

# 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 ANT FACT

The collective noun for ants is a "swarm."

It's not fair! There are only two of us. Why can't we be something cool, like a "ostentation of peacocks" or a "parliament of owls"?

Yeah! I want to be a "BATCAVE of ANTS"!

I guess at least "swarm" is more flattering than a "host of hippies" though...



Recall that last time we proved:

THM. For all  $r > 0$ , RS codes of rate  $R$  are  $(1 - \sqrt{R(1+1/r)}, r/\sqrt{R})$  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.

Choose a parameter  $r$

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

#### 1. INTERPOLATION STEP.

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

#### 2. ROOT-FINDING STEP.

Return all  $f$  so that  $Q(X, f(X)) = 0$ .

[Notice that there are  $\leq \deg_Y(Q) \leq D/r \approx r/\sqrt{R}$  of these.]

OBSERVATION: There is no reason that the  $\alpha_i$ 's need to be distinct.

What we actually proved was:

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

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

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  $\alpha_i$ 's are not distinct.

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

for all  $S_1, S_2, \dots, S_n \subseteq \mathbb{F}_q$  with  $|S_i| \leq l \forall i$ ,  
 $|\{c \in \mathcal{C} \mid c_i \in S_i \text{ for at least } t \text{ values of } i\}| \leq L$ .

PICTURE:

ADVERSARY:



This symbol  
is 3, 5 or 12

This one  
is 1 or 2

This one  
is 6, 7, or 0

...

and 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!



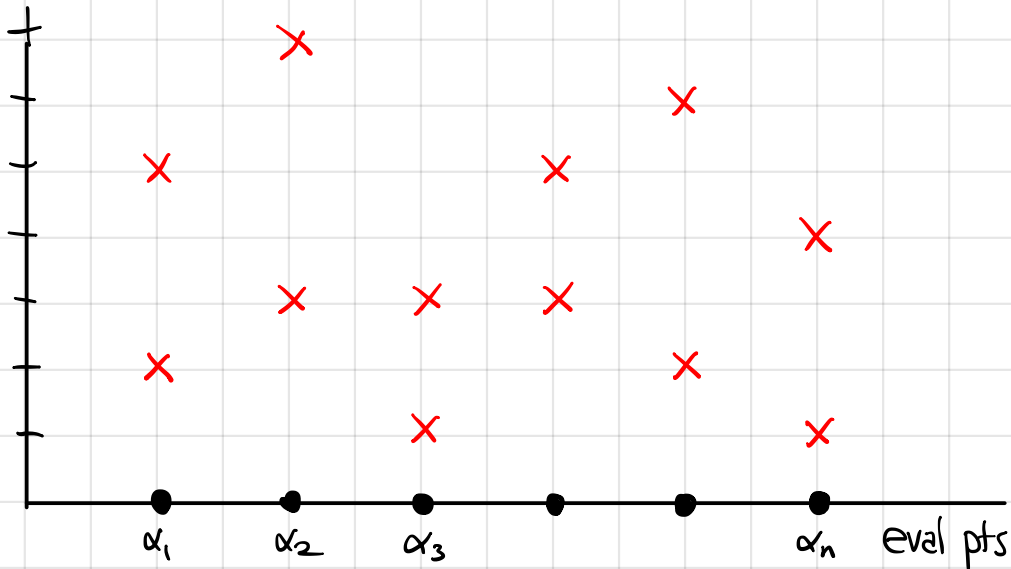
In the context of RS codes, the picture is this:

ADVERSARY:



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

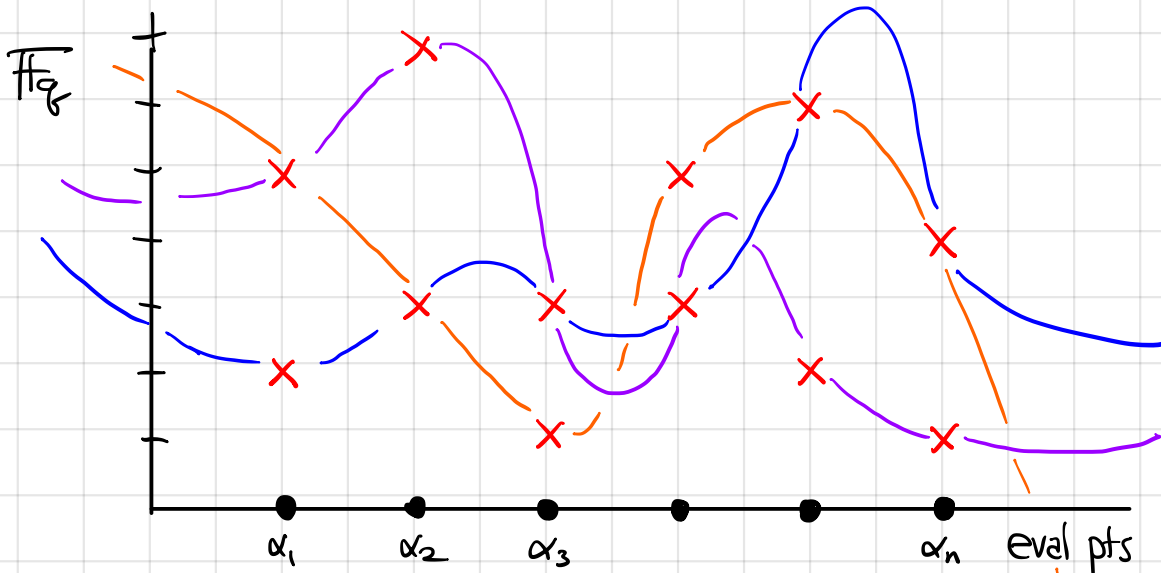
$\mathbb{F}_q$



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  $(\frac{t}{n}, \ell, L)$ -list-recoverable as long as  
 $t > \sqrt{\ell n k}$  and  $L \geq 2(n \cdot \ell)^{3/2} \sqrt{k}$ .

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



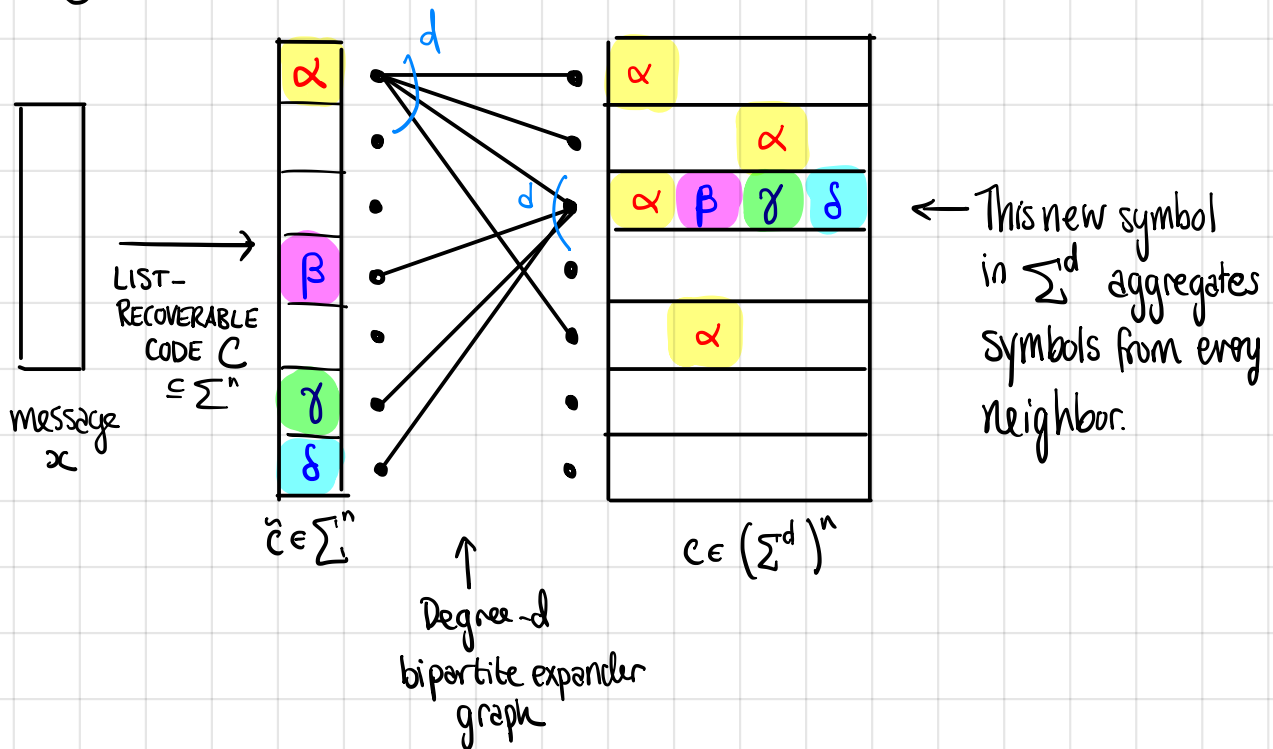
## ② List-Recovery is USEFUL! Today we will see some APPLICATIONS!

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

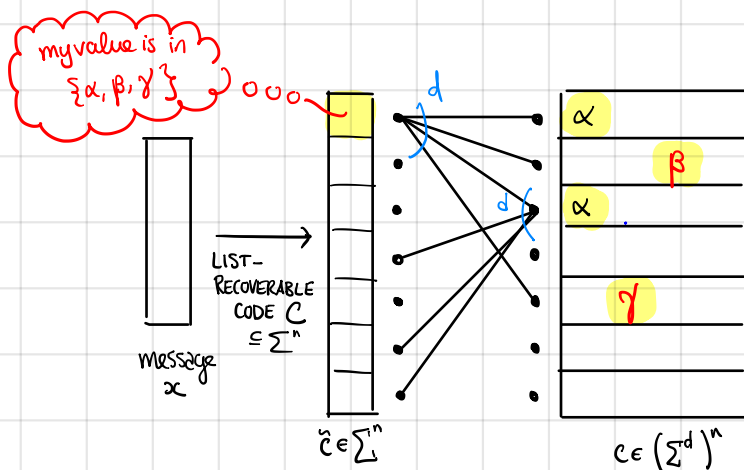
APPLICATION 1: APPLICATION to LIST-DECODING.

[Extremely sketchy - See Guruswami-Indyk, "LINEAR TIME ENC. of 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]

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  $2^{-2^{O(1/\varepsilon^3)}}$

## 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  $|\{i \mid x_i = x\}| \geq \varepsilon \cdot m$ .

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 ".

Such an  $x$  is called an " $\varepsilon$ -heavy-hitter."

## NEW PROBLEM. (Approximate probable heavy hitters)

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

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

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

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  $A_1, \dots, A_T$ , each of length  $4/\epsilon$ , initialized to 0.
- Hash functions  $h_1, \dots, h_T$ ,  $h_i: \mathcal{U} \rightarrow [4/\epsilon]$ .

UPDATE:

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

QUERY: Estimate

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

and return all the things with big estimates.

Picture looks like this:

x y y z yyy w x yyy ...

$A_1$	x		z		yyy		w
	1	2	3	4	5	6	$4/\epsilon$
$A_2$					yyy	x	z
$\vdots$							
$A_T$	yyy		z				w
	yyy		xx				
	yy						

$h_1(x) = \text{bucket 1}$   
 $h_1(y) = \text{bucket 5}$   
 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 w.h.p.

(If, say, the hash fns are uniformly random, although you don't really need that).

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

$T$  arrays w/  $O(1/\epsilon)$  buckets each, and each bucket holds an int in  $\mathbb{Z}_m$

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  $\frac{\log(N)}{\epsilon} + \log(m)$ .

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 asssm, we can make this DETERMINISTIC+EXACT.

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

(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  $\alpha_1, \dots, \alpha_n \in \mathbb{F}_q$ , say  $n=q$ .  
Let  $k = \epsilon n - 1$ , so that  $RS_q(n, k)$  is  $(1, 1/\epsilon, L)$ -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  $j=1, \dots, n$ .

PARAMETERS  $\left\{ \begin{array}{l} N = q^k = \text{size of } \mathcal{U} \\ q = \# \text{ buckets} \\ n = "T" = \# \text{ tables} \end{array} \right.$

Same data structure. The recovery algorithm is:

For  $j=1, \dots, n$ :

Let  $S_j = \{p : A_j[p] \geq \epsilon m\}$

Run RS list recovery w/ lists  $S_j$

Return the results

In RS lingo,

$S_j = \{p : \text{there are at least } \epsilon m \text{ items } f \text{ in the stream that } f(\alpha_j) = p\}$

THM.

ASSUME that the frequency distribution drops off quickly enough:

$$\sum_{x: \text{freq}(x) < \epsilon m} \text{freq}(x) < \epsilon m$$

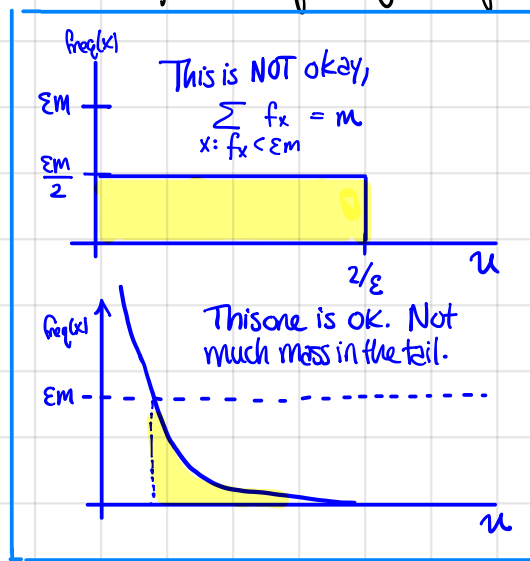
for example  $\rightarrow$

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.




Examples not part of Thm statement  $\rightarrow$

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  $F_f \geq \epsilon m$ , so  $f$  is an  $\epsilon$ -heavy-hitter.


Then  $f(x_j) \in S_j$  for all  $j=1, \dots, n$ , so the list-recovery algorithm WILL return  $f$ . 

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

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

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

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

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

Now let's establish the parameters.

*note*  $q^k = N$ ,  $k \geq \epsilon n$ , so  $n = O(\log(N)/\epsilon)$ .

UPDATE TIME: Need to compute  $f(x_j) \forall j$ ,  $\tilde{O}(n) = \tilde{O}(\log(N)/\epsilon)$

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  $O(q^2 \log(m)) = O\left(\frac{\log^2(N) \log(m)}{\epsilon^2}\right)$ .

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 assumption is necessary to get these w/  $o(N)$  space.

(23) 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  $\hat{f}_x \geq \epsilon m$ , 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 + "DIADICTRICK" (the classic soln.)	[Larsen-Nelson- Nguyễn-Thompson '16]	RS LIST RECOVERY (today)
UPDATE	$\log(N)$	$\log^2(N)$	$\log(N)$	$\tilde{O}(\log(N)/\epsilon)$
QUERY	$N$	$\text{poly}(\log(N)/\epsilon)$	$\text{poly}(\log(N)/\epsilon)$	$\text{poly}(\log(N)/\epsilon)$
SPACE	$\log(N)/\epsilon$	$\log^2(N)/\epsilon$	$\log(N)/\epsilon$	$\log(N)/\epsilon^2$

\* big-Oh's suppressed

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

Here's the idea.

CAUTION: we will need to tweak this slightly.

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

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

### DATA STRUCTURE:

- Maintain  $n$  different COUNT-MIN-SKETCH data structures,  $\text{CMS}_1, \text{CMS}_2, \dots, \text{CMS}_n$ , which have a universe  $\mathcal{U}' = \mathbb{F}_q$ , 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  $O\left(\frac{q \log(q)}{\varepsilon} + O(q \log(m))\right) = O(\log(N)/\varepsilon^2)$

### UPDATE STEP:

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

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

$\text{CMS}_i \cdot \text{UPDATE}(f(x_i))$

- So the UPDATE TIME is  $O\left(n \left( T(\text{poly evaluation}) + T(\text{CMS update}) \right)\right) = \tilde{O}(\log(N)/\varepsilon)$



## QUERY STEP:

Let  $S_j = \text{QUERY}(\text{CMS}_j)$

// the symbols  $\beta$  that frequently occurred as  $f(x_j)$

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

// find all the  $f$ 's that might have been responsible for those  $\beta$ 's.

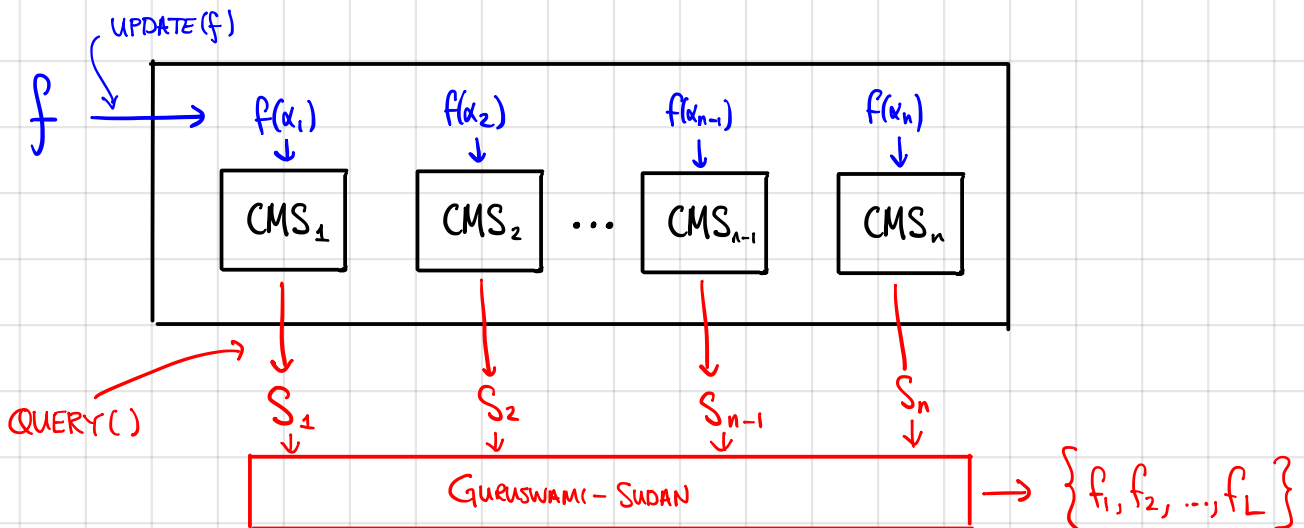
Notice that  $|S_j| \leq 2/\epsilon$ , since that's the guarantee of CMS.

Since  $\frac{k}{n} = \epsilon/2$ , Guruswami-Sudan applies.

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

And finally, why does this work?  
Here's the picture:

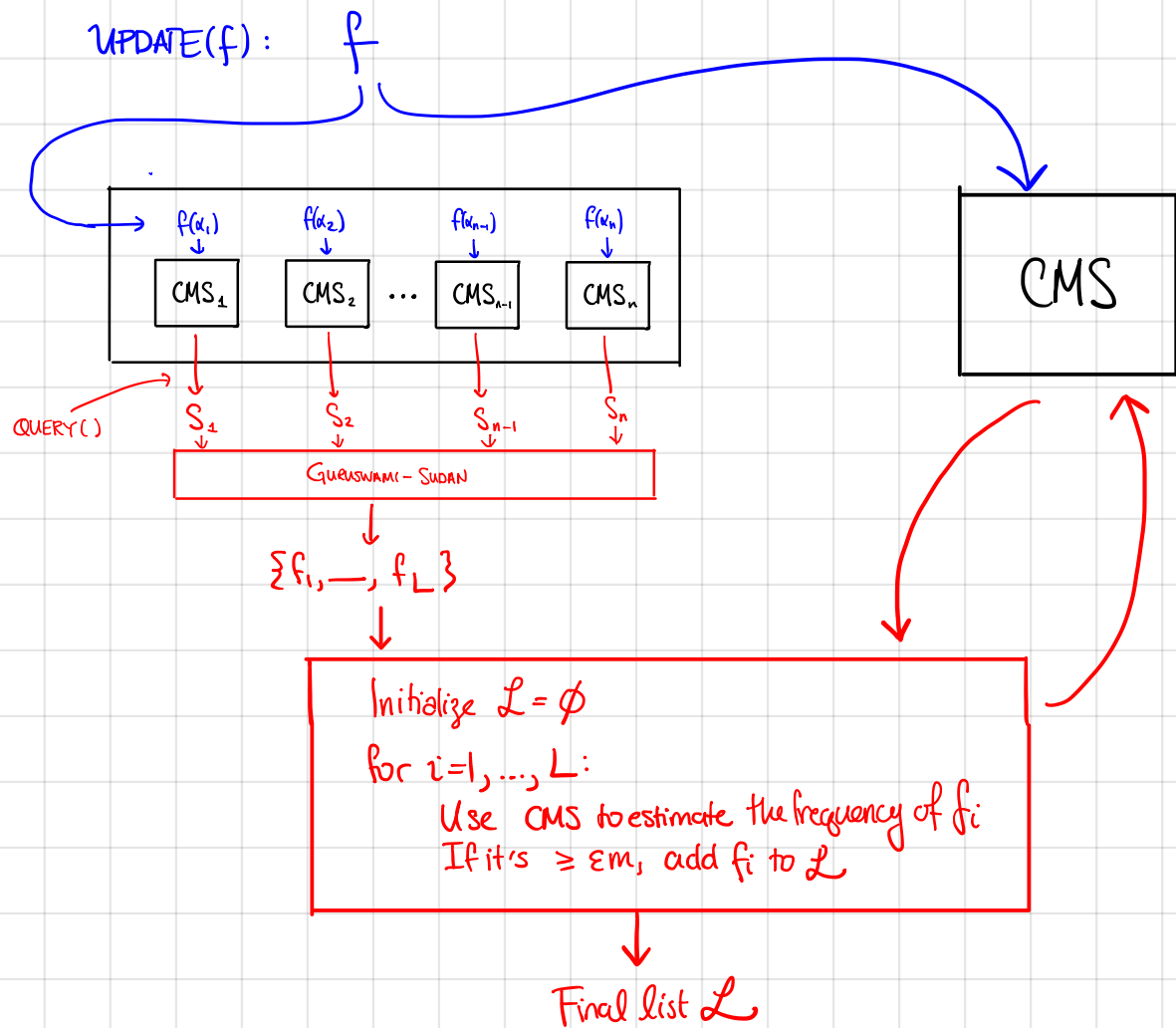
CAUTION: IT DOESN'T QUITE WORK YET.



If  $f$  is an  $\epsilon$ -heavy hitter, then  $f(x_i)$  is an  $\epsilon$ -heavy-hitter for  $\text{CMS}_i$ ,  $\forall i$ , 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... erm, well, there are  $\text{poly}(n)$  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  $\leq 2/\epsilon$ , because CMS will only say " $\geq \epsilon m$ " for at most  $2/\epsilon$  of the  $f_i$ 's.

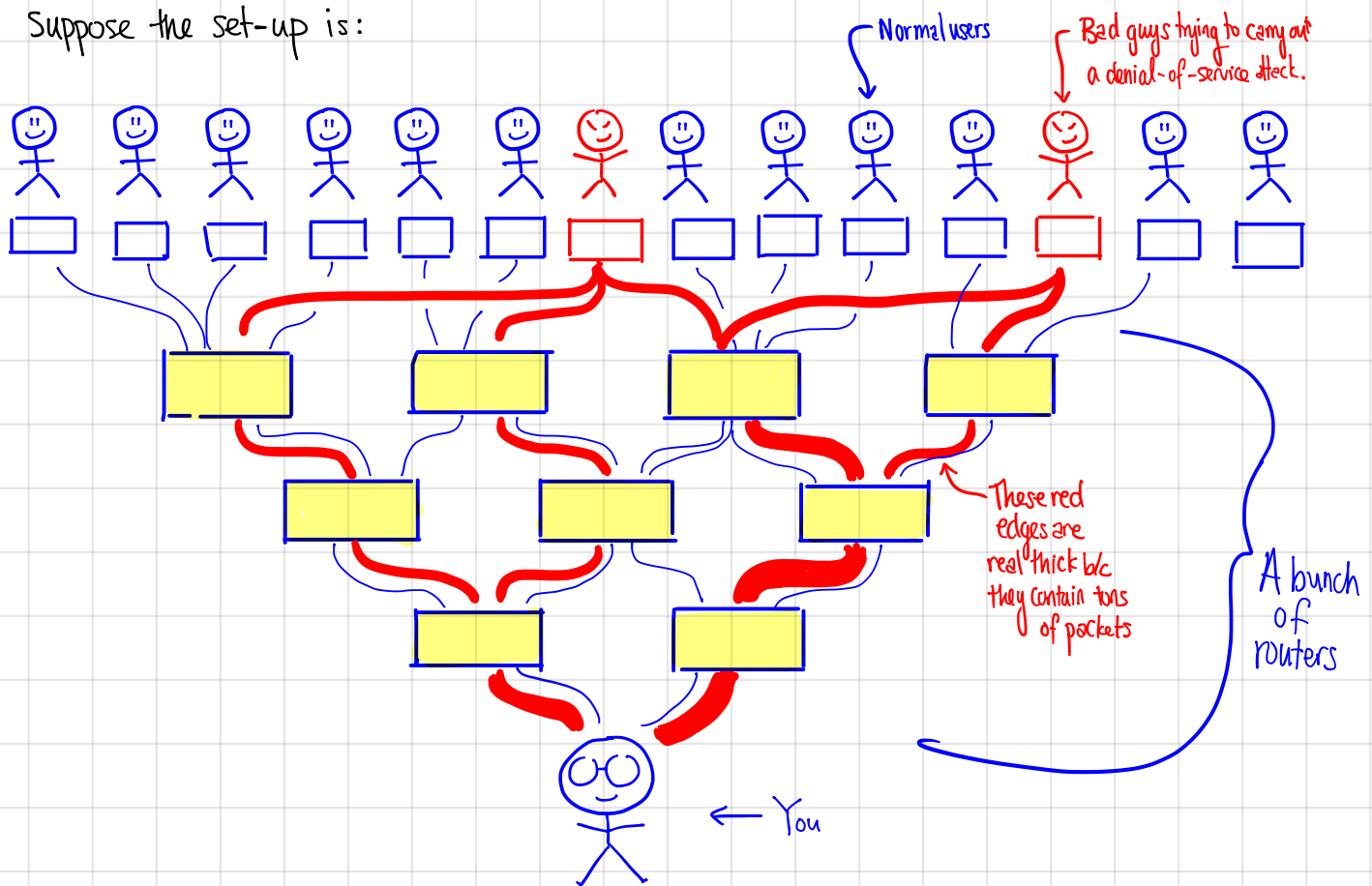
**HOORAY!** That's what we wanted, AND it's really fast!

---

## (2C) APPLICATION: Identifying attackers in DoS attacks

[Based on Dean-Franklin-Stubblefield '02]

Suppose the set-up is:



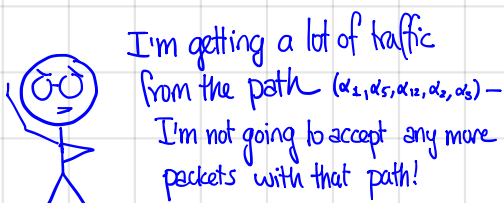
Question: How can you (with the help of the routers), identify the bad guys, and block their packets in the future?

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.

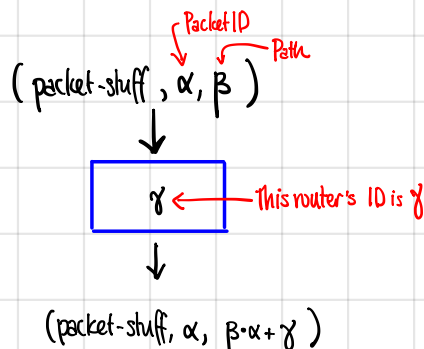
Instead:

• Each packet gets **TWO** field elements appended.

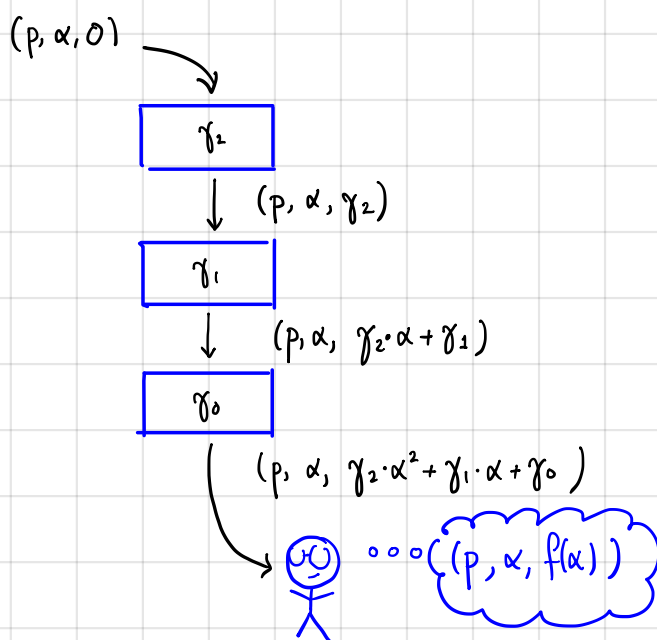
1. Packet ID - the first\* router chooses this at random.
2. Current path ID - this is initialized to 0.

\*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:

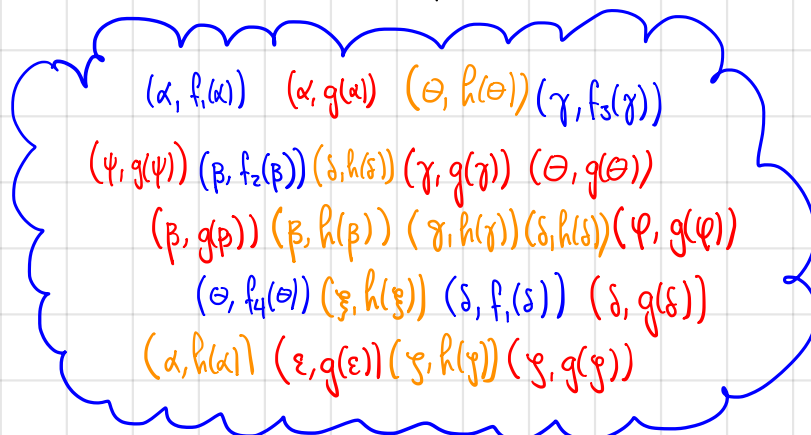


That means that what happens to a packet is:



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

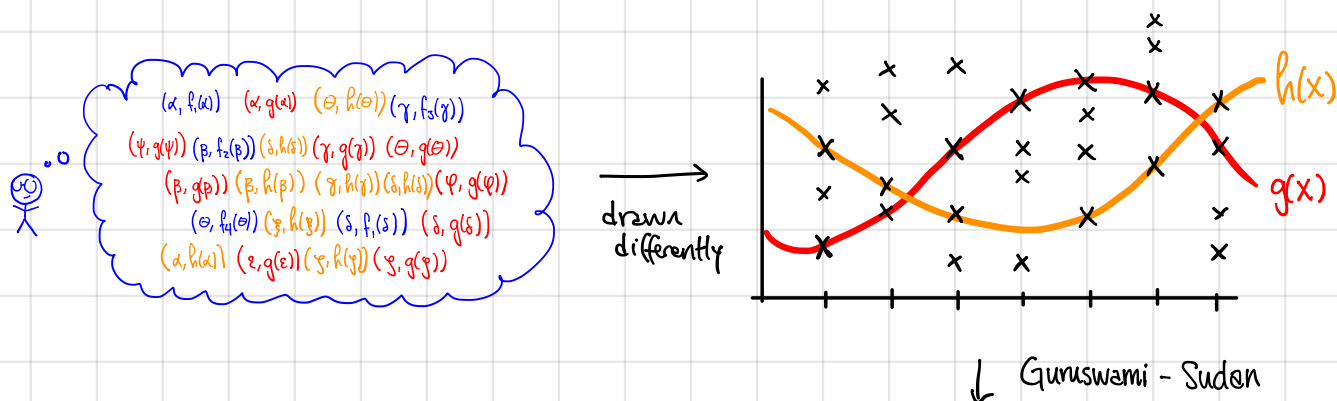
At the end of the day, what you see is:



← 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!



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???