Invertible Bloom Lookup Tables

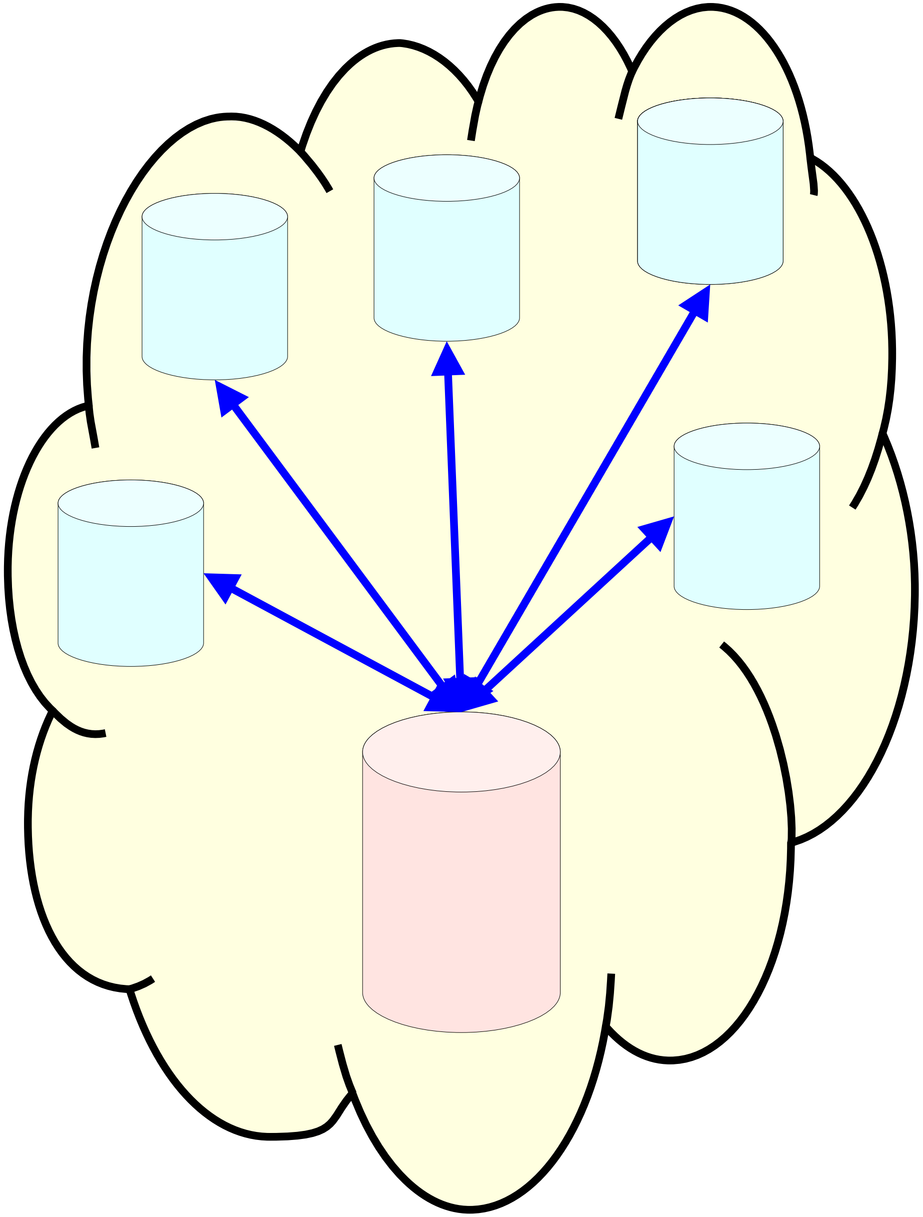
Outline for Today

- ***The Set Reconciliation Problem***
  - A problem in distributed systems.
- ***Invertible Bloom Lookup Tables***
  - A simple, fast, space-efficient solution to set reconciliation.
- ***Hypergraph Peeling***
  - An amazing technique for building data structures.
- ***Analyzing Hypergraph Peeling***
  - A *beautiful* analysis.

Motivation: ***Set Reconciliation***

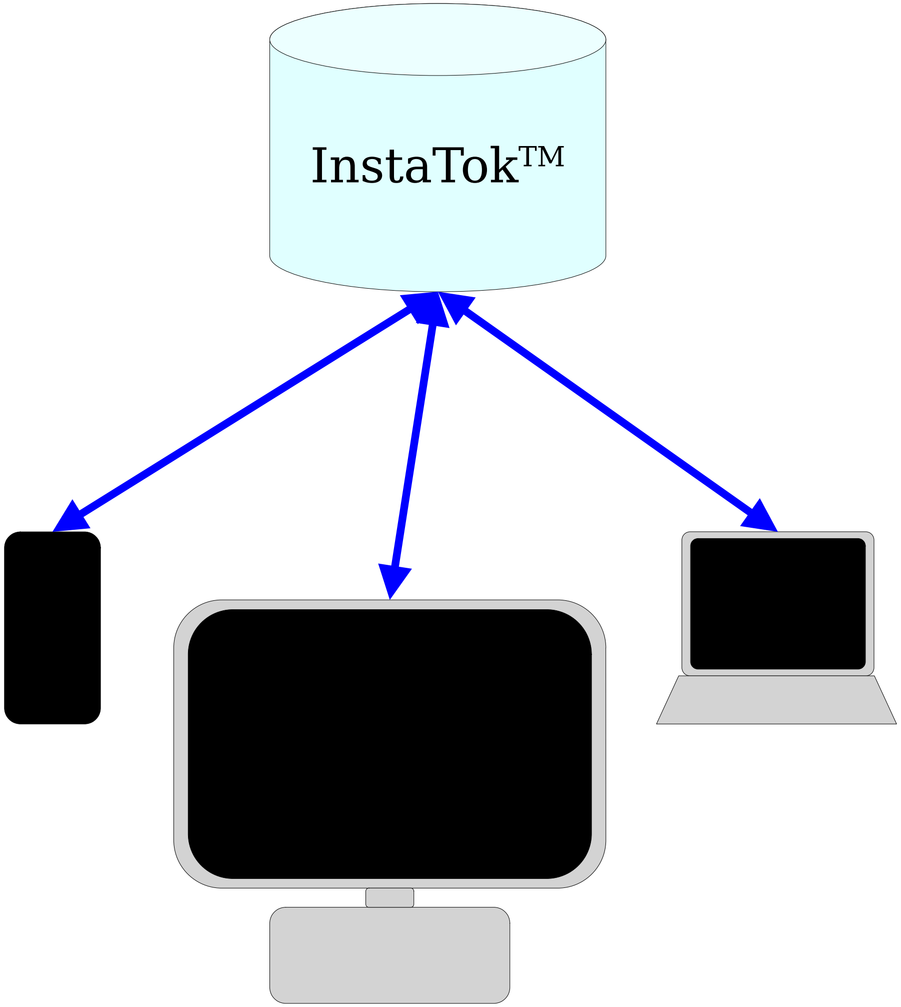
Three Motivating Problems

- We have a central backup server and multiple “frontier” servers.
- Updates happen on the frontiers, and need to be periodically synced to the backup server, but none of the frontier nodes know everything already on the server.
- ***Goal:*** Sync the backup with the frontier servers, using as little network bandwidth as possible.



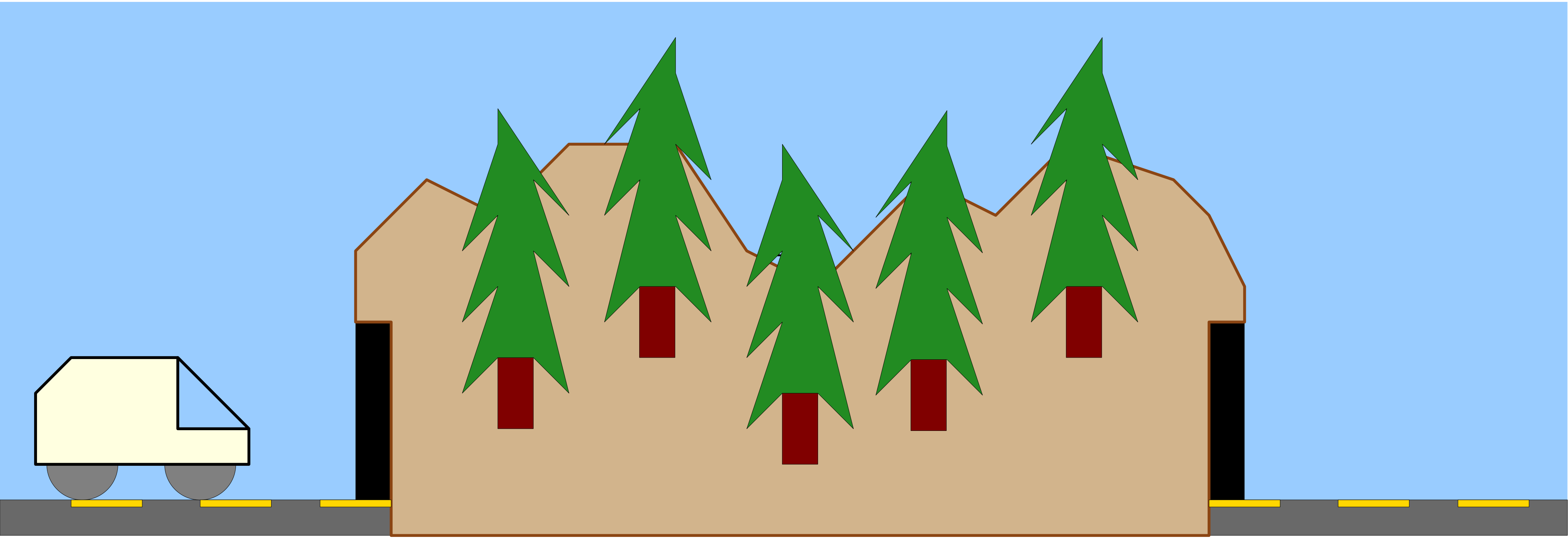
Three Motivating Problems

- 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.



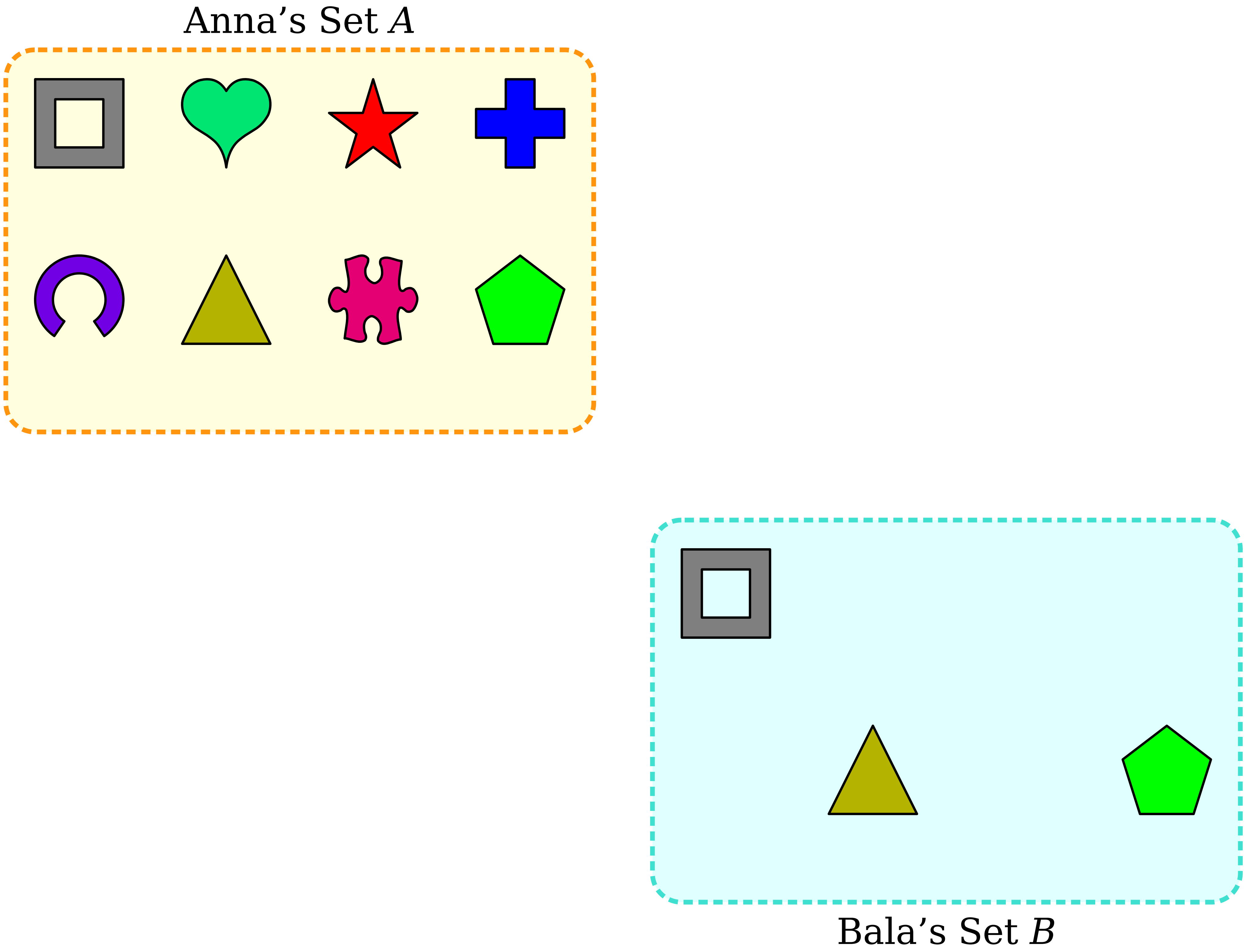
Three Motivating Problems

- 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.

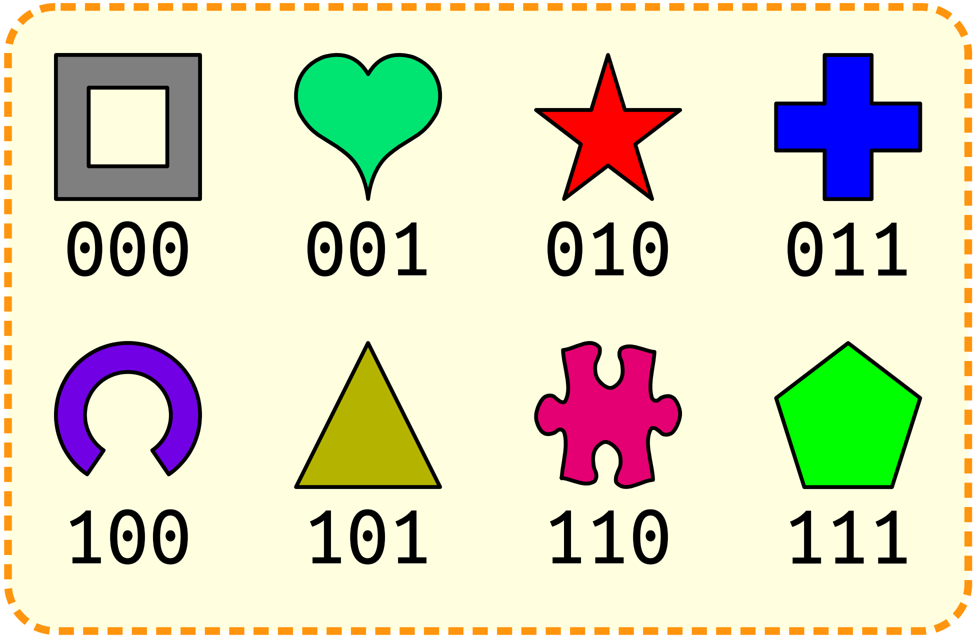


Set Reconciliation

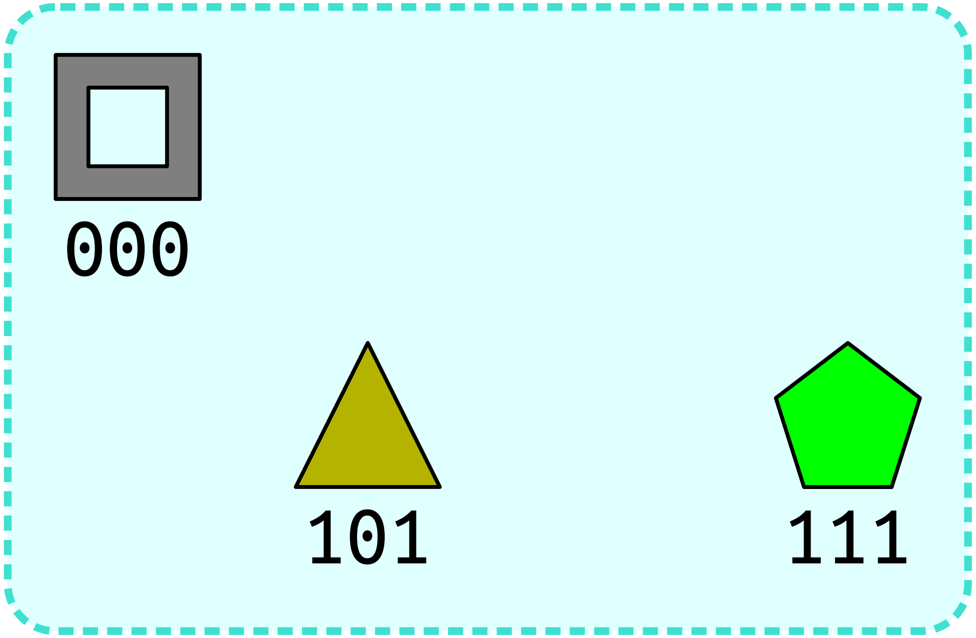
- In the ***set reconciliation problem***, we have two parties each holding sets.
  - Anna holds a set A.
  - Bala holds a set $B \subseteq A$.
- Anna and Bala need to communicate with one another so that Bala ends up holding A.
- ***Goal:*** Solve this problem without requiring Anna and Bala to send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


Anna’s Set A
Bala’s Set B

Anna’s Set *A*

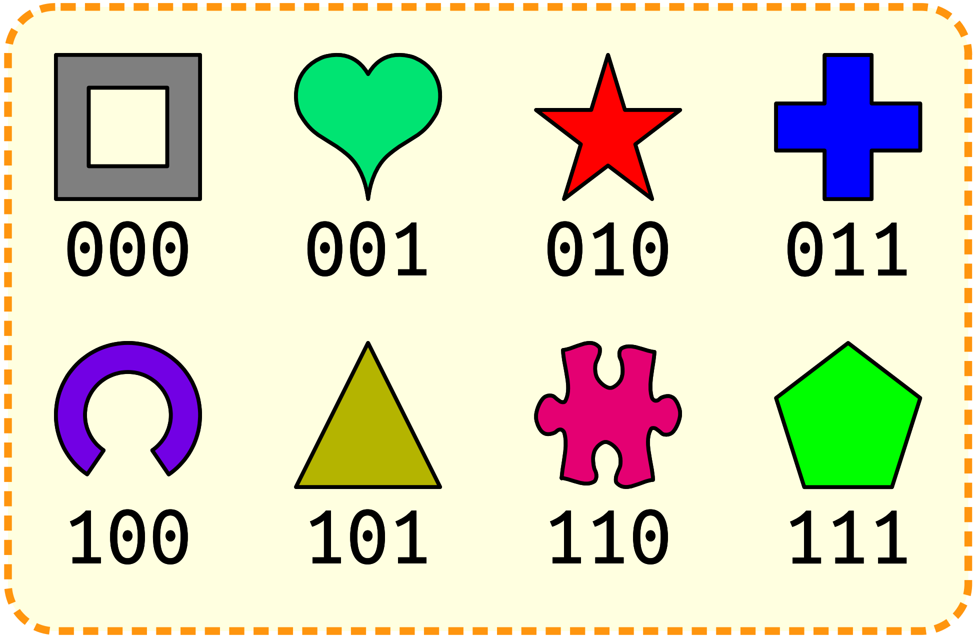


We’ll assume every item is represented as an integer of fixed size. (Say, the SHA-256 hash of the underlying data.)



Bala’s Set *B*

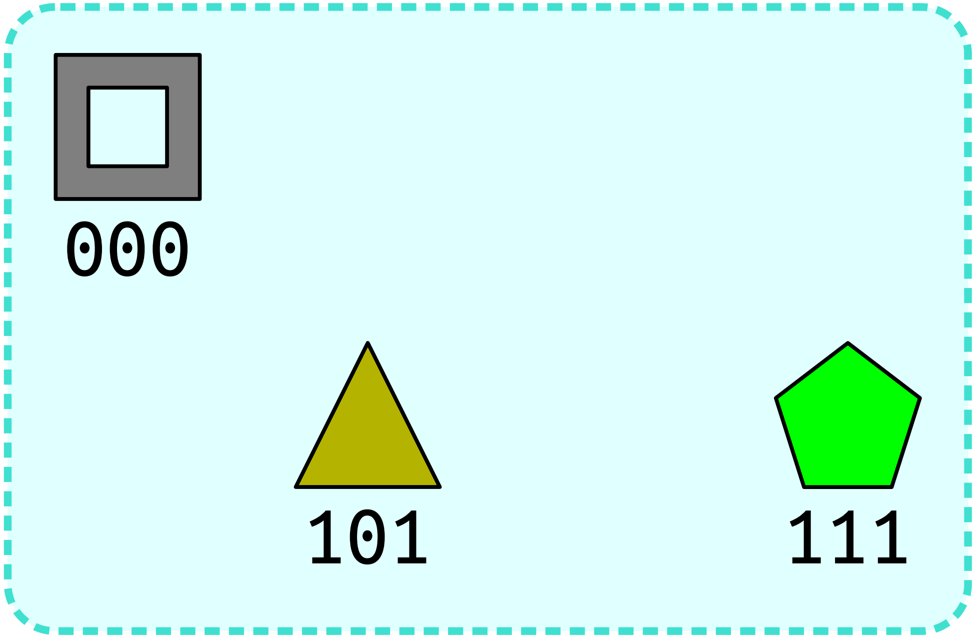
Anna’s Set A



Anna 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

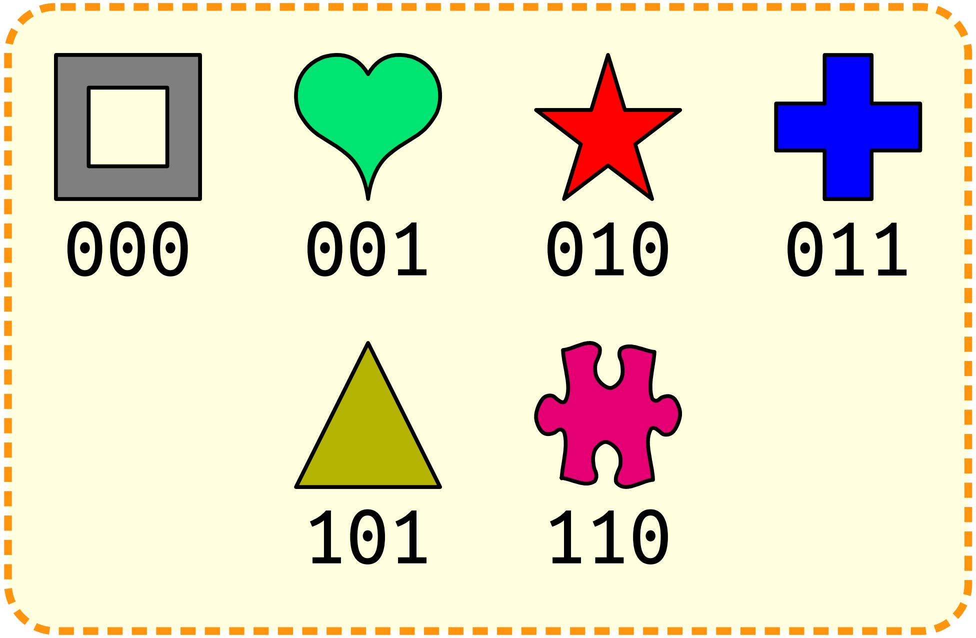
Quantifying our Approaches

- 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 Anna sending everything to Bala?
    - ***Answer:*** $O(n + k)$.
- In practice, k will be 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

Anna's Set A

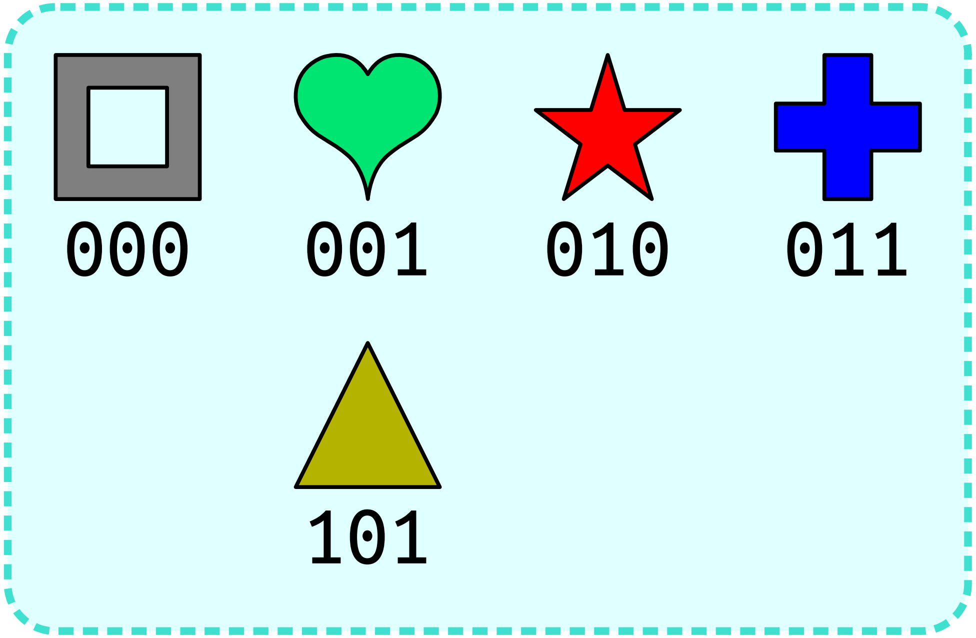


Claim: Anna and Bala can solve this problem while only exchanging O(1) 3-bit integers.

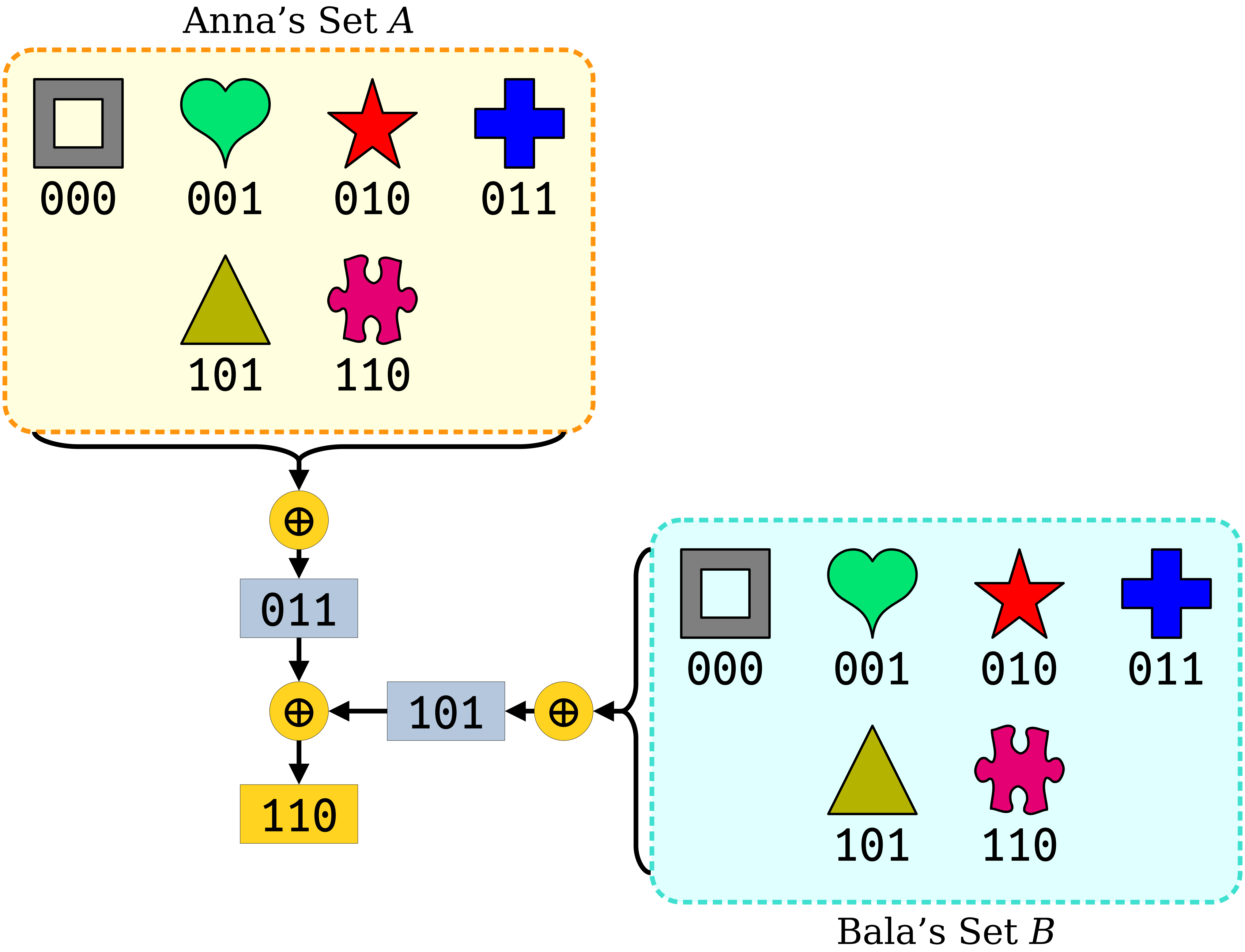
How? Answer at
https://cs166.stanford.edu/pollev

Hint!

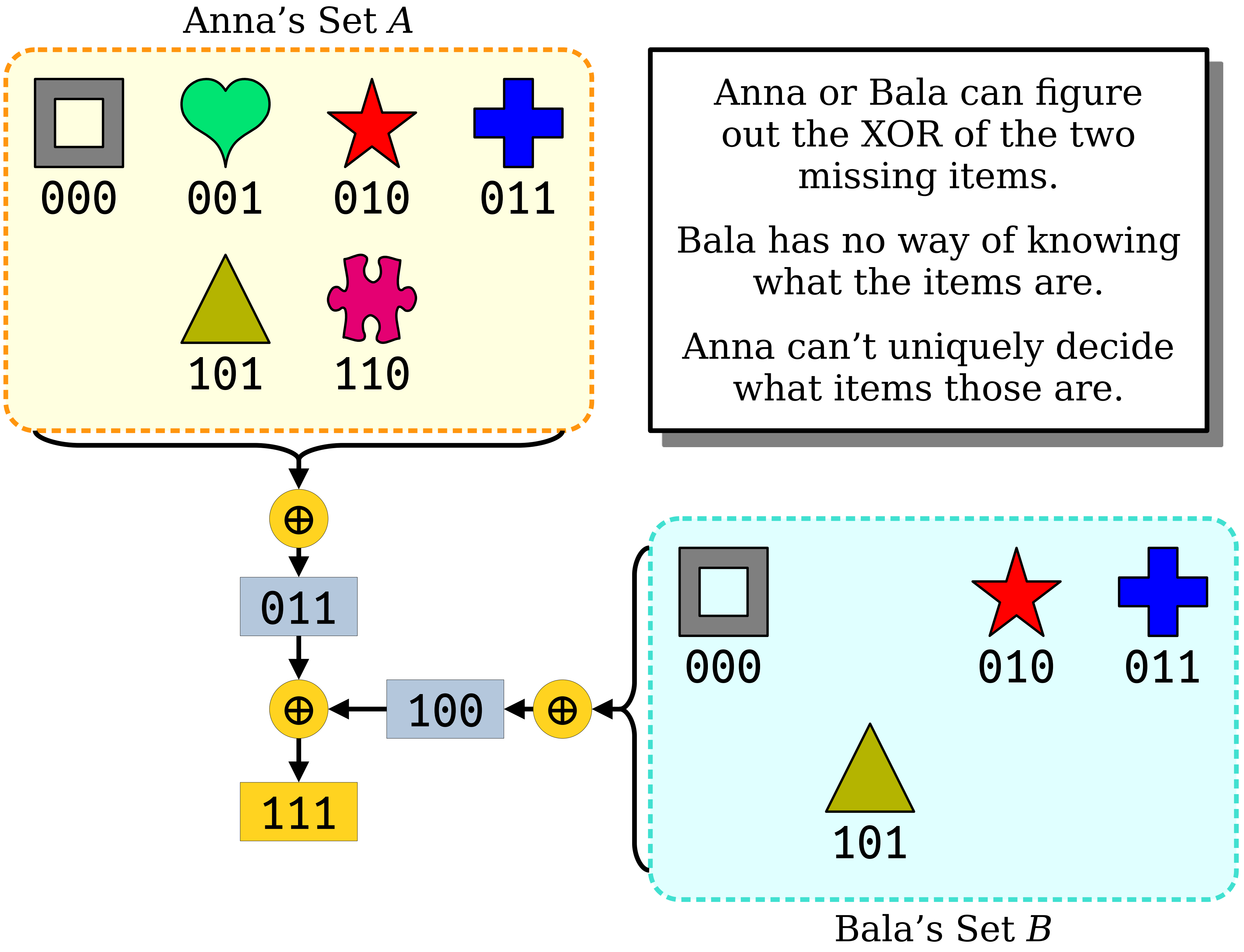
XOR



Bala's Set B


Anna’s Set A
000
001
010
011
101
110
⊕
011
⊕
101
⊕
110
000
001
010
011
101
110
Bala’s Set B

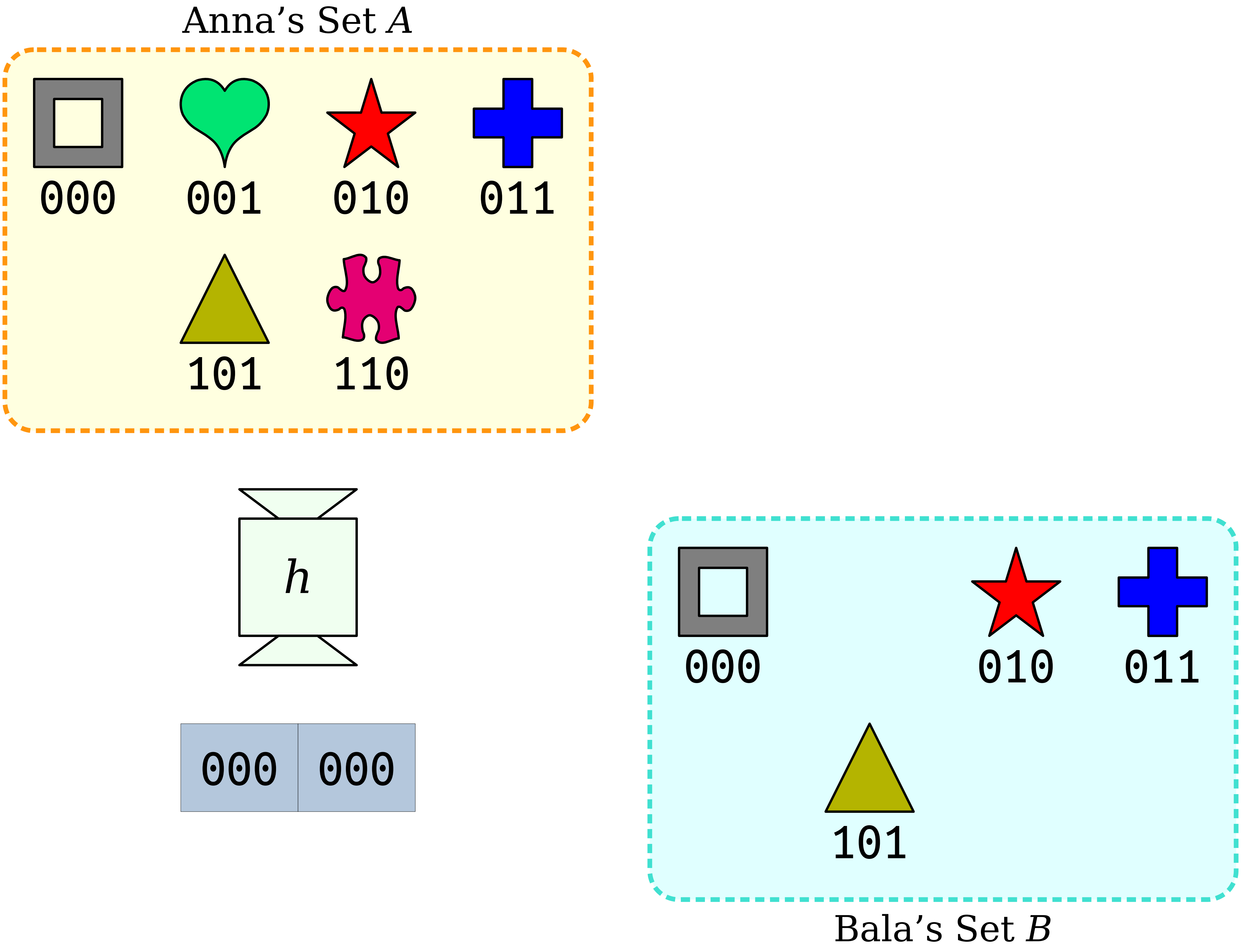
The k=2 Case

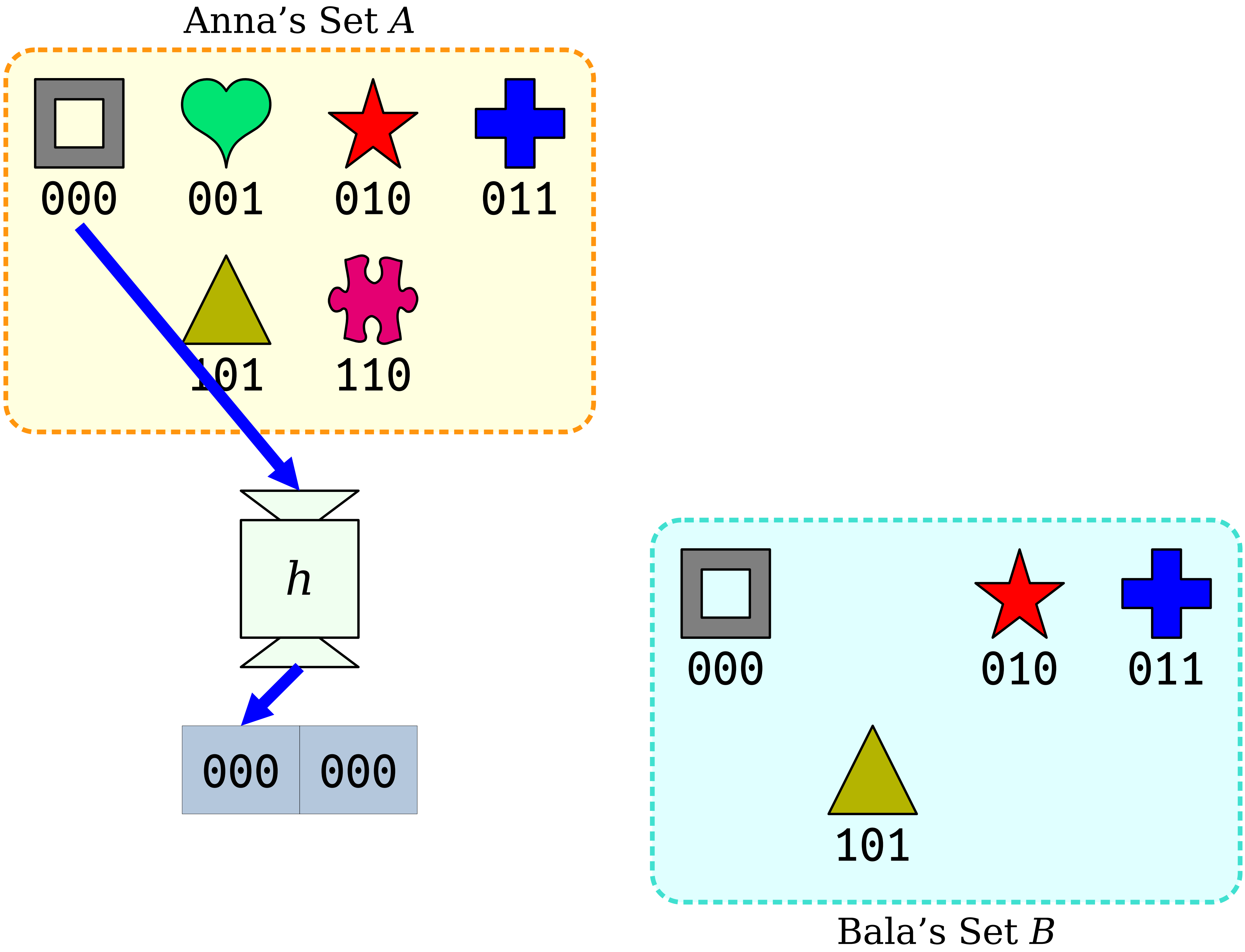


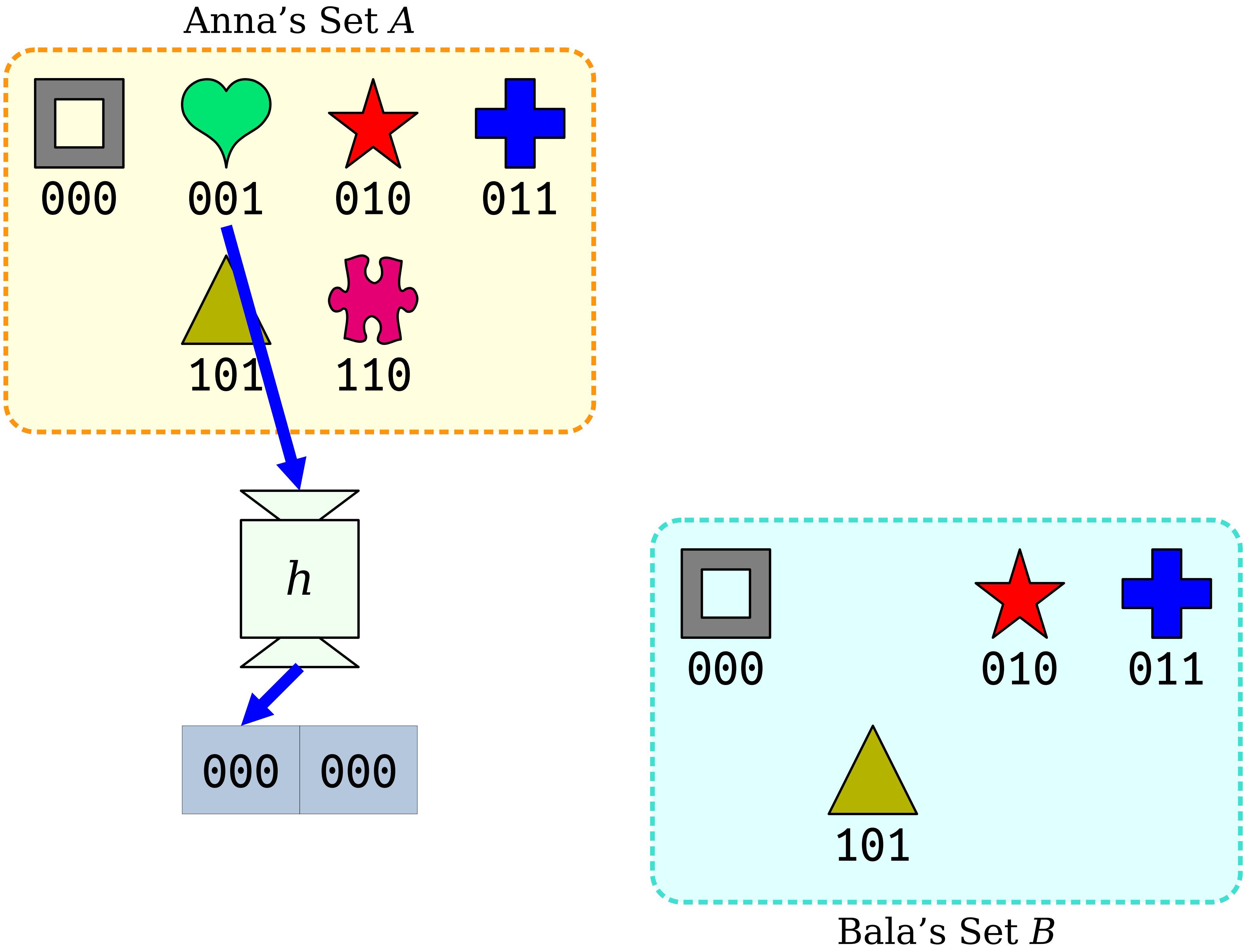
Anna or Bala can figure out the XOR of the two missing items.

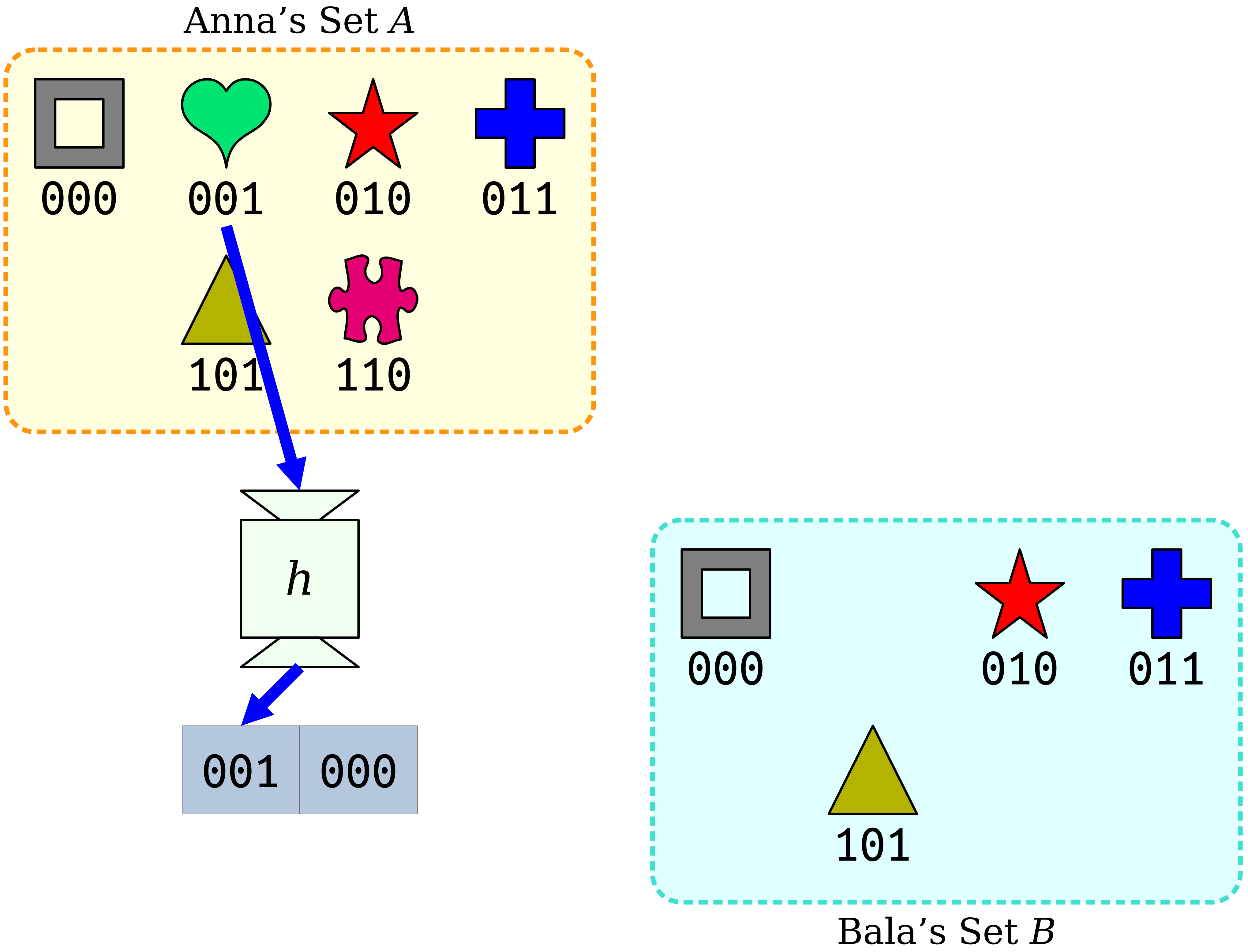
Bala has no way of knowing what the items are.

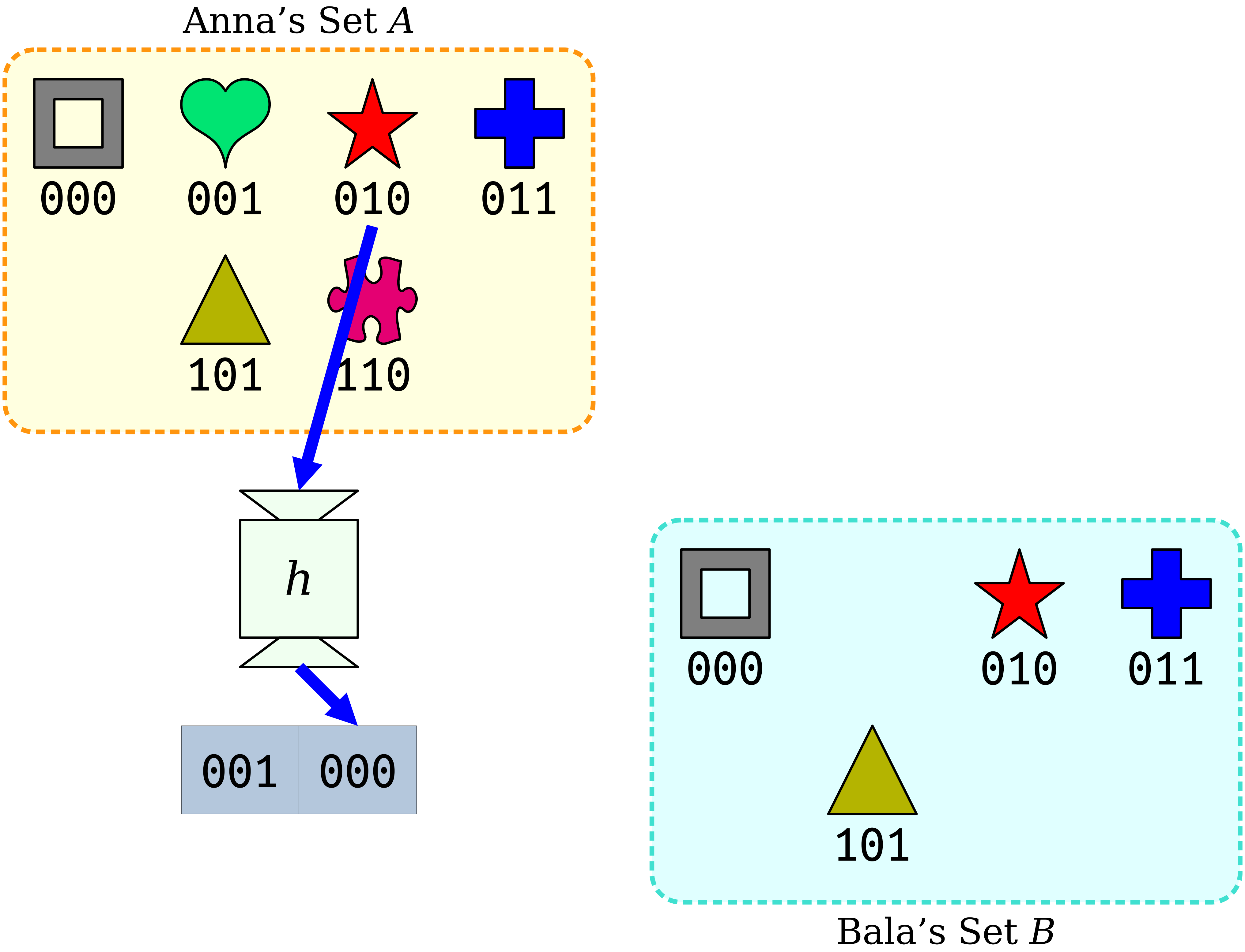
Anna can't uniquely decide what items those are.

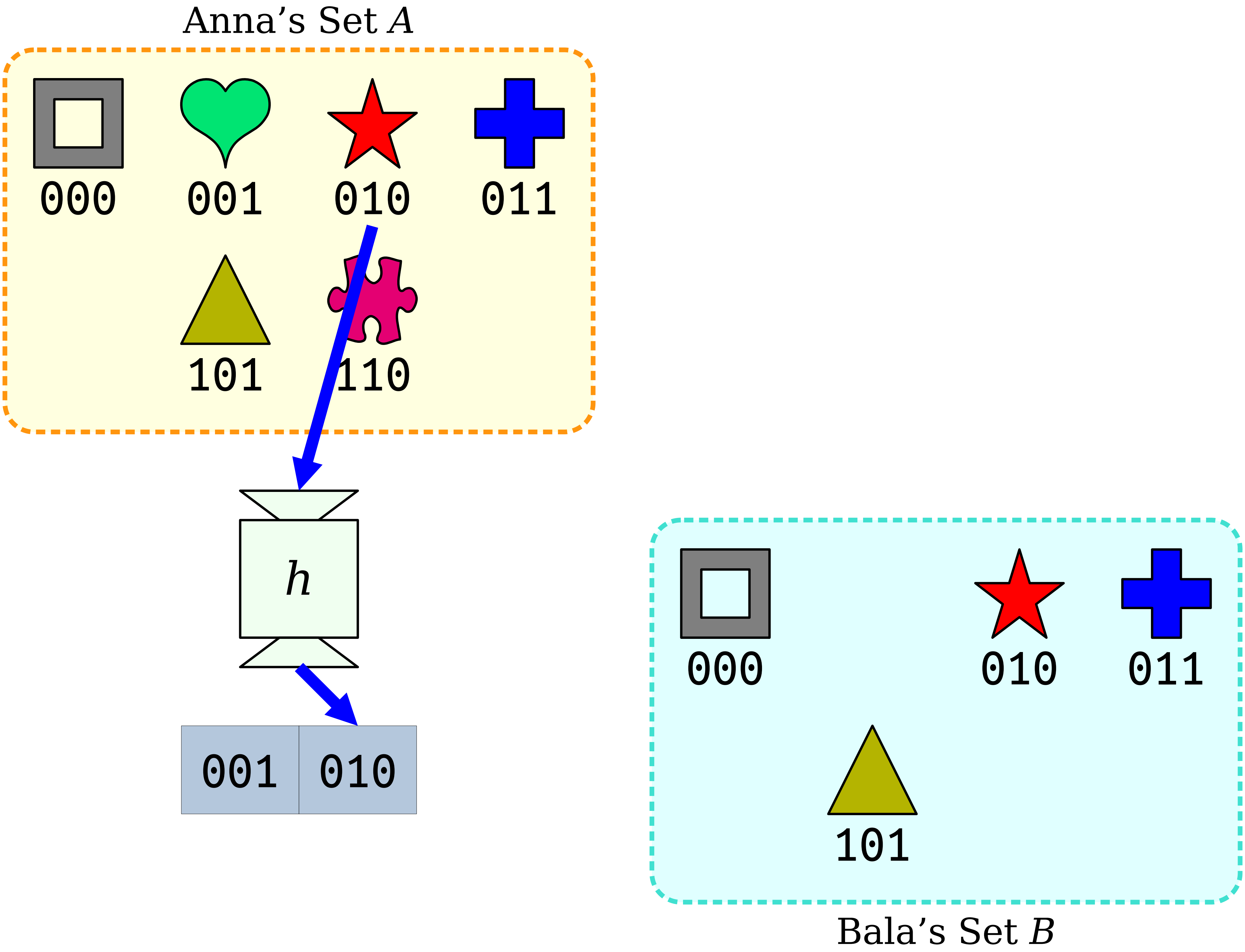

Anna's Set A
000
001
010
011
101
110
h
000
000
000
010
011
101
Bala's Set B

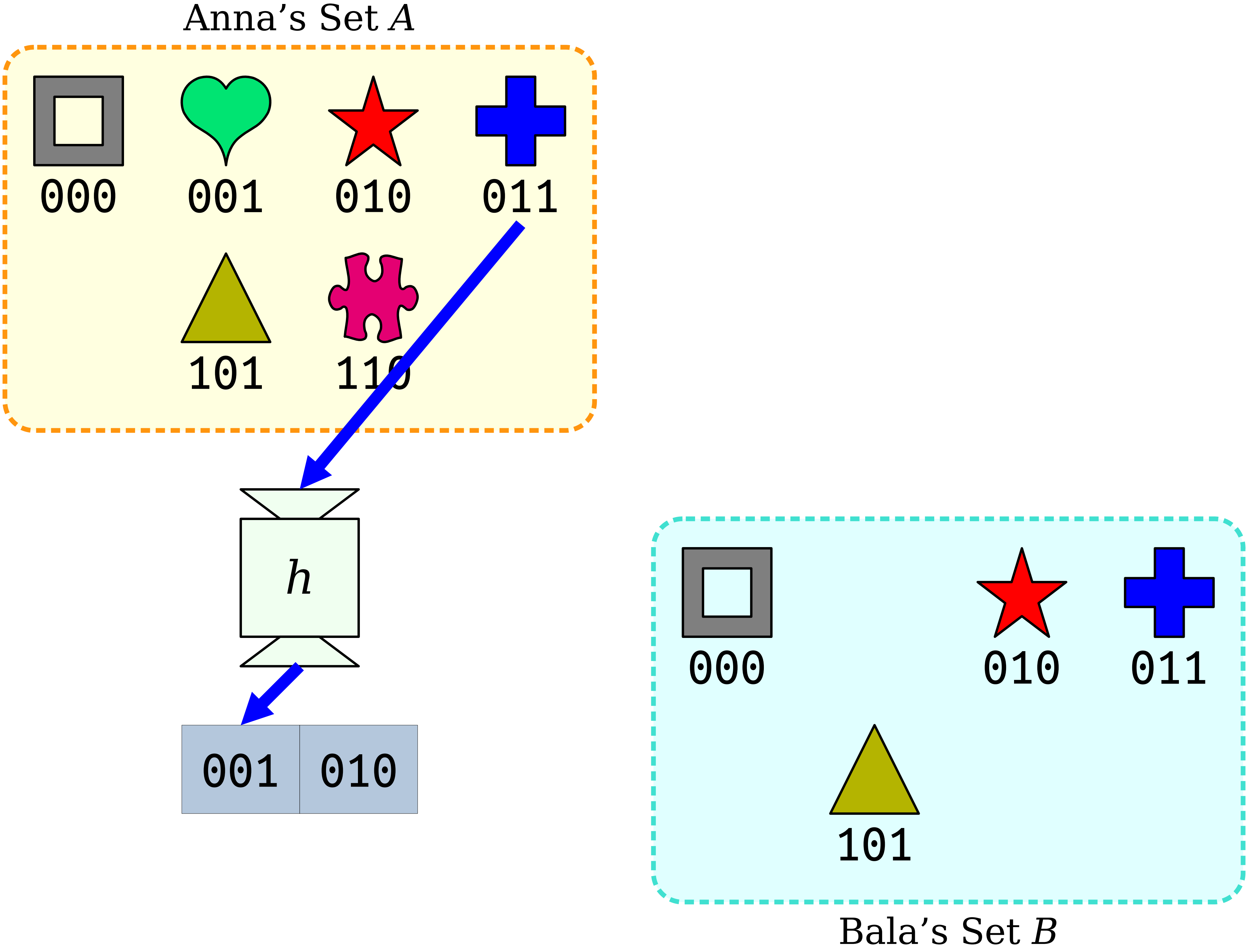

Anna's Set A
000
001
010
011
101
110
h
000
000
000
010
011
101
Bala's Set B

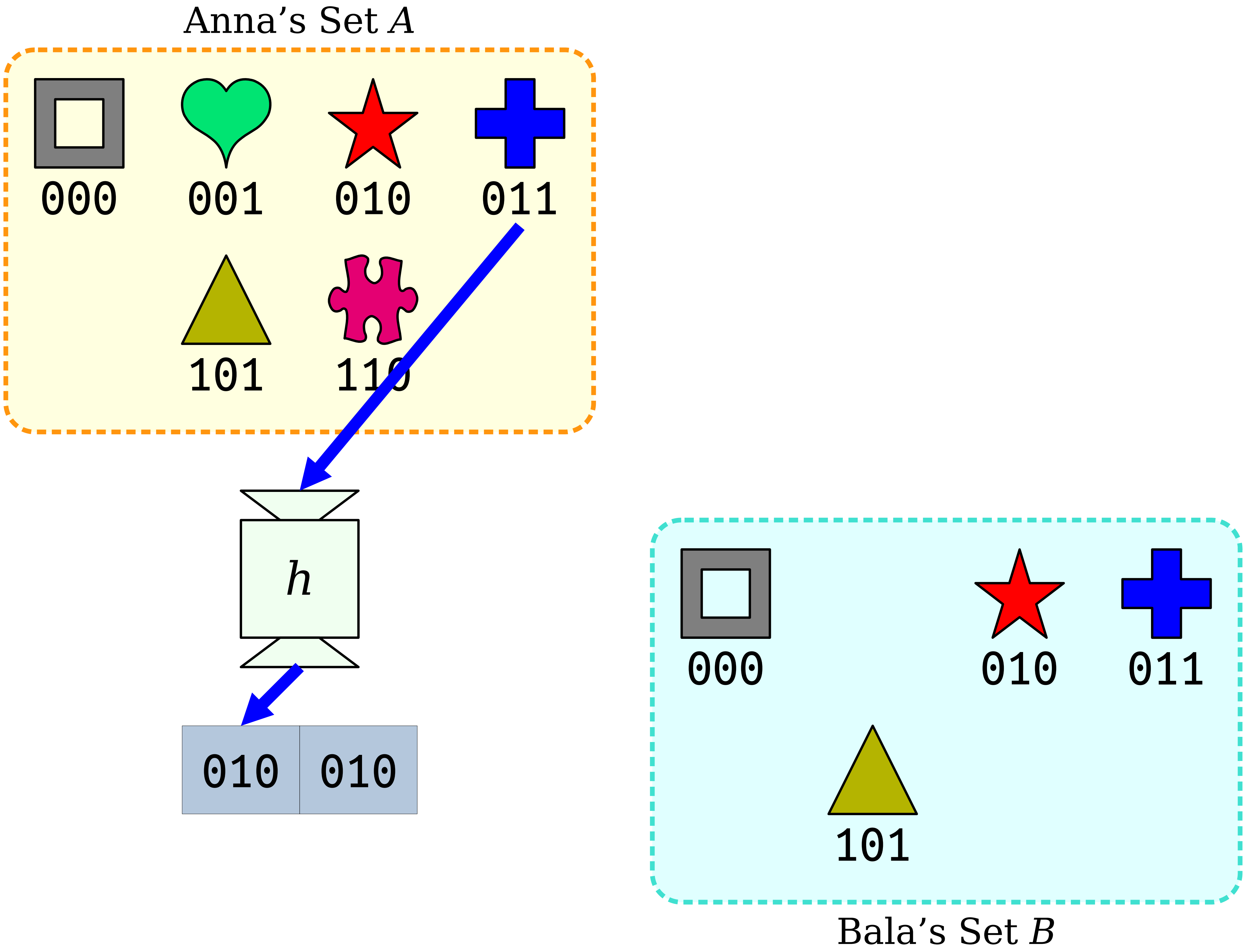

Anna's Set A
000
001
010
011
101
110
h
000
000
000
010
011
101
Bala's Set B

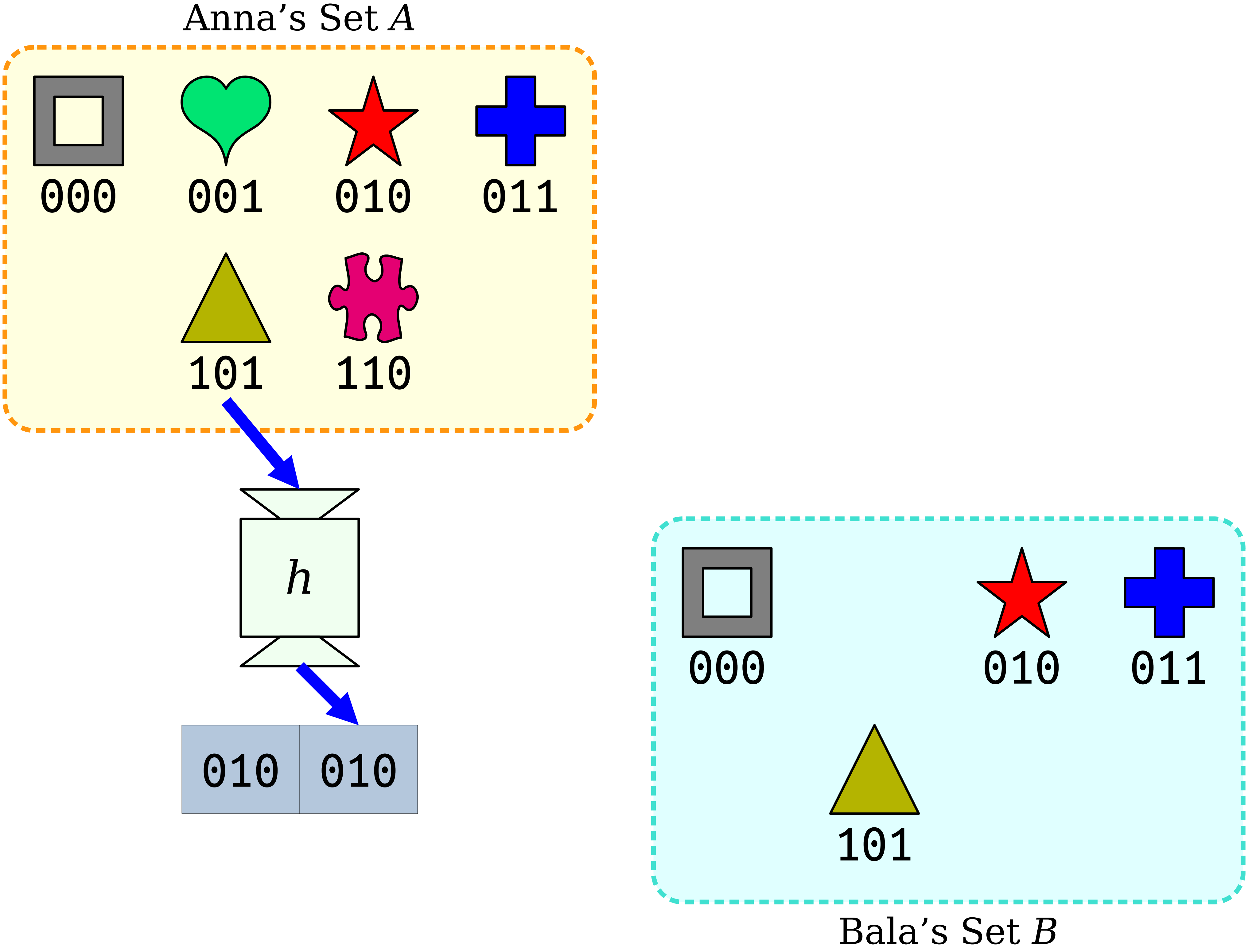

Anna's Set A
000
001
010
011
101
110
h
001
000
000
010
011
101
Bala's Set B

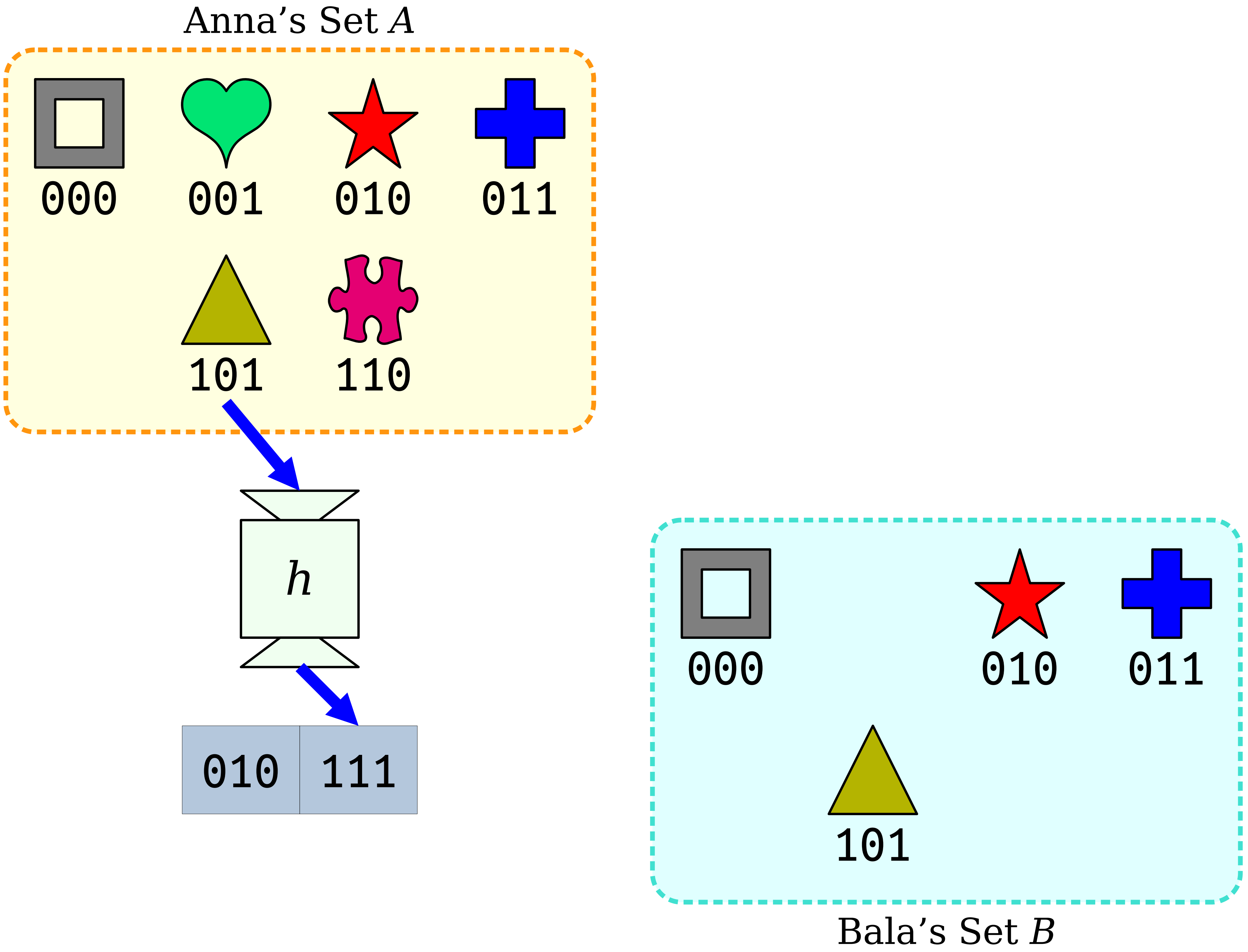

Anna’s Set A
000
001
010
011
101
110
h
001
000
000
010
011
101
Bala’s Set B

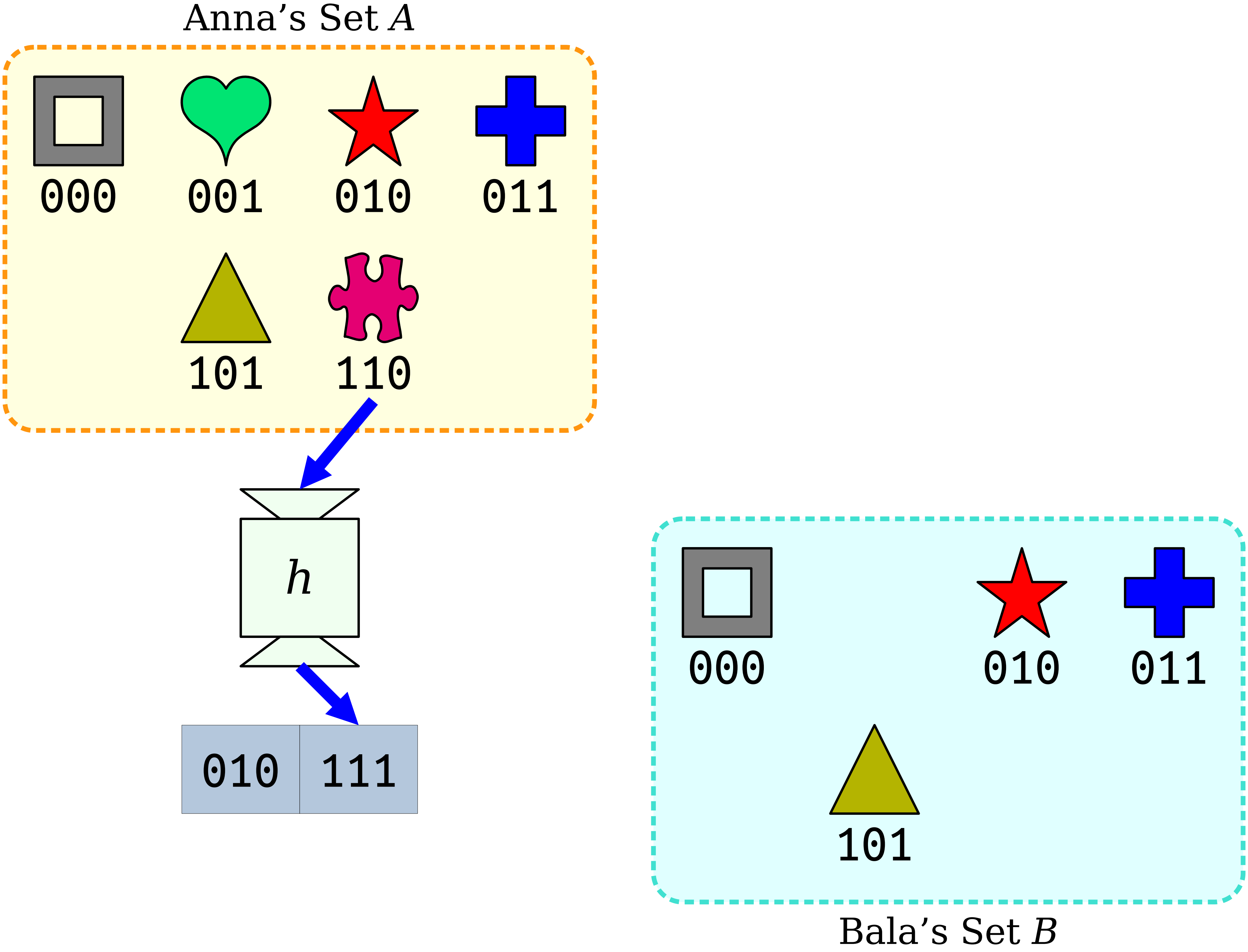

Anna's Set A
000
001
010
011
101
110
h
001
010
000
010
011
101
Bala's Set B

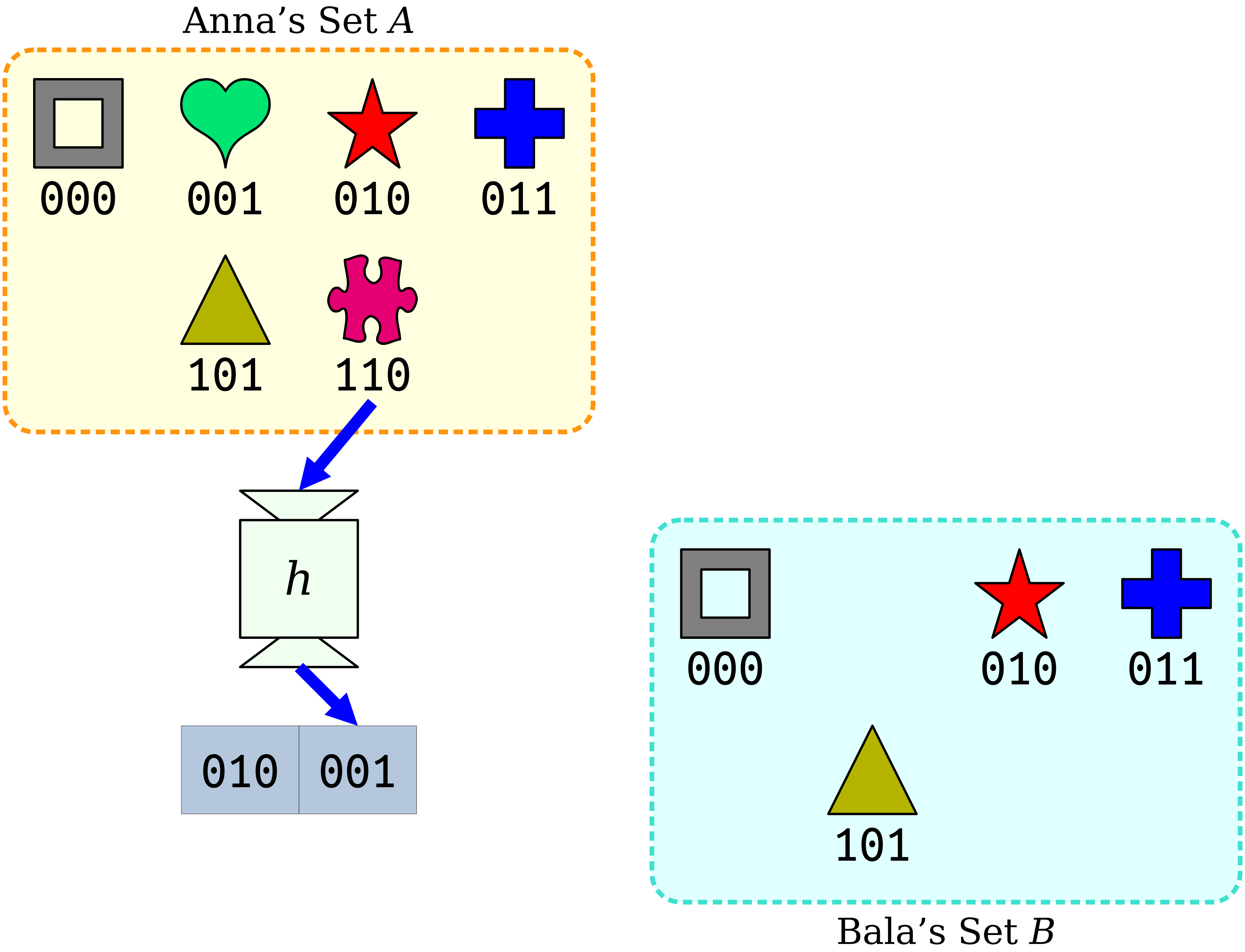

Anna's Set A
000
001
010
011
101
110
h
001
010
000
010
011
101
Bala's Set B

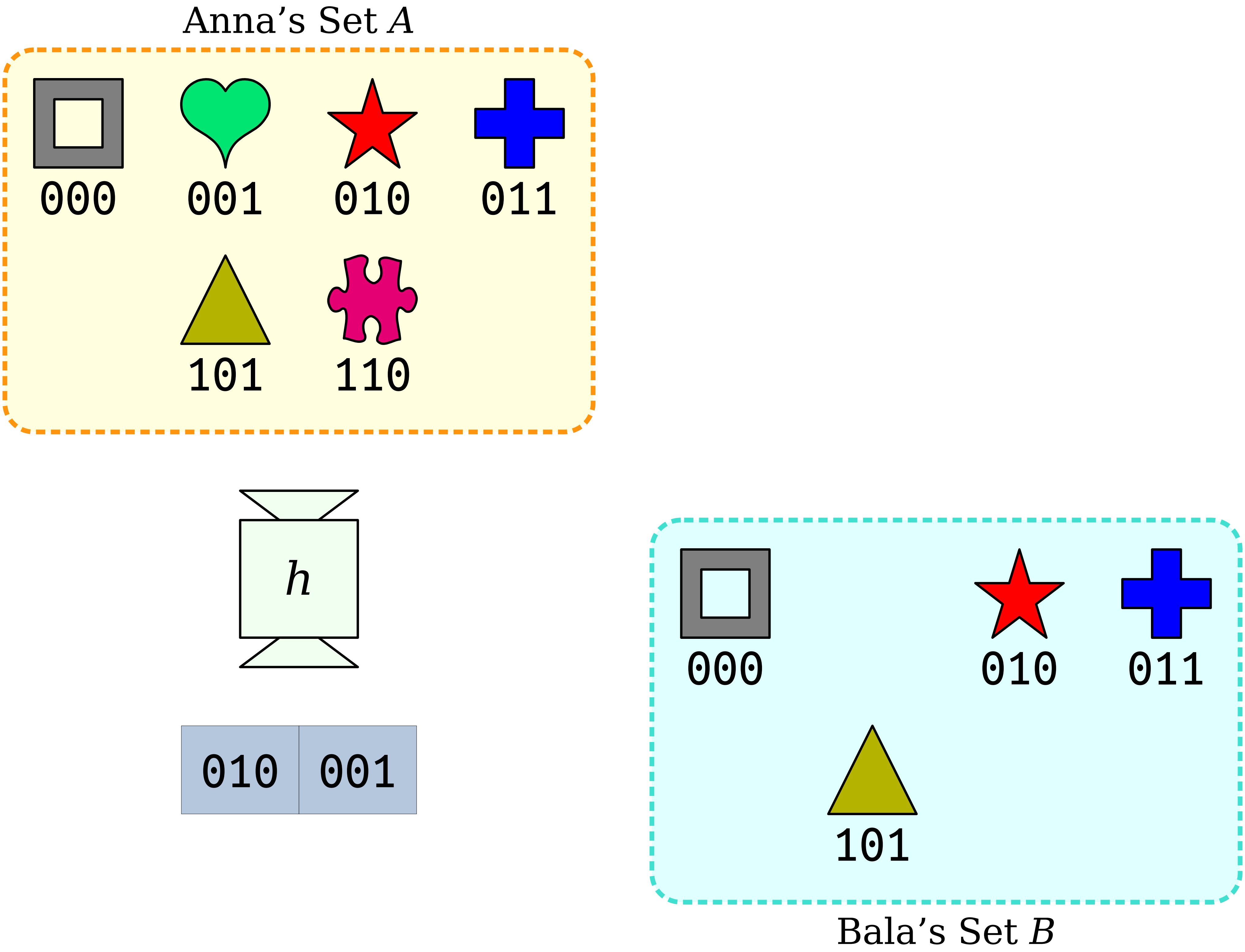

Anna's Set A
000
001
010
011
101
110
h
010
010
000
010
011
101
Bala's Set B

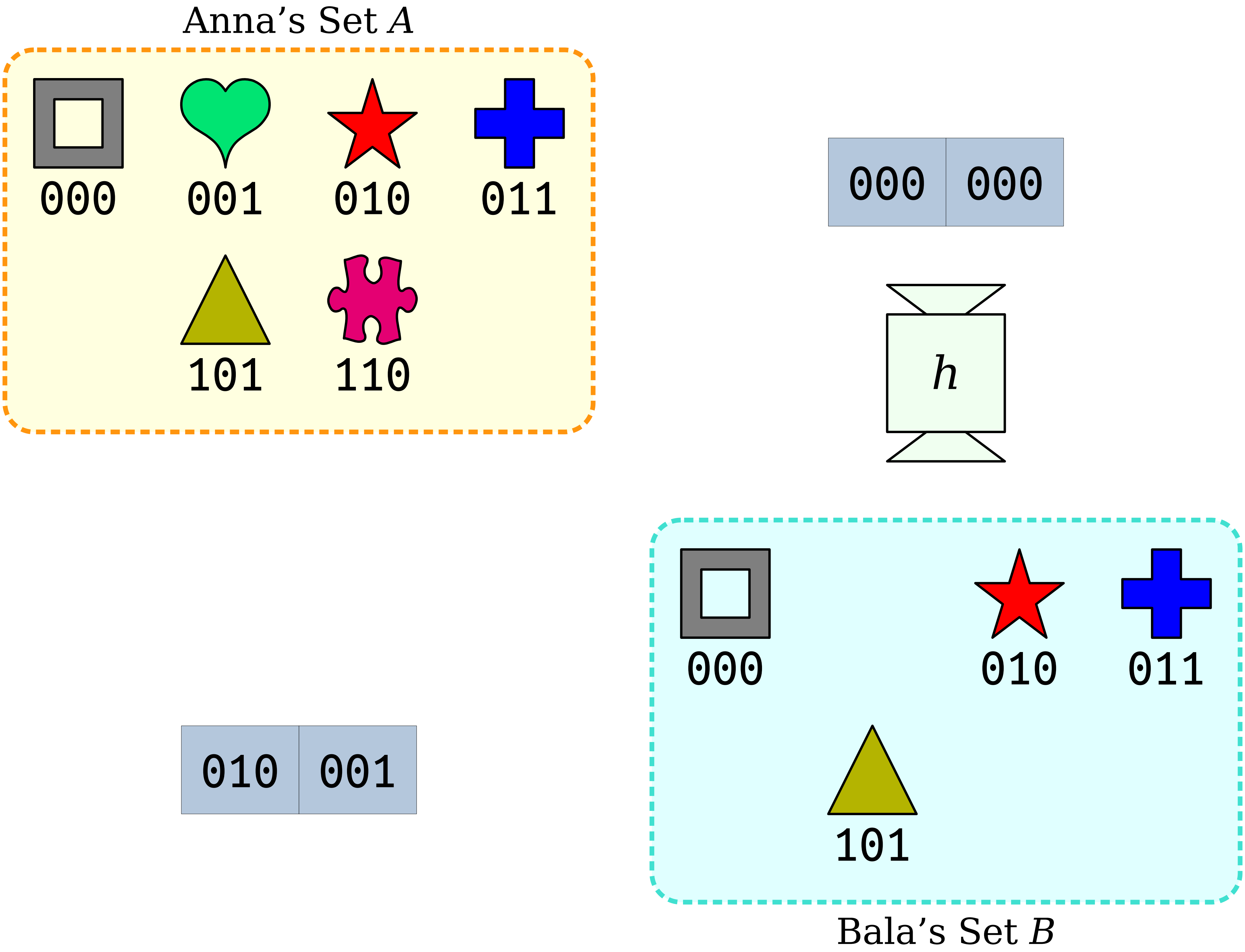

Anna's Set A
000
001
010
011
101
110
h
010
010
000
010
011
101
Bala's Set B

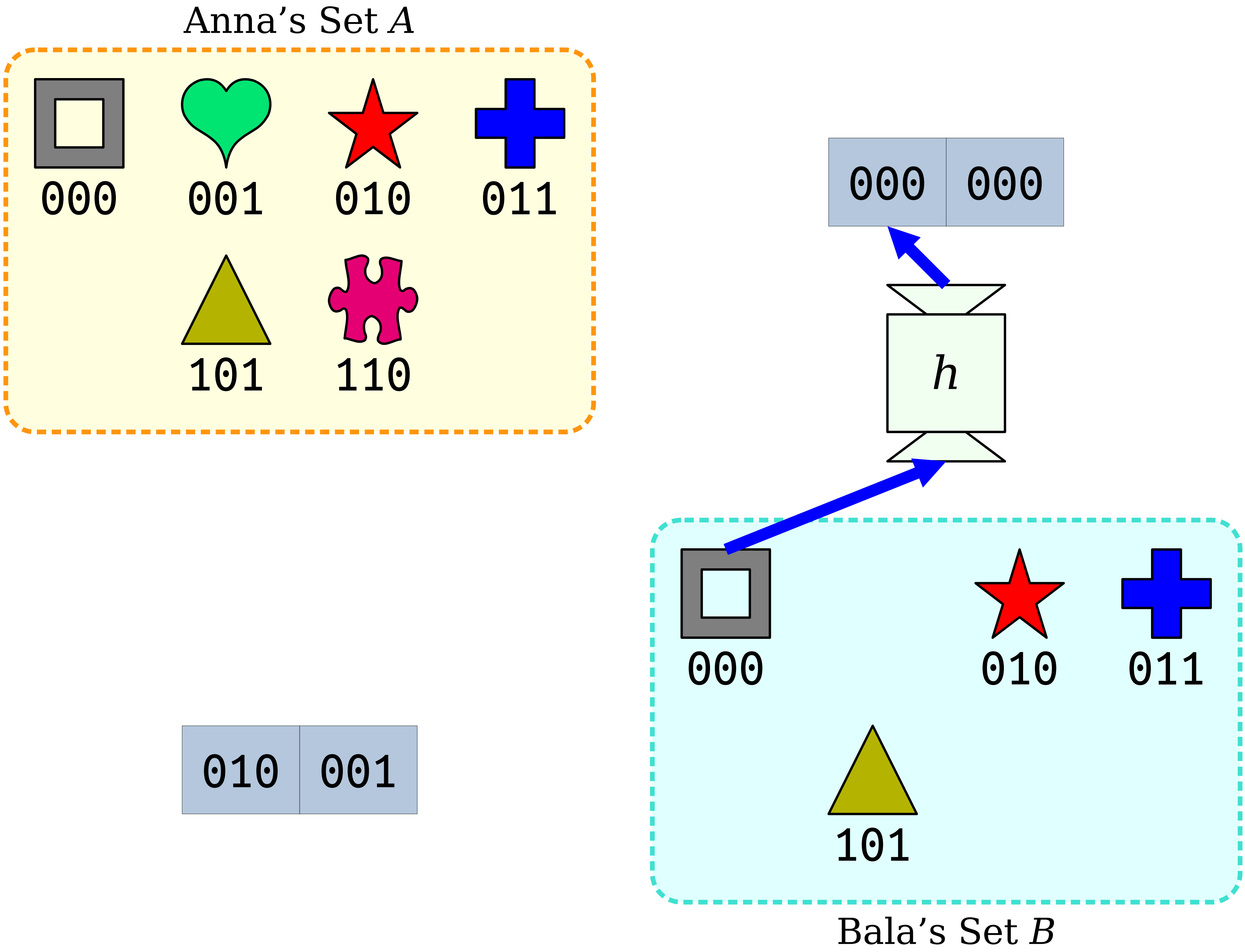

Anna's Set A
000
001
010
011
101
110
h
010
111
000
010
011
101
Bala's Set B

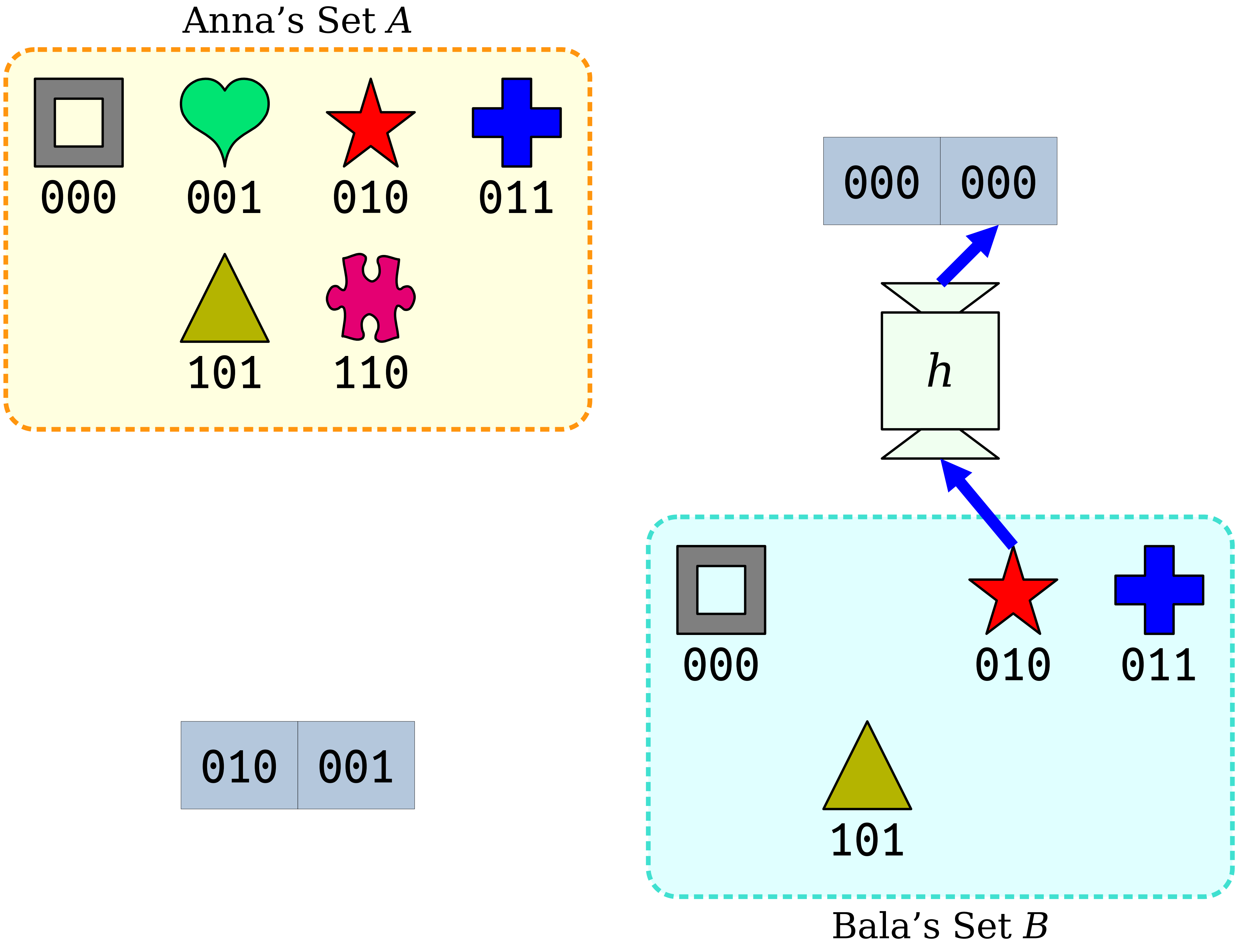

Anna's Set A
000
001
010
011
101
110
h
010
111
000
010
011
101
Bala's Set B

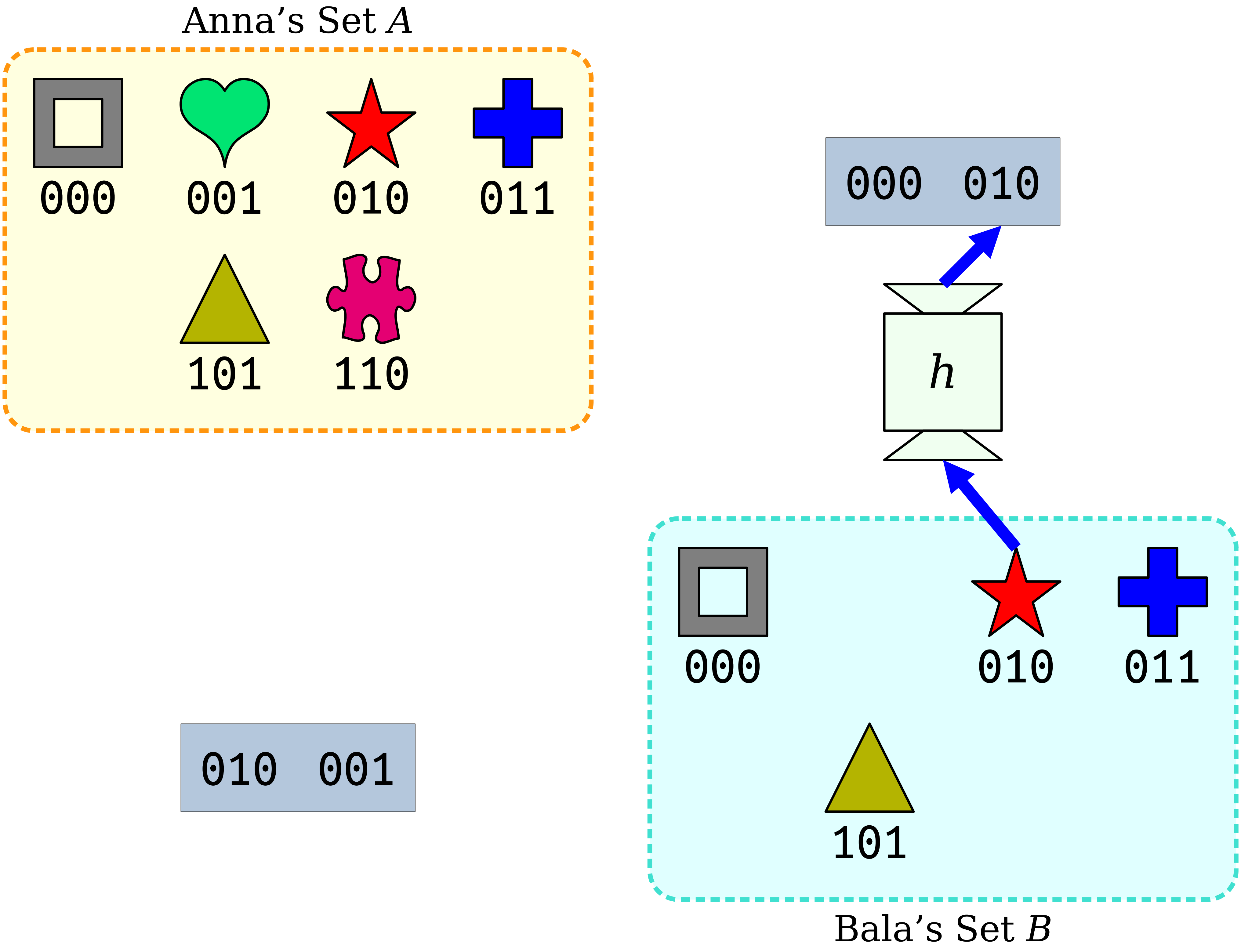

Anna’s Set A
000
001
010
011
101
110
h
010
001
000
010
011
101
Bala’s Set B

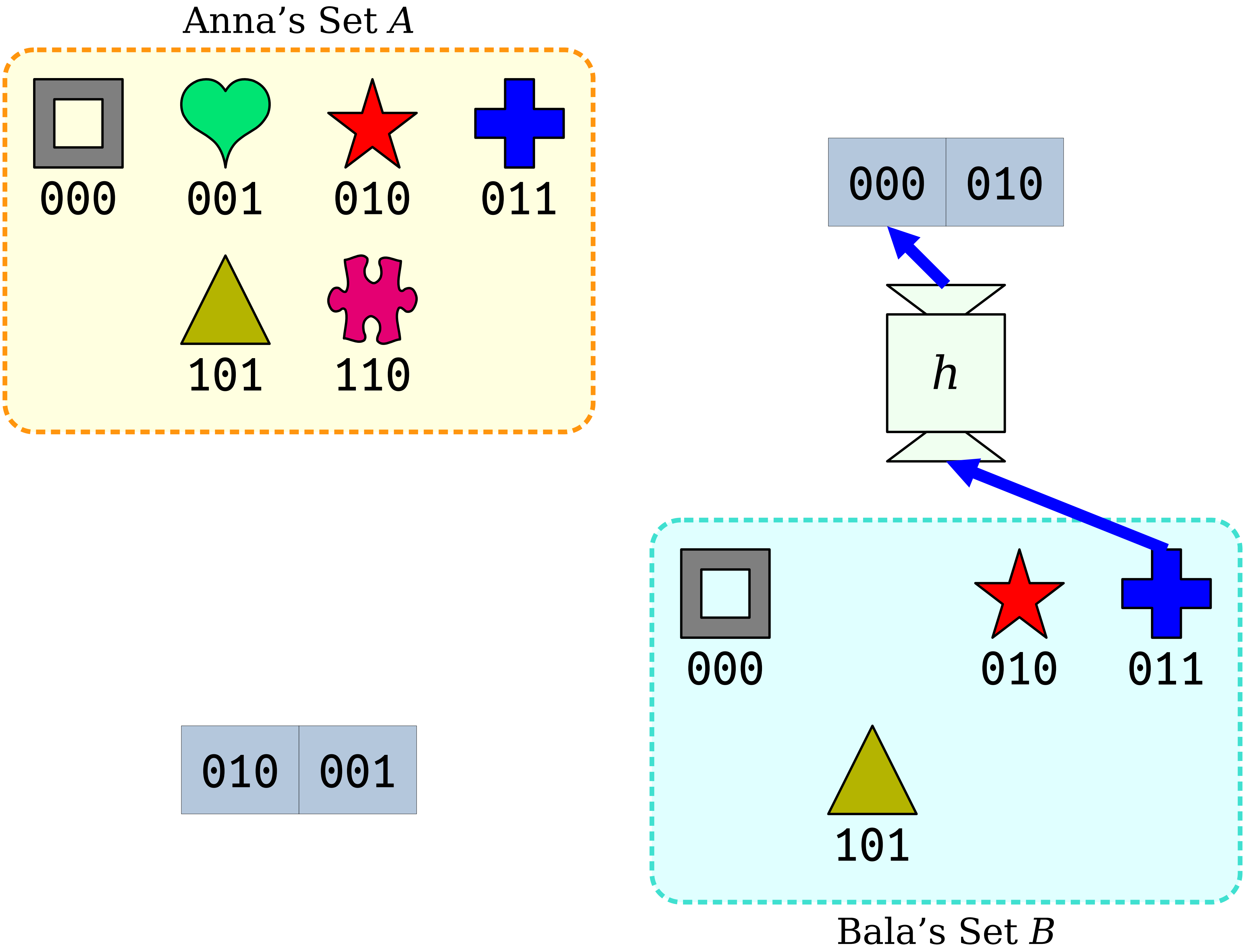

Anna's Set A
000
001
010
011
101
110
h
010
001
000
010
011
101
Bala's Set B

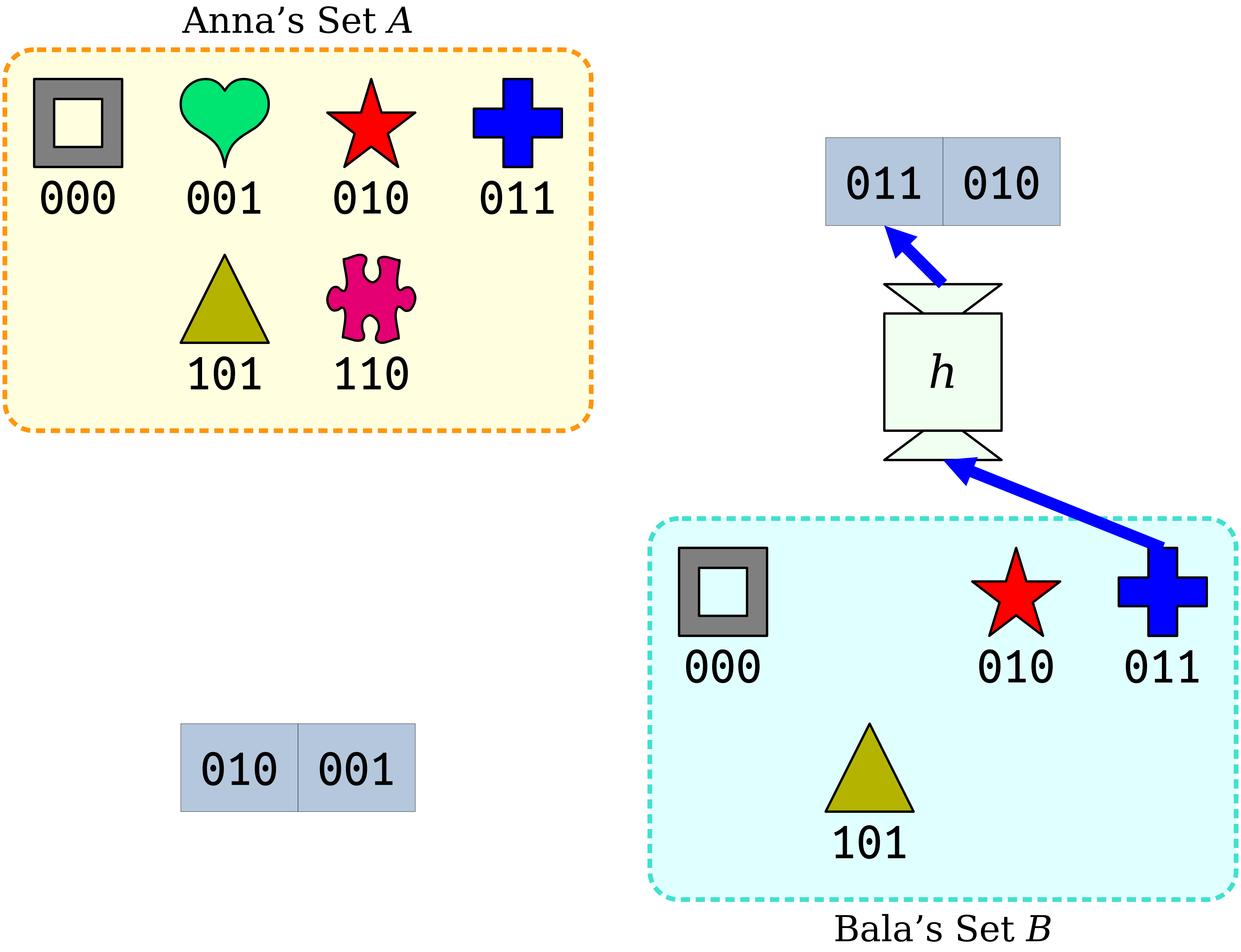

Anna's Set A
000
001
010
011
101
110
000
000
h
010
001
000
010
011
101
Bala's Set B

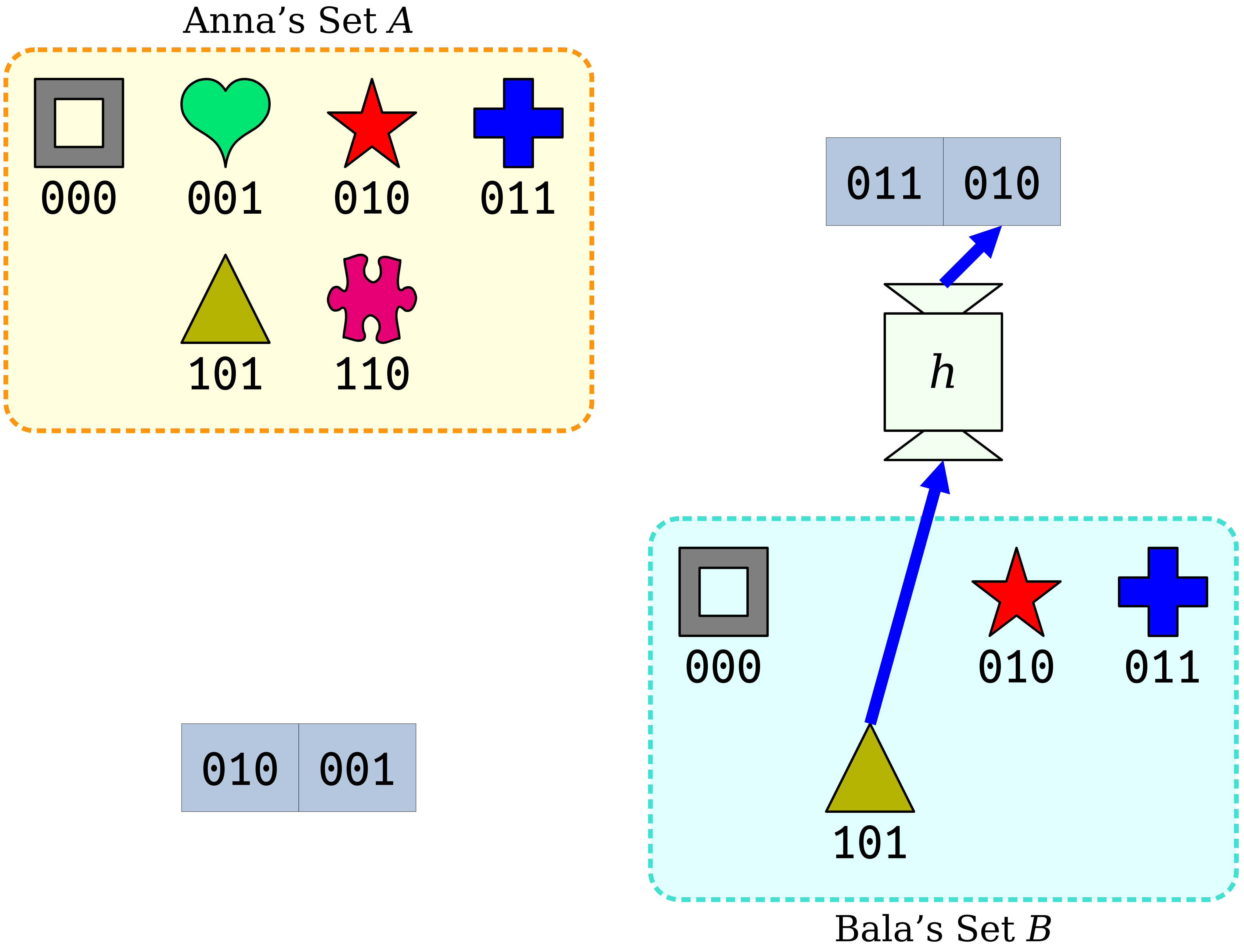

Anna’s Set A
000
001
010
011
101
110
000 000
h
010 001
000
010
011
101
Bala’s Set B

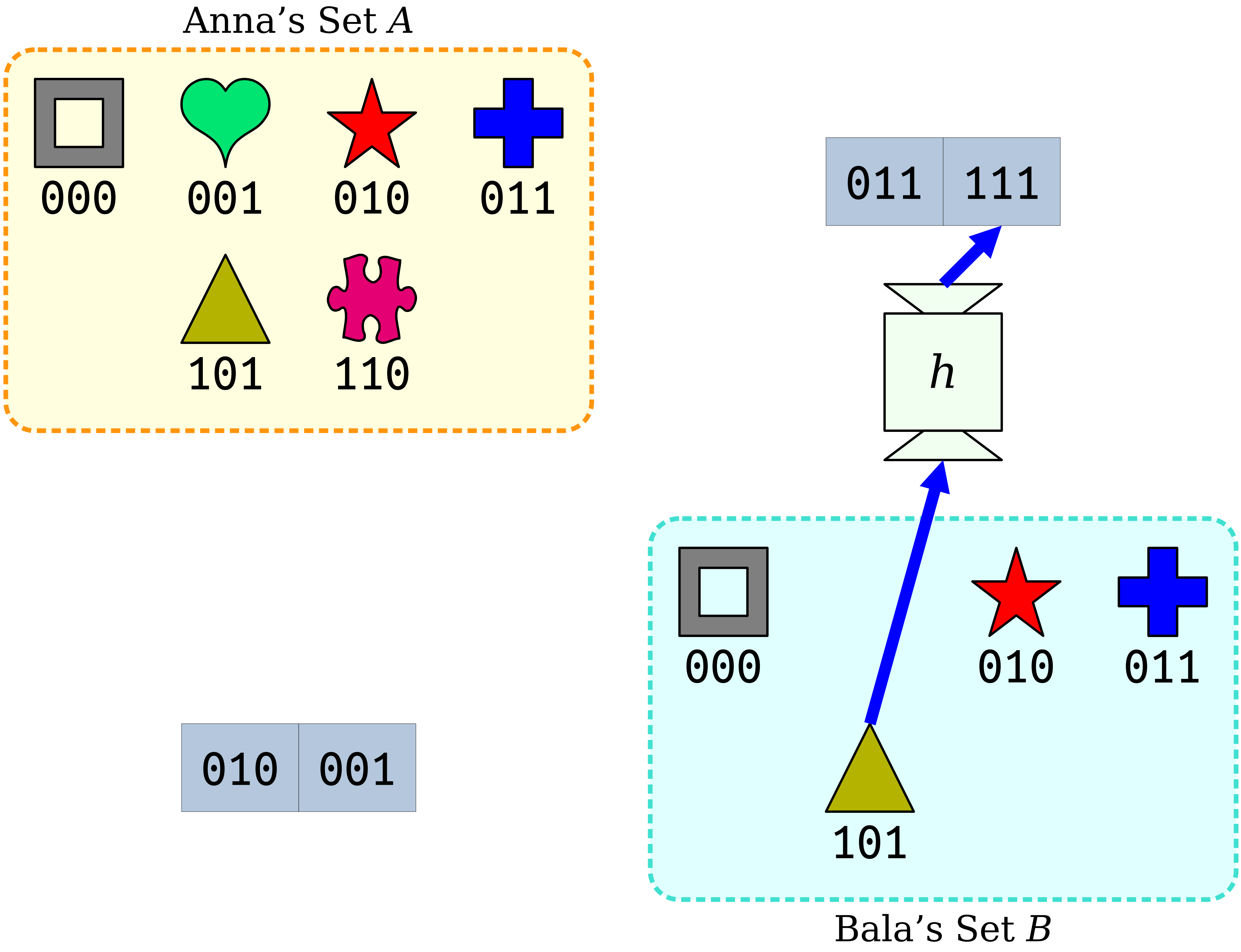

Anna's Set A
000
001
010
011
101
110
000 000
h
010 001
000
010
011
101
Bala's Set B

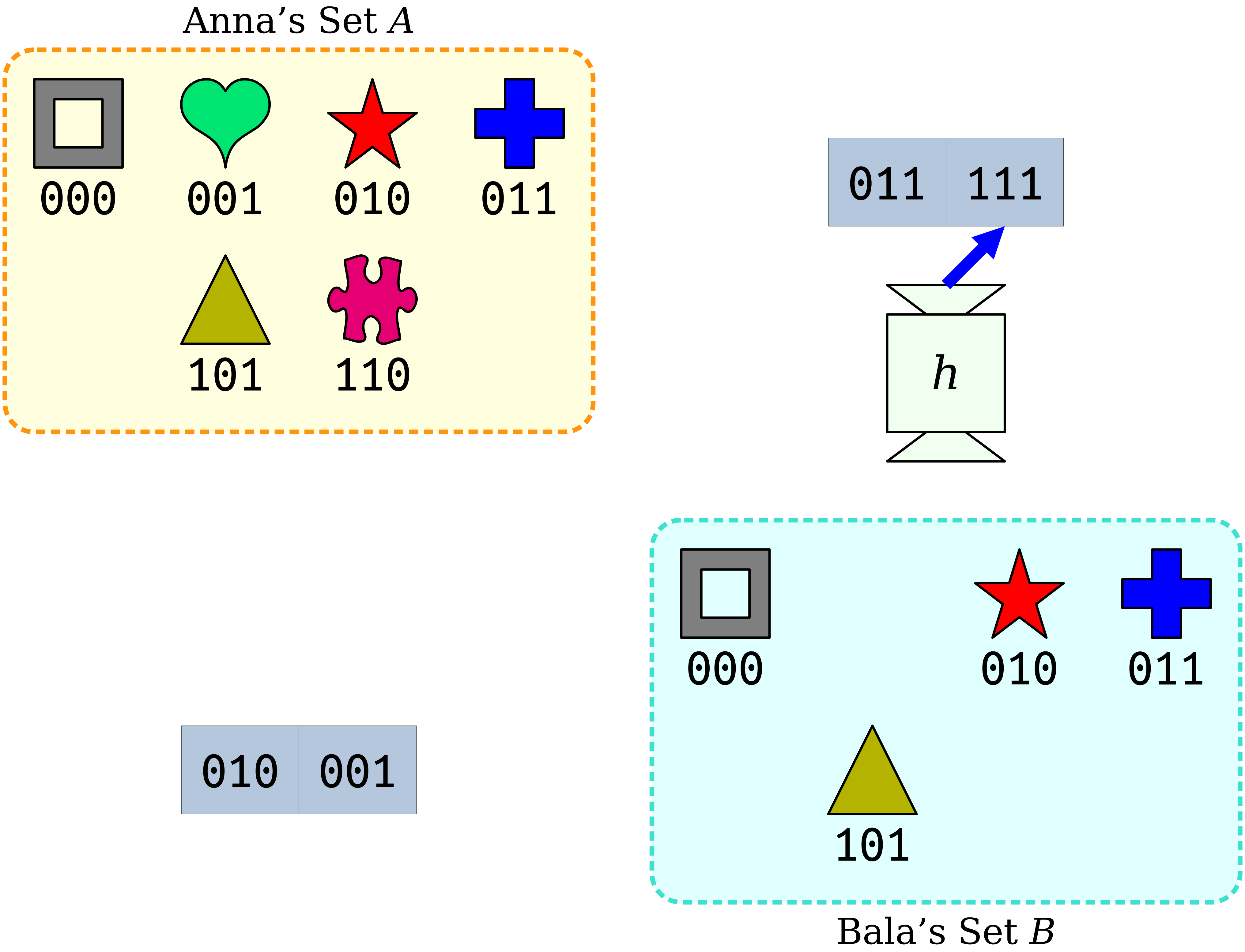

Anna’s Set *A*
000
001
010
011
101
110
000
010
h
010
001
000
010
011
101
Bala’s Set *B*

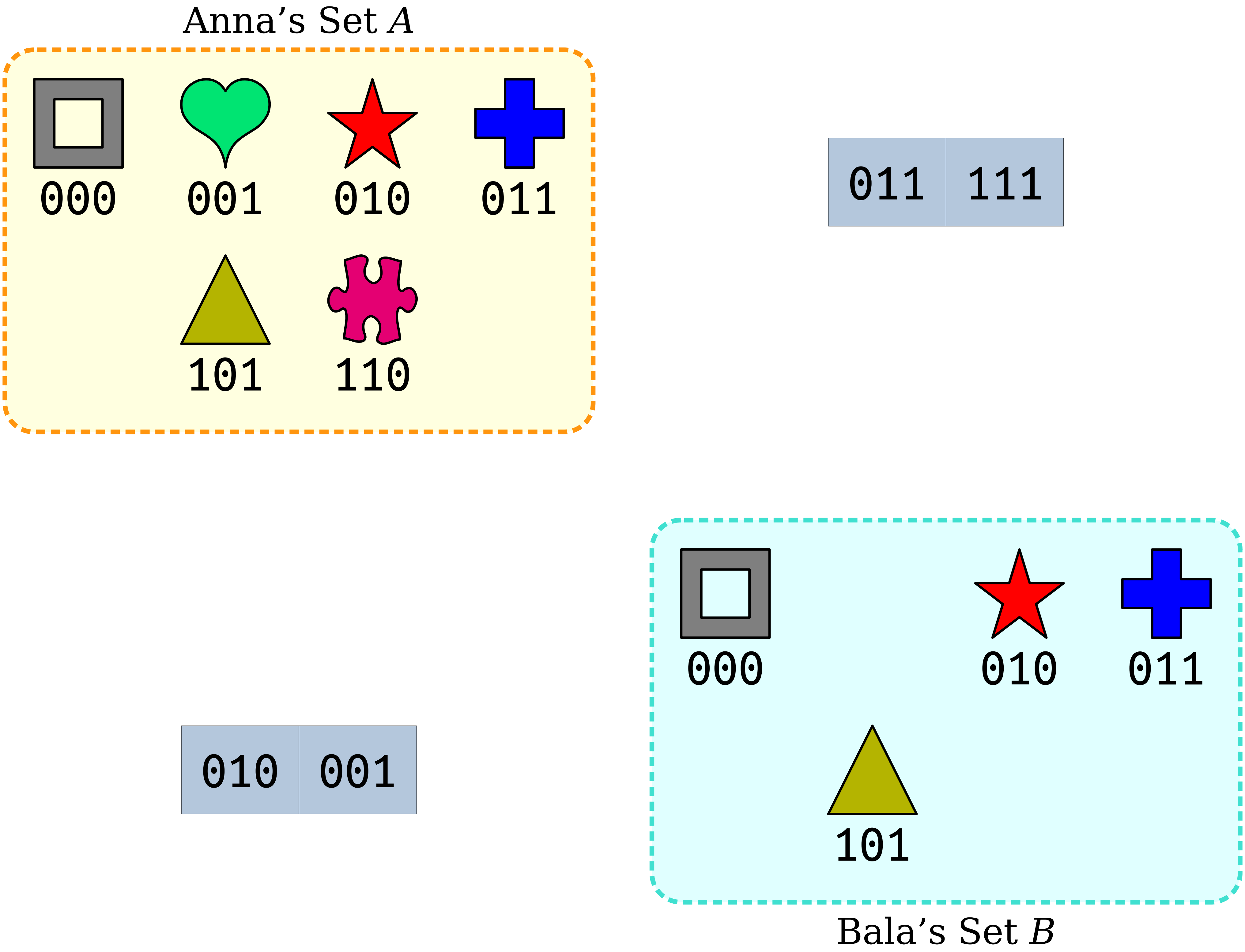

Anna's Set A
000
001
010
011
101
110
000 010
h
010 001
000
010
011
101
Bala's Set B

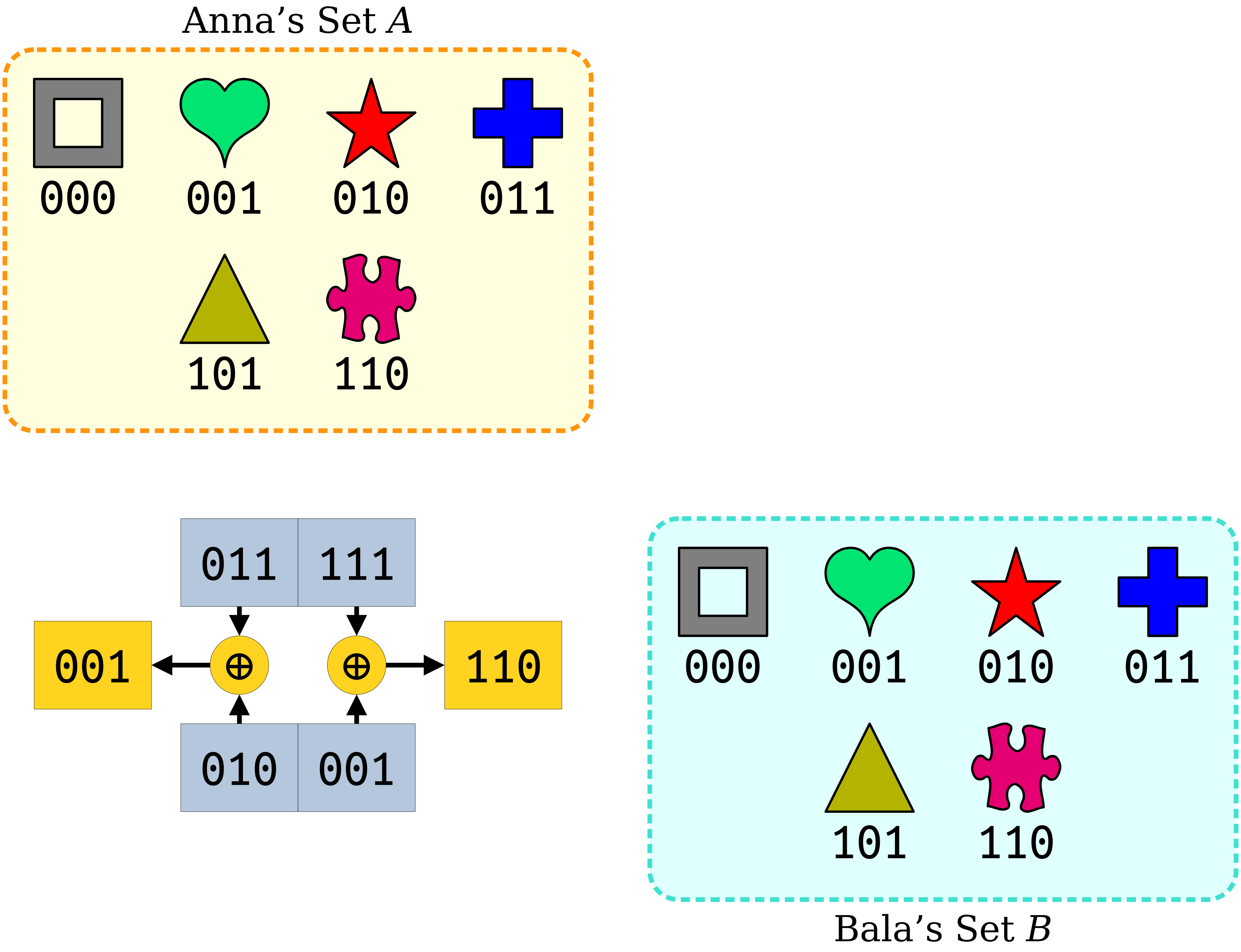

Anna’s Set A
000
001
010
011
101
110
011
010
h
010
001
000
010
011
101
Bala’s Set B

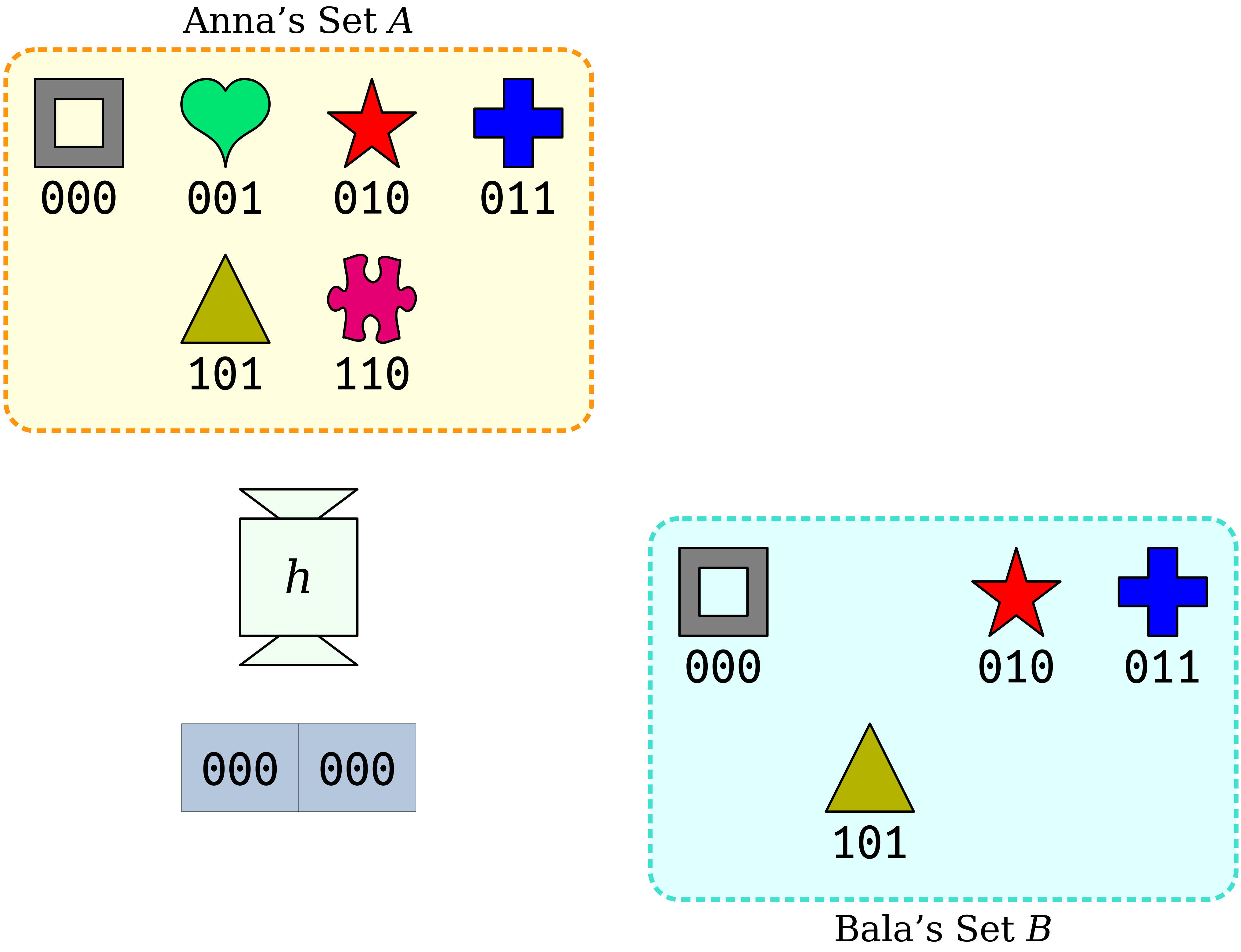

Anna's Set A
000
001
010
011
101
110
011
010
h
010
001
000
010
011
101
Bala's Set B

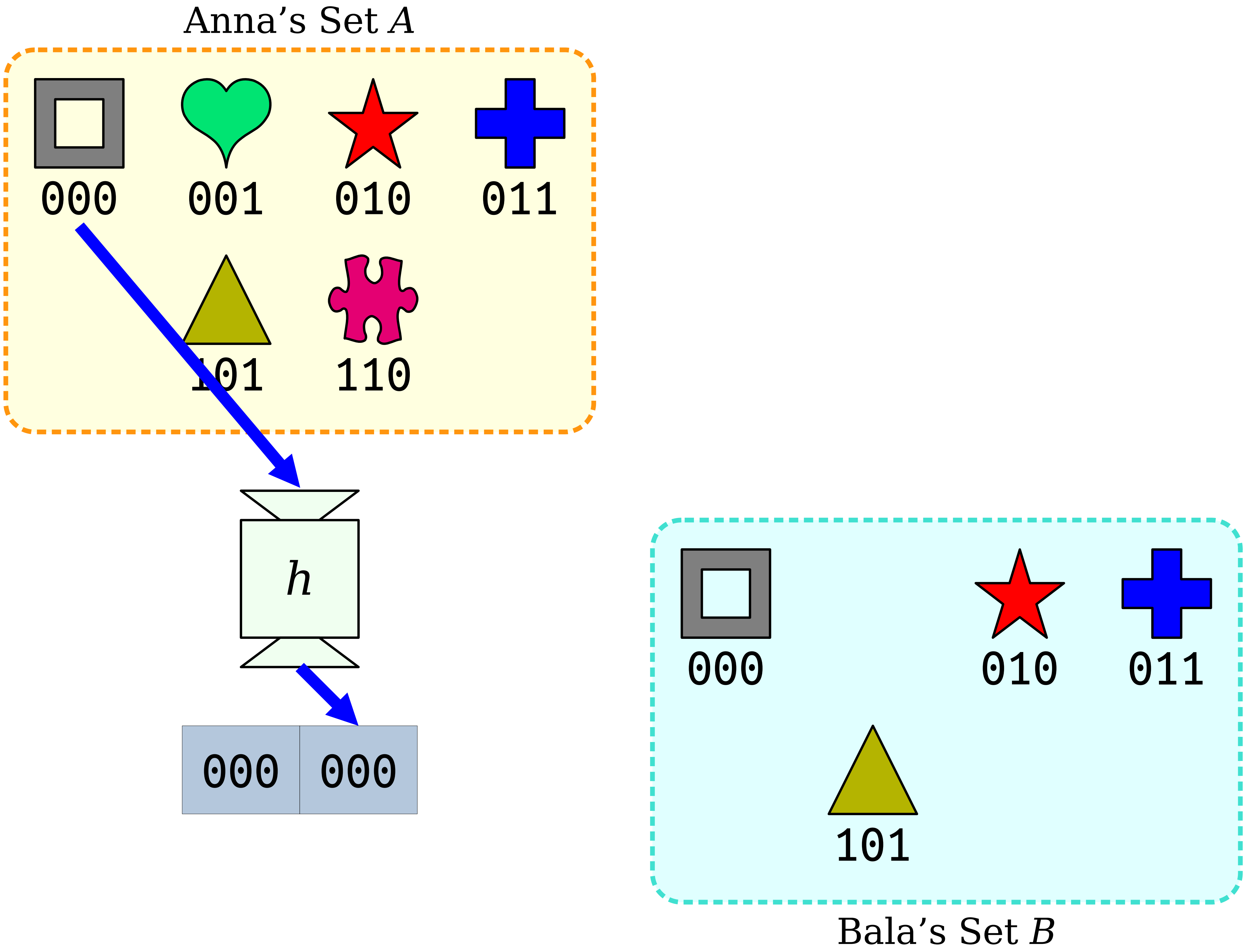

Anna's Set A
000
001
010
011
101
110
011
111
h
010
001
000
010
011
101
Bala's Set B

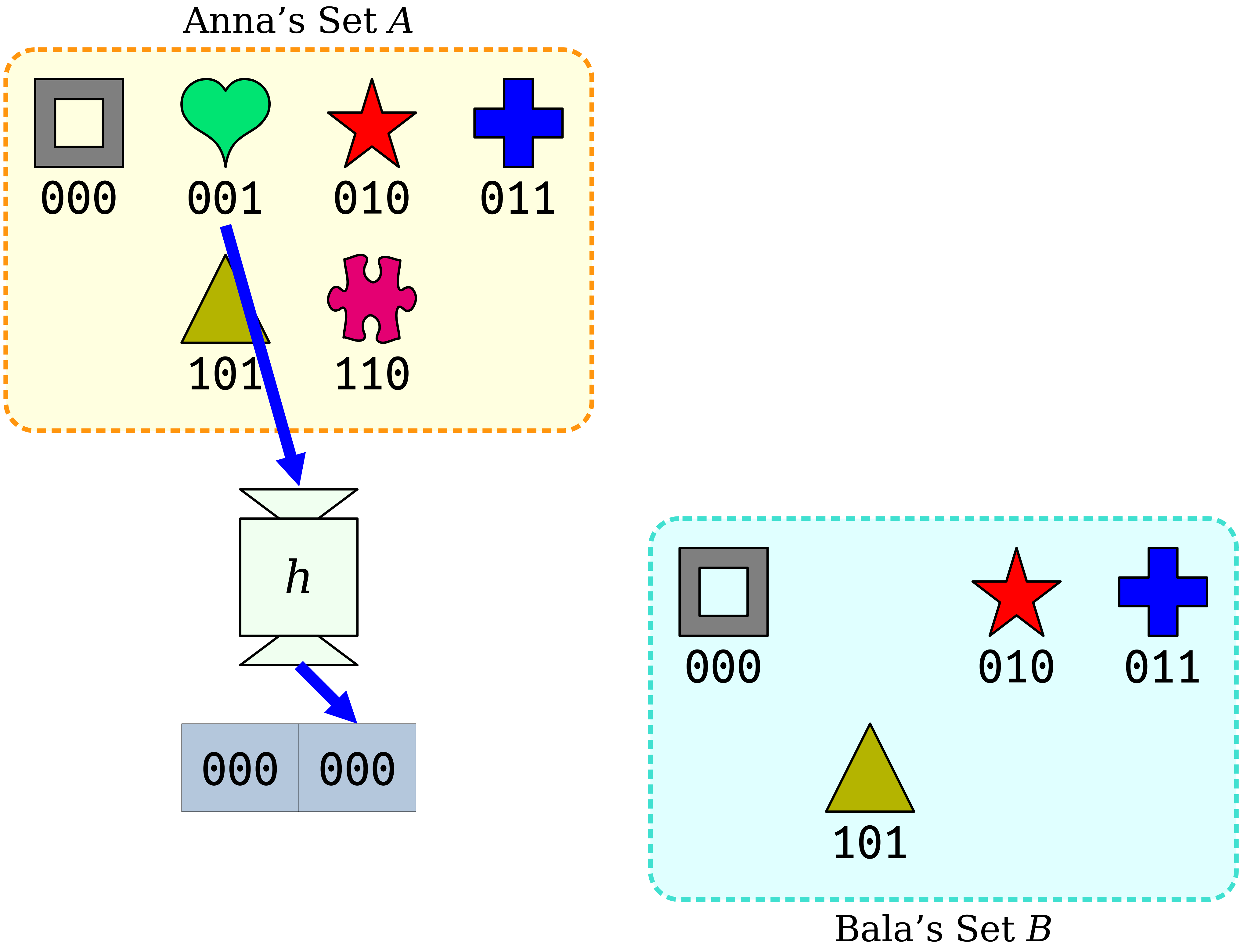

Anna's Set A
000
001
010
011
101
110
011
111
h
010
001
000
010
011
101
Bala's Set B

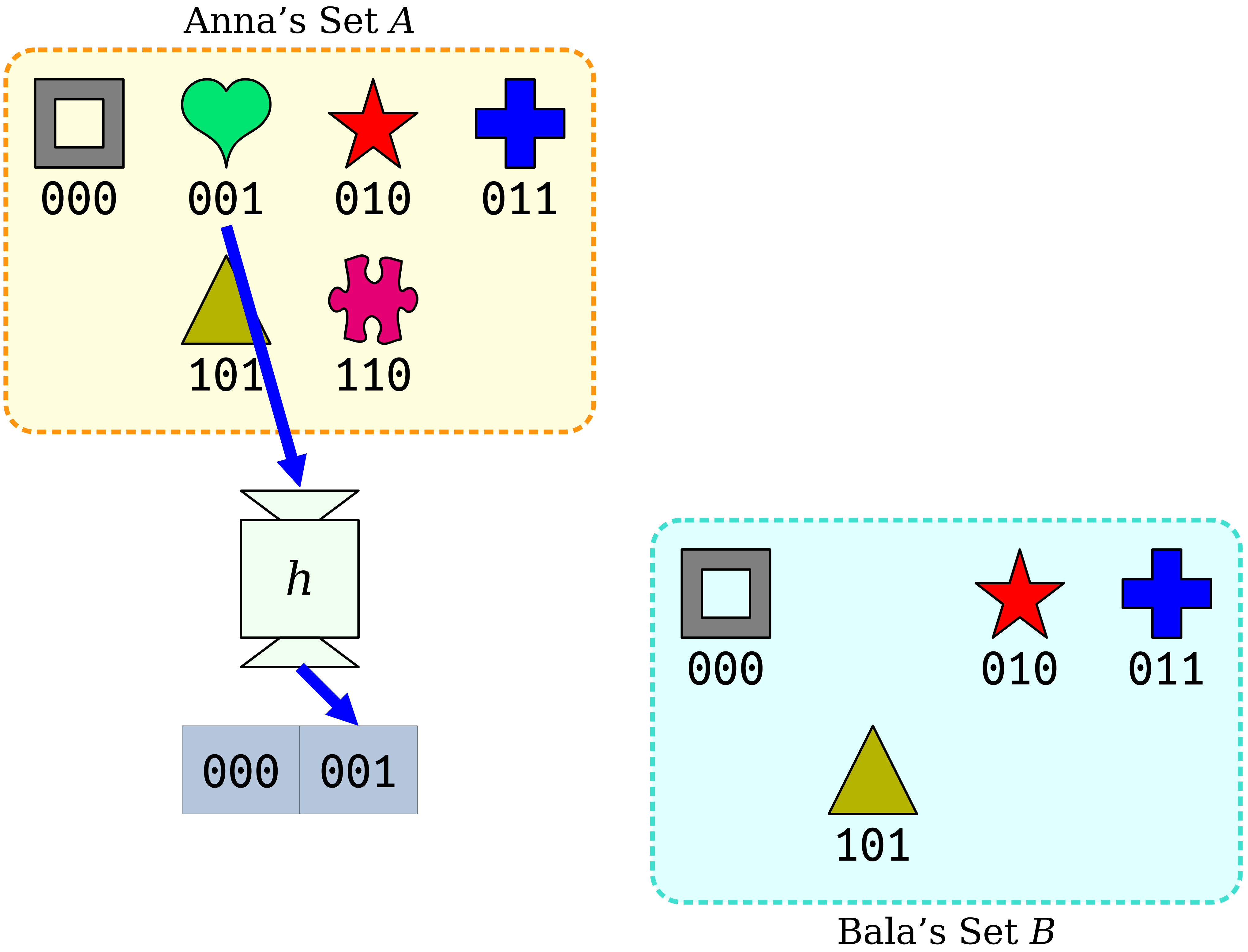

Anna's Set A
000
001
010
011
101
110
011
111
010
001
000
010
011
101
Bala's Set B

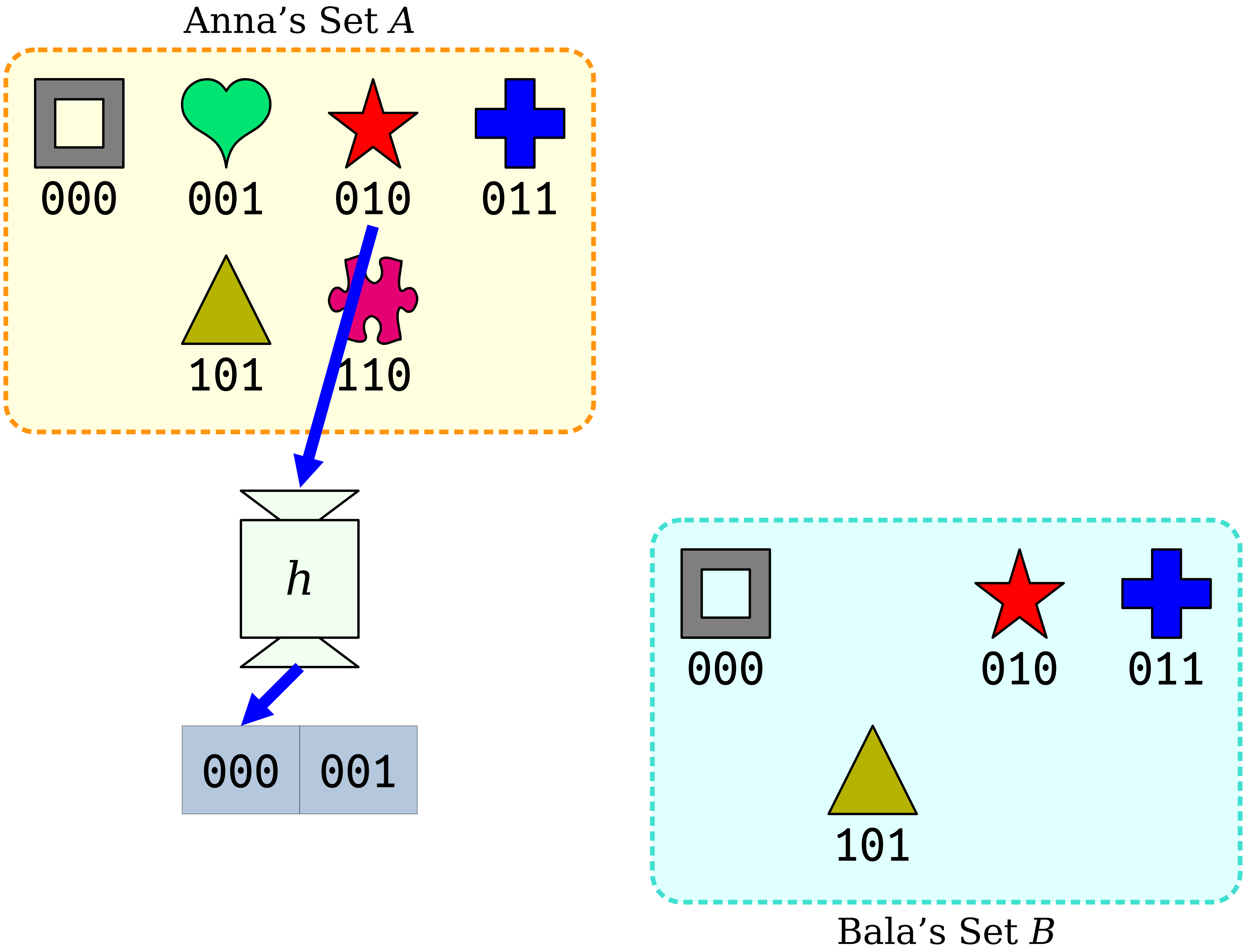

Anna's Set A
000
001
010
011
101
110
011
111
001
⊕
⊕
110
010
001
000
001
010
011
101
110
Bala's Set B

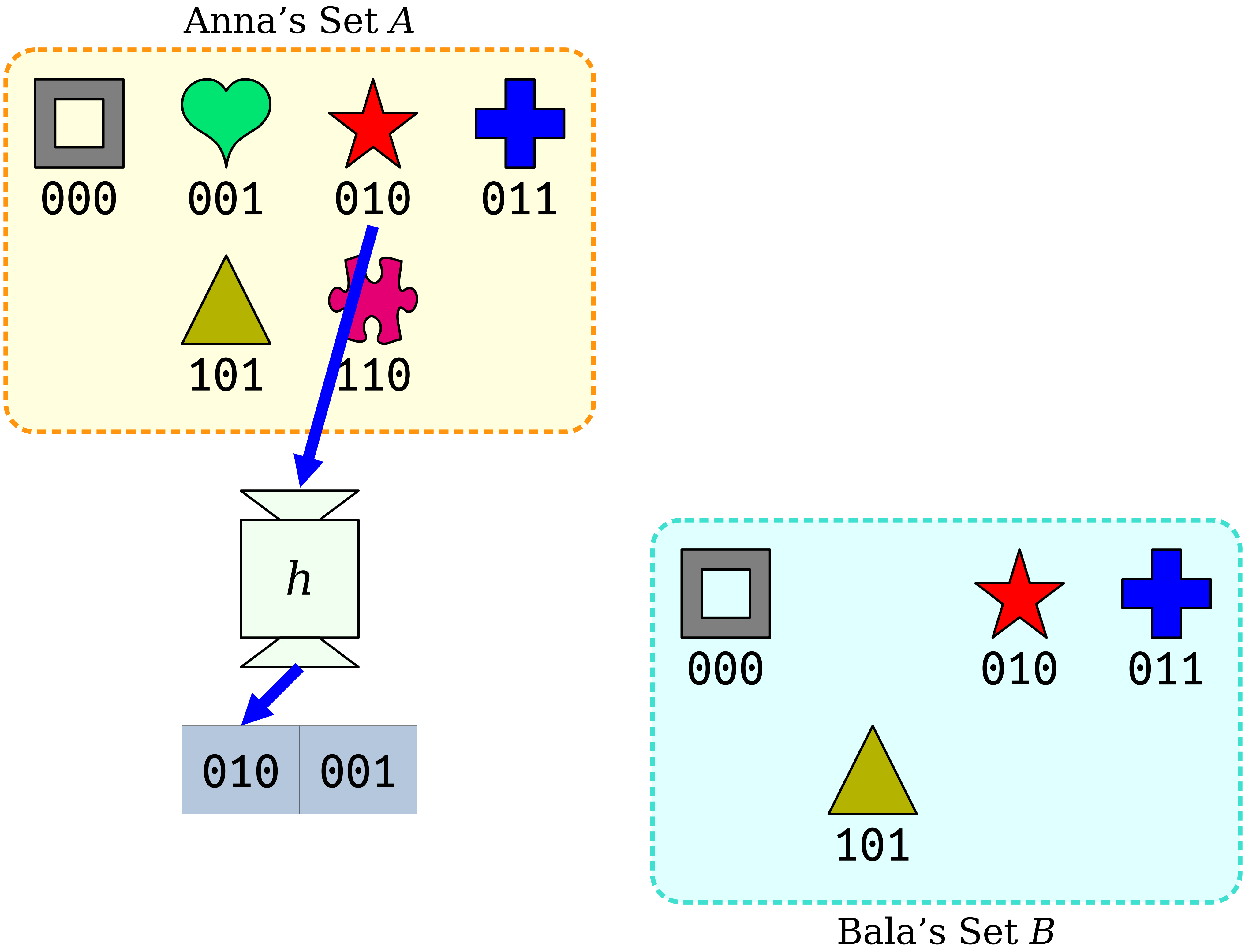

Anna’s Set *A*
000
001
010
011
101
110
h
000
000
000
010
011
101
Bala’s Set *B*

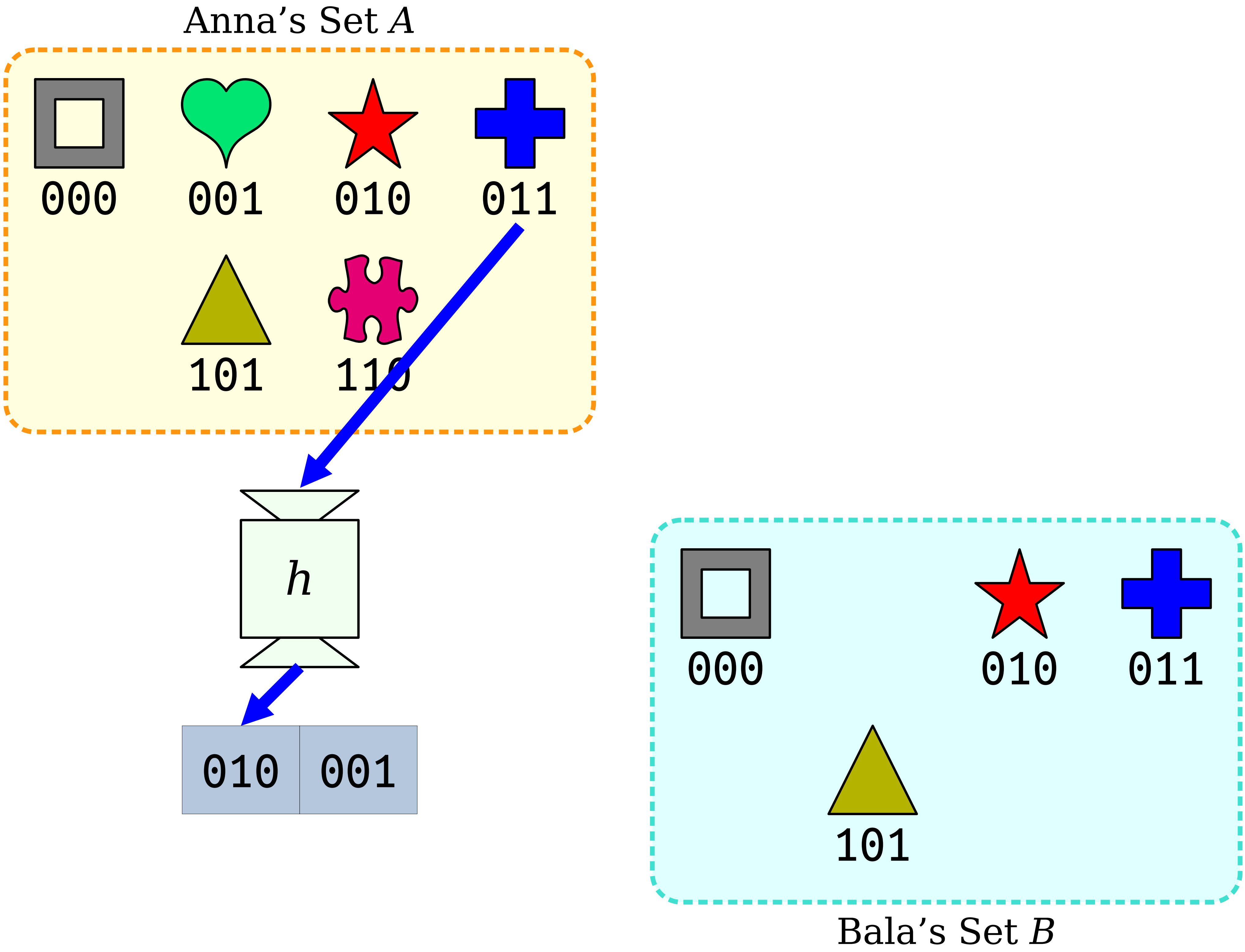

Anna's Set A
000
001
010
011
101
110
h
000
000
000
010
011
101
Bala's Set B

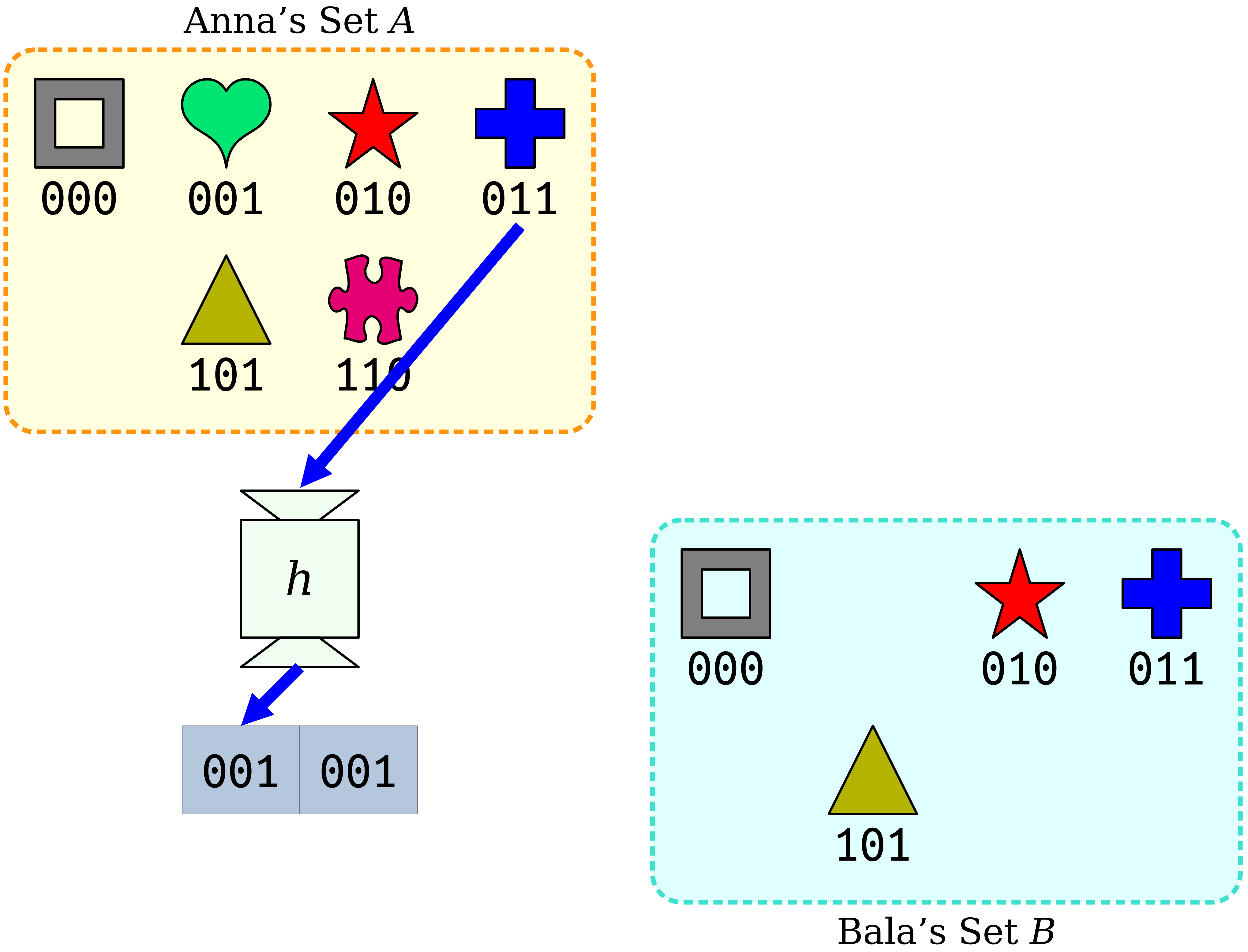

Anna’s Set A
000
001
010
011
101
110
h
000
000
000
010
011
101
Bala’s Set B

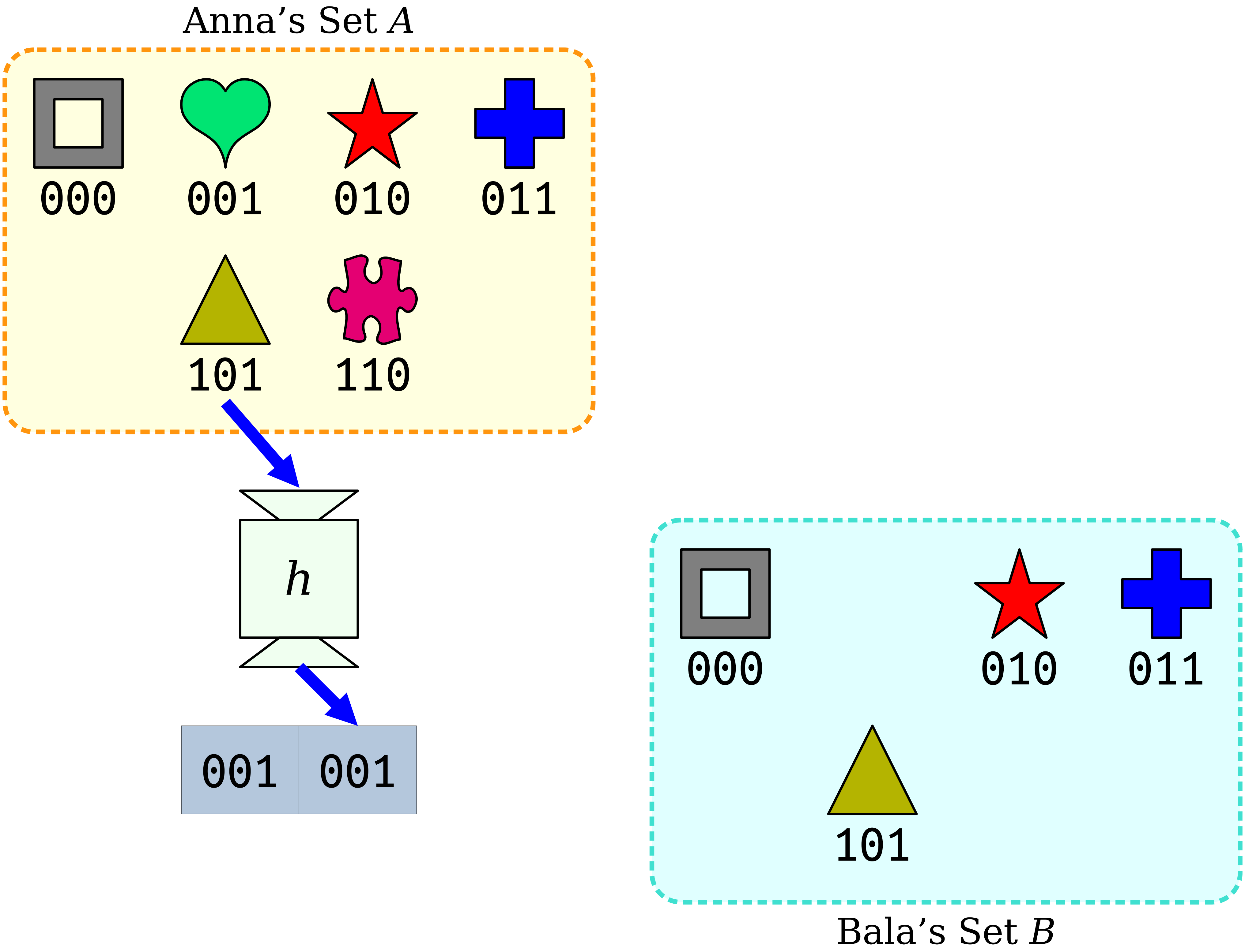

Anna’s Set A
000
001
010
011
101
110
h
000
001
000
010
011
101
Bala’s Set B

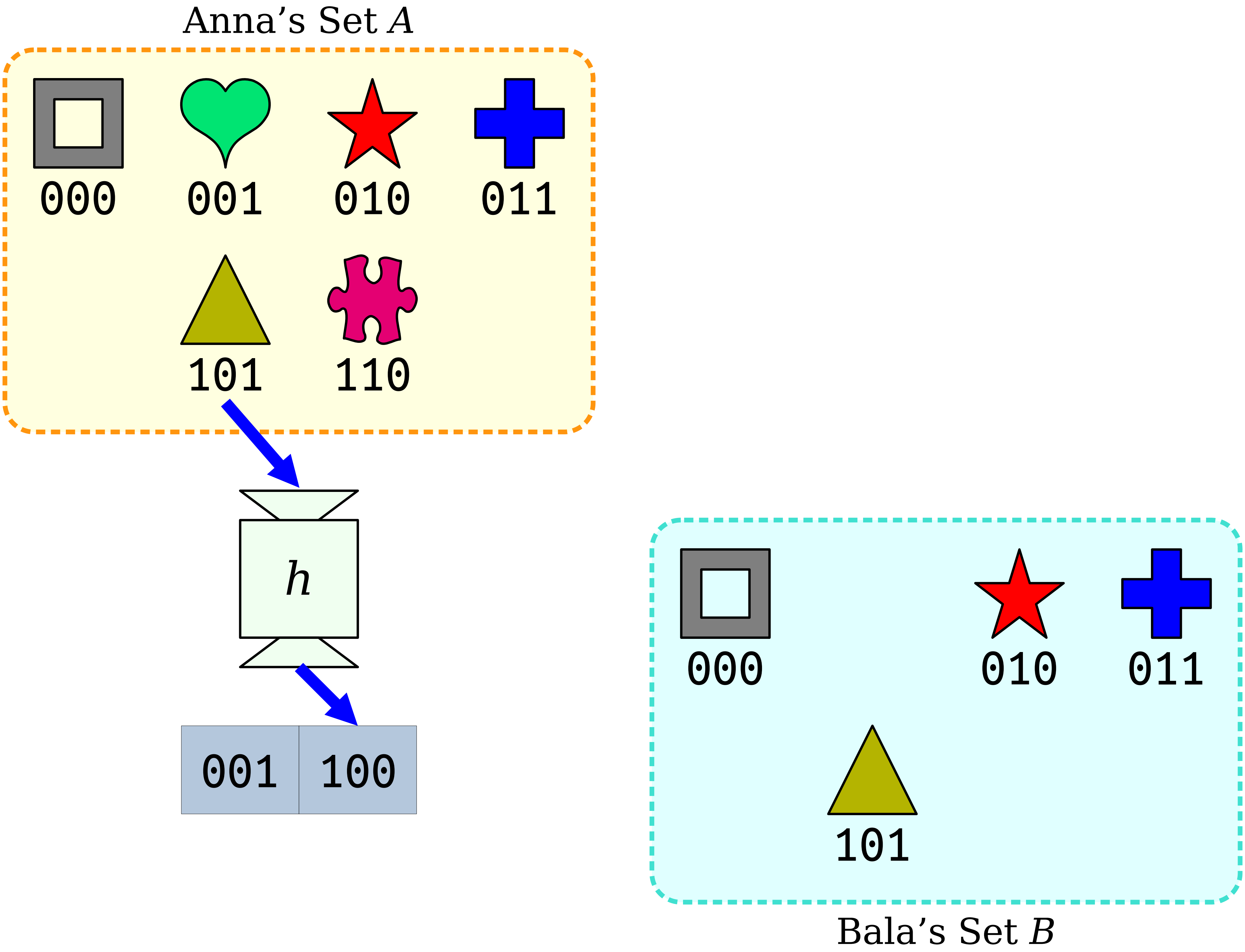

Anna’s Set A
000
001
010
011
101
110
h
000
001
000
010
011
101
Bala’s Set B

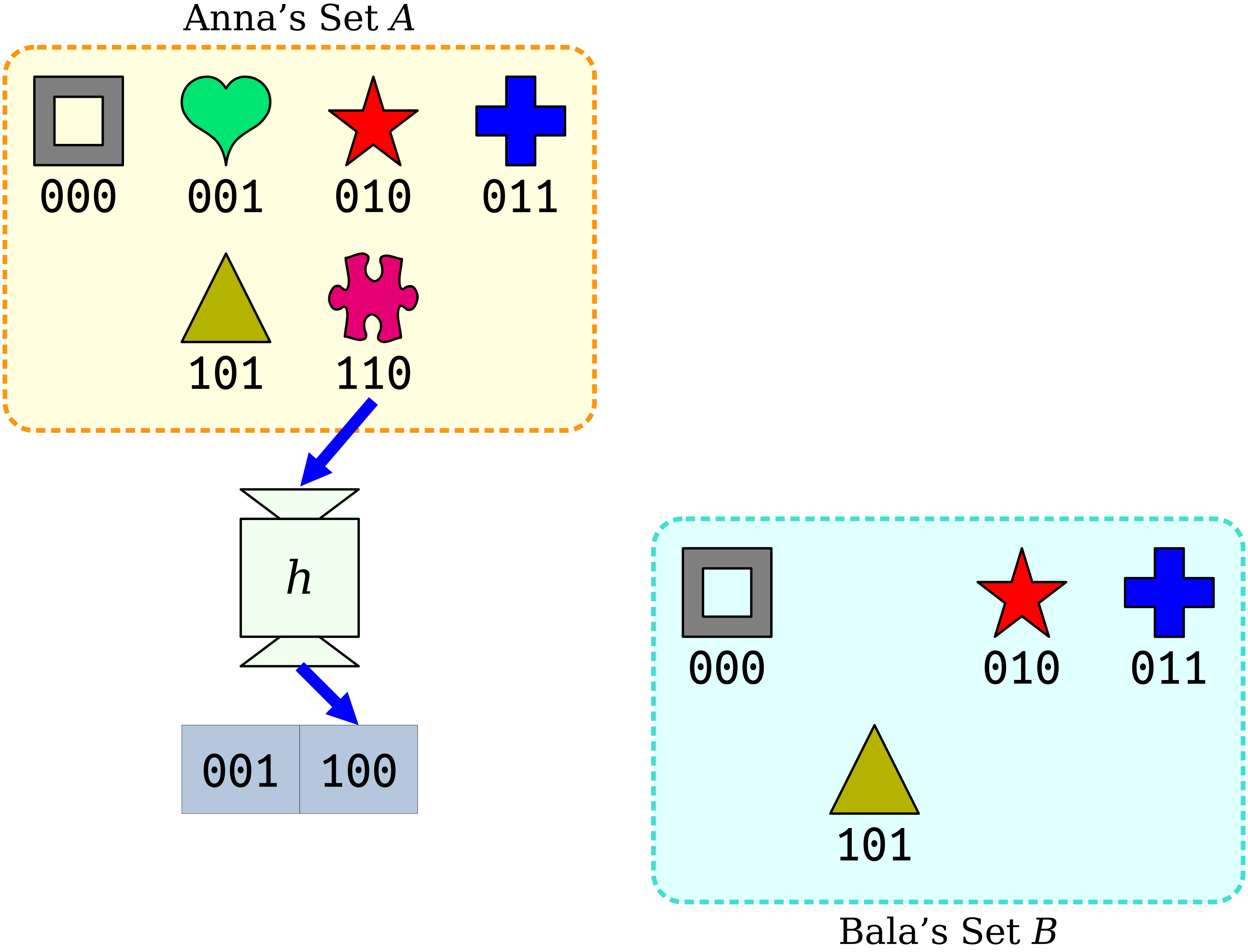

Anna's Set A
000
001
010
011
101
110
h
010
001
000
010
011
101
Bala's Set B

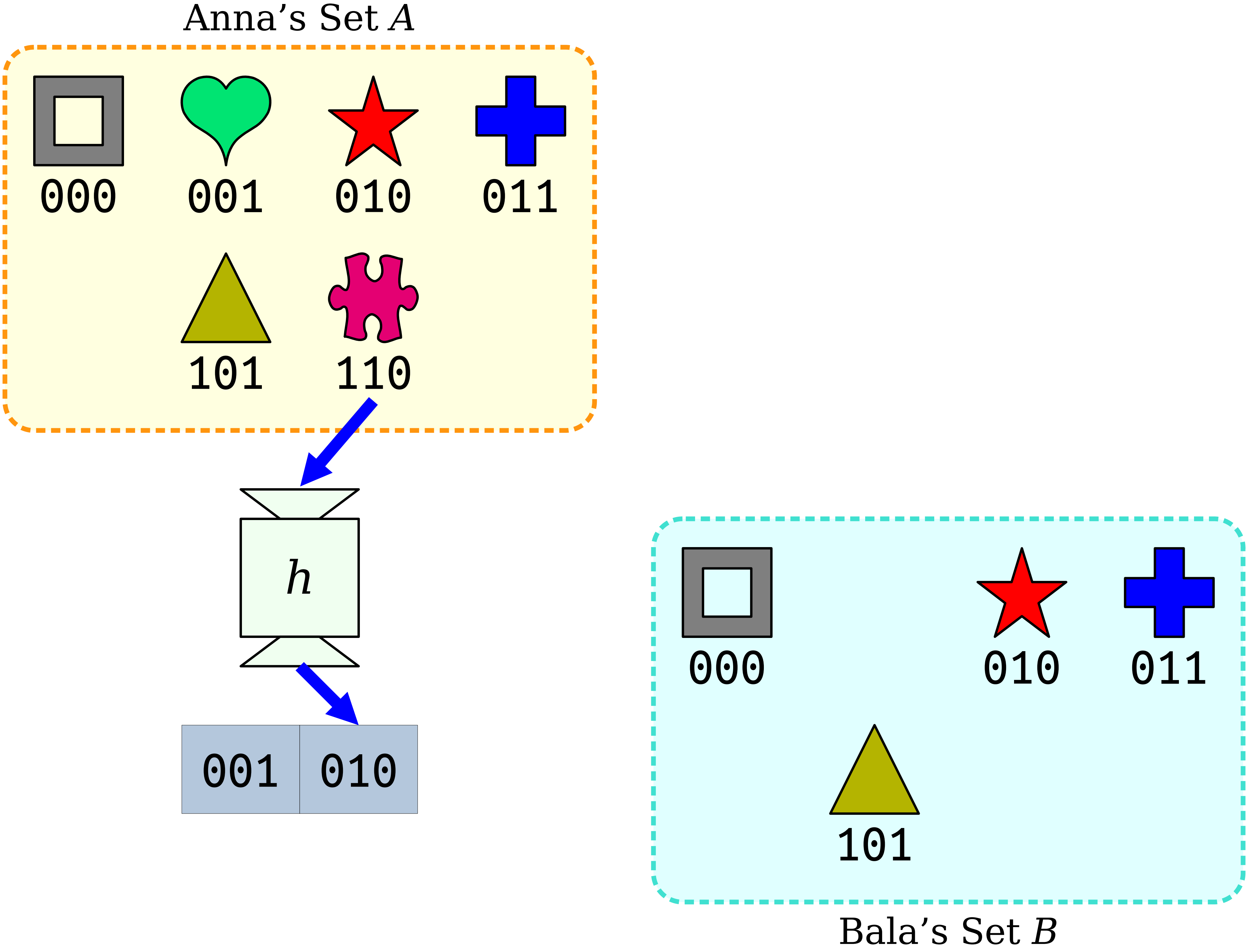

Anna’s Set A
000
001
010
011
101
110
h
010
001
000
010
011
101
Bala’s Set B

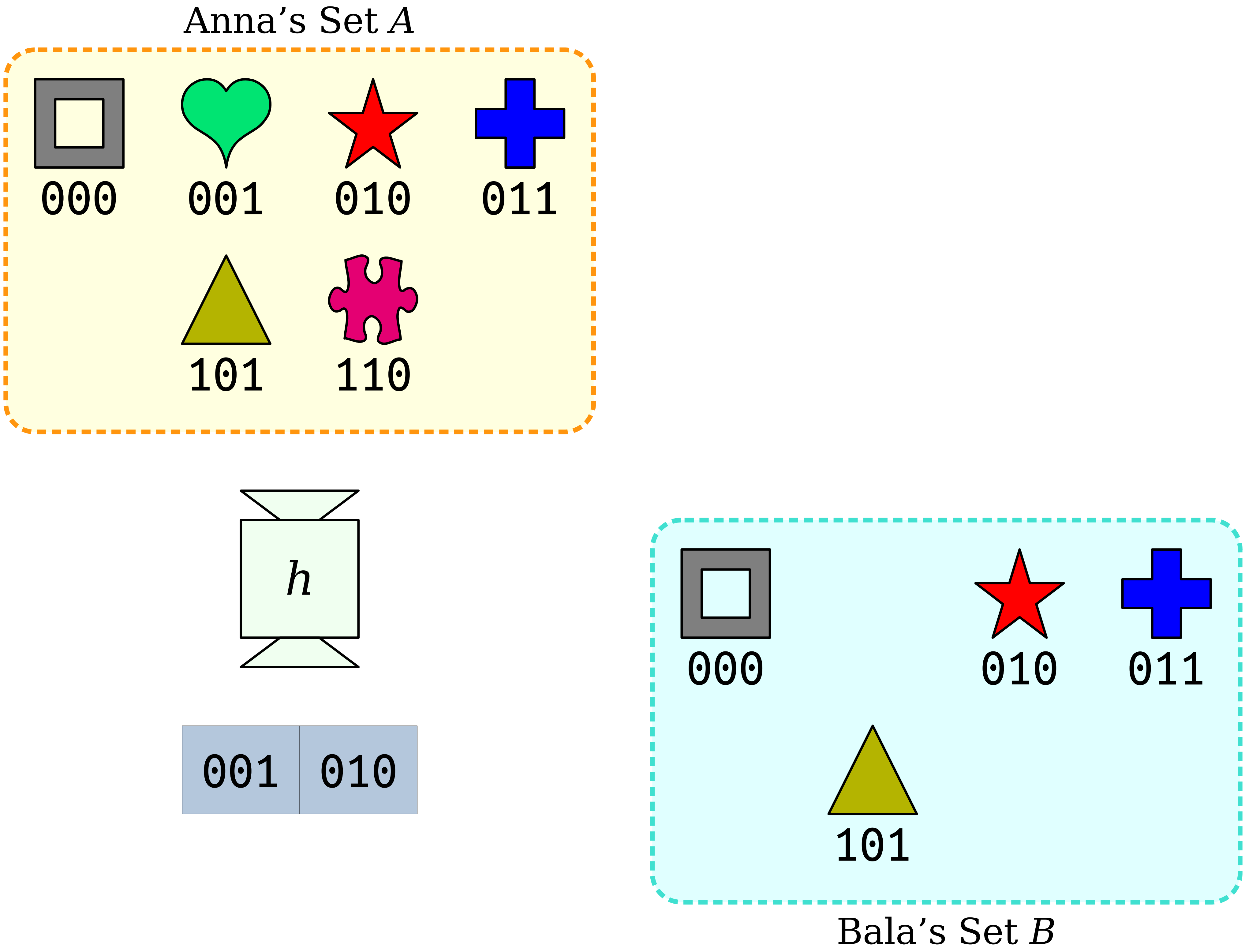

Anna’s Set *A*
000
001
010
011
101
110
h
001
001
000
010
011
101
Bala’s Set *B*

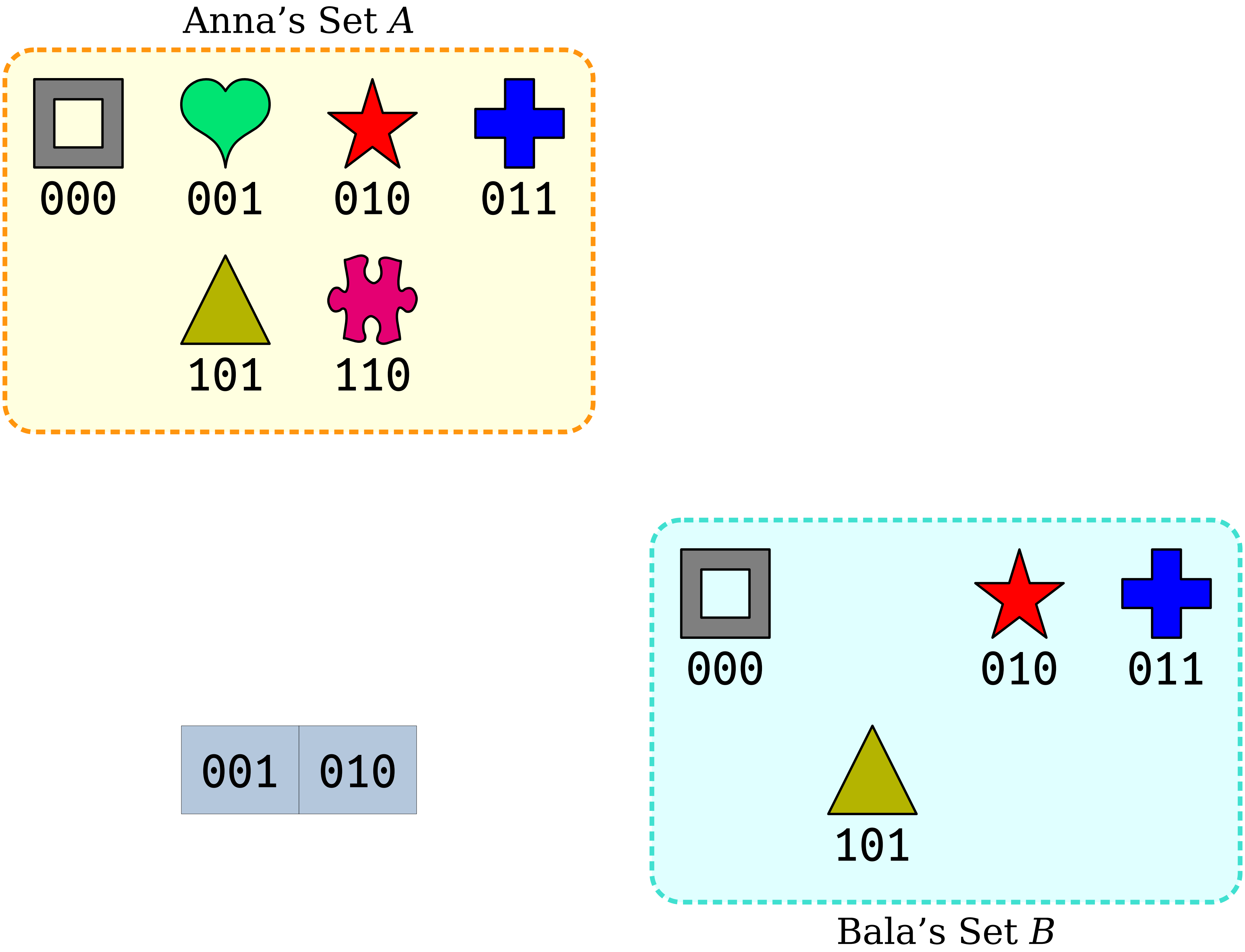

Anna’s Set A
000
001
010
011
101
110
h
001
001
000
010
011
101
Bala’s Set B

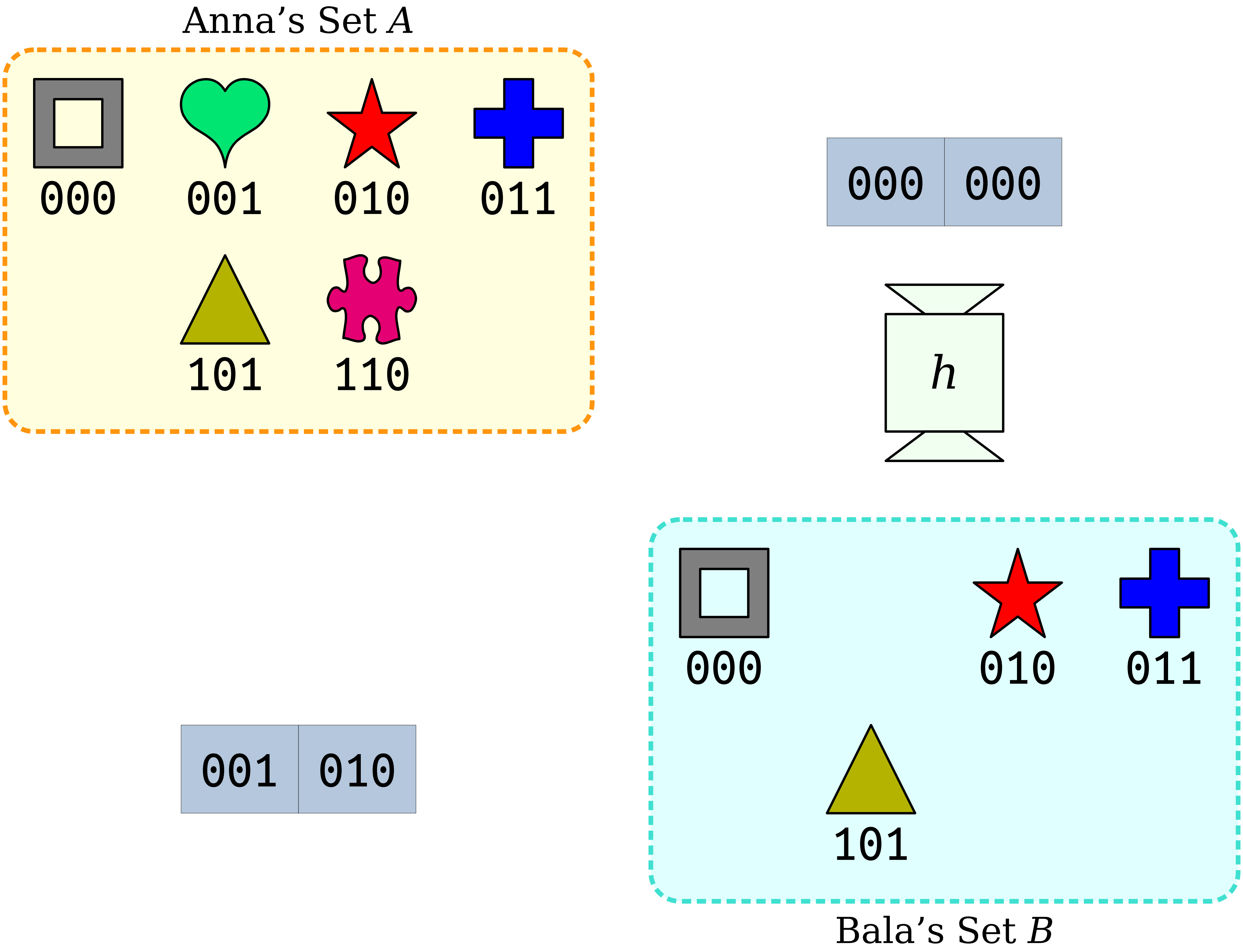

Anna’s Set A
000
001
010
011
101
110
h
001
100
000
010
011
101
Bala’s Set B

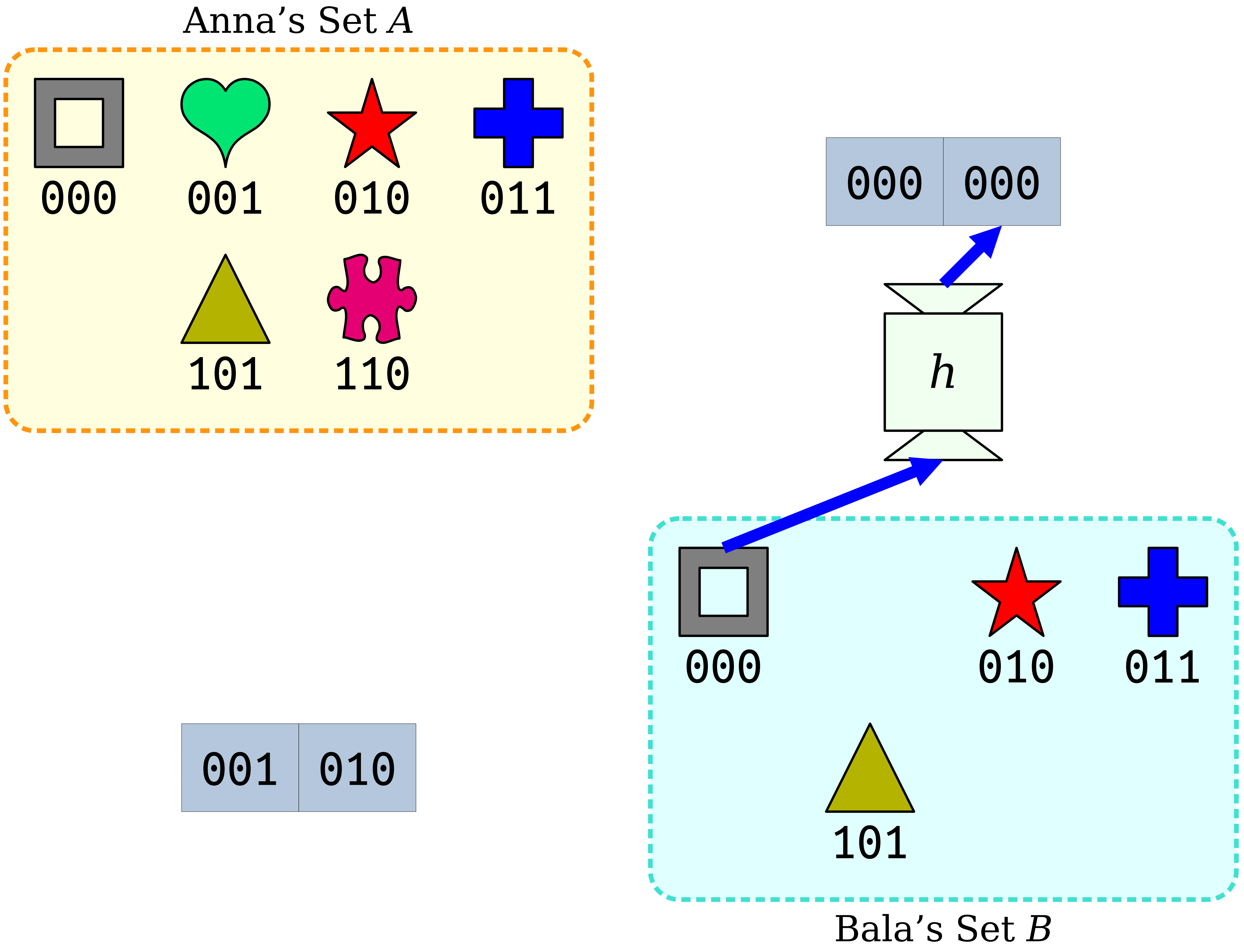

Anna's Set A
000
001
010
011
101
110
h
001
100
000
010
011
101
Bala's Set B

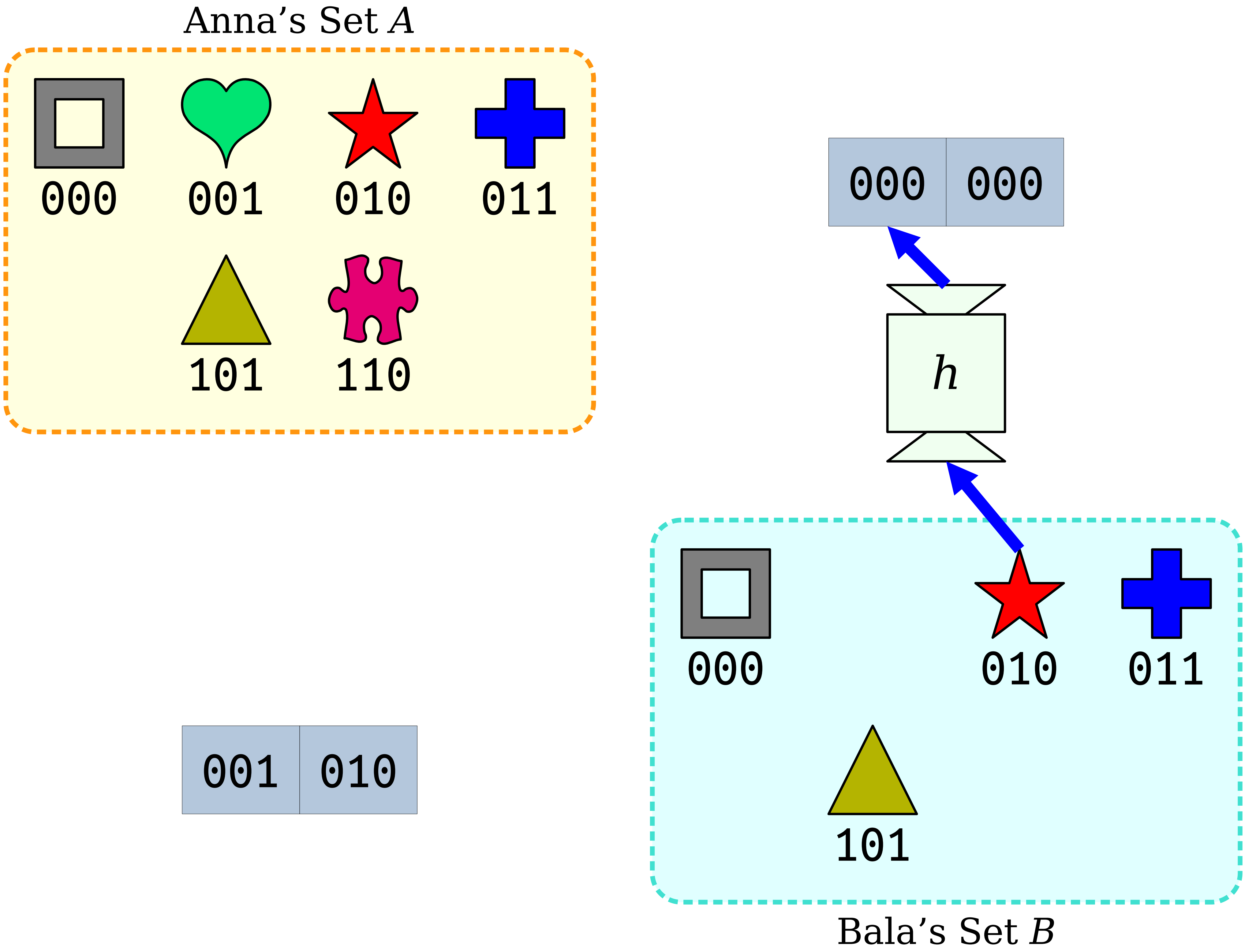

Anna's Set A
000
001
010
011
101
110
h
001
010
000
010
011
101
Bala's Set B

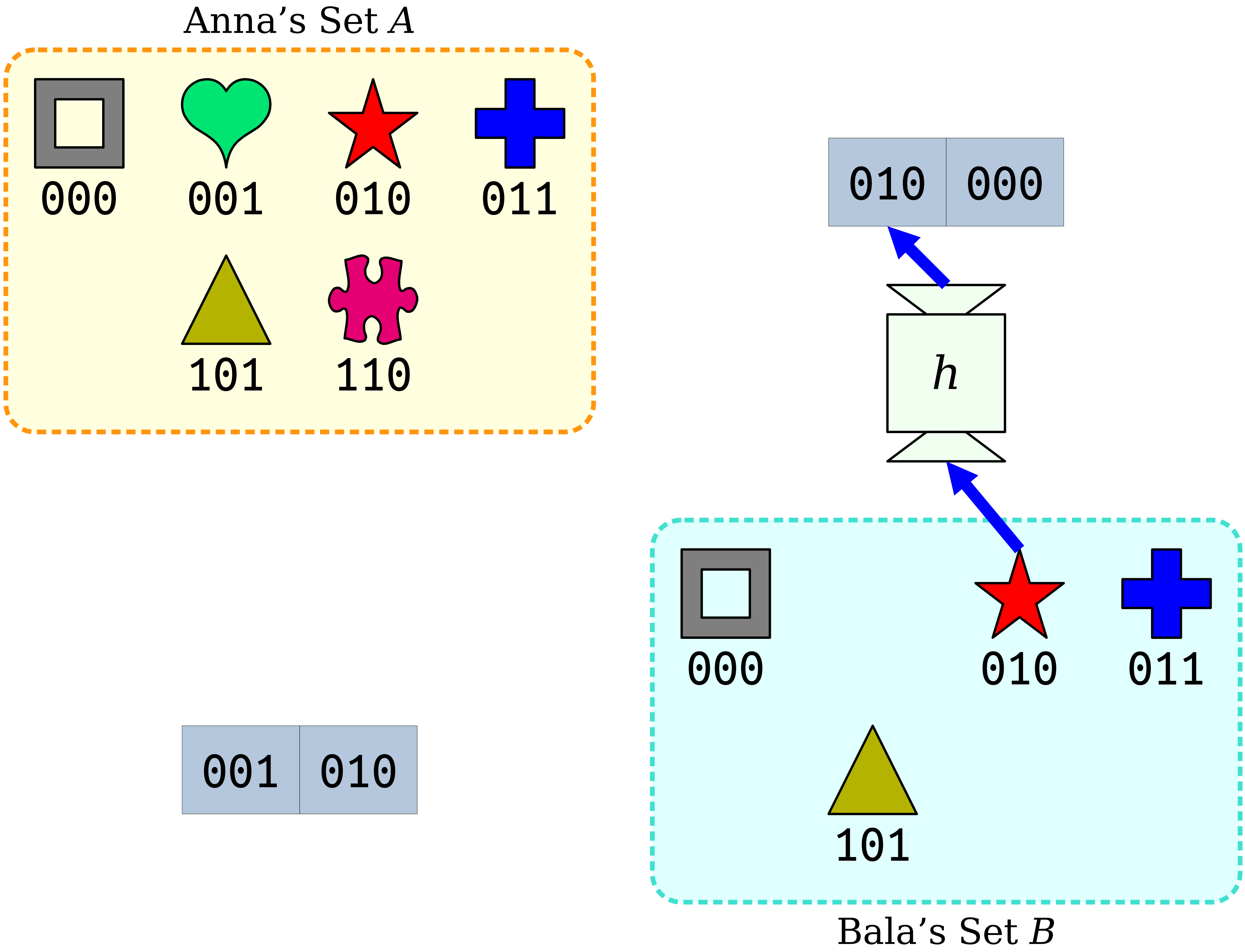

Anna's Set A
000
001
010
011
101
110
h
001
010
000
010
011
101
Bala's Set B

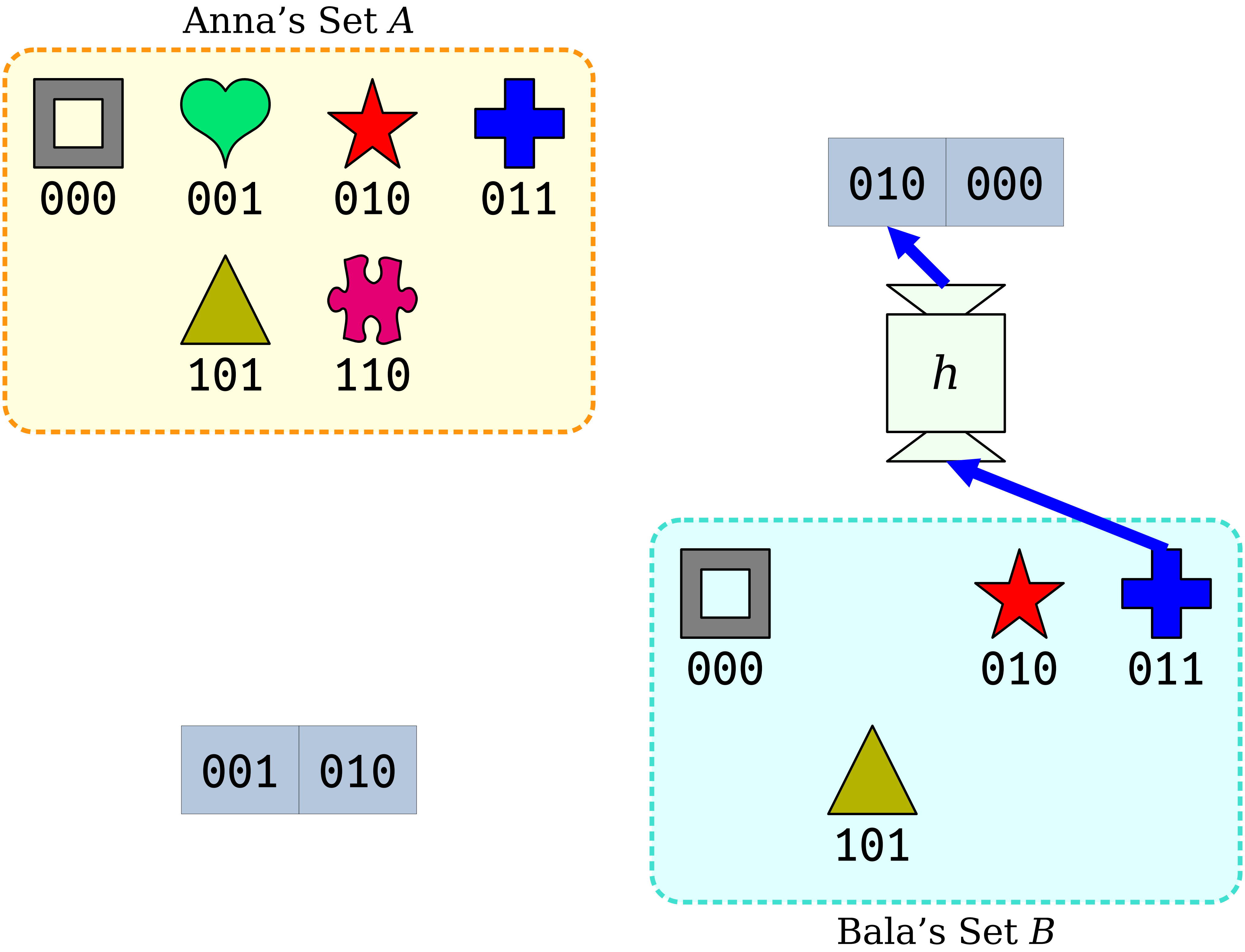

Anna’s Set A
000
001
010
011
101
110
001
010
000
010
011
101
Bala’s Set B

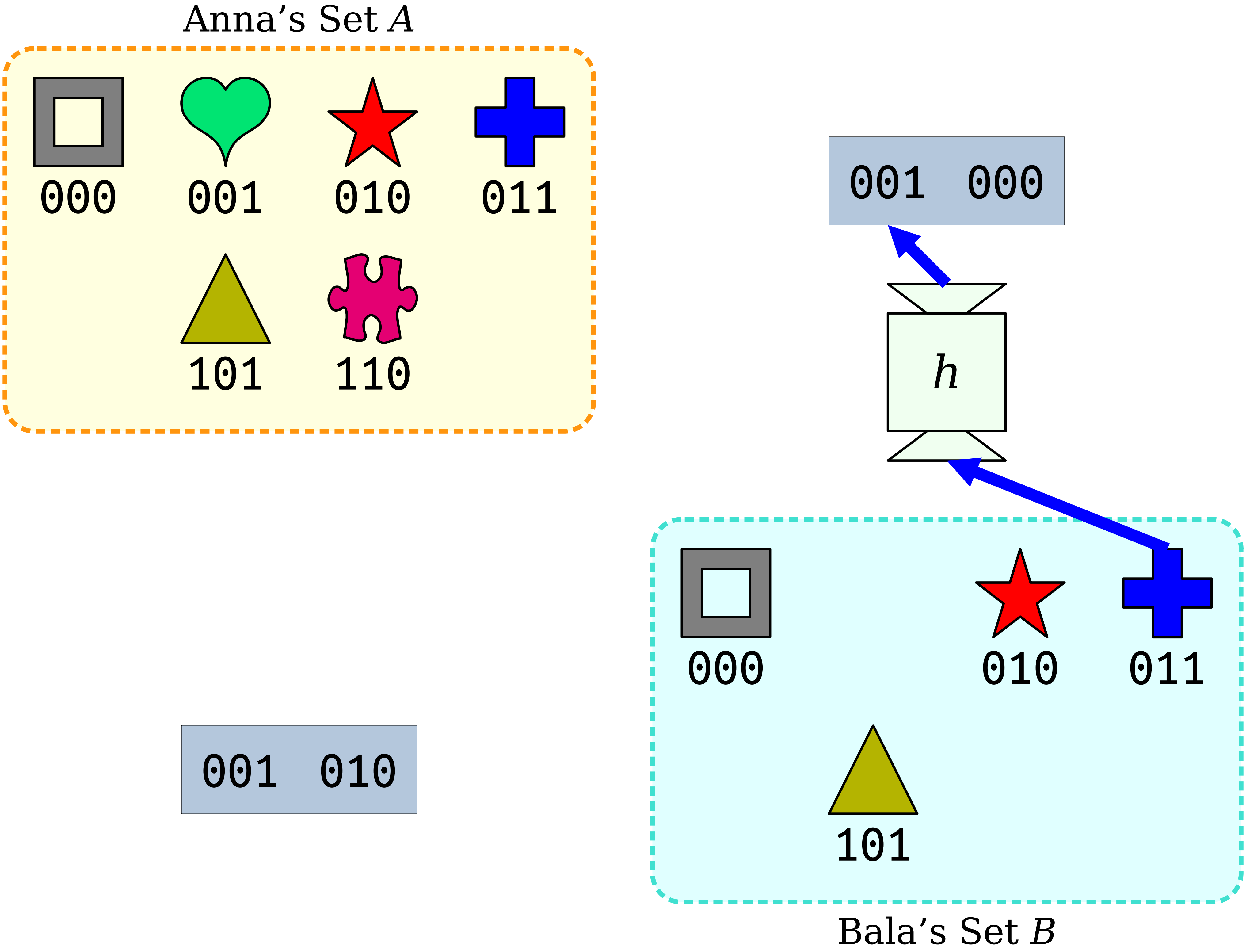

Anna’s Set A
000
001
010
011
101
110
000 000
h
001 010
000
010
011
101
Bala’s Set B

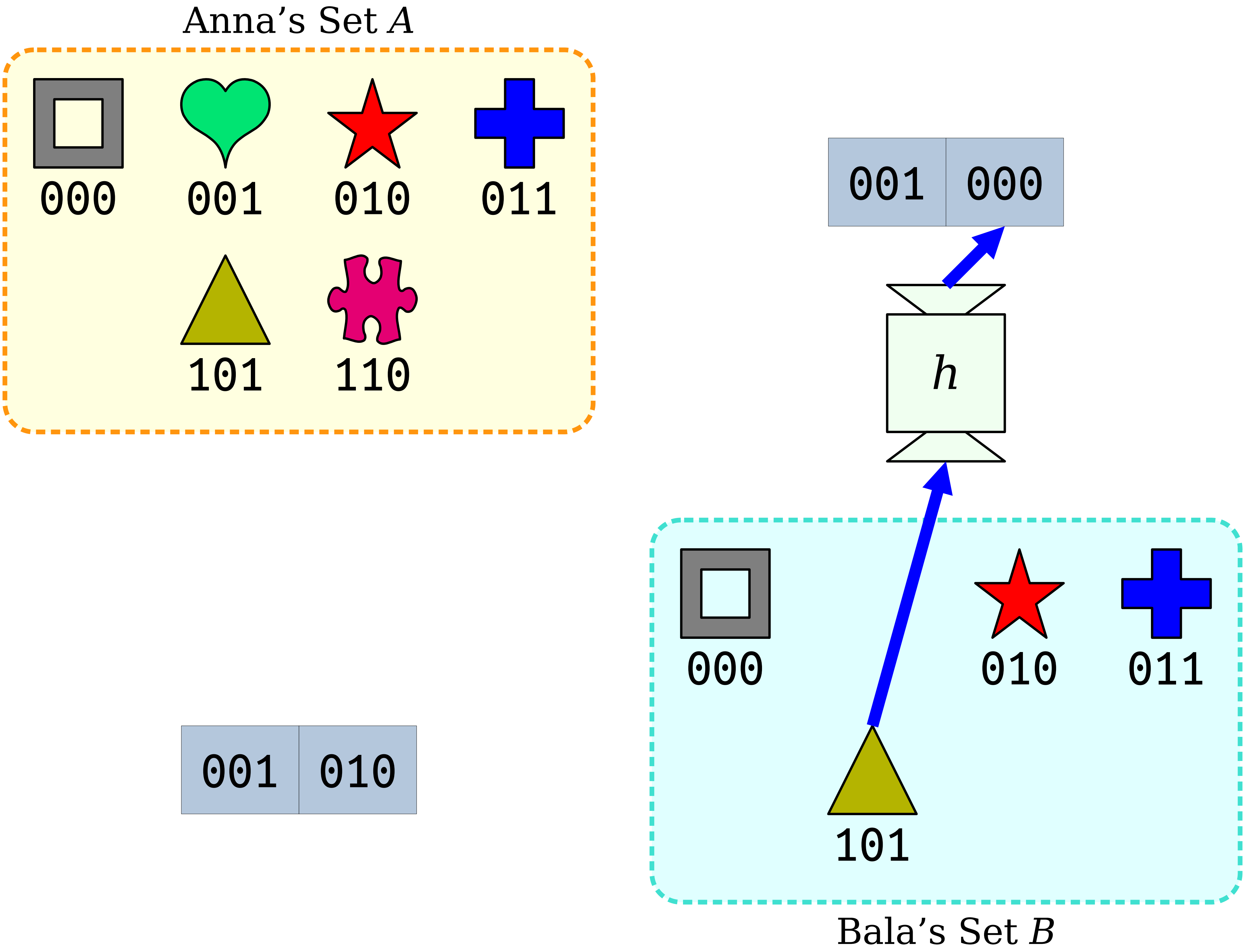

Anna’s Set A
000
001
010
011
101
110
000 | 000
h
001 | 010
000
010
011
101
Bala’s Set B

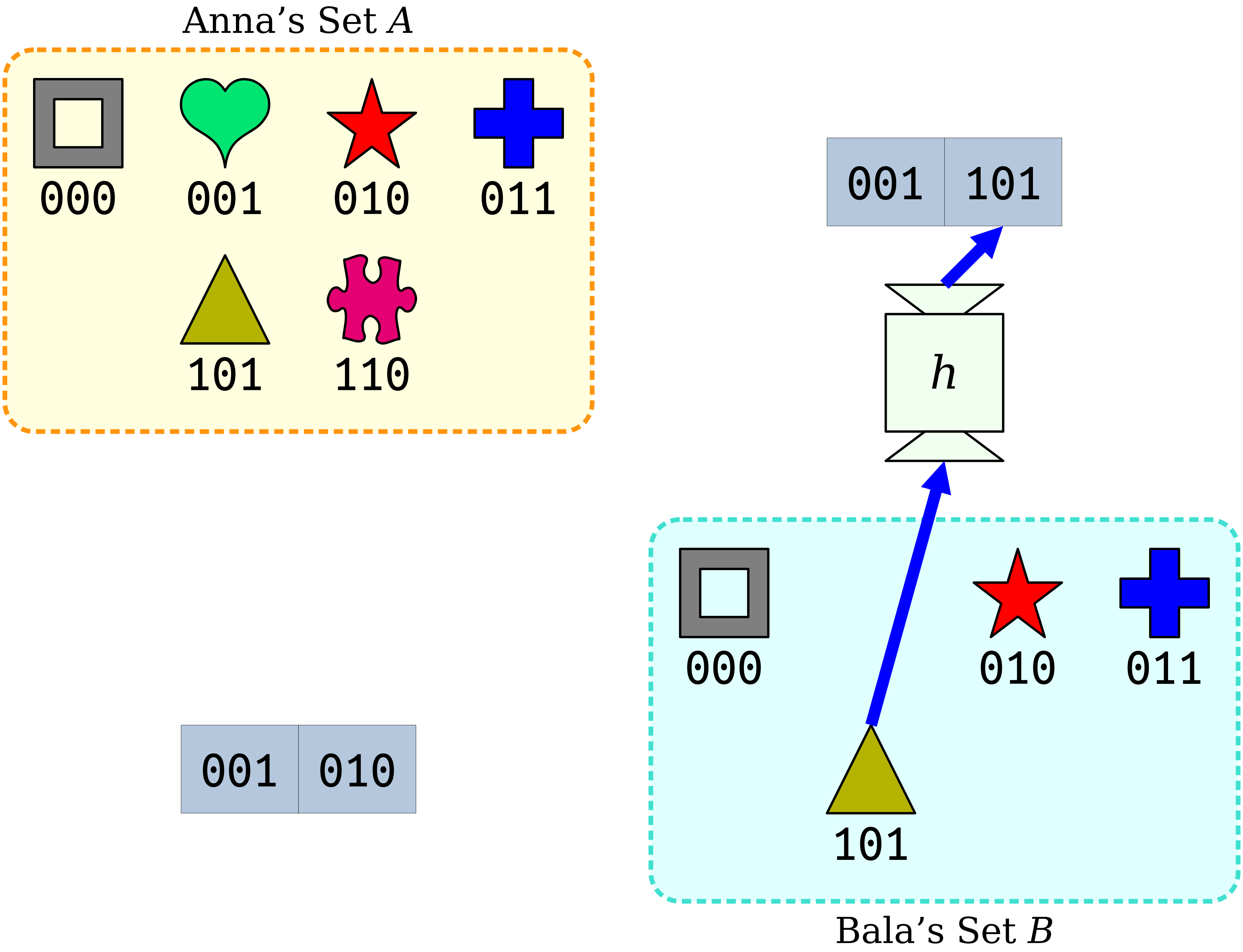

Anna’s Set A
000
001
010
011
101
110
000 000
h
001 010
000
010
011
101
Bala’s Set B

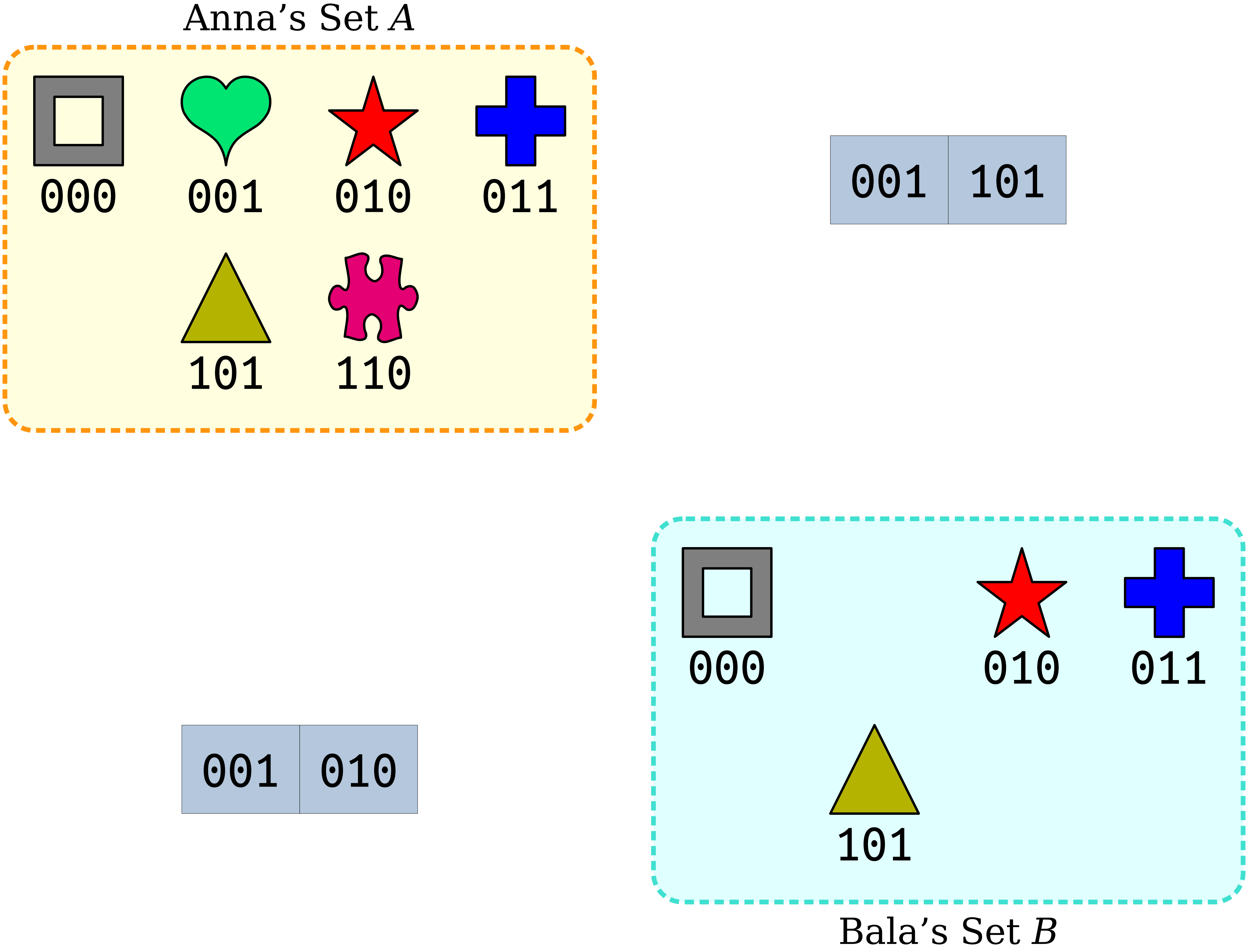

Anna's Set A
000
001
010
011
101
110
010
000
h
001
010
000
010
011
101
Bala's Set B


Anna’s Set A
000
001
010
011
101
110
010
000
h
000
010
011
101
001
010
Bala’s Set B

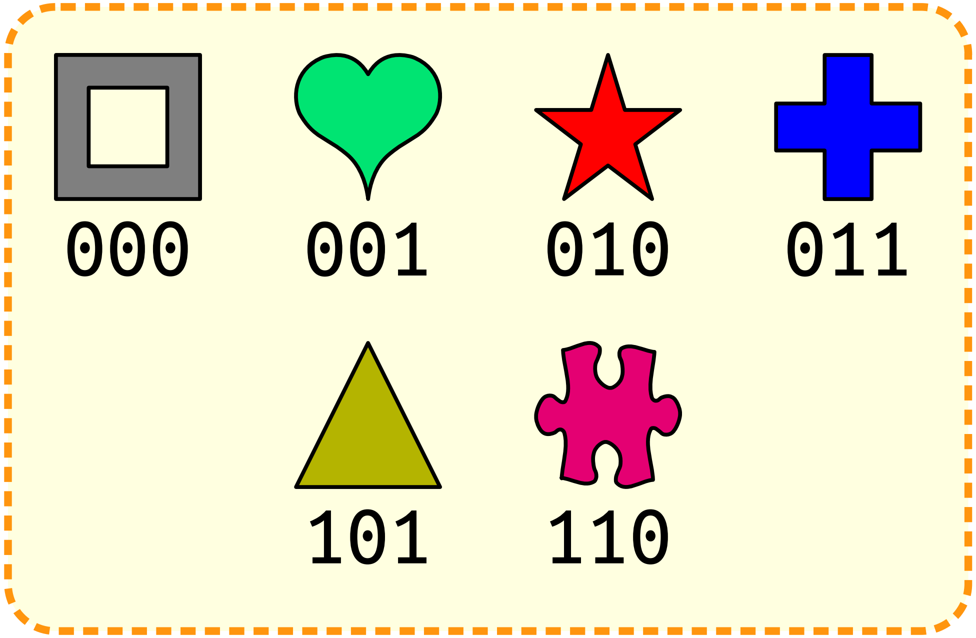

Anna's Set A
000
001
010
011
101
110
001 000
h
000
010
011
101
001 010
Bala's Set B


Anna’s Set A
000
001
010
011
101
110
001
000
h
001
010
000
010
011
101
Bala’s Set B

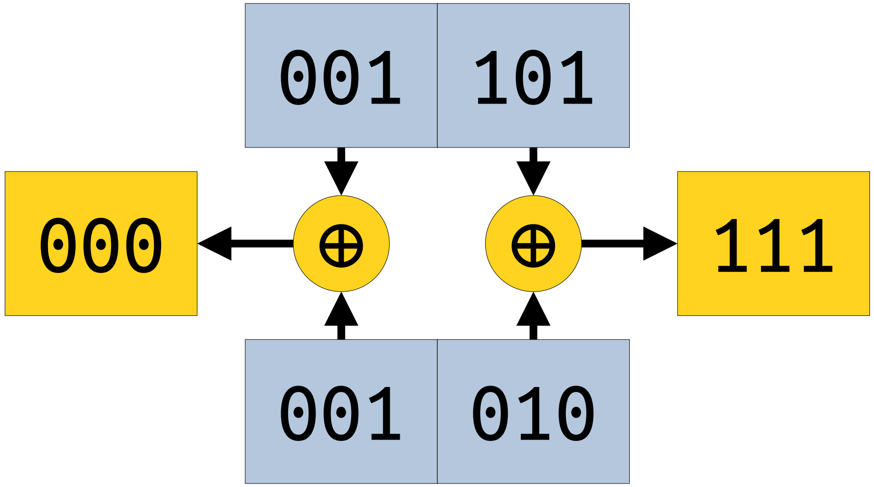

Anna's Set A
000
001
010
011
101
110
001
010
h
001
101
000
010
011
101
Bala's Set B

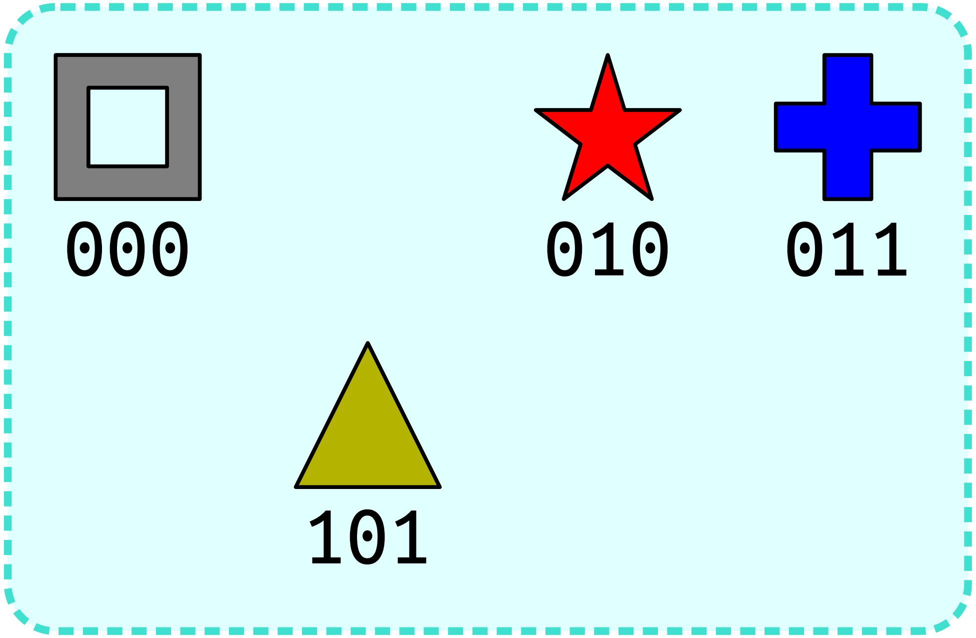

Anna's Set A
000
001
010
011
101
110
001
101
001
010
000
010
011
101
Bala's Set B

Anna’s Set A

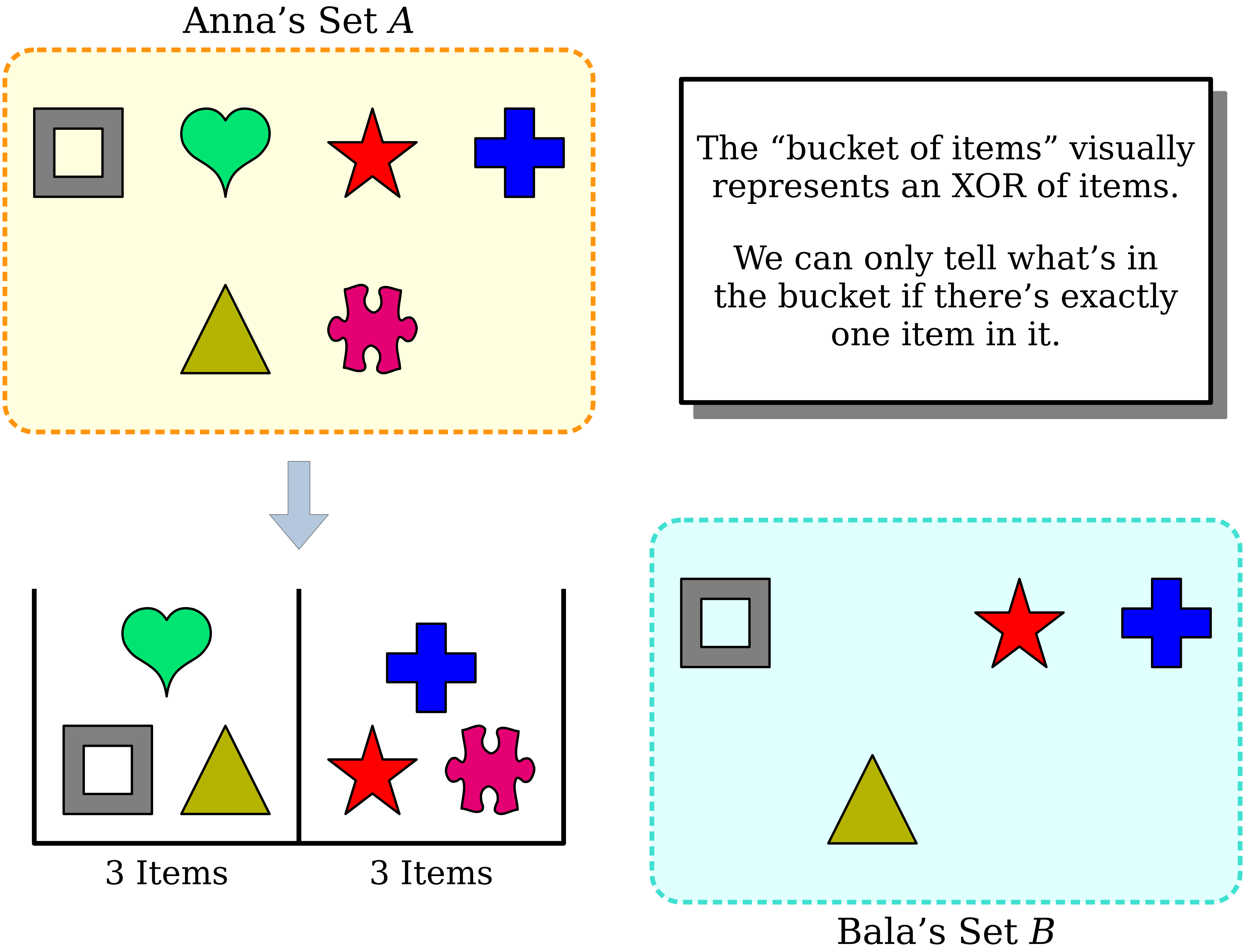


Neither party is guaranteed to know whether these are two separate items or a 0 and an XOR of two.





Bala’s Set B



Anna's Set A

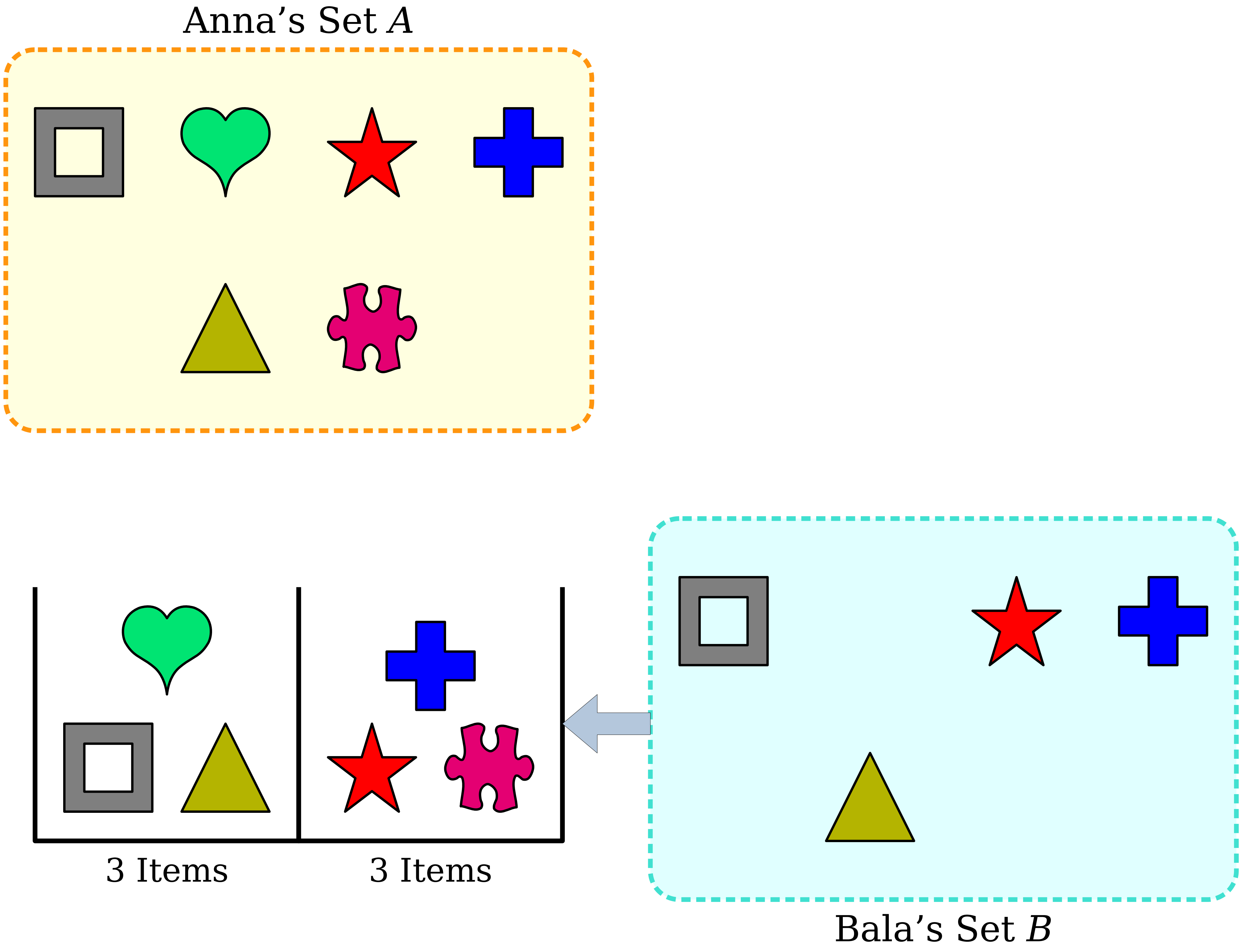
The "bucket of items" visually represents an XOR of items.

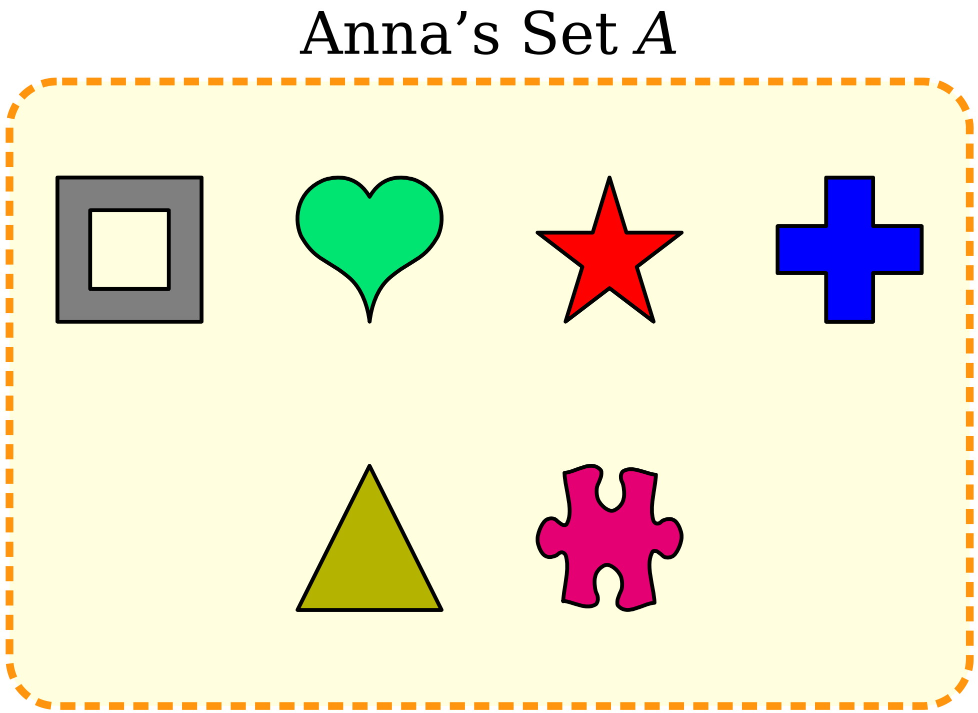
We can only tell what's in the bucket if there's exactly one item in it.

3 Items

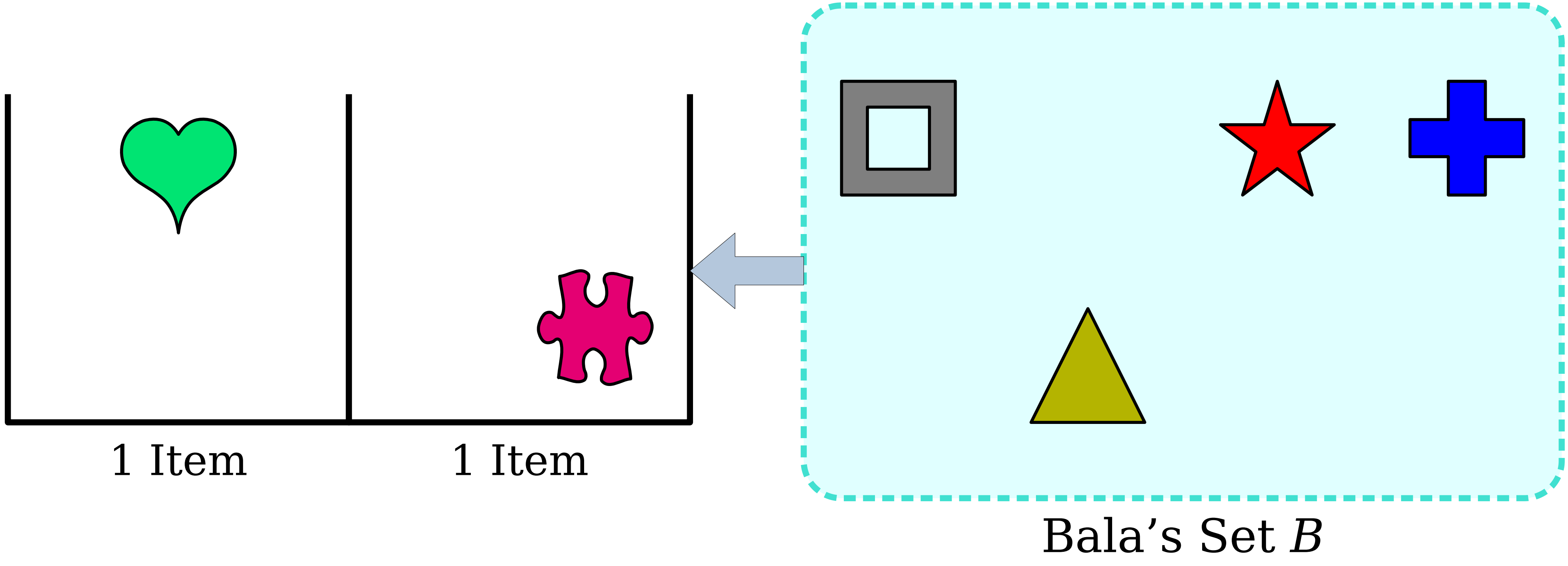
3 Items

Bala's Set B


Anna's Set A
3 Items
3 Items
Bala's Set B



Now the parties can tell whether the decoding was successful.

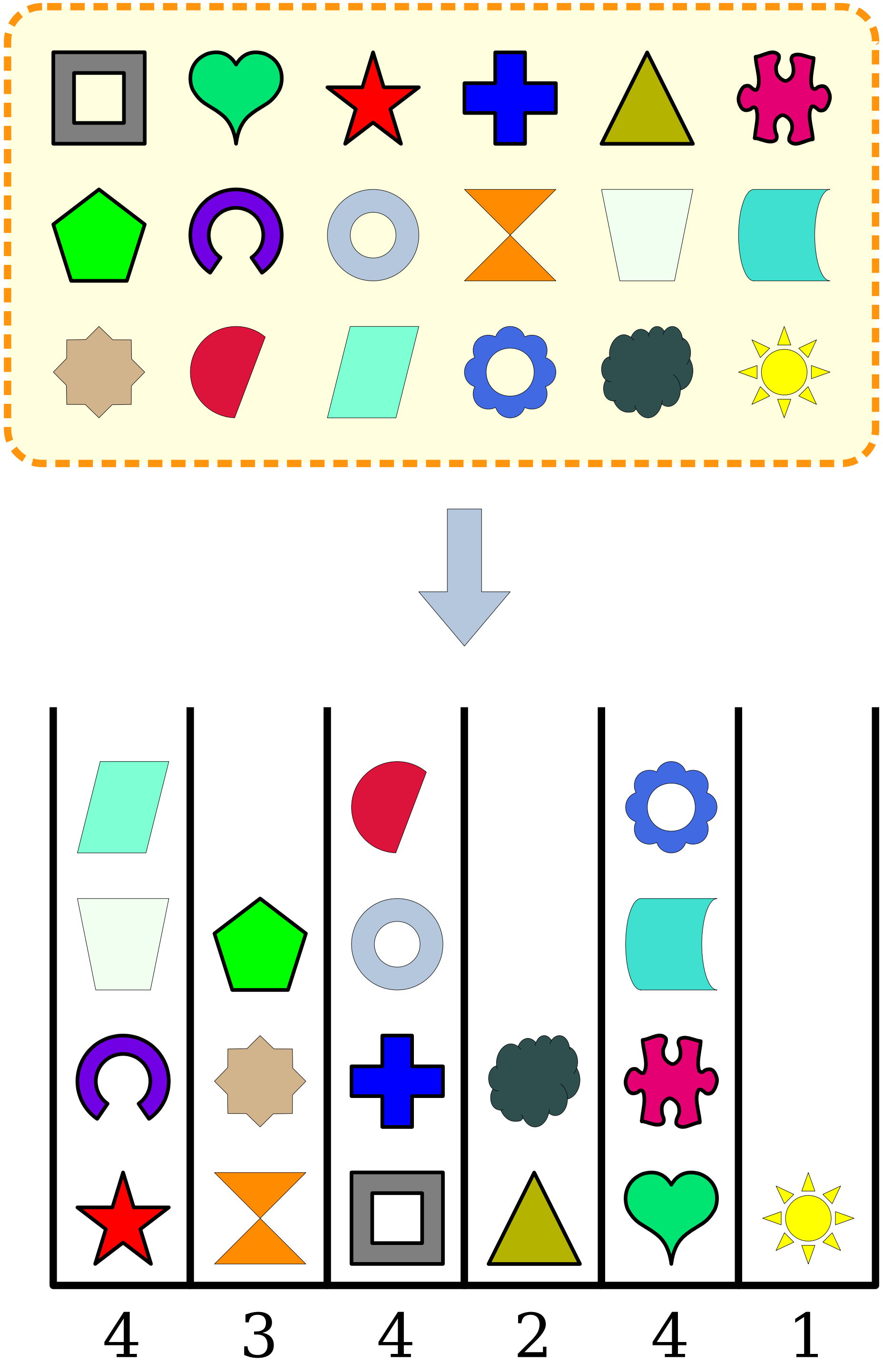


The k=2 Case

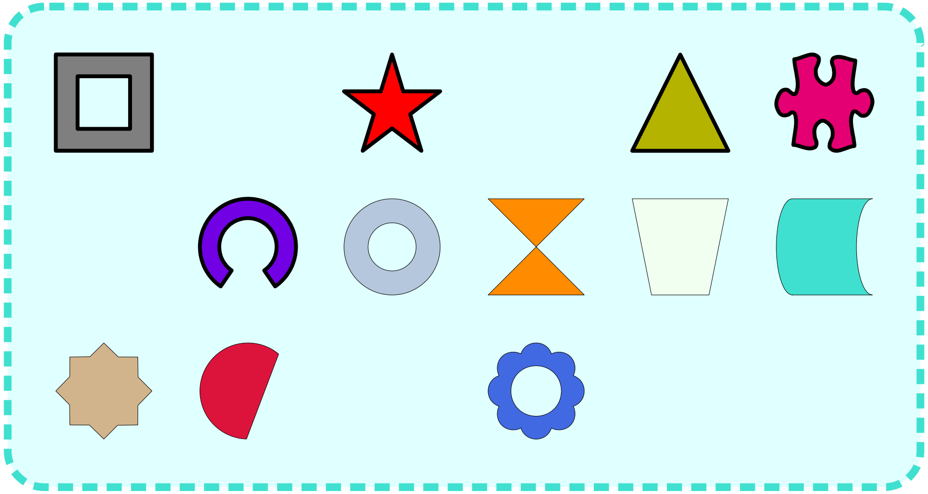
- This approach has a 50% chance of success.
  - (There are two ways to assign the items to distinct buckets and four possible assignments.)
- We can boost the success probability by replicating this approach multiple times with independent hash functions.
- With lg δ^{-1} replicated copies, we boost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

The General Case

Anna’s Set A

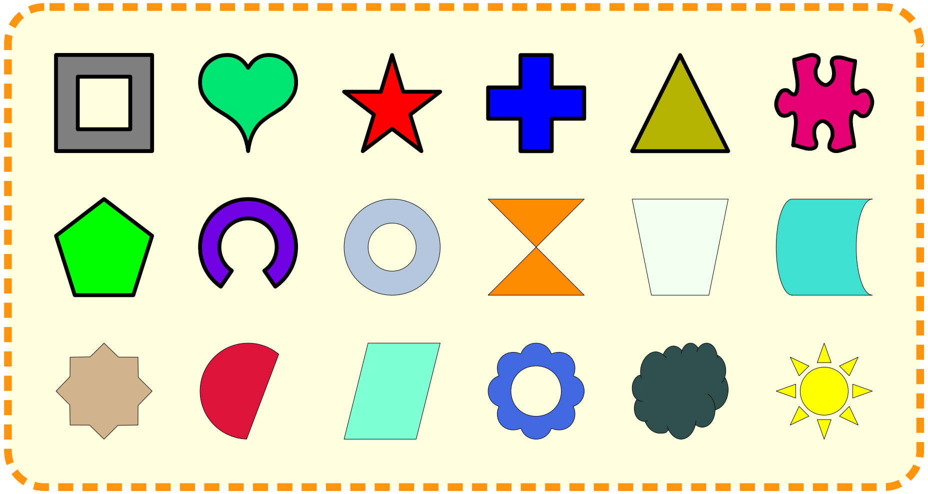


Assume Anna and Bala know what k is.
(How might they do that?)

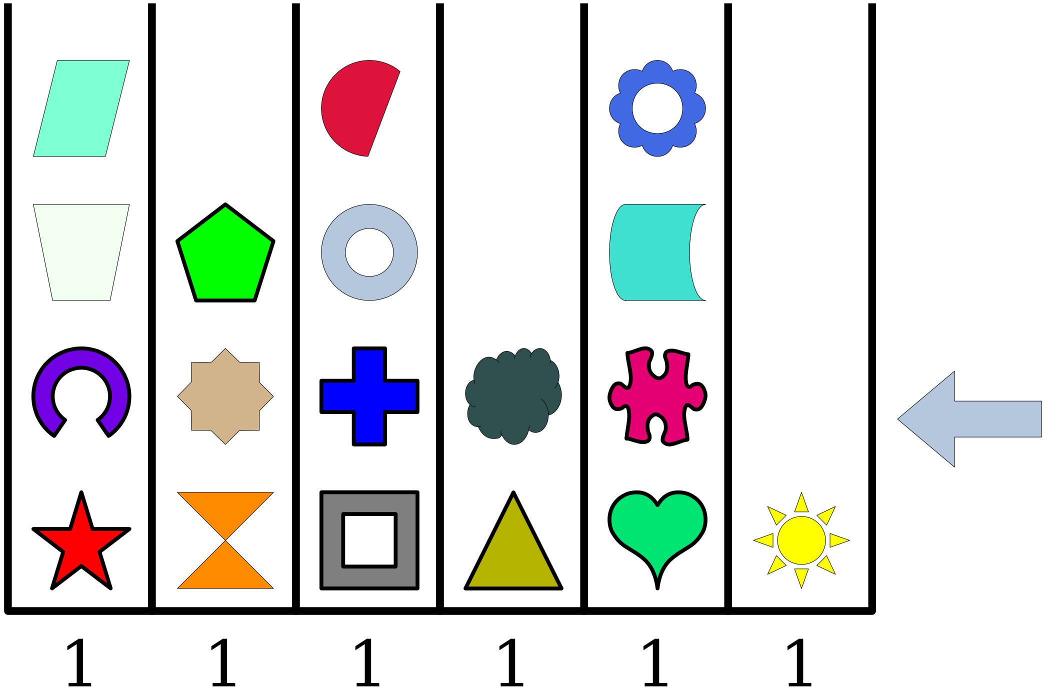


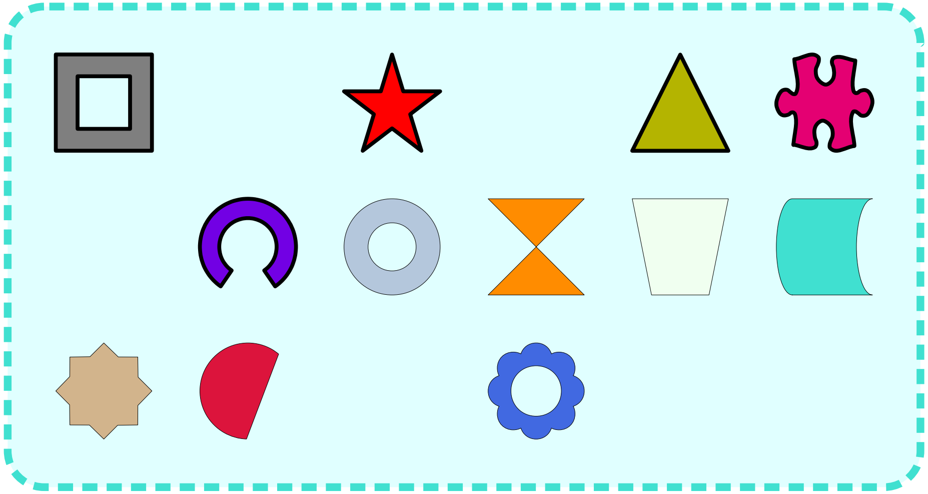
Bala’s Set B

Anna's Set A

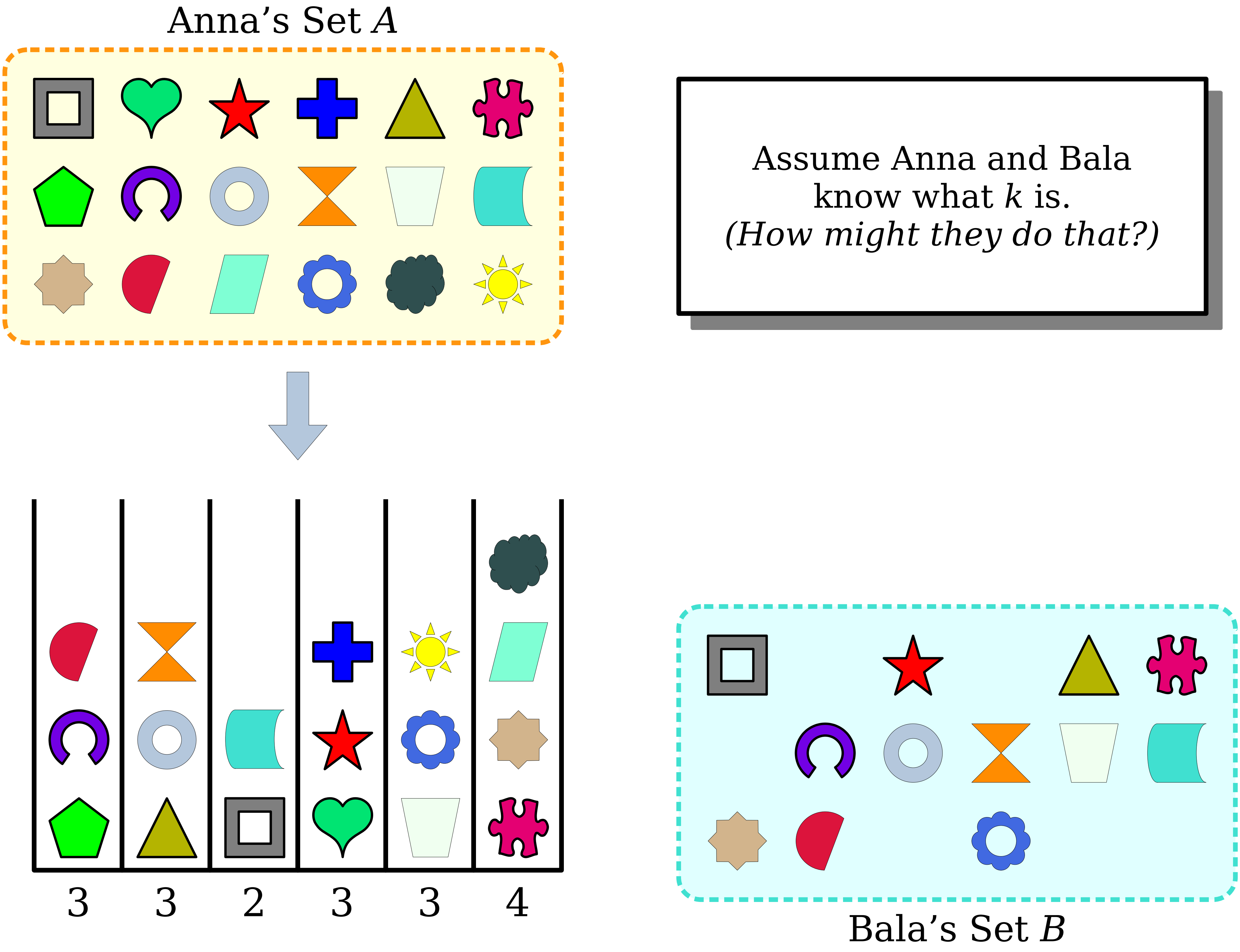


Assume Anna and Bala know what k is.
(How might they do that?)





Bala's Set B



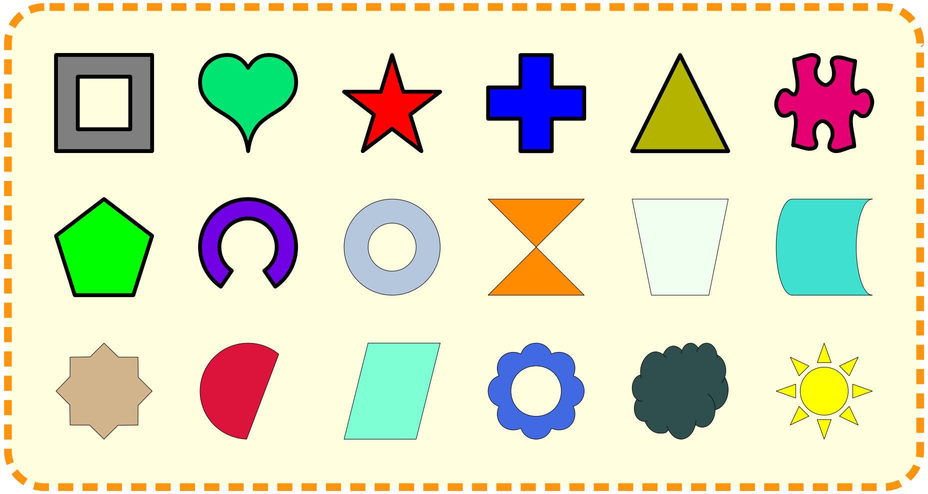
Anna’s Set *A*

Assume Anna and Bala know what k is.
(How might they do that?)

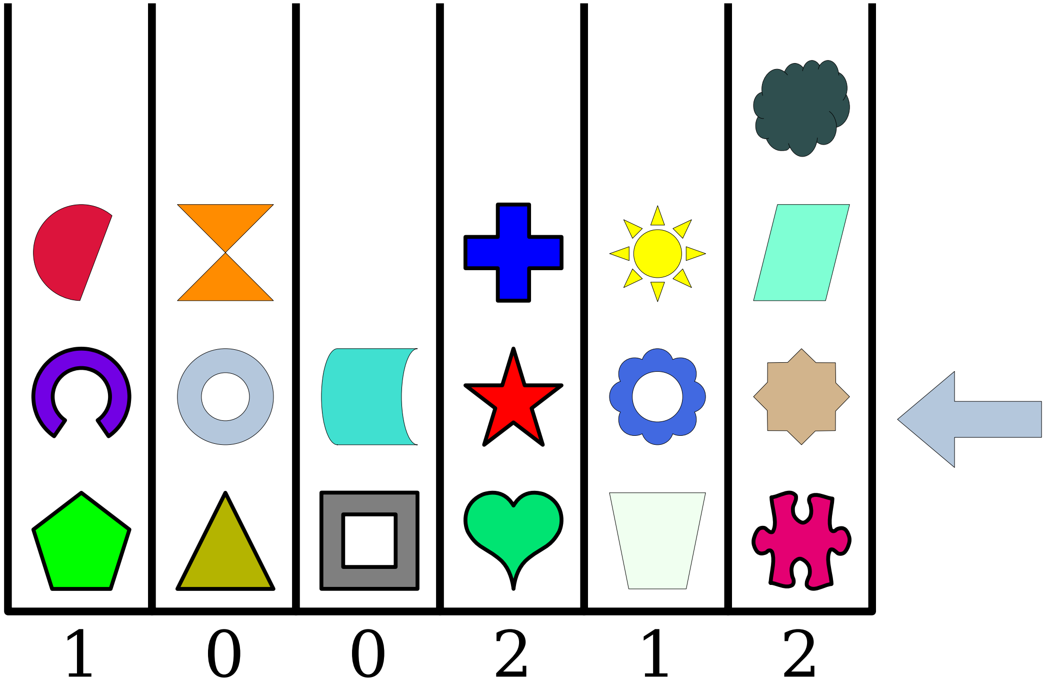
3 3 2 3 3 4

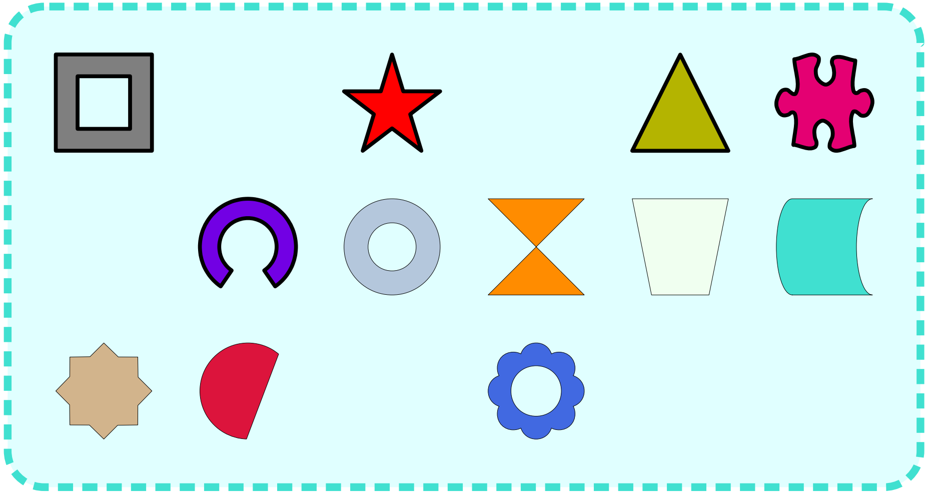
Bala’s Set *B*

Anna’s Set *A*



Assume Anna and Bala know what k is.
(How might they do that?)





Bala’s Set *B*

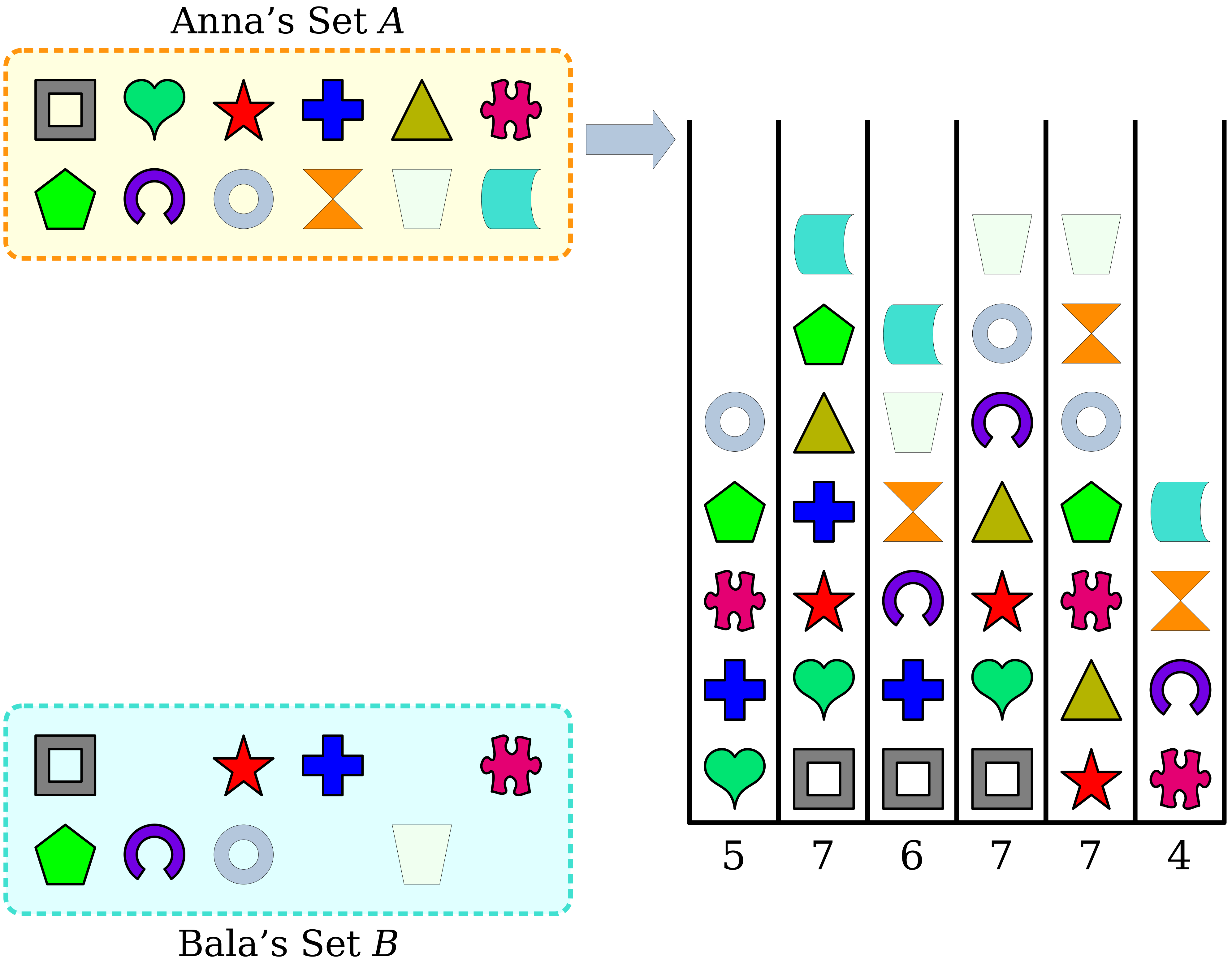
The General Case

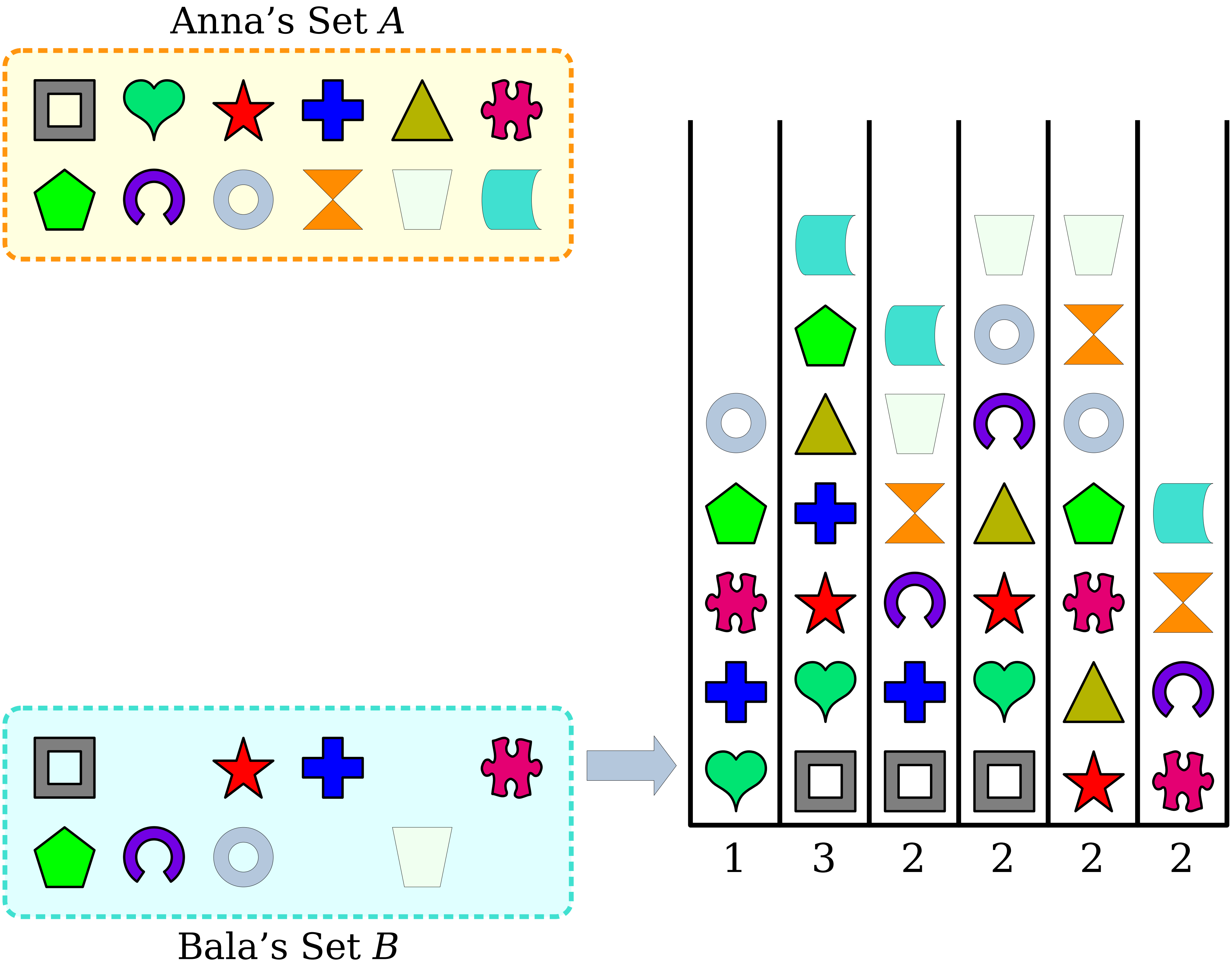
- Create an array of ***m*** buckets, all initially zero. Choose a hash function from elements to the set $\{0, 1, \ldots, m - 1\}$.
- Anna hashes her items and adds them to the appropriate buckets; Bala hashes his items and removes them.
- Anna and Bala can identify all items in buckets whose counts 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 **2*k* 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 **3.72*k*** 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.

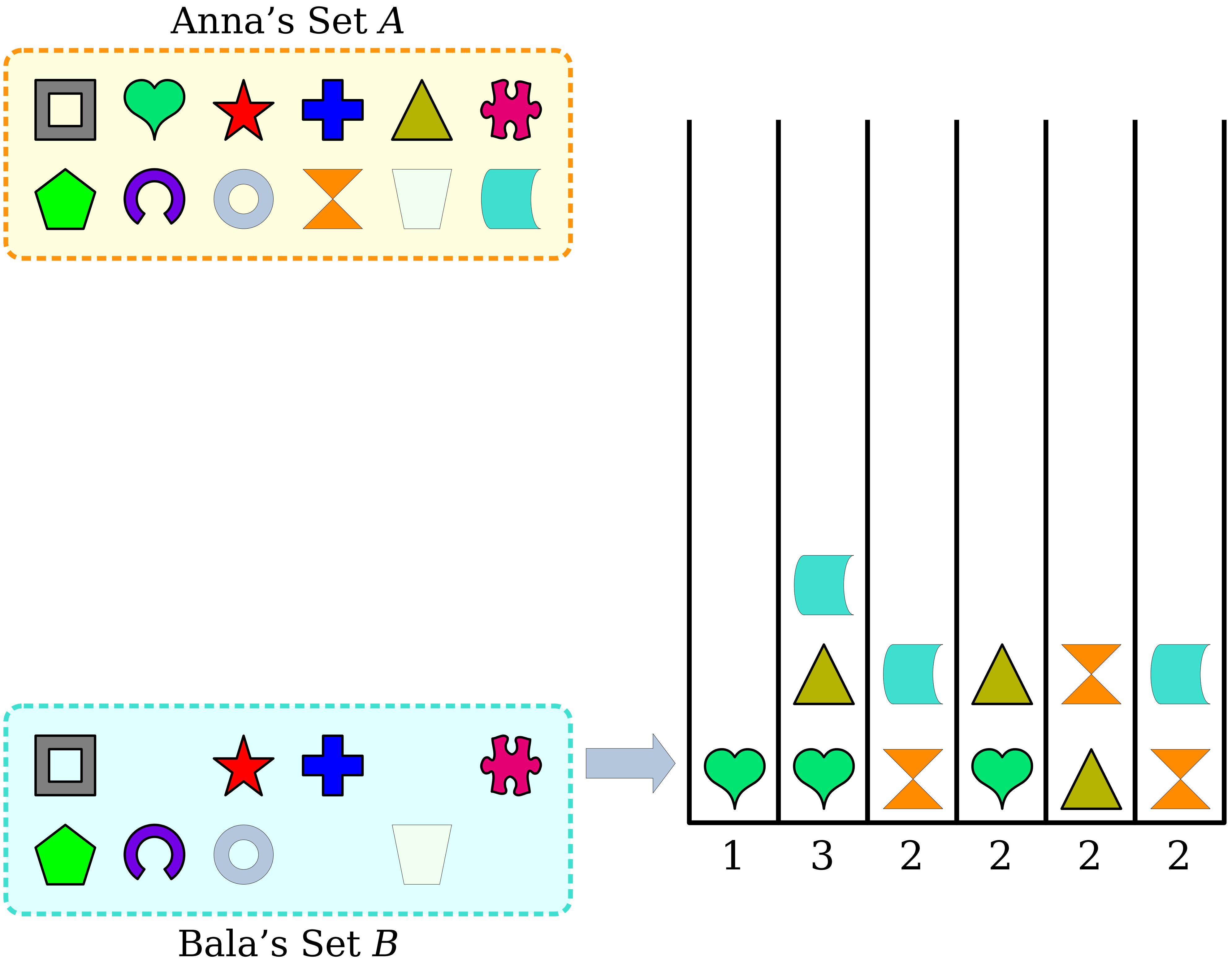
Taking a Step Back

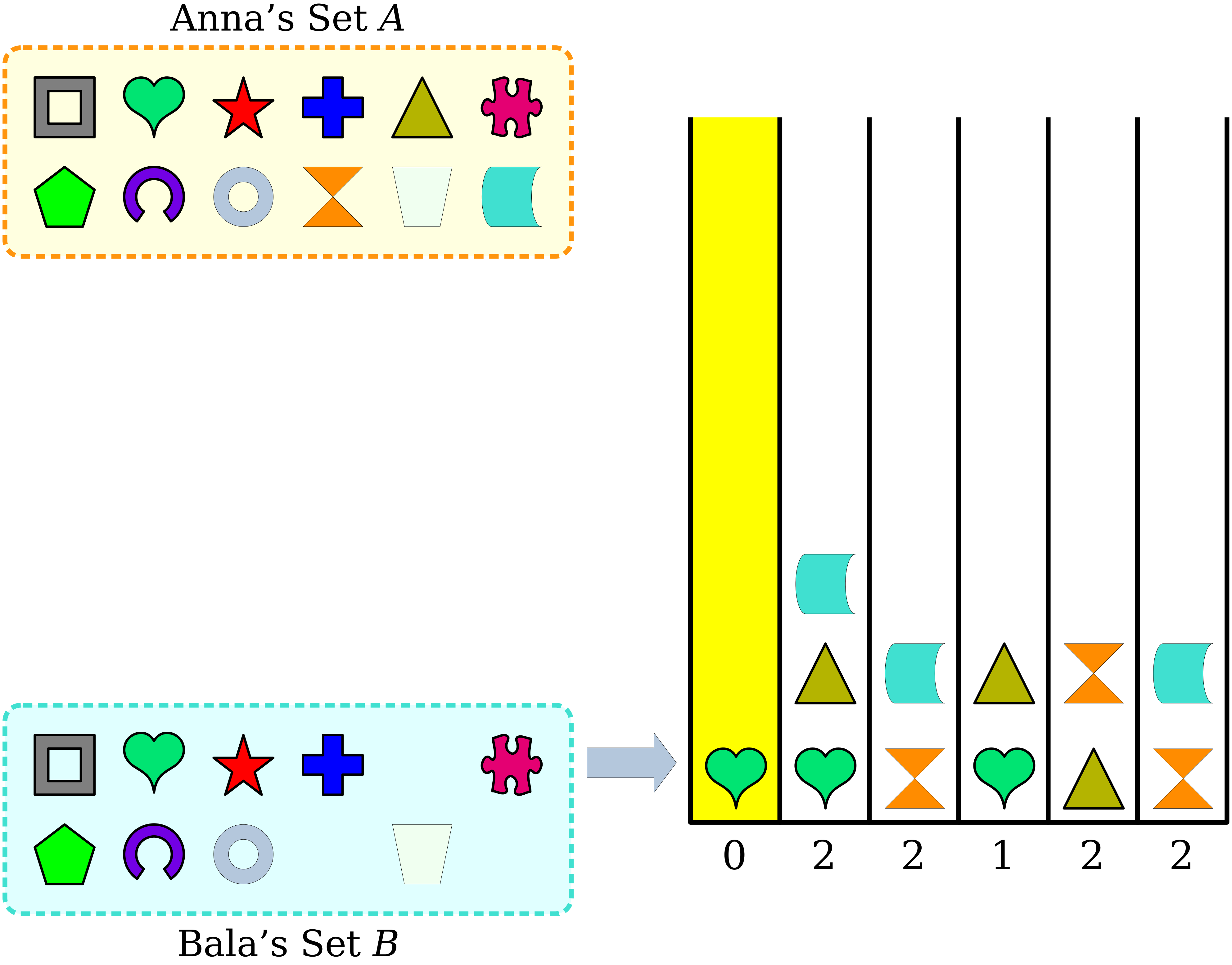
- We’re trying to find a way to hash items to buckets that doesn’t involve collisions.
- If you squint at this problem in just the right way, this kinda sorta ish looks like what cuckoo hashing was designed to solve.
- There are some major differences, though:
    - Our available memory is *way* smaller 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?

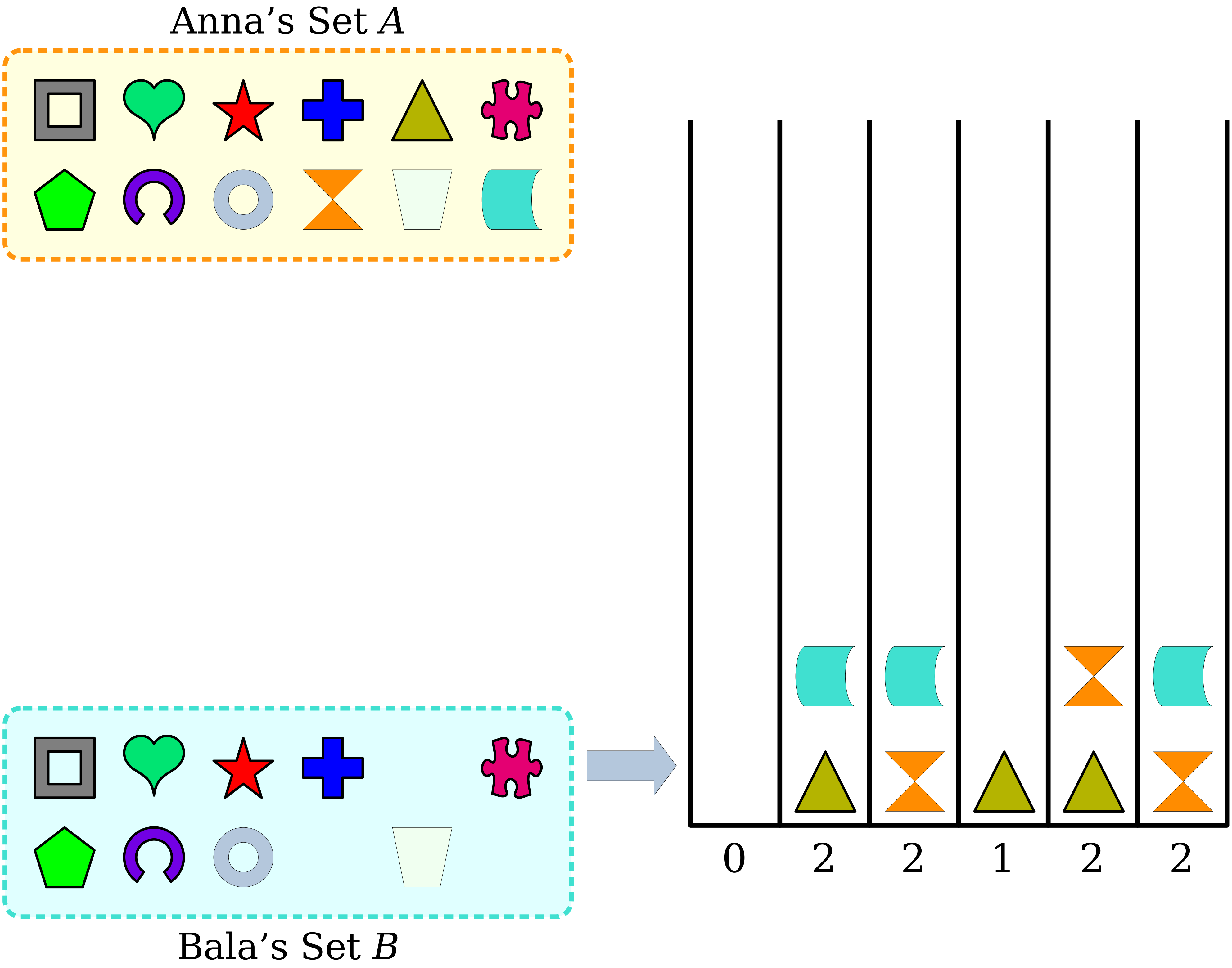
Idea: Hash each item to $d \geq 2$ buckets.

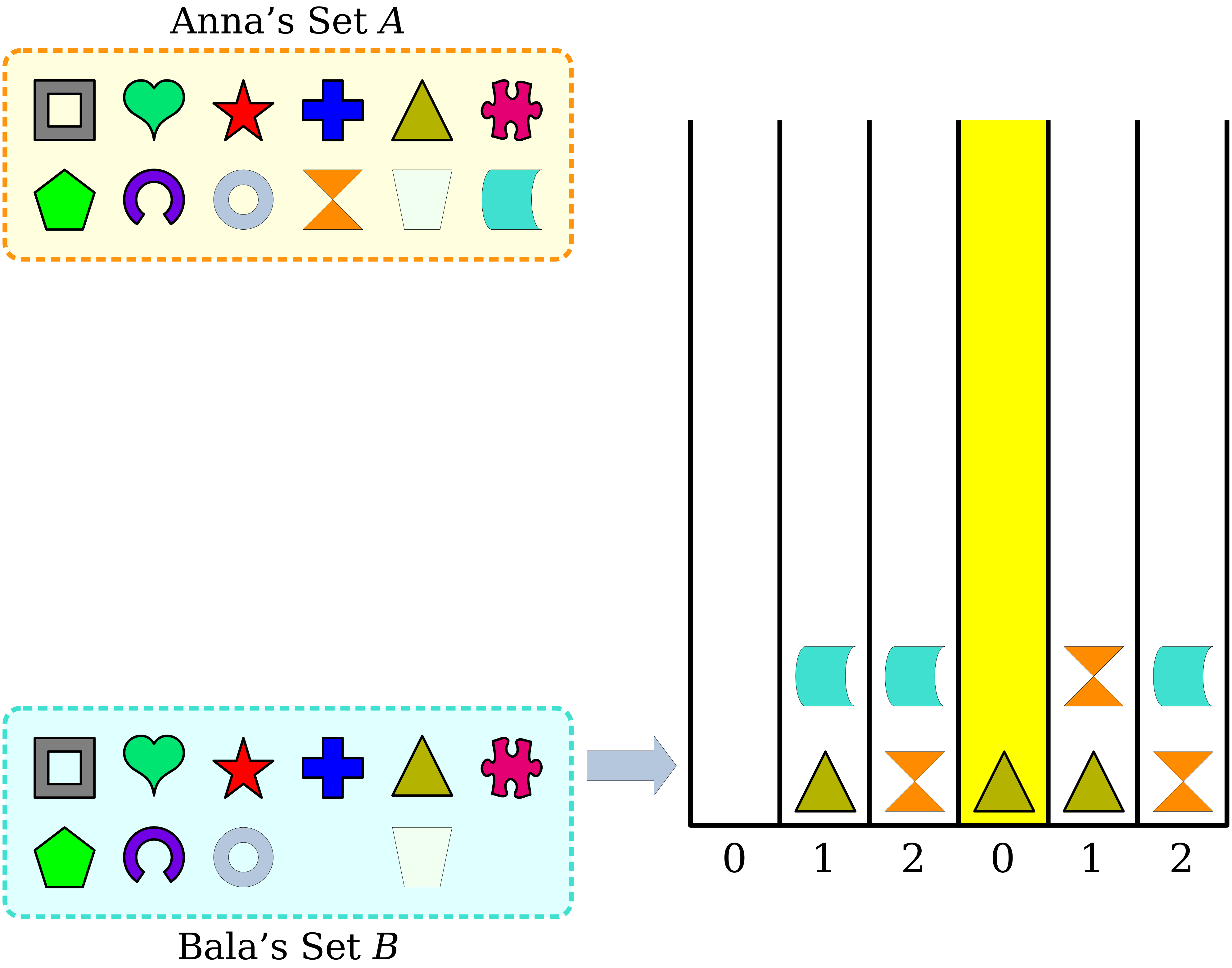

Anna's Set A
5
7
6
7
7
4
Bala's Set B

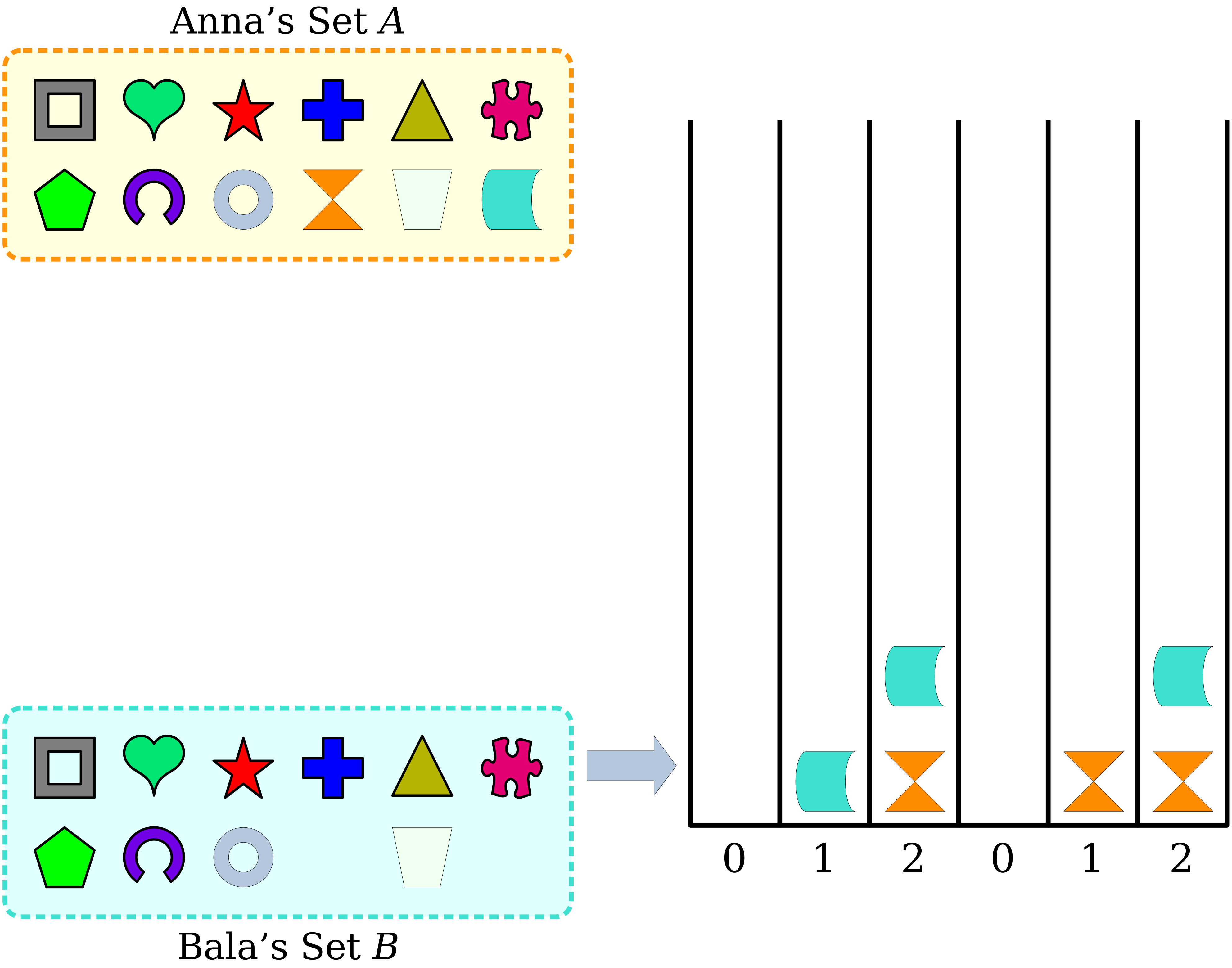

Anna’s Set A
Bala’s Set B
1
3
2
2
2
2

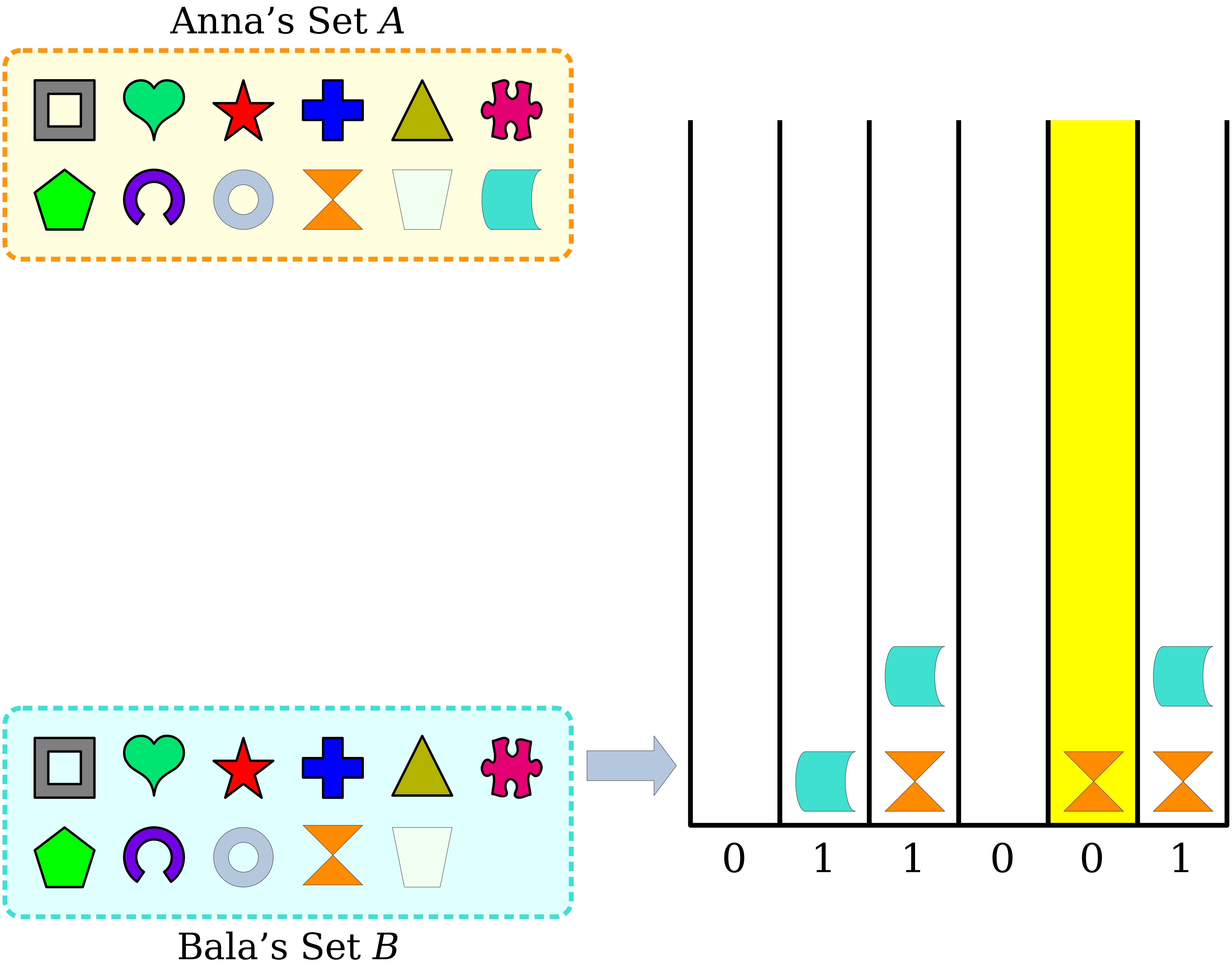

Anna’s Set A
Bala’s Set B
1
3
2
2
2
2

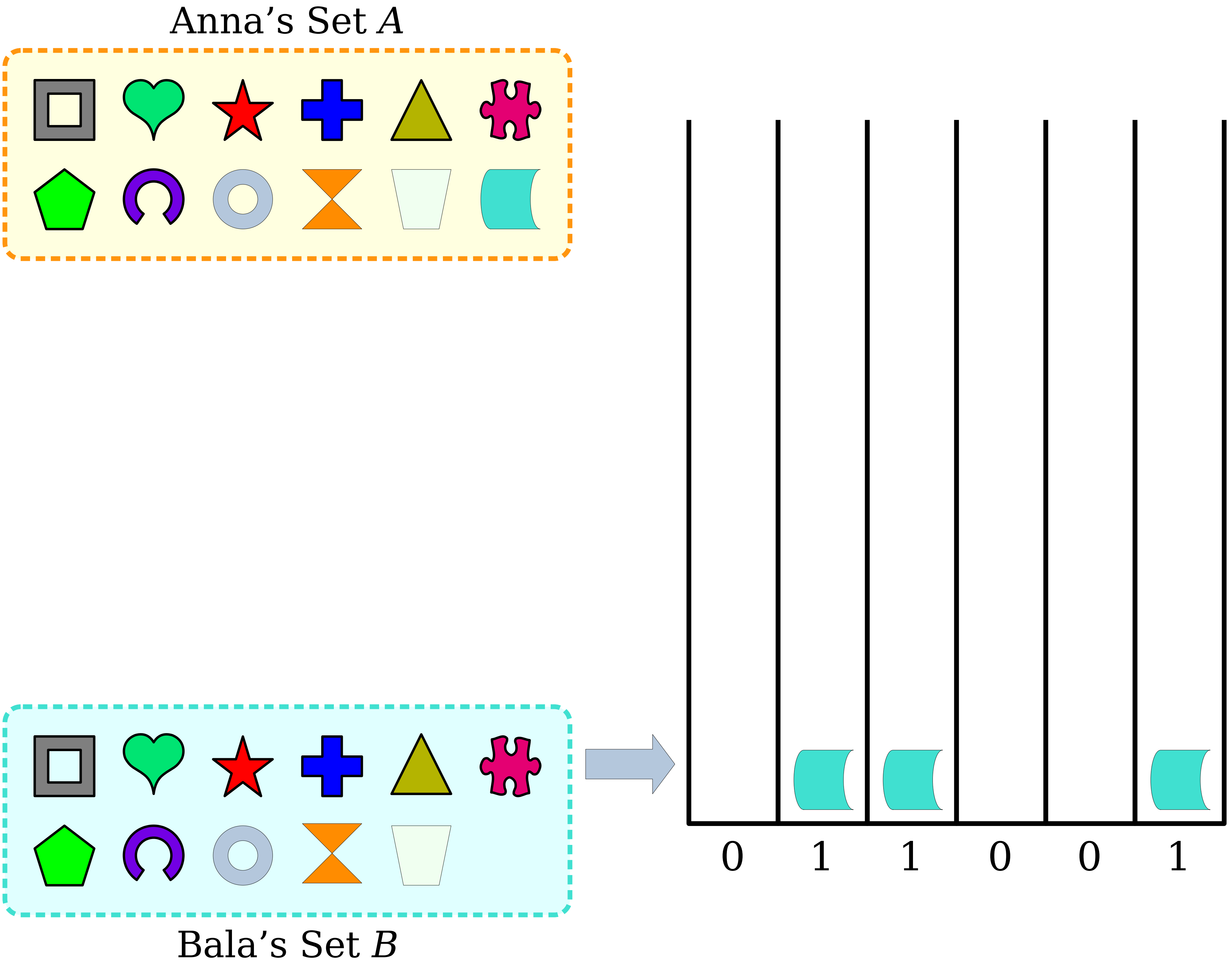

Anna's Set A
0
2
2
1
2
2
Bala's Set B

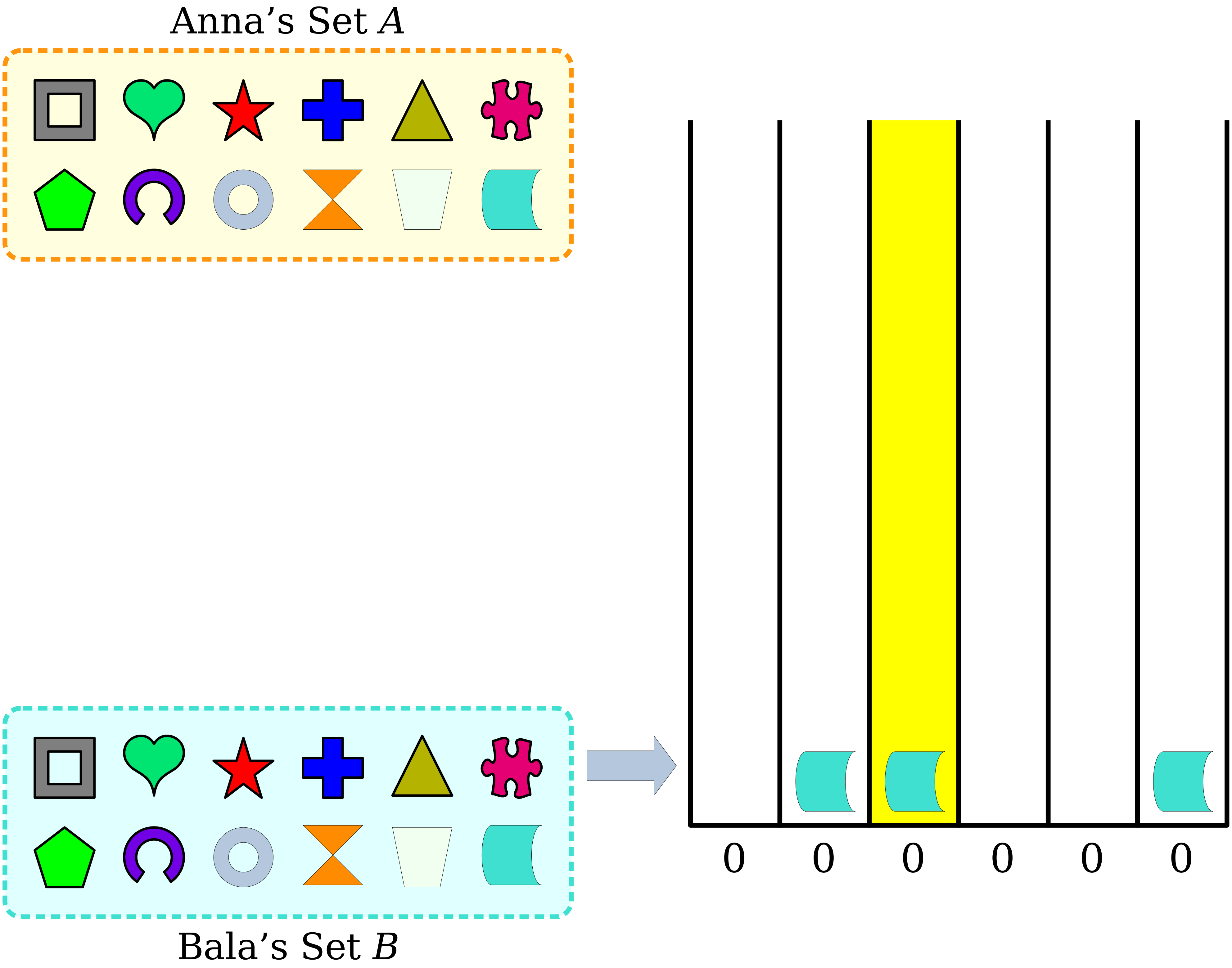

Anna’s Set *A*
Bala’s Set *B*
0
2
2
1
2
2

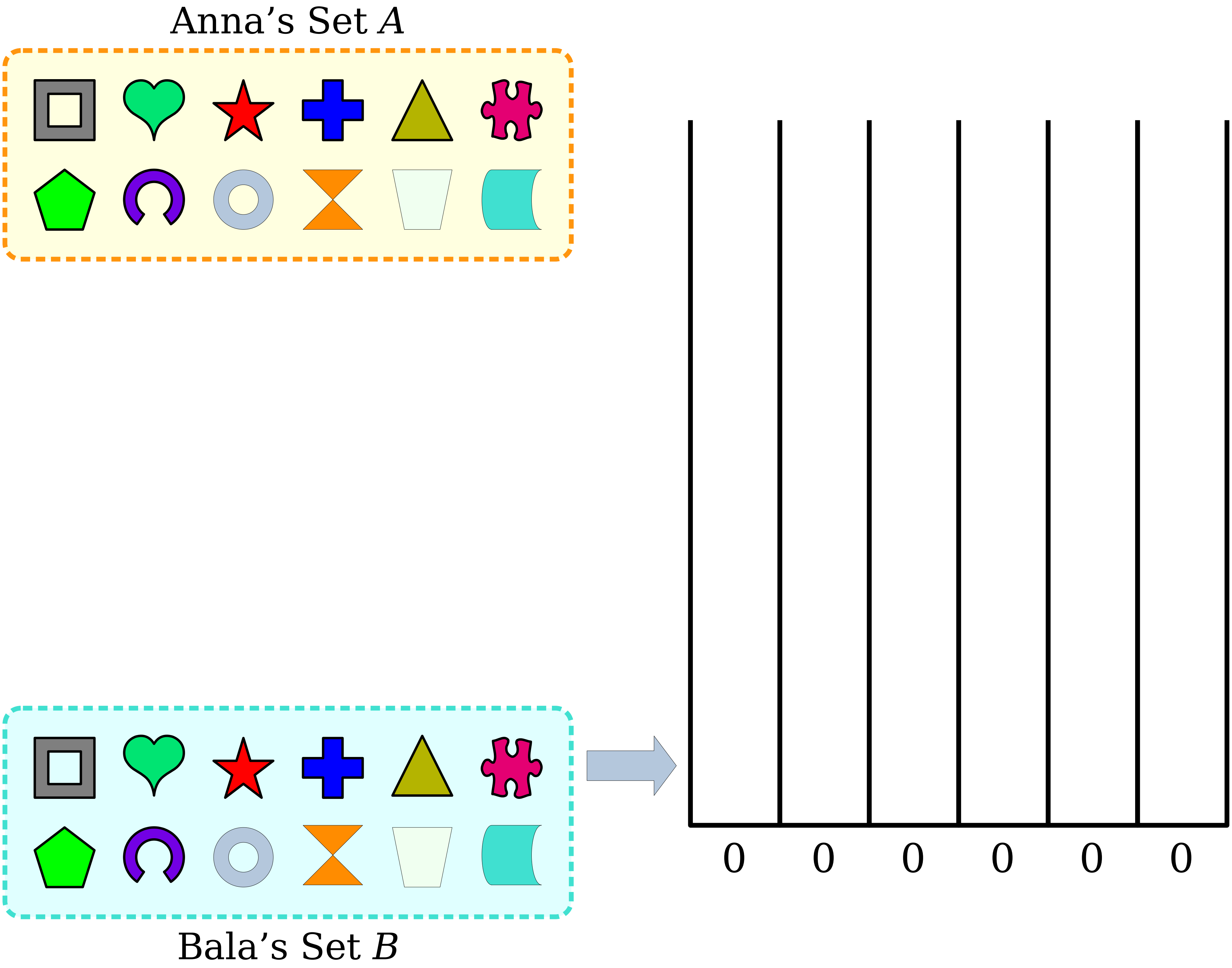

Anna's Set A
Bala's Set B
0
1
2
0
1
2


Anna’s Set A
0
1
2
0
1
2
Bala’s Set B


Anna’s Set A
Bala’s Set B
0
1
1
0
0
1

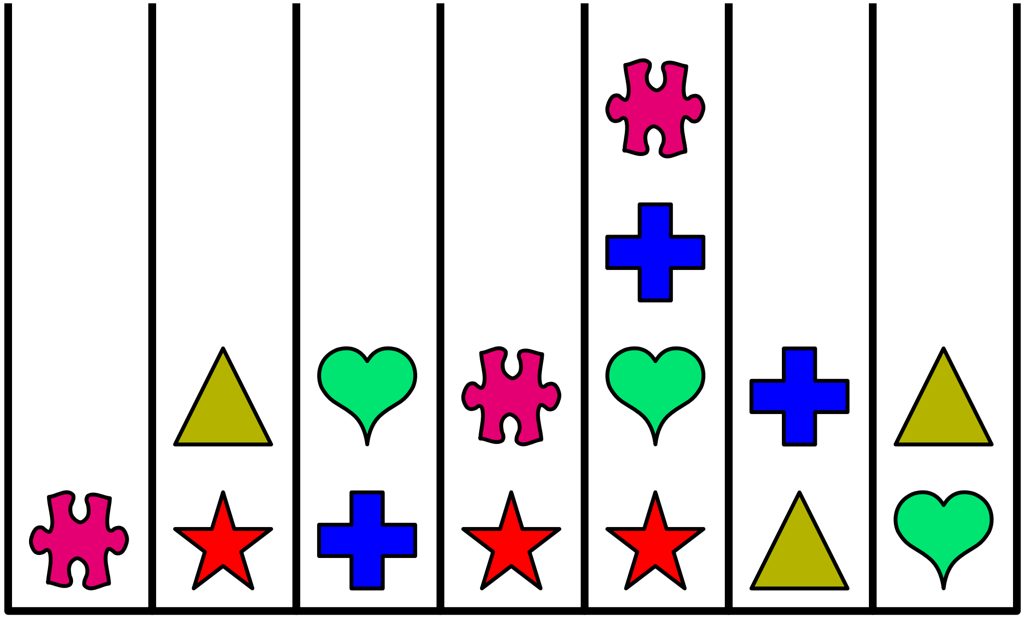

Anna's Set A
Bala's Set B
0
1
1
0
0
1


Anna’s Set A
0
0
0
0
0
0
Bala’s Set B


Anna's Set A
0
0
0
0
0
0
Bala's Set B

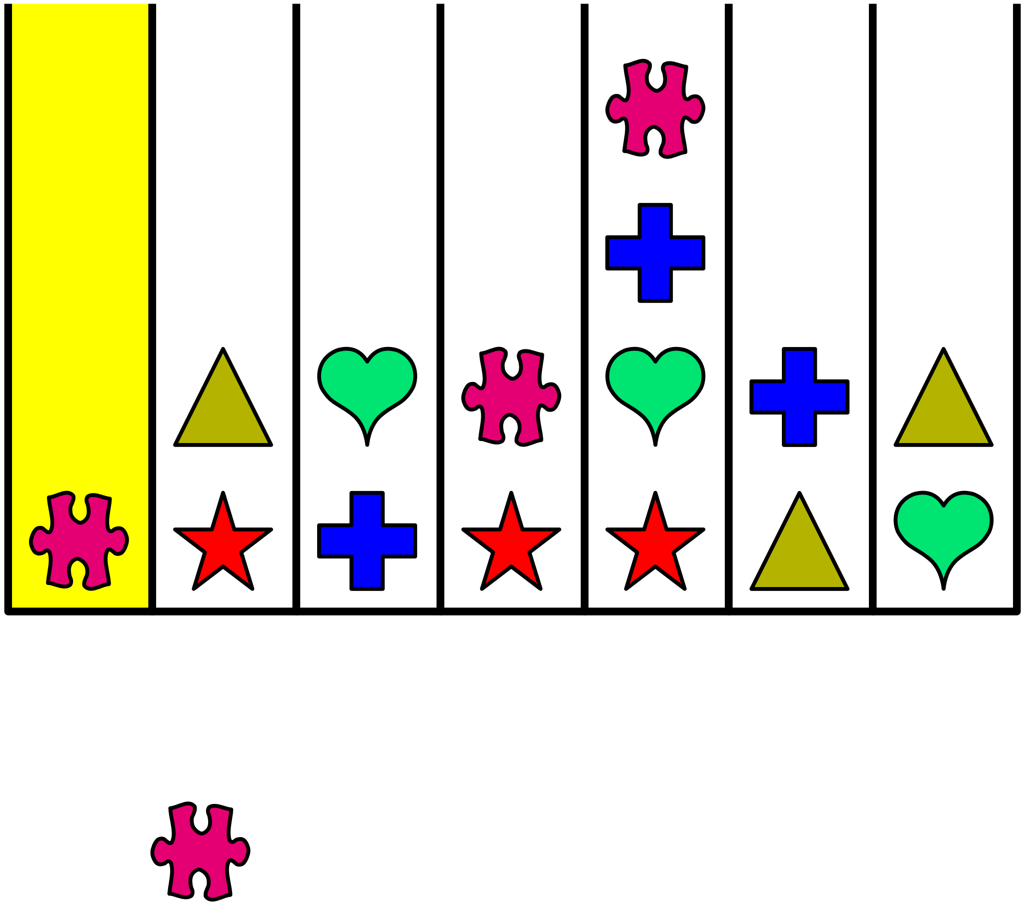
The Peeling Algorithm

- 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.)



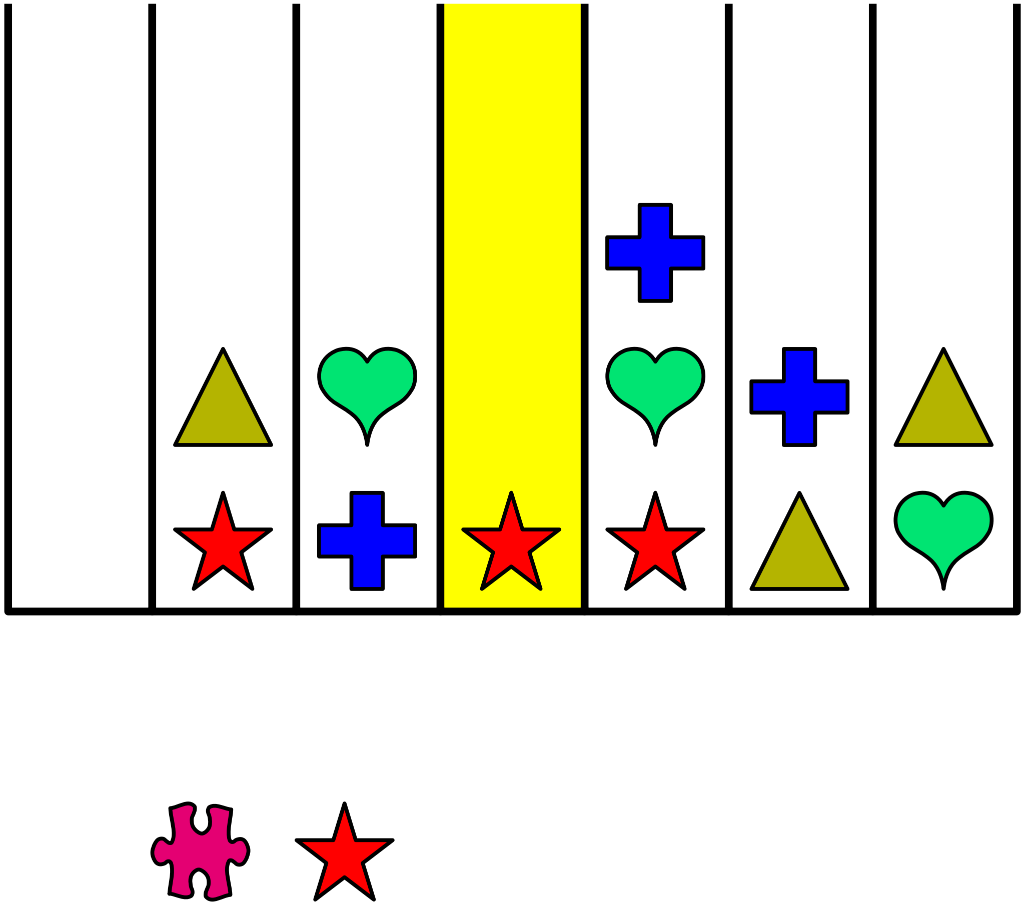
The Peeling Algorithm

- 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.)



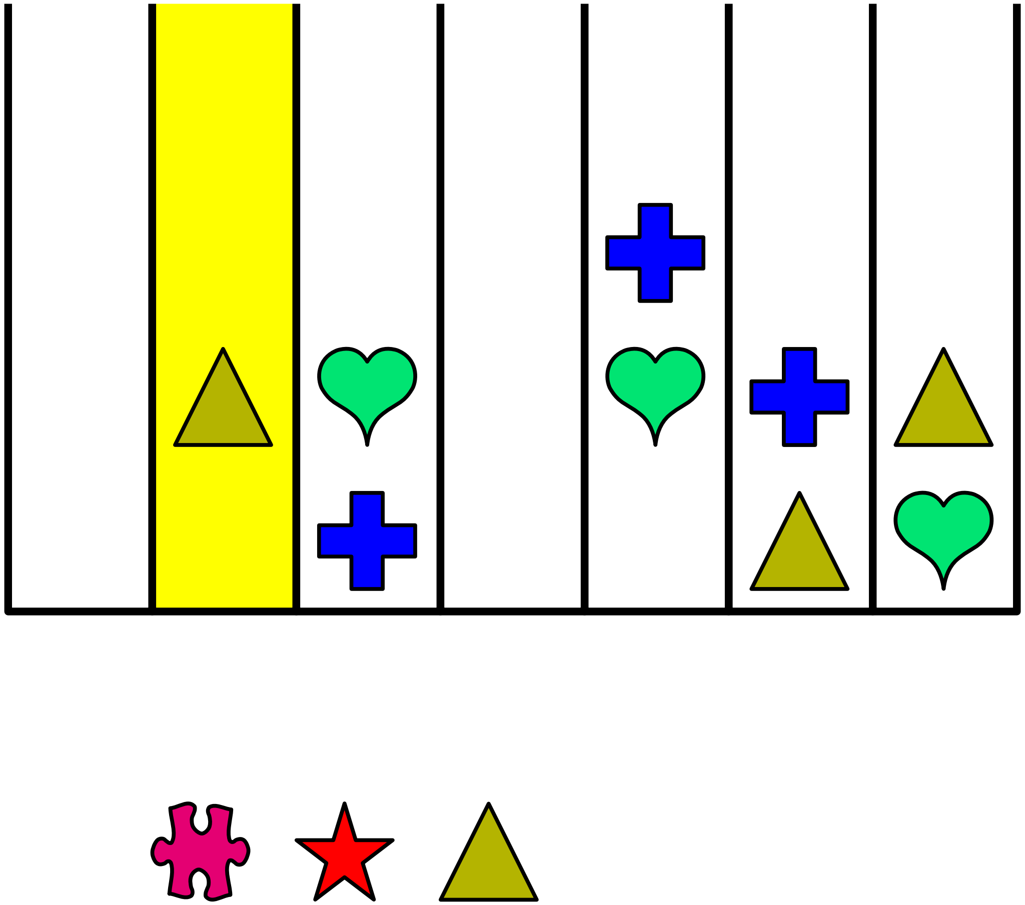
The Peeling Algorithm

- 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.)



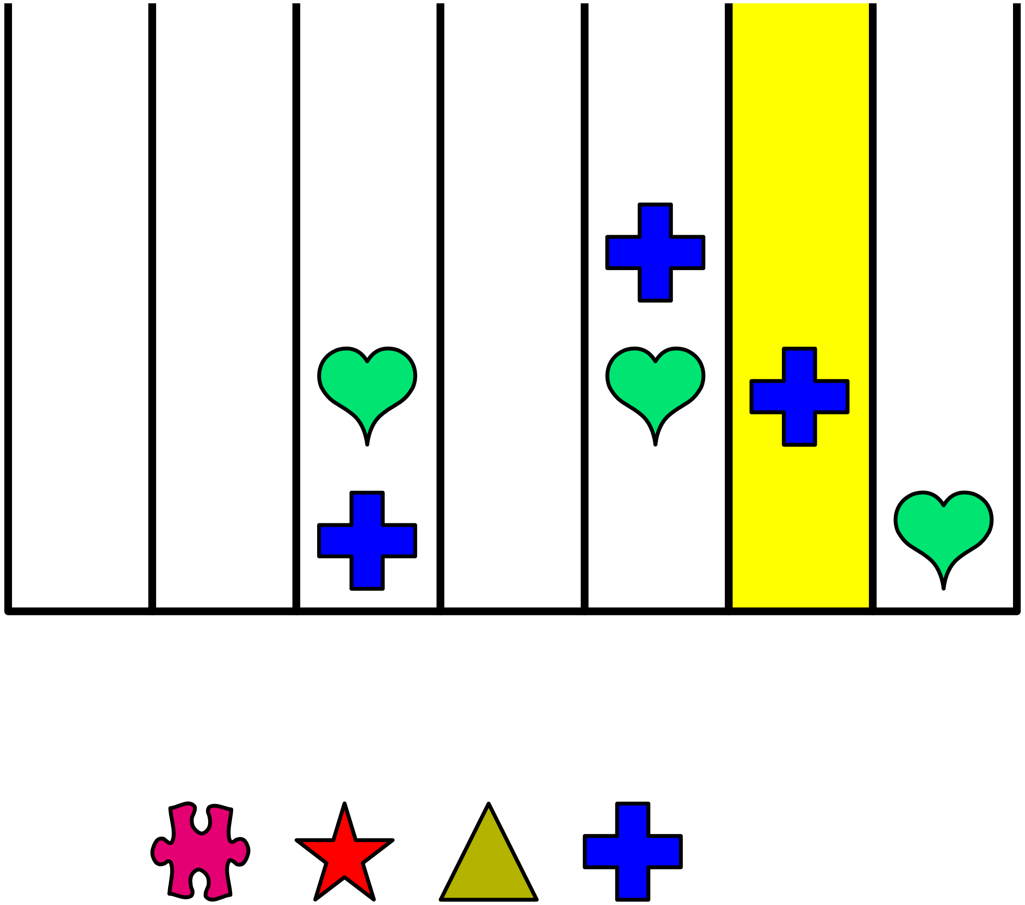
The Peeling Algorithm

- 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.)



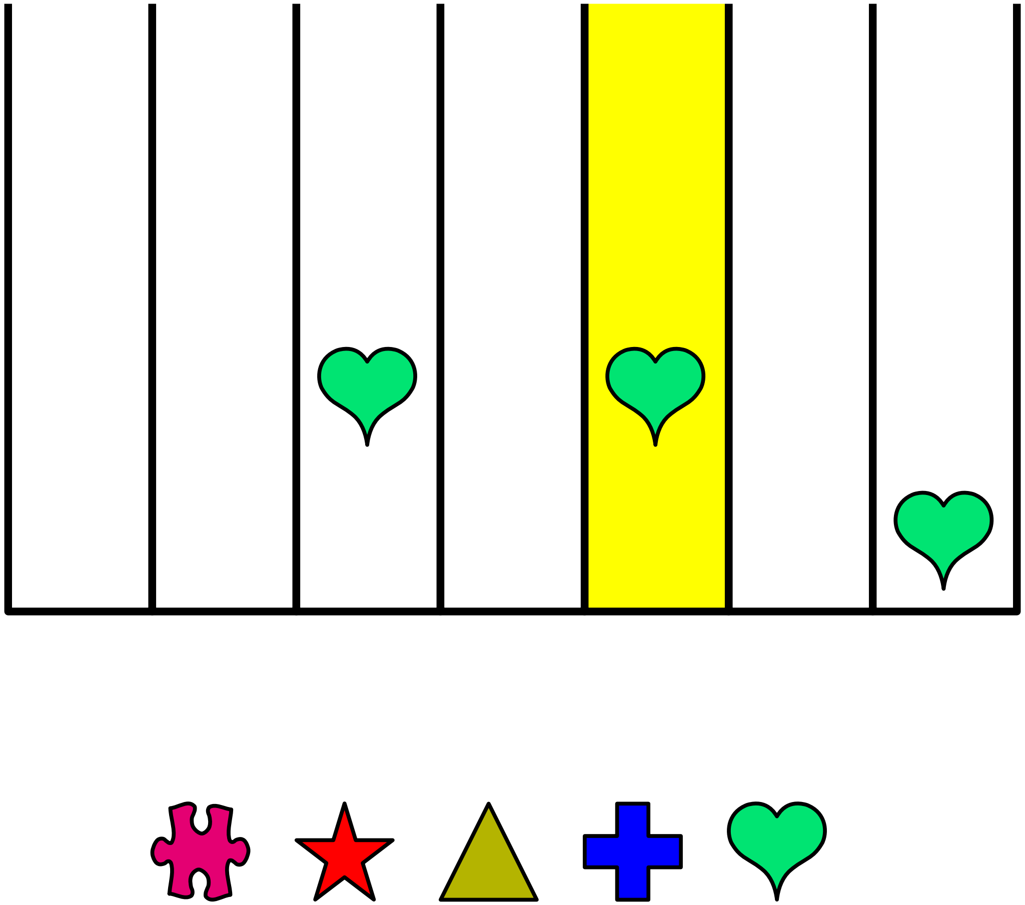
The Peeling Algorithm

- 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.)



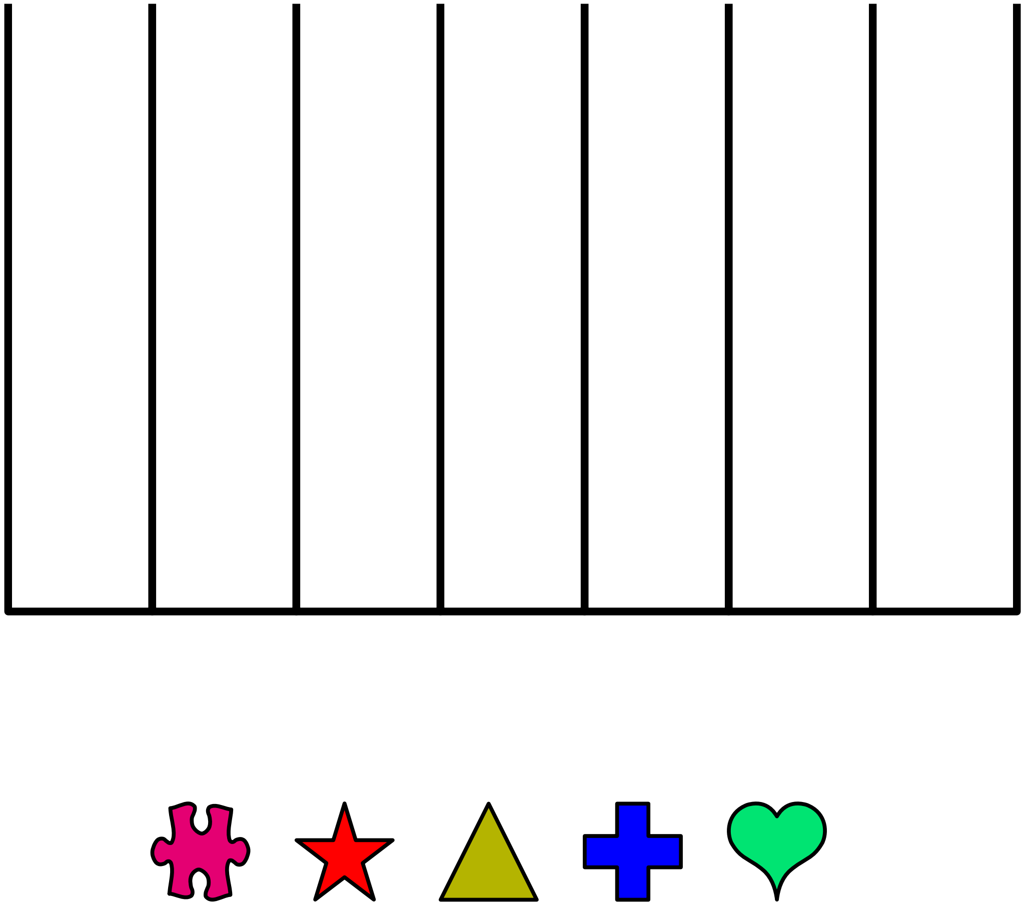
The Peeling Algorithm

- 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.)

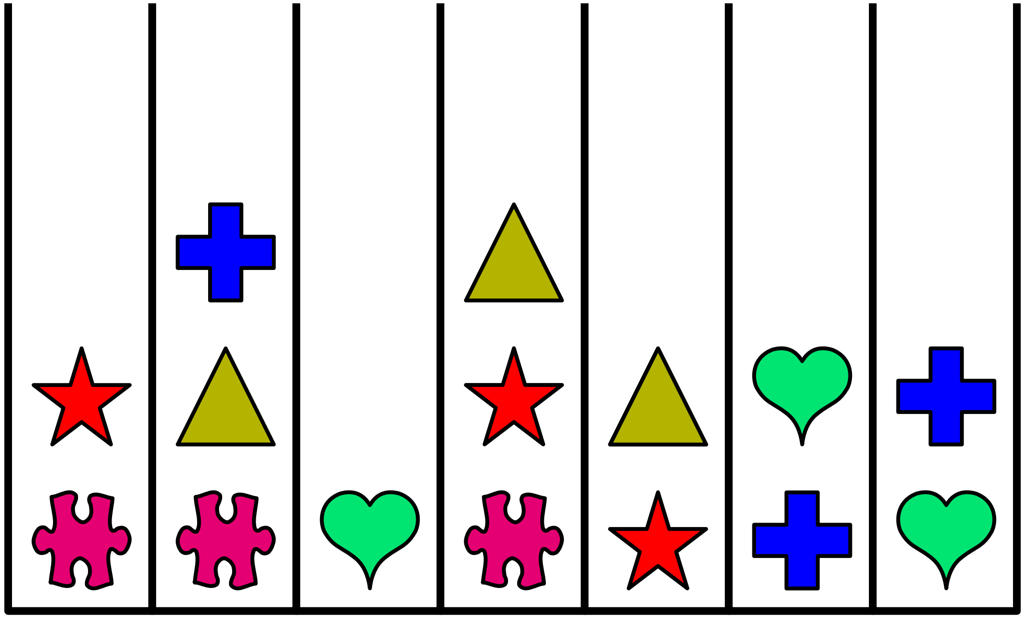


The Peeling Algorithm

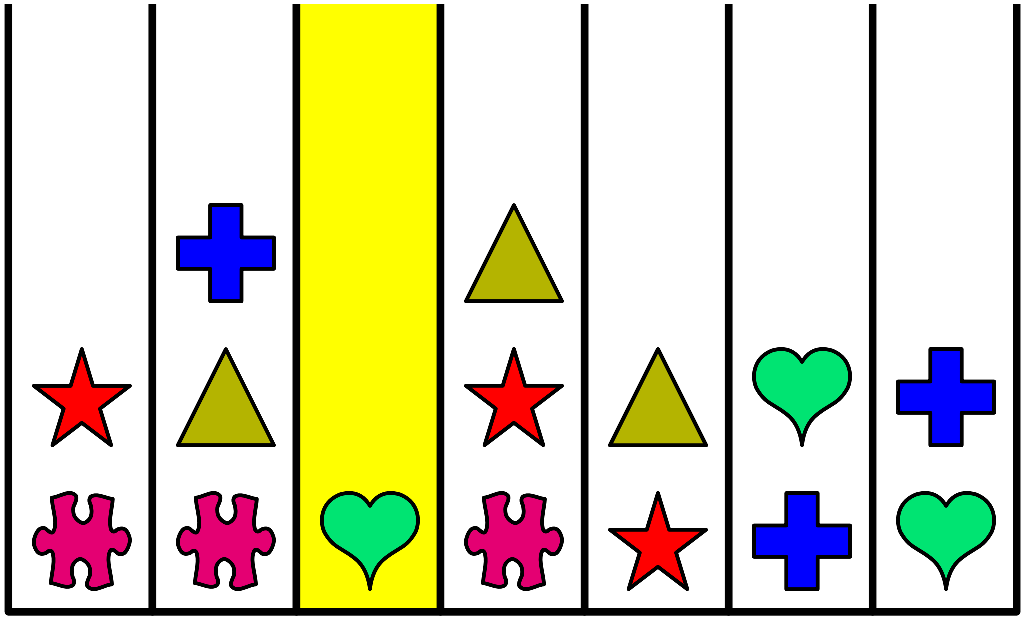
- If we have multiple copies of each item, we might still be able to recover them all even when collisions exist.
- ***Algorithm:*** Repeatedly find a bucket with one item in it, identify the item, and remove all copies of it.
- (To perform the last step: hash the item with all the hash functions, XOR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



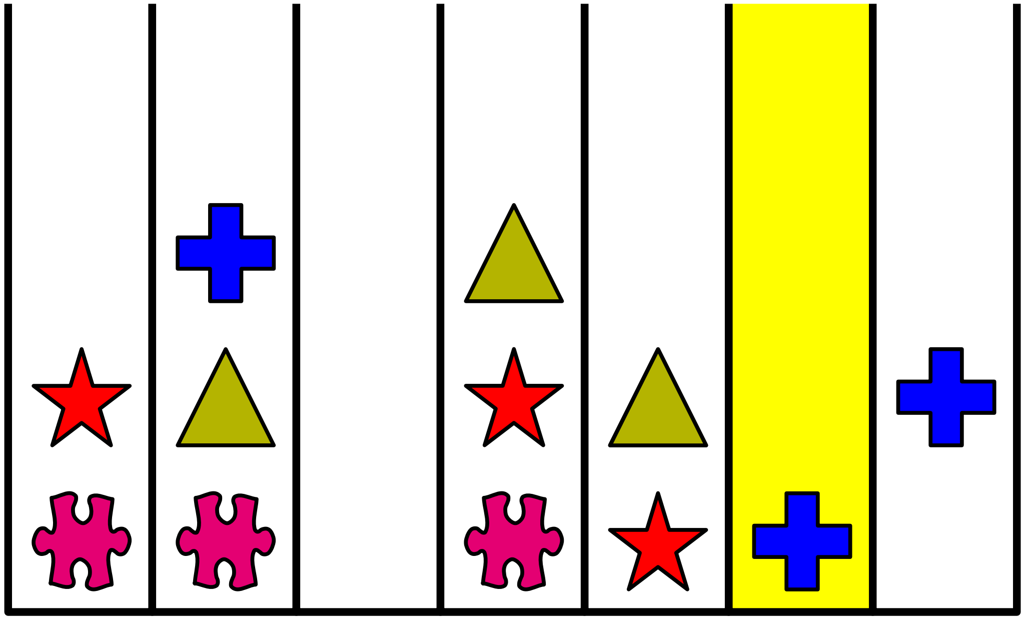
The Peeling Algorithm



The Peeling Algorithm

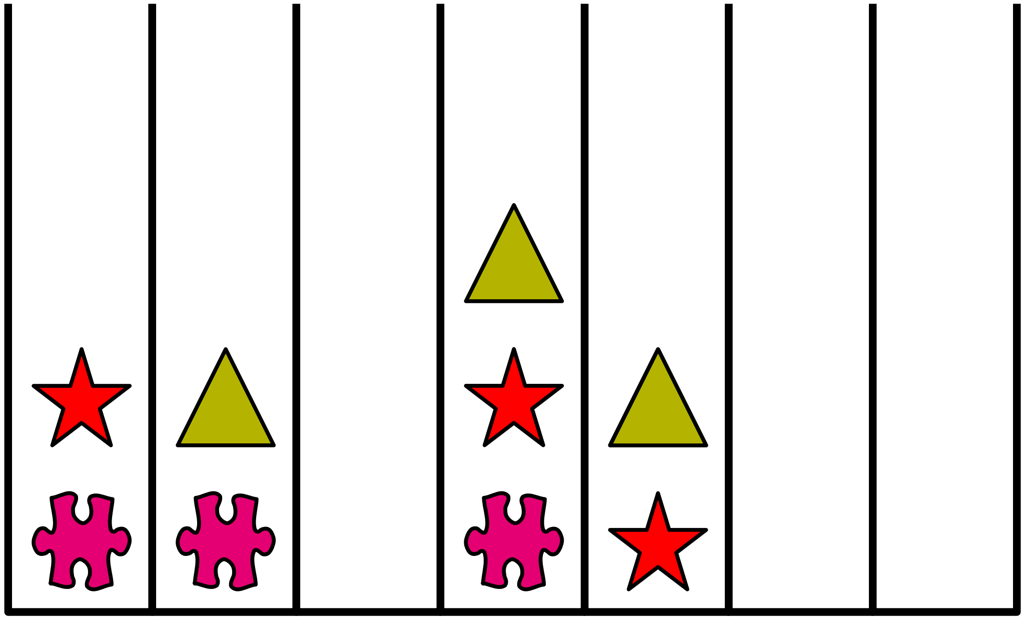


The Peeling Algorithm



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

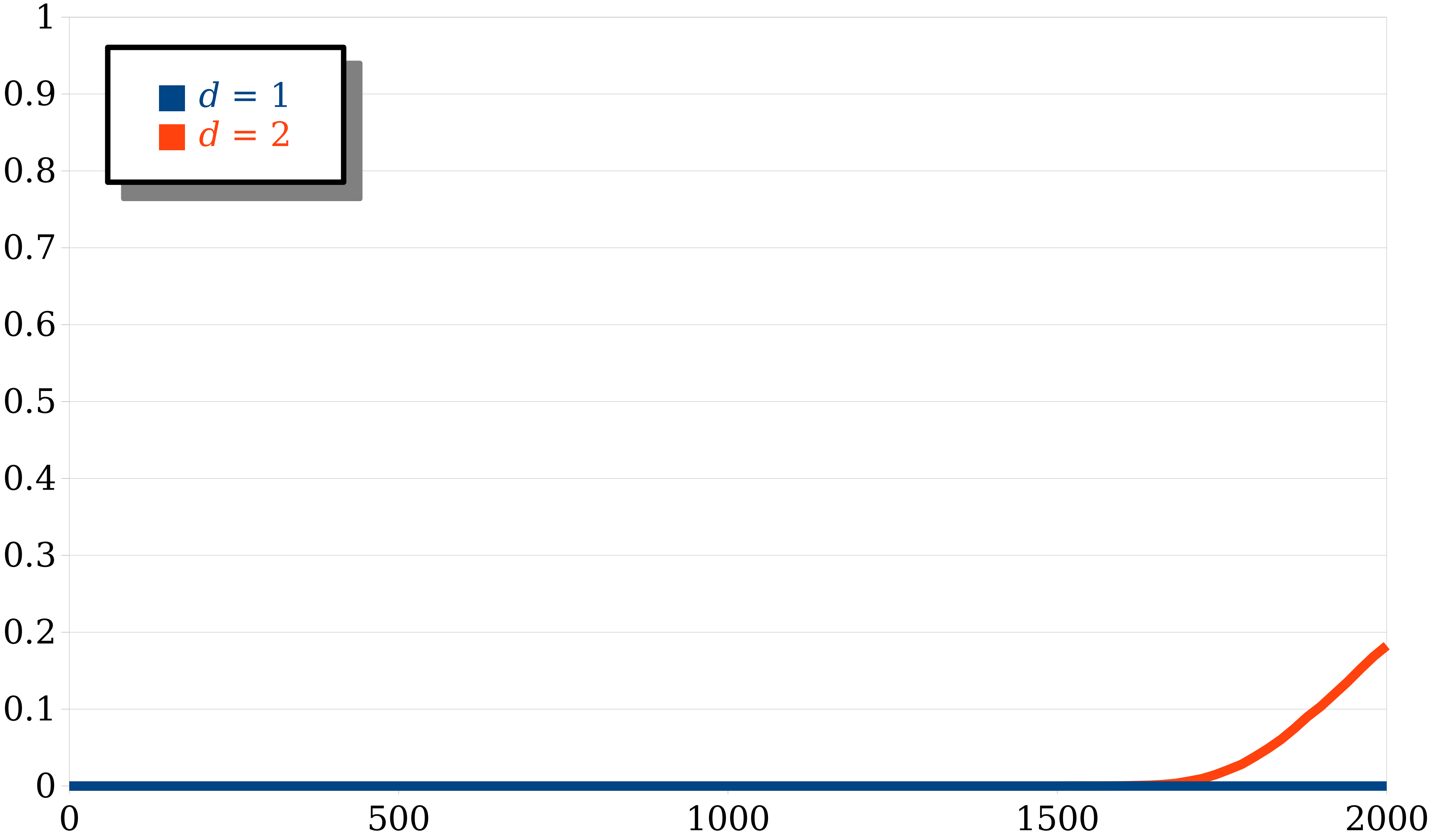
Tuning *m*

- 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.

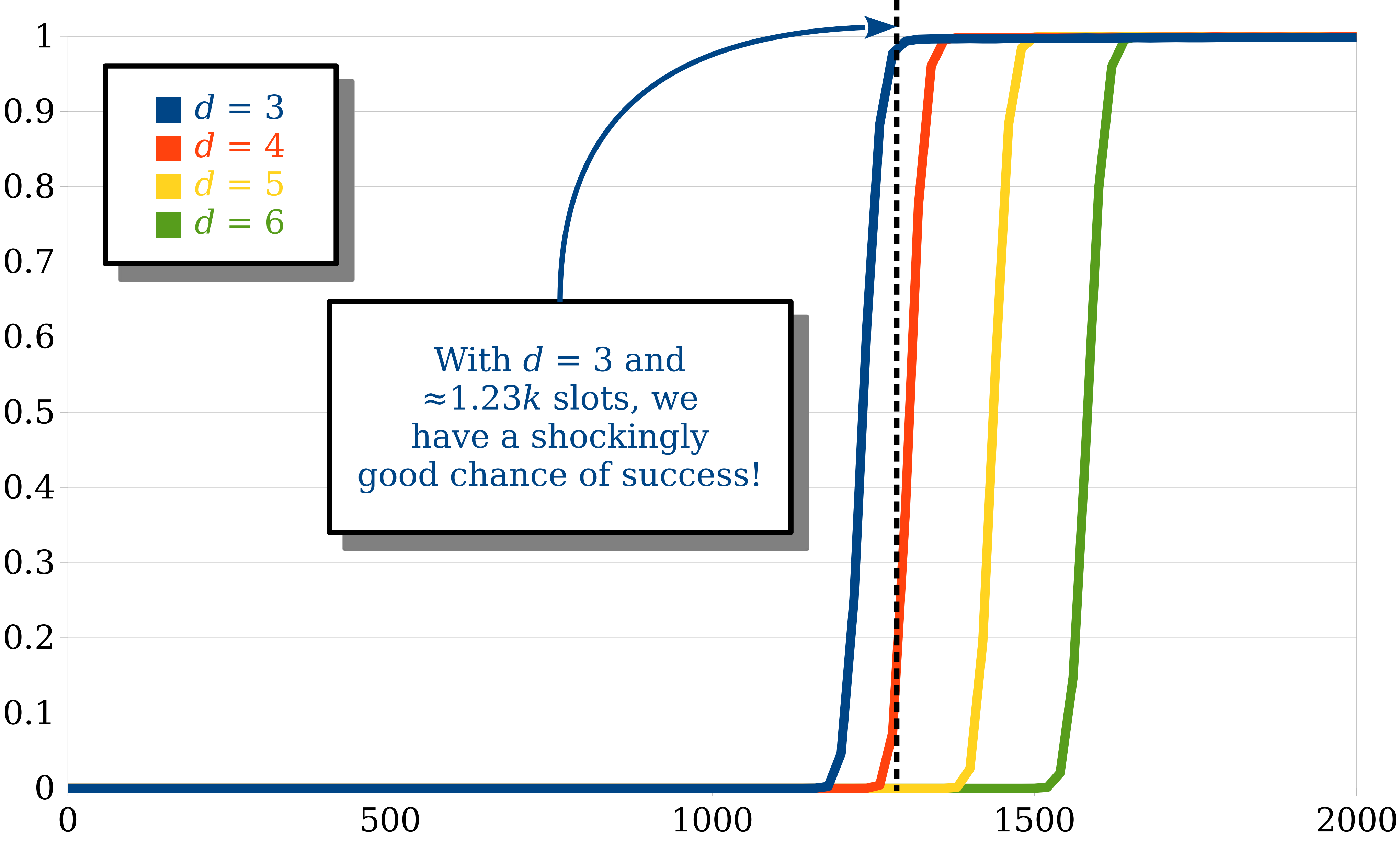
Tuning *d*

- 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?

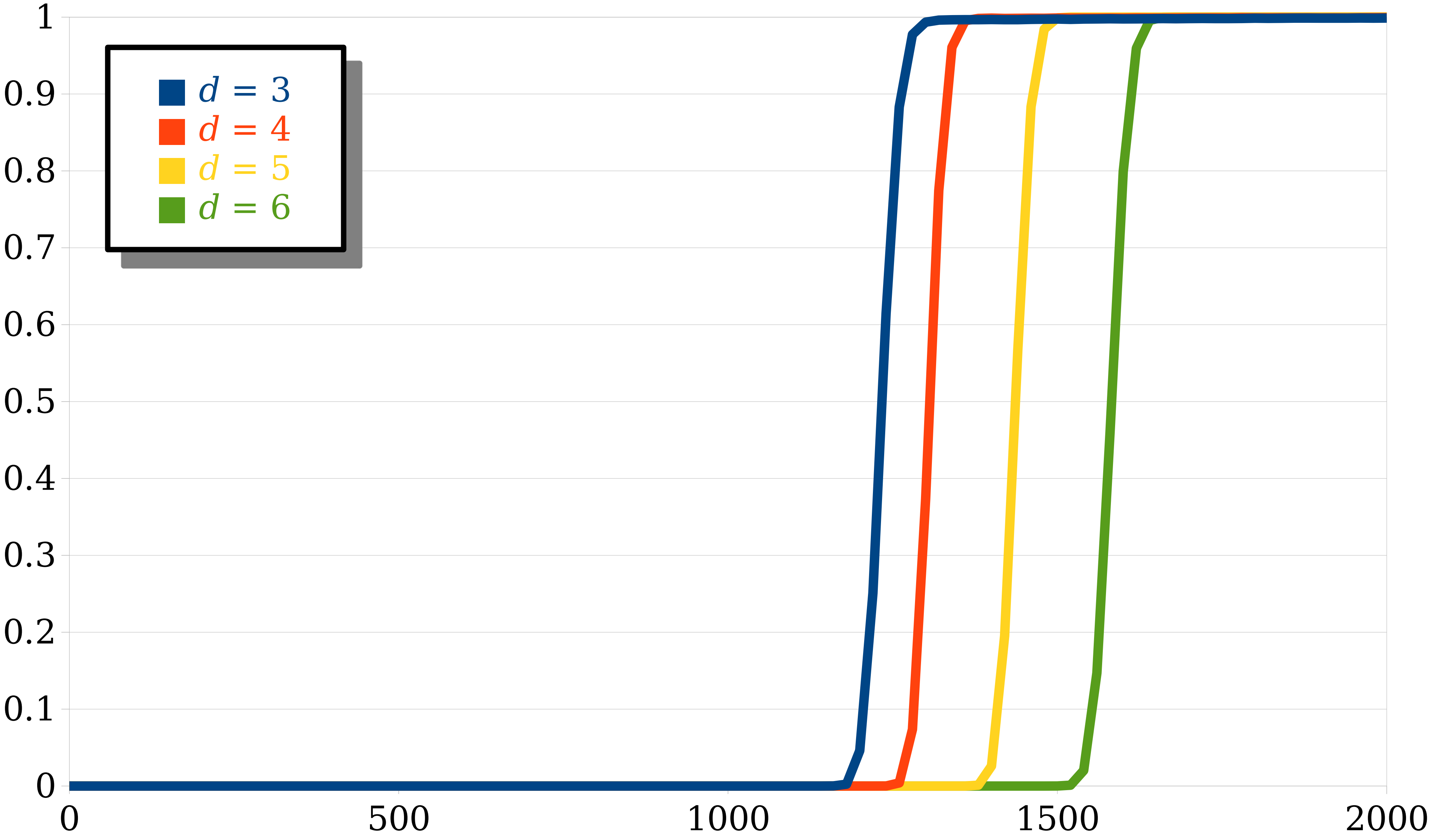


With $d = 3$ and $\approx 1.23k$ slots, we have a shockingly good chance of success!

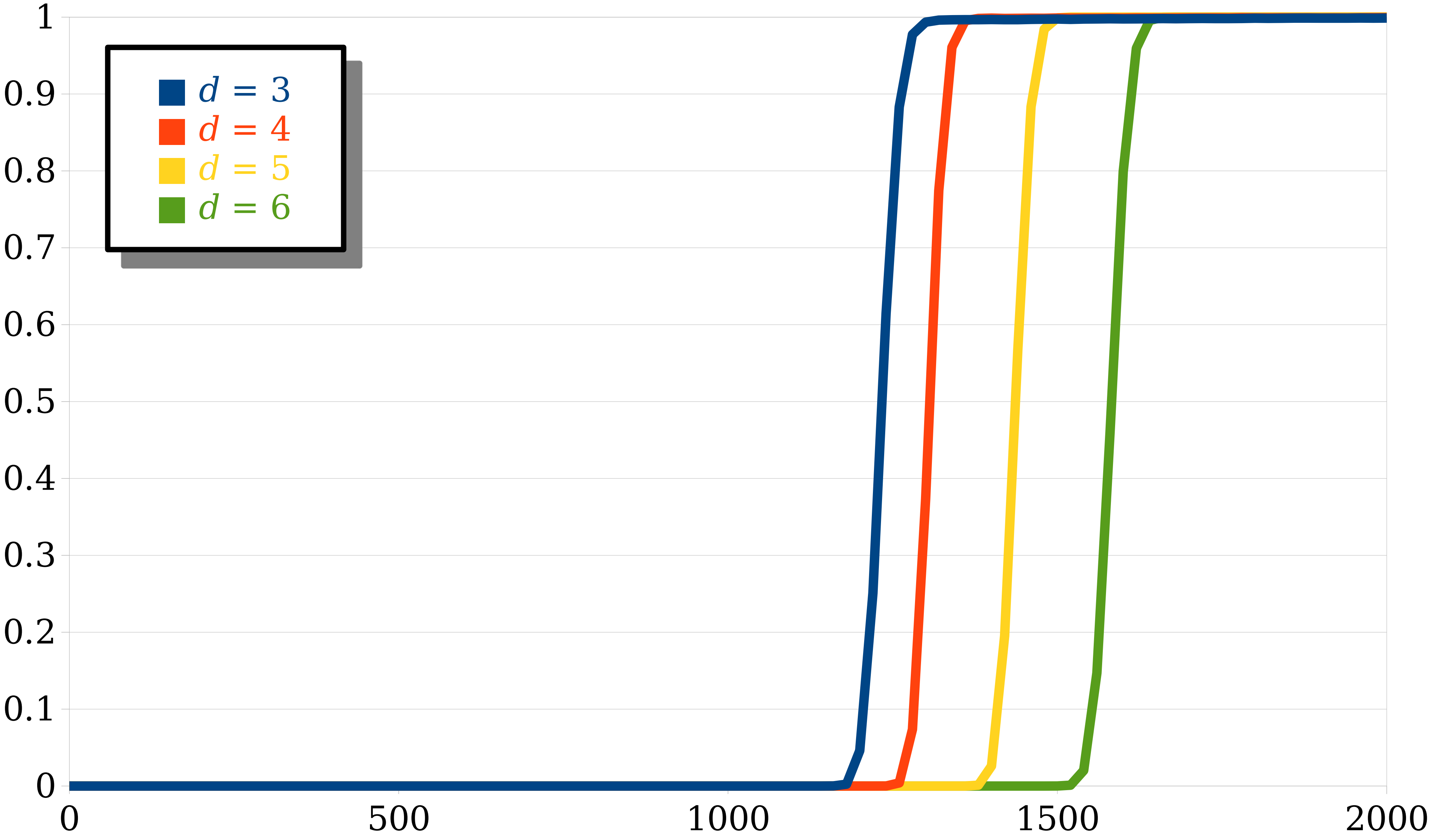
What is the probability that all elements are peeled when we have $k = 1000$ elements and m buckets, as a function of d?

IBLTs

- Create an array of ≈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.
- Anna ***add***s her items, Bala ***remove***s his, and then either can ***list*** the items to find what's missing.
- This is called an ***IBLT*** (***I***nvertible ***B***loom ***L***ookup ***T***able).
- It requires **≈1.23*k*** 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, ≈1.31k buckets, and have success probability 1 – O(k^{-2}); or five hash functions, ≈1.44k buckets, and have success probability 1 – O(k^{-3}), etc.



What’s going on here?



Spoiler: It’s awesome. And it has nothing to do with exponential growth and decay.

Goal: Explain why the phase transition exists for $d \geq 3$.

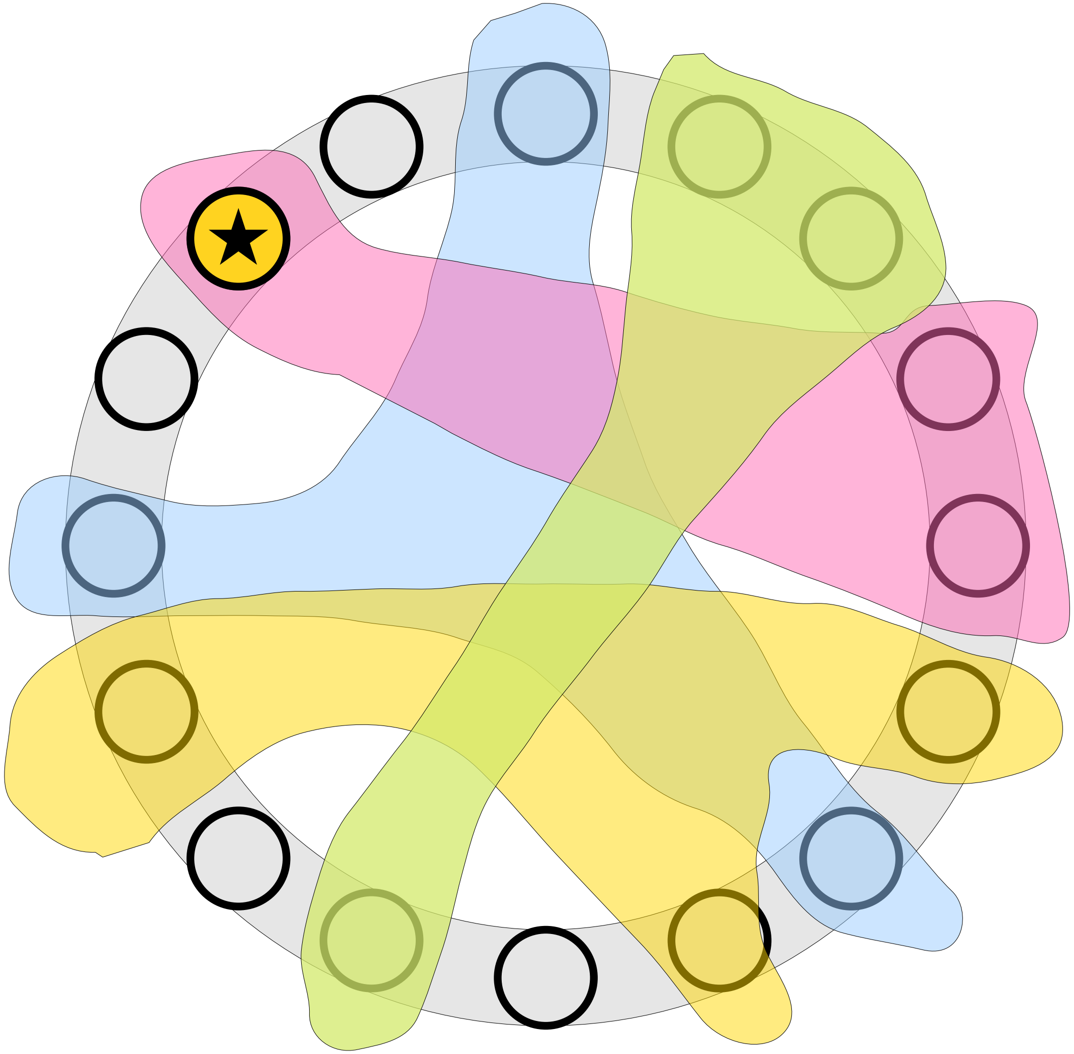
Where We’re Going

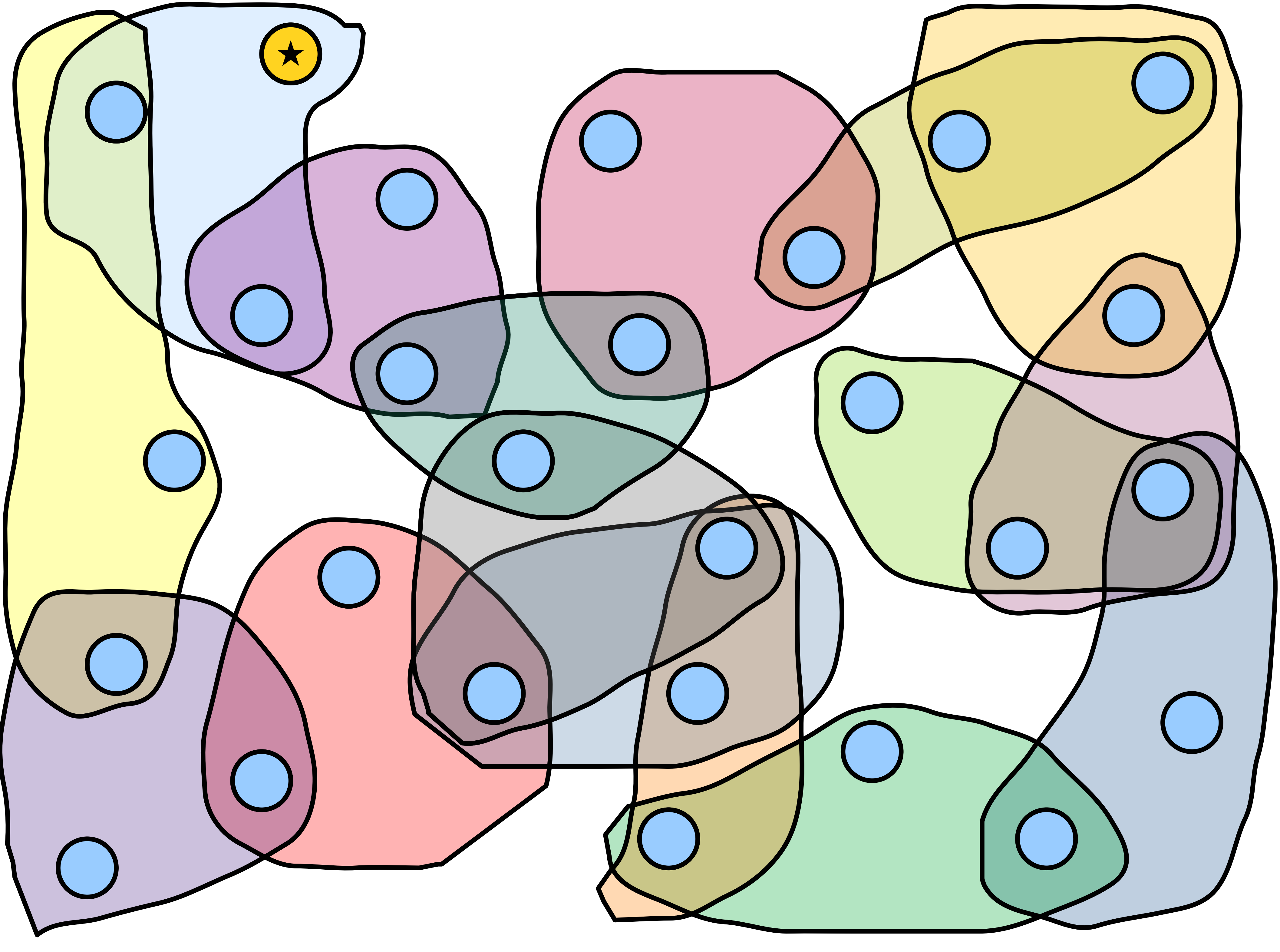
- ***Step One:*** Model the peeling process as an operation on hypergraphs.
- ***Step Two:*** Determine the probability that peeling succeeds on a random hypergraph.
- ***Step Three:*** Analyze our results from Step Two to see where the phase transition comes from.

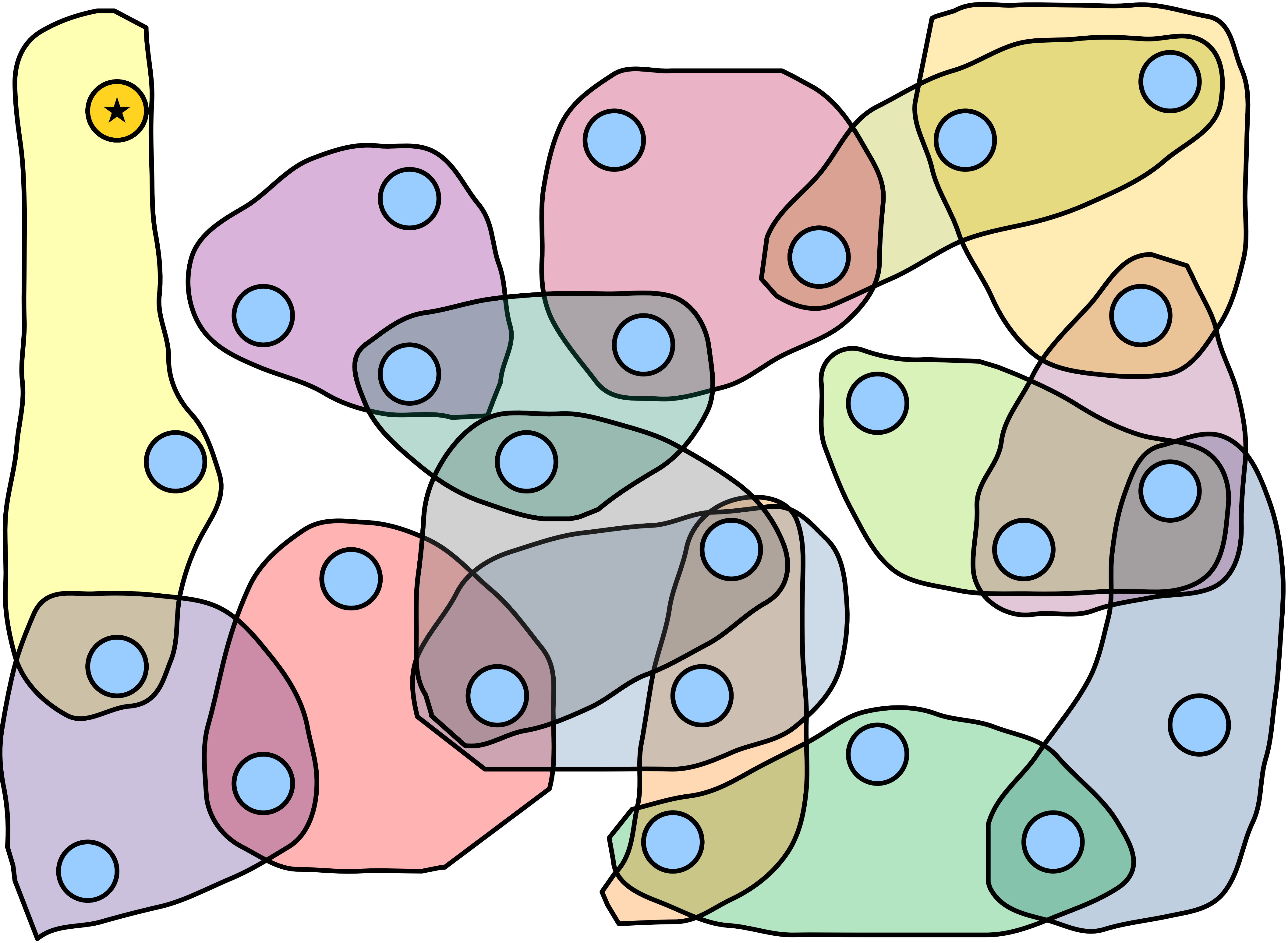
Step One: Model peeling as a process on random hypergraphs.

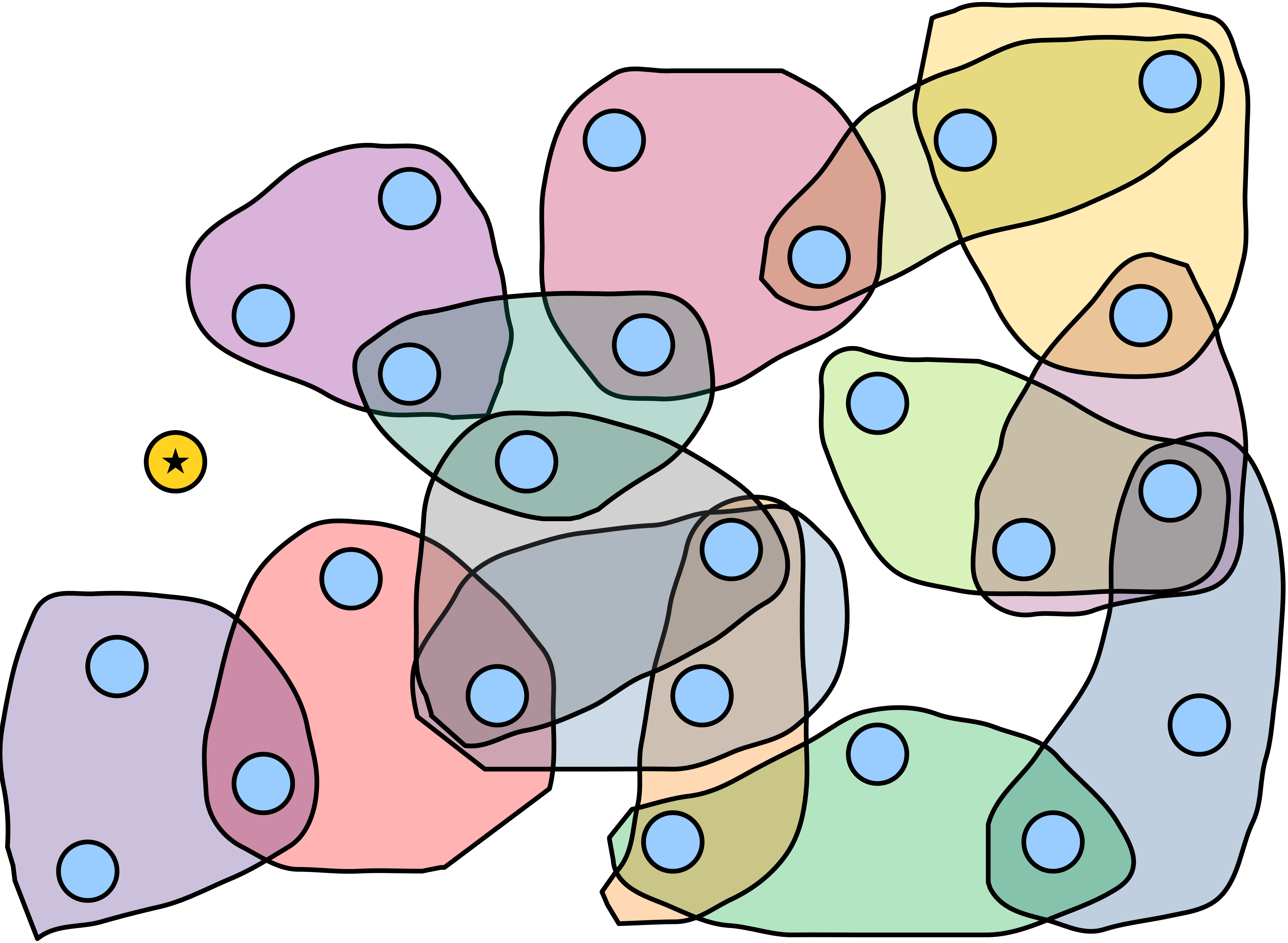
Modeling Peeling

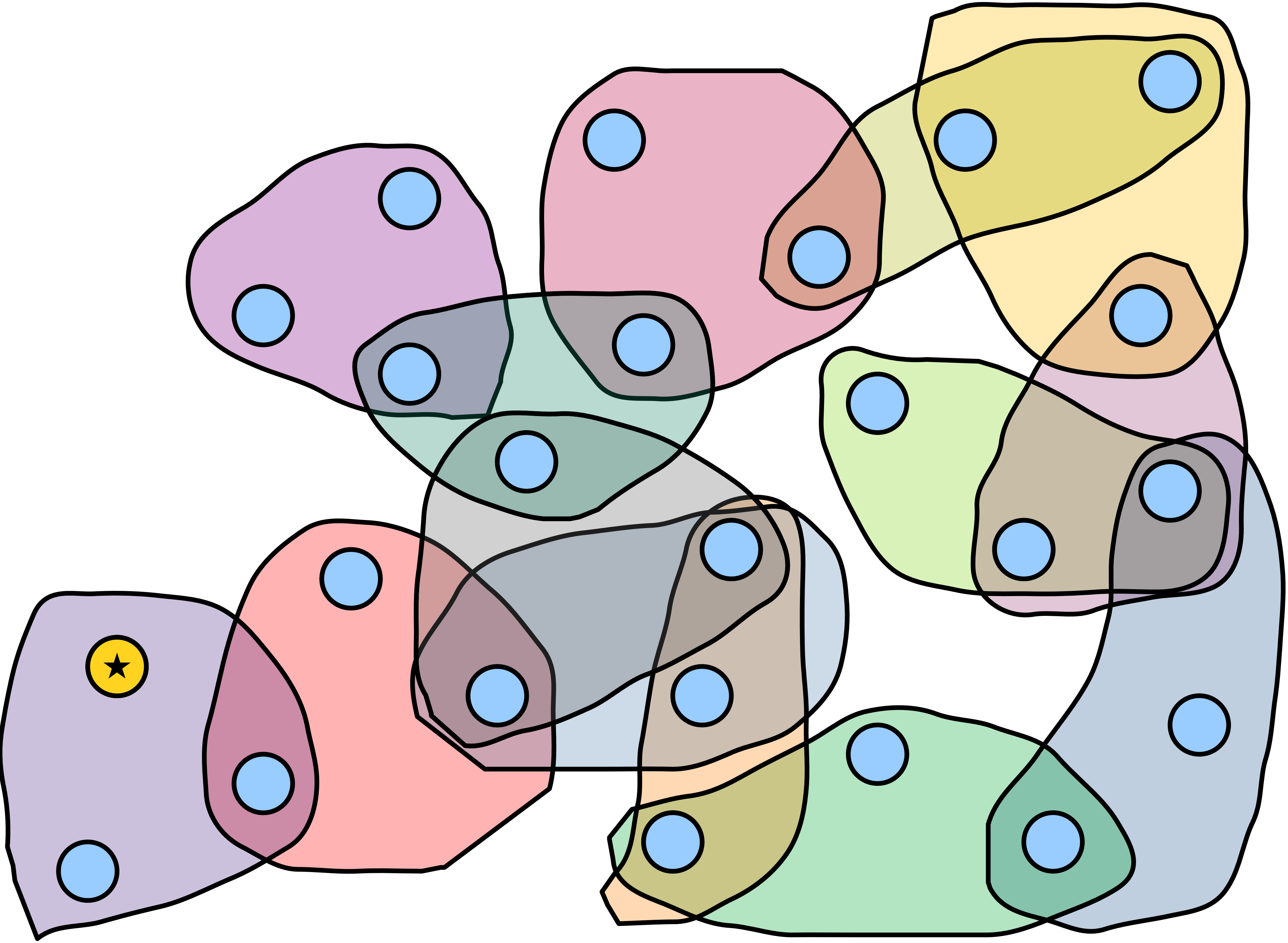
- Model the assignment of items to buckets as a hypergraph.
  - Each bucket is a node.
  - Each item is a hyperedge linking *d* buckets together.
- Model peeling as finding a node with degree one or less and deleting it and any edges touching it.

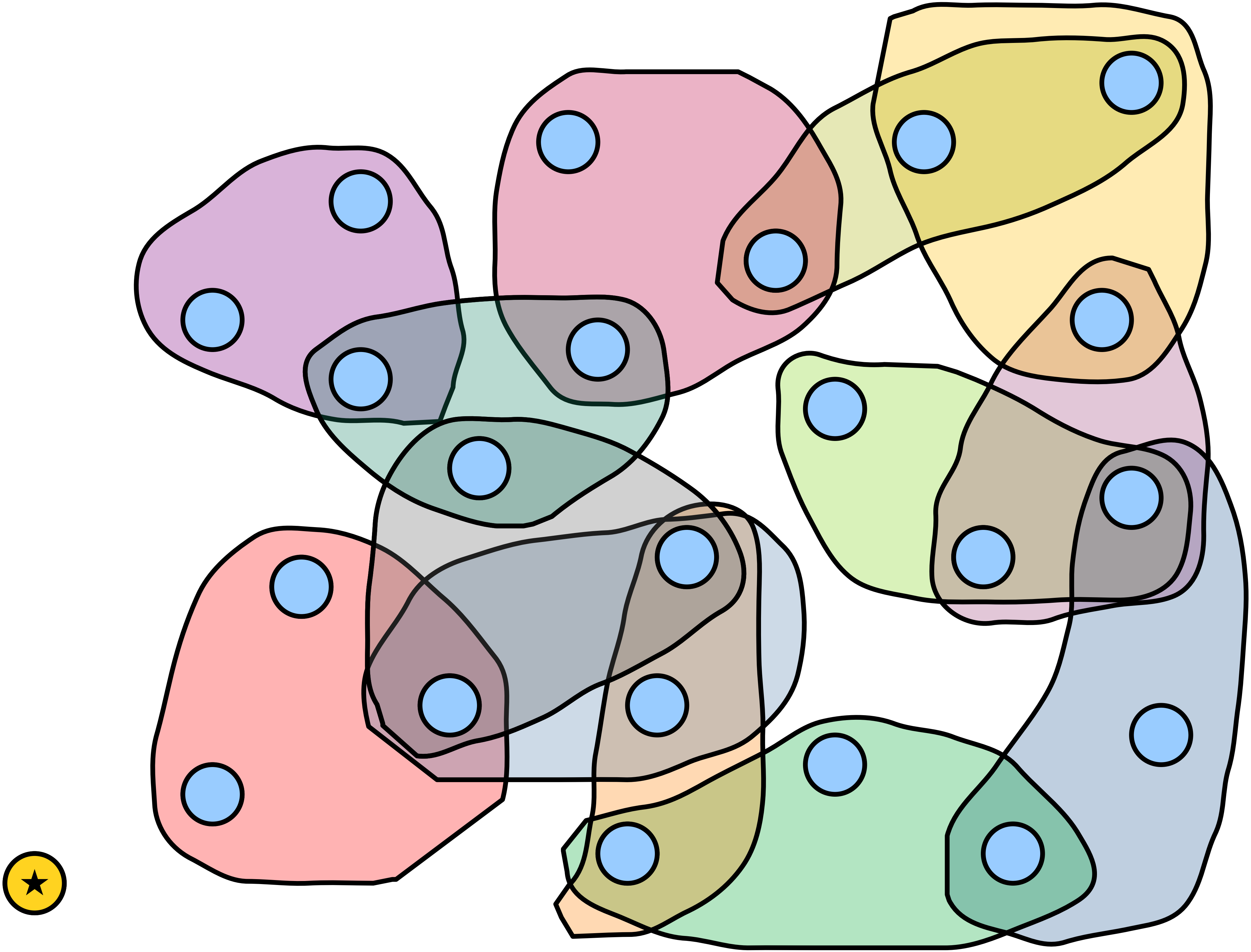


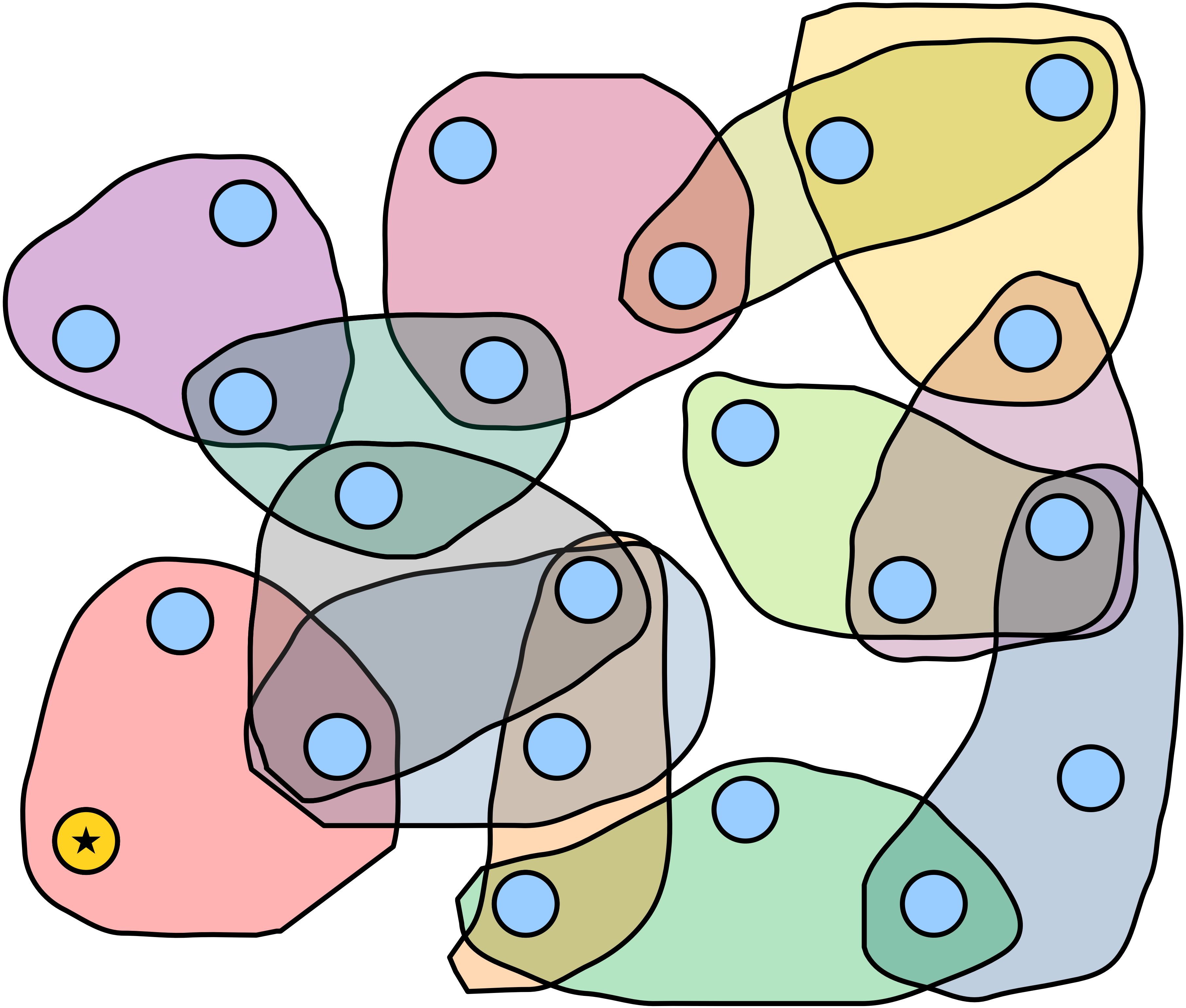


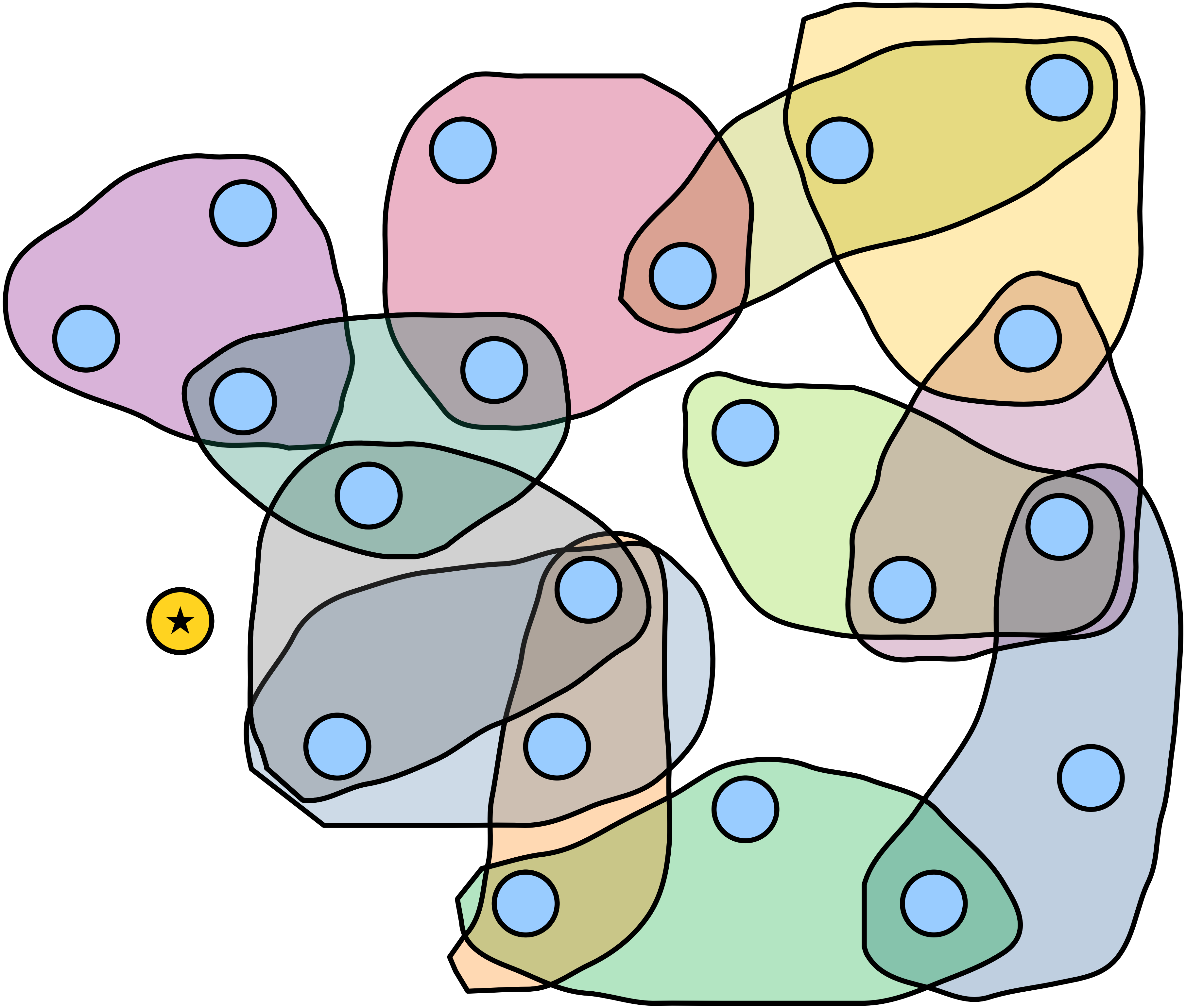


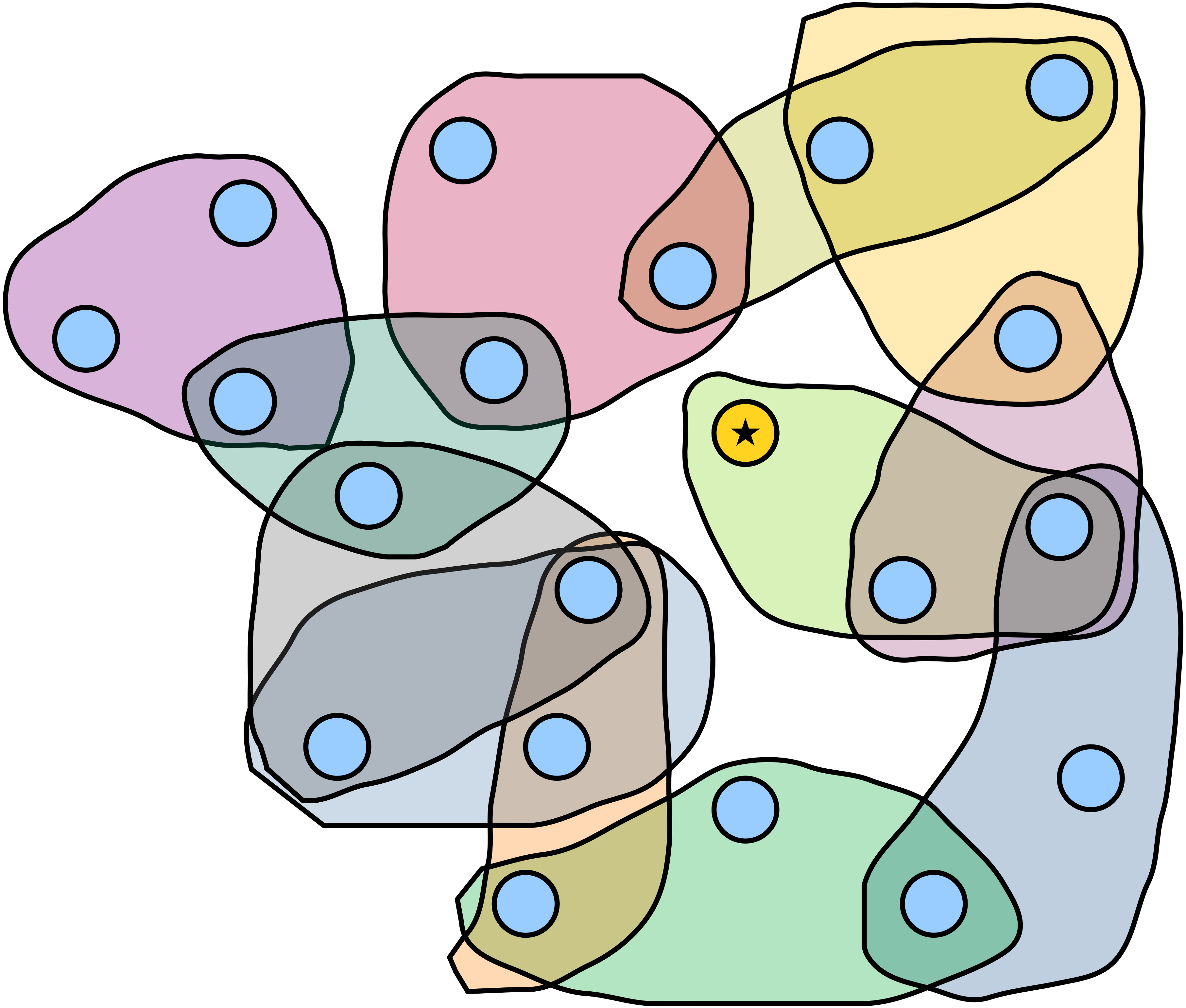


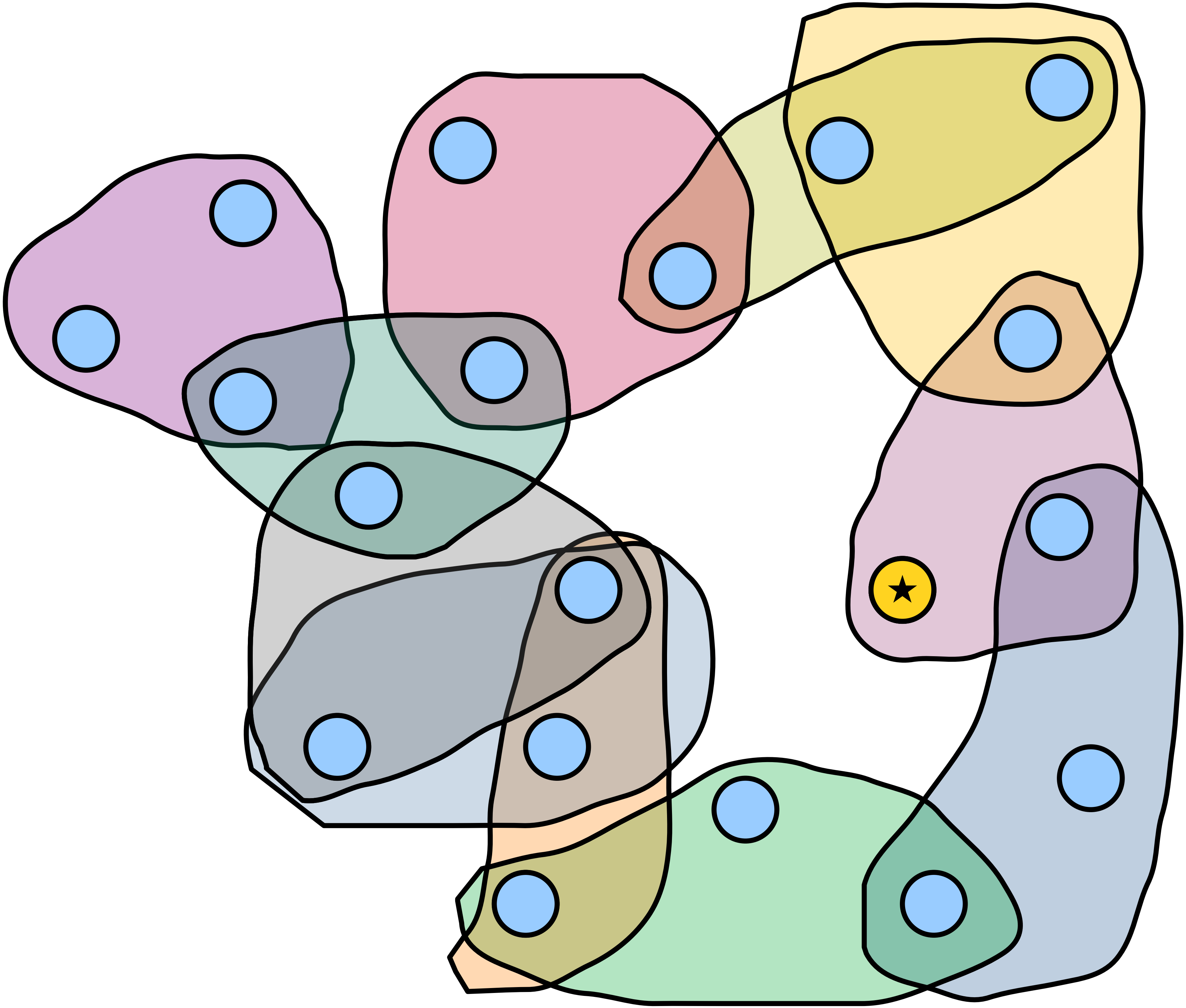


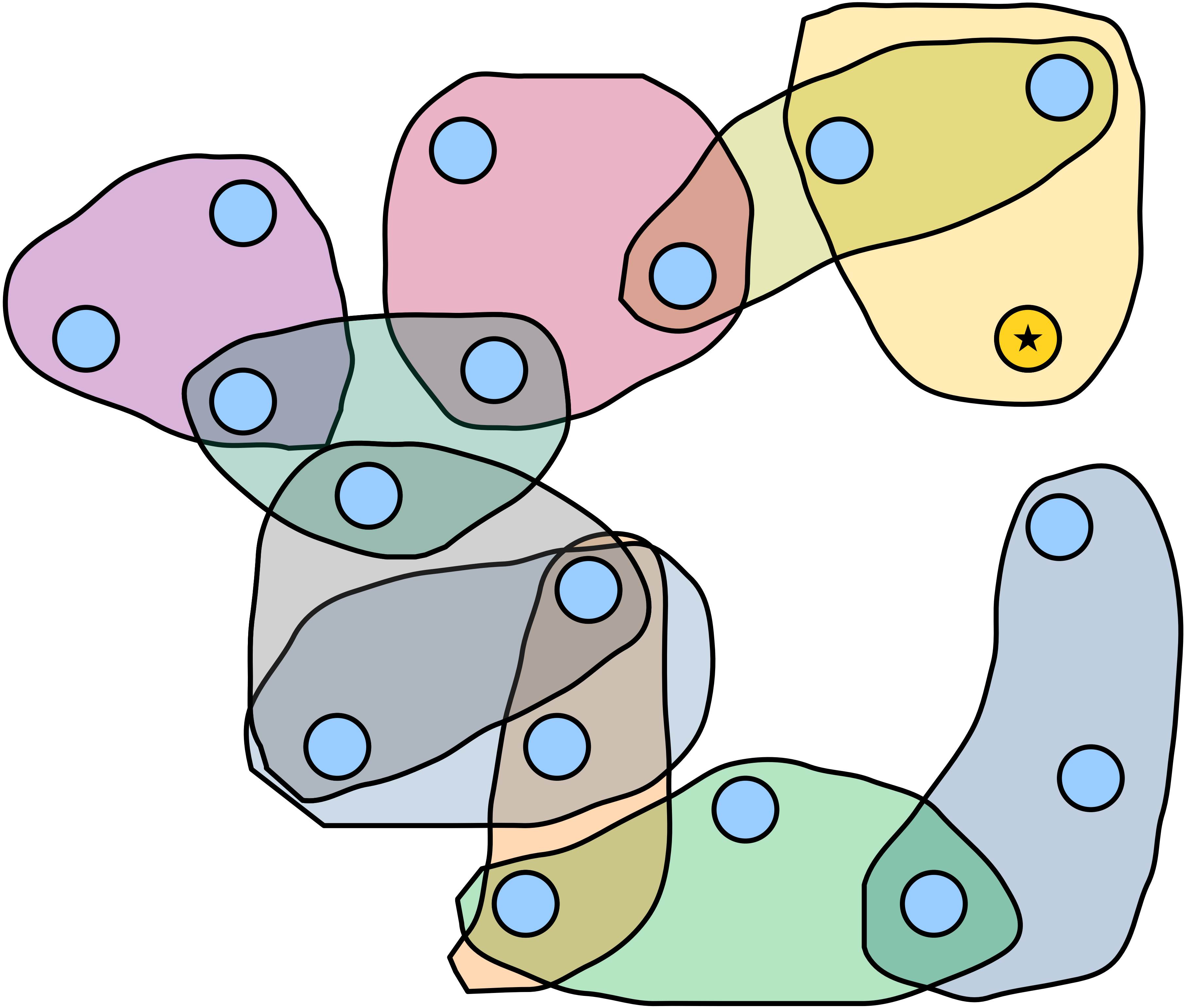


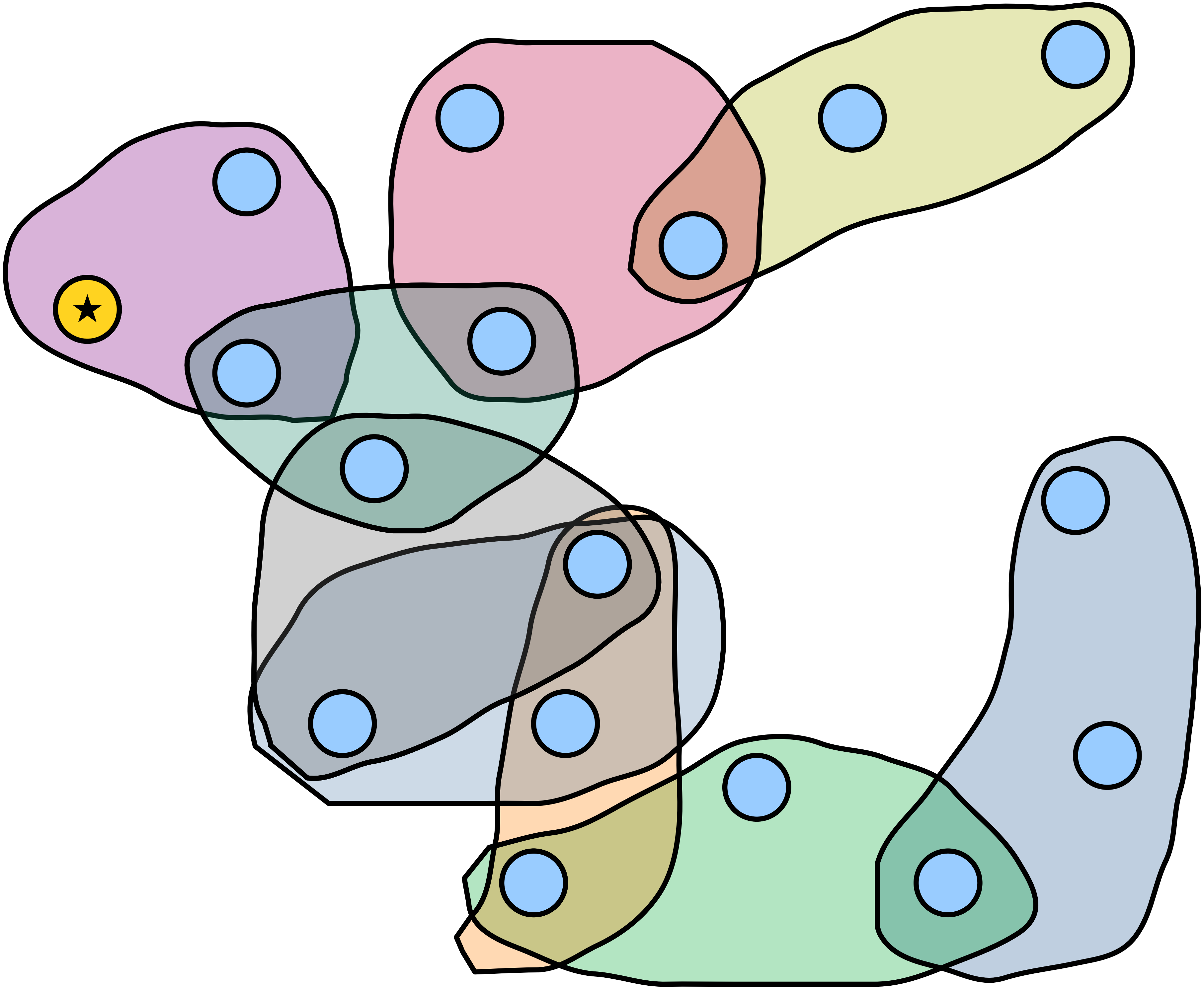


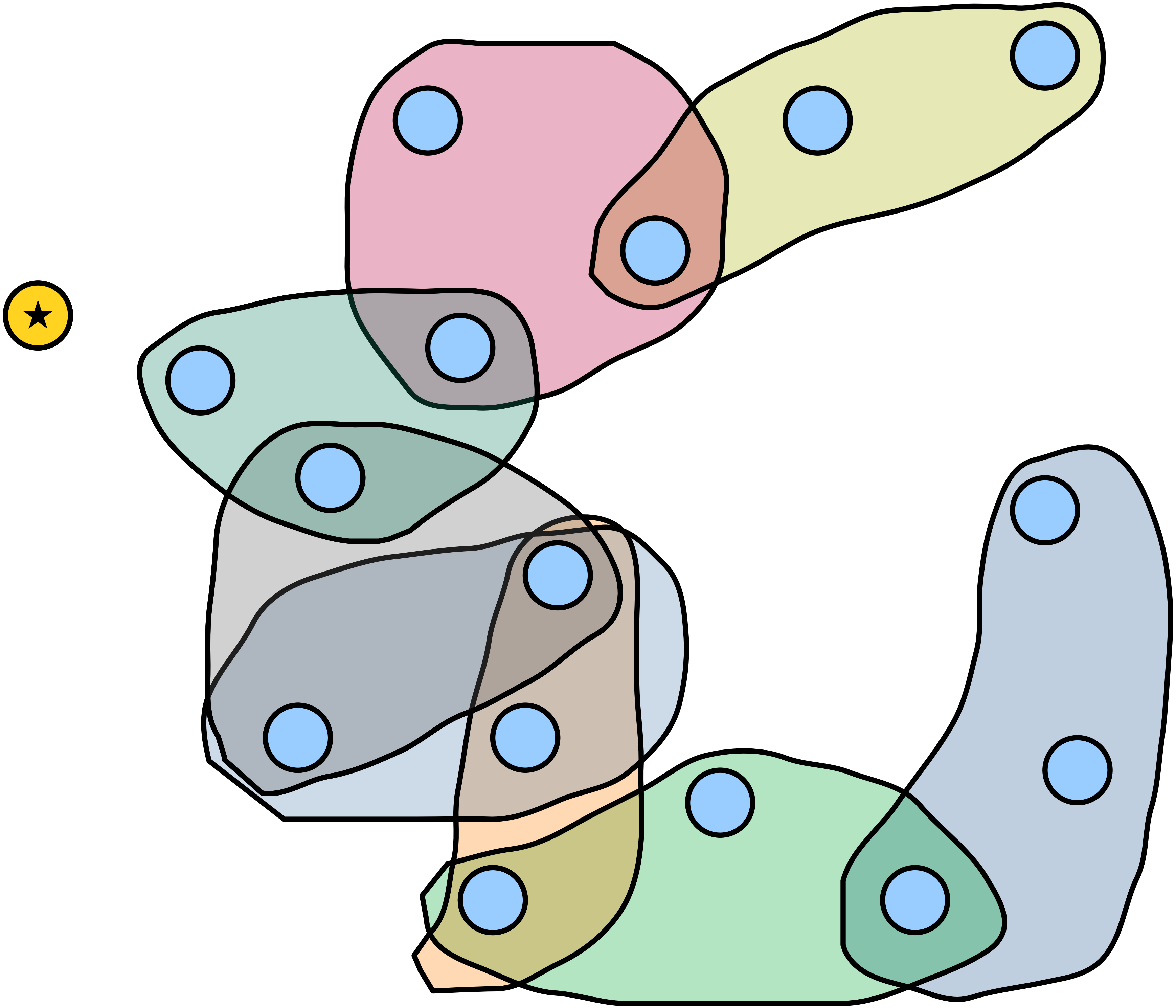


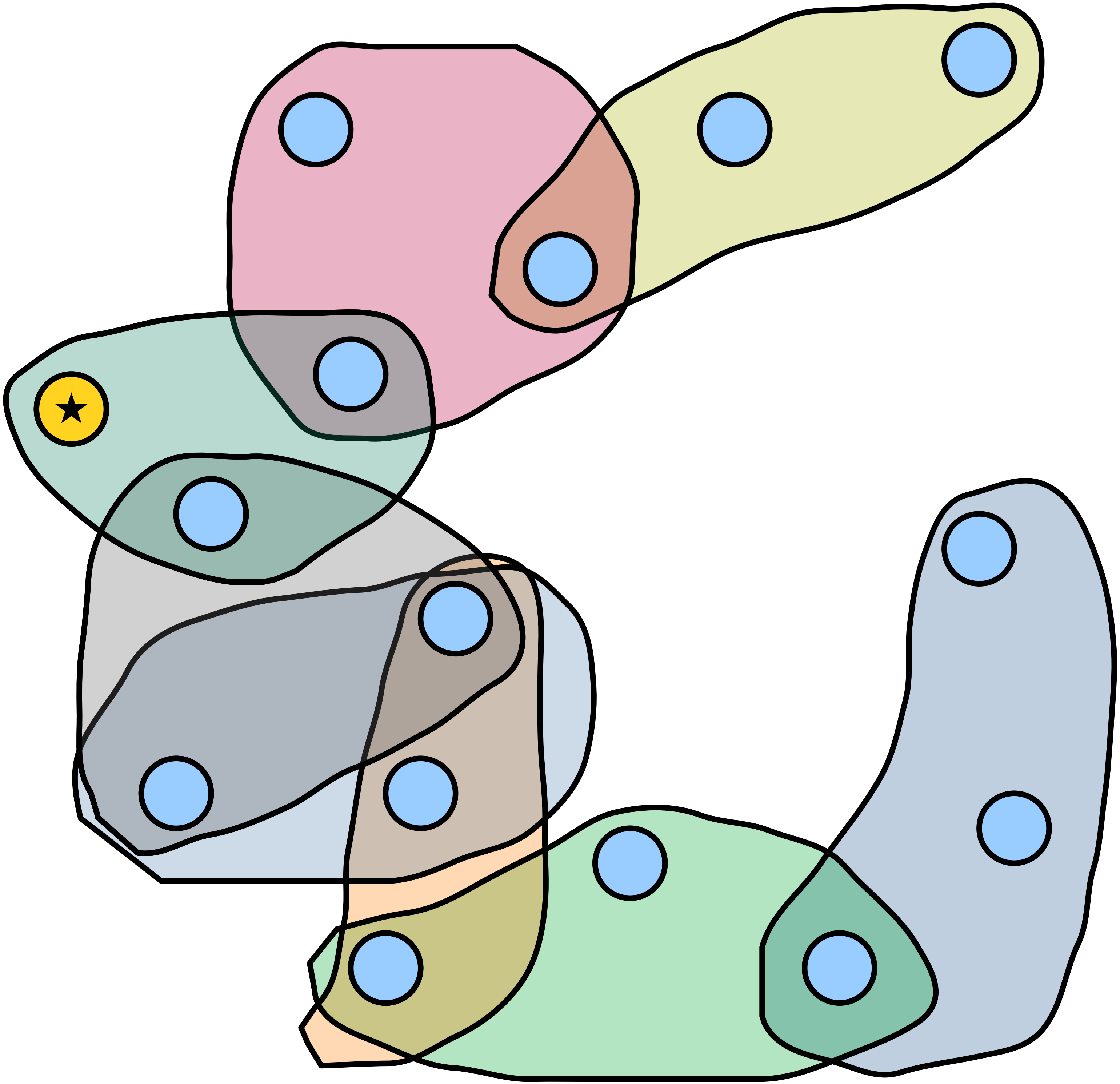


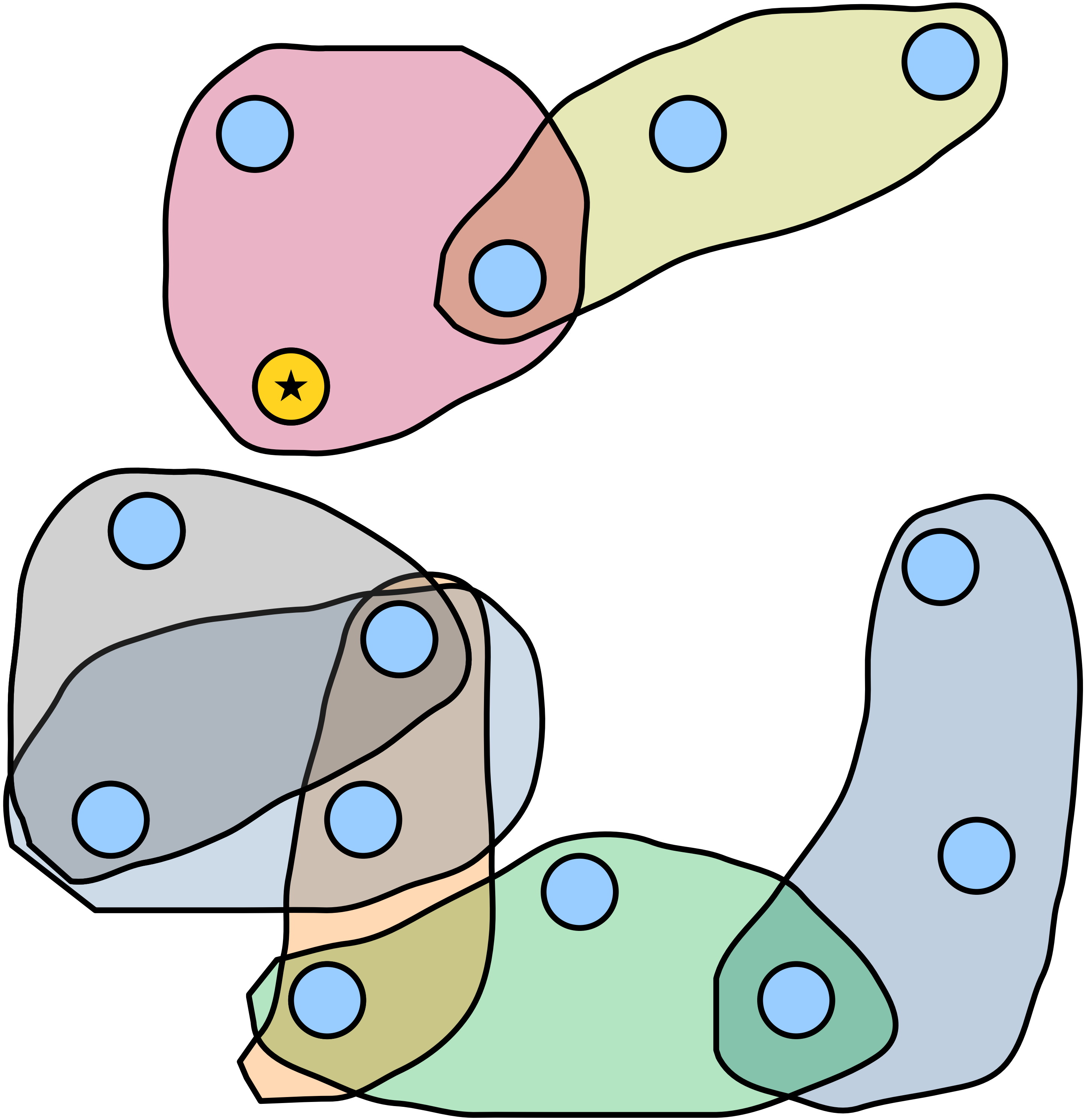


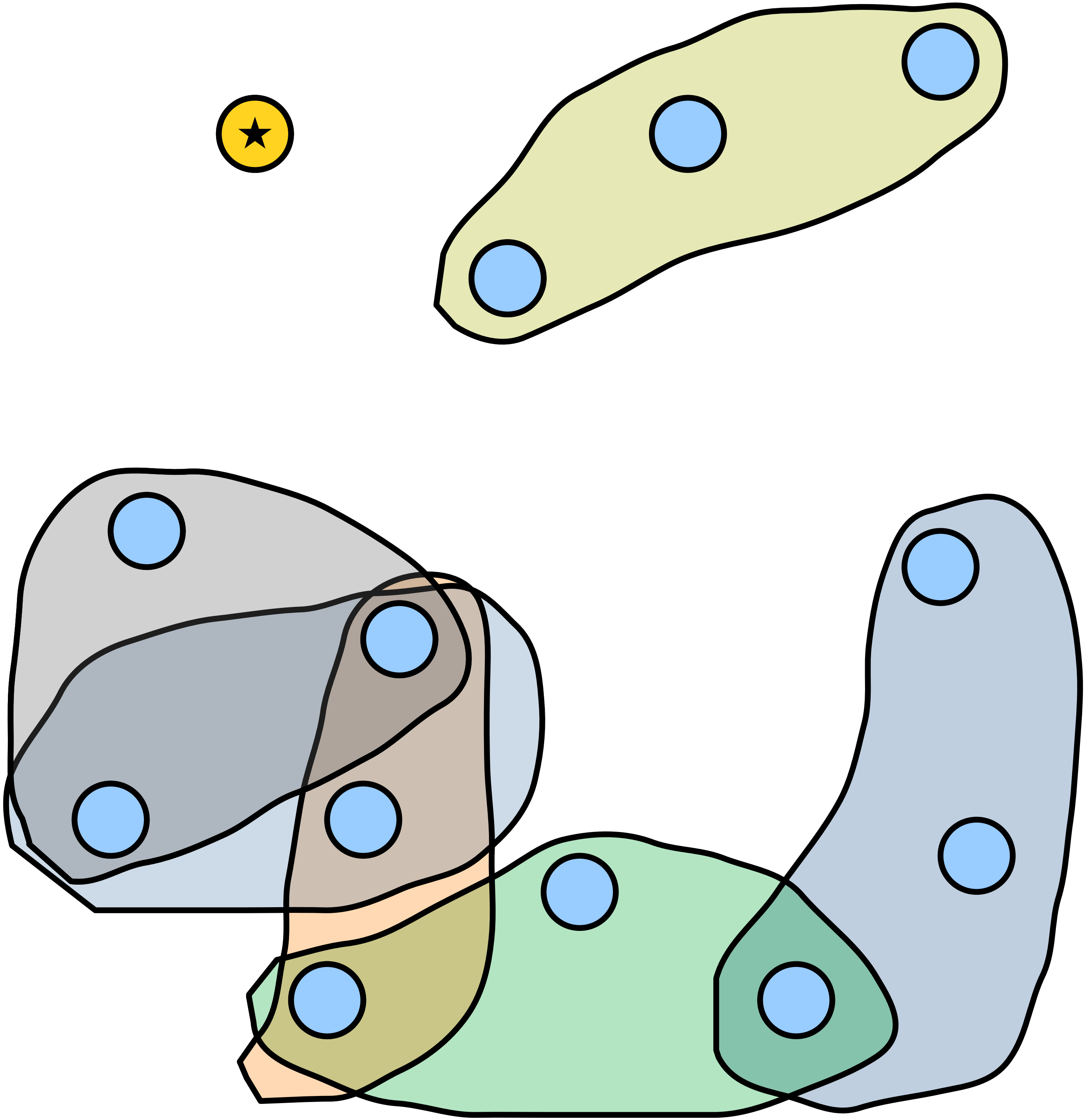


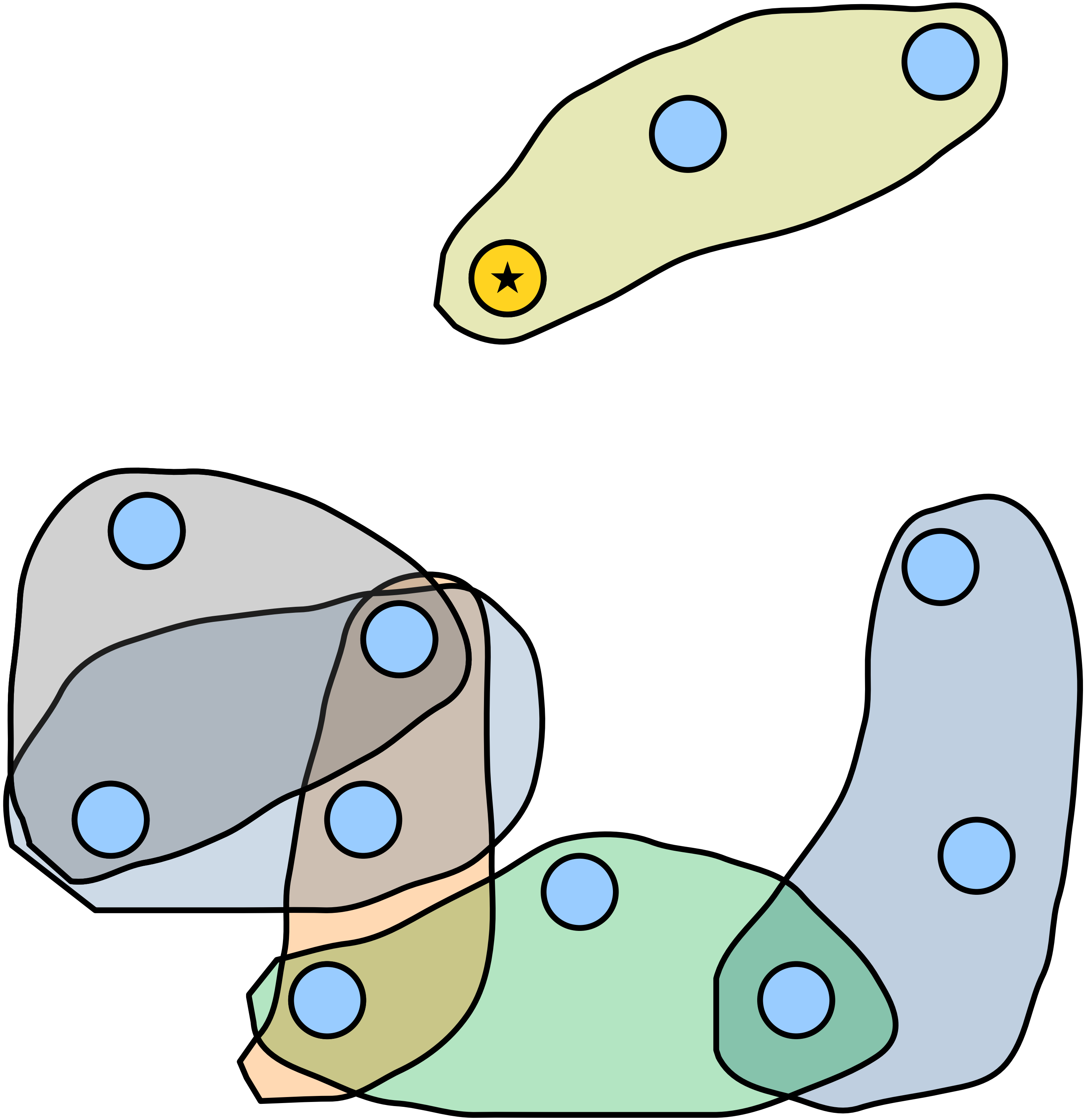


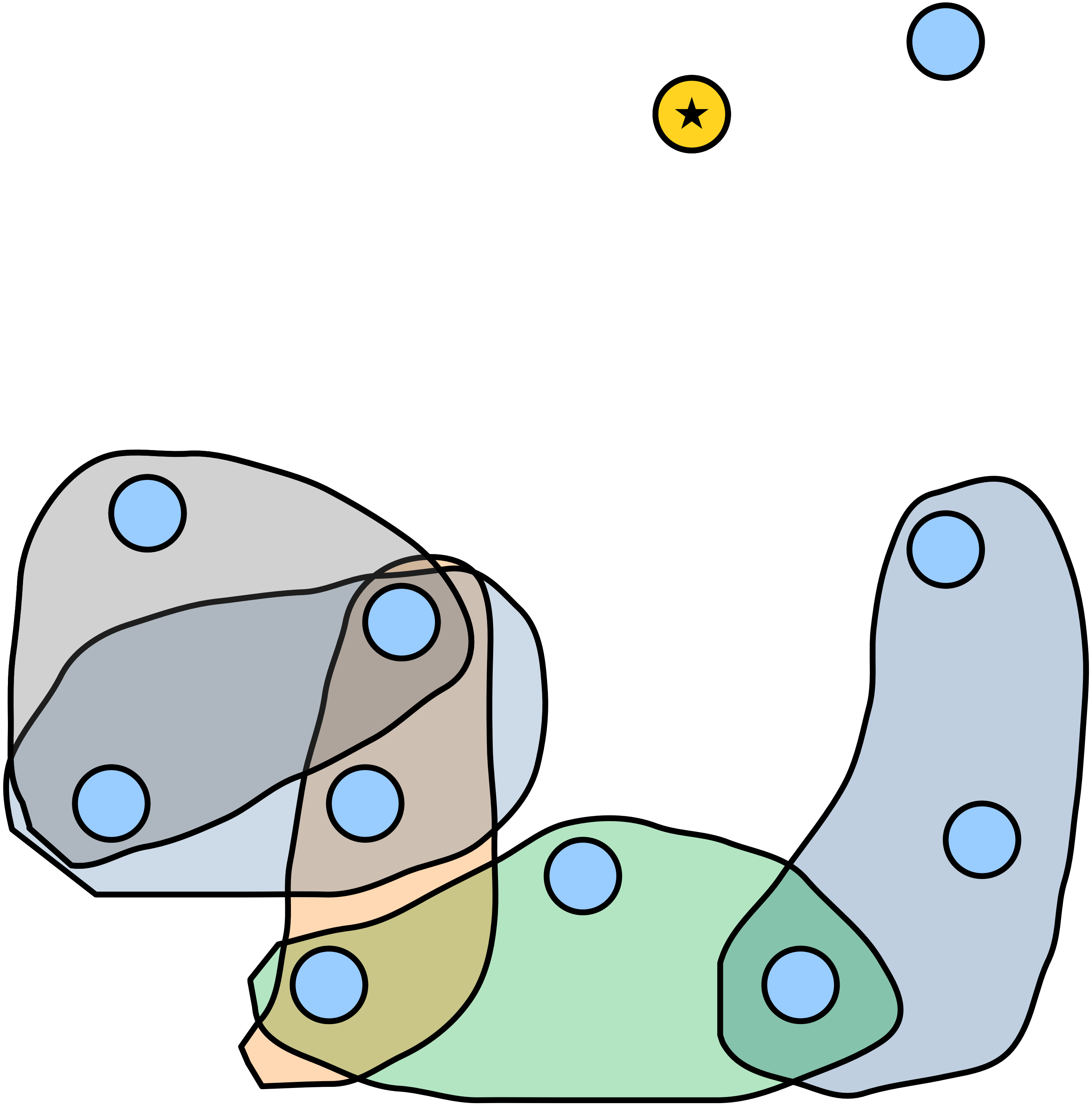


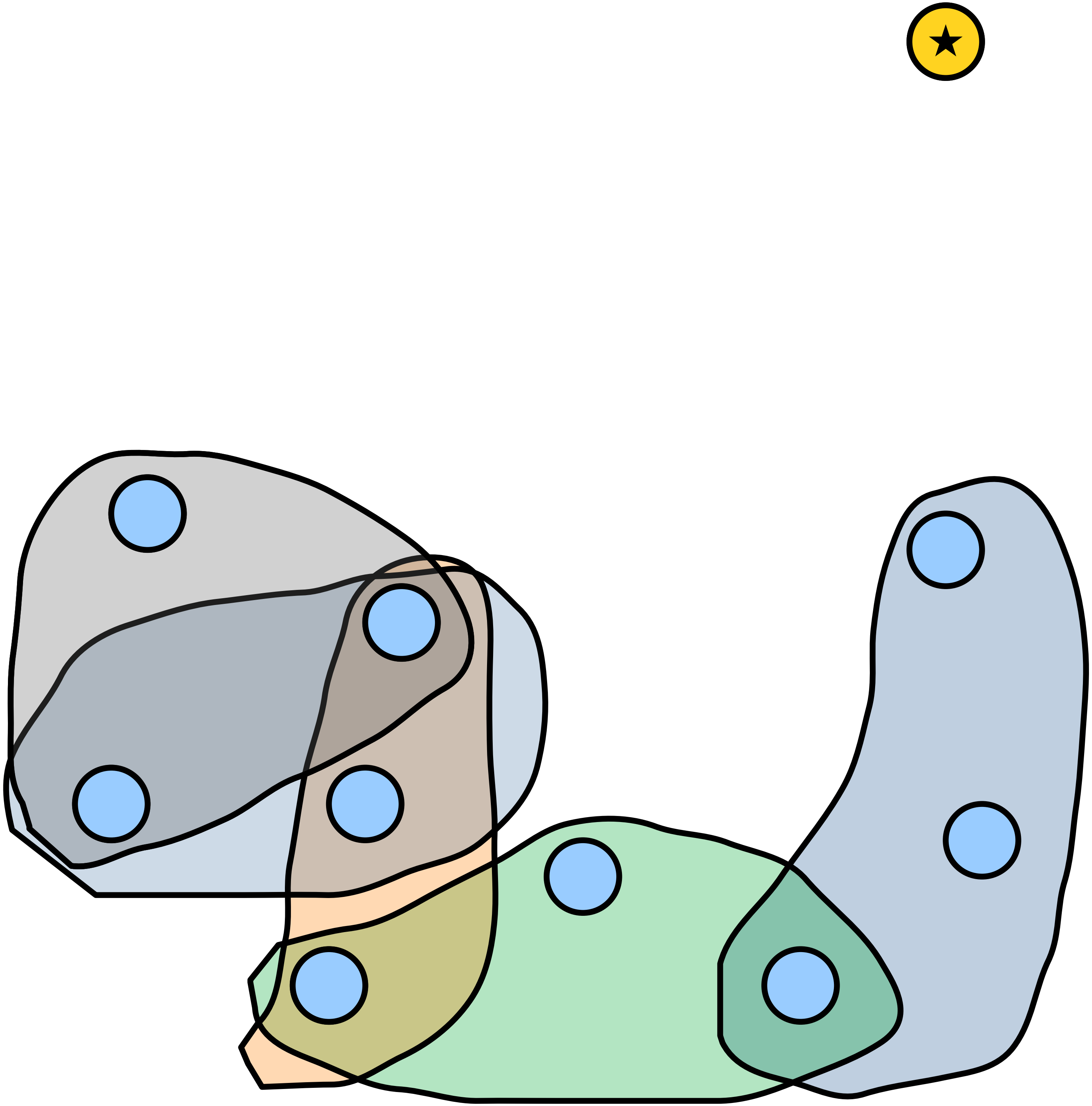


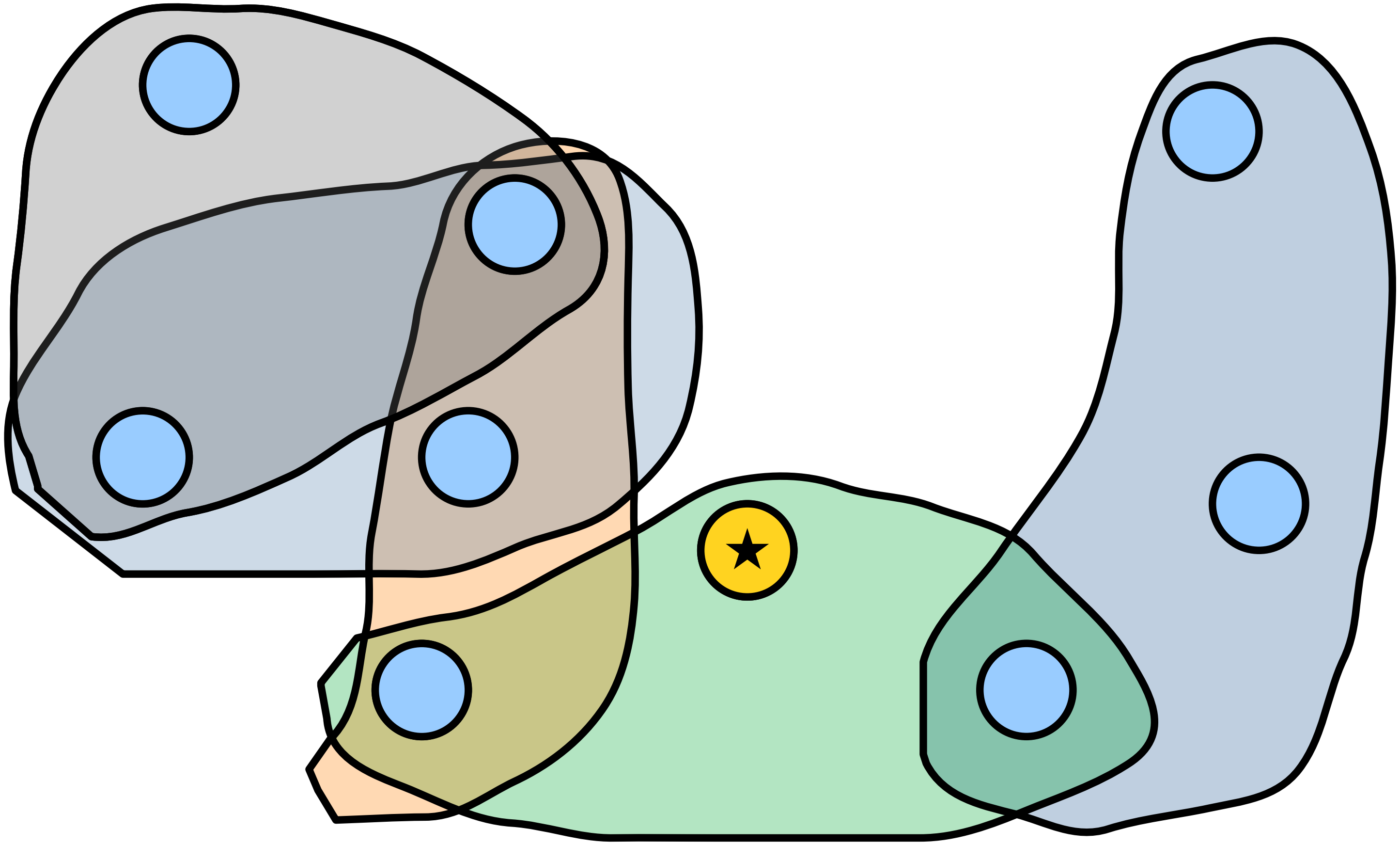


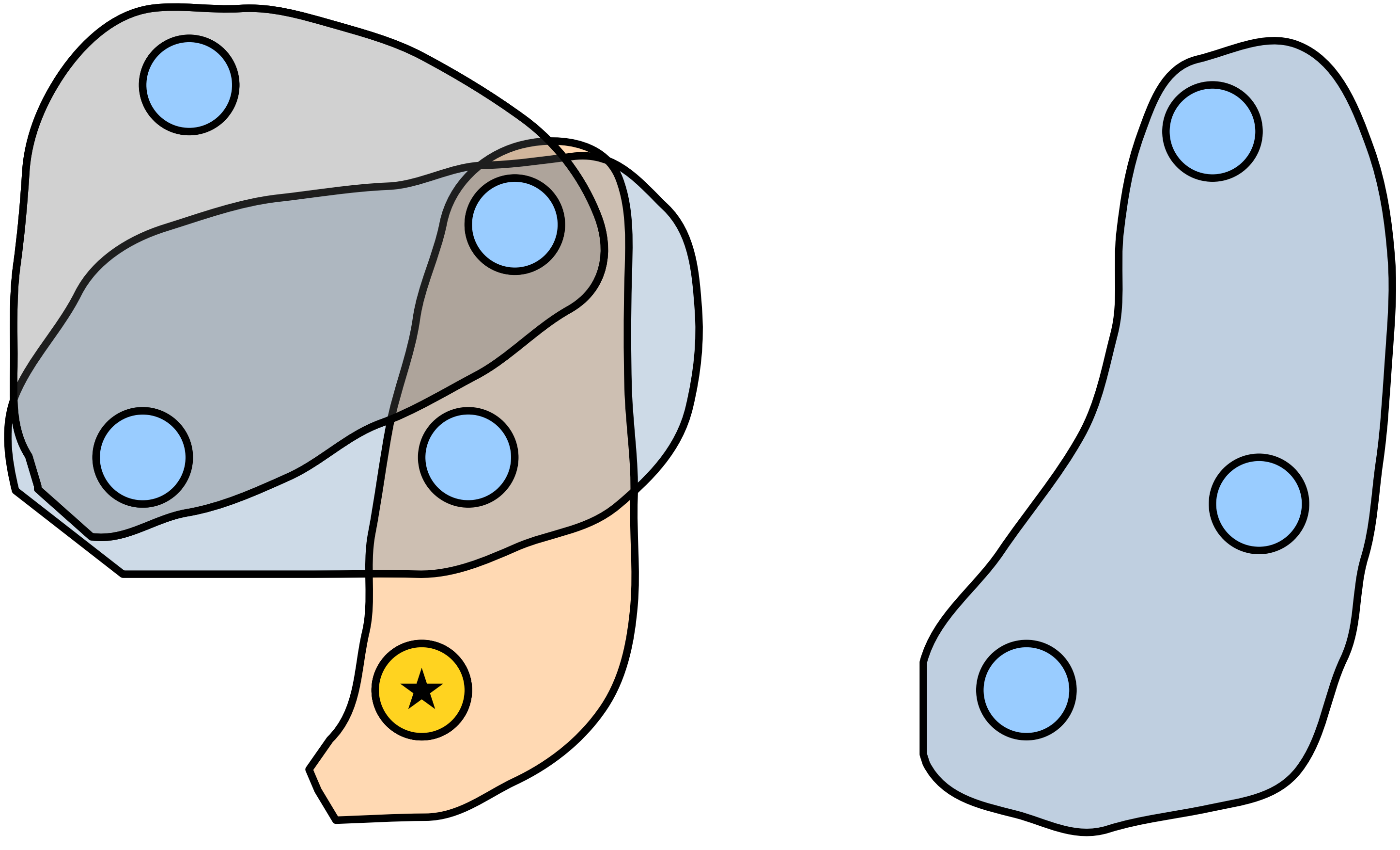


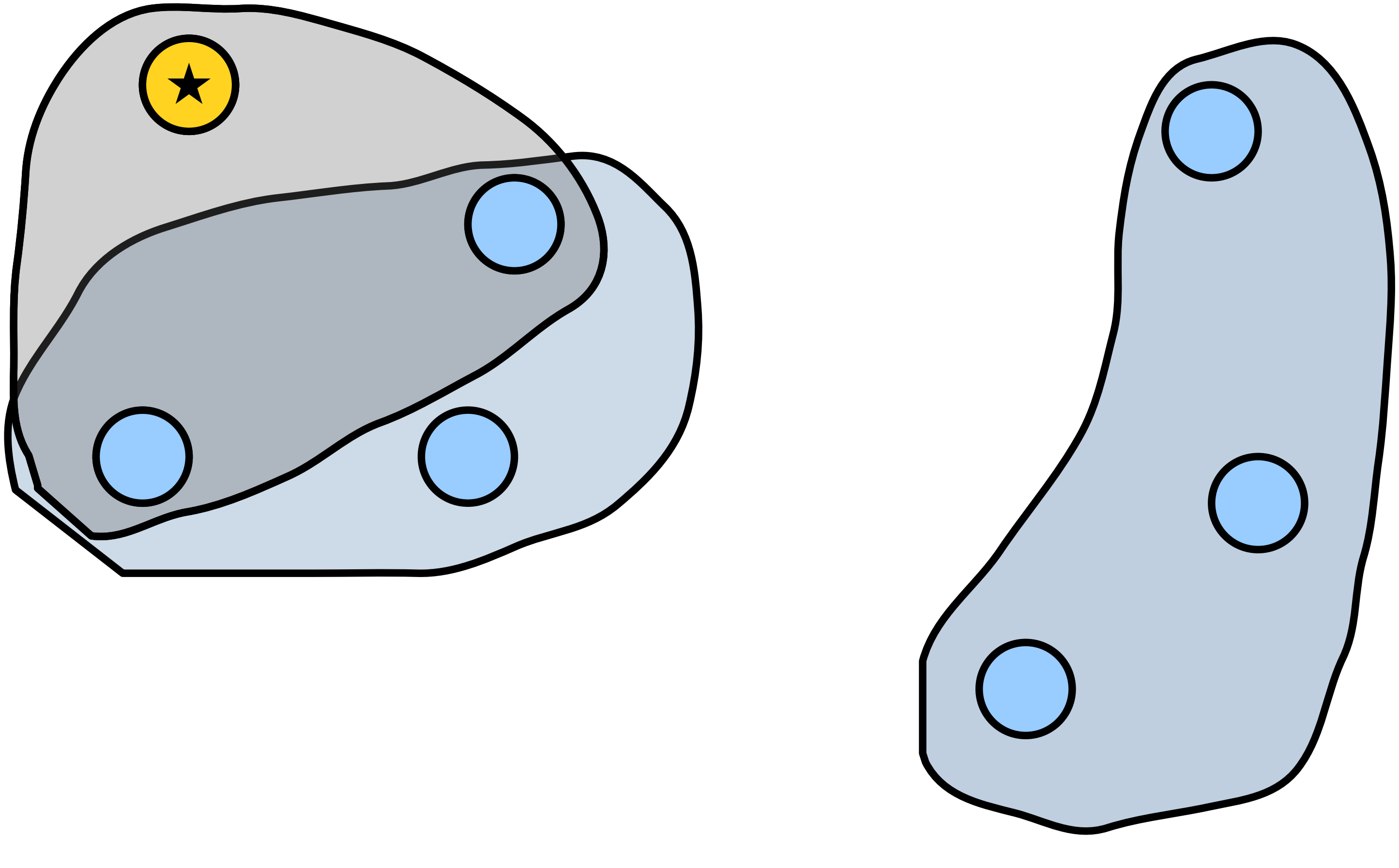


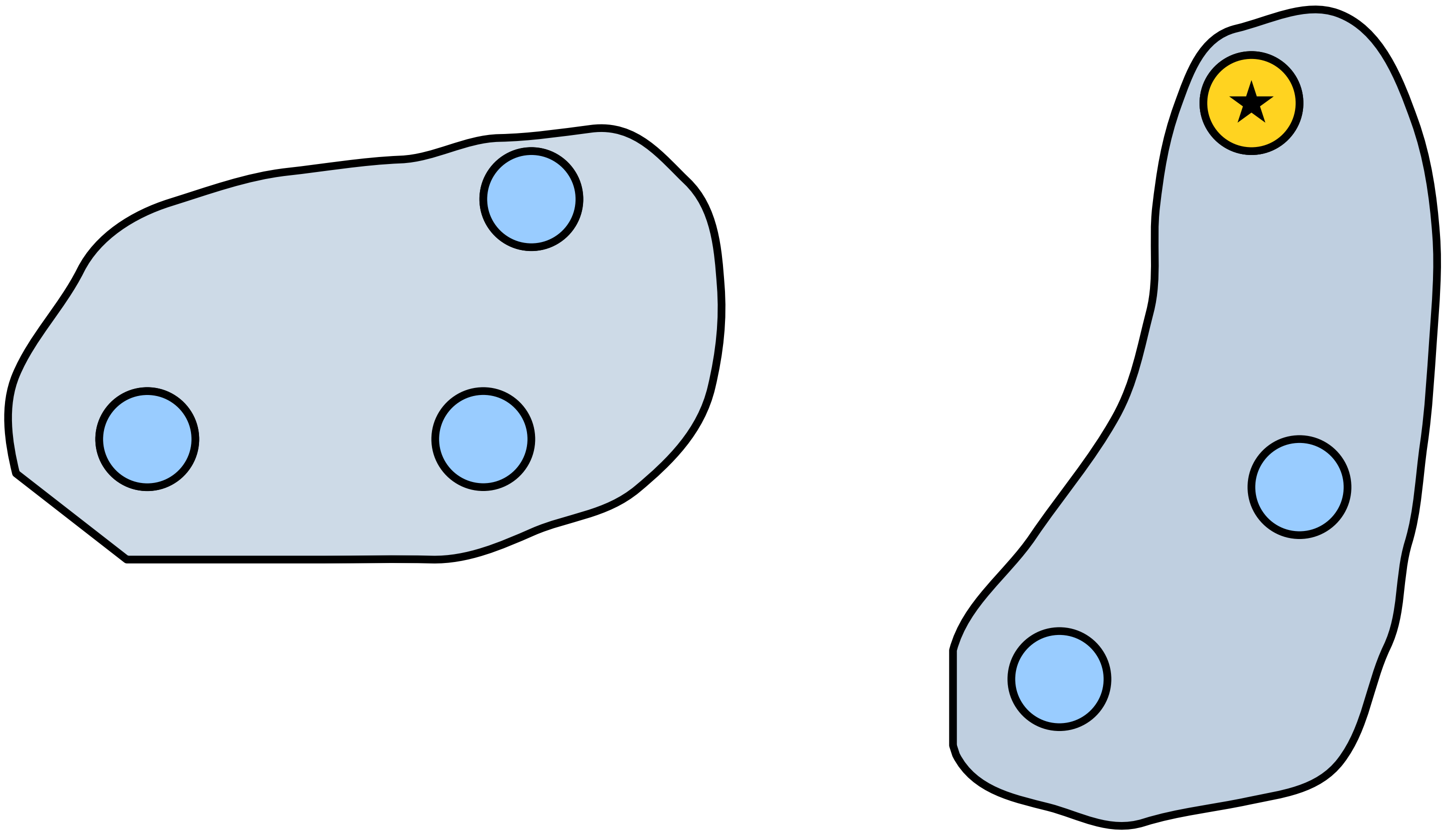


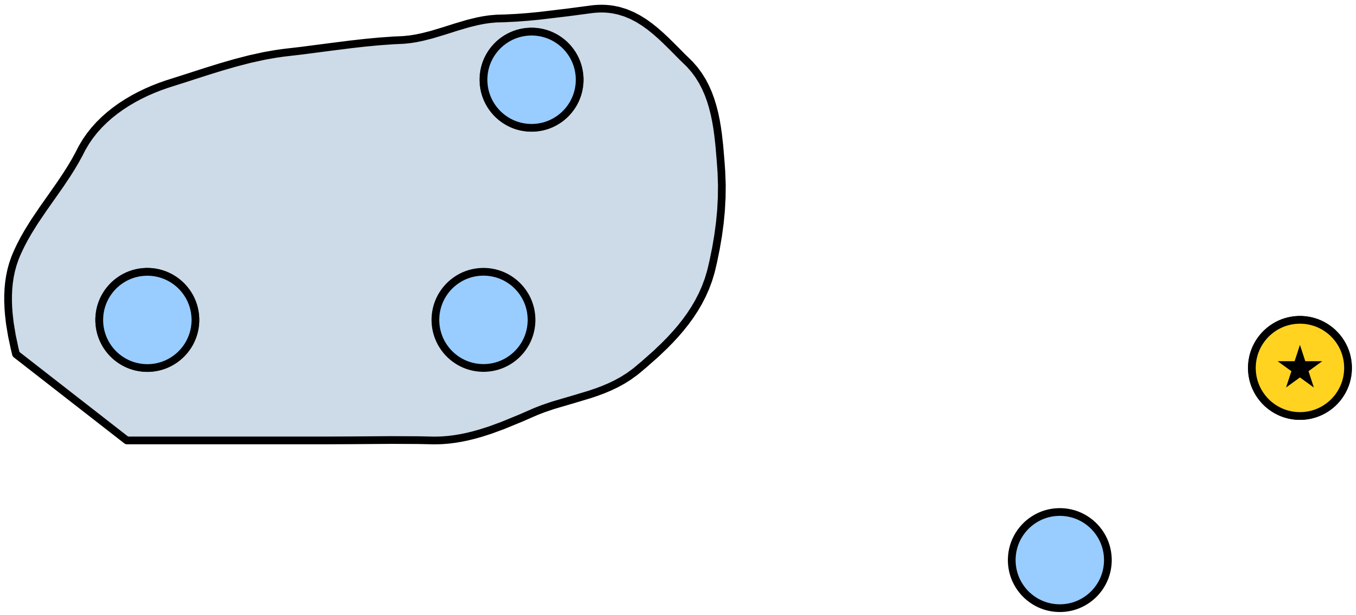


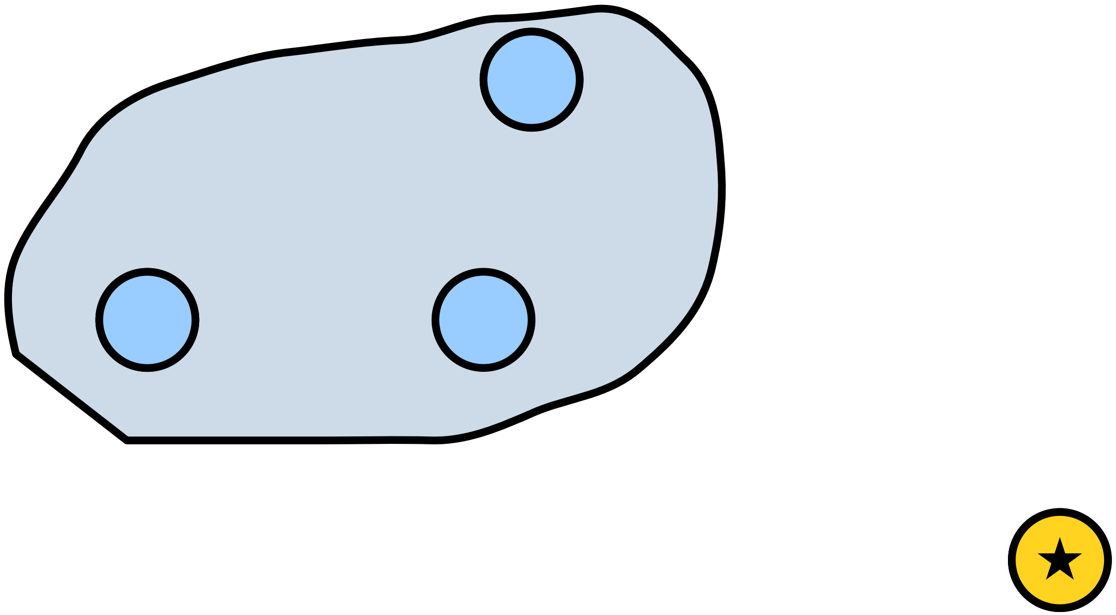


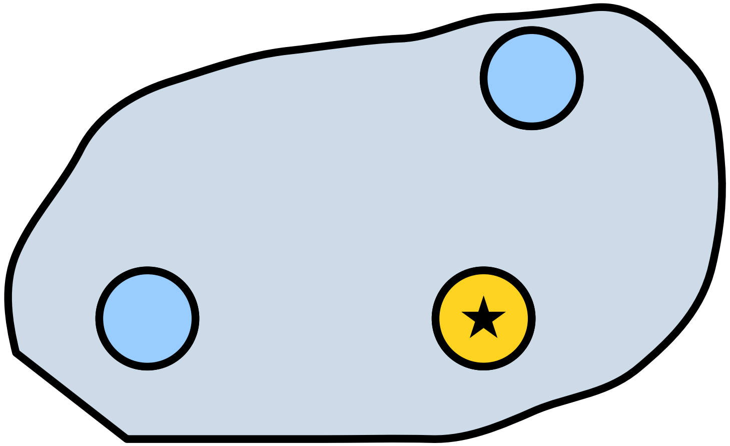


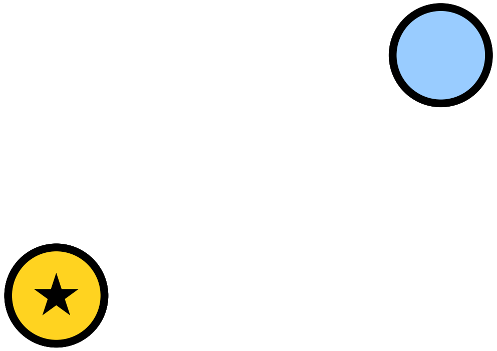








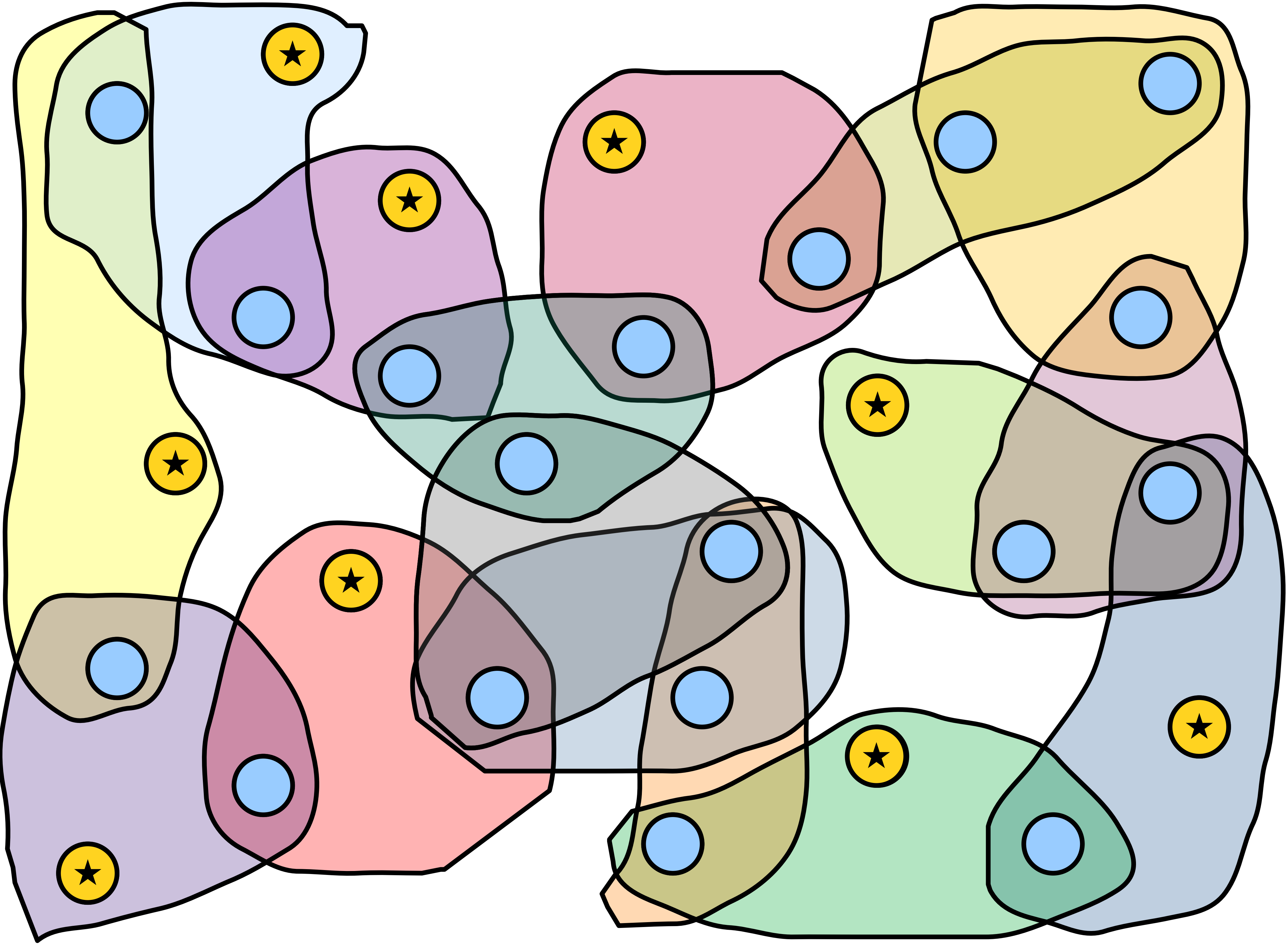


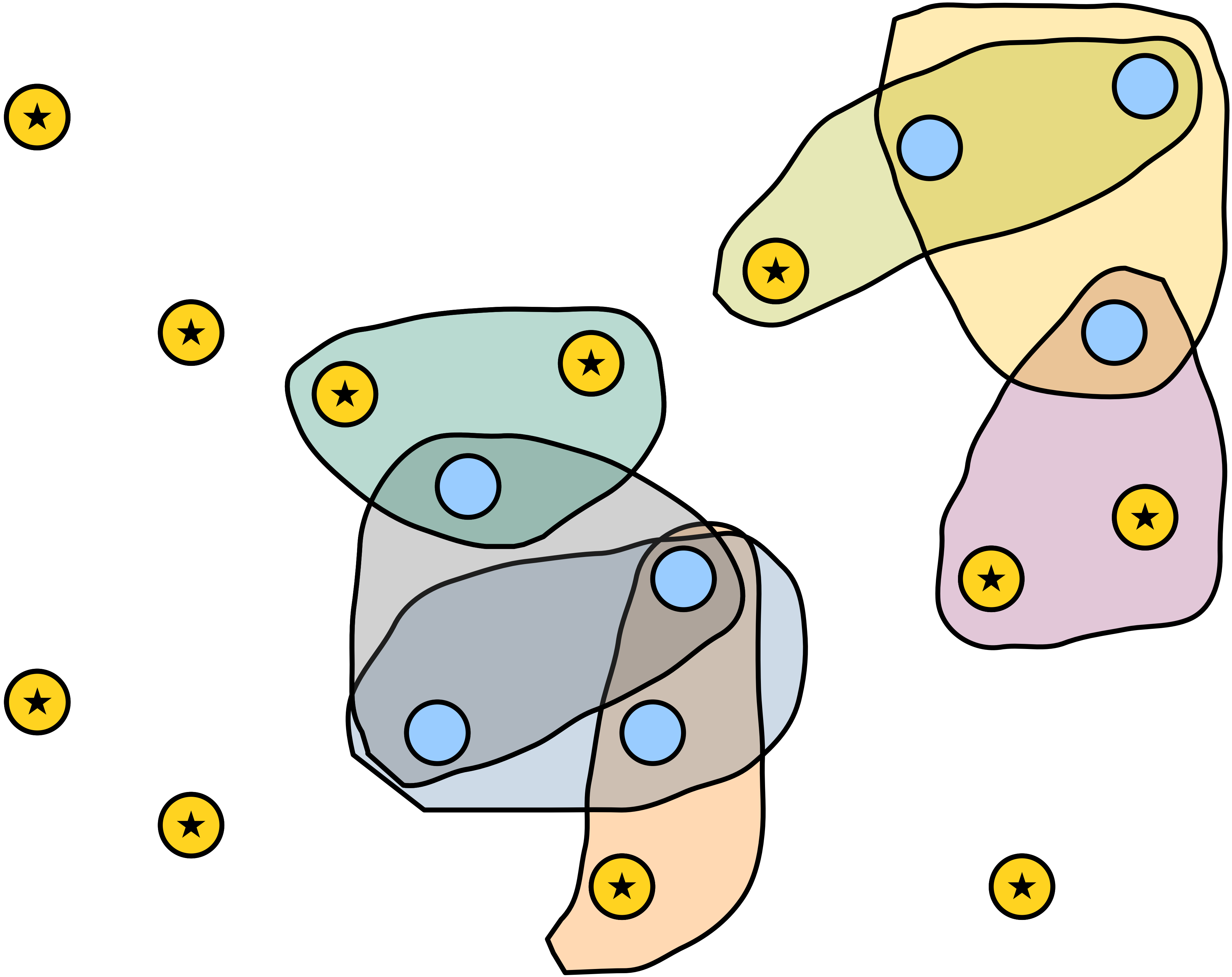


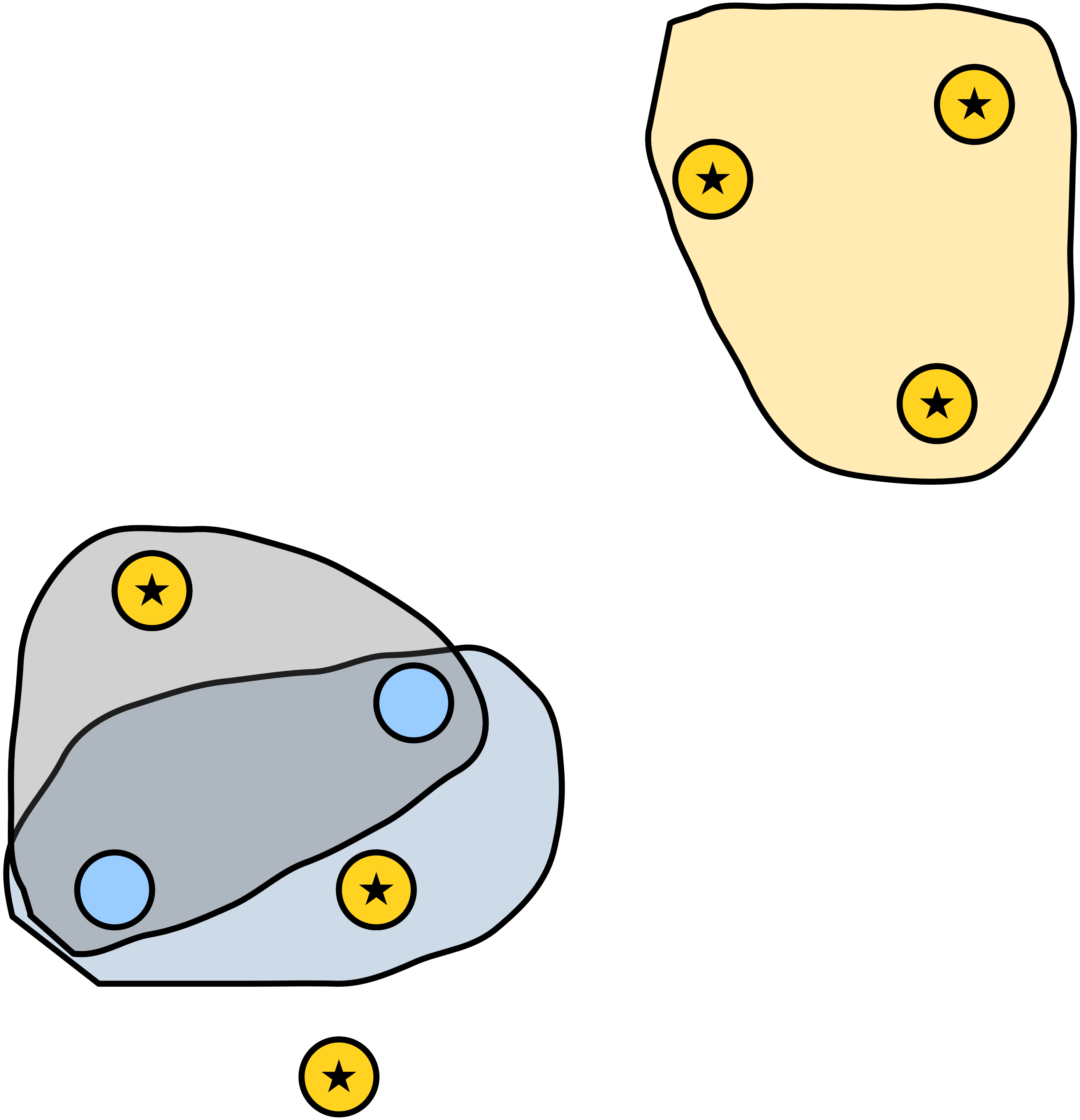
Question: Given a random hypergraph with V nodes and $E = \alpha V$ hyperedges, each of which links d nodes, what's the probability that every node is peeled?

Claim: It’s easier to analyze a parallel version of this process.

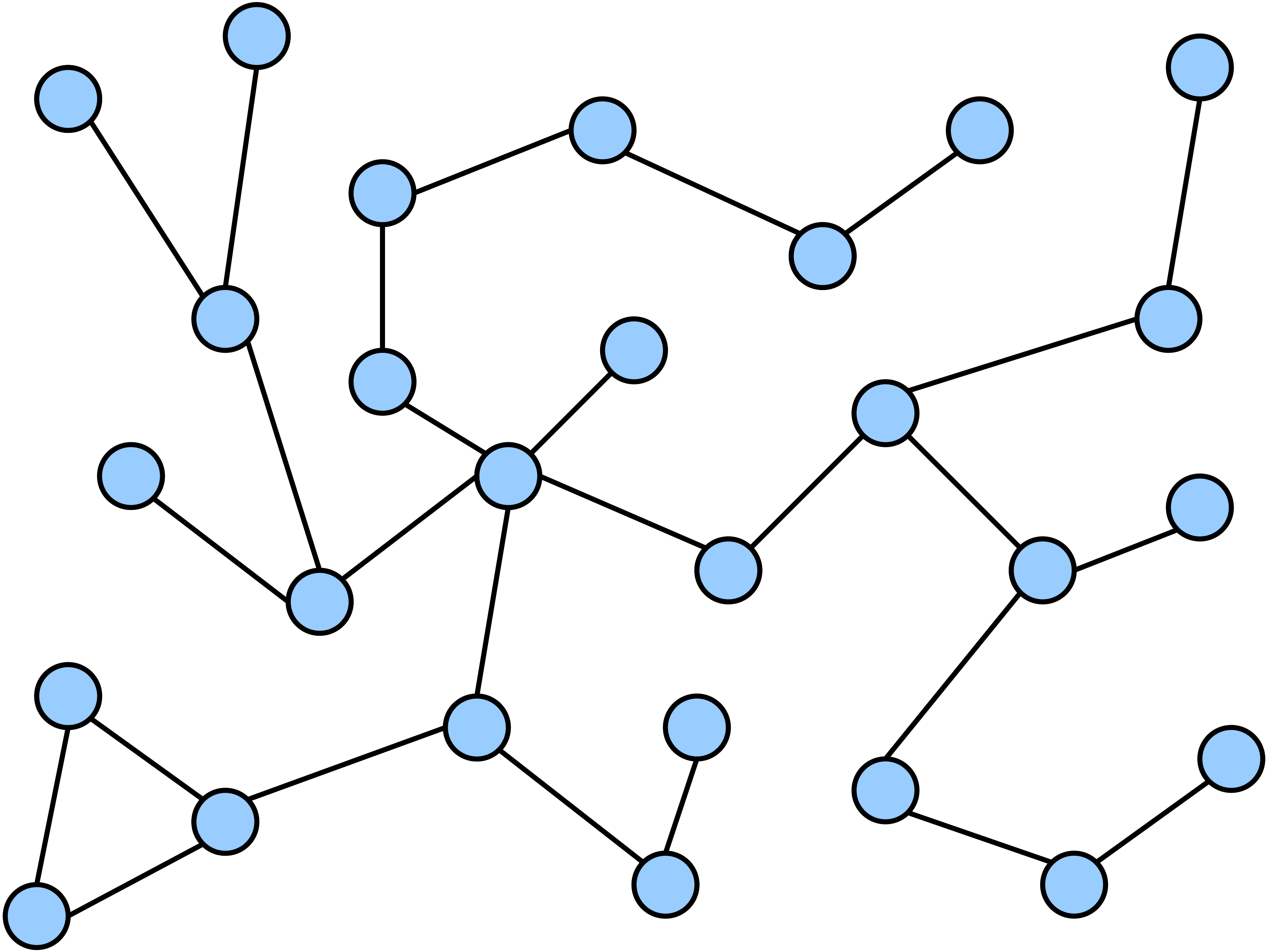
One “round” consists of identifying and peeling all nodes of degree 0 or 1.

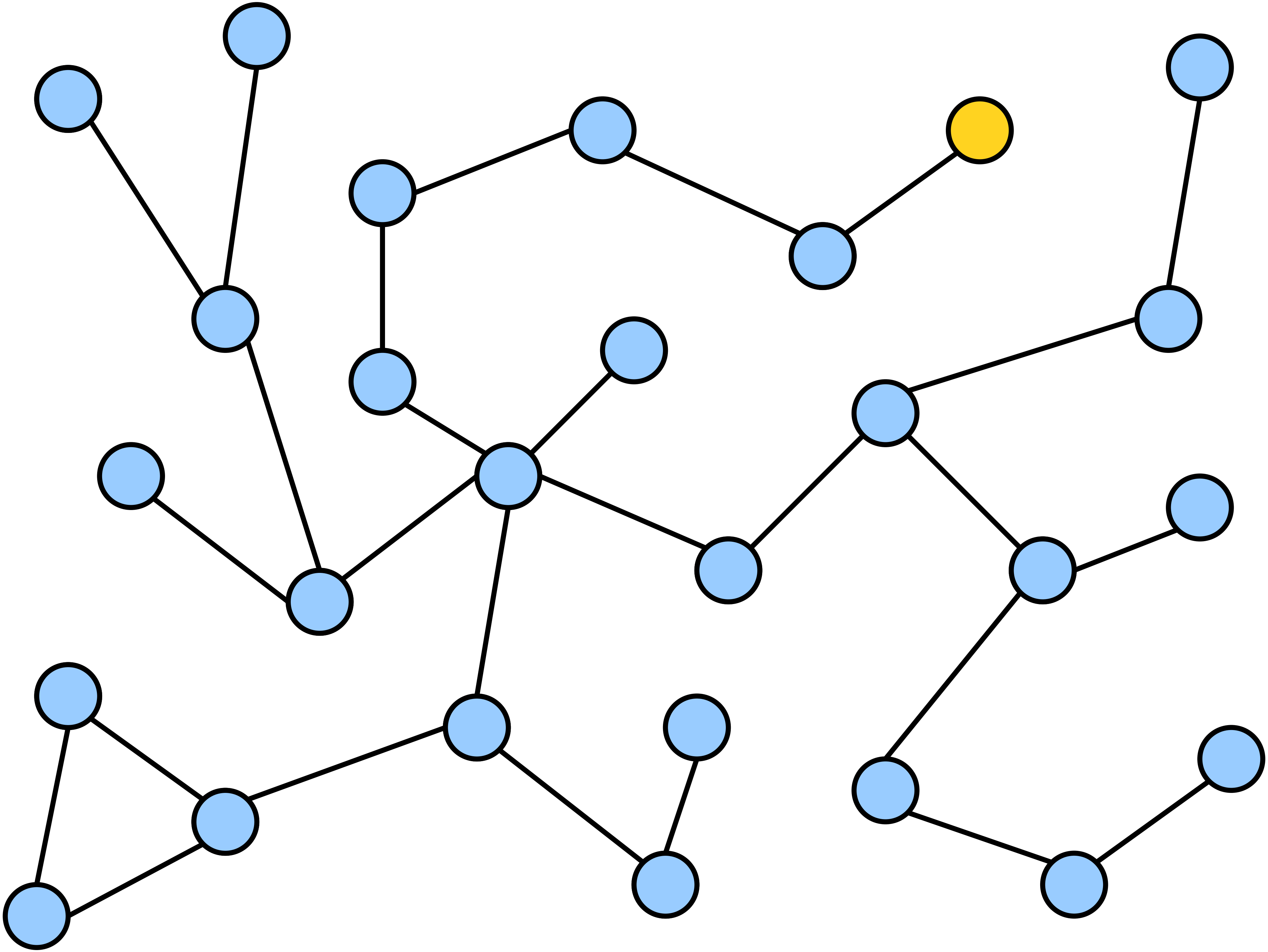


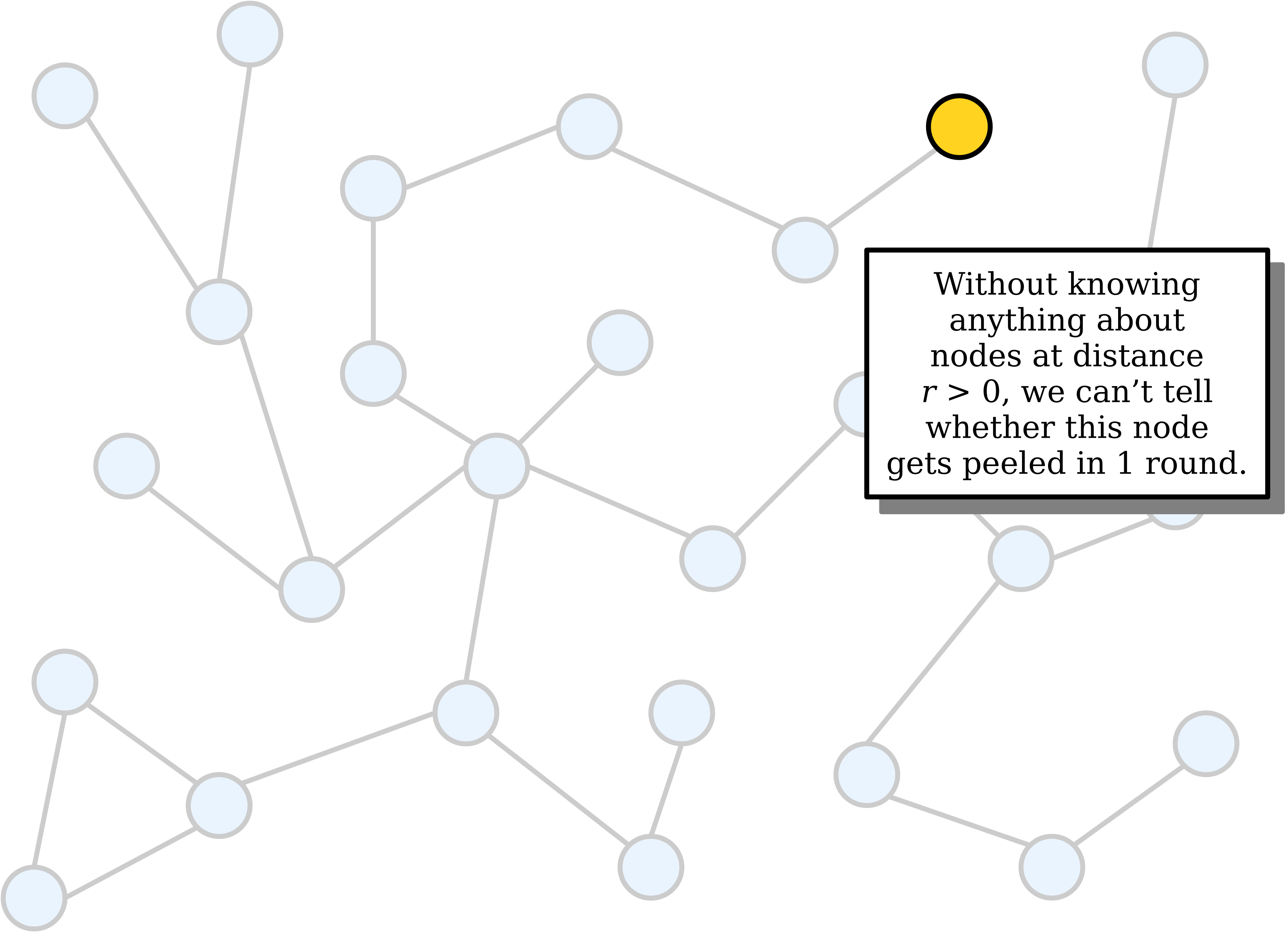




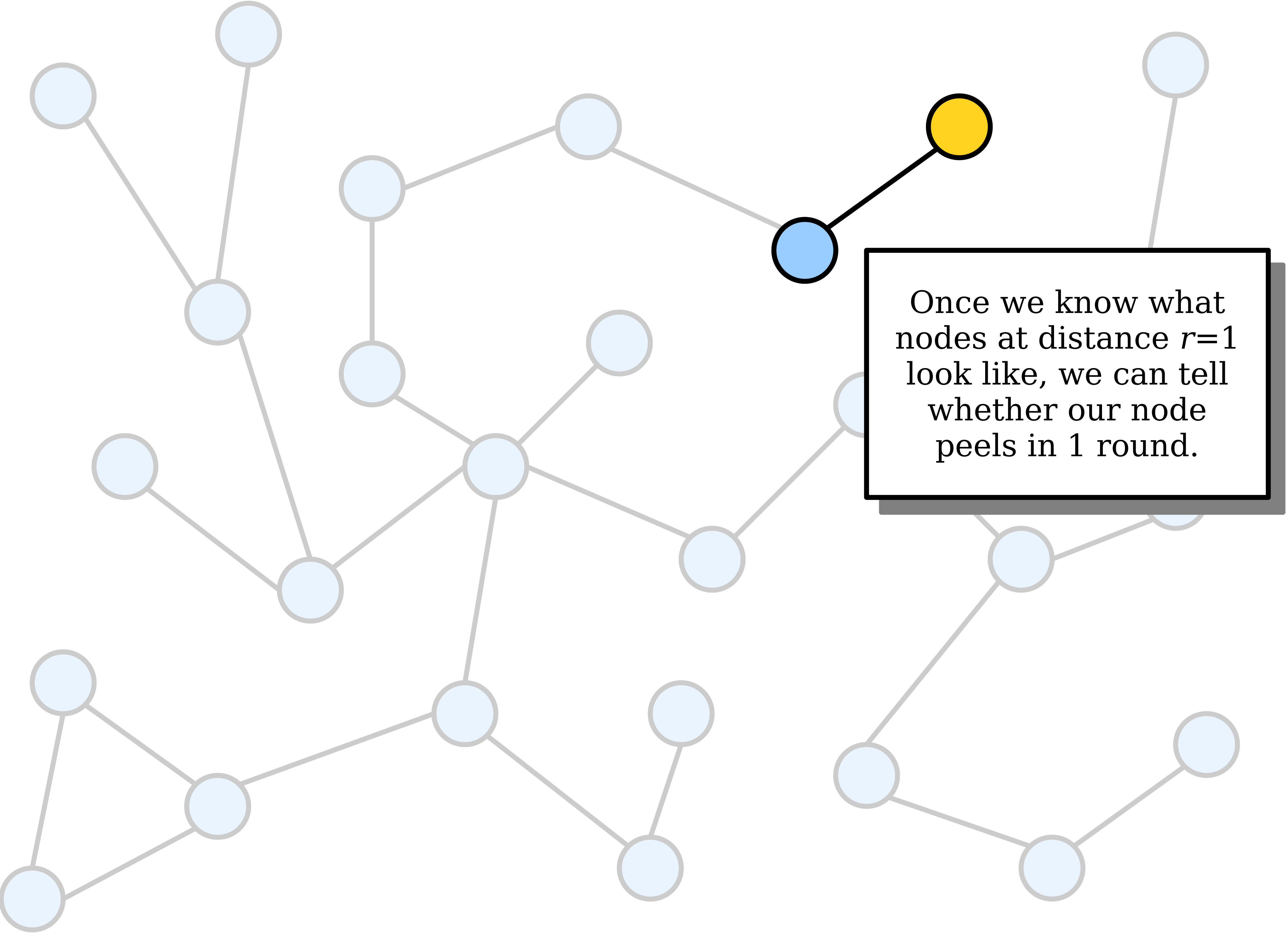
Goal: Determine the probability that a node $v \in V$ is peeled after r rounds.



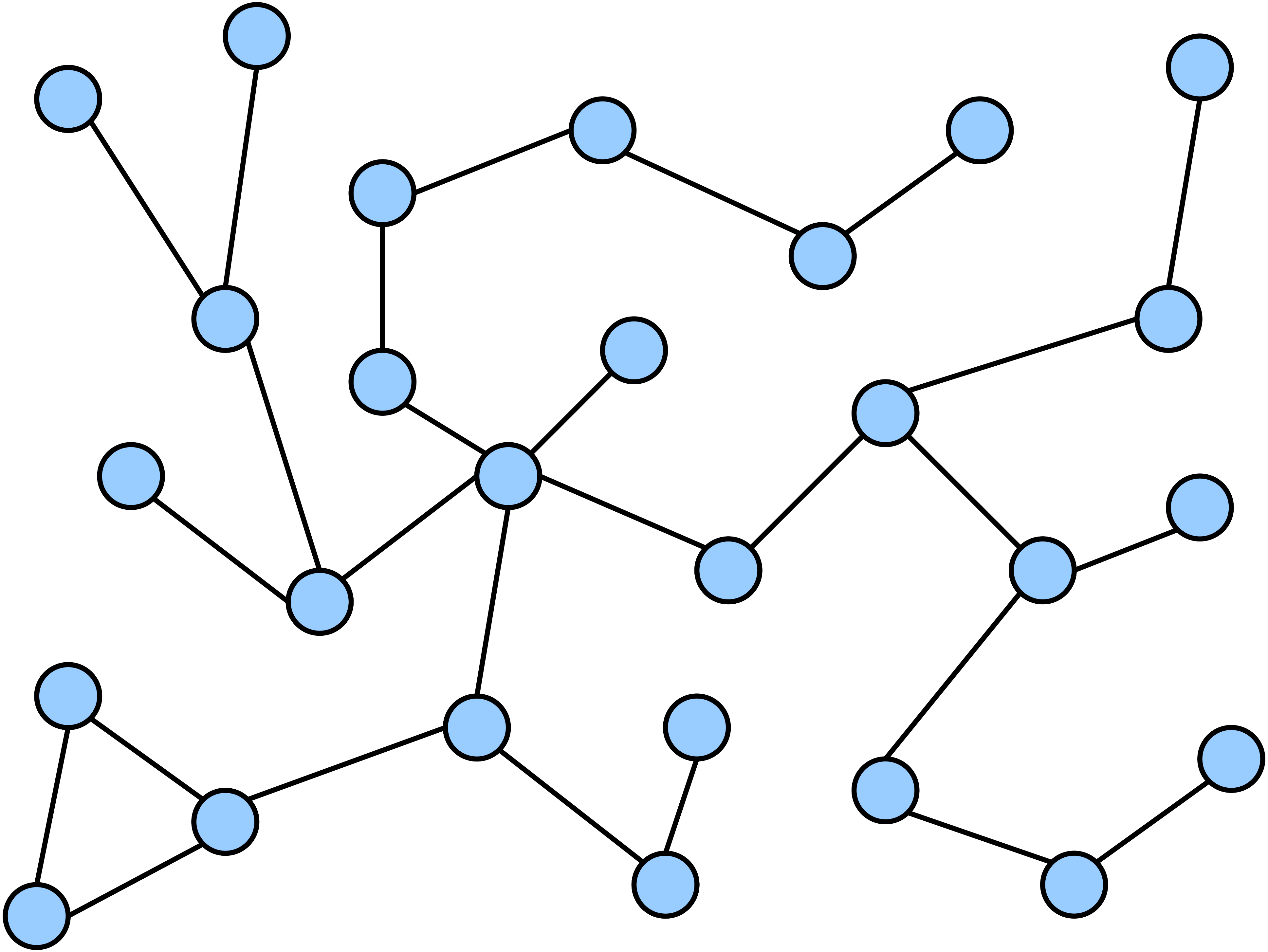


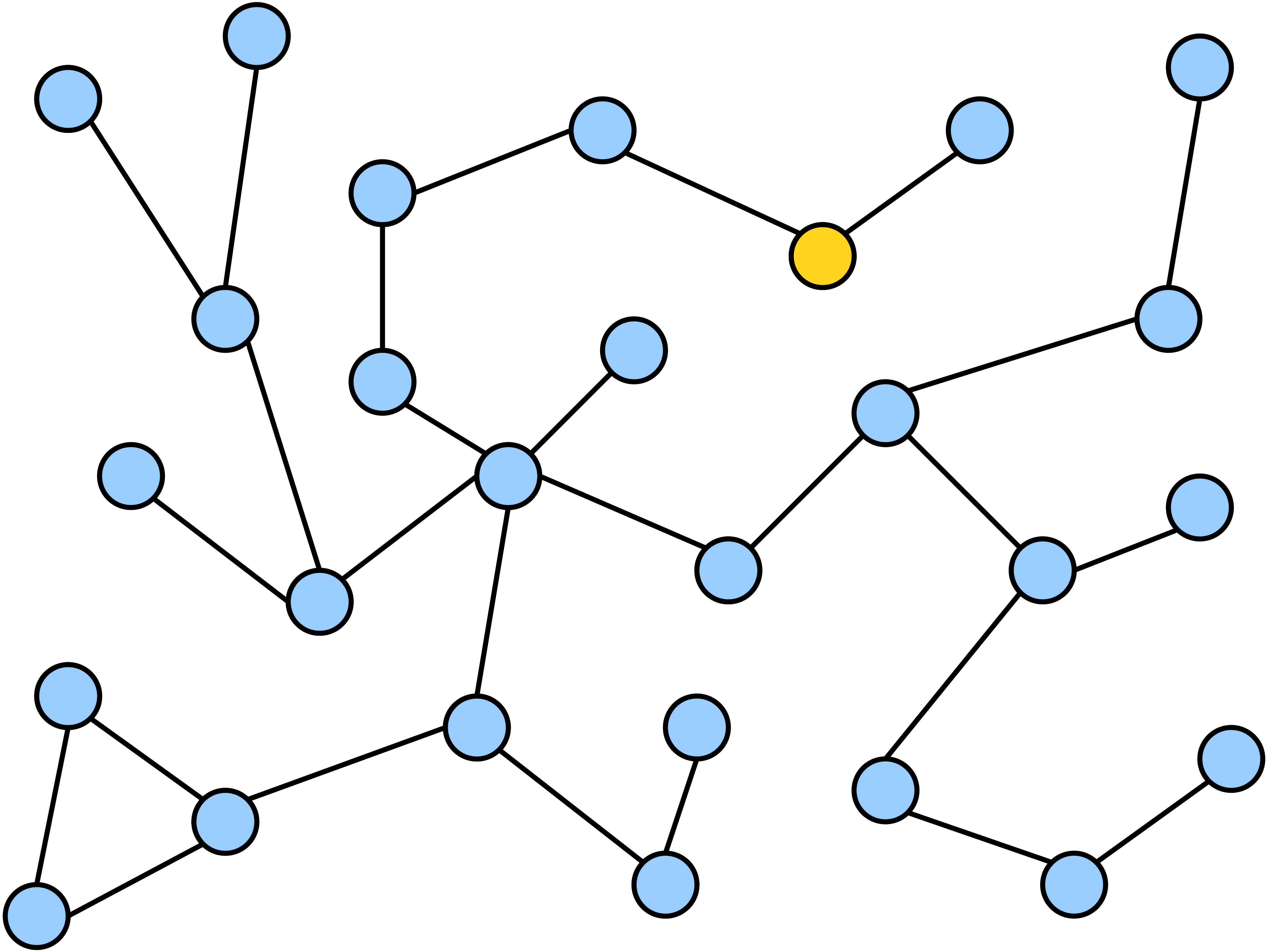


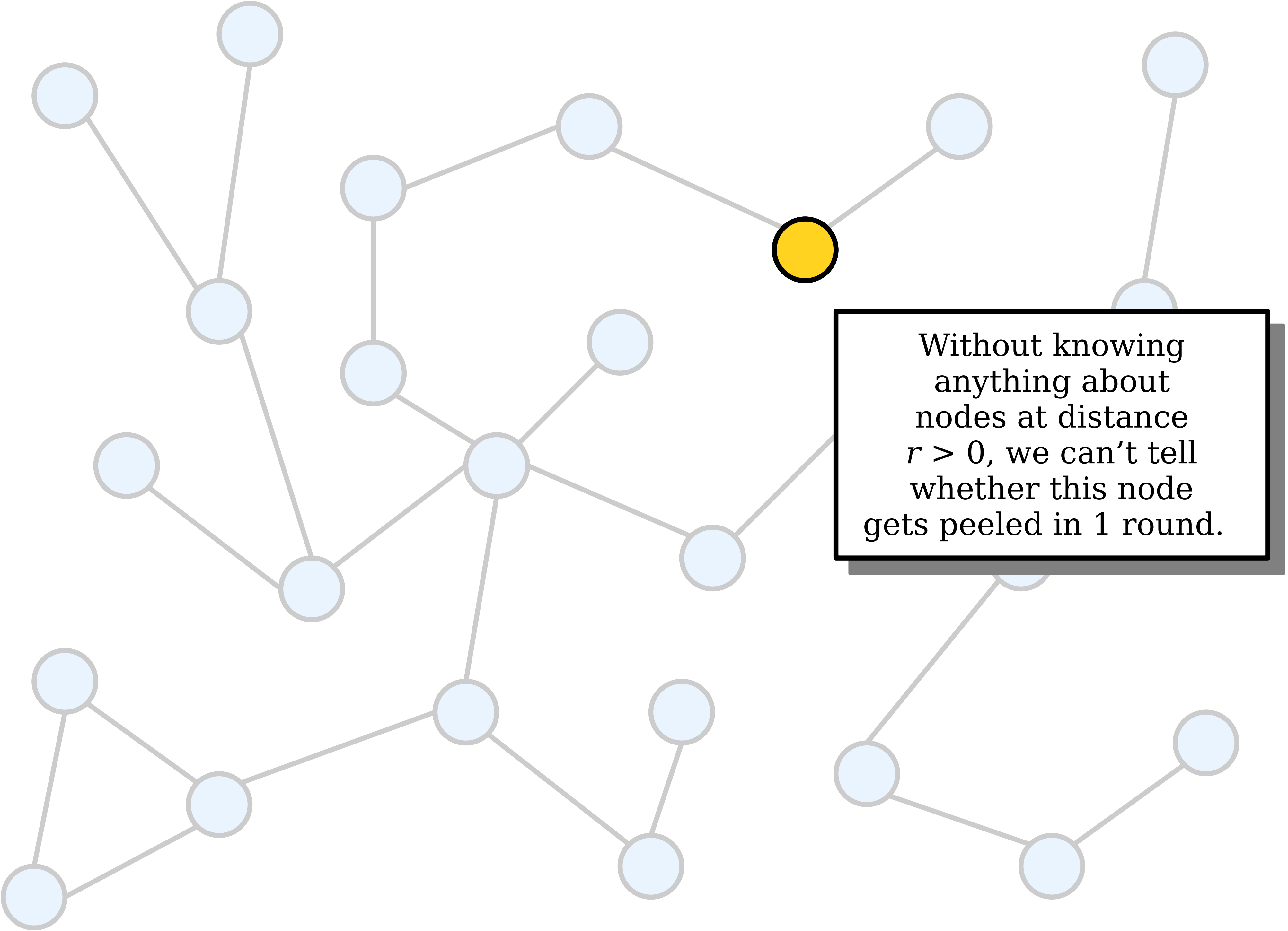
Without knowing anything about nodes at distance $r > 0$, we can’t tell whether this node gets peeled in 1 round.



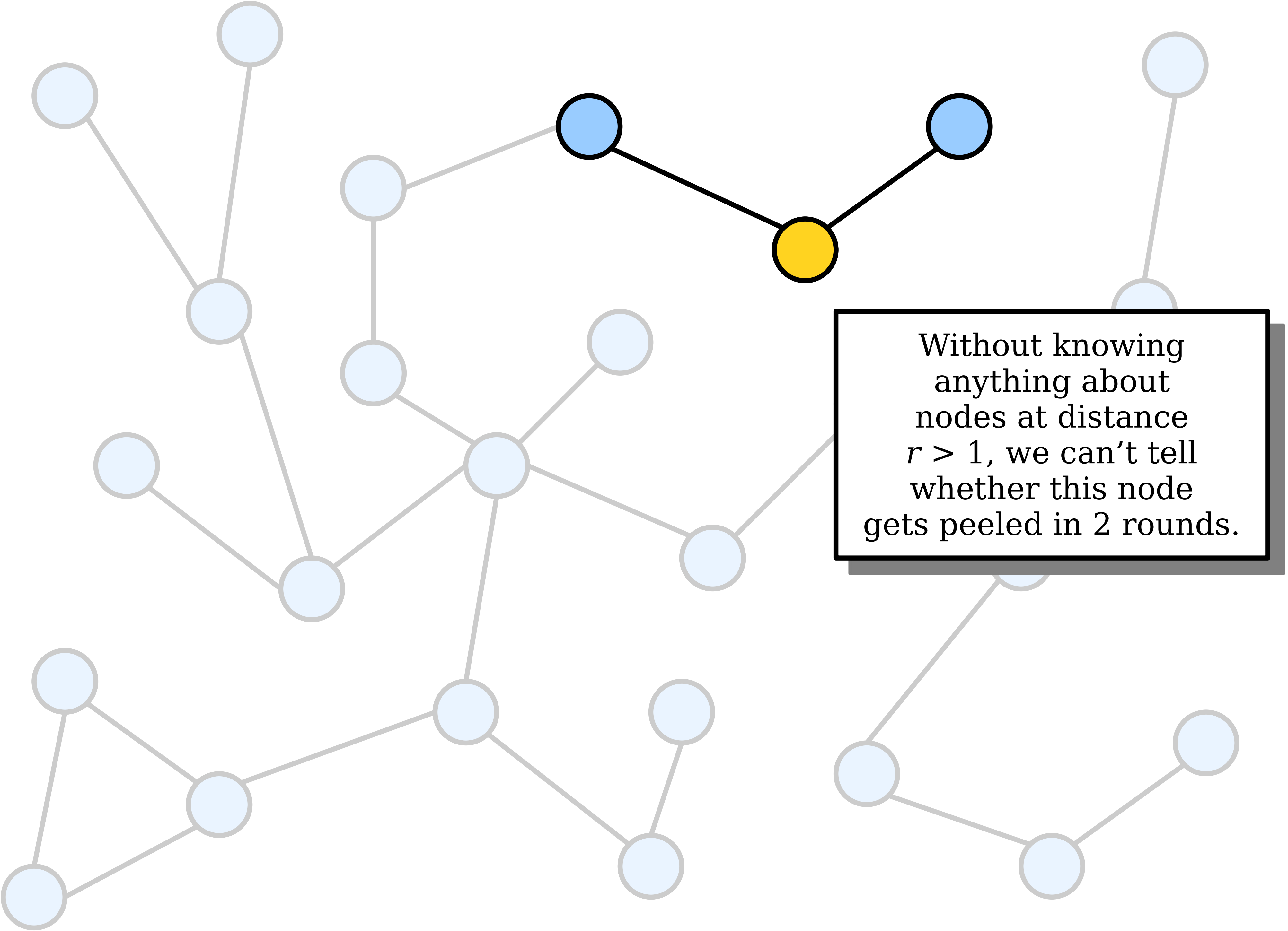
Once we know what nodes at distance r=1 look like, we can tell whether our node peels in 1 round.



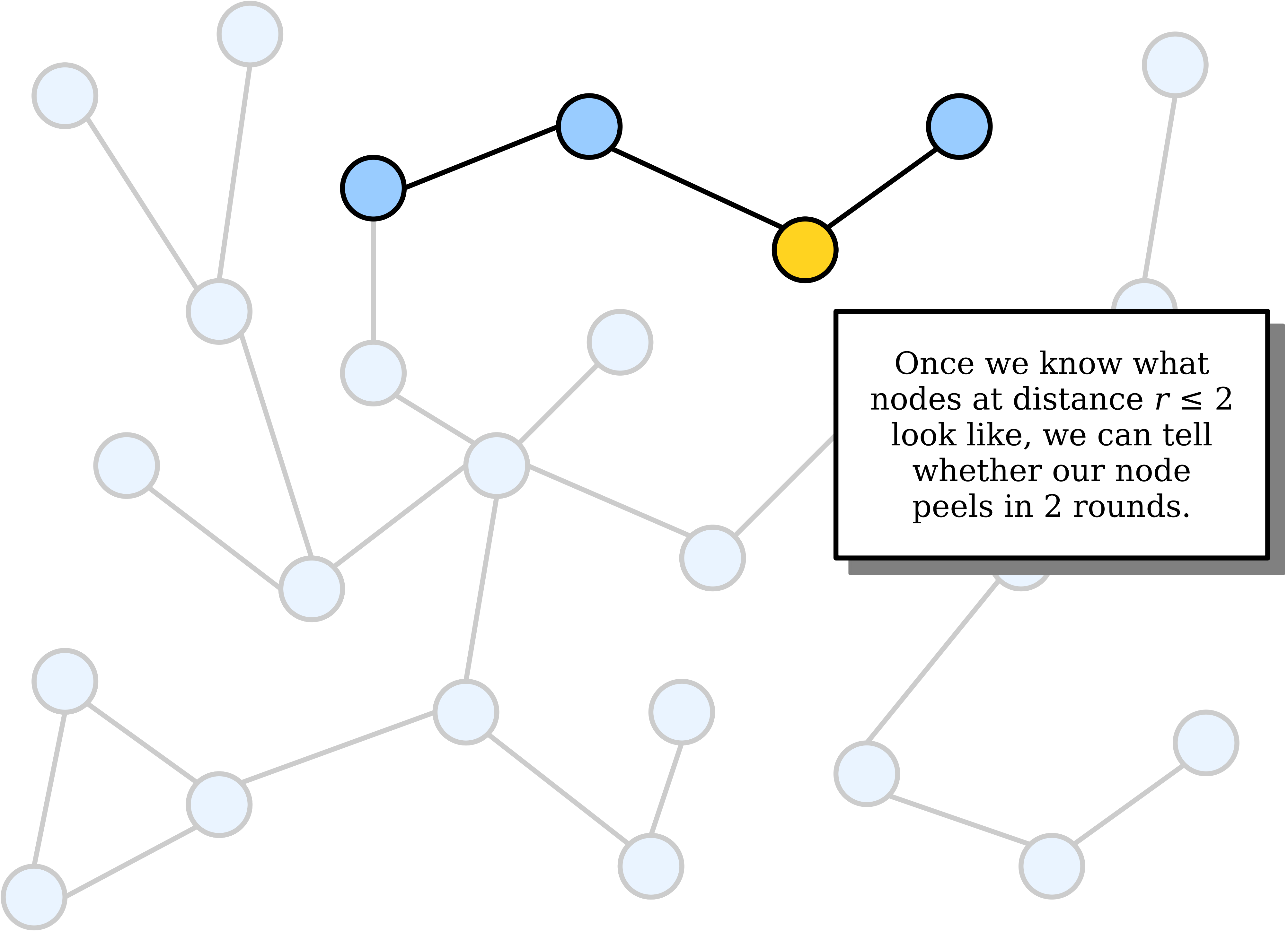




Without knowing anything about nodes at distance $r > 0$, we can't tell whether this node gets peeled in 1 round.



Without knowing anything about nodes at distance $r > 1$, we can’t tell whether this node gets peeled in 2 rounds.



Once we know what nodes at distance $r \leq 2$ look like, we can tell whether our node peels in 2 rounds.

Observation: To determine whether v is peeled within r rounds, we only need to focus on nodes and edges at distance r or less from v.

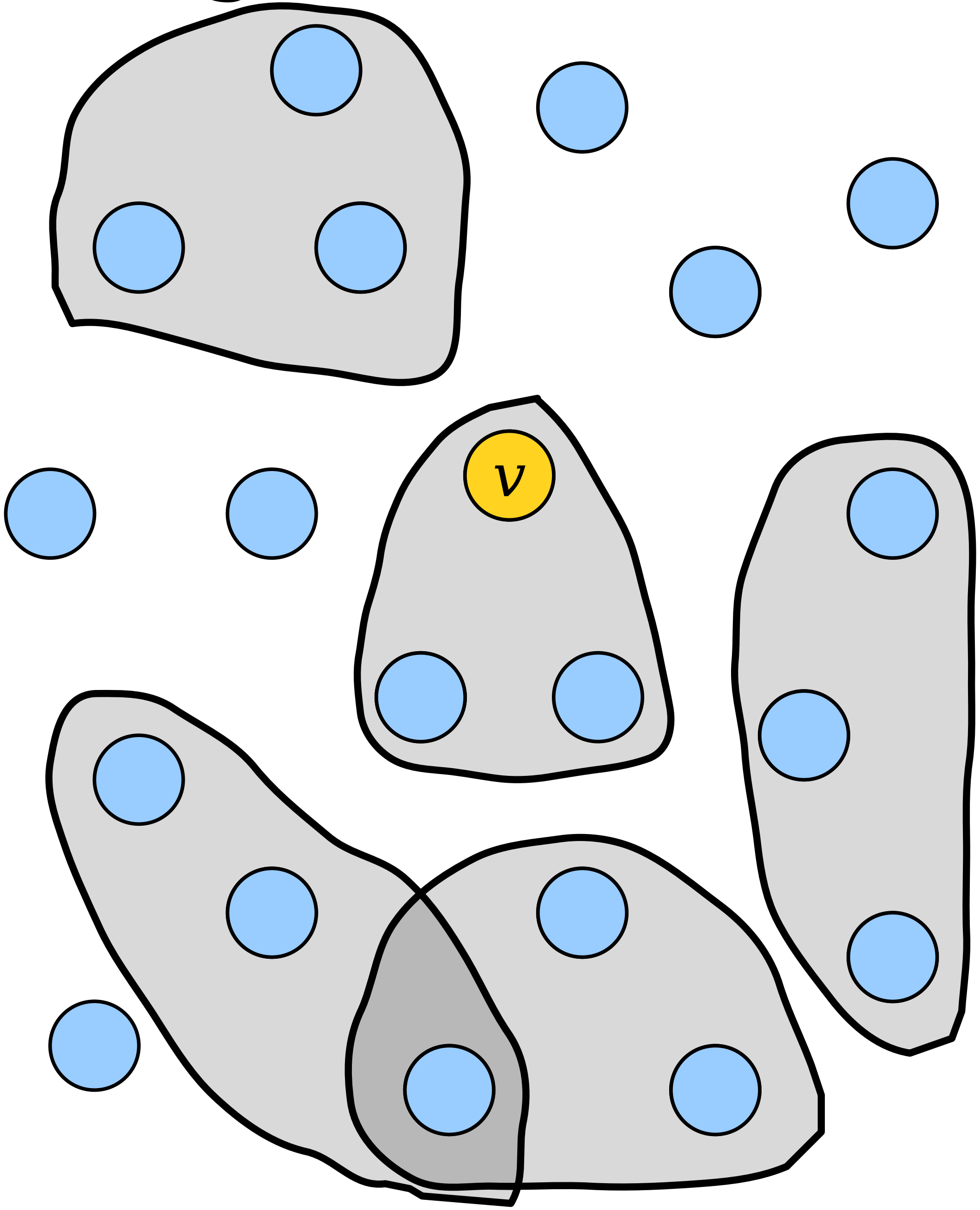
Algorithms Question: How would you find all the nodes at distance r or less from a starting node?

Exploring the Neighborhood

- Pick any node $v \in V$.
- Focus on any one hyperedge in the graph. What is the probability that it includes v?

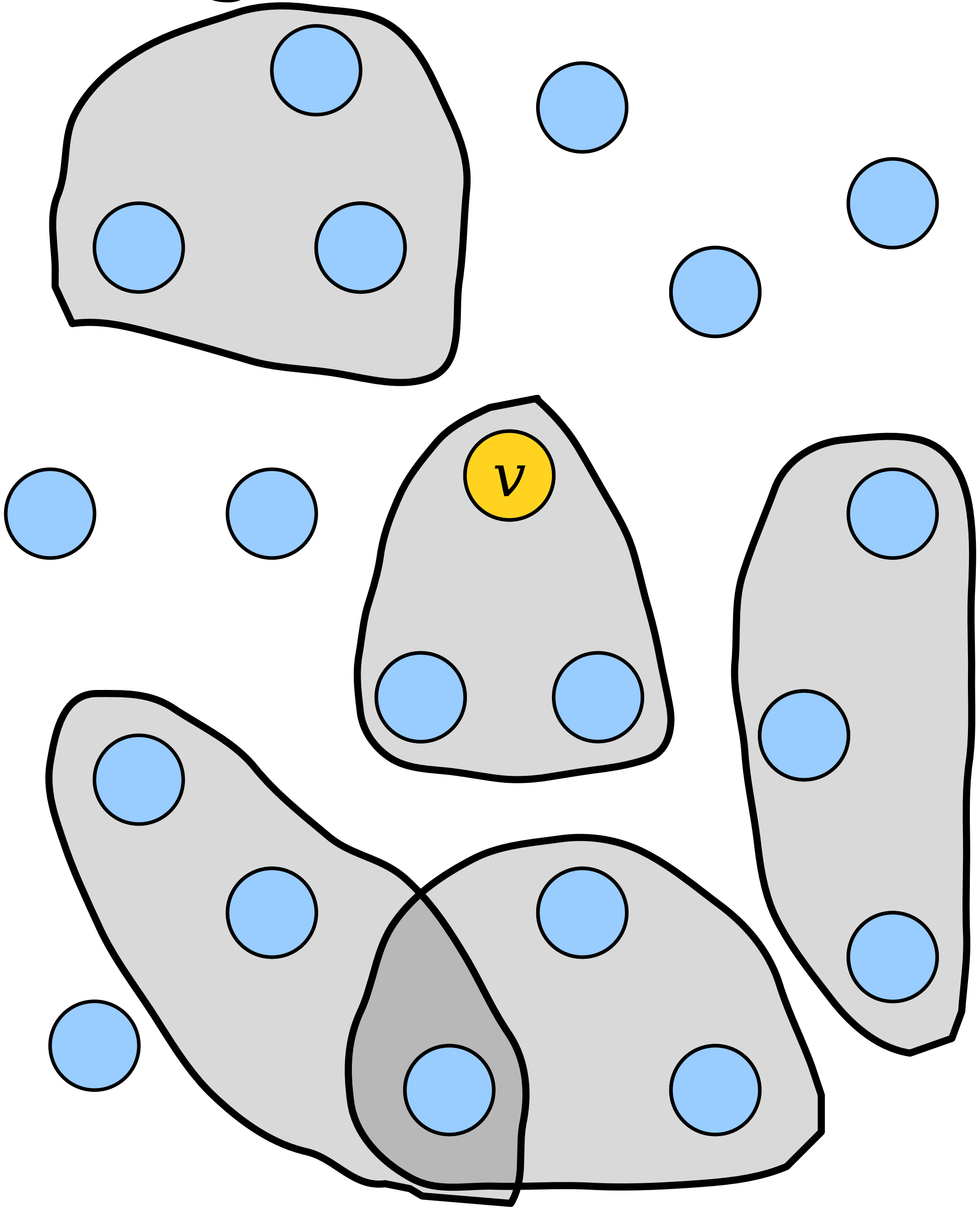
What is it, and why?
Answer at

https://cs166.stanford.edu/pollev



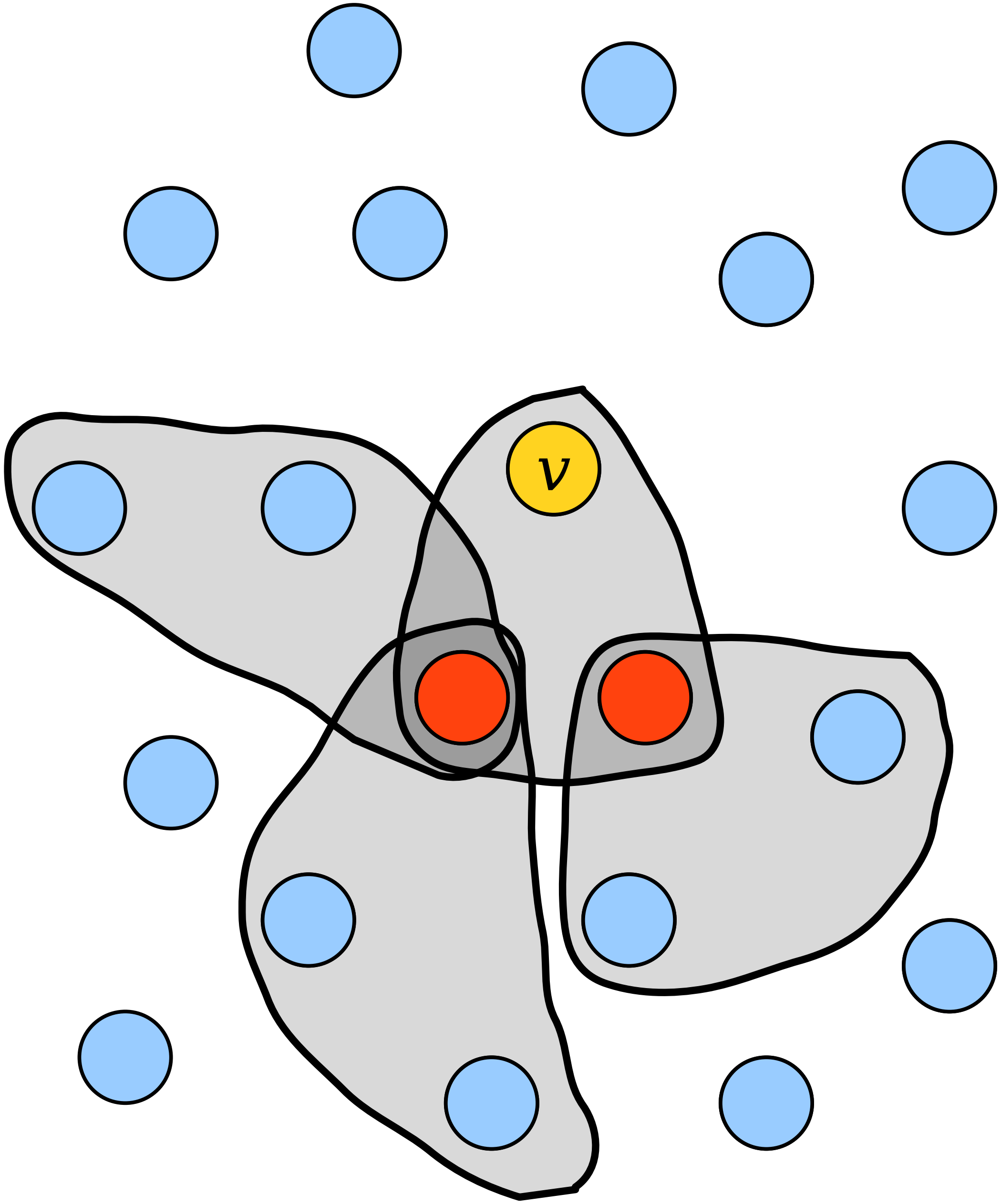
Exploring the Neighborhood

- Pick any node $v \in V$.
- Focus on any one hyperedge in the graph. What is the probability that it includes v?
  - ***Answer:*** ${}^{d}/_{V}$.
- There are $E = \alpha V$ hyperedges, so the degree of v is a Binom(αV, ${}^{d}/_{V}$) variable.
- We'll approximate this as a **Poisson(αd)** variable.



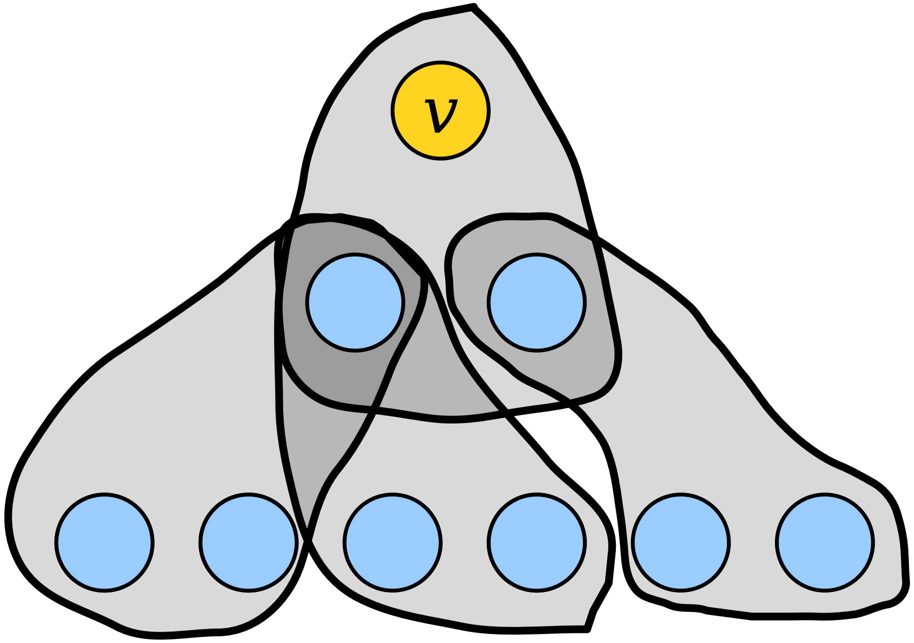
Exploring the Neighborhood

- Look at the nodes one hop from *v*.
- Think about where their outgoing edges go.
- ***Claim:*** As *V* tends to infinity, the likelihood that these edges include a cycle drops to zero. *(Why?)*
- Thus their outgoing edges lead away from these nodes along the lines of a tree.



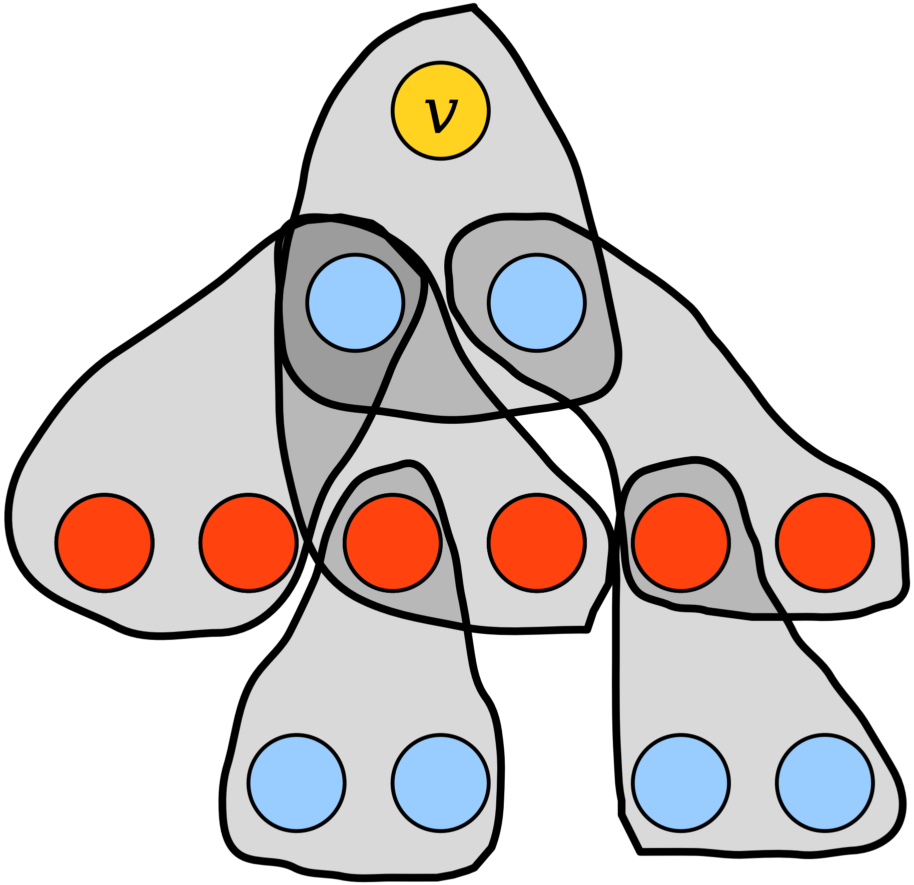
Exploring the Neighborhood

- Look at the nodes one hop from *v*.
- Think about where their outgoing edges go.
- ***Claim:*** As *V* tends to infinity, the likelihood that these edges include a cycle drops to zero. *(Why?)*
- Thus their outgoing edges lead away from these nodes along the lines of a tree.



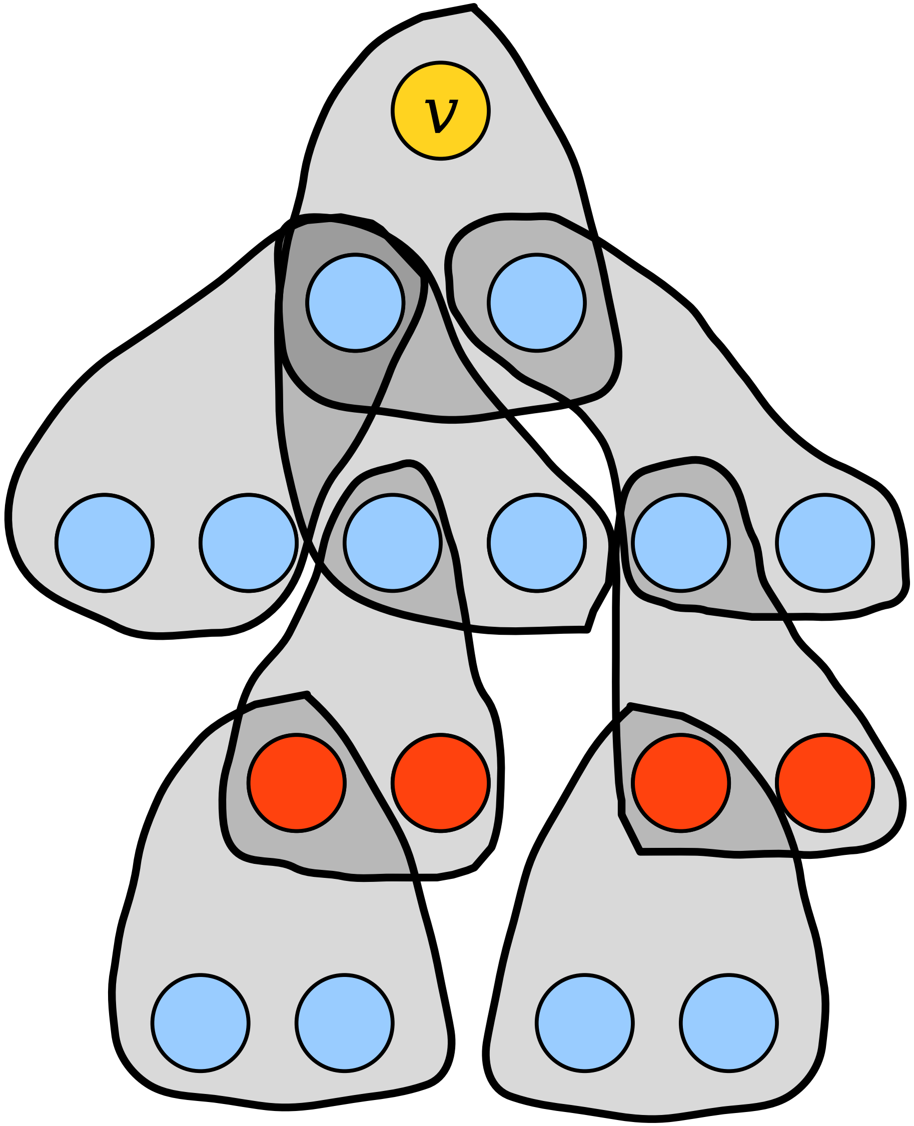
Exploring the Neighborhood

- Now look at the nodes two levels down.
- Excluding the edge to v, what's the distribution of the degrees of these nodes?
- ***Claim:*** As V approaches infinity, each node's degree is well-described as a **Poisson(αd)** variable.
    - *(What details are we ignoring? Why can we ignore them?)*
- Additionally, the chance that these edges close a cycle drops to zero as n goes to infinity.



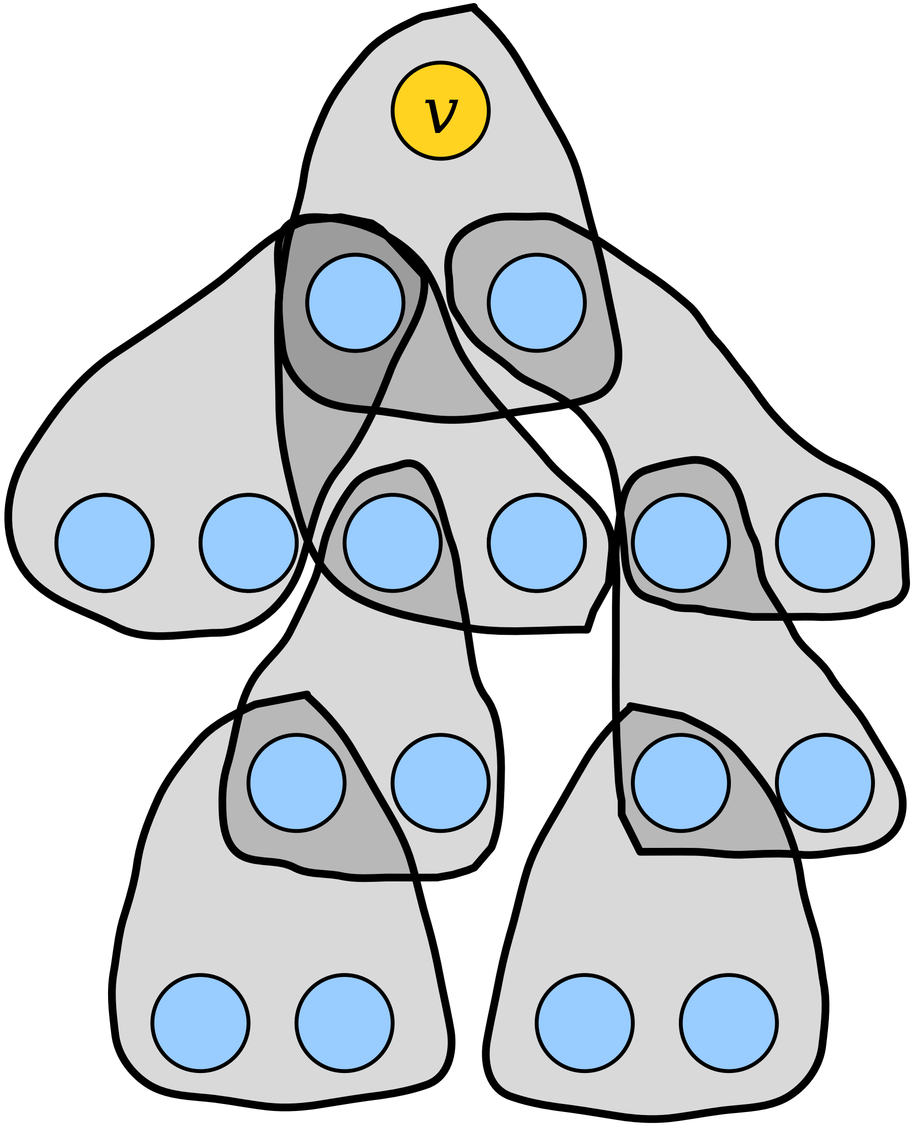
Exploring the Neighborhood

- Now look at these new nodes.
- ***Claim:*** As before, for sufficiently large *V*, the probability any outgoing edges from these nodes cause a cycle drops to zero.
- And these nodes' degrees, excluding their parent edges, are well-modeled as **Poisson(*αd*)** variables.



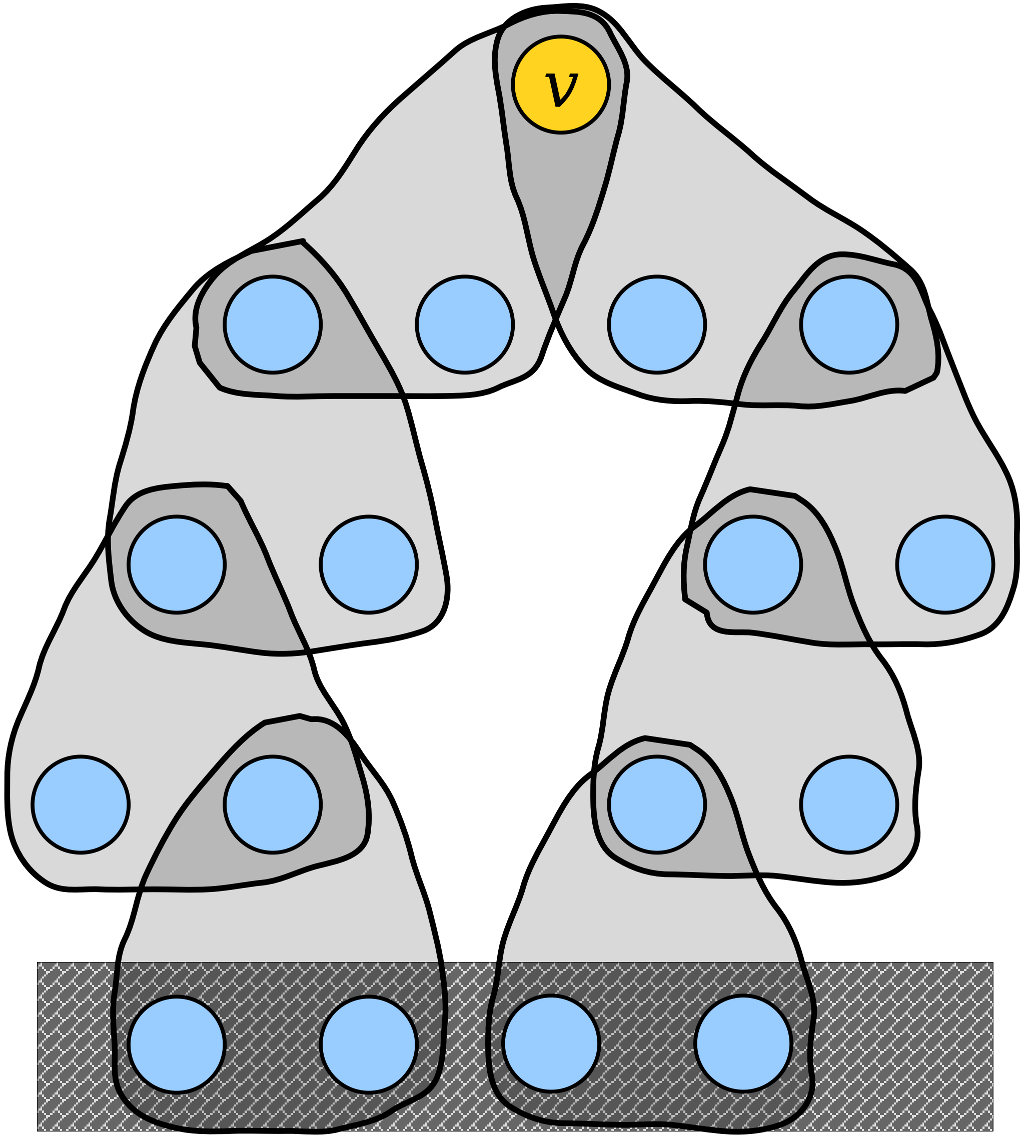
Exploring the Neighborhood

- ***General Claim:*** For any fixed constant r, and as V tends toward infinity:
  - The neighborhood of radius r around v is tree-shaped.
  - Each node's child edge count is distributed as Poisson(αd) variables.
- ***Question:*** For a fixed constant r, what is the probability that v gets peeled?



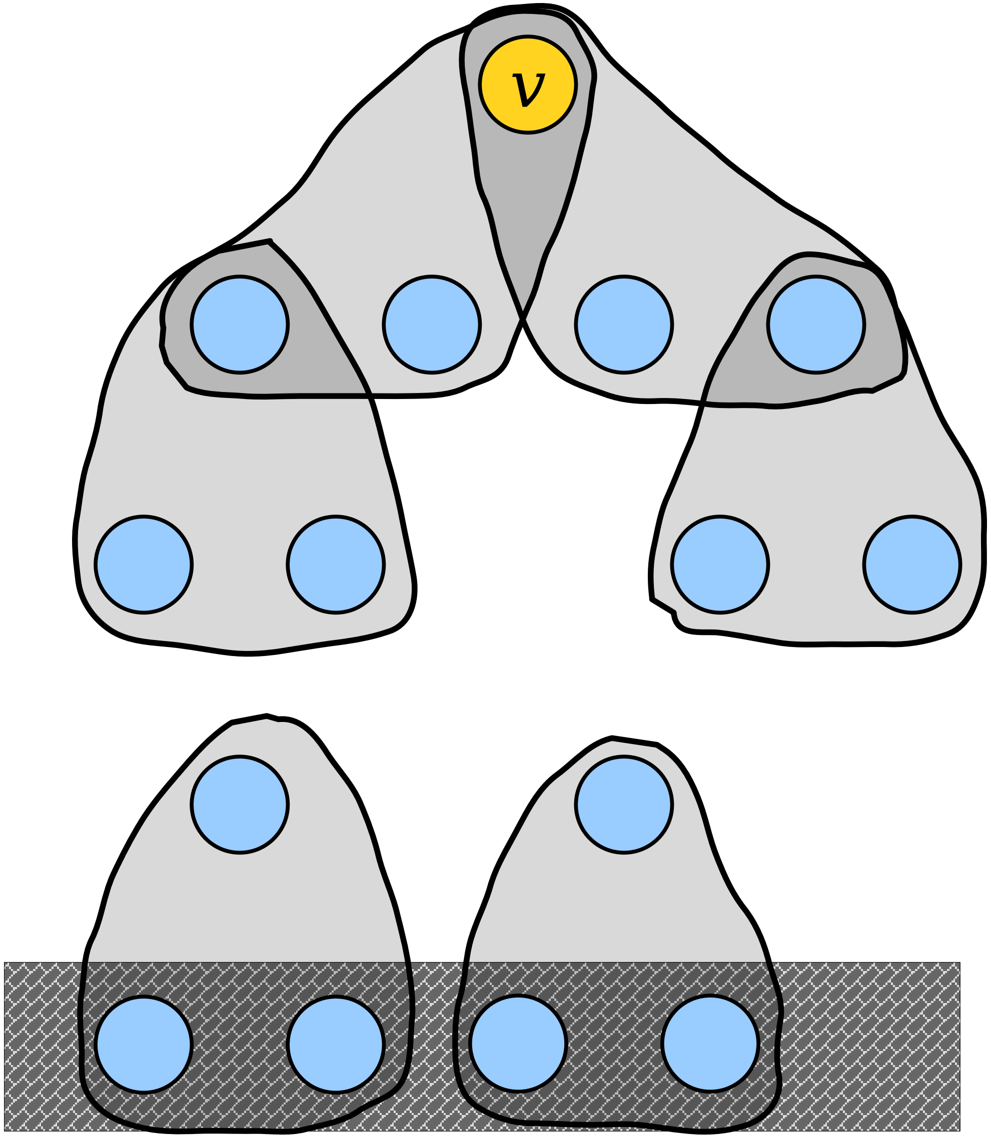
Bottom-Up Peeling

- To simplify the analysis, we'll look at a restricted form of peeling.
- We will assume all nodes at distance *r* from *v* are not peeled. (We don't know how they're linked up.)
- We'll then peel all peelable nodes at distance *r* – 1, then *r* – 2, then *r* – 3, …, then 1, and finally (if applicable) *v* itself.



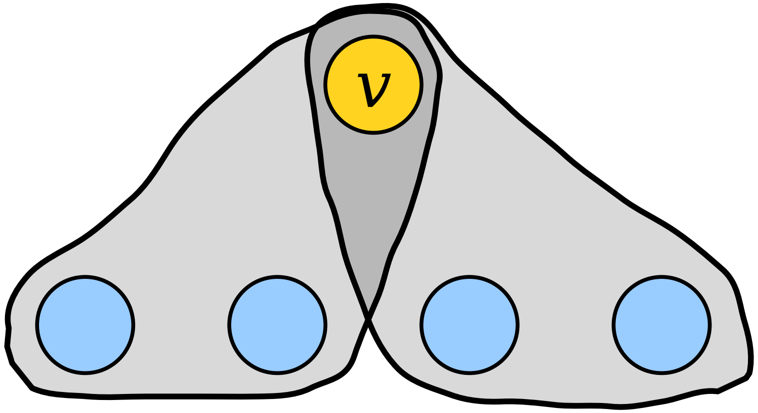
Bottom-Up Peeling

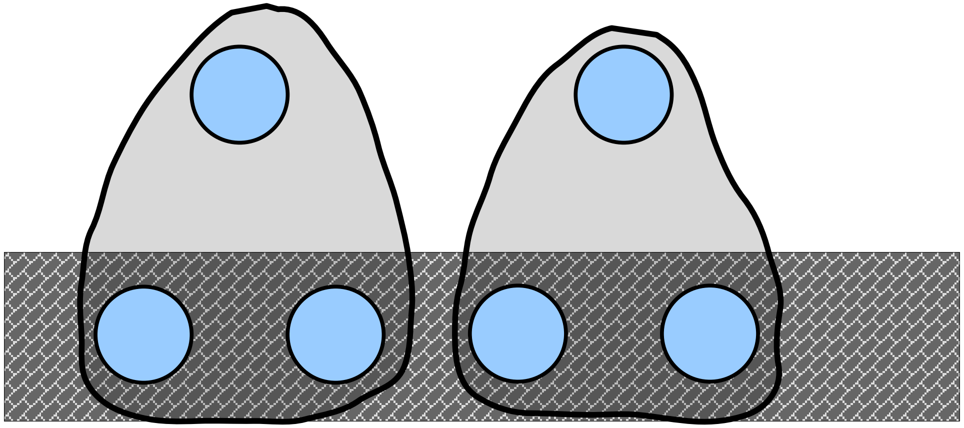
- To simplify the analysis, we'll look at a restricted form of peeling.
- We will assume all nodes at distance *r* from *v* are not peeled. (We don't know how they're linked up.)
- We'll then peel all peelable nodes at distance *r* – 1, then *r* – 2, then *r* – 3, …, then 1, and finally (if applicable) *v* itself.



Bottom-Up Peeling

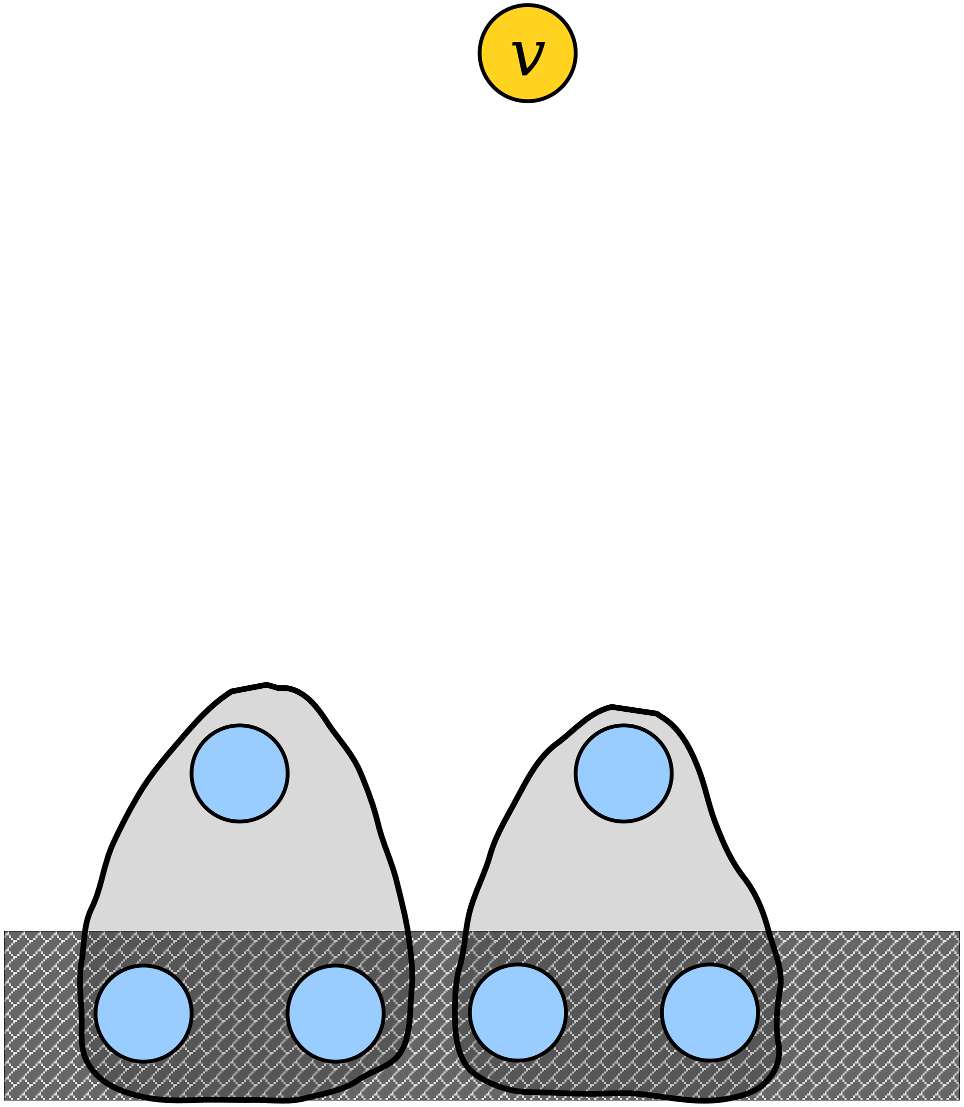
- To simplify the analysis, we'll look at a restricted form of peeling.
- We will assume all nodes at distance r from v are not peeled. (We don't know how they're linked up.)
- We'll then peel all peelable nodes at distance r – 1, then r – 2, then r – 3, …, then 1, and finally (if applicable) v itself.





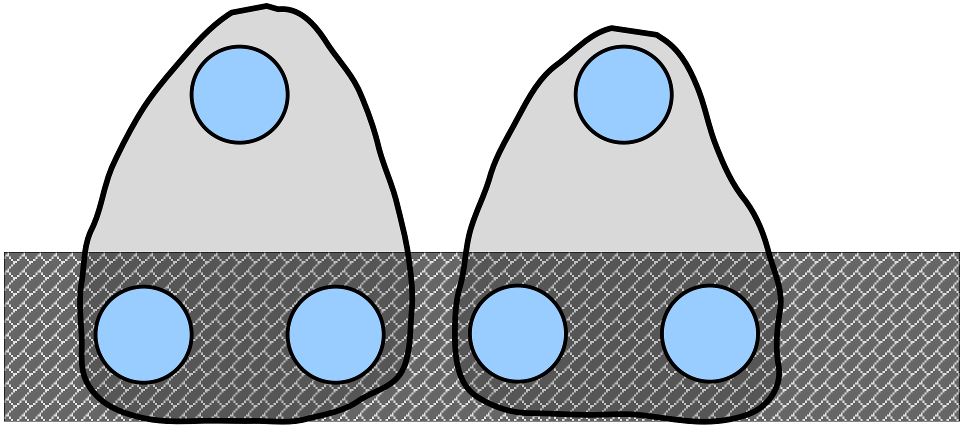
Bottom-Up Peeling

- To simplify the analysis, we'll look at a restricted form of peeling.
- We will assume all nodes at distance *r* from *v* are not peeled. (We don't know how they're linked up.)
- We'll then peel all peelable nodes at distance *r* – 1, then *r* – 2, then *r* – 3, …, then 1, and finally (if applicable) *v* itself.



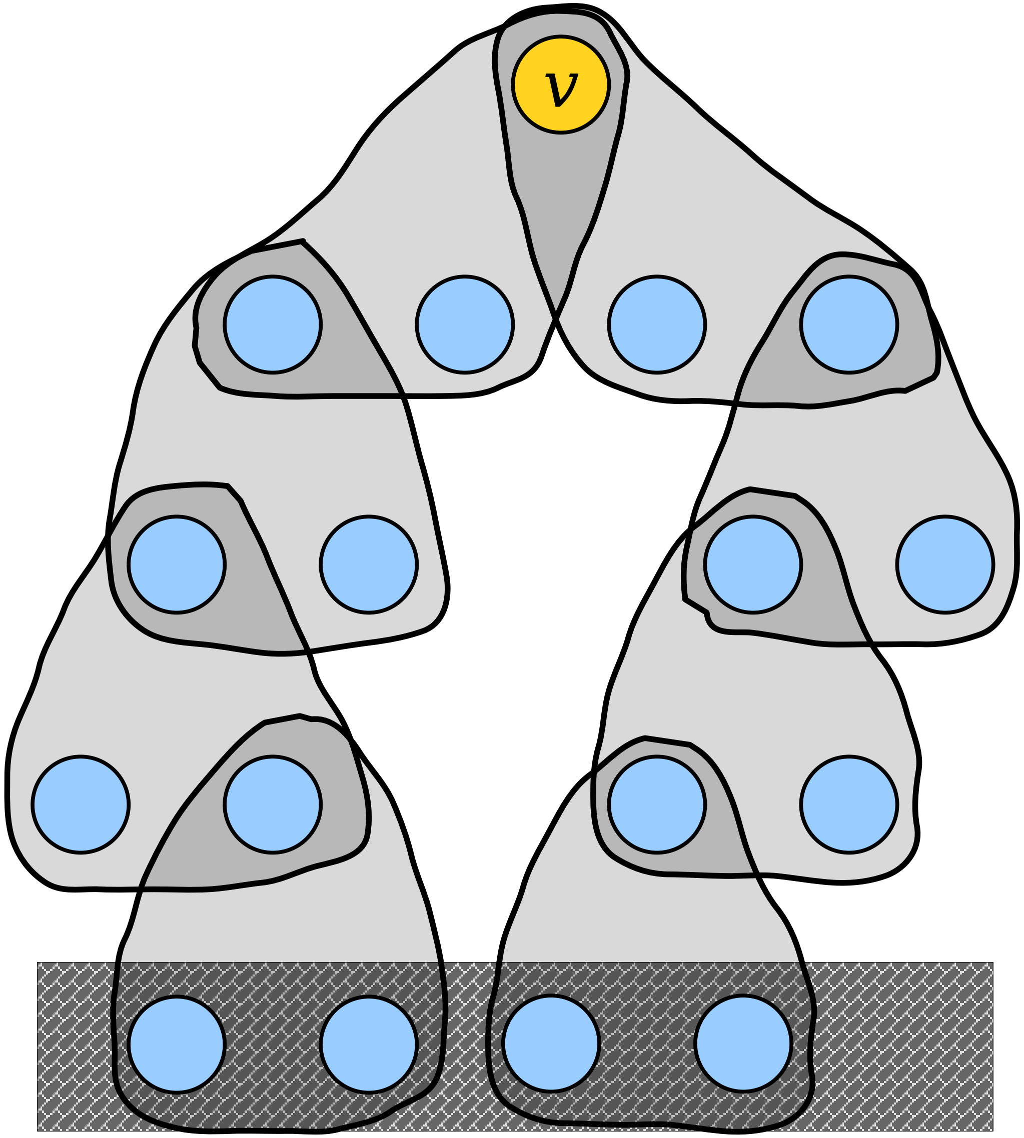
Bottom-Up Peeling

- To simplify the analysis, we’ll look at a restricted form of peeling.
- We will assume all nodes at distance *r* from *v* are not peeled. (We don’t know how they’re linked up.)
- We’ll then peel all peelable nodes at distance *r* – 1, then *r* – 2, then *r* – 3, …, then 1, and finally (if applicable) *v* itself.



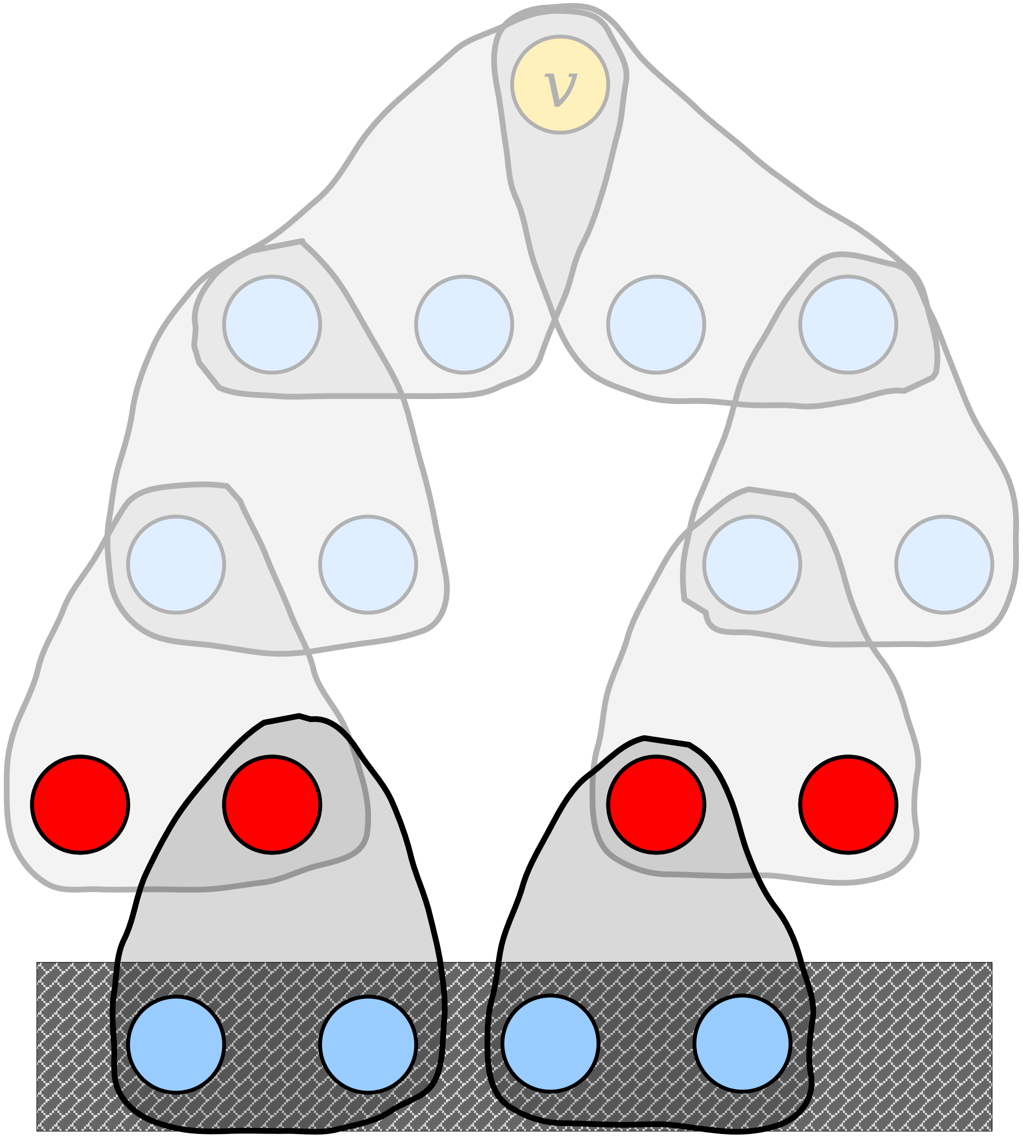
Bottom-Up Peeling

- To simplify the analysis, we'll look at a restricted form of peeling.
- We will assume all nodes at distance r from v are not peeled. (We don't know how they're linked up.)
- We'll then peel all peelable nodes at distance r – 1, then r – 2, then r – 3, …, then 1, and finally (if applicable) v itself.



Bottom-Up Peeling

- Pick a node x in the penultimate layer of the tree. What’s the probability x is peeled?
  - x has degree at least one (the edge back to its parent).
  - We never peel its children.
  - Thus it’s peeled only in the case where it has no children.
  - Its child edge count is distributed as a Poisson($d\alpha$) variable.
- So Pr[x peeled] = $e^{-d\alpha}$.

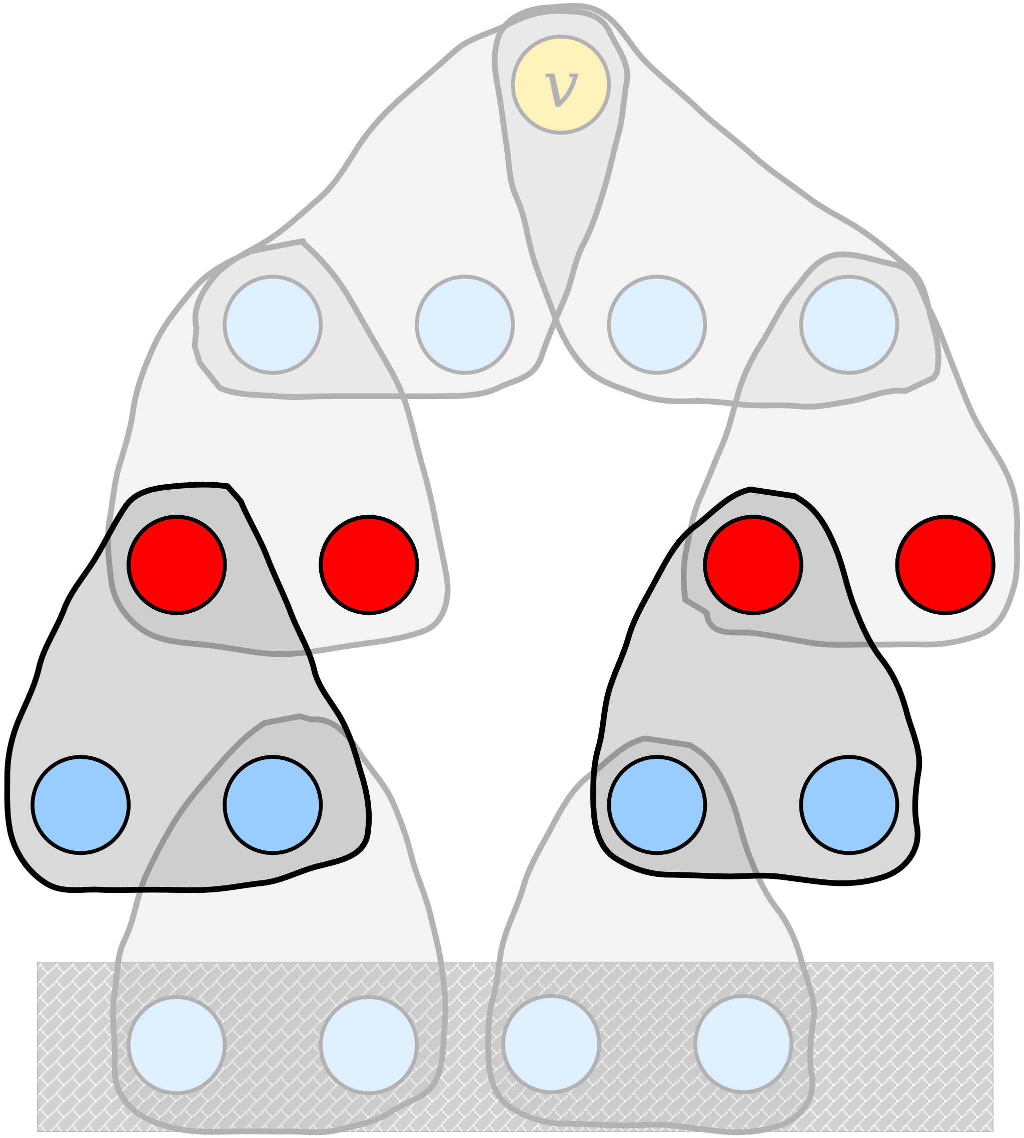


Bottom-Up Peeling

- Now look at a node *x* in the layer just above the previous one. What’s the probability that *x* is peeled?
- ***Claim:*** The probability is ***not*** the same as for the row below this.

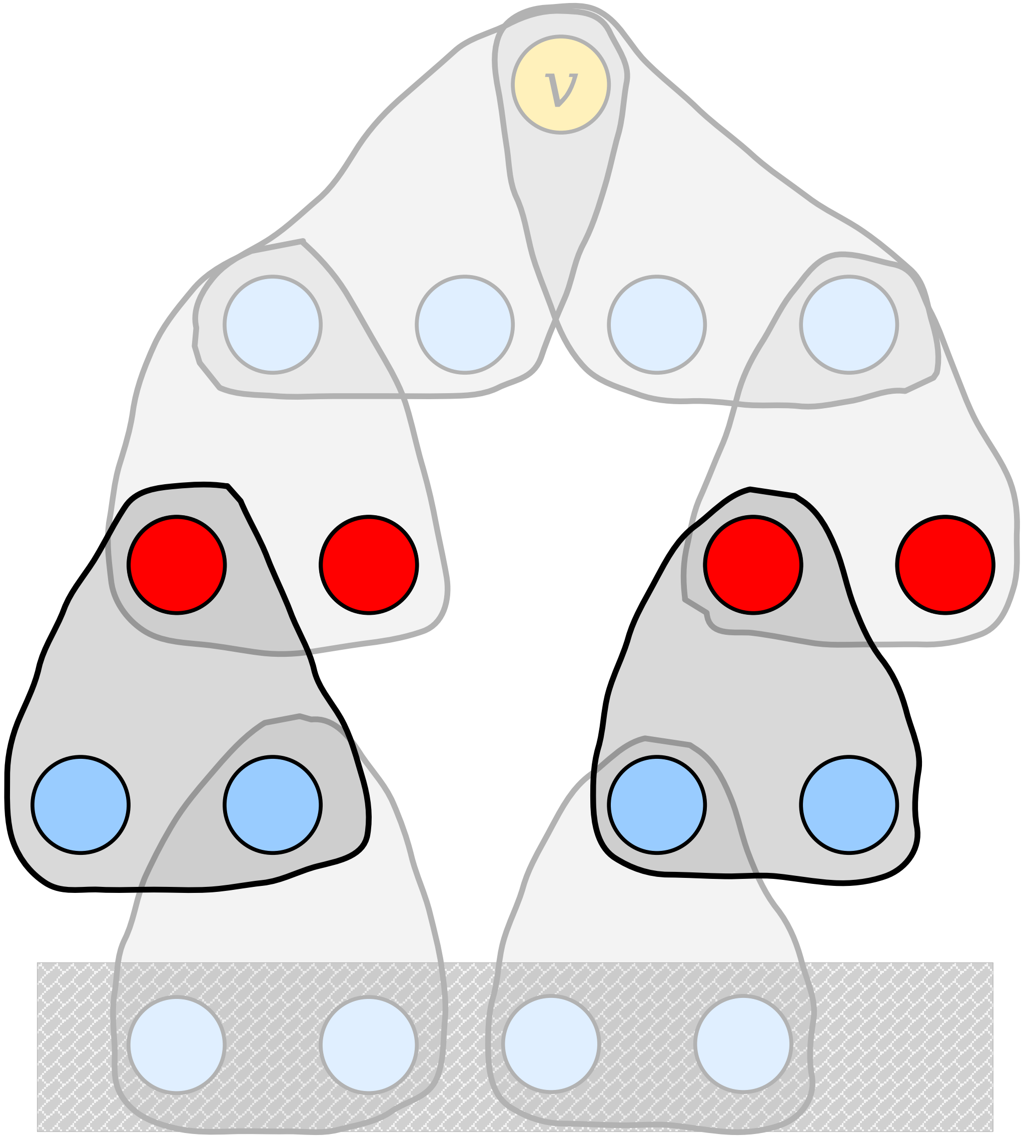
Why? Answer at

[***https://cs166.stanford.edu/pollev***](https://cs166.stanford.edu/pollev)



Bottom-Up Peeling

- Now look at a node x in the layer just above the previous one. What’s the probability that x is peeled?
- ***Claim:*** The probability is ***not*** the same as for the row below this.
  - Nodes in the next row are peeled only if they have no children.
  - Nodes in this row are peeled if they have no children ***after the first round of peeling***.



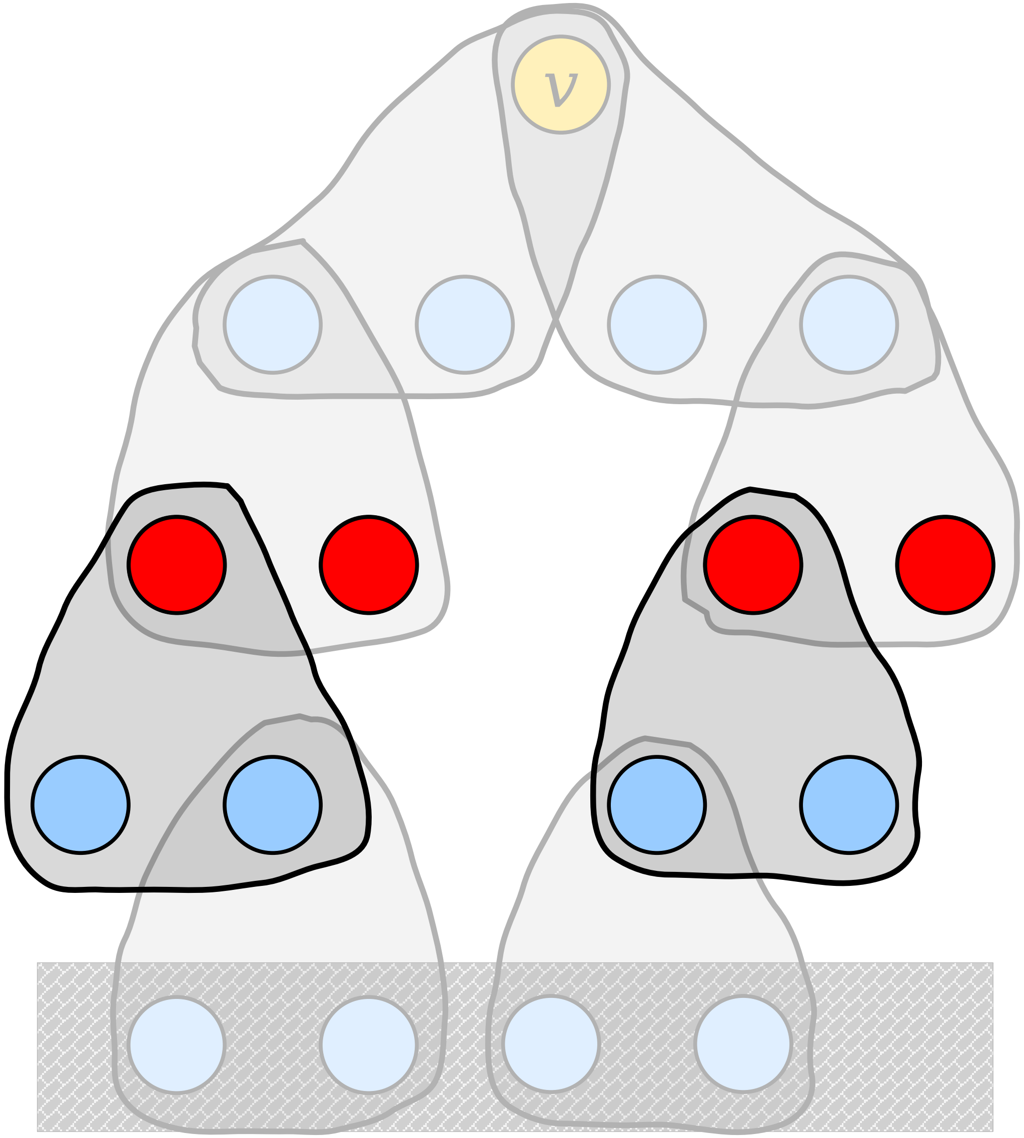
Bottom-Up Peeling

- ***Claim:*** After one peeling round, x has

  $$\text{Poisson}(\alpha d(1 - p_1)^{d-1})$$

  child edges, where p_1 is the probability any individual node in the next layer was peeled.
  - Originally, x's child count was a Poisson(αd) variable.
  - Each child hyperedge involves **d – 1** other nodes: d total nodes, minus one for x itself.
  - Each child edge remains iff none of those d – 1 nodes was peeled: probability $(1 - p_1)^{d-1}$.
- Pr[x is peeled] =

  $$\exp(-\alpha d(1 - p_1)^{d-1}).$$



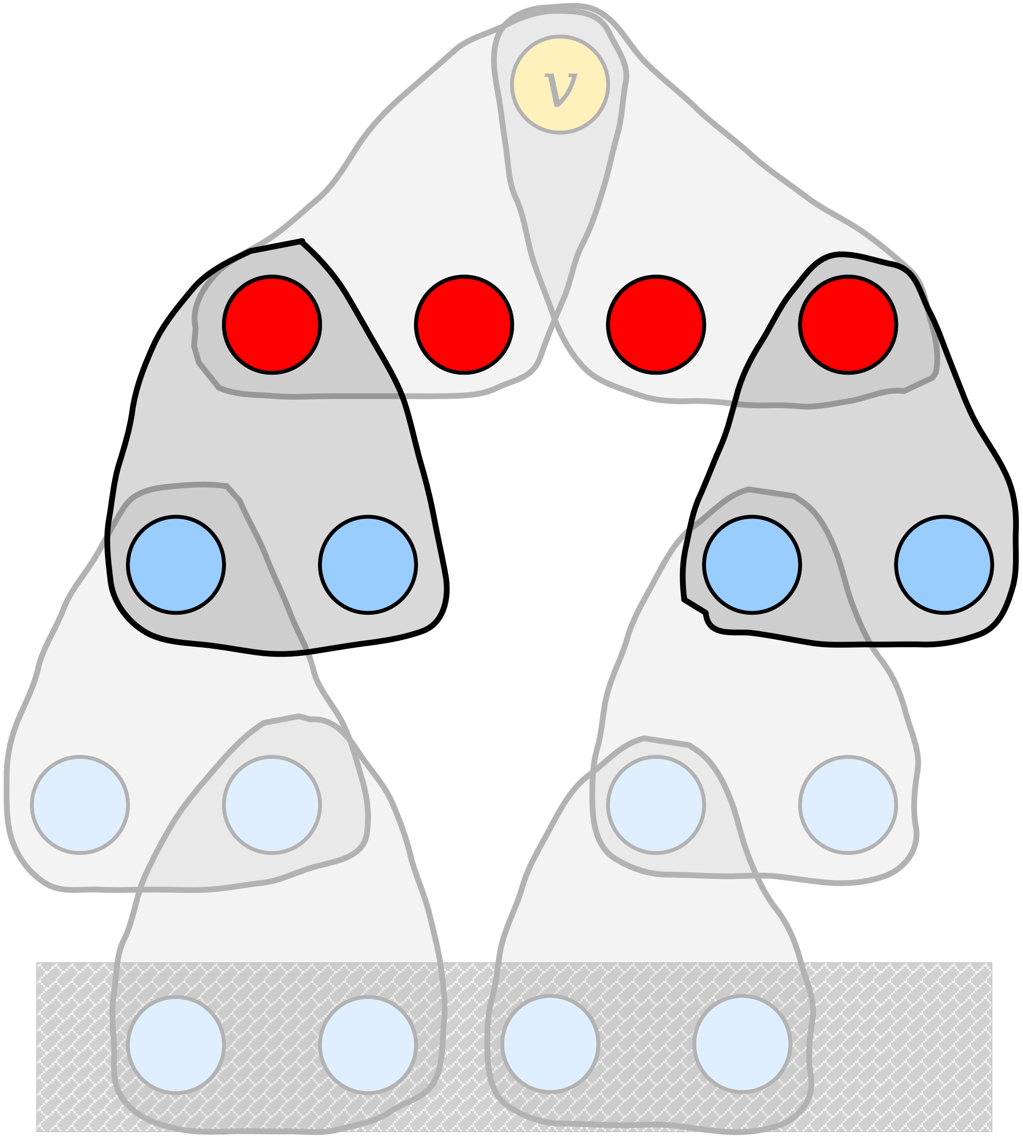
Bottom-Up Peeling

- Now look at a node x one level higher than before.
- Via analogous reasoning to the previous case, its unpeeled child edge count is a

  $$\textbf{Poisson}(\alpha d(1 - p_2)^{d-1})$$

  variable, where p_2 is the probability any individual node in the layer below it is peeled.
- Pr[x peeled] =

  $$\exp(-\alpha d(1 - p_2)^{d-1}).$$



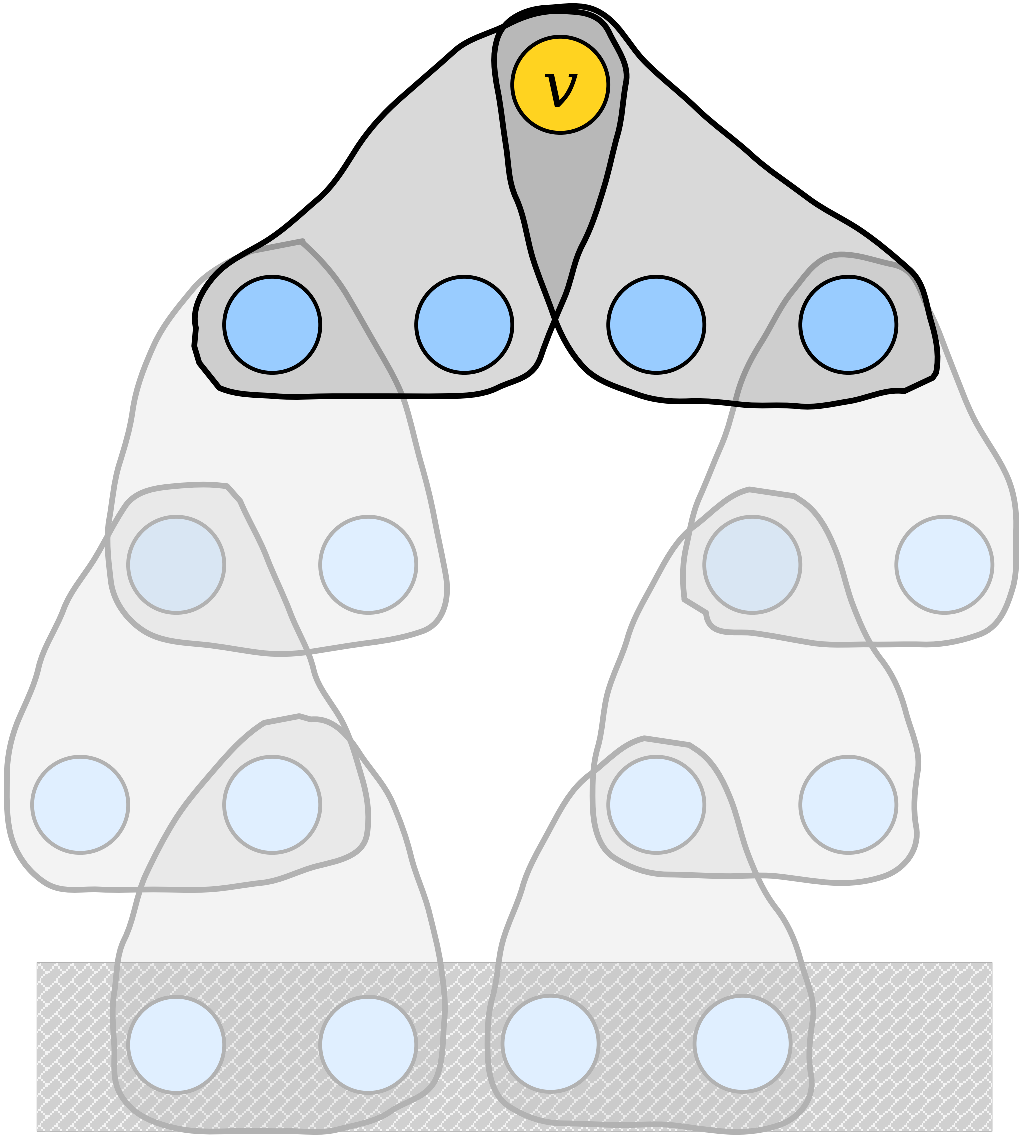
Bottom-Up Peeling

- Finally, look at *v* itself.
- Via analogous reasoning to the previous case, its unpeeled child edge count is a

  $$\mathbf{Poisson}(\alpha d(1 - p_3)^{d-1})$$

  variable, where p_3 is the probability any individual node in the layer below it is peeled.
- Unlike other nodes in the tree, this node has no parent. Thus it's peeled if it has zero *or one* child.
- ***Claim:*** The one-child case makes a negligible contribution to the probability. (We'll substantiate this later.)
- Pr[*v* peeled] ≈

  $$\exp(-\alpha d(1 - p_3)^{d-1})$$



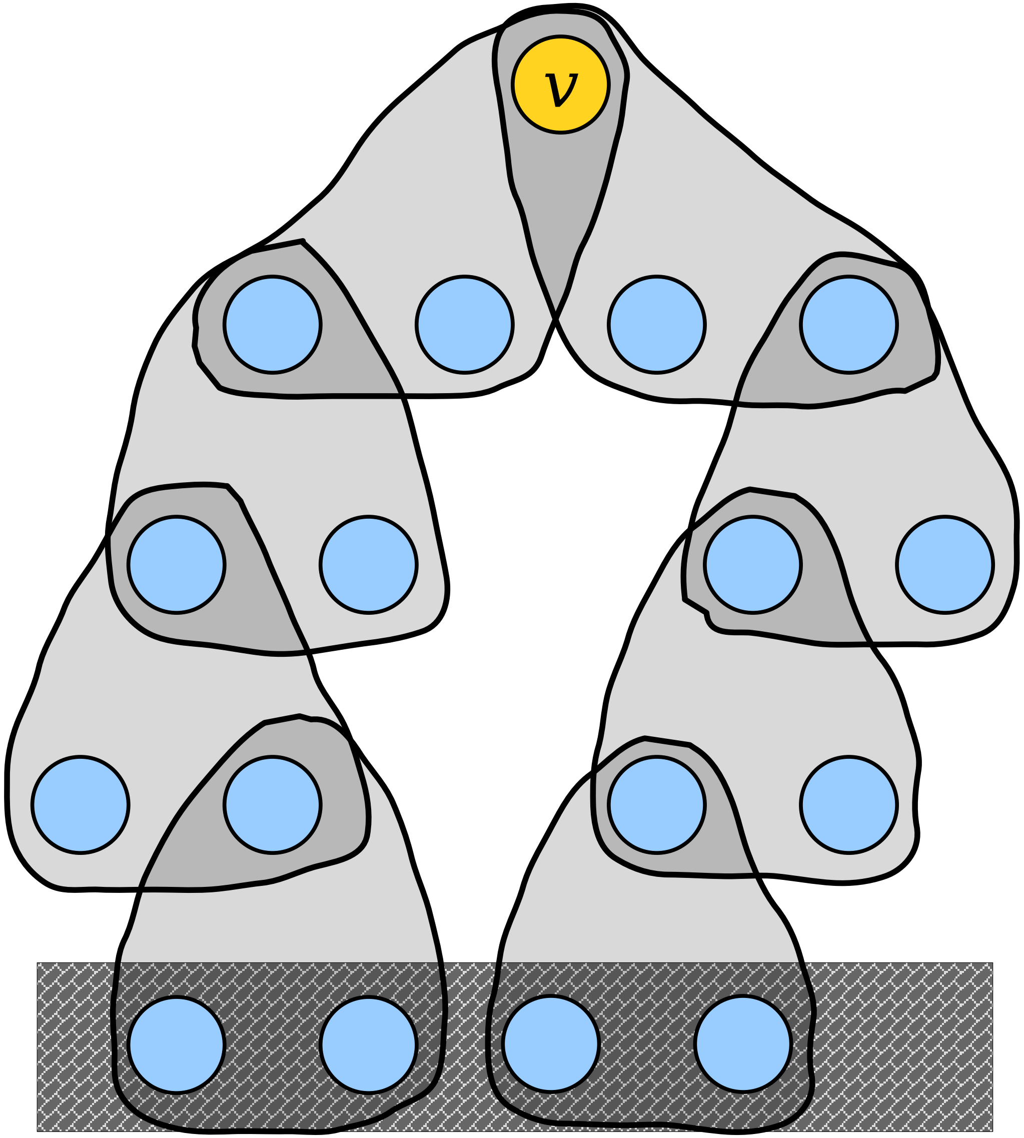
Bottom-Up Peeling

- More generally, let p_i be the probability that a node i layers from the bottom is peeled.
- We have the following recurrence relation:

$$p_0 = 0$$

$$p_{i+1} = e^{-\alpha d(1 - p_i)^{d-1}}$$

- How does this recurrence behave?
- Let's run some simulations!



Summarizing our Simulations

- For any fixed d, there seems to be some fixed threshold α_d* where
  - when $\alpha < \alpha_d$*, the probability of peeling the node rapidly approaches 1; and
  - when $\alpha > \alpha_d$*, the probability of peeling the node rapidly approaches a constant less than 1.
- This (plus some final technical details) explains why we're seeing the phase transition:
  - If $\alpha < \alpha_d$*, with high probability all nodes are peeled.
  - If $\alpha > \alpha_d$*, with high probability many nodes aren't.
- ***But why does this α_d* exist in the first place?***

Kleene’s Fixed-Point Theorem

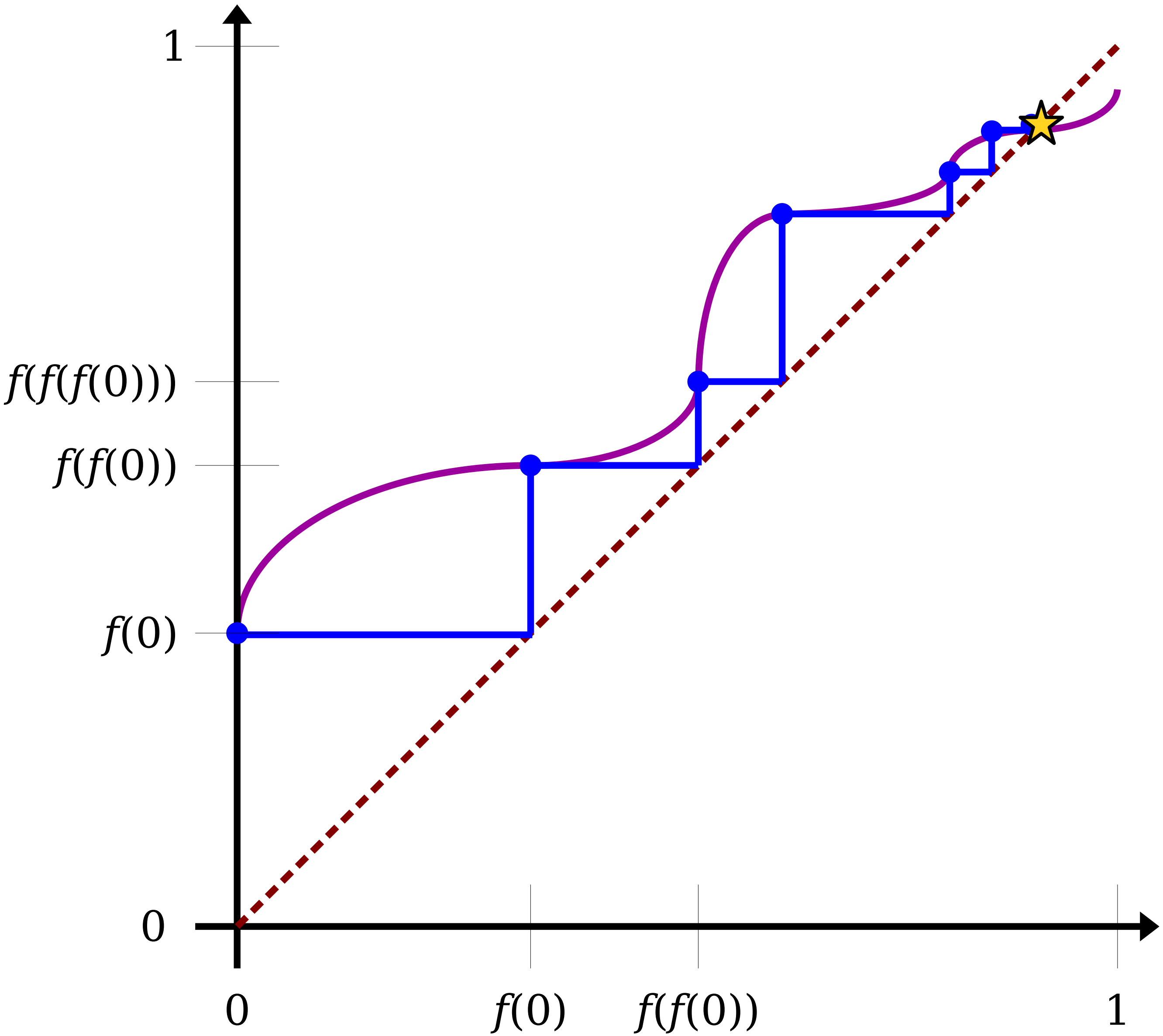
Iterated Functions

- Let $f(p) = exp(\text{-}\alpha d(1 – p)^{d-1})$.
- We’re looking at the limit of this sequence:

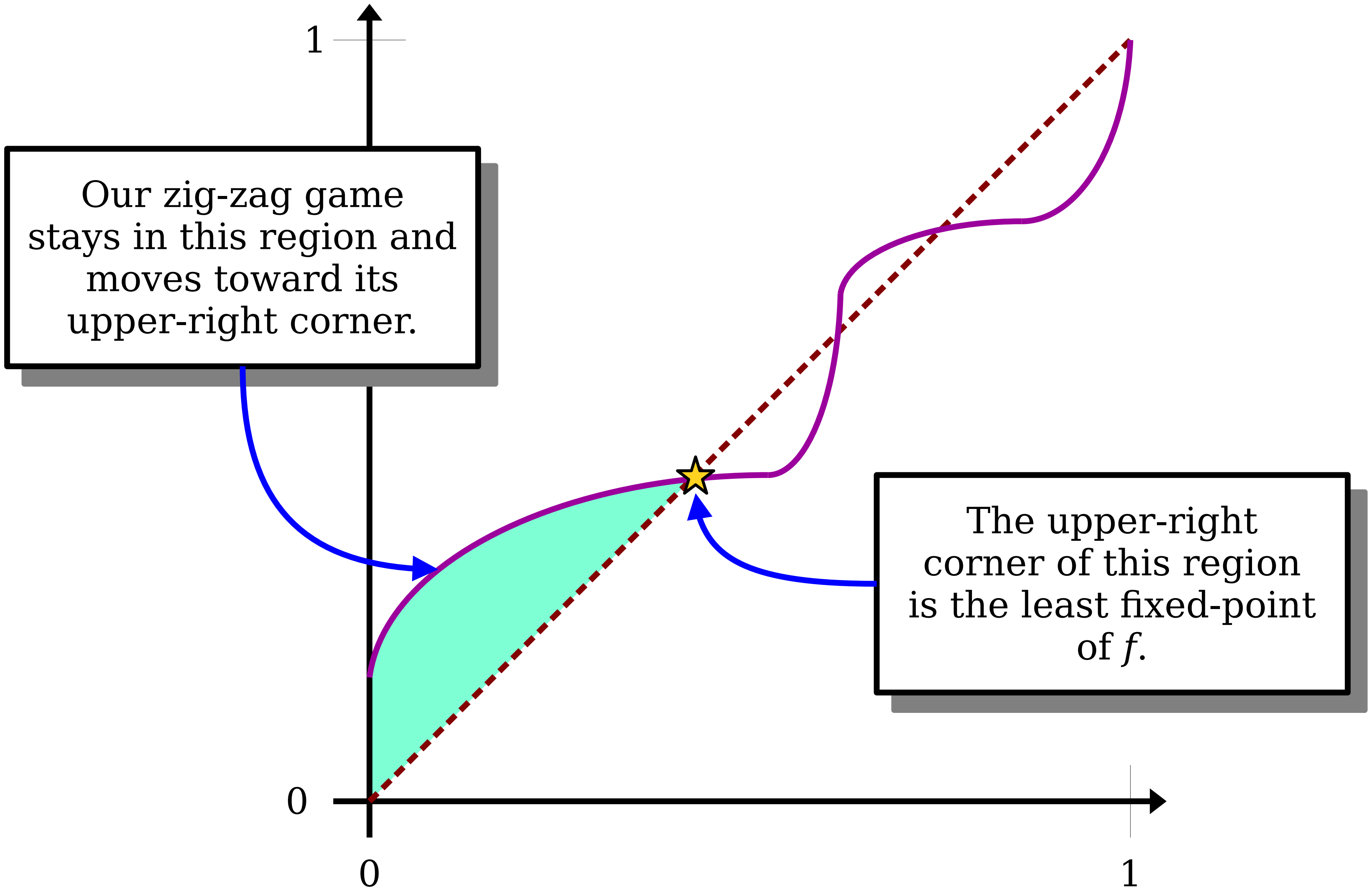
$$0, \quad f(0), \quad f(f(0)), \quad f(f(f(0))), \quad \ldots$$

- What happens if you iterate the same function lots and lots of times?

$$\boxed{\begin{aligned} p_0 &= 0 \\ p_{i+1} &= \boldsymbol{f(p_i)} \end{aligned}}$$



Let $f : [0, 1] \to [0, 1]$ be a continuous, monotone function. Then the sequence $0, f(0), f(f(0)), f(f(f(0))), \ldots$ converges to the smallest x for which $f(x) = x$.



Our zig-zag game stays in this region and moves toward its upper-right corner.

The upper-right corner of this region is the least fixed-point of *f*.

Let $f : [0, 1] \to [0, 1]$ be a continuous, monotone function. Then the sequence $0, f(0), f(f(0)), f(f(f(0))), \ldots$ converges to the smallest x for which $f(x) = x$.

Something ***very interesting*** happens
if we apply Kleene’s Fixed-Point Theorem
to our particular recurrence…

Putting it All Together

- The probability that a node gets peeled increases the more layers we peel back, according to the transform

$$p \mapsto exp(-\alpha d(1 - p)^{d-1}).$$

- This converges to the least fixed point of this map.
- This curve always has a fixed point of $p = 1$, but for each d there's an α_d* where the curve gains a second fixed point below 1 as it intersects the line $y = x$.
- When $\alpha < \alpha_d$*, we converge to the fixed point of 1 and peeling almost always succeeds.
- When $\alpha > \alpha_d$*, we converge to the lower fixed point and peeling has only a constant (non-1) probability of success.

Recap from Today

- IBTLs require $\approx 1.23k$ buckets to be communicated to solve set reconciliation when the two sets differ in k positions.
- Hypergraph peeling is a powerful technique for packing a lot of retrievable information into a small space.
- Hypergraph peeling has a phase transition due to the behavior of an iterated system of functions converging to a least fixed point.

Next Time

- ***Bloom Filters***
  - Saving valuable time by eliminating unnecessary work, most of the time.
- ***… Are Now Obsolete***
  - And the replacements are *amazing*.