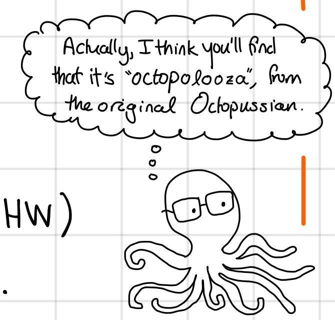
CS 250 / EE 387 - Lecture 12 - List Recovery

Agenda

1. List-Recovery
2. Applications!
   - Sublinear-time Group Testing (HW)
   - Applications to list-DECODING...
   - Heavy hitters, two ways.
   - Cute approach to IP-traceback.



Today's Octopus Fact

"Octopus" comes from the Greek ὀκτώ (eight) and πούς (foot), and the Greek plural would be "octopodes." The traditionally accepted English plural is "octopuses." It's tempting to pluralize it as "octopi", but that is not generally accepted since "octopus" in Latin – since it came from the Greek – actually isn't a second-declension noun.
All this has led to an amazing example of being stuck up about grammar while simultaneously shaming people who are too stuck up about grammar: Fowler's Modern English Usage says that the only acceptable plural is "octopuses," and that "octopi" is misconceived, while "octopodes" is pedantic.

Recall that last time we proved:

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

We did this via the following algorithm:

Guruswami-Sudan Algorithm.

Choose a parameter r
Suppose $t \geq \sqrt{kn(1+1/r)}$

1. Interpolation Step.
   Find a polynomial $Q(X,Y)$ with $(1,k)$-degree $D = \sqrt{kn \cdot r \cdot (r+1)}$ so that $Q(\alpha_i, y_i) = 0$ with multiplicity r for $i = 1, \ldots, n$.

2. Root-Finding Step.
   Return all f so that $Q(X, f(X)) \equiv 0$.
   [Notice that there are $\leq \deg_Y(Q) \leq D/k \approx r/\sqrt{R}$ of these.]

OBSERVATION: There is no reason that the α_i's need to be distinct.

What we actually proved was:

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

$$f(\alpha_i) = y_i \quad \text{for at least} \quad t \geq \sqrt{Mk(1 + 1/r)} \quad i\text{'s}.$$

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

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

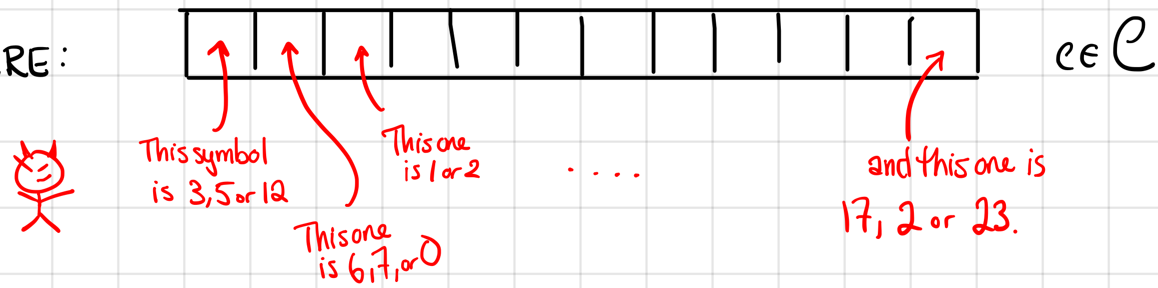
DEF. A code $\mathcal{C} \subseteq \mathbb{F}_q^n$ is $\left(\frac{t}{n}, \ell, L\right)$-LIST-RECOVERABLE if:

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

$$\left|\{ c \in \mathcal{C} \mid c_i \in S_i \text{ for at least } t \text{ values of } i \}\right| \leq L.$$

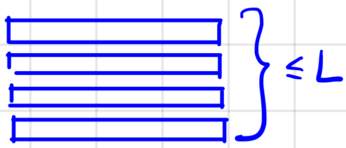
PICTURE:

ADVERSARY:



You:

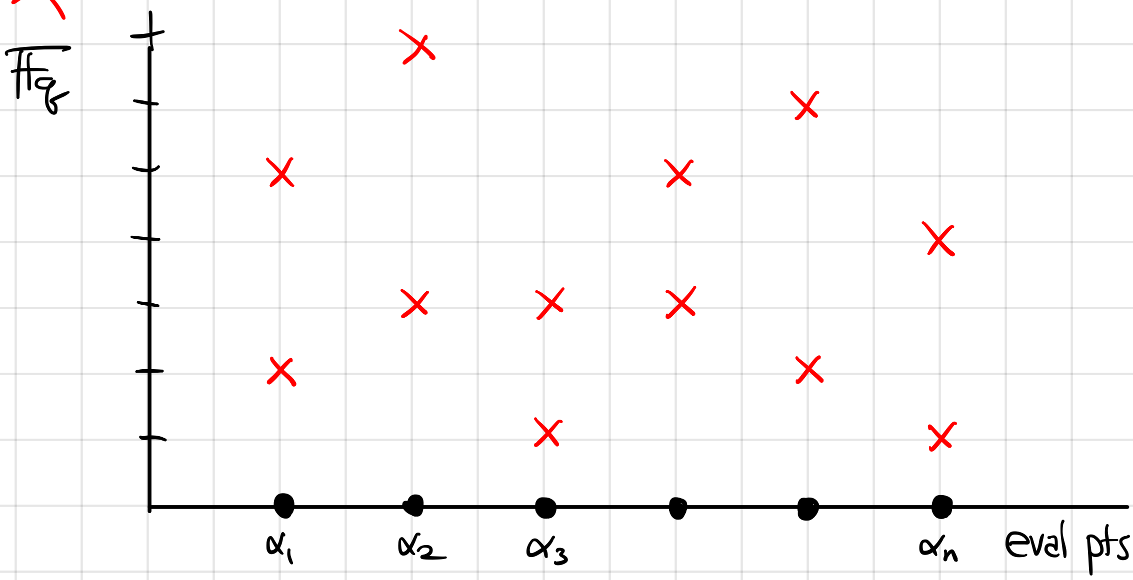
Aha! There are not too many codewords that meet all of those constraints ... and here are all of them!



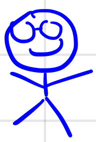
In the context of RS codes, the picture is this:

ADVERSARY:

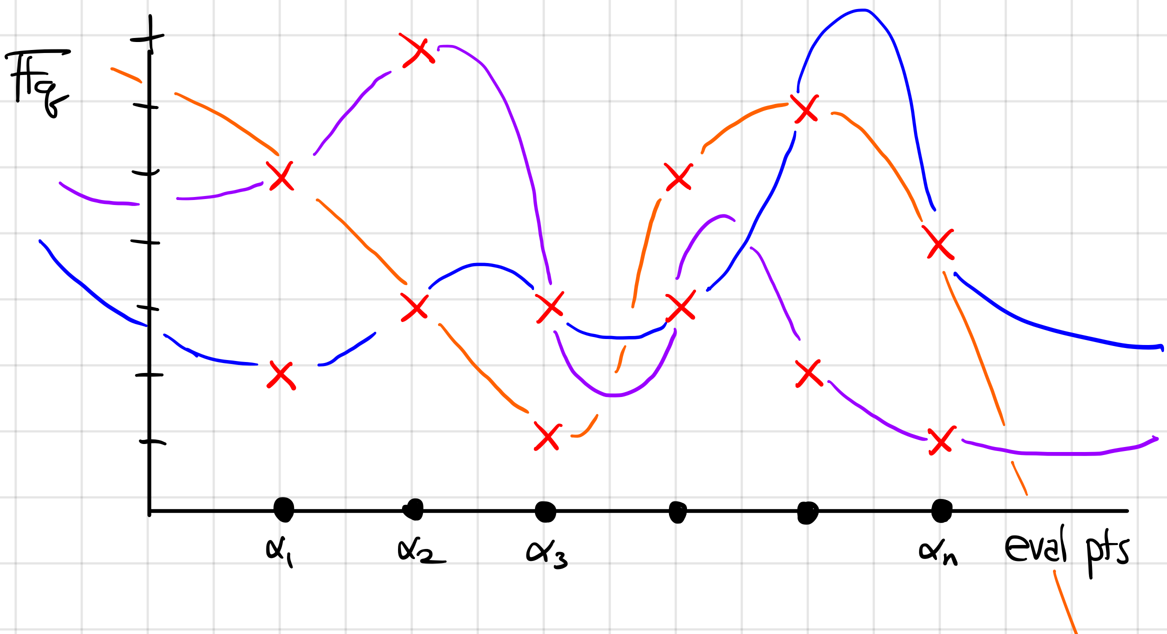
I'm thinking of a low-degree polynomial that goes through only "x" pts.



YOU:



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



And, the Guruswami-Sudan algorithm precisely solves this problem!

COR. $RS_q(n,k)$ is $\left(\frac{t}{n}, \ell, L\right)$-list-recoverable as long as

$$t > \sqrt{\ell n k} \quad \text{and} \quad L \geq 2(n \cdot \ell)^{3/2}\sqrt{k}.$$

Proof: Replace M by $n \cdot \ell$ and choose $r = 2n\ell k$.

and maybe use the fact that t is an integer...

Some Notes About List Recovery:

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

Fun Exercises:

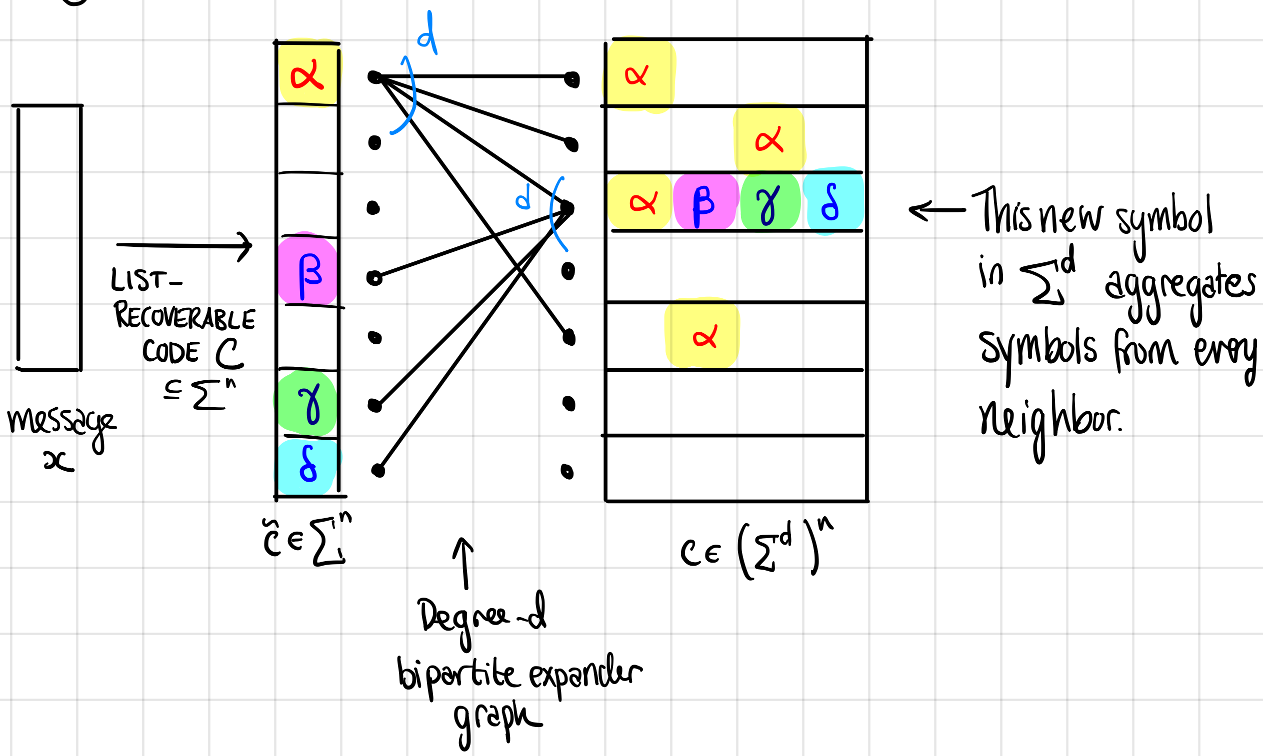
- Show that there exist high-rate (ℓ, L)-list-recoverable codes for reasonable ℓ. [HINT: try a random code]
- Show that RS codes of high rate are NOT (ℓ, L)-list-rec. [HINT: BCH codes form a big list of codewords whose symbols all live in smaller lists.]

② List-Recovery is USEFUL! Today we will see some sketches of APPLICATIONS!

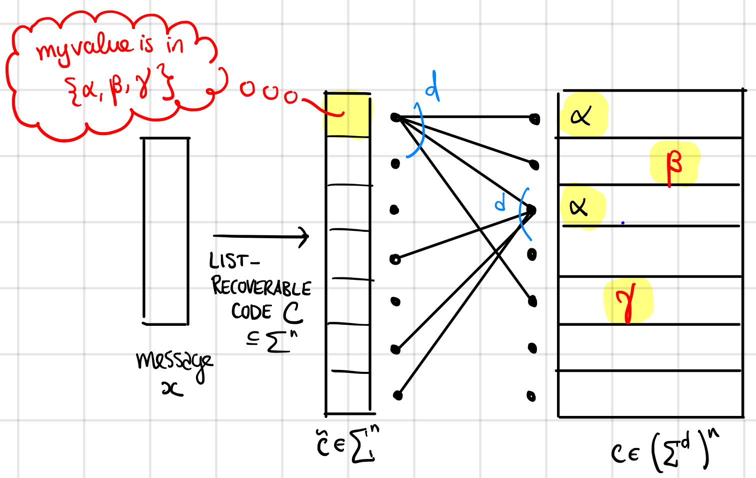
APPLICATION 0: SUBLINEAR-TIME GROUP-TESTING ALGORITHMS (on your HW)

APPLICATION 1: APPLICATION to LIST-DECODING. [Extremely sketchy - See Guruswami-Indyk, "LINEAR TIME ENC. & LIST-DECODABLE CODES" STOC 2003 for more details]

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]

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

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

APPLICATION 2. HEAVY HITTERS.

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

"universe"

Such an x is called an "ε-heavy-hitter."

Easy! Store a histogram $(f_1, f_2, \dots, f_N)$ which counts the # of elements.
Or, just store $x_1, \dots, x_m$ 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 :(.

NEW PROBLEM. (Approximate probable heavy hitters)

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

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

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

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.

We will just give a sketch of the sketch here. See the original paper by Cormode + Muthukrishnan for more details... or just Google "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 $h_1, \ldots, h_T$, $h_i : \mathcal{U} \to [4/\varepsilon]$.

UPDATE:

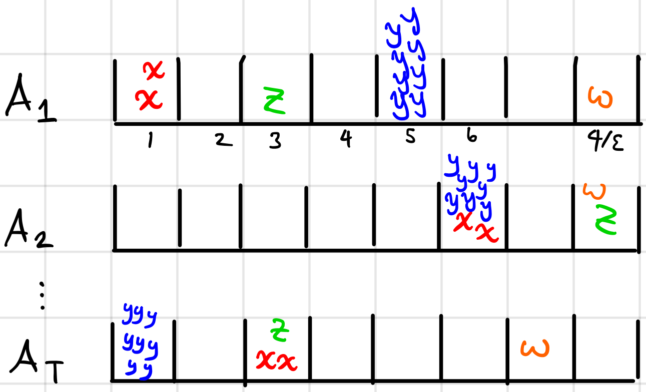
- When you see $x \in \mathcal{U}$, for each $i = 1, \ldots, T$:
  $A_i[h_i(x)] \mathrel{+}= 1$

QUERY: Estimate

$$\#\text{times } x \text{ appeared} = \min_{i=1,\ldots,T} A_i[h_i(x)]$$

and return all the things with big estimates.

Picture looks like this: x y y z y y y w x y y y ....



$h_1(x)$ = bucket 1
$h_1(y)$ = bucket 5
etc...

Each bucket just stores the count of the # items in it.

Now, HOPEFULLY, each heavy hitter is "reasonably" isolated in at least one bucket (in that no other heavy hitter lands there too), and then the min is a good bet.

FUN EXERCISE: Show that this works whp. (If, say, the hash fns are uniformly random, although you don't really need that.)

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

T arrays w/ $O(1/\varepsilon)$ buckets each, and each bucket holds an int in $[m]$

NOTE 1. Cheating b/c we also need to store the h_i's, but it turns out that's OK.

NOTE 2. Can improve this to $\frac{\lg(N)}{\varepsilon} + \lg(m)$.

THIS IS AN AWESOME DATA STRUCTURE ← It is literally my FAVORITE data structure/randomized alg.

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

(2A) Under an additional assm, we can make this DETERMINISTIC + EXACT.

See [Nelson, Nguyễn, Woodruff '14] for nonexact deterministic, w/ space $O_\varepsilon(\lg N)$

(2B) Better query time

There are better algs out there, but this one is real cute and uses RS codes.

(2A) 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 $n=q$.

agreement $t=n$ $\ell = 1/\varepsilon$ is the size of the "inner" lists.

Let $k = \varepsilon n - 1$, so that $RS_q(n,k)$ is $(1, 1/\varepsilon, L)$-list-recoverable, for reasonable L

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

PARAMETERS
- $N = q^k$ = size of $\mathcal{U}$
- q = # buckets
- n = "T" = # tables

Same data structure. The recovery algorithm is:

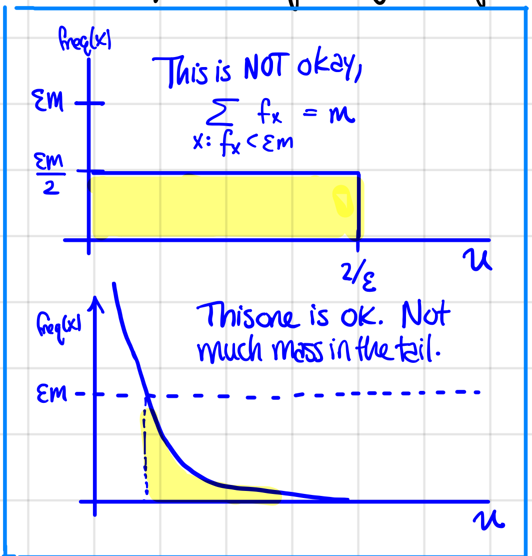
$$\boxed{\begin{array}{l} \text{For } j = 1, \ldots, n: \\ \quad \text{Let } S_j = \{ \beta : A_j[\beta] \geq \varepsilon m \} \\ \text{Run RS list recovery w/ lists } S_j \\ \text{Return the results} \end{array}}$$

In RS lingo,
$$S_j = \left\{ \beta : \begin{array}{l} \text{there are at least } \varepsilon m \\ \text{items } f \text{ in the stream} \\ \text{that } f(\alpha_j) = \beta \end{array} \right\}$$

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

$$\sum_{x : \, freq(x) < \varepsilon m} freq(x) < \varepsilon m$$

for example



Examples not part of Thm statement

Then this algorithm exactly returns the heavy hitters, with:

UPDATE TIME: $\tilde{O}(\log(N)/\varepsilon)$

QUERY TIME: $\mathrm{poly}(\log(N)/\varepsilon)$

SPACE: $O\left(\frac{\log^2(N) \log(m)}{\varepsilon^2} \right)$ bits.

Pf. First, let's see why this works.
For an "item" (aka, polynomial) f, let F_f denote the frequency of f.

CASE 1. Suppose $F_f \geq \varepsilon m$, so x is an ε-heavy-hitter.
Then $f(\alpha_j) \in S_j$ for all $j = 1, \dots, n$, so the list-recovery algorithm WILL return f.

CASE 2. Suppose $F_f < \varepsilon m$, so f is NOT a heavy hitter.

- There are $\leq 1/\varepsilon$ ACTUAL heavy hitters, $g_1, \dots, g_{1/\varepsilon}$.
- f agrees with each of those in $\leq k$ places
- Since $n > k/\varepsilon$, there is at least one j s.t. $f(\alpha_j) \neq g_i(\alpha_j)$ $\forall i = 1, \dots, 1/\varepsilon$.
- That means that there is some j s.t. $A_j[f(\alpha_j)]$ receives no contributions from any of the heavy hitters.

- **CLAIM:** If $A_j[\beta]$ has no contributions from the heavy hitters, then $A_j[\beta] \leq \varepsilon m$.

pf. Follows from our ASSUMPTION. Even if ALL the non-HH contributed, that's still $\leq \varepsilon m$.

- Thus, $A_j[f(\alpha_j)] \leq \varepsilon m$, so $f(\alpha_j) \notin S_j$
- Then the list recovery algorithm will NOT return f.

Now let's establish the parameters.

note $q^k = N$, $k \approx \varepsilon n$, so $n = O(\log(N)/\varepsilon)$.

UPDATE TIME: Need to compute $f(\alpha_j)$ $\forall j$, $\tilde{O}(n) = \tilde{O}\left(\log(N)/\varepsilon\right)$

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

SPACE: q tables w/ q buckets each, so $O(q^2 \log(m)) = O\left(\frac{\log^2(N)\log(m)}{\varepsilon^2}\right)$.

This approach does not have optimal space, and it requires an additional assm,
BUT it is:

- deterministic
- exact
- really cute!!

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

(2B) Back to the randomized, approximate setting.

As presented CMS has a slow recovery algorithm:

- For $x \in \mathcal{U}$:
  - Estimate f_x as $\hat{f}_x$
  - If $\hat{f}_x \geq \varepsilon m$, include x in the heavy hitters list.

which takes time $O(N)$, really not good.

There are better algs known:

	VANILLA CMS (what we saw)	CMS + "DYADIC TRICK" (the classic soln.)	[Larsen-Nelson-Nguyễn-Thorup '16ish] (best I know of)	RS LIST RECOVERY (today)
UPDATE	$\log(N)$	$\log^2(N)$	$\log(N)$	$\tilde{O}(\log(N)/\epsilon)$
QUERY	N	$\mathrm{polylog}(N)/\epsilon$	$\mathrm{polylog}(N)/\varepsilon$	$\mathrm{poly}(\log(N)/\epsilon)$
SPACE	$\log(N)/\varepsilon$	$\log^2(N)/\epsilon$	$\log(N)/\epsilon$	$\log(N)/\varepsilon^2$

* big-Oh's suppressed

What we'll see today isn't the best known, but it's competitive and very cute!

Here's the idea. CAUTION: we will need to tweak this slightly.

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

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

DATA STRUCTURE:

- Maintain n different COUNT-MIN-SKETCH data structures,

  $CMS_1, CMS_2, \ldots, CMS_n$,

  which have a universe $\mathcal{U}' = \mathbb{F}_q$, and the same parameter ε.

- Each has:

  UPDATE : $O(\log(q))$

  QUERY : $O(q)$

  SPACE : $O\left(\frac{\log(q)}{\epsilon} + \lg(m)\right)$

- So the **SPACE** for my data structure is $O\left(\frac{q \log(q)}{\epsilon} + O(q \log(m))\right)$

  $= O(\log(N)/\varepsilon^2)$

UPDATE STEP:

- When $f \in \mathcal{U}$ appears:

  for $i = 1, \ldots, n$:

  $\quad$ $CMS_i.\mathrm{UPDATE}(f(\alpha_i))$

- So the **UPDATE TIME** is $O\left(n\left(T(\text{poly evaluation}) + T(\text{CMS update})\right)\right)$

  $= \tilde{O}(\log(N)/\epsilon)$

QUERY STEP:

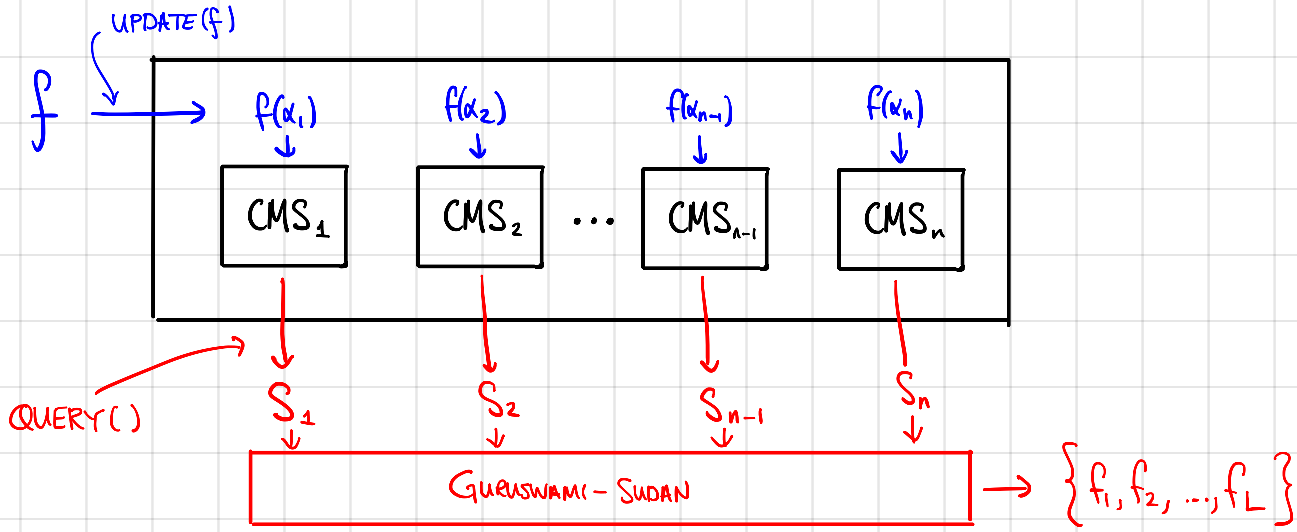
Let S_j = QUERY(CMS_j) \\ the symbols β that frequently occured as $f(\alpha_j)$

Run Guruswami-Sudan on the S_j's, and return the output. \\ find all the f's that might have been responsible for those β's.

Notice that $|S_j| \le 2/\varepsilon$, since that's the guarantee of CMS.
Since $\frac{k}{n} = \varepsilon/2$, Guruswami-Sudan applies.

QUERY TIME: $O(n \cdot q) + \mathrm{poly}(n) = \mathrm{poly}\left(\frac{\log(N)}{\varepsilon}\right)$

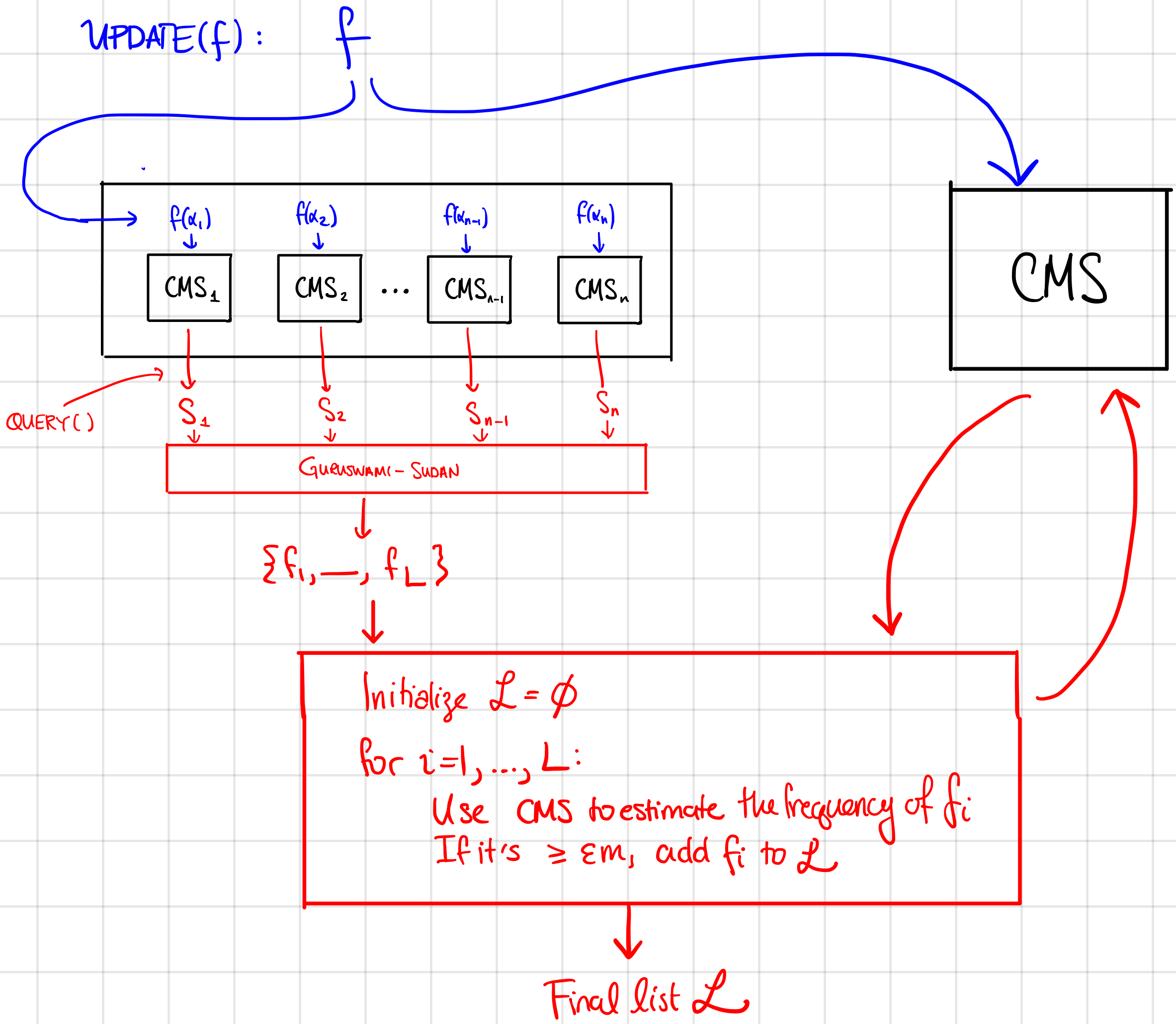
And finally, why does this work? CAUTION: IT DOESN'T QUITE WORK YET.
Here's the picture:



If f is an ε-heavy hitter, then $f(\alpha_i)$ is an ε-heavy-hitter for CMS_i, $\forall i$, and so $f(\alpha_i) \in S_i$ $\forall i$, and so GURUSWAMI-SUDAN returns f in the output list.

If f is **NOT** an ε-heavy hitter... erm, 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 $\mathcal{U}$:

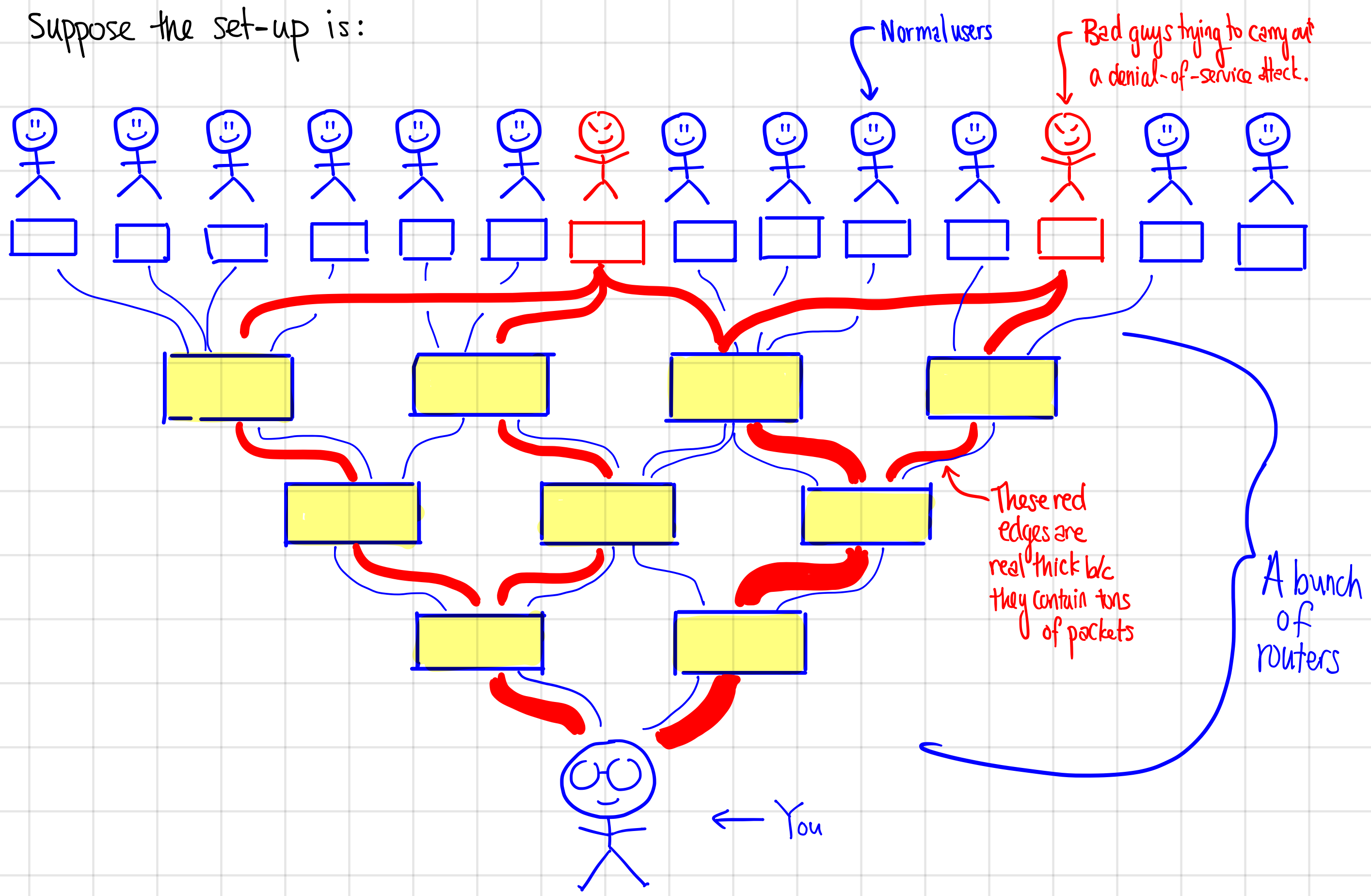


Now, this has output list size $\leq 2/\varepsilon$, because CMS will only say "$\geq \varepsilon m$" for at most $2/\varepsilon$ of the f_i's.

HOORAY! That's what we wanted, AND it's really fast!

(2C) APPLICATION: Identifying attackers in DoS attacks [Based on Dean-Franklin-Stubblefield '02]

Suppose the set-up is:



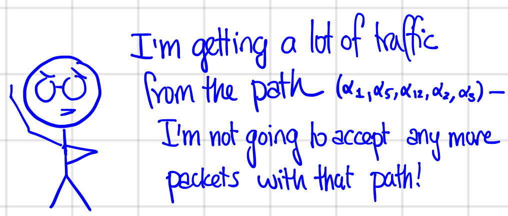
Question: How can you (with the help of the routers), identify the bad guys, and block their packets in the future?

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 it's address.

This works:



But the downside is that the packets get REALLY big, 32 extra bits 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.

*How does a router know it is first? It doesn't. One way to get around this is to randomize – a router just guesses that it is first with some small probability. This can be made to work.

- The rule for a router is:

$$(\text{packet-stuff}, \underbrace{\alpha}_{\text{Packet ID}}, \underbrace{\beta}_{\text{Path}})$$
$$\downarrow$$
$$\boxed{\gamma} \leftarrow \text{This router's ID is } \gamma$$
$$\downarrow$$
$$(\text{packet-stuff}, \alpha, \beta\cdot\alpha+\gamma)$$

That means that what happens to a packet is:

$$(p, \alpha, 0)$$
$$\downarrow$$
$$\boxed{\gamma_2}$$
$$\downarrow \quad (p, \alpha, \gamma_2)$$
$$\boxed{\gamma_1}$$
$$\downarrow \quad (p, \alpha, \gamma_2\cdot\alpha+\gamma_1)$$
$$\boxed{\gamma_0}$$
$$\downarrow \quad (p, \alpha, \gamma_2\cdot\alpha^2+\gamma_1\cdot\alpha+\gamma_0)$$

$\circ\circ\circ\ (p, \alpha, f(\alpha))$

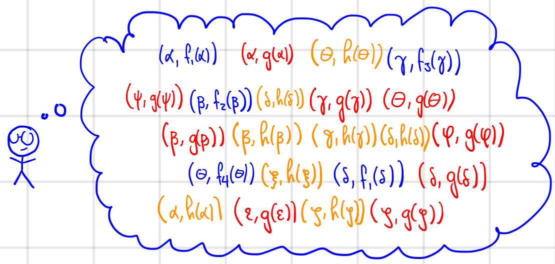
That is, each path has associated to it a POLYNOMIAL $f(X)$.
At the end of the day, what you see is:

$(\alpha, f_1(\alpha))$ $(\alpha, g(\alpha))$ $(\theta, h(\theta))$ $(\gamma, f_3(\gamma))$
$(\psi, g(\psi))$ $(\beta, f_2(\beta))$ $(\delta, h(\delta))$ $(\gamma, g(\gamma))$ $(\theta, g(\theta))$
$(\beta, g(\beta))$ $(\beta, h(\beta))$ $(\gamma, h(\gamma))$ $(\delta, h(\delta))$ $(\varphi, g(\varphi))$
$(\theta, f_4(\theta))$ $(\xi, h(\xi))$ $(\delta, f_1(\delta))$ $(\delta, g(\delta))$
$(\alpha, h(\alpha))$ $(\varepsilon, g(\varepsilon))$ $(\zeta, h(\zeta))$ $(\zeta, g(\zeta))$

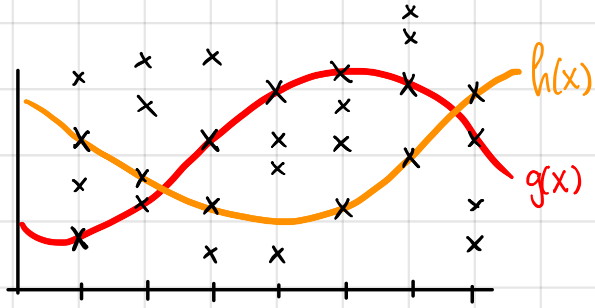
$\leftarrow$ In this example, the paths corresponding to g and h have a malicious user somewhere upstream.

That is, you are given a bunch of points (α_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!

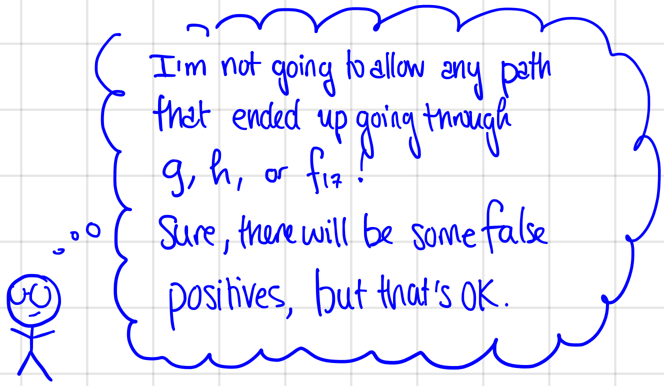


$\longrightarrow$ drawn differently



$\downarrow$ Guruswami - Sudan

$$\{ g(X), h(X), f_{17}(X) \}$$



That's it!

QUESTION TO PONDER

What can list-recovery do for you???