lambda)^{3/2} \sqrt{n}. \]

**Proof:** Replace \( M \) by \( n \cdot \lambda \) and choose \( r = 2n \lambda \ln k \).

**Some Notes About List Recovery:**

1. List recovery is interesting even if \( t = n \).
2. If \( \lambda = 1 \), this is just list-decoding again.
3. We need \( L \geq \lambda \) [why?]
4. The theorem above for RS codes requires \( R \leq \frac{1}{2} \lambda \), since at best \( t = n \), and we’d need \( n > \sqrt{n \ln k} \).
5. That turns out to be tight for RS codes... but we can do better for other codes!

**Fun Exercises:**

- Show that there exist high-rate \((\lambda, L)\)-list-recoverable codes for reasonable \( \lambda \). [HINT: try a random code]
- Show that RS codes of high rate are NOT \((l, L)\)-list-rec [HINT: BCH codes form a big list of codewords whose symbols all live in smaller lists.]
List-Recovery is USEFUL! Today we will see some APPLICATIONS!

**APPLICATION 0:** SUBLINEAR-TIME GROUP-TESTING ALGORITHMS (on your HW)

**APPLICATION 1:** Application to LIST-DECODING.

Consider a code with the following encoding procedure:

To decode, suppose there are a few errors:

This naturally sets up a list-recovery problem for $C$.

We are guaranteed that the good codewords agree w/ a lot of the inner lists b/c of expandiness of the expander.

Thus, this whole thing gives a list-DECODING algorithm.
The **GOOD THING** about this:
- we can tolerate way more error than we could without this expander trick (it's "distance amplification")

The **BAD THING**:
- The rate takes a factor-of-$d$ hit.

But! You can fix that other thing and use this framework to get constant-rate codes that correct a $(1-\varepsilon)$ fraction of errors* in **LINEAR** time [Guruswami-Indyk'03]

Subsequent work has used a similar framework to get rate $R$, list-decodable, up to a $1-R$ fraction of errors.

* Over large alphabets, and the "constant" in "constant rate" is $2^{-2\Theta(1/\varepsilon^3)}$

**APPLICATION 2.  HEAVY HITTERS.**

**PROBLEM.** Given a data stream $x_1, x_2, \ldots, x_m$, where $x_i \in \mathcal{U}$, where $|\mathcal{U}| = N$, find all the $x$ so that $|\{ i \mid x_i = x \}| \geq \varepsilon \cdot m$.

Easy! Store a histogram $(f_1, f_2, \ldots, f_N)$ which counts the # of elements.
Or, just store $x_1, \ldots, x_m$ and do the count on the fly.

**CATCH.** You have limited (say logarithmic) space.

Hard! Actually you need $\Omega(N)$ space to solve this problem.}

"universe"

Such an $x$ is called an "$\varepsilon$-heavy-hitter."
NEW PROBLEM. (Approximate probable heavy hitters)

Given access to a data stream $x_1, \ldots, x_m \in U$, with $|U|=N$, find a set $S \subseteq U$ so that, with probability $99/100$:

- $\forall x$ with $|\{i | x_i = x \}| > \varepsilon m$, $x \in S$
- $|S| \leq \frac{2}{\varepsilon}$

Notice by Markov's inequality, there are at most $\frac{1}{\varepsilon}$ $x$'s so that $|\{i | x_i = x \}| > \varepsilon m$.

So this is allowing us to return a superset of those, with some failure probability.

**THIS IS DO-ABLE!**

Here's a classic solution, called **COUNT-MIN-SKETCH**.

Let $T = O(\log(N))$

**DATA STRUCTURE:**

- Arrays $A_1, \ldots, A_T$, each of length $4/\varepsilon$, initialized to 0.
- Hash functions $h_1, \ldots, h_T$, $h_i: U \rightarrow [4/\varepsilon]$.

**UPDATE:**

- When you see $x \in U$, for each $i=1, \ldots, T$:
  $A_i \left[ h_i(x) \right] += 1$

**QUERY:** Estimate

$\# \text{times } x \text{ appeared} = \min_{i=1, \ldots, T} A_i \left[ h_i(x) \right]$ for more details, or just Google "count min sketch."
Each bucket just stores the count of the # items in it.

**Fun Exercise:** Show that this works with... (If, say, the hash funs are uniformly random, although you don't really need that.)

**SPACE:** $O\left(\frac{T \log(m)}{\varepsilon}\right) = \frac{\log(N) \log(m)}{\varepsilon}$

This is an awesome data structure

But as presented there are 2 things list recovery can help with.

1. Under an additional assm, we can make this **DETERMINISTIC, EXACT**
   
   See Nelson, Nguyen, Woodruff '14 [1] for nonexact deterministic, w/ space $O(\varepsilon N)$

2. Better query time
   
   There are better algs out there, but this one is real cool and uses RS codes.
DETERMINISTIC CONSTRUCTION.

The randomized part is the hash fns, so we'll have to replace those... ...with a Reed-Solomon code!

IDEA. Fix \( \alpha_1, \ldots, \alpha_n \in \mathbb{F}_q \), say \( n=q \).

Let \( k = \epsilon n - 1 \), so that \( RS_q(m, k) \) is \((1, \frac{\epsilon}{e}, L)\)-list-recoverable, for reasonable \( L \).

Set:
- \( U = \{ f \in \mathbb{F}[x] : \deg(f) < k \} \)
- \( p_j(f) = f(\alpha_j) \in \mathbb{F}_q \), for \( j = 1, \ldots, n \).

Same data structure. The recovery algorithm is:

For \( j = 1, \ldots, n \):
- Let \( S_j = \{ \beta : A_j[\beta] \geq 4\epsilon m \} \)
- Run RS list recovery w/ lists \( S_j \)
- Return the results

THM. Assume that the frequency distribution drops off quickly enough:

\[ \sum_{x : \text{freq}(x) < 4\epsilon m} \text{freq}(x) < 3\epsilon m \]

Then this algorithm exactly returns the heavy hitters, with:

- UPDATE TIME: \( \tilde{O}(\log(N)/\epsilon) \)
- QUERY TIME: \( \text{poly}(\log(N)/\epsilon) \)
- SPACE: \( O\left(\frac{\log^2(N) \log(m)}{\epsilon^2}\right) \) bits.
Pt. First, let's see why this works.
For an "item" (aka, polynomial) \( f \), let \( F_f \) denote the frequency of \( f \).

**CASE 1.** Suppose \( F_f \geq \epsilon m \), so \( x \) is an \( \epsilon \)-heavy-hitter.
Then \( f(x_j) \in S_j \) for all \( j = 1, \ldots, n \), so the list-recovery algorithm will return \( f \).

**CASE 2.** Suppose \( F_f < \epsilon m \), so \( f \) is NOT a heavy hitter.

- There are \( \frac{1}{\epsilon} \) actual heavy hitters, \( g_1, \ldots, g_{\frac{1}{\epsilon}} \).
- \( f \) agrees with each of those in \( \leq k \) places.
- Since \( n > \frac{k}{\epsilon} \), there is at least one \( j \) s.t. \( f(x_j) \neq g_j(x_j) \forall i = 1, \ldots, n \).
- That means that there is some \( j \) s.t. \( A_j \supset f(x_j) \) receives no contributions from any of the heavy hitters.

**CLAIM:** If \( A_j \supset \beta \) has no contributions from the heavy hitters, then \( A_j \supset \beta \leq \epsilon m \).

**pf.** Follows from our ASSUMPTION. Even if ALL the non-HH contributed, that's still \( \leq \epsilon m \).

- Thus, \( A_j \supset f(x_j) \leq \epsilon m \), so \( f(x_j) \notin S_j \).
- Then the list-recovery algorithm will NOT return \( f \).

Now let's establish the parameters.

**UPDATE TIME:** Need to compute \( f(x_j) \forall j \), \( O(n) = \tilde{O}\left(\frac{\log(N)}{\epsilon}\right) \)

**QUERY TIME:** (To find all heavy hitters): Run Guruswami-Sudan, \( \text{poly}(n) = \text{poly}\left(\frac{\log(N)}{\epsilon}\right) \)

**SPACE:** \( q \) tables w/ \( q \) buckets each, so \( O\left(q^2 \log(m)\right) = O\left(\frac{\log^2(N) \log(m)}{\epsilon}\right) \).
This approach does not have optimal space, and it requires an additional assumption, but it is:

- deterministic
- exact
- really cute!!

Notice that some sort of assm is necessary to get these w/ \( o(N) \) space.

Back to the randomized, approximate setting.

As presented CMS has a slow recovery algorithm:

- For \( x \in \mathcal{U} \):
  
  \[ \hat{f}_x \text{ is an estimate of } f_x \]

  if \( \hat{f}_x \geq \varepsilon m \), include \( x \) in the heavy hitters list.

Which takes time \( \mathcal{O}(N) \), really not good.

There are better algorithms known:

<table>
<thead>
<tr>
<th></th>
<th>VANILLA CMS (what we saw)</th>
<th>CMS+ &quot;DYADICTRICK&quot; (the classic soln.)</th>
<th>[Larson-Nelson-Nguyen-Thorup ‘16ish] (best I know of)</th>
<th>RS LIST RECOVERY (today)</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPDATE</td>
<td>( \log(N) )</td>
<td>( \log^2(N) )</td>
<td>( \log(N) )</td>
<td>( \tilde{\mathcal{O}}(\log(N)/\varepsilon) )</td>
</tr>
<tr>
<td>QUERY</td>
<td>( N )</td>
<td>( \text{polylog}(N)/\varepsilon )</td>
<td>( \text{polylog}(N)/\varepsilon )</td>
<td>( \text{poly}(\log(N)/\varepsilon) )</td>
</tr>
<tr>
<td>SPACE</td>
<td>( \log(N)/\varepsilon )</td>
<td>( \log^2(N)/\varepsilon )</td>
<td>( \log(N)/\varepsilon )</td>
<td>( \log(N)/\varepsilon^2 )</td>
</tr>
</tbody>
</table>

* big-\( \Omega \)'s suppressed.
Here's the idea. **CAUTION**: we will need to tweak this slightly.

Let $C = RS_q(n, k)$ w/ $k \approx \frac{\varepsilon n}{2}$, so it is $(1, \frac{1}{2}, L)$-list-recoverable with $L = \text{poly}(n)$. Again choose $q \approx n$.

Again let $U = \{ f \in \mathbb{F}_q[x] : \deg(f) < k \}$

**DATA STRUCTURE:**

- Maintain $n$ different COUNT-MIN-SKETCH data structures,
  $CMS_1, CMS_2, ..., CMS_n$,
  which have a universe, $U' = \mathbb{F}_q$, and the same parameter $\varepsilon$.

- Each has:
  
  - **UPDATE**: $O(\log(q))$
  - **QUERY**: $O(q)$
  - **SPACE**: $O \left( \frac{\log(q)}{\varepsilon} + \log(m) \right)$

- So the **SPACE** for my data structure is $O \left( \frac{q \log(q)}{\varepsilon} + O(q \log(m)) \right)$
  
  $= O \left( \log(N)/\varepsilon^2 \right)$

**UPDATE STEP:**

- When $f \in U$ appears:
  
  for $i = 1, ..., n$:
  
  $L \leftarrow CMS_i \cdot \text{UPDATE}(f(x_i))$

- So the **UPDATE TIME** is $O \left( n \left( T(\text{poly evaluation}) + T(\text{CMS update}) \right) \right)$
  
  $\approx O \left( \log(N)/\varepsilon \right)$
QUERY STEP:

Let $S_j = \text{QUERY}(\text{CMS}_j)$ \(\setminus\) the symbols $\beta$ that frequently occurred as $f(x_j)$

Run Guruswami-Sudan on the $S_j$'s and return the output.

Notice that $|S_j| \leq 2\epsilon$, since that's the guarantee of CMS.
Since $\frac{1}{\epsilon} = \frac{\epsilon}{2}$, Guruswami-Sudan applies.

QUERY TIME: $\mathcal{O}(n \cdot q^2) + \text{poly}(n) = \text{poly}(\log(N))$

And finally, why does this work? **CAUTION: IT DOESN'T QUITE WORK YET.**

Here's the picture:

If $f$ is an $\epsilon$-heavy hitter, then $f(x_i)$ is an $\epsilon$-heavy hitter for CMS$_i$, $\forall i$, and so $f(x_i) \in S_i$, $\forall i$, and so Guruswami-Sudan returns $f$ in the output list.

If $f$ is NOT an $\epsilon$-heavy hitter... errm, well, there are poly(n) many such $f$ that end up in the output list! **OOPS! So that doesn't quite work. Fix on next page...**
The fix is to keep one more CMS, this one for the universe $U$:

**UPDATE($f$):**

1. $f_1, f_2, \ldots, f_n$
2. $S_1, S_2, \ldots, S_n$
3. $\{f_1, \ldots, f_n\}$
4. Initialize $L = \emptyset$
5. For $i = 1, \ldots, n$:
   - Use CMS to estimate the frequency of $f_i$
   - If it's $\geq \varepsilon m$, add $f_i$ to $L$
6. Final list $L$

Now, this has output list size $\leq 2/\varepsilon$, because CMS will only say "$\geq \varepsilon m$" for at most $2/\varepsilon$ of the $f_i$'s.

Hooray! That's what we wanted, AND it's really fast!
**APPLICATION:** Identifying attackers in DoS attacks

[Based on Dean-Franklin-Stubblefield '02]

Suppose the set-up is:

![Diagram of routers and traffic]

**Question:** How can you (with the help of the routers), identify the bad guys, and block their packets in the future?

Here's a cute (but grossly over-simplified) version based on RS codes.

Say there are \( n < q \) routers, and each router is addressed by some \( \alpha \in \mathbb{F}_q \).

For example, let's take \( q = 2^{32} \), so that every router has a 32-bit address.

**NAIVE SCHEME:** Every time a router handles a packet, it appends its address.

**This works:**

- I'm getting a lot of traffic from the path \( (\alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_n) \).
- I'm not going to accept any more packets with that path!

**But the downside is that the packets get REALLY big, 32 exhibits for each router they stop at.**
Instead:

- Each packet gets **TWO** field elements appended.
  1. Packet ID - the first router chooses this at random.
  2. Current path ID - this is initialized to 0.

The rule for a router is:

```
(packet-stuff, α, β)    
↓                         
γ ← This router's ID is γ
↓                         
(packet-stuff, α, β+α+γ)
```

That means that what happens to a packet is:

```
(p, α, 0)    
↓             
γ₀           
↓             
(p, α, γ₀+α+γ₀)
```

That is, each path has associated to it a POLYNOMIAL $f(x)$.

At the end of the day, what you see is:

- $(x, f(a))$  $(x, g(a))$  $(\Theta, f(\Theta))(\gamma,f_0(\gamma))$
- $(\psi, g(\psi))(p, f_2(p))(\delta, f_4(\delta))(\gamma, q(\gamma))$  $(\Theta, g(\Theta))$
- $(\beta, g(b))(p, f(p))$  $(\gamma, h(\gamma))(\delta, h(\delta))(\psi, q(\psi))$
- $(\Theta, f_2(\Theta)(\gamma, h(\gamma))(\delta, f_4(\delta))(\gamma, q(\gamma))$
- $(x, h(x))$  $(z, q(z))(\gamma, h(\gamma))(\psi, q(\psi))$

In this example, the paths corresponding to `g` and `h` have a malicious user somewhere upstream.
That is, you are given a bunch of points \((x_i, y_i)\) so that the "BAD" paths correspond to polynomials that pass through many of these points — and you want to find these bad polynomials.

That's what the Guruswami–Sudan algorithm does!

\[
\begin{align*}
(x_0, y_0) & (x_1, y_1) (x_2, y_2) (x_3, y_3) \\
(x_4, y_4) & (x_5, y_5) (x_6, y_6) (x_7, y_7) (x_8, y_8) \\
(x_9, y_9) & (x_{10}, y_{10}) (x_{11}, y_{11}) (x_{12}, y_{12}) \\
(x_{13}, y_{13}) & (x_{14}, y_{14}) (x_{15}, y_{15}) (x_{16}, y_{16}) \\
(x_{17}, y_{17}) & (x_{18}, y_{18}) (x_{19}, y_{19}) (x_{20}, y_{20})
\end{align*}
\]

\[
\exists g(x), h(x), f_{17}(x) \}
\]

I'm not going to allow any path that ended up going through \(g, h, \) or \(f_{17} \)!

Sure, there will be some false positives, but that's OK.

That's it!

**QUESTION TO PONDER**

What can list-recovery do for you???