Peeling Algorithms

Part One

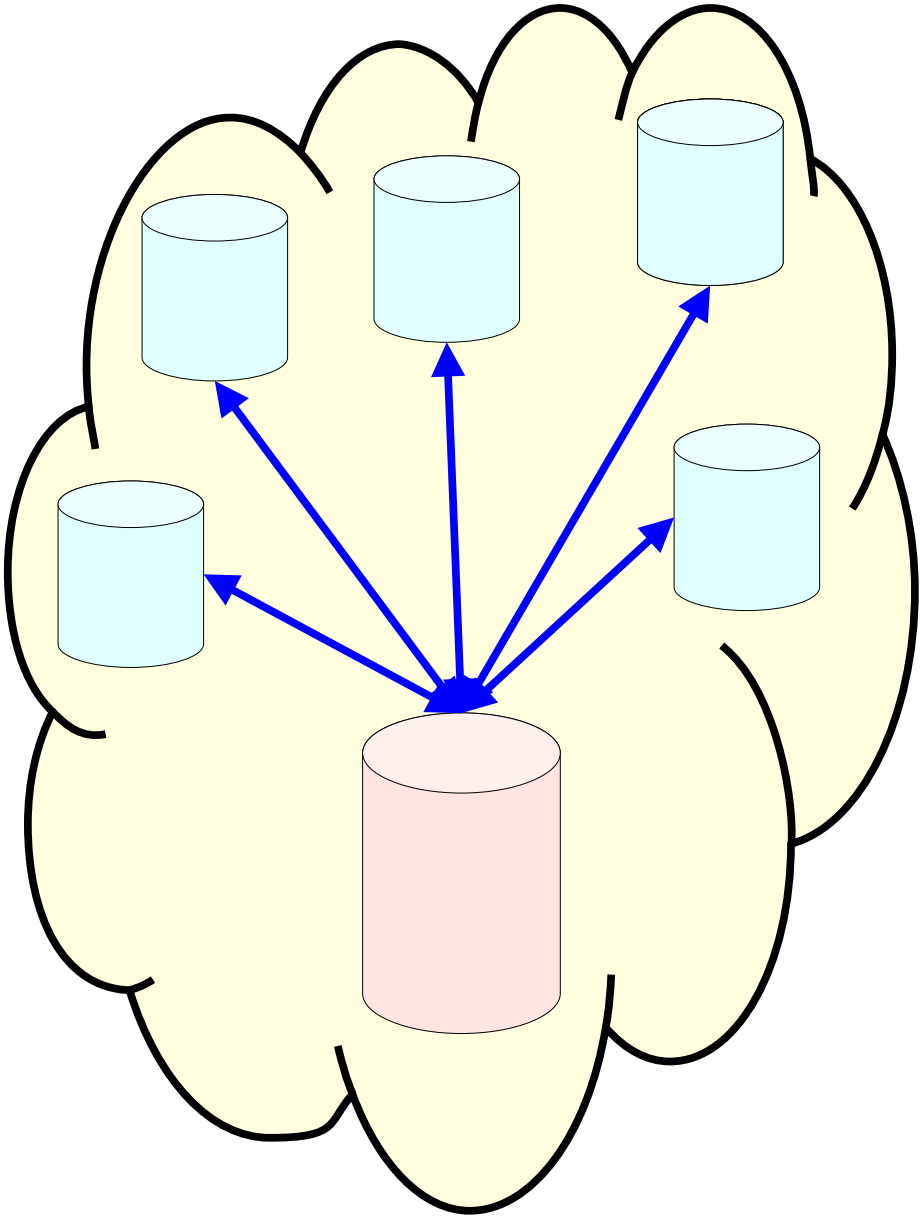
Outline for Today

- ***Set Reconciliation***
  - A common problem in distributed systems.
- ***Invertible Bloom Lookup Tables***
  - A simple, fast, space-efficient solution to set reconciliation.
- ***Peeling Algorithms***
  - The coolest algorithmic primitive you've never heard of.
- ***XOR Filters***
  - Another modern AMQ structure.

Motivation: ***Set Reconciliation***

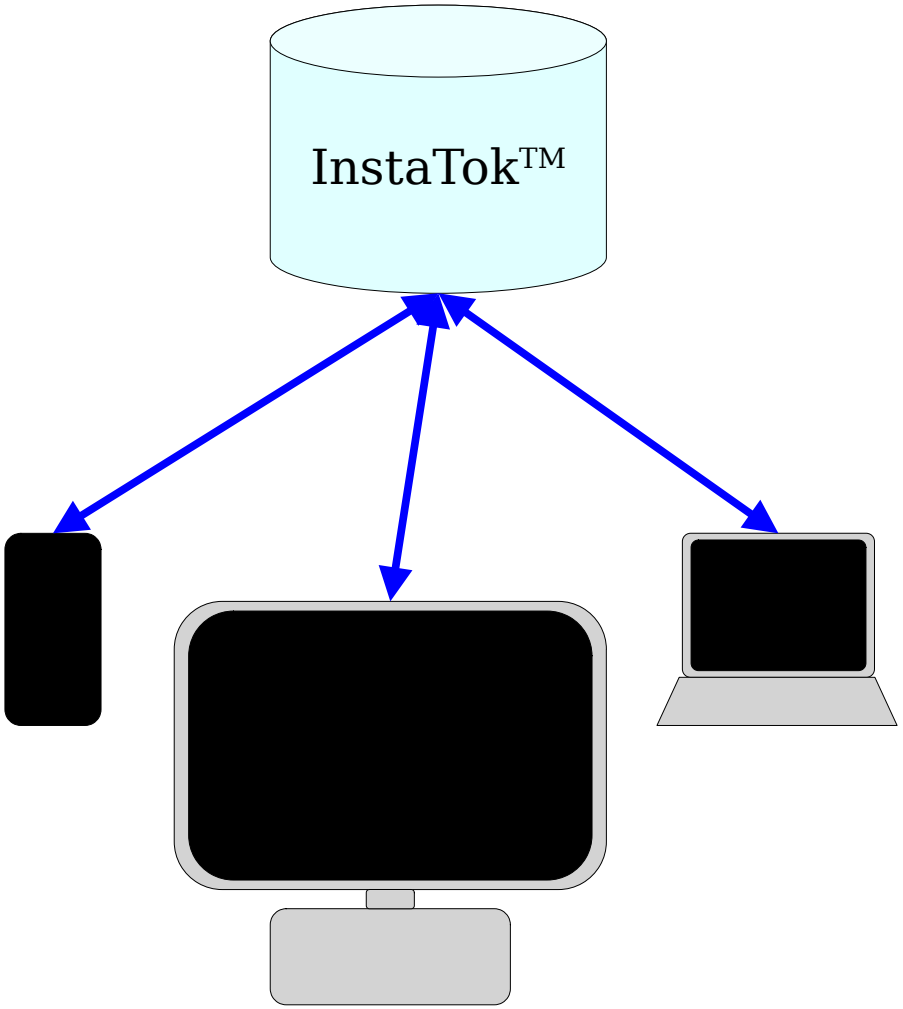
Three Motivating Problems

- We have a central backup server and multiple “frontier” servers.
- Updates happen on the frontiers and must be periodically synced to the backup server.
- ***Goal:*** Sync the backup with the frontier servers, using as little network bandwidth as possible.



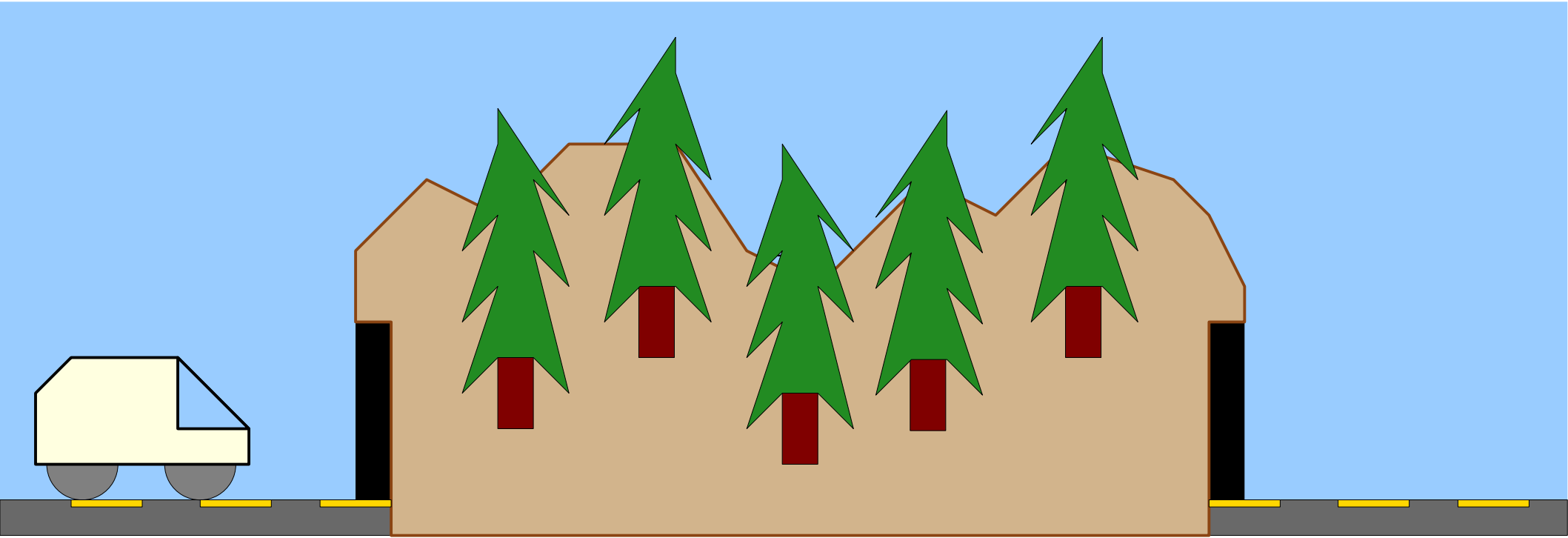
Three Motivating Problems

- Some services have multiple frontends (web, iOS, Android, etc.).
- Suppose you haven't logged into your account on one particular device in a while.
- You want to get updated copies of everything from the central server, but the server has no idea what data you already have.
- ***Goal:*** Do so, using as little network bandwidth as possible.



Three Motivating Problems

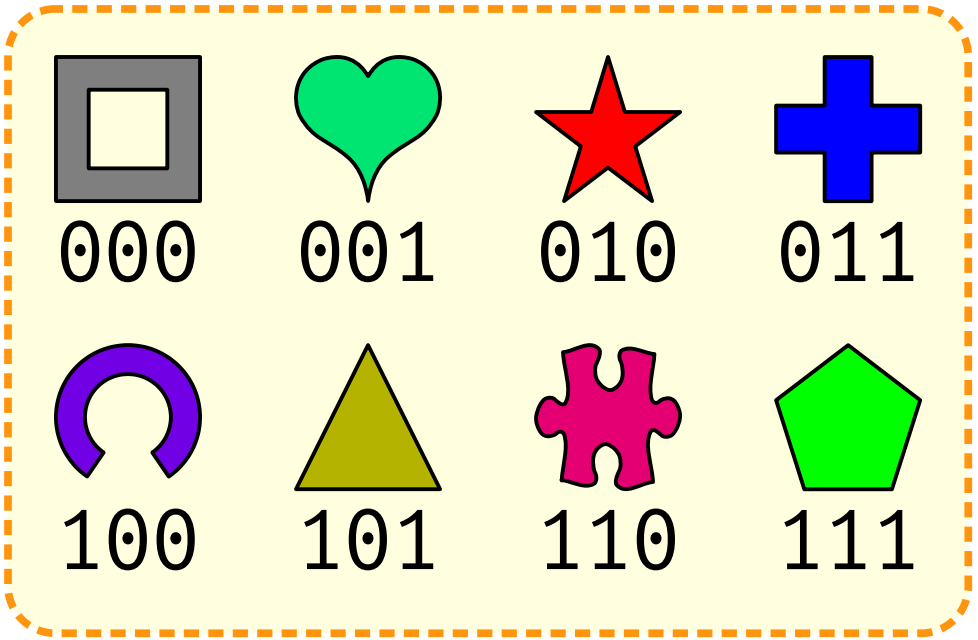
- You are designing the firmware for a modern car.
- The car was in the middle of downloading a software update when it lost signal (e.g. drove into a tunnel), and you don't know what data it received.
- ***Goal:*** Send the rest of the update to the car, using as little network bandwidth as possible.



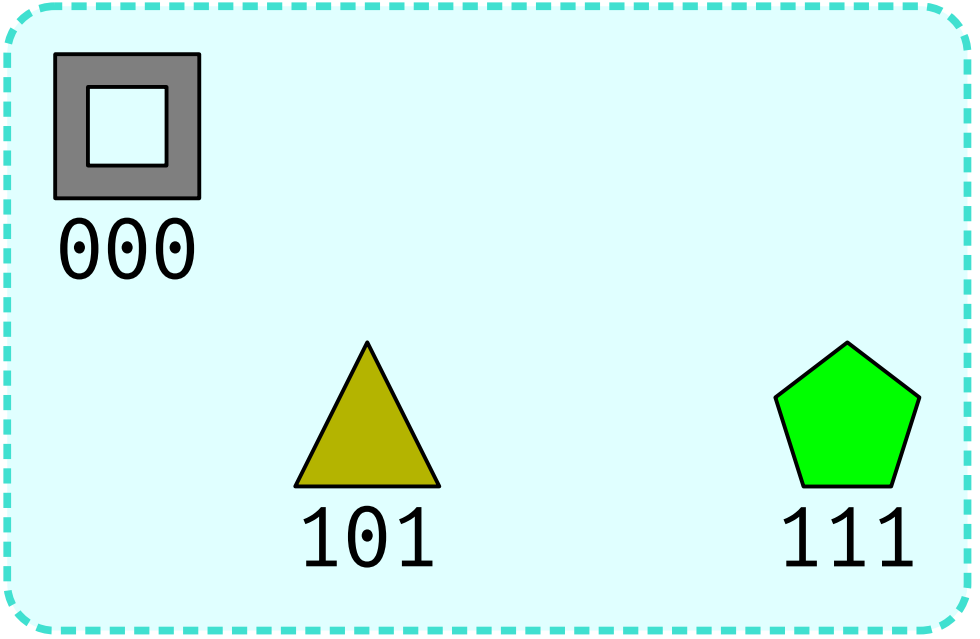
Set Reconciliation

- In the ***set reconciliation problem***, we have two parties each holding sets.
  - Agam holds a set A.
  - Bala holds a set $B \subseteq A$.
- Agam and Bala need to communicate with one another so that Bala ends up with A.
- ***Goal:*** Solve this problem without requiring Agam and Bala to send “too much” information to each other.

Agam's Set A

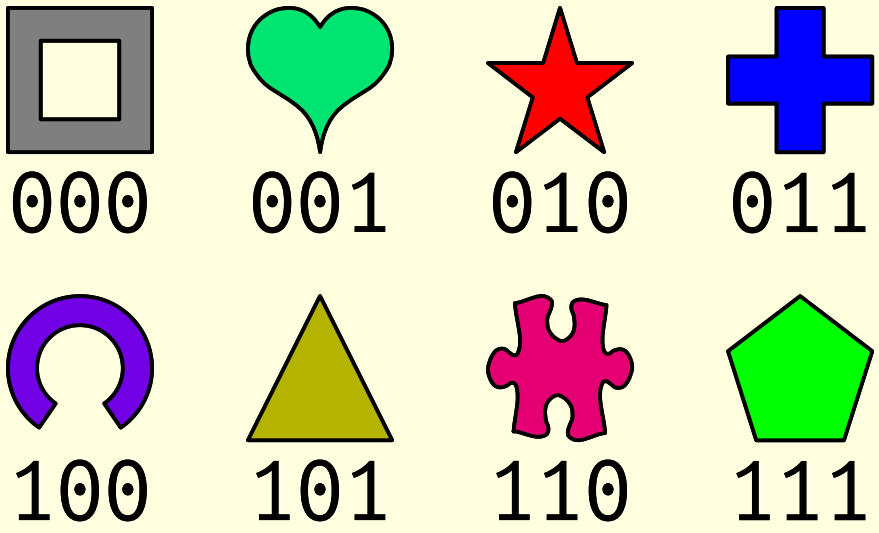


We'll assume every item is represented as an integer of fixed size. (Say, the SHA-256 hash of the underlying data.)



Bala's Set B

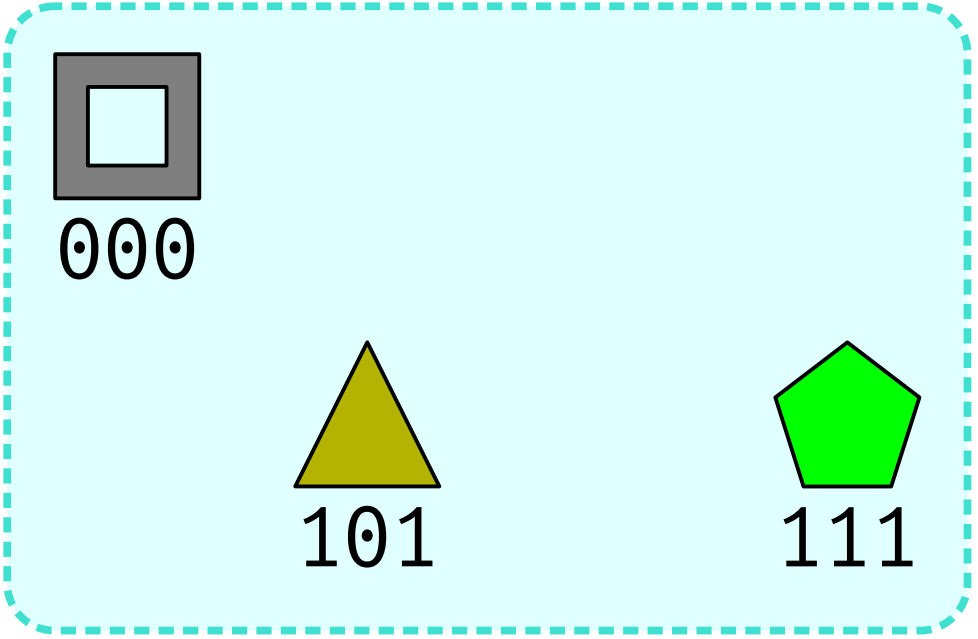
Agam's Set A



Agam can send Bala everything in the set A.

This does not scale as $|A|$ gets larger.

Goal: Solve this problem with less communication.



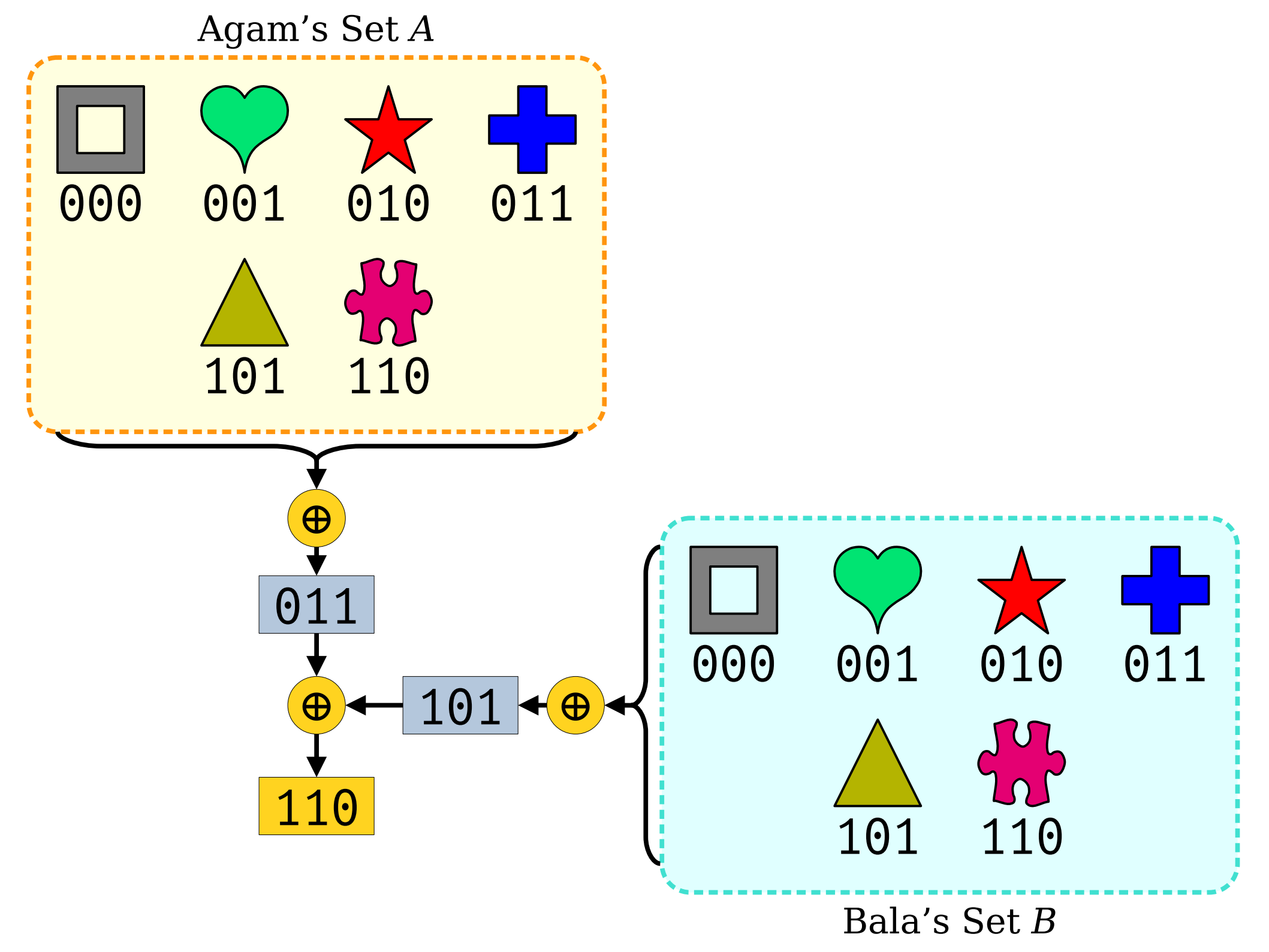
Bala's Set B

Quantifying our Approaches

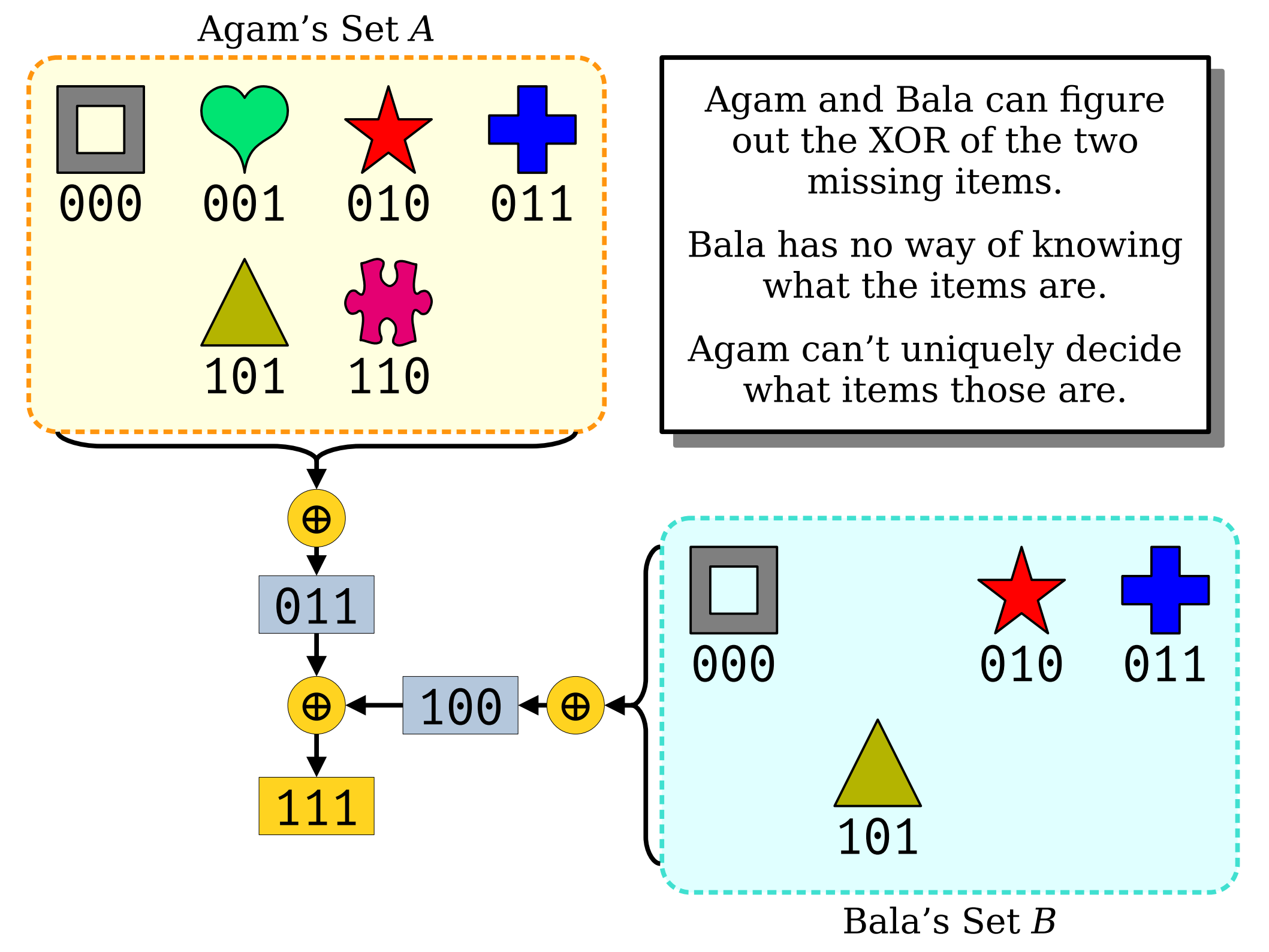
- Let's let ***n*** = $|B|$ be the number of elements in Bala's set.
- Let's let ***k*** = $|A - B|$ be the number of elements Bala doesn't yet have.
- What's the space complexity of Agam sending everything to Bala?
  - ***Answer:*** $O(n + k)$.
- In practice, k will be much, *much* smaller than n.
- ***Goal:*** Solve this problem with space complexity $O(poly(k))$, with no dependency on n.
- How is this even possible?

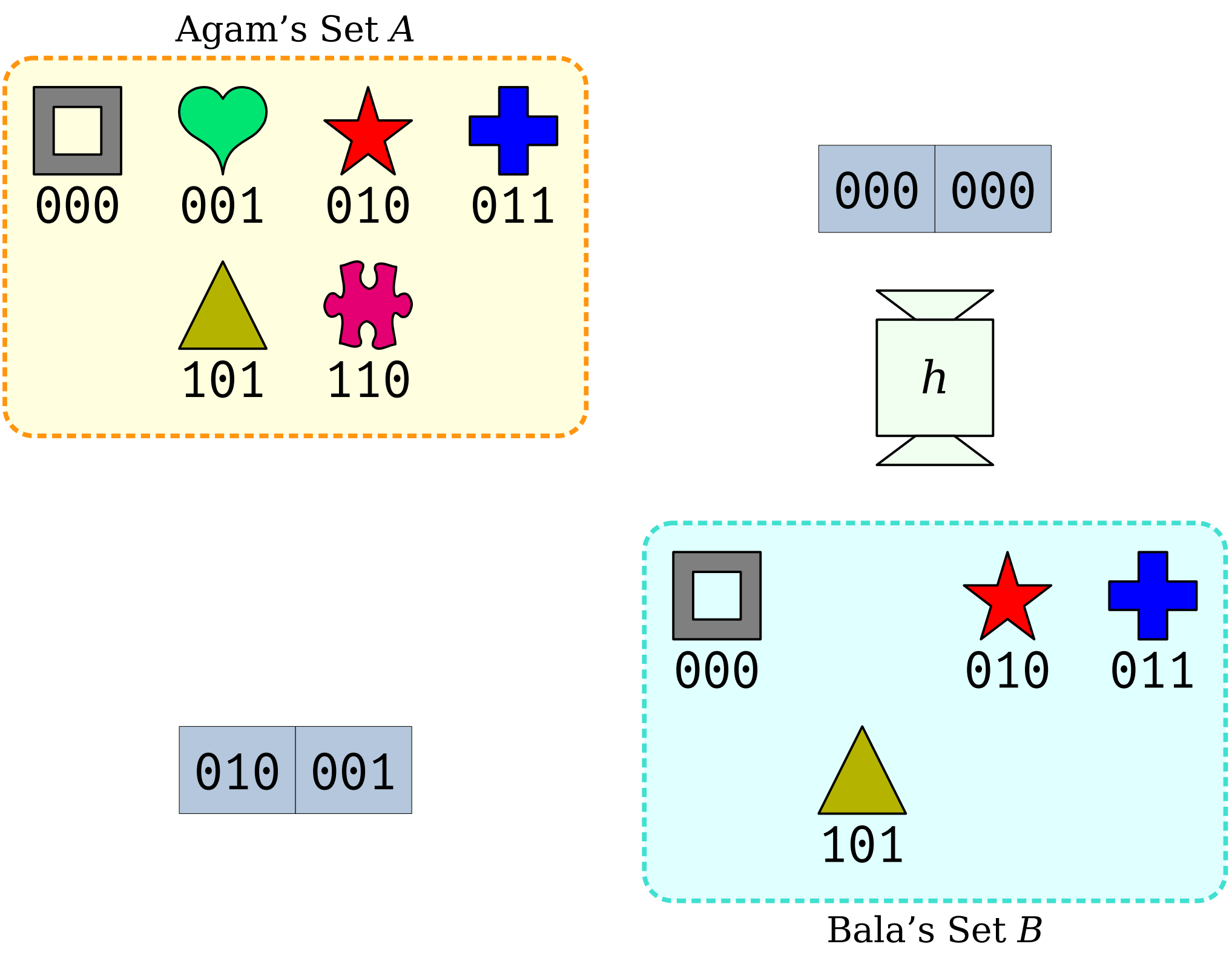
Problem-Solving Technique: When a problem looks too hard, try solving a simpler version of it.

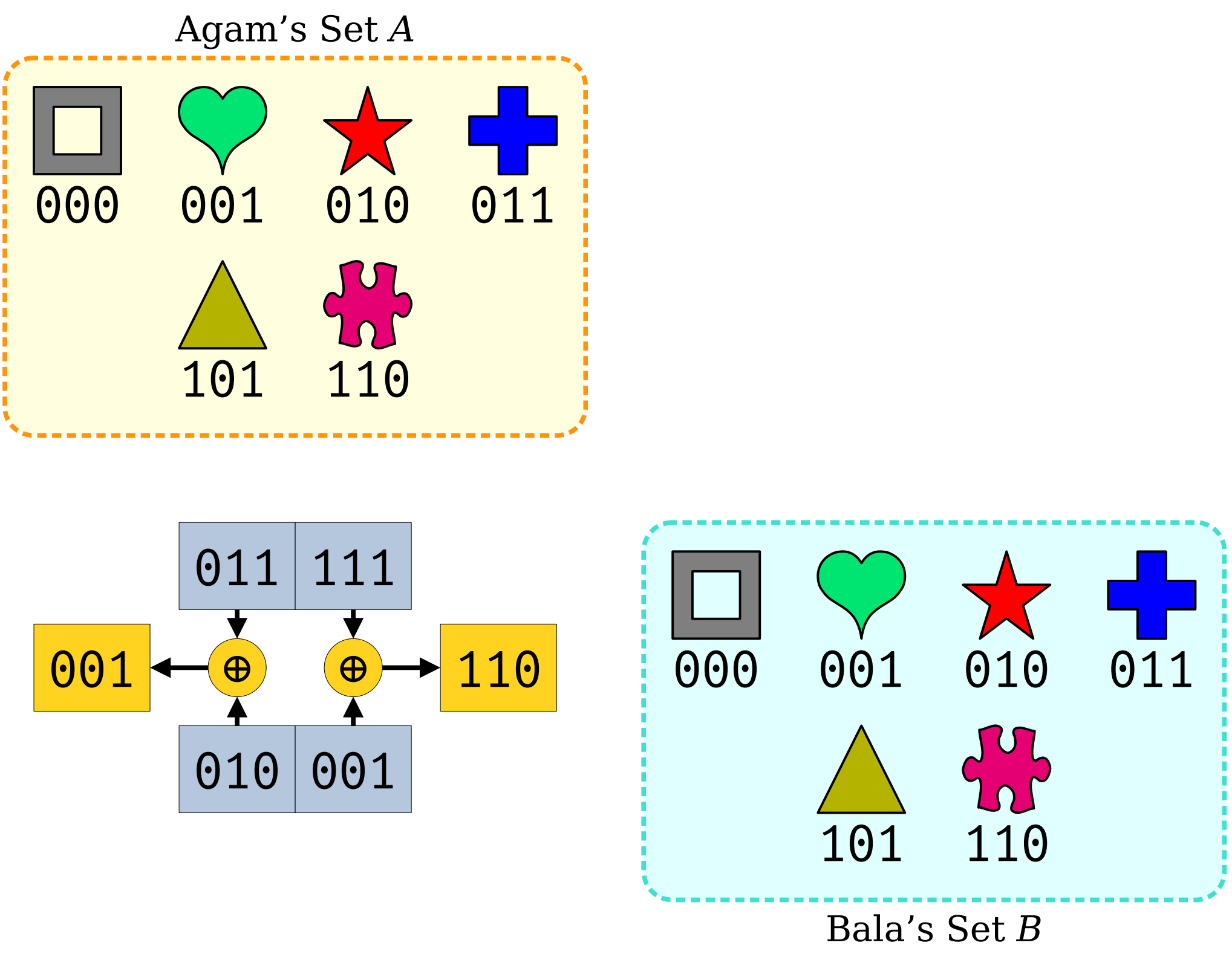
The $k=1$ Case


Agam's Set A
000
001
010
011
101
110
⊕
011
⊕
101
⊕
110
000
001
010
011
101
110
Bala's Set B

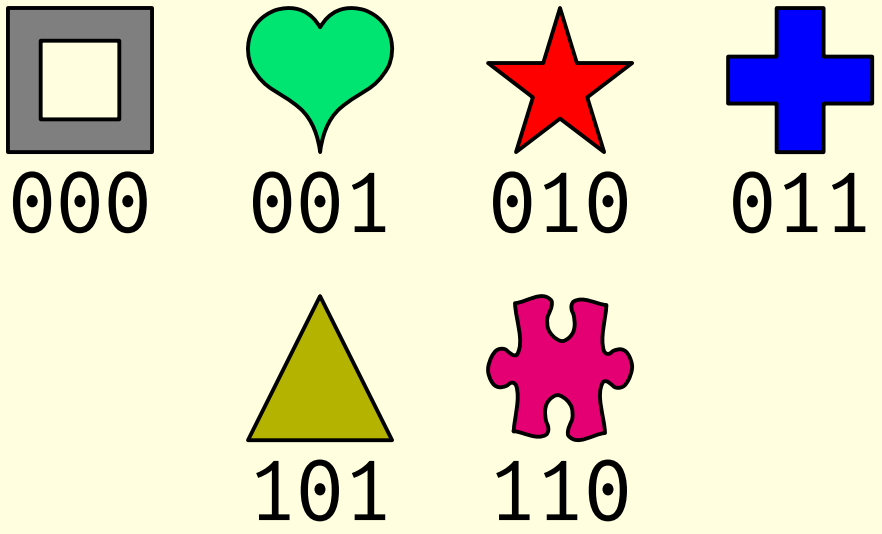
The $k=2$ Case


Agam's Set A
000
001
010
011
101
110
Agam and Bala can figure out the XOR of the two missing items.
Bala has no way of knowing what the items are.
Agam can't uniquely decide what items those are.
⊕
011
⊕
100
⊕
111
000
010
011
101
Bala's Set B

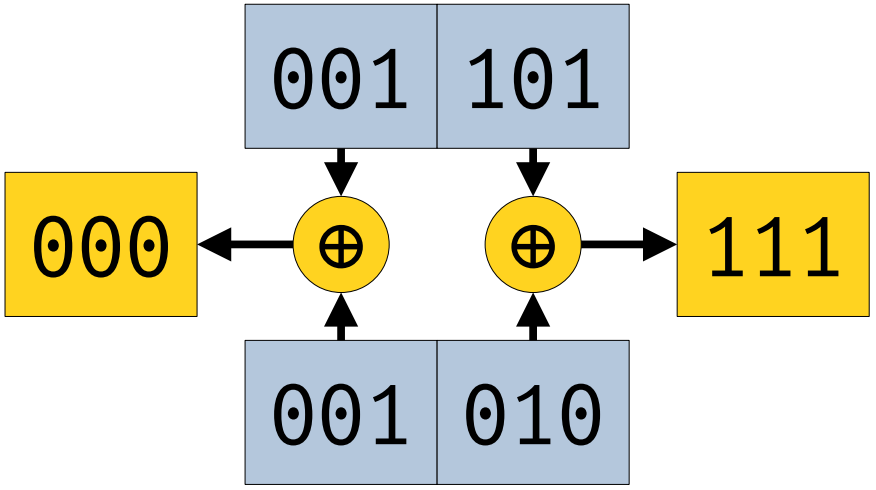

Agam's Set A
000
001
010
011
101
110
000 000
h
000
010
011
101
010 001
Bala's Set B

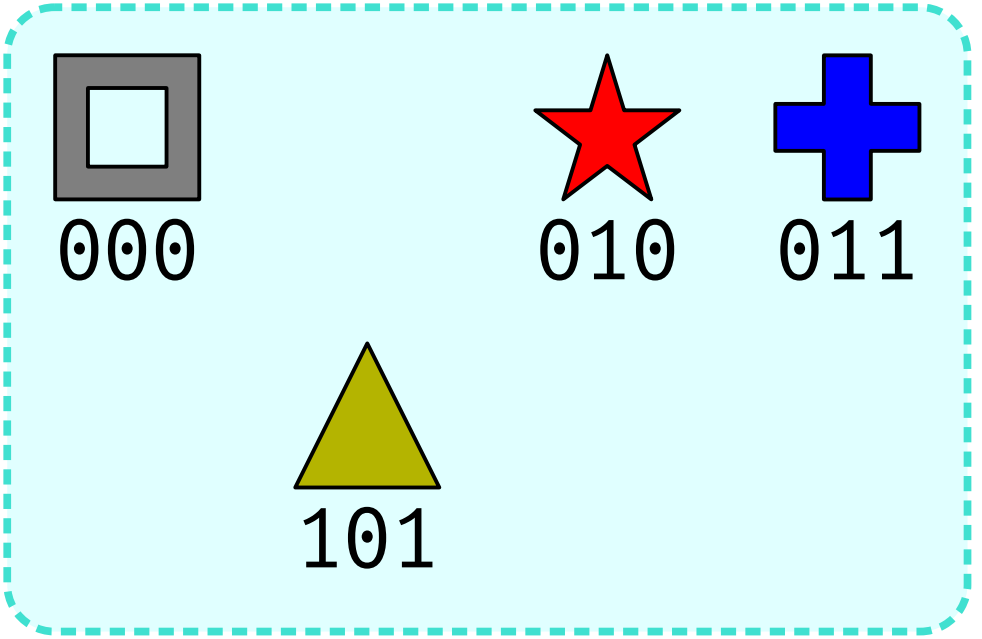

Agam's Set A
000
001
010
011
101
110
011
111
001
110
010
001
101
110
000
001
010
011
Bala's Set B

Agam's Set A



Neither party is guaranteed to know whether these are two separate items or a 0 and an XOR of two.





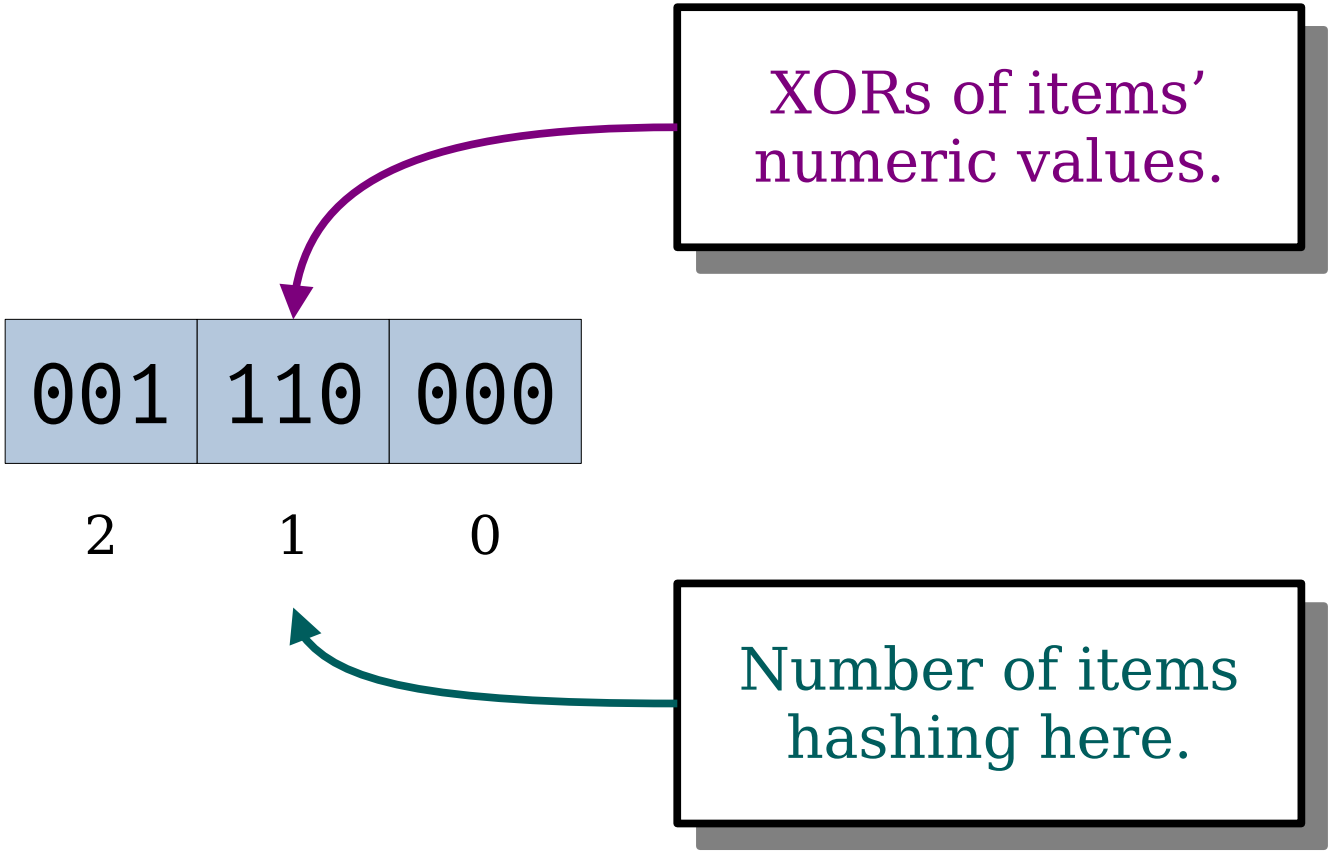
Bala's Set B

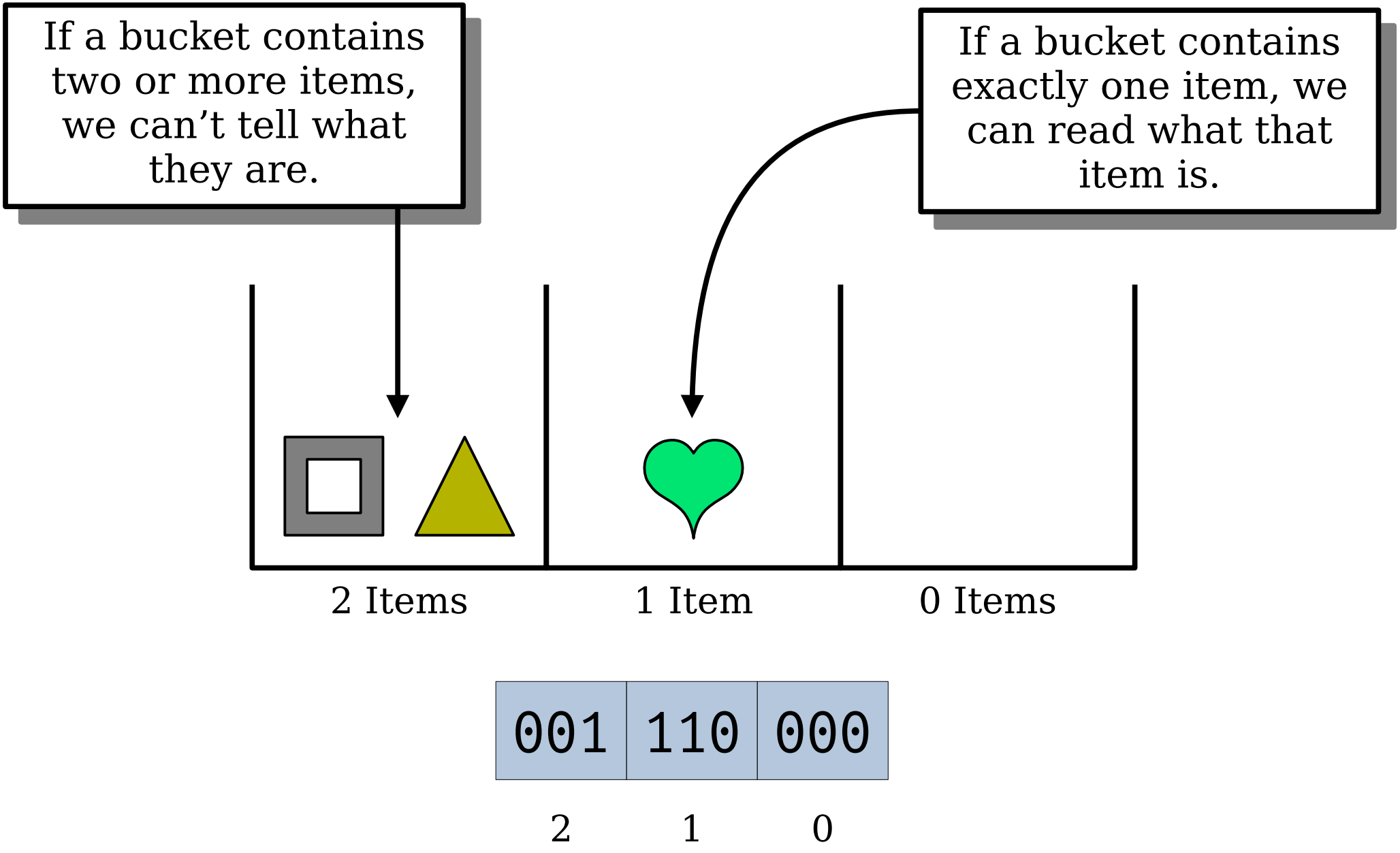
The $k=2$ Case

- This approach has a 50% chance of success.
  - (There are two ways to assign the items to distinct counters and four possible assignments.)
- We can boost the success probability by replicating this approach multiple times with independent hash functions.
- With $\lg \delta^{-1}$ replicated copies, we boost the success probability to $1 - \delta$.
- ***Question:*** Does this generalize to $k \geq 2$?

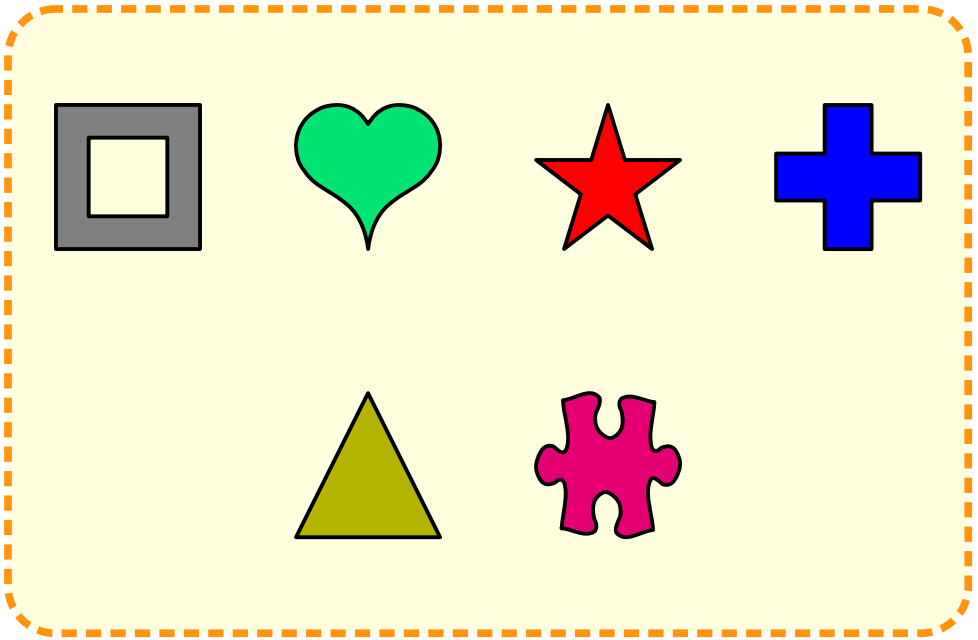
The General Case

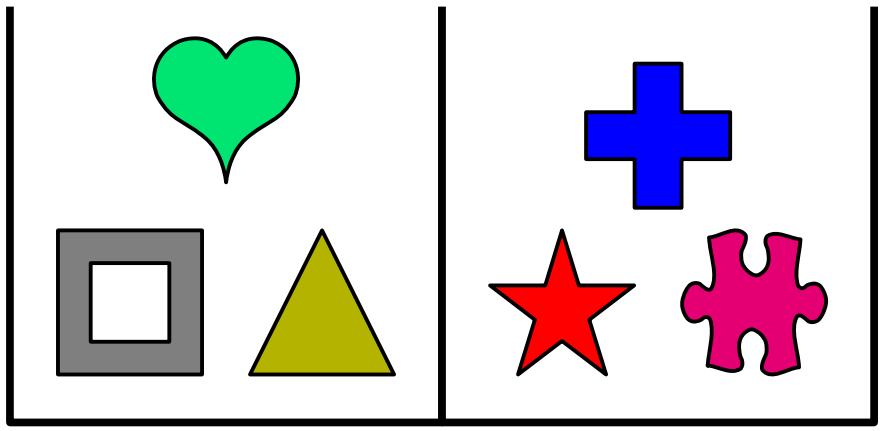
First, a New Perspective


XORs of items' numeric values.
001
110
000
2
1
0
Number of items hashing here.

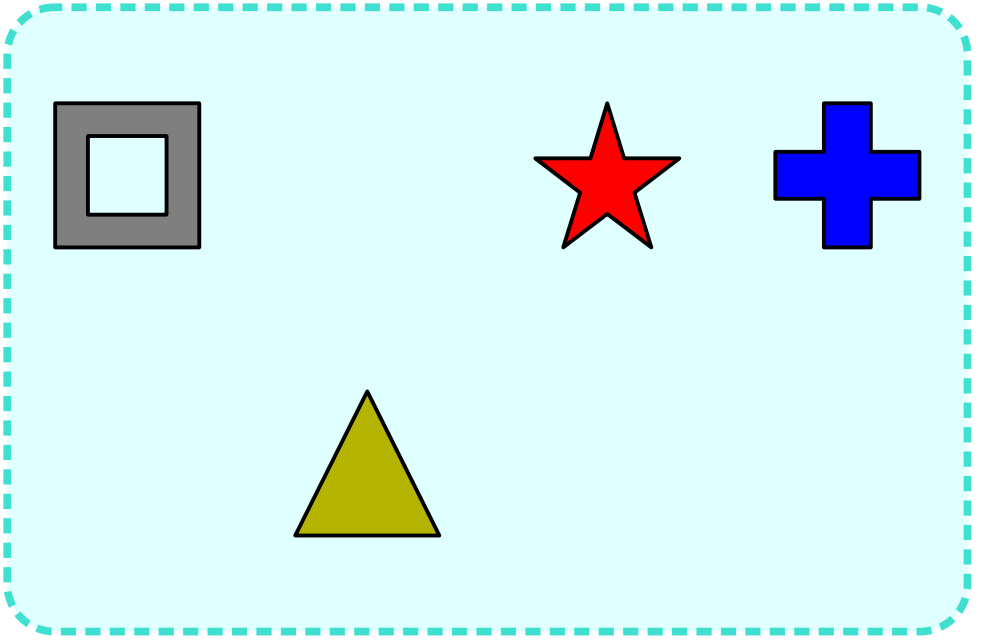

If a bucket contains two or more items, we can't tell what they are.
If a bucket contains exactly one item, we can read what that item is.
2 Items
1 Item
0 Items
001
110
000
2
1
0

Agam's Set A

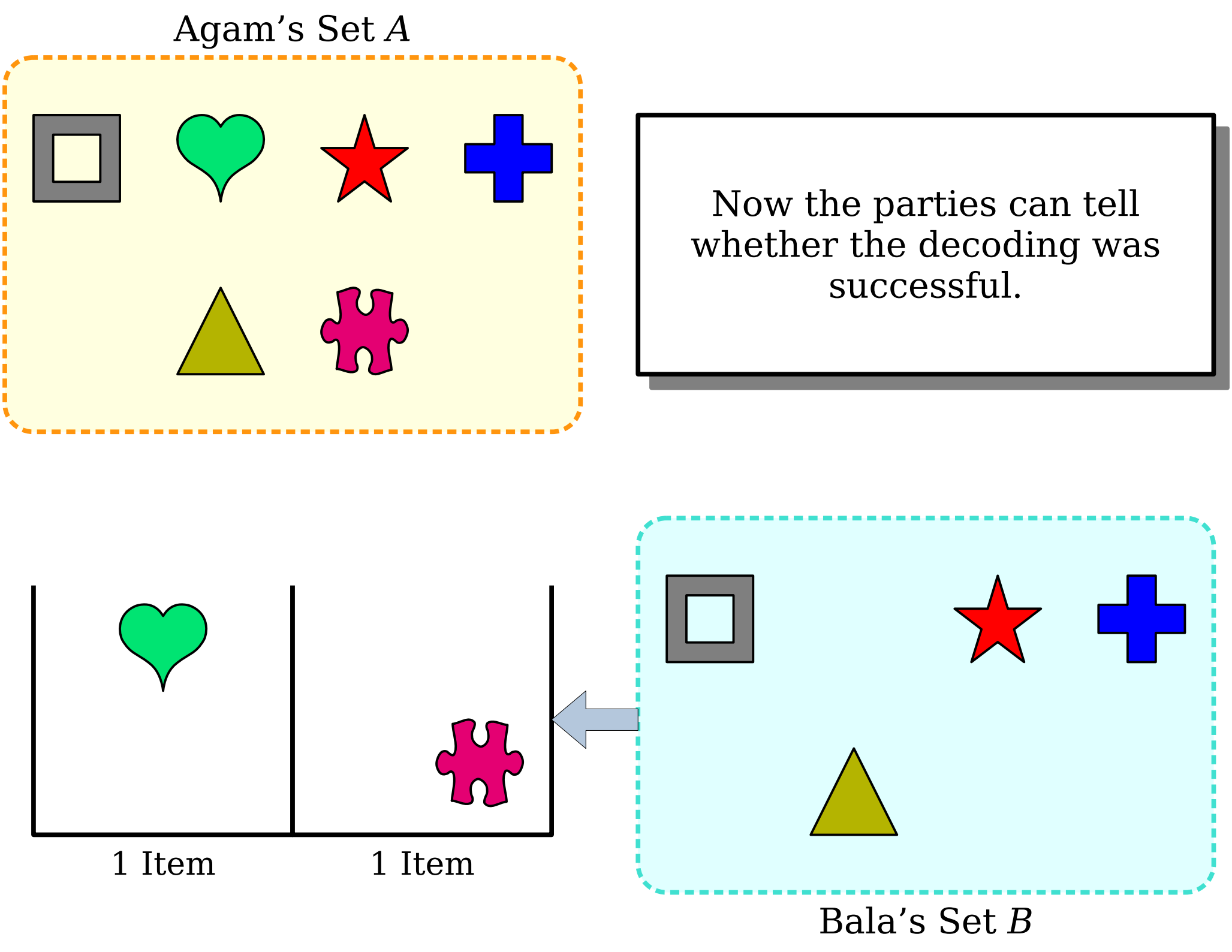




3 Items 3 Items



Bala's Set B


Agam's Set A
Now the parties can tell whether the decoding was successful.
1 Item
1 Item
Bala's Set B

The General Case

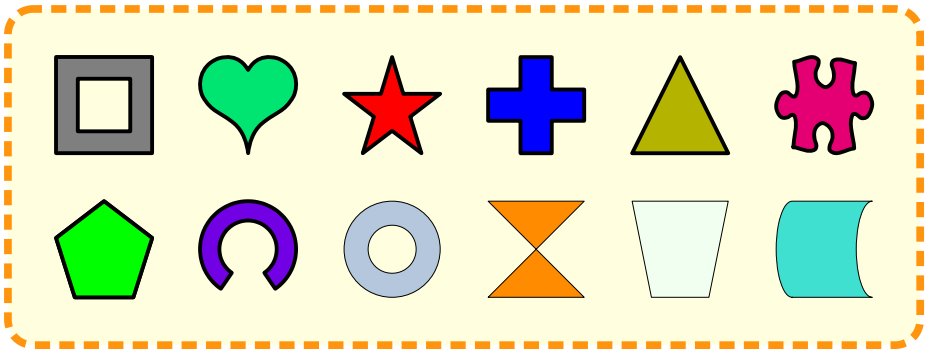
- Create an array of ***m*** buckets, all initially zero. Choose a hash function from elements to the set $\{0, 1, \ldots, m - 1\}$.
- Agam hashes his items and adds them to the appropriate buckets; Bala hashes his items and removes them.
- Agam and Bala can identify all items in buckets whose counts are equal to one.
- This can be done without too much network communication.
  - Picking $m = k$ and replicating this $2 \ln k$ times gives probability $1 - O(k^{-1})$ of recovering all items and requires $\mathbf{2k \ln k}$ buckets to be transmitted.
  - Picking $m = k$, recovering as many items as possible, then repeating this process on the remaining elements requires roughly $\mathbf{3.72k}$ buckets to be transmitted (on expectation, with high probability), but requires a lot of computation by both parties and multiple separate transmissions.
- ***Claim:*** We can do substantially better than this.

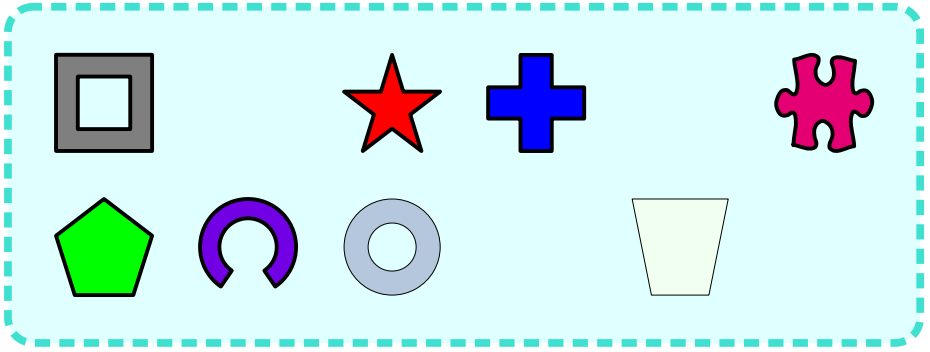
Taking a Step Back

- We're trying to find a way to hash items to buckets that doesn't involve collisions.
- If you squint at this problem in just the right way, this kinda sorta ish looks like what cuckoo hashing was designed to solve.
- There are some major differences, though:
  - Our available memory is *way* smaller than the number of items.
  - We only care about collisions between *missing* items, and we don't know what those are up front.
- ***Question:*** Are there any ideas we could adapt that would work here?

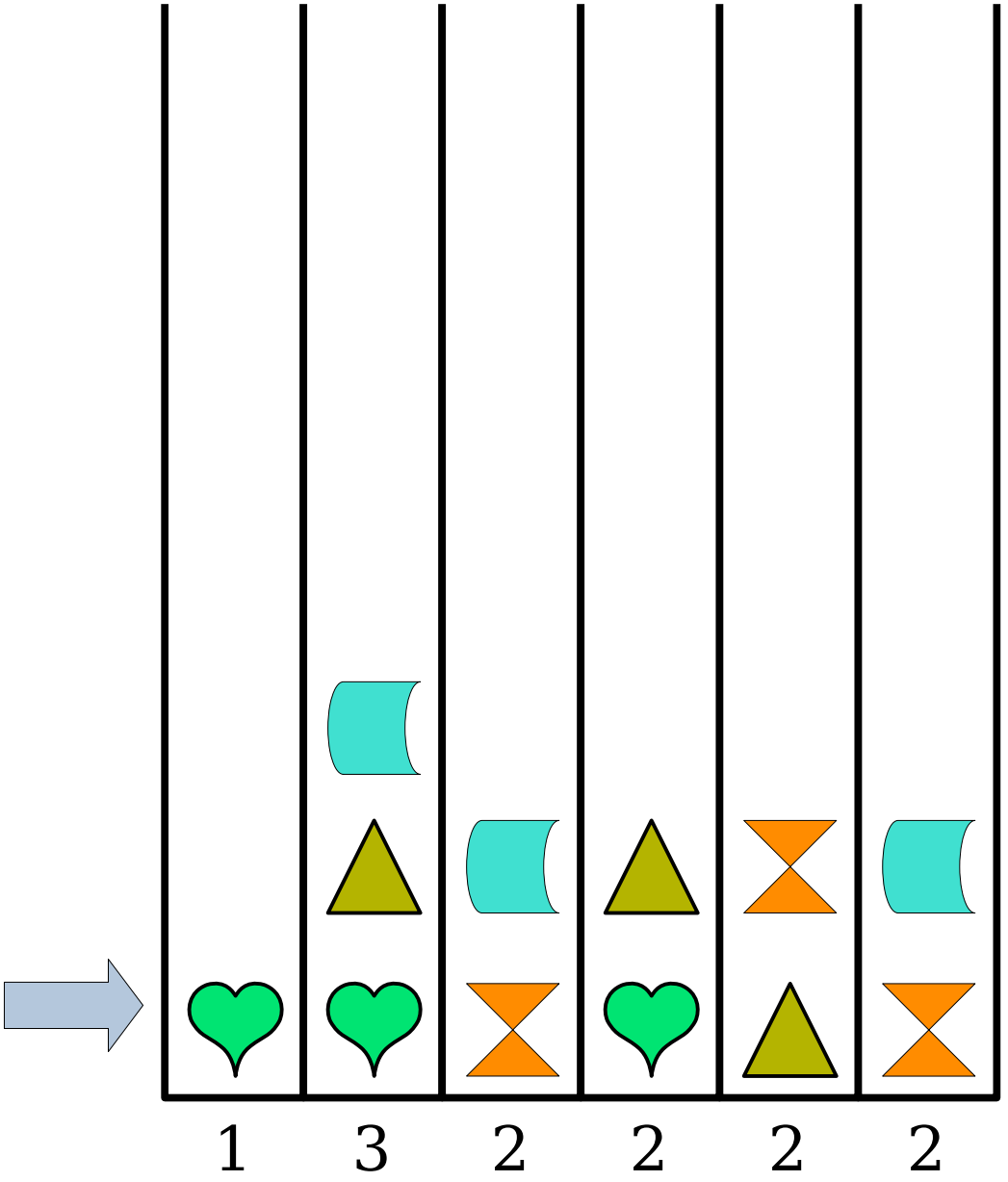
Idea: Hash each item to $d \geq 2$ buckets.

Agam's Set A



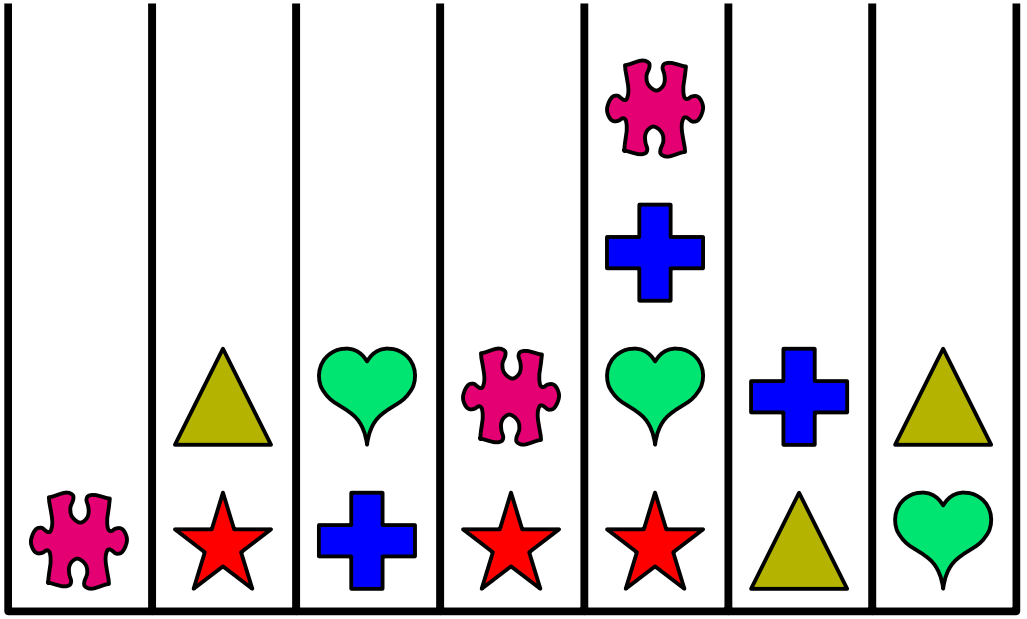


Bala's Set B



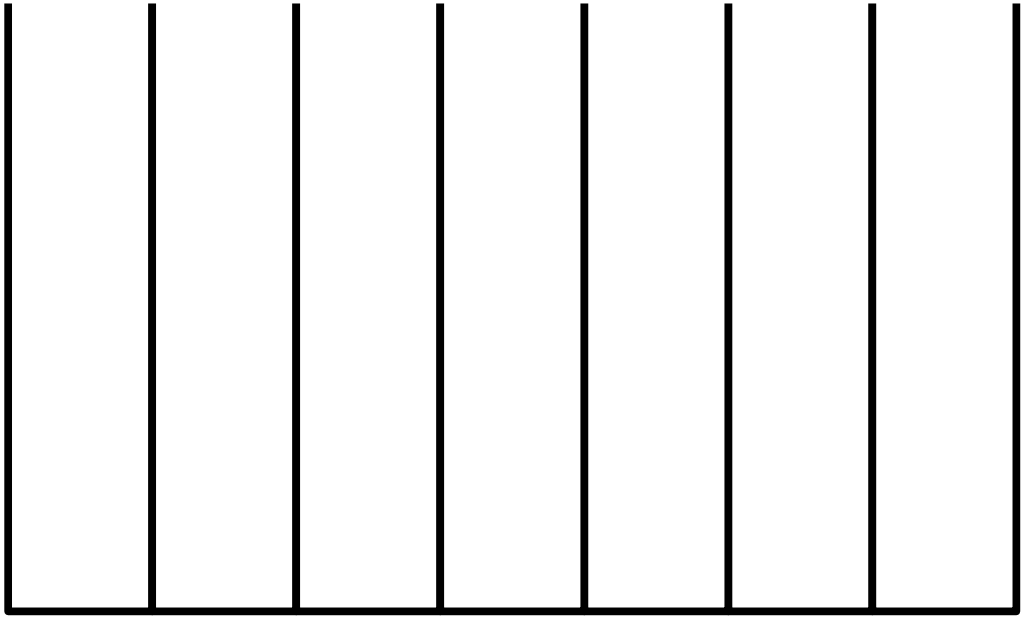
The Peeling Algorithm

- If we have multiple copies of each item, we might still be able to recover them all even when collisions exist.
- ***Algorithm:*** Repeatedly find a bucket with one item in it, identify the item, and remove all copies of it.
- (To perform the last step: hash the item with all the hash functions, XOR it out of the buckets it's in, and decrement the appropriate counters.)



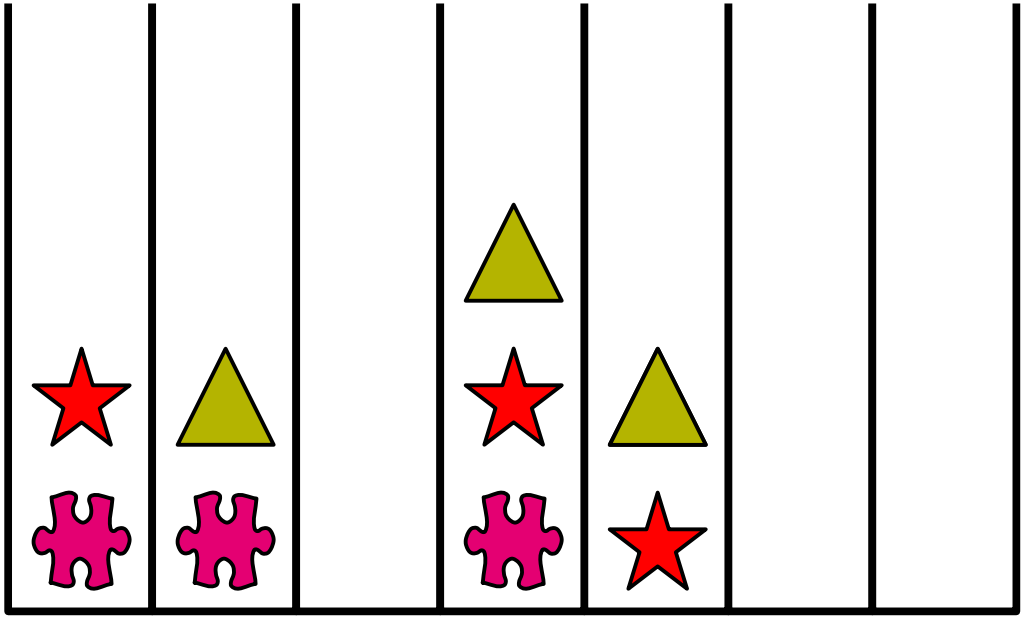
The Peeling Algorithm

- If we have multiple copies of each item, we might still be able to recover them all even when collisions exist.
- ***Algorithm:*** Repeatedly find a bucket with one item in it, identify the item, and remove all copies of it.
- (To perform the last step: hash the item with all the hash functions, XOR it out of the buckets it's in, and decrement the appropriate counters.)



The Peeling Algorithm

- Peeling doesn't always work; we can get "stuck" with no more peelable elements.
- ***Question:*** How likely is it that we'll be able to peel all the elements away?



The Peeling Algorithm

- We have three parameters to consider:
  - k, the number of items to distribute.
  - m, the number of buckets we choose.
  - d, the number of hash functions we pick. (Equivalently, the number of copies of each item we distribute across the buckets.)
- We can't control k. However, we can pick m and d.
- We want to know the chance that we can recover ***every*** element through peeling.
- What happens to the peeling probability as we vary m?
- What happens to the peeling probability as we vary d?
- Think about what happens if m and d are each quite small or quite large.

Answer at
https://cs166.stanford.edu/pollev

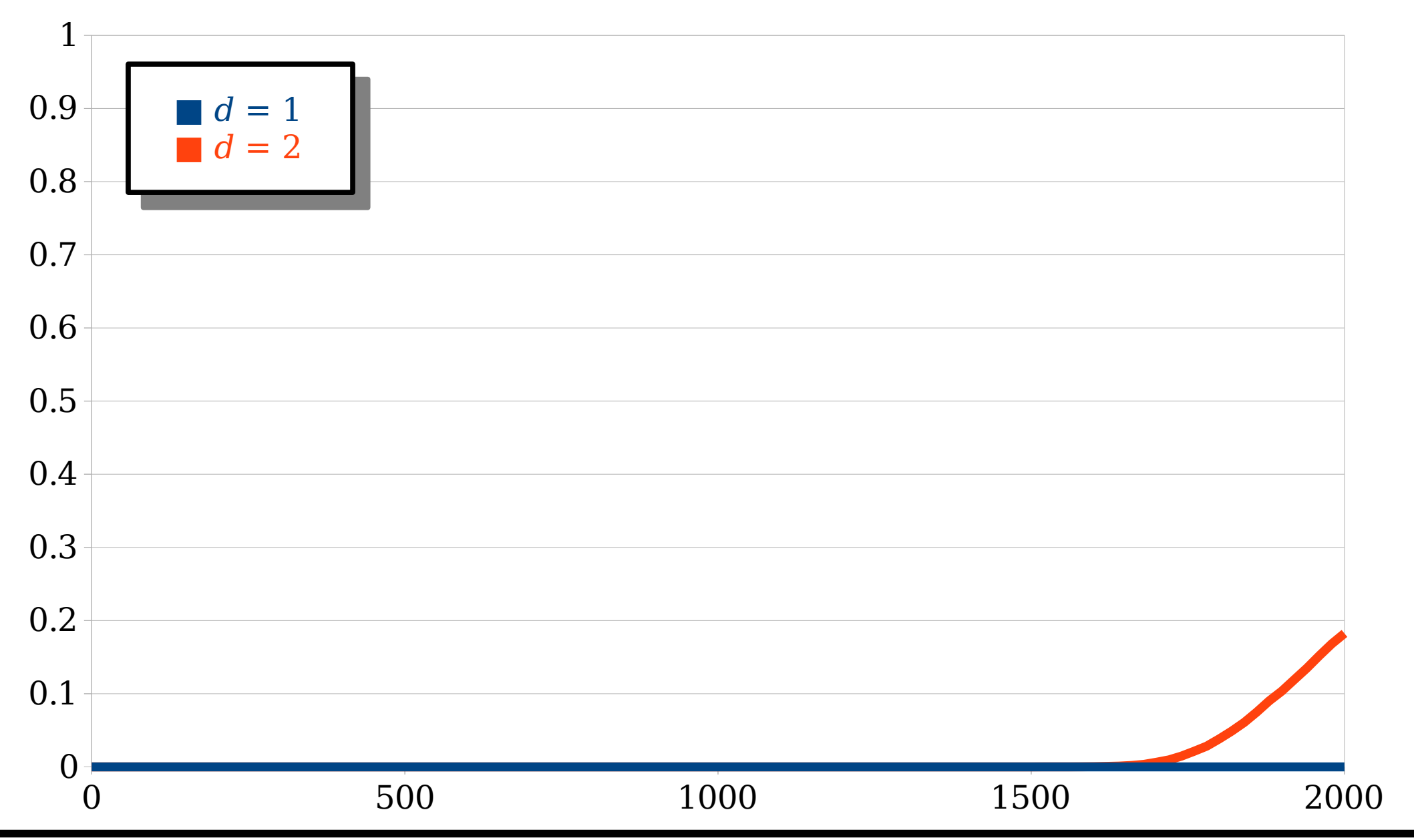
Tuning m

- If m is too small, we expect the success probability to drop to zero.
  - Extreme case: All in one bucket means we can never peel.
- If m gets larger, we expect the success probability to increase toward one.
  - Extreme case: with infinitely many buckets, we never get collisions.
- ***Goal:*** Make m large enough to ensure good success probability, but small enough that we don’t have to transmit too much data.

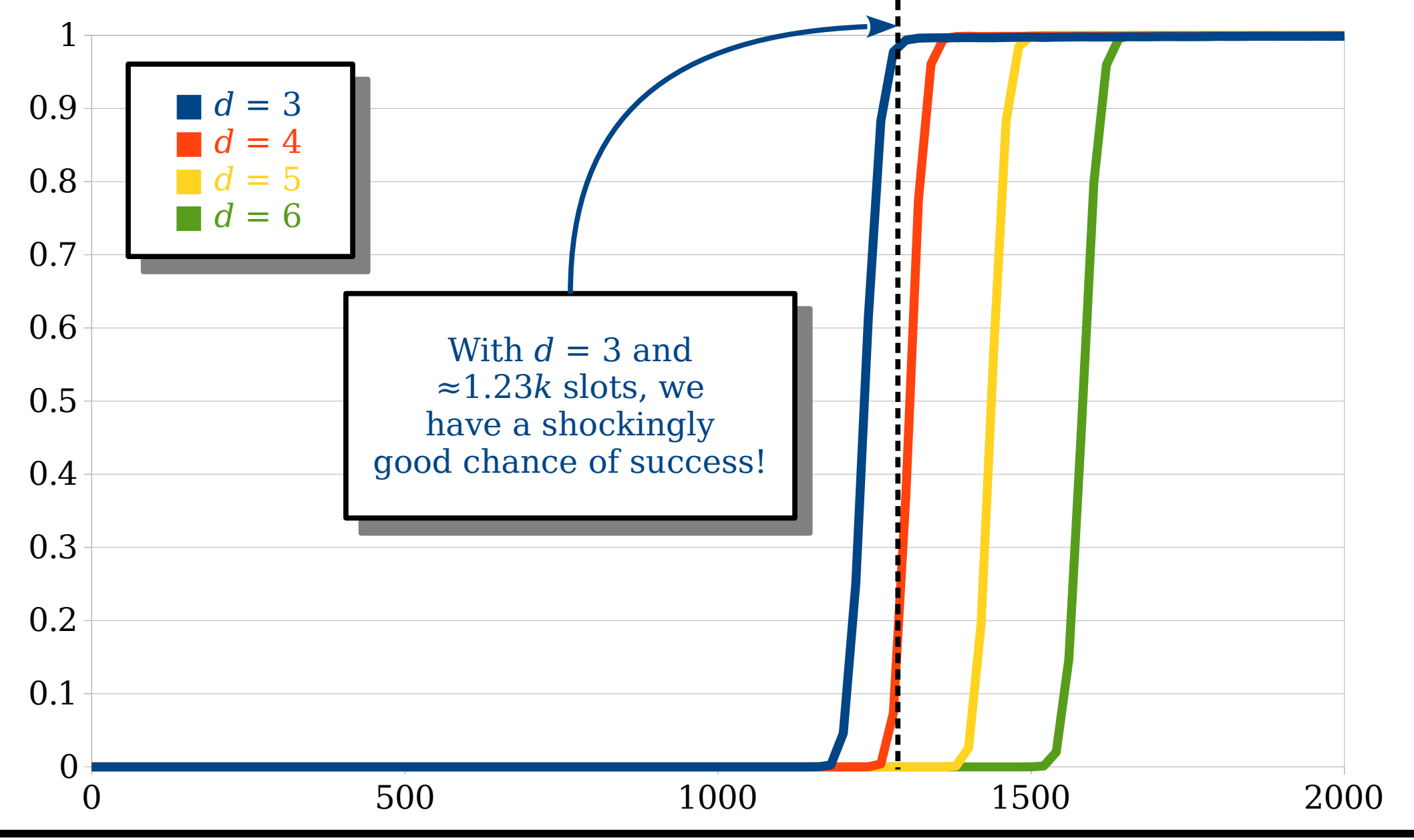
Tuning *d*

- The impact of *d* is a bit more subtle.
- If *d* is too small, collisions will be harder to resolve.
  - Extreme case: if $d = 1$, we're back in the same situation we started with earlier.
- If *d* is too large, we'll get so many collisions we won't be able to find anything to peel.
  - Extreme case: if $d = m$, peeling always fails.
- ***Goal:*** Find a "sweet spot" for *d* to ensure the highest probability of success.

Pro Tip: When designing a data structure, it never hurts to get empirical data first!



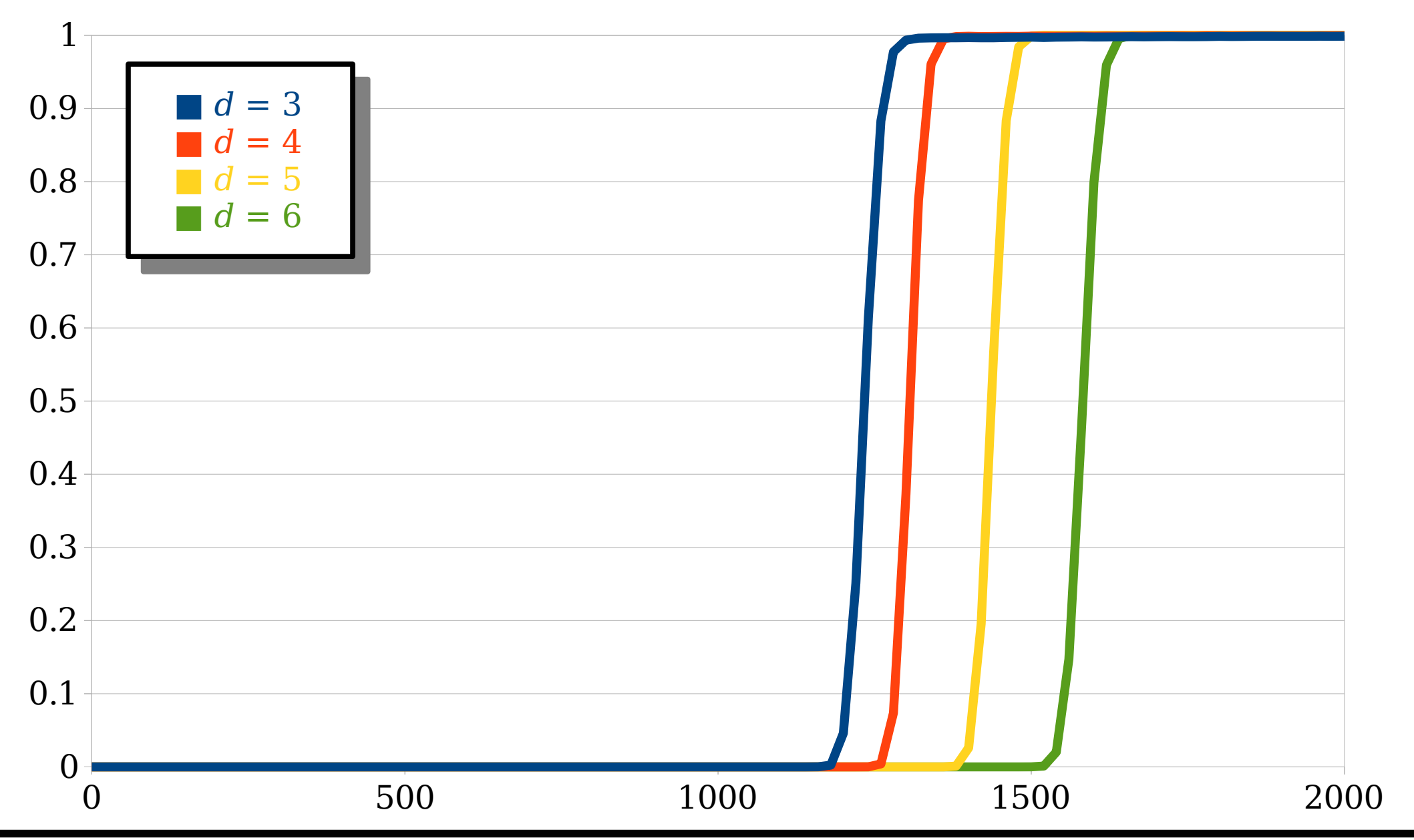
What is the probability that all elements are peeled when we have $k = 1000$ elements and m buckets, as a function of d?



What is the probability that all elements are peeled when we have $k = 1000$ elements and m buckets, as a function of d?

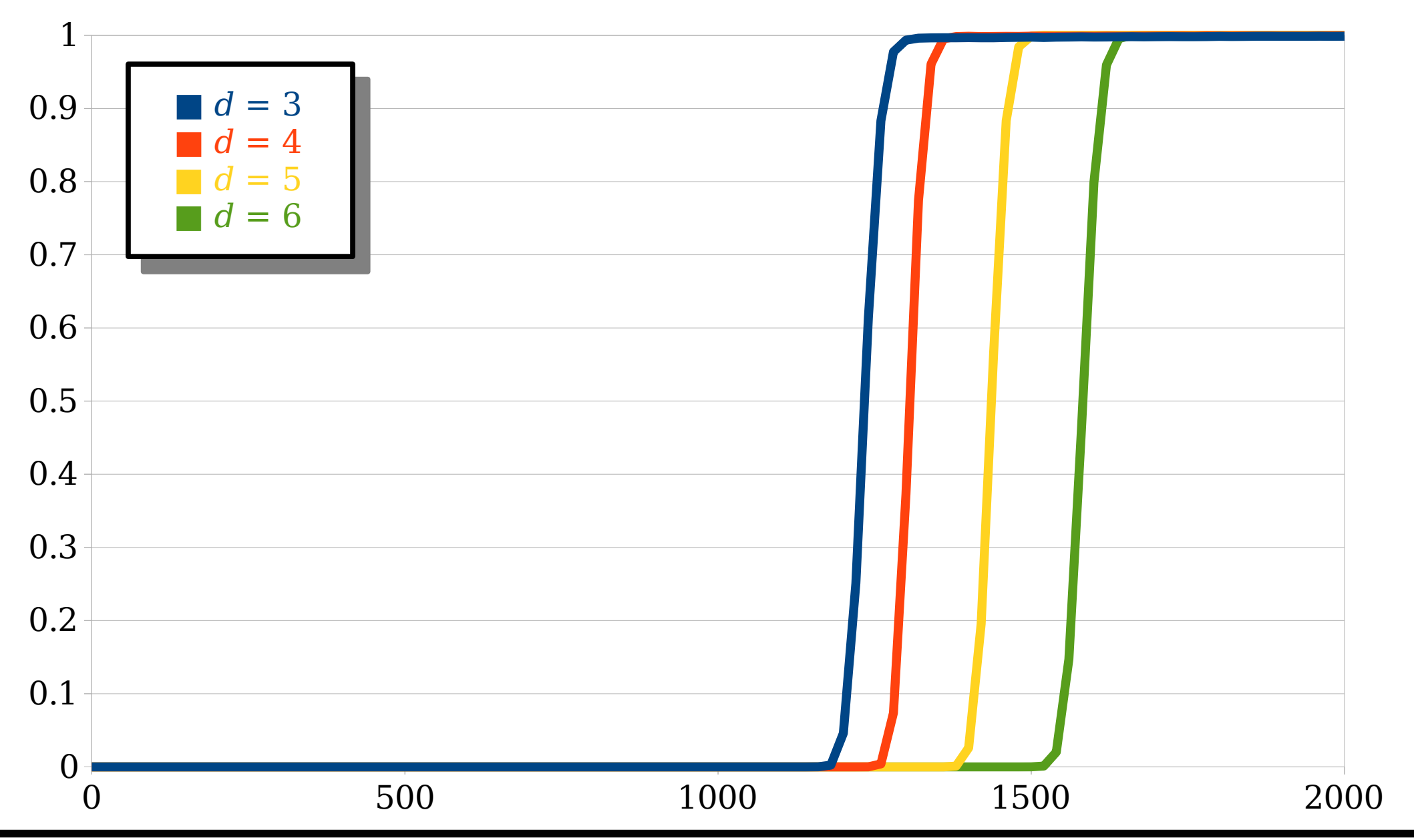
Invertible Bloom Lookup Tables

- Create an array of $\approx 1.23k$ empty buckets. Choose 3 hash functions h_1, h_2, and h_3. Assume they behave truly randomly.
- We support three operations:
  - ***add***(x): Add x to buckets $h_1(x)$, $h_2(x)$, and $h_3(x)$.
  - ***remove***(x): Remove x from buckets $h_1(x)$, $h_2(x)$, and $h_3(x)$.
  - ***list***(): Run the peeling algorithm.
- Agam ***add***s his items, Bala ***remove***s his, and then either can ***list*** the items to find what's missing.
- This is called an ***invertible Bloom lookup table*** or ***IBLT***.
  - Unlike its namesake, it is *not* obsolete. It's the basis for many production systems.
- It requires $\approx 1.23k$ buckets to be transmitted, has success probability $1 - O(k^{-1})$, and requires very little computation by the two parties.
  - Or you can use four hash functions, $\approx 1.31k$ buckets, and have success probability $1 - O(k^{-2})$; or five hash functions, $\approx 1.44k$ buckets, and have success probability $1 - O(k^{-3})$, etc.



Why does this happen?

Next Time!



What else can we do with this?

Time-Out for Announcements!

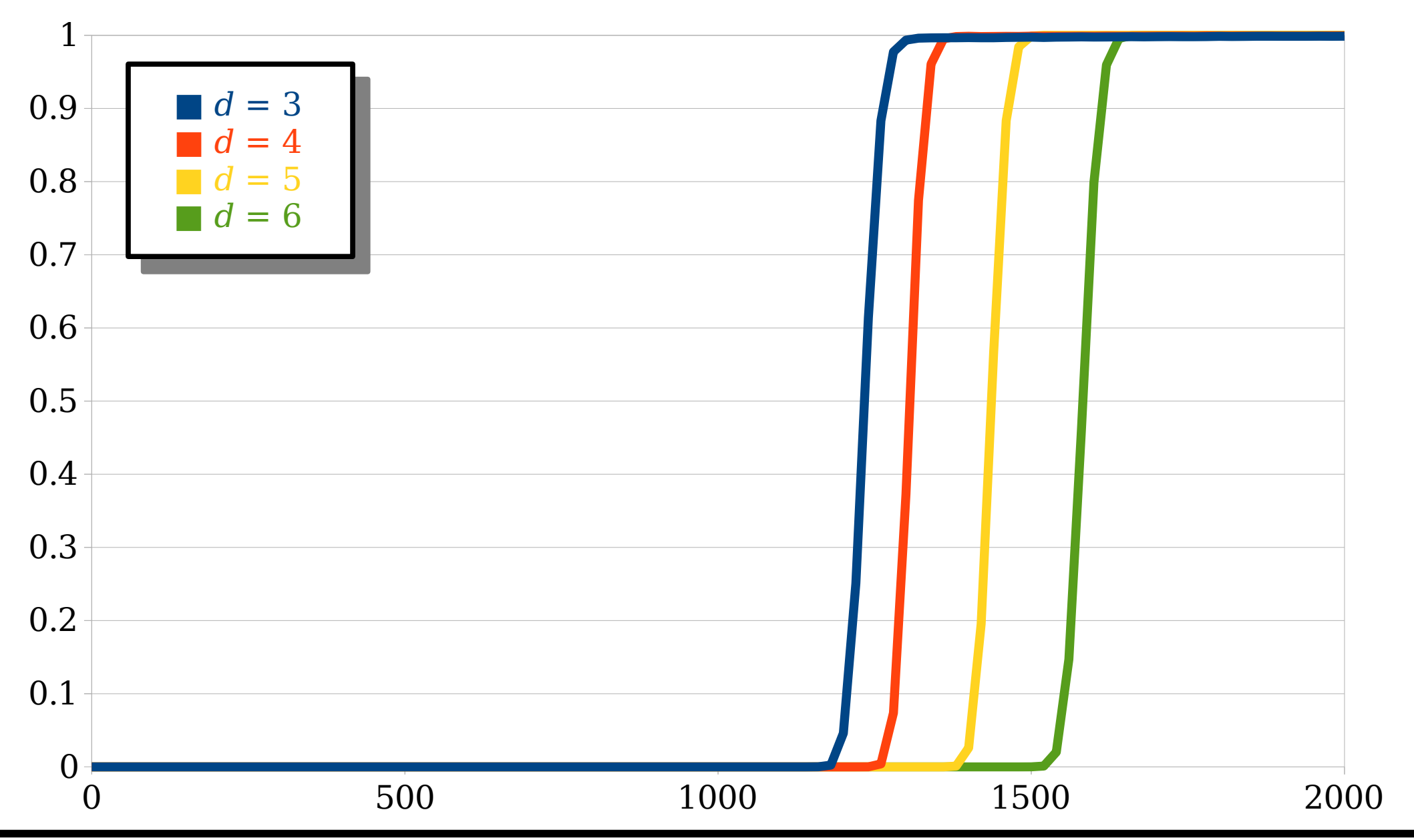
Problem Set 5

- Problem Set 5 was due at 1PM today.
  - Need more time? Use one late day to extend the deadline by 24 hours or two to extend it by 48 hours.
  - No submissions will be accepted after that.
- Problem Set 4 has been graded; solutions are on the course website.
- That’s it for problem sets this quarter! Congratulations!

Final Exam Logistics

- Our final exam is on ***Saturday, June 6th*** from ***3:30PM – 6:30PM*** right here in STLC 114.
- Exam is cumulative. All topics from lectures and the problem sets are fair game.
- The exam is closed-book, closed-computer, and limited-note. You can bring one double-sided 8.5” × 11” sheet of notes with you.
- A practice exam (last year’s final) is on the course website.
- Students with OAE accommodations: we think we’ve reached out to all of you to coordinate an alternate exam time. If not, ping us ASAP!

Back to CS166!



What else can we do with this?

XOR Filters

Recap: AMQ

- ***Goal:*** Store an "approximation" of a set S of n elements.
  - False *negatives* are not allowed.
  - False *positives* happen with probability at most ε.
- Here's where we left off. Can we do better?

	Bits Per Element	Hashes/Query
Bloom Filter (1970)	$1.44 \lg \varepsilon^{-1}$	$\lg \varepsilon^{-1}$
Cuckoo Filter, $b = 4$ (2014)	$1.02 \lg \varepsilon^{-1} + 3.06$ *(for sufficiently small ε)*	3

Bloom and Cuckoo Revisited

- From the Bloom filter, we have the following ideas:

  Store a large array of tiny items.

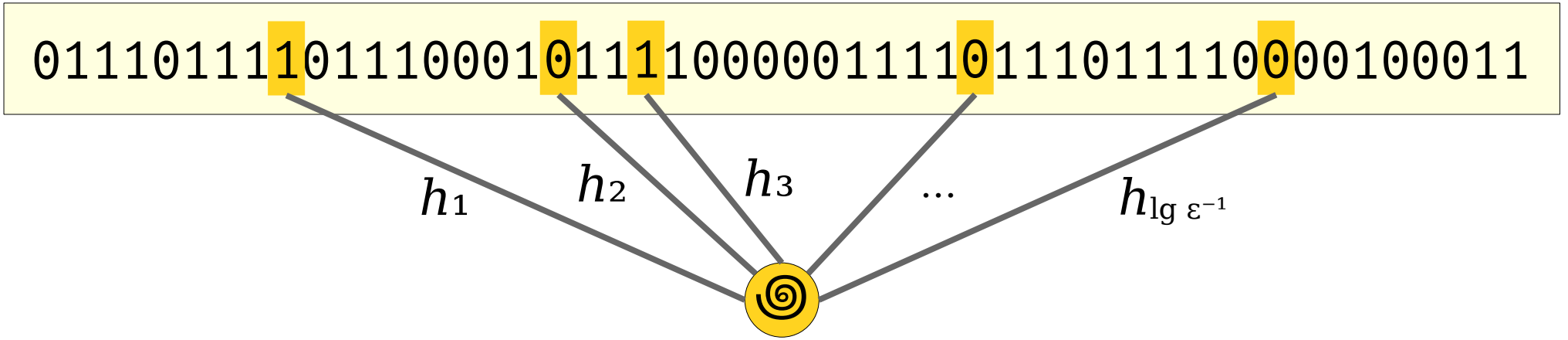
  Hash an item to multiple positions in an array, aggregate those array positions together, and see whether you get what you want.

- From the cuckoo filter, we have the following idea:

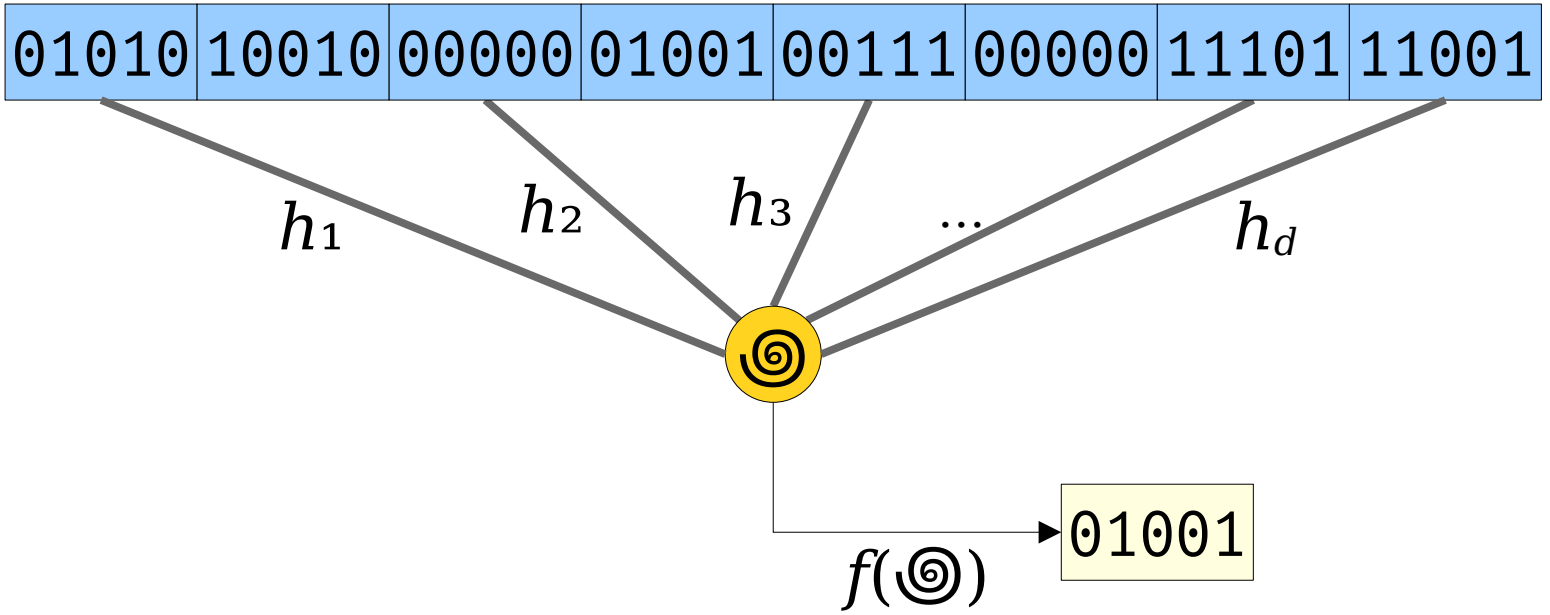
  Store a medium-sized array of medium-sized items.

  Hash each item to a fingerprint, then store the fingerprints in a space-efficient manner.

- What happens if we combine these ideas together?



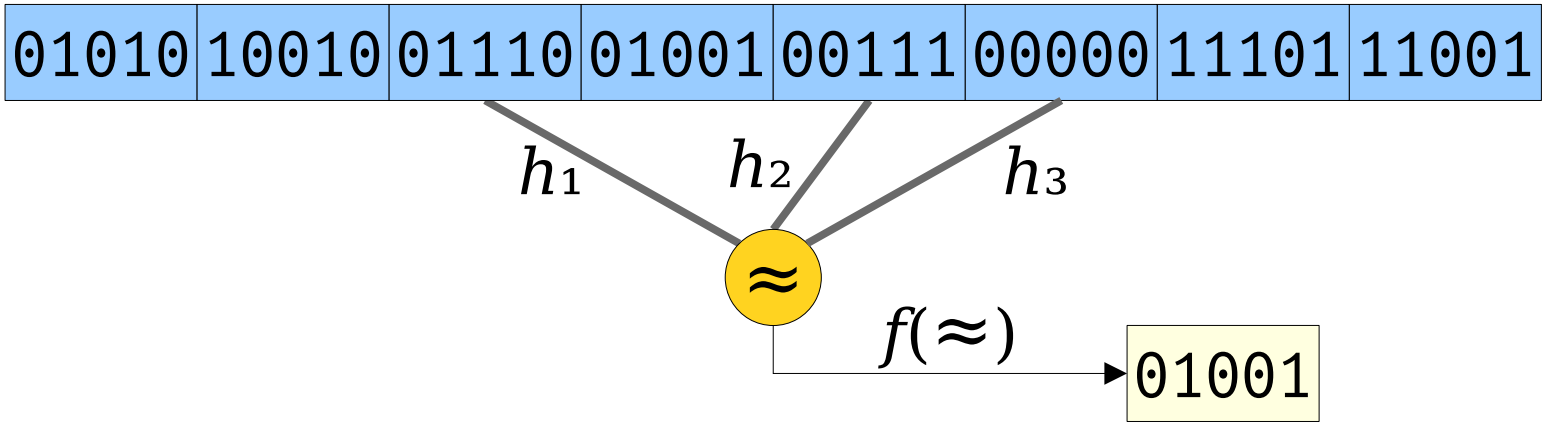
	Bloom Filters	
Store an array of items	Array of bits	
Pick some array slots	Hash item with $\lg \varepsilon^{-1}$ hash functions	
Derive single value from slots	AND	
Compare value against expected	Should be 1	



	Bloom Filters	Something New
Store an array of items	Array of bits	Array of L-bit values
Pick some array slots	Hash item with $\lg \varepsilon^{-1}$ hash functions	Hash item with d hash functions
Derive single value from slots	AND	XOR
Compare value against expected	Should be 1	Should be item fingerprint

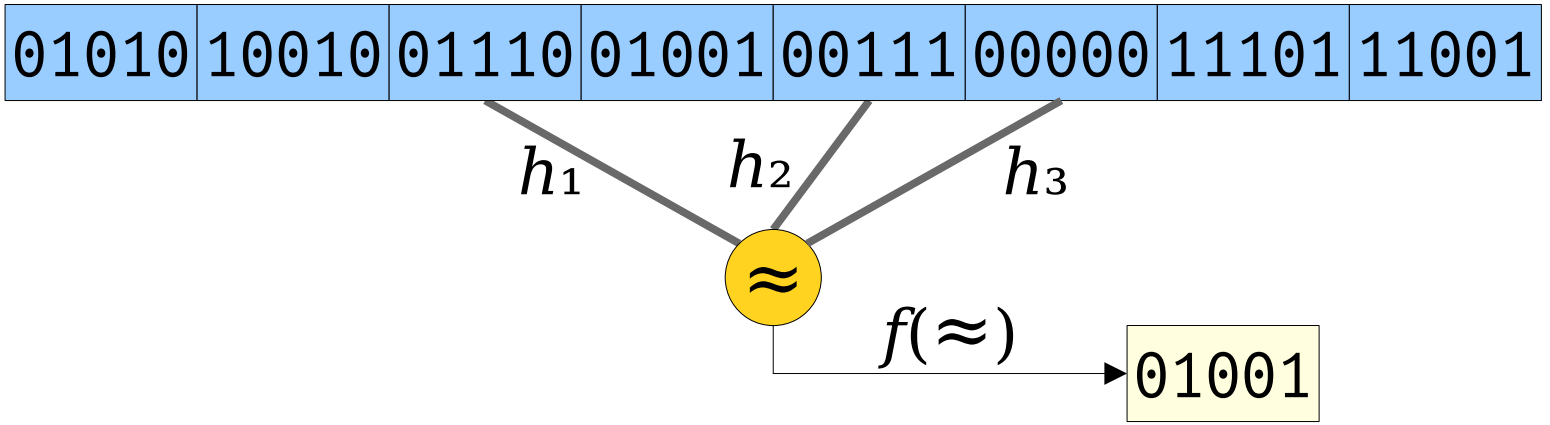
The XOR Filter

- Create an array of $\Theta(n)$ slots, each of which stores a single L-bit number.
- Pick d hash functions h_1, h_2, …, h_d from items to slots.
- Pick a hash function f from items to L-bit fingerprints
- To query whether an item x is in the set:
  - Compute $h_1(x)$, $h_2(x)$, …, $h_d(x)$ and look at those slots.
  - XOR the contents of those slots together.
  - Return whether the XORed value matches $f(x)$.



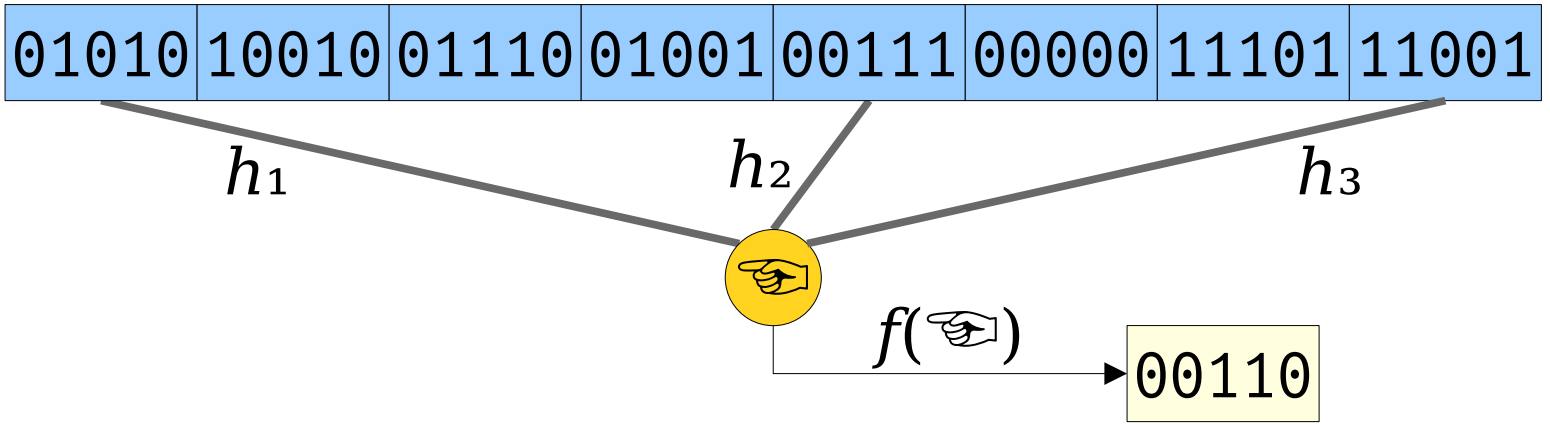
The XOR Filter

- Questions to address:
  - How do we pick L, the fingerprint length?
  - How big should our array be?
  - How do we initialize the array contents?
  - How many hash functions should we use?
- Let's go through each of these in turn.



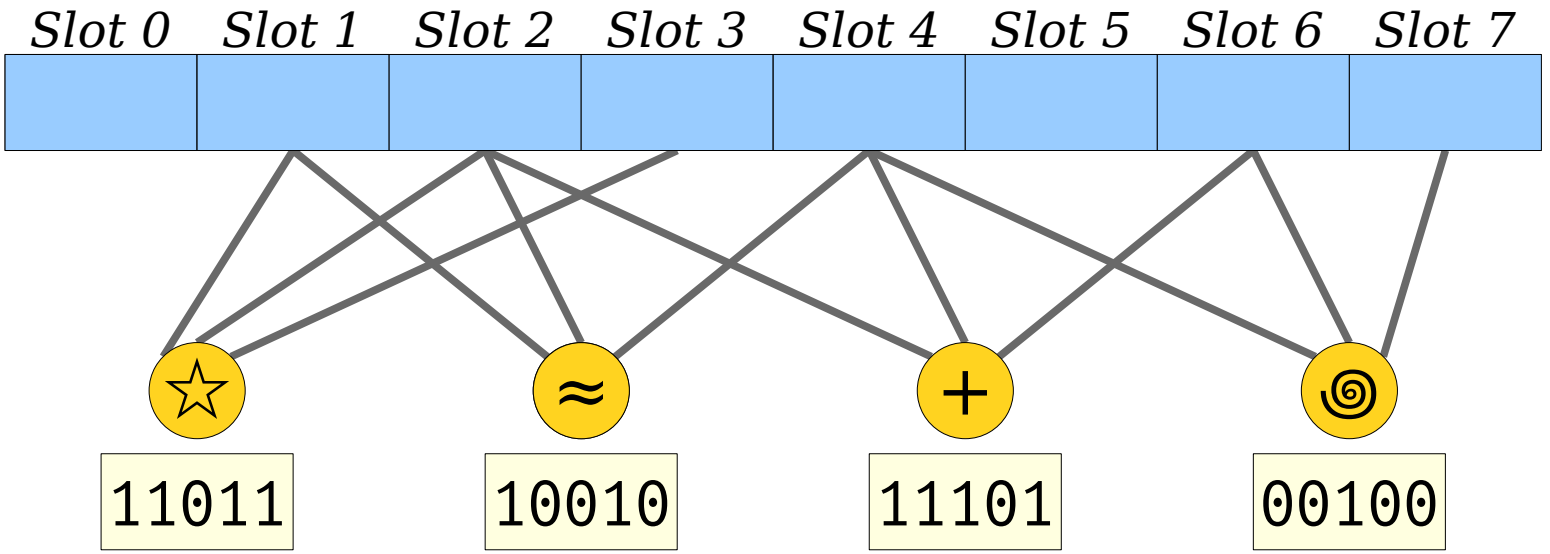
The Fingerprint Size

- Take any item $x \notin S$.
- We hash x to some table locations, then XOR all those locations together to get a value $T[h_1(x)] \oplus \ldots \oplus \ldots T[h_d(x)]$.
- We also compute the fingerprint $f(x)$.
- For simplicity, assume that all hash functions here (h_1, …, h_d and f) are truly random. This makes $f(x)$ independent of the computed value $T[h_1(x)] \oplus \ldots \oplus \ldots T[h_d(x)]$.
- Probability of a false positive is therefore the probability that $f(x) = T[h_1(x)] \oplus \ldots \oplus \ldots T[h_d(x)]$, which is 2^{-L}.
- Setting $L = \mathbf{\lg \varepsilon^{-1}}$ then ensures an appropriate false positive rate.



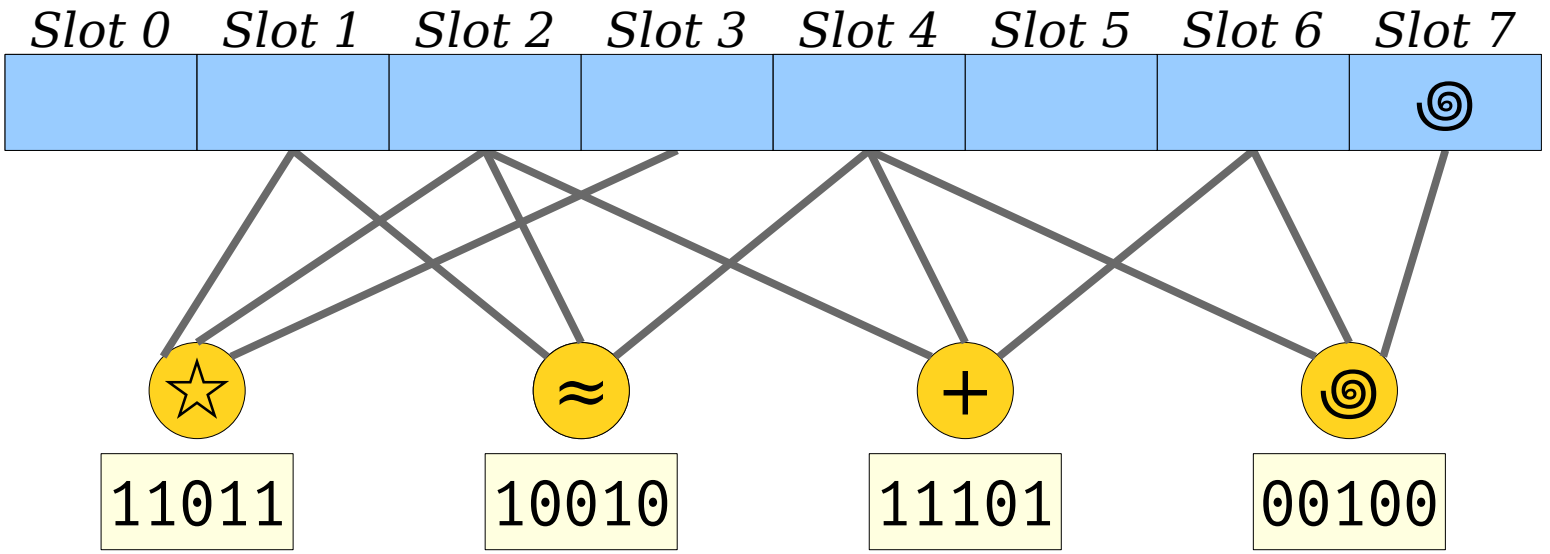
Filling Our Table

- Here's a concrete example of a table to fill in. Lines between items and slots indicate where each item hashes.
- Is it possible to set the bits of this table in a way that makes everything work out?



Filling Our Table

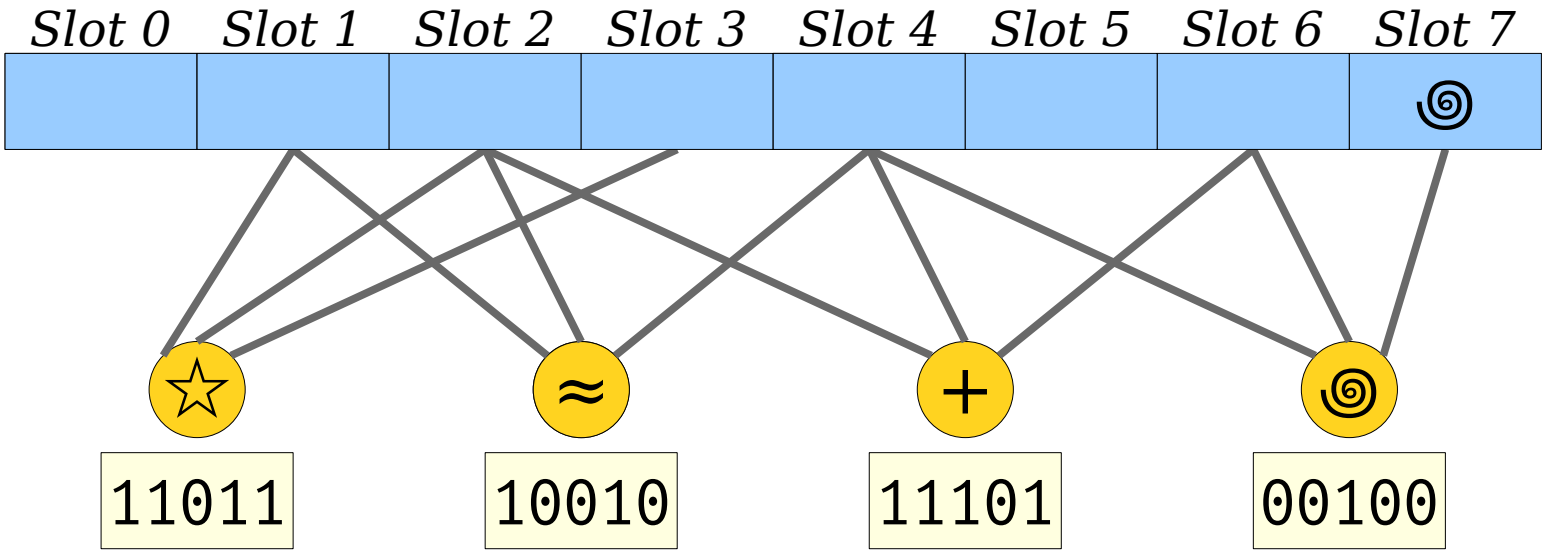
- Notice that slot 7 has only one item (namely, ◎) hashing to it.
- This means that tuning slot 7 can only impact whether ◎ has the correct XOR of its slots. It has no impact on any other items.
- ***Idea:*** "Assign" ◎ to slot 7.



Filling Our Table

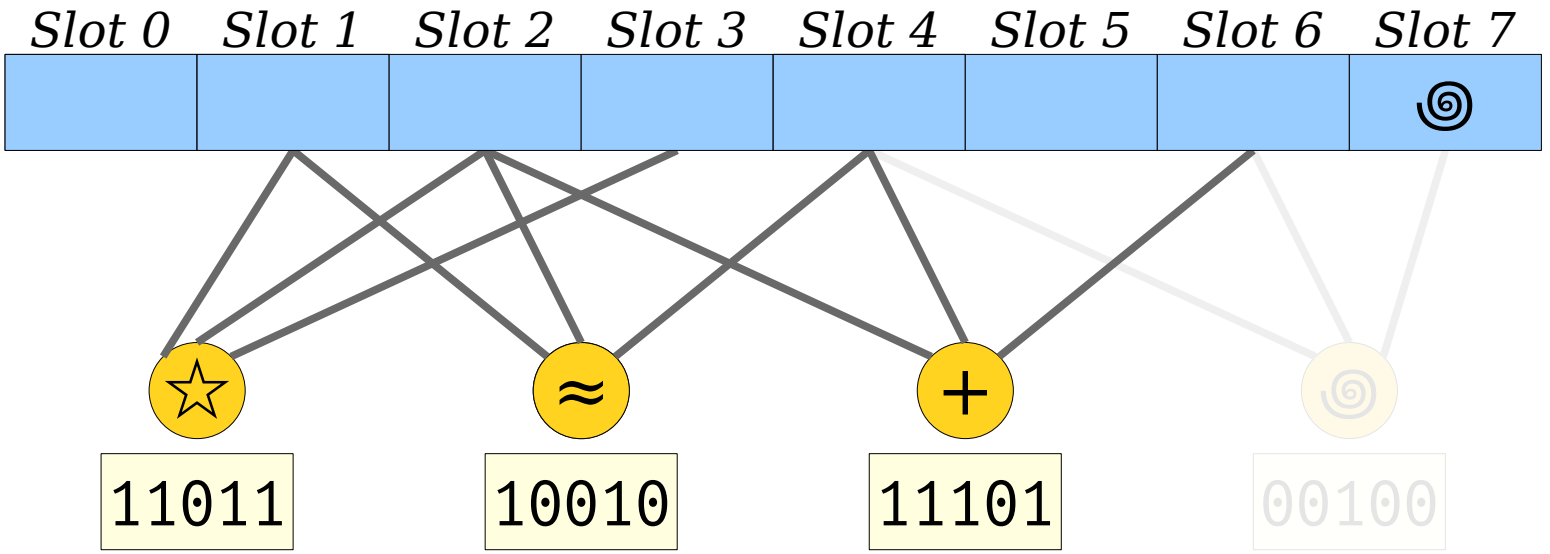
- ***Claim:*** Regardless of the values of the other slots, we can set slot 7's value so that T[4] ⊕ T[6] ⊕ T[7] = `00100`.
- Specifically, set T[7] = T[4] ⊕ T[6] ⊕ `00100`.

$$
\begin{aligned}
T[4] \oplus T[6] \oplus T[7] &= T[4] \oplus T[6] \oplus (T[4] \oplus T[6] \oplus \texttt{00100}) \\
&= (T[4] \oplus T[4]) \oplus (T[6] \oplus T[6]) \oplus \texttt{00100} \\
&= \texttt{00100}.
\end{aligned}
$$



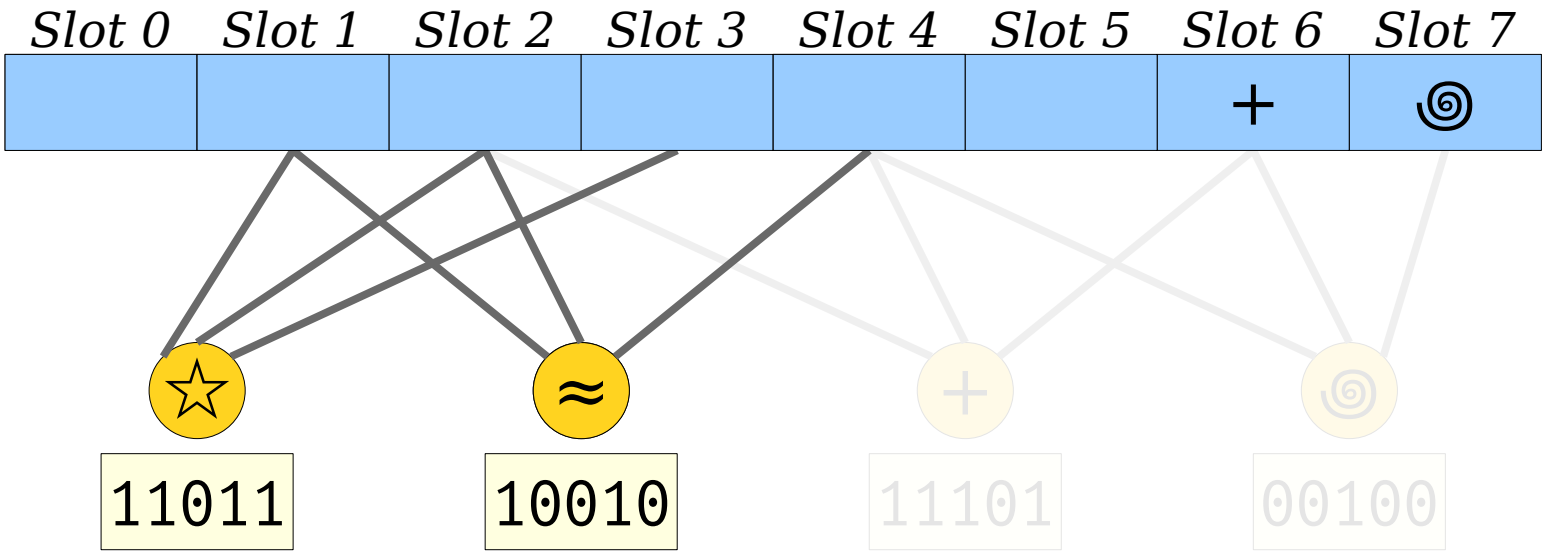
Filling Our Table

- Because we've "claimed" slot 7 for 🌀, we can worry about tuning slot 7 for 🌀 after we tune all the other slots.
- ***Idea:*** Remove 🌀 from the set of items to place. Recursively place everything else, then assign slot 7 so 🌀's XORs pan out.



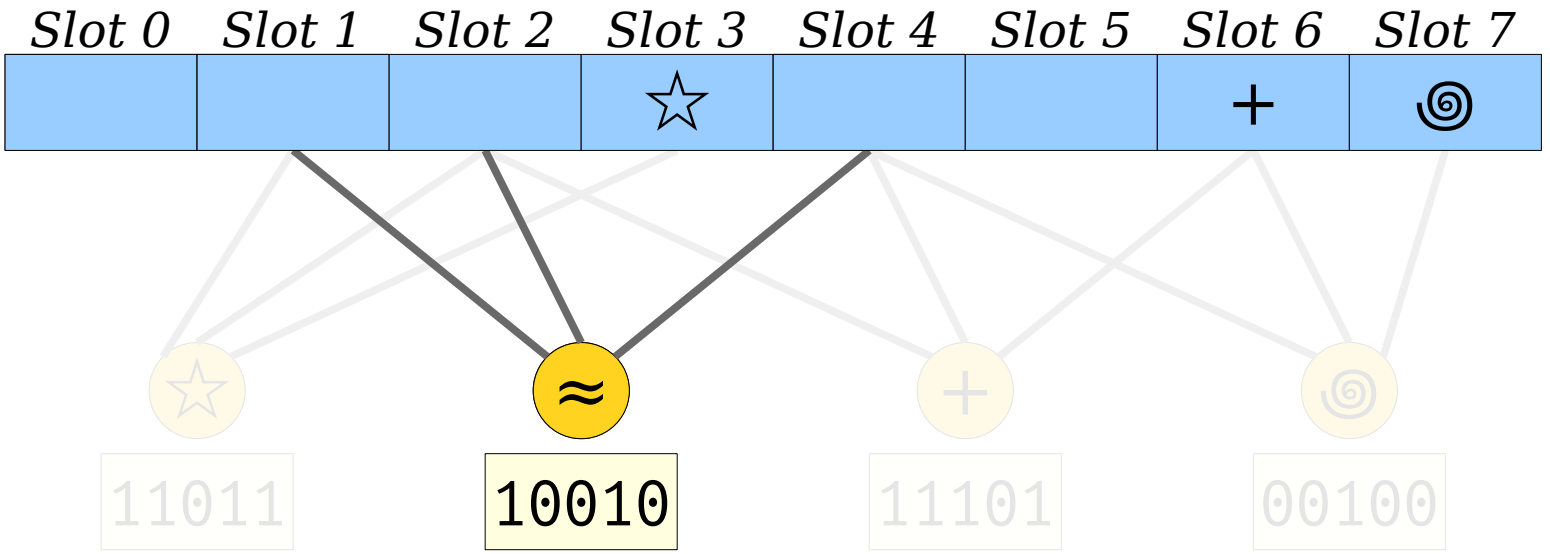
Filling Our Table

- After removing ◎, notice that + is now the only item that hashes to slot 6.
- As before, we'll "assign" + to that slot. Regardless of what goes in slot 2 or slot 4, we can always tune slot 6 so everything XORs out appropriately.



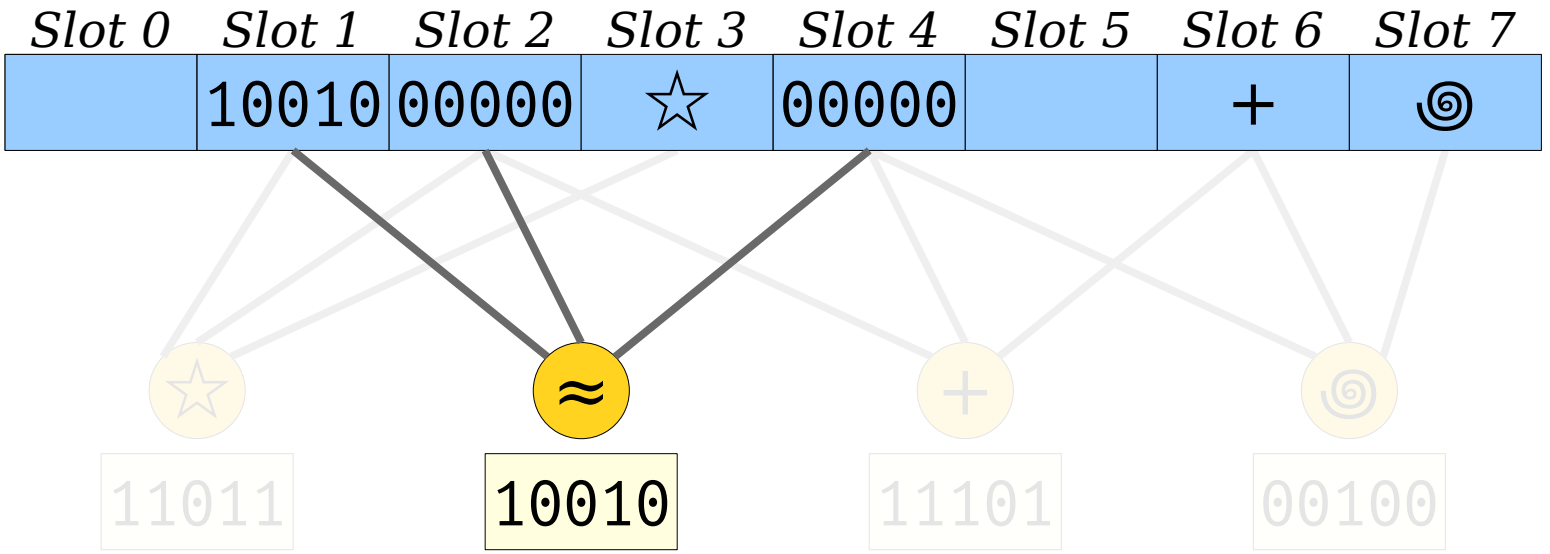
Filling Our Table

- Now, ☆ has sole ownership of slot 3, so we can assign it there.
- As before, once everything else is placed, we can tune slot 3 to make everything XOR out properly.



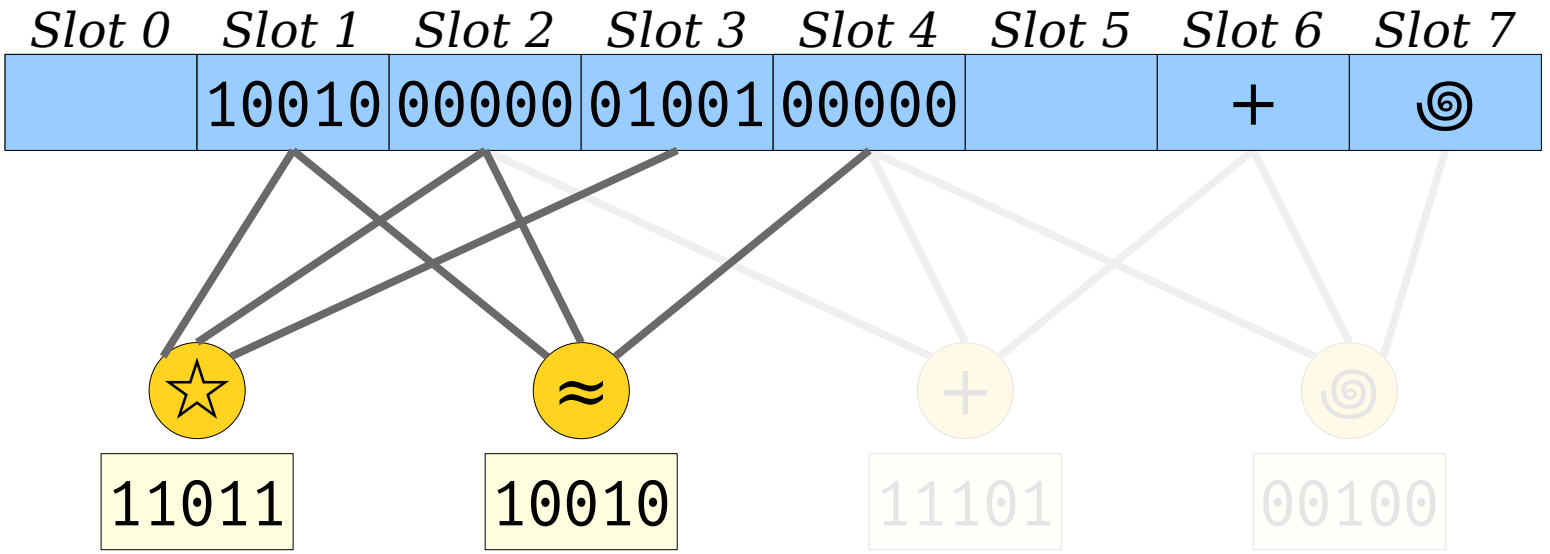
Filling Our Table

- We're left with just ≈ and no other items to place. We can set the values of these slots however we'd like, as long as they XOR to ≈'s fingerprint (10010).
- A simple option: Put 10010 in one of them and zeros in the others.



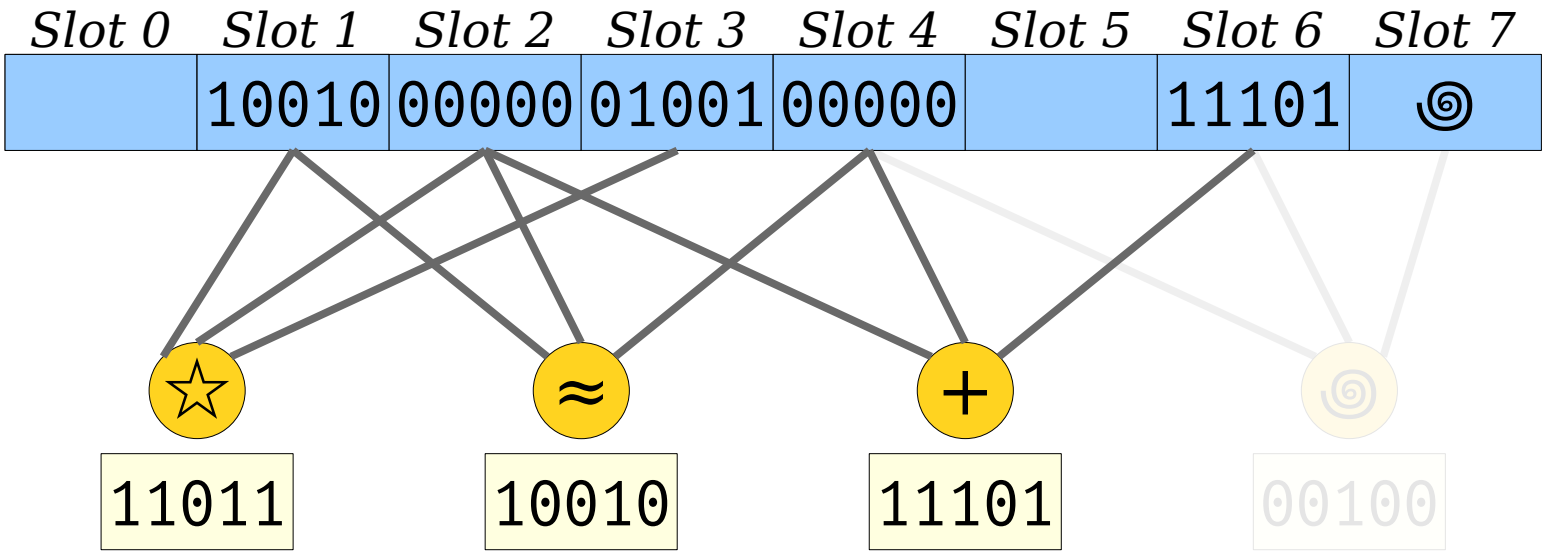
Filling Our Table

- The last item we removed was ☆, so let's add that back in.
- We now set slot 3 to the XOR of ☆'s fingerprint (11011) and the values in its other two slots.



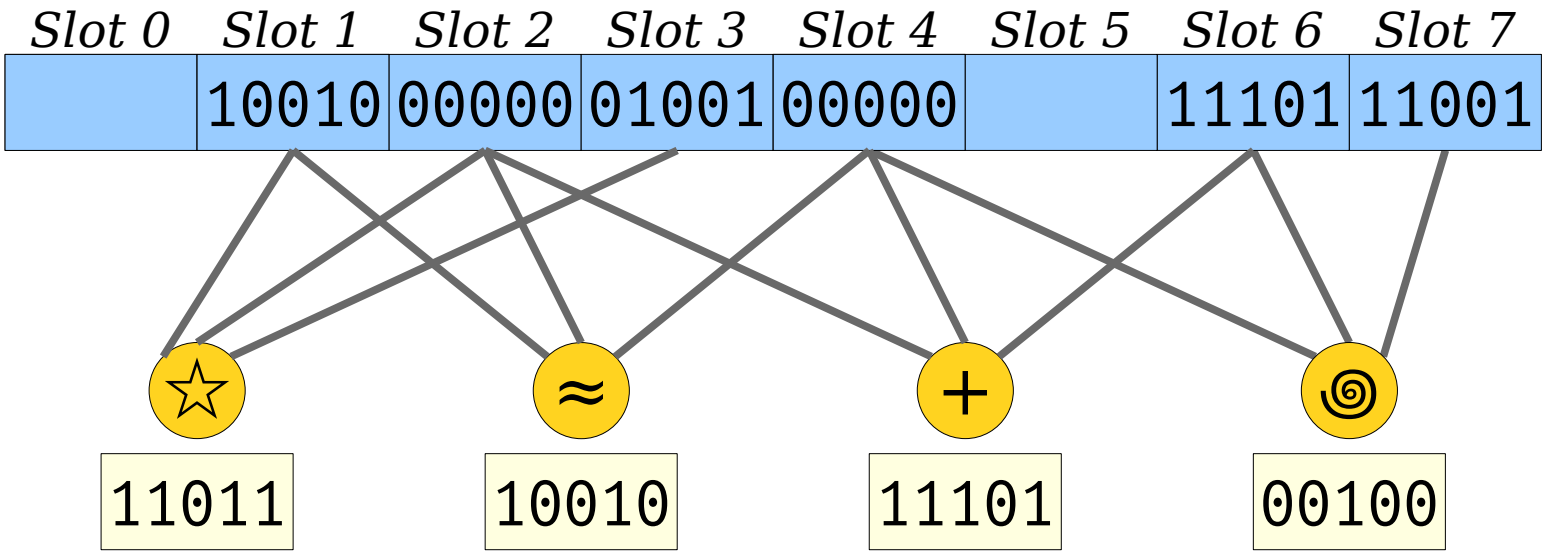
Filling Our Table

- Let’s add the + back in now.
- We’ll set slot 6 to the XOR of slot 2, slot 4, and +’s fingerprint (11101).



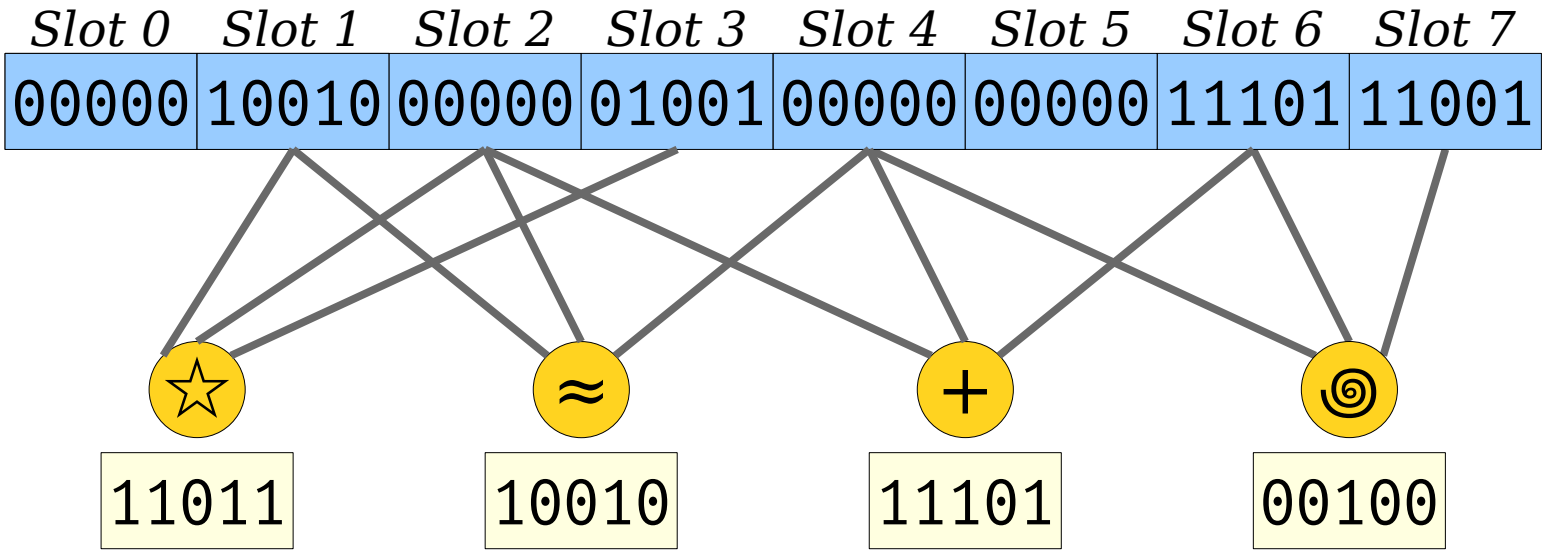
Filling Our Table

- Finally, we add 🌀 back in.
- We set slot 7 to the XOR of slot 4, slot 6, and slot 🌀's fingerprint (`00100`).



Filling Our Table

- All that's left to do now is set the remaining table entries.
- It doesn't matter what we put here.
  - Items in the table don't use those slots.
  - Items not in the table have random fingerprints, and what we pick here doesn't impact the collision probability.
- ***Simple implementation:*** Just set those values to 0.



Filling Our Table

- Fill the initial table with 0 bits.
- If there are no items to place, there's nothing to do.
- Otherwise:
  - Find an x that hashes to a slot nothing else hashes to.
  - Recursively fill in the table to place the remaining items.
  - Add x back in, setting the value in the unique slot to the XOR of x's other slots and its fingerprint.
- With the right data structures, this can be implemented in time $O(nd + m)$, where n is the number of items, d the number of hashes, and m the number of slots.
- ***Wait... where have we seen this before?***

Filling Our Table

- ***This is the same peeling process as before!***
- Empirically, it seems like with $k_d \cdot n$ slots, we almost always suceed, where
  - ... for $d = 3$, we have $k_d \approx 1.23$.
  - ... for $d = 4$, we have $k_d \approx 1.31$.
  - ... for $d = 5$, we have $k_d \approx 1.44$.
- Space is minimized when we use **$d = 3$** hash functions and a table of size $\approx 1.23n$.

The XOR Filter

- Putting all the pieces together, here's our construction algorithm:
  - Create an array of ≈**1.23*n*** slots, each of which holds a (lg ε^{-1})-bit number.
  - Choose ***three*** random hash functions h_1, h_2, and h_3 from items to slots, plus a fingerprinting function f from items to (**lg ε^{-1}**)-bit hash codes.
  - Initialize the array using the peeling algorithm: find an item that is incident to a slot of degree one, remove it, recursively fill in the rest of the table, then place the item back and put the correct value in the final slot.
- Our query algorithm looks like this:
  - Return whether $T[h_1(x)] \oplus T[h_2(x)] \oplus T[h_3(x)] = f(x)$.

The Story So Far

- Our XOR filter is strictly better than a Bloom filter in terms of space usage, both practically and theoretically.
- It requires fewer hash functions whenever $\varepsilon < 1/16$.
- The query procedure has at most three cache misses.
- One drawback: all elements have to be known in advance before the filter can be constructed.
- ***Question:*** Can we do better?

	Bits Per Element	Hashes/Query
Bloom Filter (1970)	$1.44 \lg \varepsilon^{-1}$	$\lg \varepsilon^{-1}$
Cuckoo Filter, $b = 4$ (2014)	$1.02 \lg \varepsilon^{-1} + 3.06$ *(for sufficiently small ε)*	3
XOR Filter (2020)	$1.23 \lg \varepsilon^{-1}$	4

Reducing Space

- The XOR filter uses $1.23n$ table slots to hold n items.
- ***Recall:*** This is because the peeling algorithm stops working for smaller tables.
- This is an inherent property of the peeling algorithm, not something particular about our implementation.
- ***Question 1:*** Why does peeling start failing around that threshold?
- ***Question 2:*** Does the answer help us improve our peeling data structures?

Next Type

- ***Hypergraph Peeling***
  - A mathematical model for peeling algorithms.
- ***The Peeling Phase Transition***
  - What's up with $1.23n$?
  - Why is there a phase transition?
- ***Spatial Coupling***
  - Dramatically improving peeling algorithms.