Approximate Membership Queries

Outline for Today

- ***Approximate Membership Queries***
  - Storing sets… sorta.
- ***Bloom Filters***
  - The original approximate membership query structure – and still the most popular!
- ***Quotient Filtering***
  - Linear probing as space reduction.

Approximate Membership Queries

Exact Membership Queries

- The ***exact membership query*** problem is the following:

Maintain a set S in a way that supports queries of the form "is x ∈ S?"

- You now have a ton of tools available for solving this problem:

Red/black trees · Splay trees · Skiplists
B-trees · Linear probing · Cuckoo hashing
Iacono's structure · Sorted arrays
Chained hashing · Robin Hood hashing
Second-choice hashing

Exact Membership Queries

- Suppose you're in a memory-constrained environment where every bit of memory counts.
- Examples:
  - You're working on an embedded device with some maximum amount of working RAM.
  - You're working with large n (say, $n = 10^9$) on a modern machine.
  - You're building a consumer application like a web browser and don't want to hog all system resources.
- ***Question:*** How many bits of memory are needed to solve the exact membership query problem?

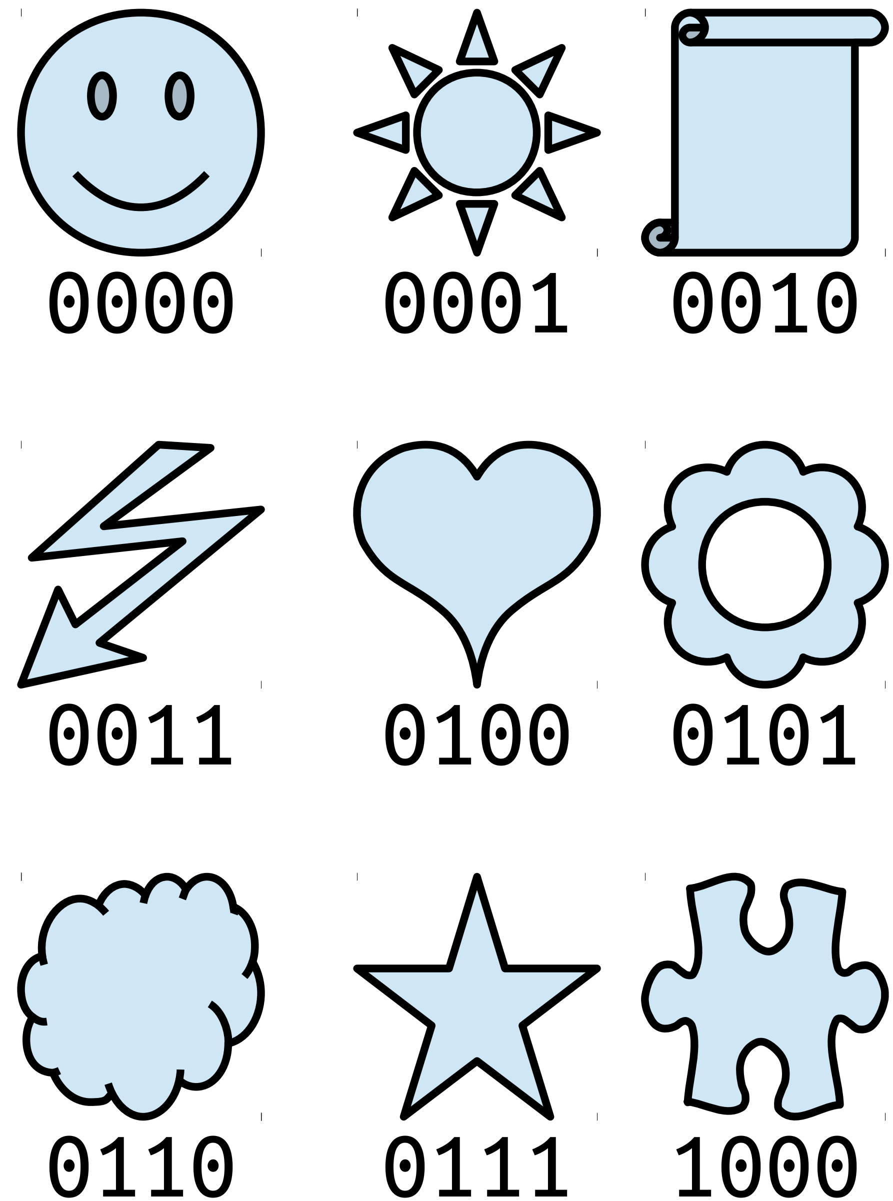
A Quick Detour

Goal: Design a simple data structure that can hold a single one of the objects shown to the right.

What is the minimum number of *bits* (not *words*) required for this data structure in the worst case?

We can get away with four bits by numbering each item and just storing the number.

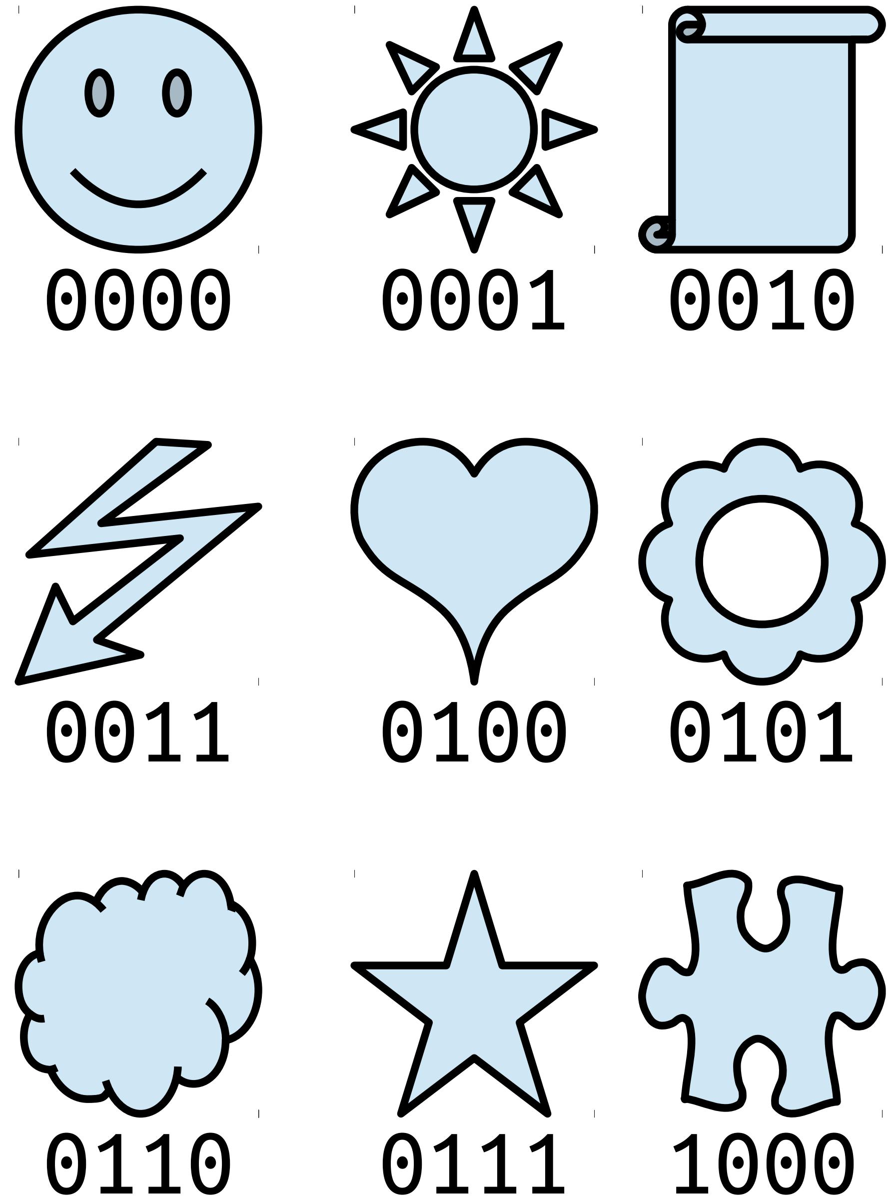
Question: Can we do better?



Goal: Design a simple data structure that can hold a single one of the objects shown to the right.

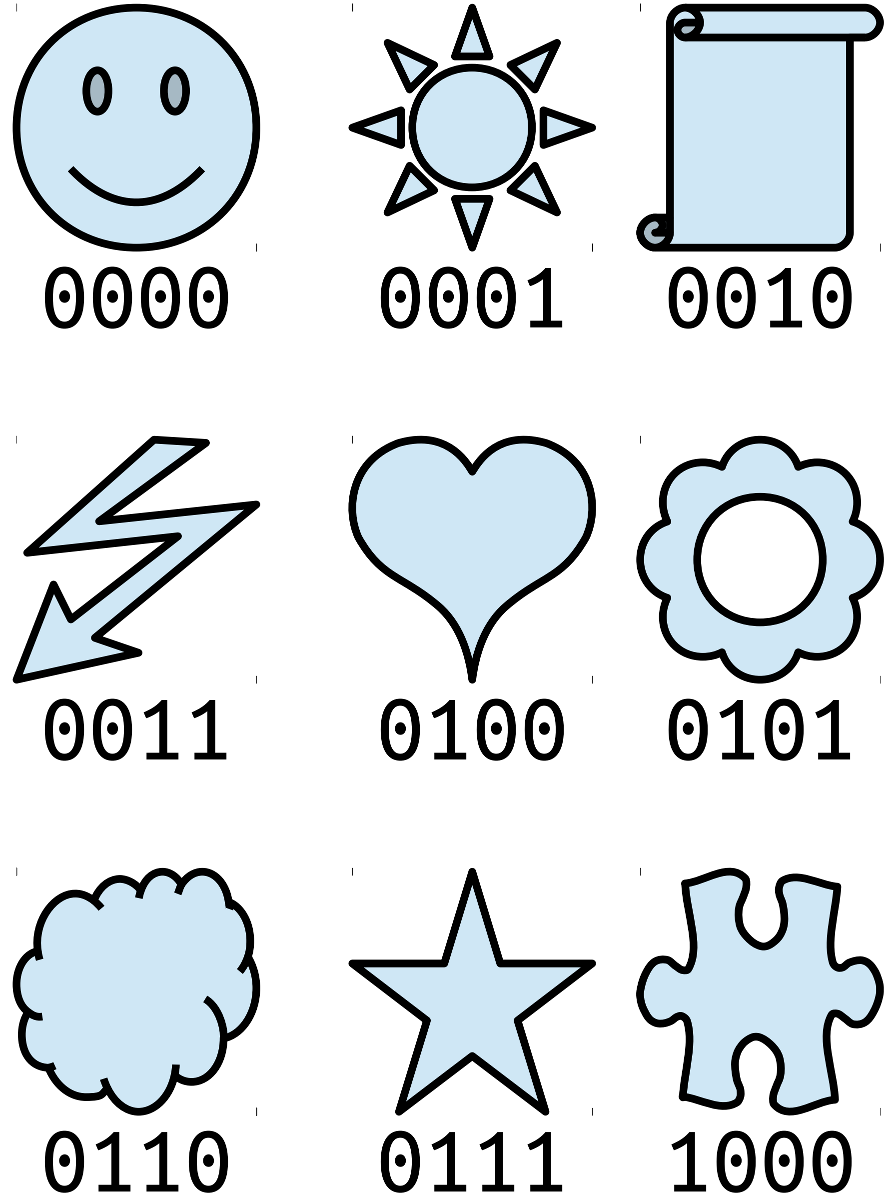
Claim: Every data structure for this problem must use at least four bits of memory in the worst case.

Proof: If we always use three or fewer bits, there are at most $2^3 = 8$ combinations of those bits, not enough to uniquely identify one of the nine different items.



Theorem: A data structure that stores one object out of a set of k possibilities must use at least $\lg k$ bits in the worst case.

Proof: Using fewer than $\lg k$ bits means there are fewer than $2^{\lg k} = k$ possible combinations of those bits, not enough to uniquely identify each item out of the set.



Question: How much memory is needed to solve the exact membership query problem?

Suppose we want to store a set $S \subseteq U$ of size n. How many bits of memory do we need?

Number of n-element subsets of universe U:

$$\binom{|U|}{n}$$

$$\begin{aligned}
& \lg \binom{|U|}{n} \\
= \quad & \lg \left(\frac{|U|\,!}{n\,!\,(|U|-n)\,!} \right) \\
\geq \quad & \lg \left(\frac{(|U|-n)^n}{n^n} \right) \\
= \quad & n \lg \left(\frac{|U|-n}{n} \right) \\
= \quad & n \lg \left(\frac{|U|}{n} - 1 \right) \\
\geq \quad & n \lg \left(\frac{|U|}{n} \right) - n \\
\geq \quad & n \lg |U| - n \lg n - n \\
= \quad & \boldsymbol{\Omega \left(n \lg |U| - n \lg n \right)}
\end{aligned}$$

Bitten by Bits

- Solving the exact membership query problem requires $\Omega(n \log |U| - n \log n)$ bits of memory in the worst case.
- Assuming $|U| \gg n$, we need $\mathbf{\Omega(n \log |U|)}$ bits to encode a solution to the exact membership query problem.
- If we're resource-constrained, this might be way too many bits for us to fit things in memory.
  - Think $n = 10^8$ and U is the set of all possible URLs or human genomes.
- Can we do better?

Approximate Membership Queries

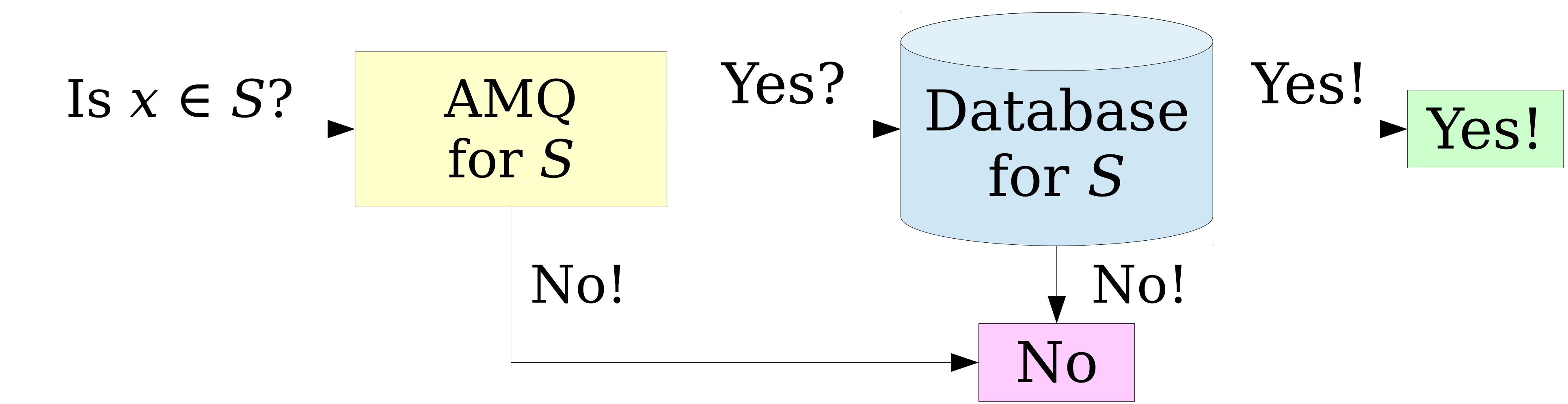
- The ***approximate membership query*** problem is the following:

  Maintain a set S in a way that gives approximate answers to queries of the form “is x ∈ S?”

- Questions we need to answer:
  - How do you give an “approximate” answer to the question “is $x \in S$?”
  - Does this relaxation let us save memory?
- We’ll address each of these in turn.

Our Model

- ***Goal:*** Design our data structures to allow for false positives but not false negatives.
- That is:
  - if $x \in S$, we always return true, but
  - if $x \notin S$, we have a small probability of returning true.
- This is often a good idea in practice.



Our Model

- Let's assume we have a tunable accuracy parameter $\varepsilon \in (0, 1)$.
- ***Goal:*** Design our data structure so that
  - if $x \in S$, we always return true;
  - if $x \notin S$, we return false with probability at least $1 - \varepsilon$; and
  - the amount of space we need depends only on n and ε, not on the size of the universe.
- Is this even possible?

Bloom Filters

Idea 1: Adapt the "hash to a bucket" idea of the count-min and count sketches.

As an example, let's have

$S = \{103, 137, 166, 271, 314\}$

`0000000000000000000000000000000000000000`

Number of bits: ***m***

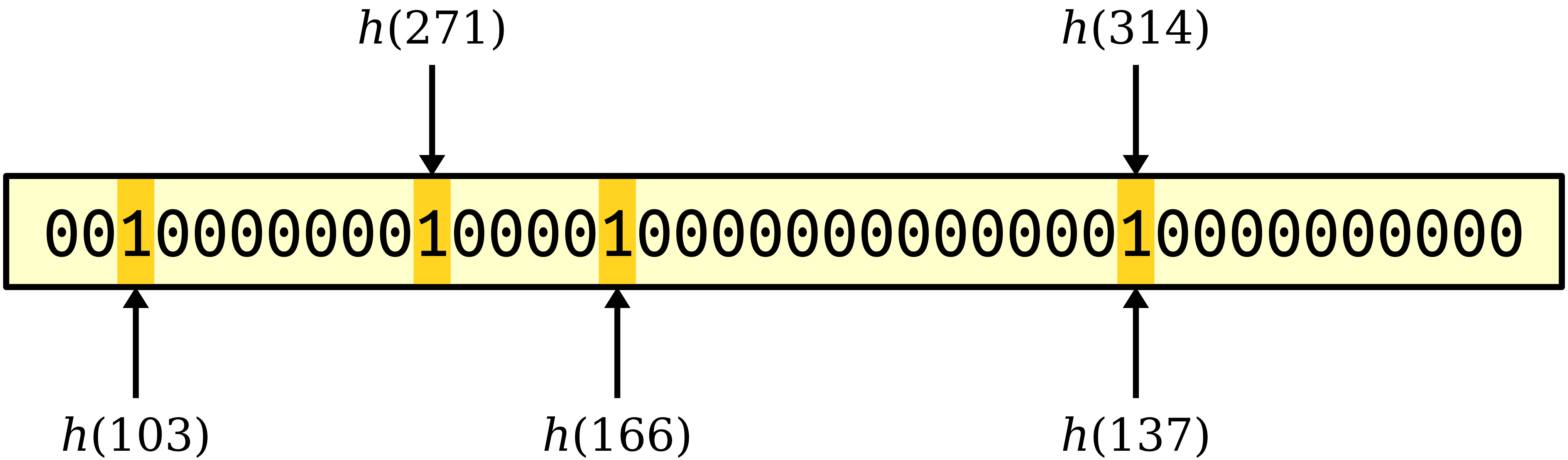
(We'll pick m later.)

How can we approximate a set in a small number of bits and with a low error rate?

Idea 1: Adapt the "hash to a bucket" idea of the count-min and count sketches.

As an example, let's have

$S = \{103, 137, 166, 271, 314\}$



How can we approximate a set in a small number of bits and with a low error rate?

Idea 1: Adapt the "hash to a bucket" idea of the count-min and count sketches.

As an example, let's have

$S = \{103, 137, 166, 271, 314\}$

`0010000000100001000000000000010000000000`

$h(103)$ — *True positive*

$h(161)$ — *True negative*

$h(261)$ — *False Positive*

How can we approximate a set in a small number of bits and with a low error rate?

Idea 1: Adapt the "hash to a bucket" idea of the count-min and count sketches.

Suppose we store a set of n elements in collection of m bits.

We want the probability of a false positive to be ε.

Question: How should we choose m based on n and ε?

`0010000000100001000000000000000001000000000`

Intuition: At most n of our m bits will be 1. We only have false positives if we see a 1. So we want $n / m = \varepsilon$, or $m = n \cdot \varepsilon^{-1}$.

Does the math match?

How can we approximate a set in a small number of bits and with a low error rate?

Idea 1: Adapt the "hash to a bucket" idea of the count-min and count sketches.

Suppose we look up some element $x \notin S$. What is the probability that we see a 1?

Probability that any one fixed element of S hashes here:

$$^1/_m.$$

Applying the union bound to all n elements gives a false positive rate of at most

$$^n/_m,$$

matching our intuition. So we need to pick $m = \boldsymbol{n \cdot \varepsilon^{-1}}$.

`0010000000001000010`

$h(x)$

False Positive

How can we approximate a set in a small number of bits and with a low error rate?

Idea 1: Adapt the "hash to a bucket" idea of the count-min and count sketches.

Cost of a query: **O(1)**.
Space usage: $\mathbf{O}(n \cdot \varepsilon^{-1})$.

Question: Can we do better?

`001000000010000100000000000000001000000000`

How can we approximate a set in a small number of bits and with a low error rate?

Idea 2: Adapt the "run in parallel" approach of the count-min sketch.

Make several copies of the previous data structure, each with a random hash function.

`0001000000000000000011000000010000000001`

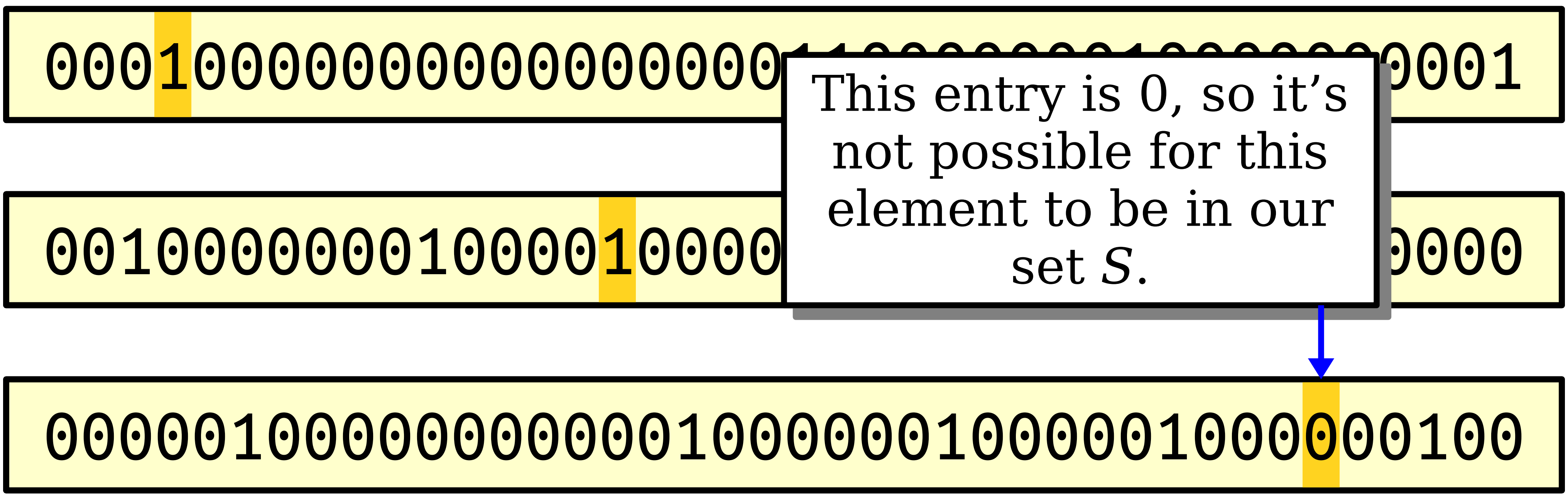
`0010000000100001000000000000010000000000`

`0000010000000000010000001000001000000100`

Can we get the same accuracy while using fewer bits?

Idea 2: Adapt the "run in parallel" approach of the count-min sketch.

Question: Each copy provides its own estimate. Which one should we pick?



Can we get the same accuracy while using fewer bits?

Idea 2: Adapt the “run in parallel” approach of the count-min sketch.

Question: Each copy provides its own estimate. Which one should we pick?

```
0001000000000000000011000000010000000001
0010000000100001000000000000010000000000
000001000000000001000000100000
```

We only say “yes” if all bits are 1’s.

Can we get the same accuracy while using fewer bits?

Idea 2: Adapt the "run in parallel" approach of the count-min sketch.

We have some fixed number of bits to use. How should we split them across these copies?

```
001000110010
001000110010
010100001000
000001110000
```

More copies means fewer bits per copy, making for a higher error rate.

Can we get the same accuracy while using fewer bits?

Idea 2: Adapt the "run in parallel" approach of the count-min sketch.

Approach: Use one giant array. Have all hash functions edit and read that array.

This is called a ***Bloom filter***, named after its inventor.

`01000010011000100100100000010001000100100`

Can we get the same accuracy while using fewer bits?

```
01000010011000100100100000010001000100100
```

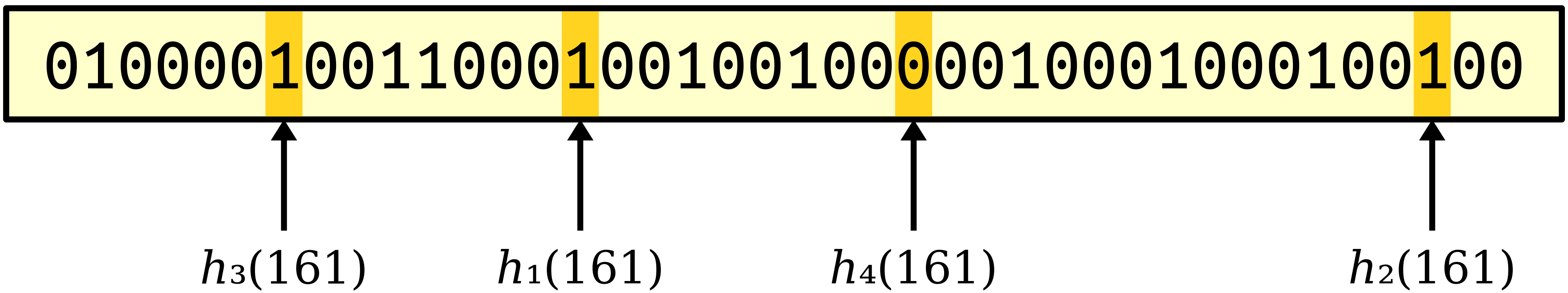
Number of bits: $\boldsymbol{m}$

(We will no longer set $m = n \cdot \varepsilon^{-1}$ because that analysis assumed we had one hash function. We'll pick m later.)

Can we get the same accuracy while using fewer bits?

Assume we use k hash functions, each of which is chosen independently of the others. We'll pick k later on.

(In this example, $k = 4$.)

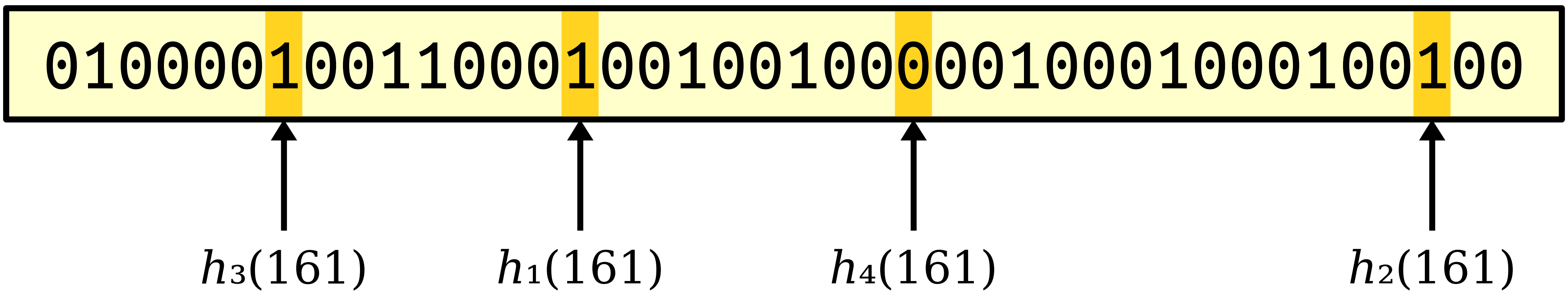


Can we get the same accuracy while using fewer bits?

create(S): Select k hash functions. Hash each element with all hash functions and set the indicated bits to 1.

query(x): Hash x with all k hash functions.

Return whether all the indicated bits are 1.



Can we get the same accuracy while using fewer bits?

We have two knobs to turn: the number of bits m, and the number of hash functions k.

Intuition: If m is too low, we'll get too many false positives. If m is too large, we'll use too much memory.

`01000010011000100100100000010001000100100`

Can we get the same accuracy while using fewer bits?

We have two knobs to turn: the number of bits m, and the number of hash functions k.

Idea: Set $m = \alpha n$ for some constant α that we'll pick later on. (Use a constant number of bits per element.)

`0100001001100010010010000001000100100100`

Can we get the same accuracy while using fewer bits?

We have two knobs to turn: the number of bits m, and the number of hash functions k.

Intuition: If $m = \alpha n$ and k is either too low *or* too high, we'll get too many false positives.

Question: How do we tune k, the number of hash functions?

`0100001001100010010010000001000100100100`

Can we get the same accuracy while using fewer bits?

Question: In what circumstance do we get a false positive?

Answer: Each of the element's bits are set, but the element isn't in the set S.

Question: What is the probability that this happens?

`0100001001100010010010000001000100010 0100`

How do we quantify our error rate?

Question 1: What is the probability that any particular bit is set?

`0011010100 0101000`

Focus on a bit at index i.

Pick some $x \in S$ and hash function h.

What's the probability that $h(x) \neq i$? (Assume truly random hash functions.)

Answer: $1 - {}^1\!/_m$.

What's the probability that, across all n elements and k hash functions, bit i isn't set?

Answer: $(1 - {}^1\!/_m)^{kn}$.

How do we quantify our error rate?

Question 1: What is the probability that any particular bit is set?

`0011010100 0101000`

Useful fact: $(1 - {}^{1}/_{p})^{p} \approx e^{-1}$.

Probability that bit i is unset after inserting n elements:

$$
\begin{aligned}
& \left(1 - \frac{1}{m}\right)^{kn} \\
= \quad & \left(\left(1 - \frac{1}{m}\right)^{m}\right)^{\frac{kn}{m}} \\
\approx \quad & e^{-\frac{kn}{m}} \\
= \quad & e^{-k\,\alpha^{-1}}
\end{aligned}
$$

How do we quantify our error rate?

Question 2: What is the probability of a false positive?

`0011010100010101000`

This value isn't exactly correct because certain bits being 1 decrease the probability that other bits are 1. With a more advanced analysis we can show that this is very close to the true value.

Probability that a fixed bit is 1 after n elements have been added:

$$\approx 1 - e^{-k\,\alpha^{-1}}$$

False positive probability is approximately

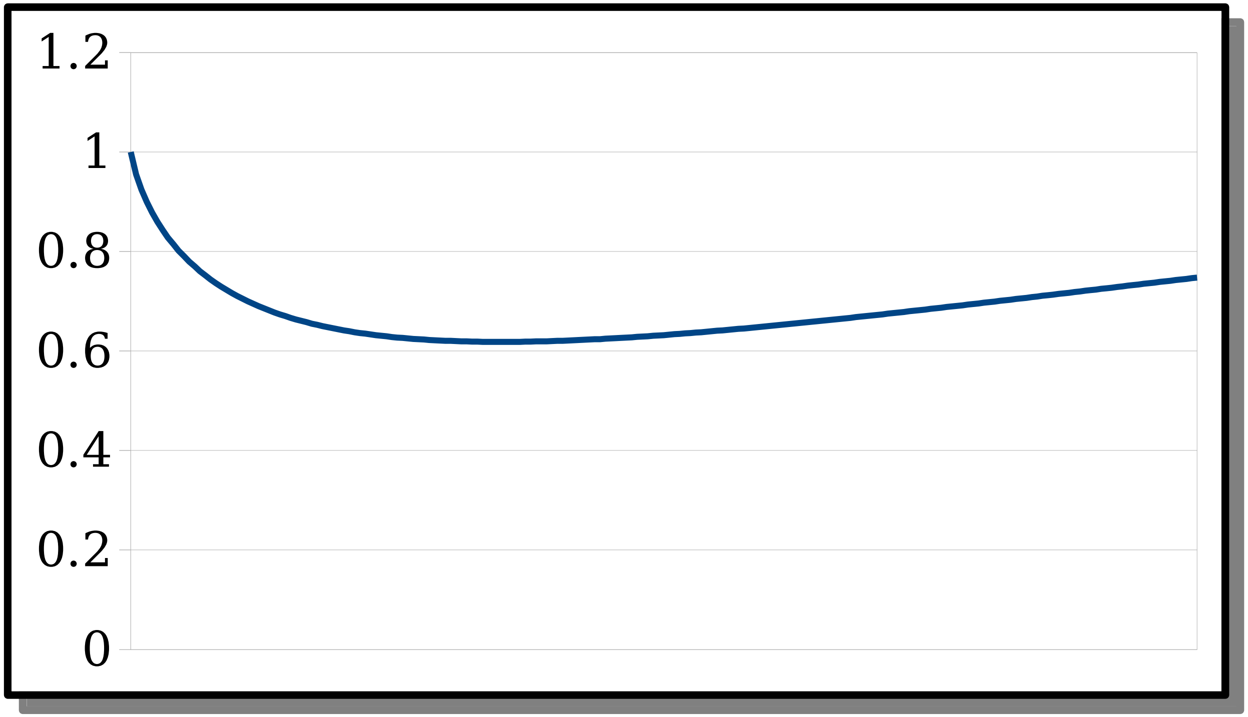
$$(1 - e^{-k\,\alpha^{-1}})^k$$

Question: What choice of k minimizes this expression?

How do we quantify our error rate?

Goal: Pick k to minimize

$$(1 - e^{-k\,\alpha^{-1}})^k.$$



How many hash functions should we use?

Goal: Pick k to minimize

$$(1 - e^{-k\,\alpha^{-1}})^k.$$

Claim: This expression is minimized when

$$\boldsymbol{k = \alpha \ln 2}.$$

You can show this using some symmetry arguments or calculus.

Good exercise: This claim is often repeated and seldom proved. Confirm I am not perpetuating lies.

Challenge: Give an explanation for this result that is “immediately obvious” from the original expression.

How many hash functions should we use?

The false positive rate is

$$(1 - e^{-k\,\alpha^{-1}})^k.$$

and we know to pick

$$k = \alpha \ln 2.$$

Plugging this value into the expression gives a false positive rate of

$$\mathbf{2^{-\alpha \ln 2}}.$$

(The derivation, for those of you who are curious.)

$$\begin{aligned}
& \left(1 - e^{-k\,\alpha^{-1}}\right)^k \\
= \quad & \left(1 - e^{-\alpha \ln 2\,\alpha^{-1}}\right)^{\alpha \ln 2} \\
= \quad & \left(1 - e^{-\ln 2}\right)^{\alpha \ln 2} \\
= \quad & \left(1 - \frac{1}{2}\right)^{\alpha \ln 2} \\
= \quad & 2^{-\alpha \ln 2}
\end{aligned}$$

Knowing what we know now, how many bits do we need to get a false positive rate of ε?

Our false positive rate, as a function of α, is

$$2^{-\alpha \ln 2}.$$

Our goal is to get a false positive rate of ε.

To do so, pick

$$\boldsymbol{\alpha = (\lg \varepsilon^{-1}) / \ln 2}.$$

(The derivation, for those of you who are curious.)

$$\begin{aligned} 2^{-\alpha \ln 2} &= \varepsilon \\ -\alpha \ln 2 &= \lg \varepsilon \\ \alpha &= -\frac{\lg \varepsilon}{\ln 2} \\ \alpha &= \frac{\lg \varepsilon^{-1}}{\ln 2} \end{aligned}$$

Knowing what we know now, how many bits do we need to get a false positive rate of ε?

Given a number of elements n and an error rate ε, pick

$$m = (n \lg \varepsilon^{-1}) / \ln 2$$

$$k = \lg \varepsilon^{-1}$$

Space usage: **$\Theta(n \log \varepsilon^{-1})$**.
(Much better than before!)

Query time: **$O(\log \varepsilon^{-1})$**.
(Slightly worse than before.)

The constant factors here are very, very small. This is an immensely practical data structure, and it gets used all the time!

How did we do overall?

Time-Out for Announcements!

Project Checkpoints

- We’ve spent the weekend reading over project checkpoints and should have those returned with feedback soon.
- Best of luck working on your “interesting” components – we’re very excited to see what you come up with!

Problem Sets

- Problem Set Four was due today at 2:30PM.
  - Want to use a late period? Feel free to do so and turn it in by Thursday at 2:30PM.
- Problem Set Five goes out today. It’s due next Tuesday at 2:30PM.
  - Play around with randomized data structures and the topics we’ve explored in class!
  - While you are allowed to use a late period on this one, we don’t recommend it.

Take-Home Midterm

- Our take-home midterm will be going out next Tuesday at 2:30PM. It'll be due next Thursday at 2:30PM.
- Exam covers topics up through and including randomization. All topics from those lectures are fair game, as are topics from the problem sets.
- The exam is open-note and open-book in the following sense:
  - You can refer to any notes you yourself have taken.
  - You can refer to anything on the course website.
  - You can use CLRS if you'd like (though the exam is designeed so that you shouldn't need it).
  - You may not use any other sources.
- The exam is to be done individually. No collaboration is permitted.

Back to CS166!

Given a number of elements n and an error rate ε, pick

$$m = (n \lg \varepsilon^{-1}) / \ln 2$$

$$k = \lg \varepsilon^{-1}$$

Space usage: **$\Theta(n \log \varepsilon^{-1})$**.
(Much better than before!)

Query time: **$O(\log \varepsilon^{-1})$**.
(Slightly worse than before.)

Question 1: Can we improve this space usage?

Question 2: Can we improve this query time?

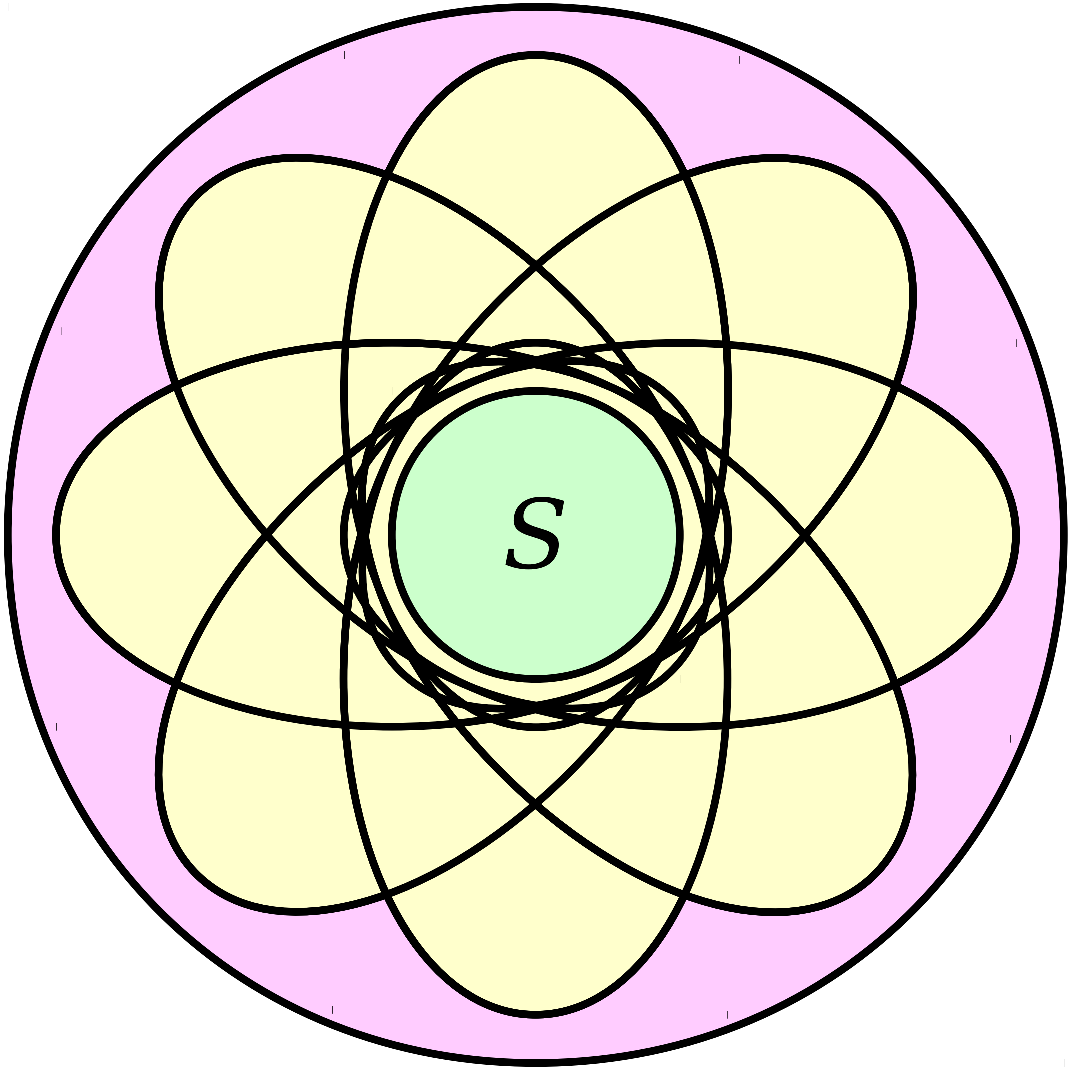
Can we do better?

Suppose we're storing an n-element set S with error rate ε.

Intuition: An AMQ structure stores a set $\hat{S}$: S plus approximately $\varepsilon|U|$ extra elements due to the error rate.

Importantly, we don't care *which* $\varepsilon|U|$ elements those are.

How does that affect our lower bound?



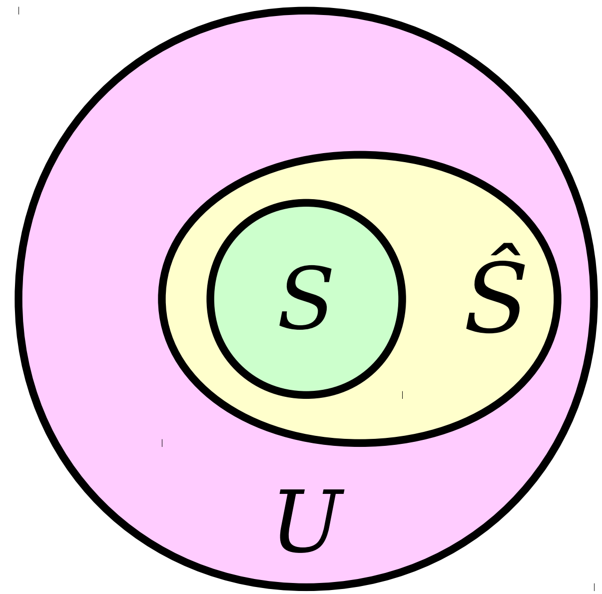
How much memory is needed to solve the approximate membership query problem?

Idea: We can use an AMQ, plus some more bits, to describe a set S.

With b bits, write down an AMQ structure. This describes a set $\hat{S} \subseteq U$ of size around $\varepsilon|U|$.

Write down some more bits to identify which n elements in $\hat{S}$ make up the set S. Bits needed:

$$\lg \binom{\varepsilon |U|}{n}$$



$$b + \lg \binom{\varepsilon |U|}{n} \geq \lg \binom{|U|}{n}$$

(Number of bits needed to describe S.)

(Lower bound on number of bits needed.)

How much memory is needed to solve the approximate membership query problem?

Theorem: Assuming $\varepsilon|U| \gg n$, any AMQ structure needs at least approximately $\boldsymbol{n \lg \varepsilon^{-1}}$ bits in the worst case.

Observation: A Bloom filter uses

$(n \lg \varepsilon^{-1}) / (\ln 2)$

bits, within a factor of $(1 / \ln 2) \approx 1.44$ of optimal.

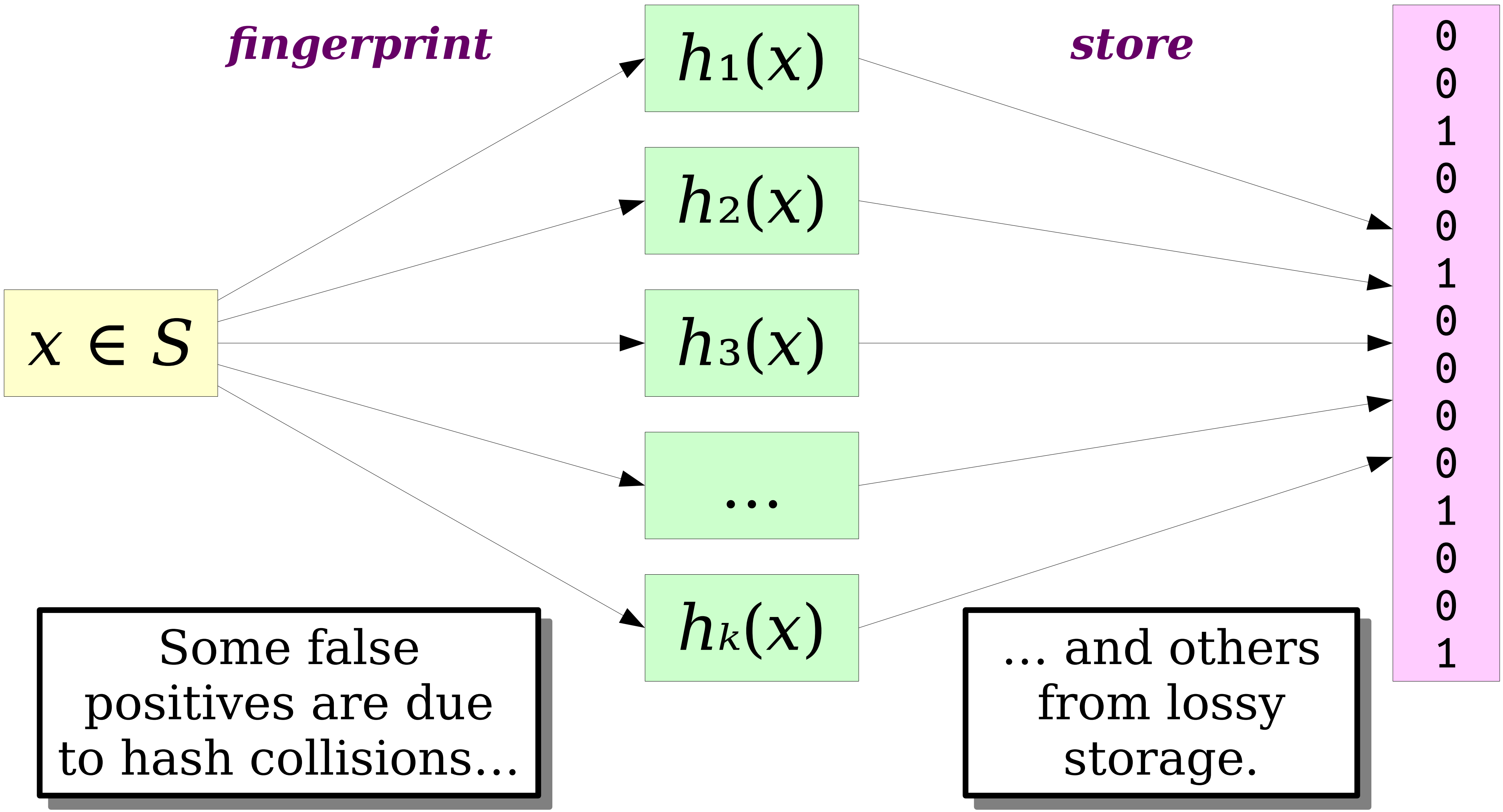
We can only improve on this by a constant factor.

The math, if you're curious:

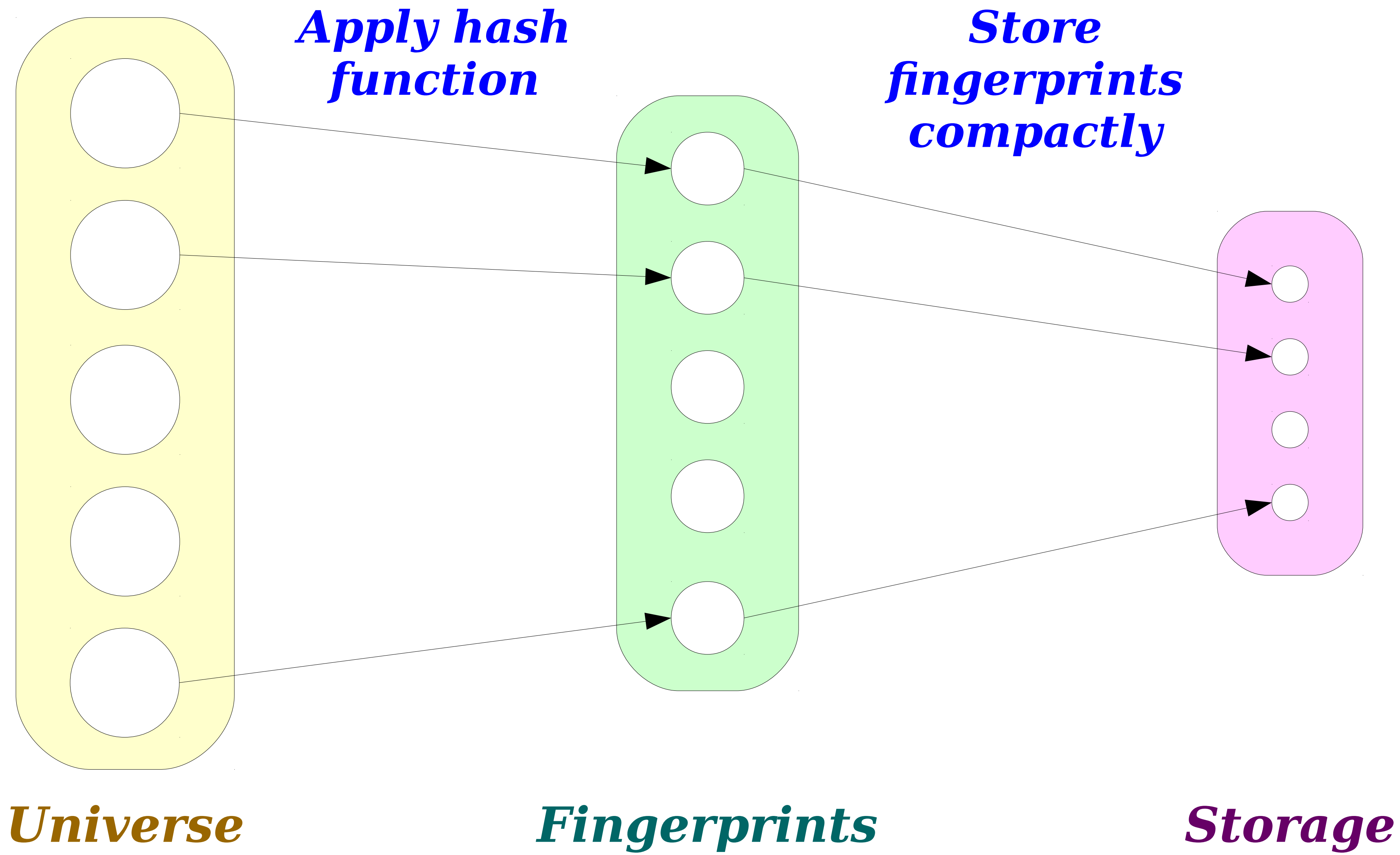
$$
\begin{aligned}
& \lg\left(\frac{\binom{|U|}{n}}{\binom{\varepsilon|U|}{n}}\right) \\
\approx\quad & \lg\left(\frac{|U|^n / n!}{(\varepsilon|U|)^n / n!}\right) \\
=\quad & \lg \varepsilon^{-n} \\
=\quad & n \lg \varepsilon^{-1}
\end{aligned}
$$

How much memory is needed to solve the approximate membership query problem?

Key Question: Fundamentally, what makes Bloom filters tick? And knowing that, can we design other strategies that accomplish the same goal?



Bloom filters compute ***fingerprints*** for each element, then store those fingerprints space-efficiently.



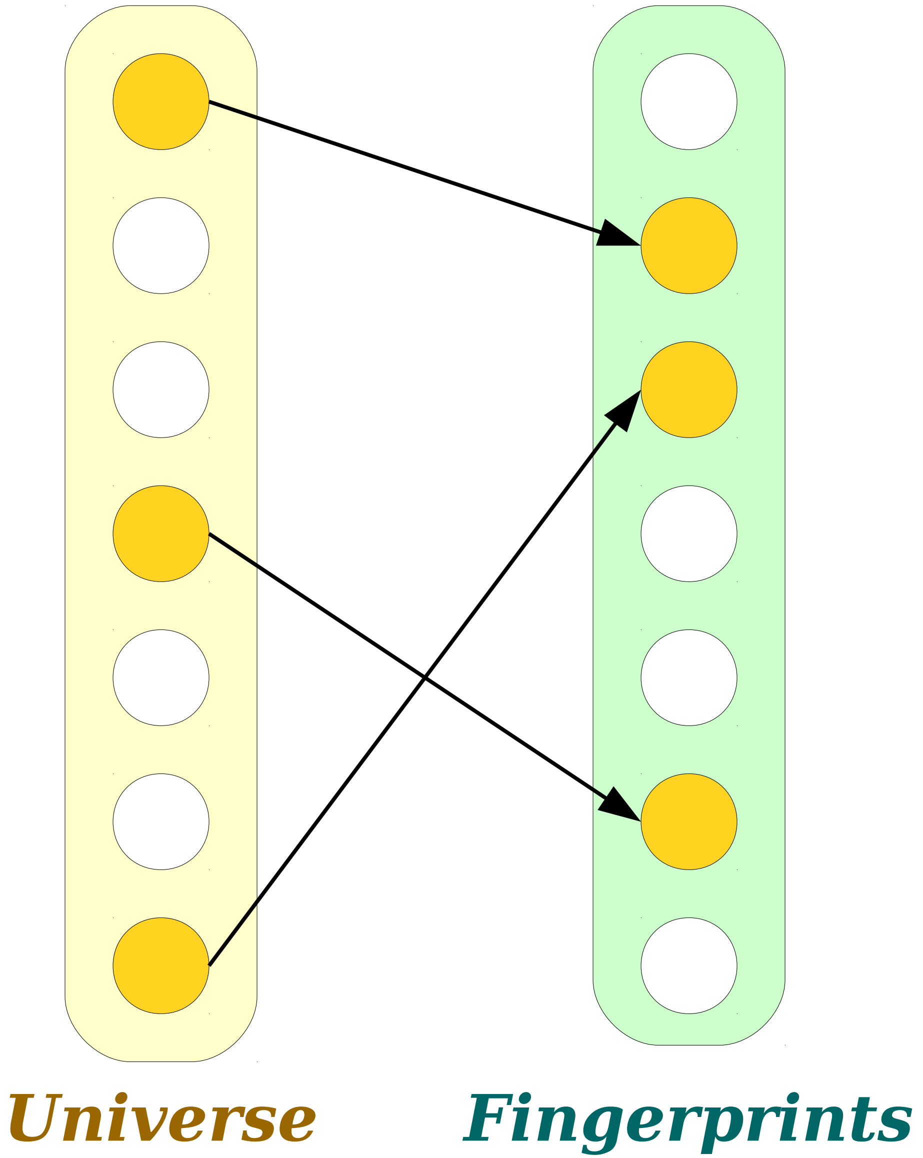
This is a special case of a more general architecture for building approximate membership query structures.

Pick a truly random function $h : U \to [m]$ for fingerprinting.

Select an n-element set $S \subseteq U$ to store, where $n \ll U$.

We incorrectly report $x \in S$ if $h(x) = h(y)$ for some $y \notin S$.

Question: How big should m be so the probability of a false positive is at most ε?

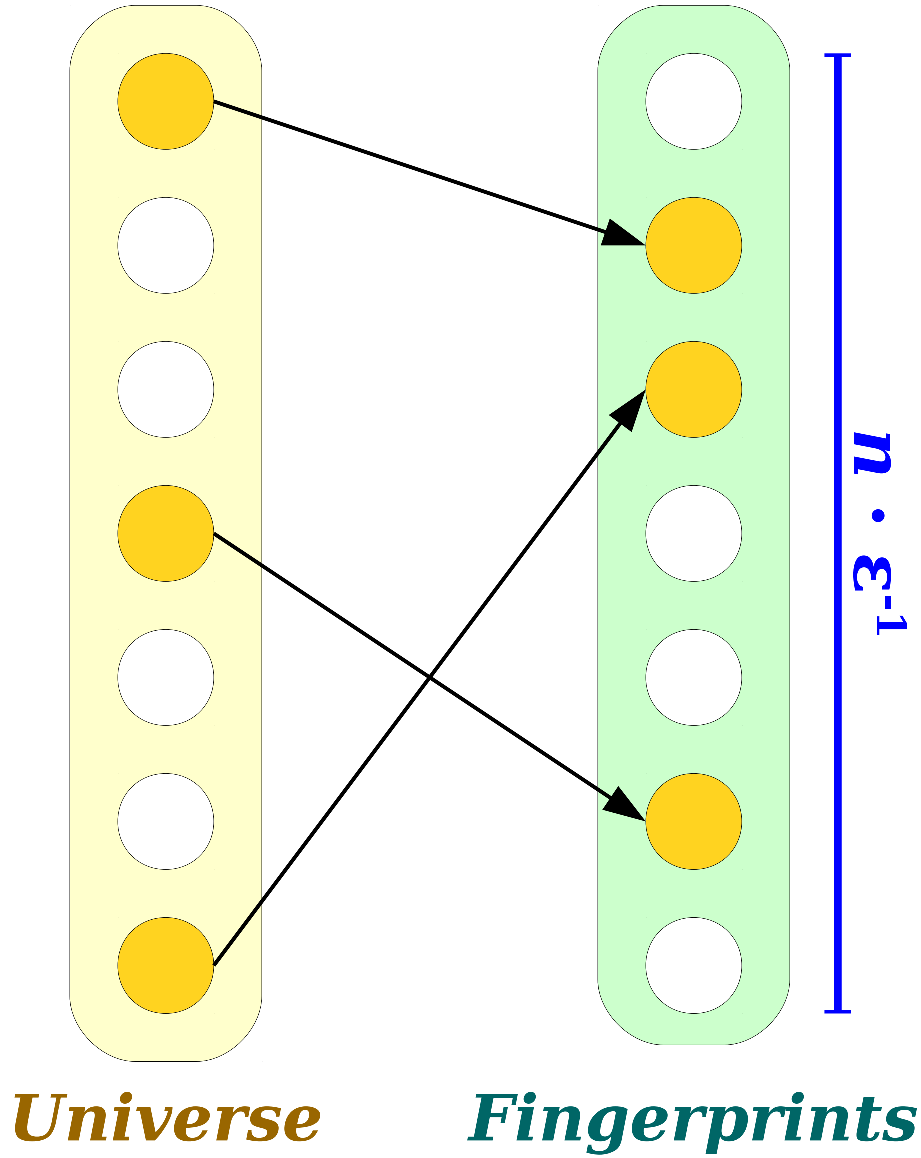


Idea: Choose the number of possible fingerprints to keep the error rate at ε.

Recall: From earlier, if we want a false positive rate of ε hashing *n* elements into *m* slots, we need to pick

$$\boldsymbol{m = n \cdot \varepsilon^{-1}}.$$

So we'll hash our elements to one of $n \cdot \varepsilon^{-1}$ different fingerprints, then store those fingerprints.



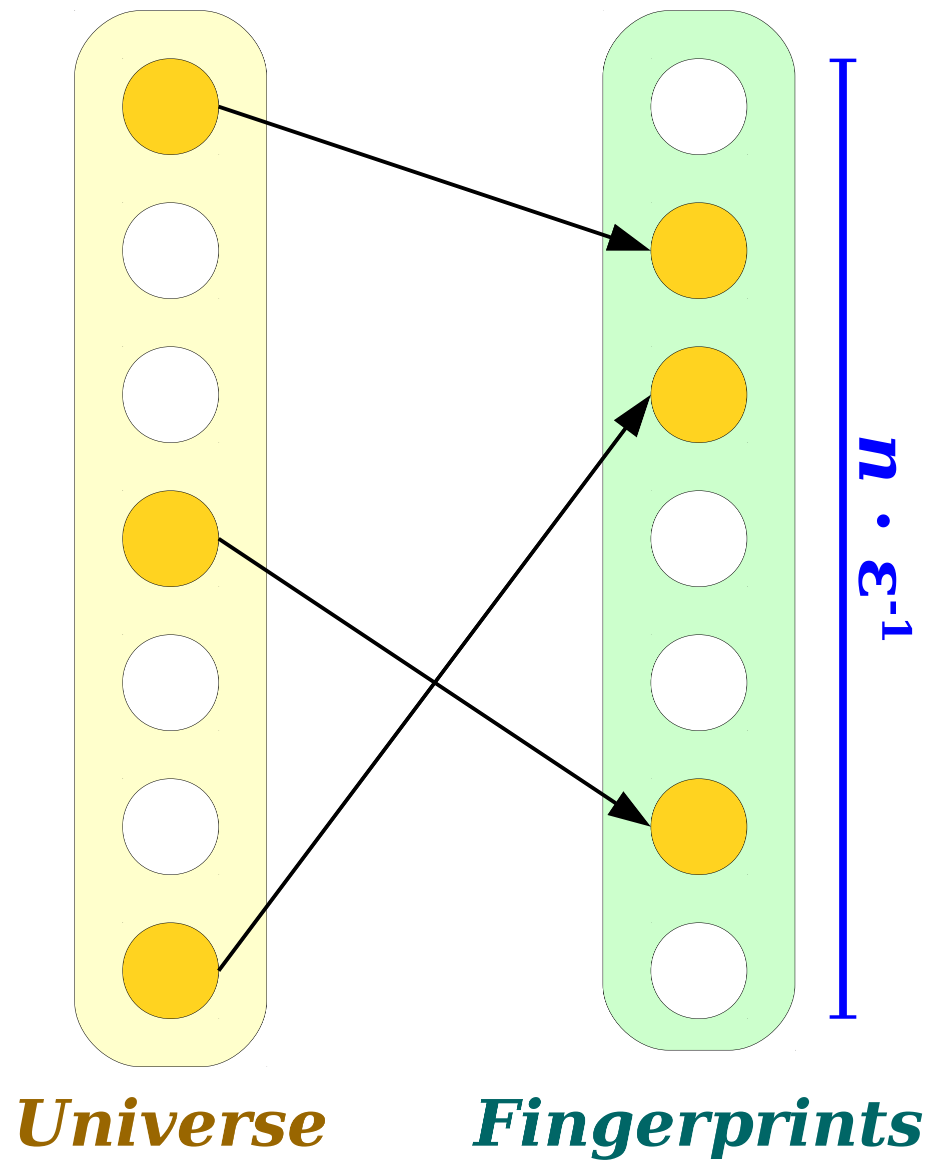
Idea: Choose the number of possible fingerprints to keep the error rate at ε.

Imagine we directly store all n fingerprints, each of which is drawn from a set of $n \cdot \varepsilon^{-1}$ possibilities.

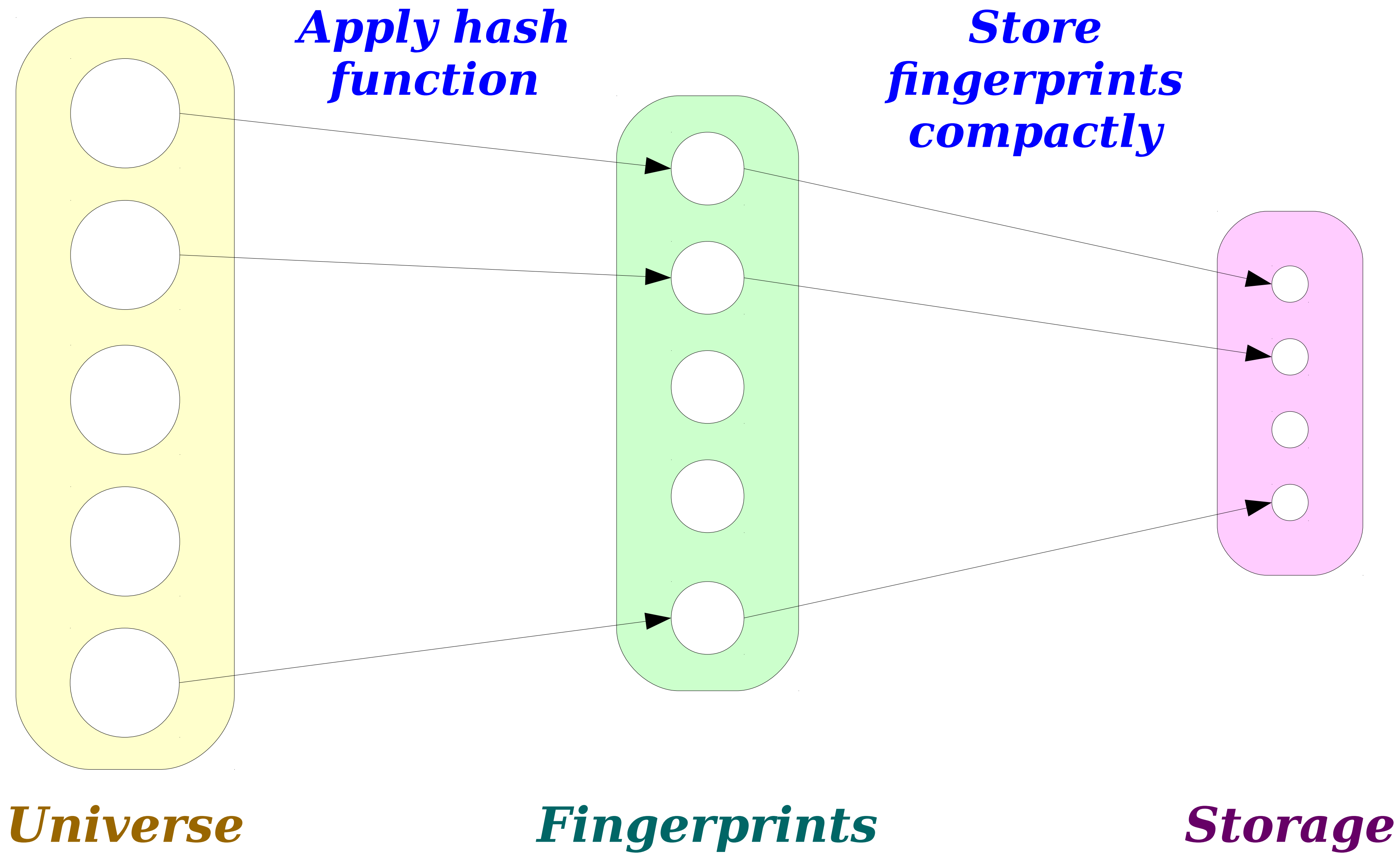
Question: How much space will this use?

Answer: $n \log (n \cdot \varepsilon^{-1})$ bits, more than the $\Theta(n \log \varepsilon^{-1})$ bits used by a Bloom filter.

So we can't just throw all the fingerprints into a hash table and call it a day. We need to be a bit more clever.



Idea: Choose the number of possible fingerprints to keep the error rate at ε.

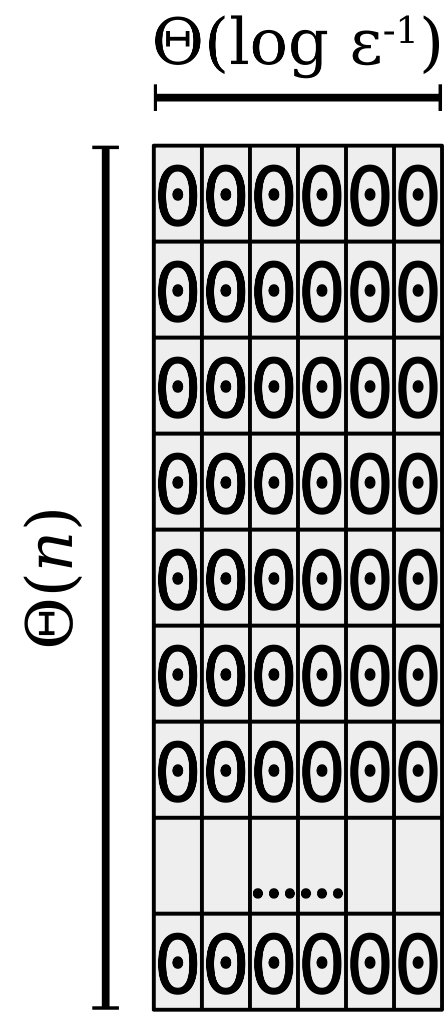


Idea: Find a compact way to encode all n fingerprints in space $\Theta(n \log \varepsilon^{-1})$.

Idea: Work backwards! We know we have $\Theta(n \log \varepsilon^{-1})$ bits to work with. How might we arrange them?

What about a table of size $\Theta(n) \times \Theta(\log \varepsilon^{-1})$?

This is just a hunch at this point, but let's run with it! Let's see what happens.



Idea: Find a compact way to encode all n fingerprints in space $\Theta(n \log \varepsilon^{-1})$.

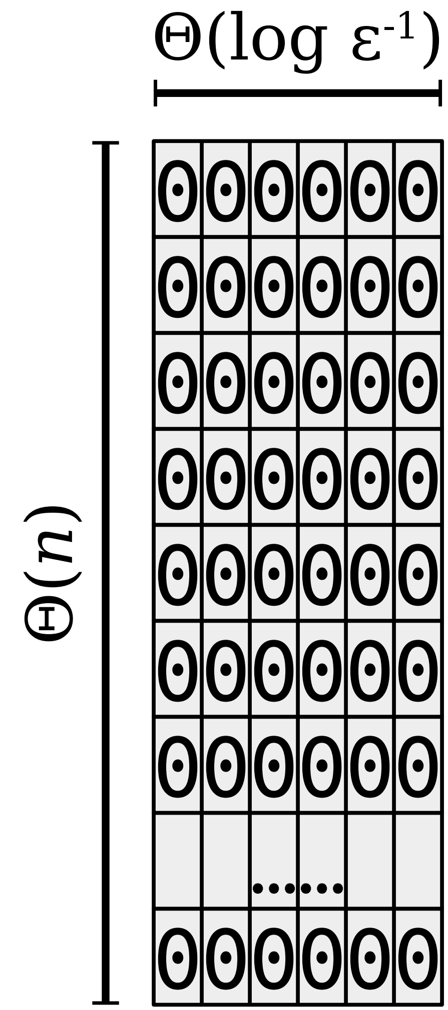
There are $n \cdot \varepsilon^{-1}$ possible fingerprints. Each requires $\log (n \cdot \varepsilon^{-1})$ bits to write out.

Very cool observation:

$$\lg (n \cdot \varepsilon^{-1}) = \underbrace{\lg n} + \lg \varepsilon^{-1}$$

Bits required to store a number between 0 and n – 1, inclusive.

Key Idea: Decompose the fingerprint into a ***table index*** and a ***residual hash***.



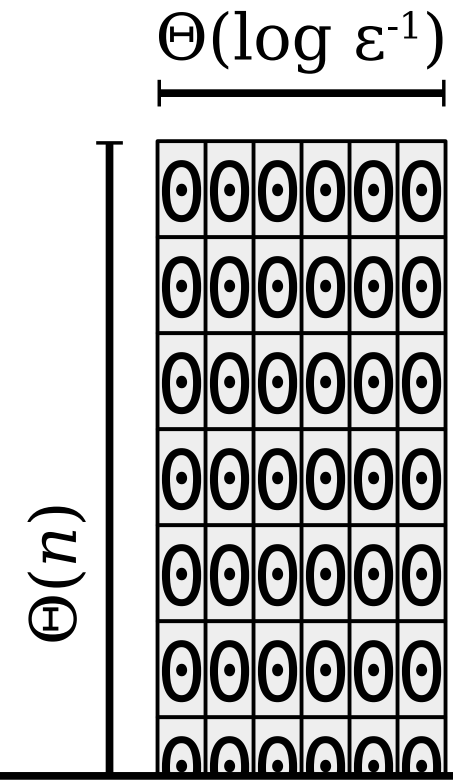
Idea: Find a compact way to encode all n fingerprints in space $\Theta(n \log \varepsilon^{-1})$.

Key Idea: Decompose each fingerprint into a ***table index*** and a ***residual hash***.

Go to this index...

... and store this value.

10101111	**001101**
01101110	**010111**
01111000	**100110**



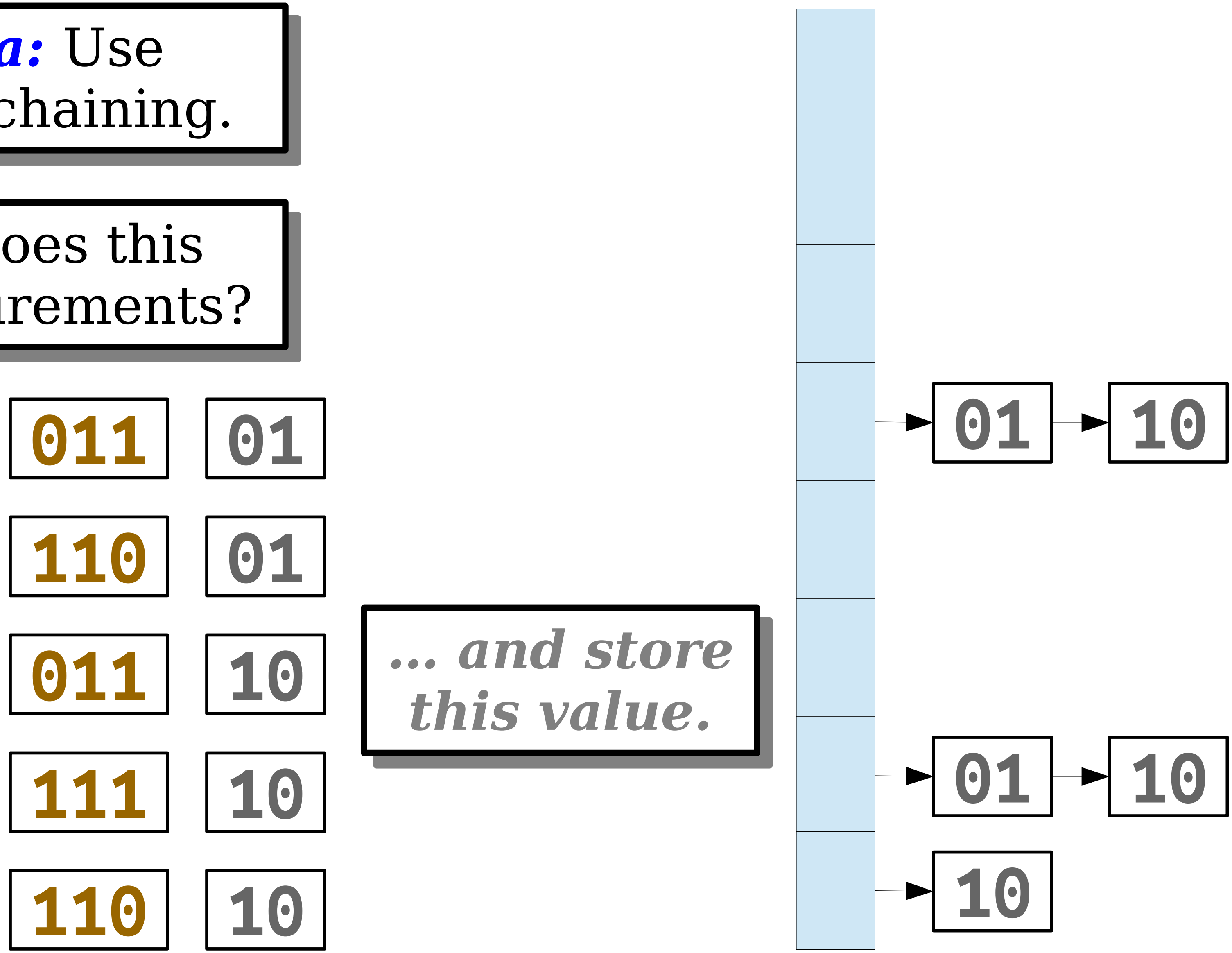
Insight: We're essentially being asked to create a compressed hash table!

Idea: Find a compact way to encode all n fingerprints in space $\Theta(n \log \varepsilon^{-1})$.

Initial Idea: Use hashing with chaining.

Question: Does this meet our requirements?

Go to this index…



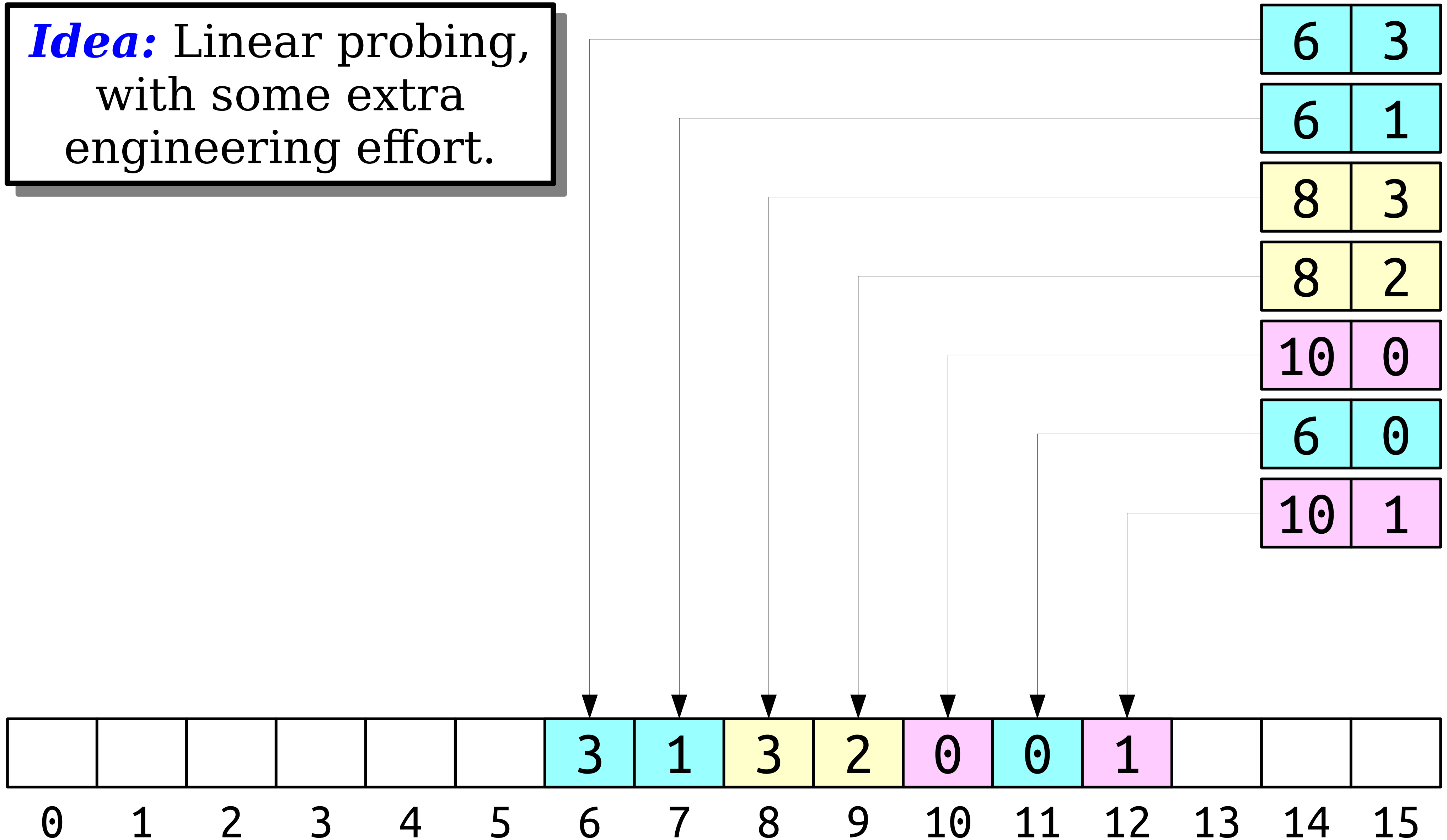
Challenge: Building this compressed data structure is trickier than it seems.

Idea: Linear probing, with some extra engineering effort.

0110	11
0110	01
1000	11
1000	10
1010	00
0110	00
1010	01

Challenge: Building this compressed data structure is trickier than it seems.

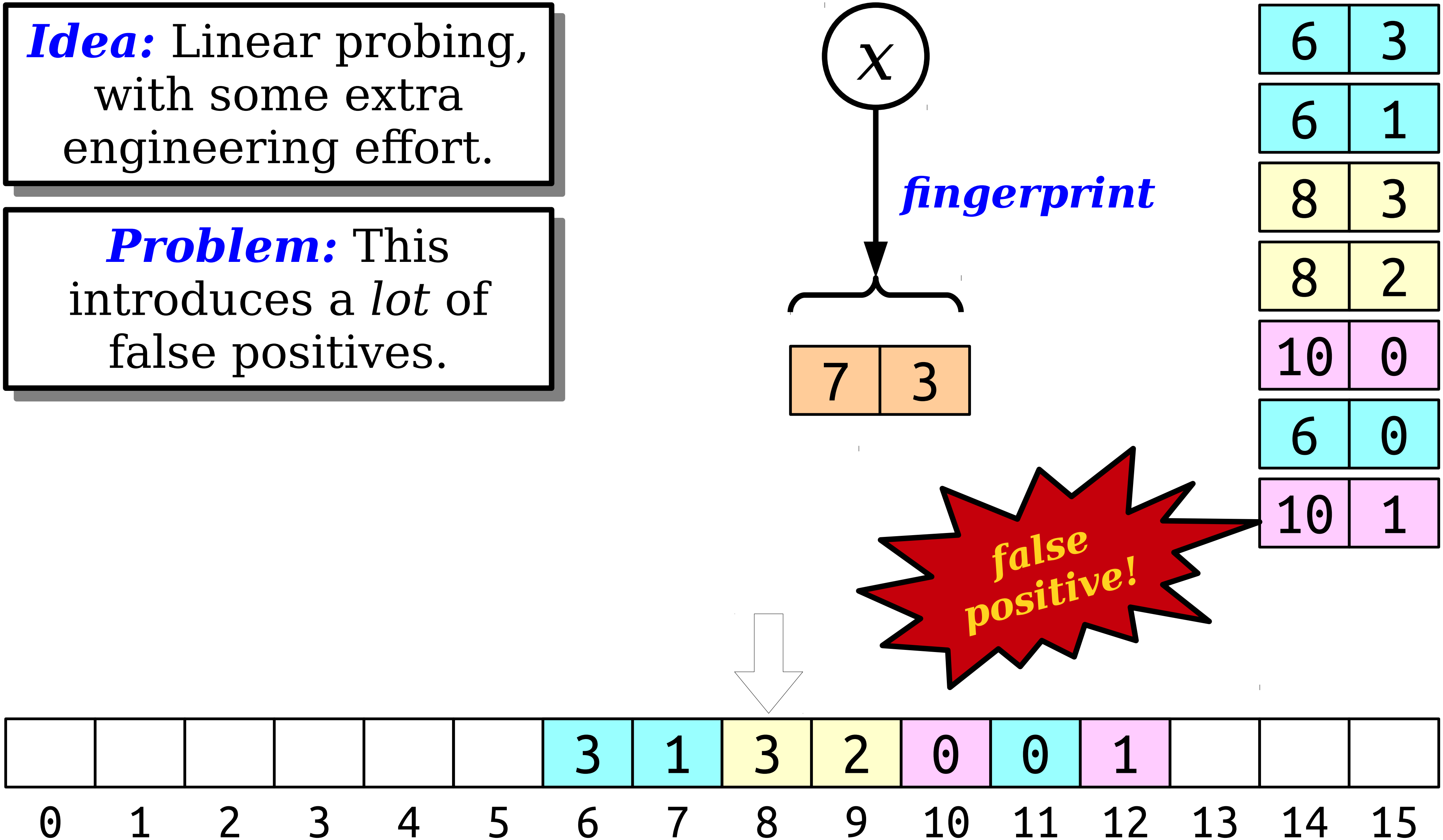
Idea: Linear probing, with some extra engineering effort.



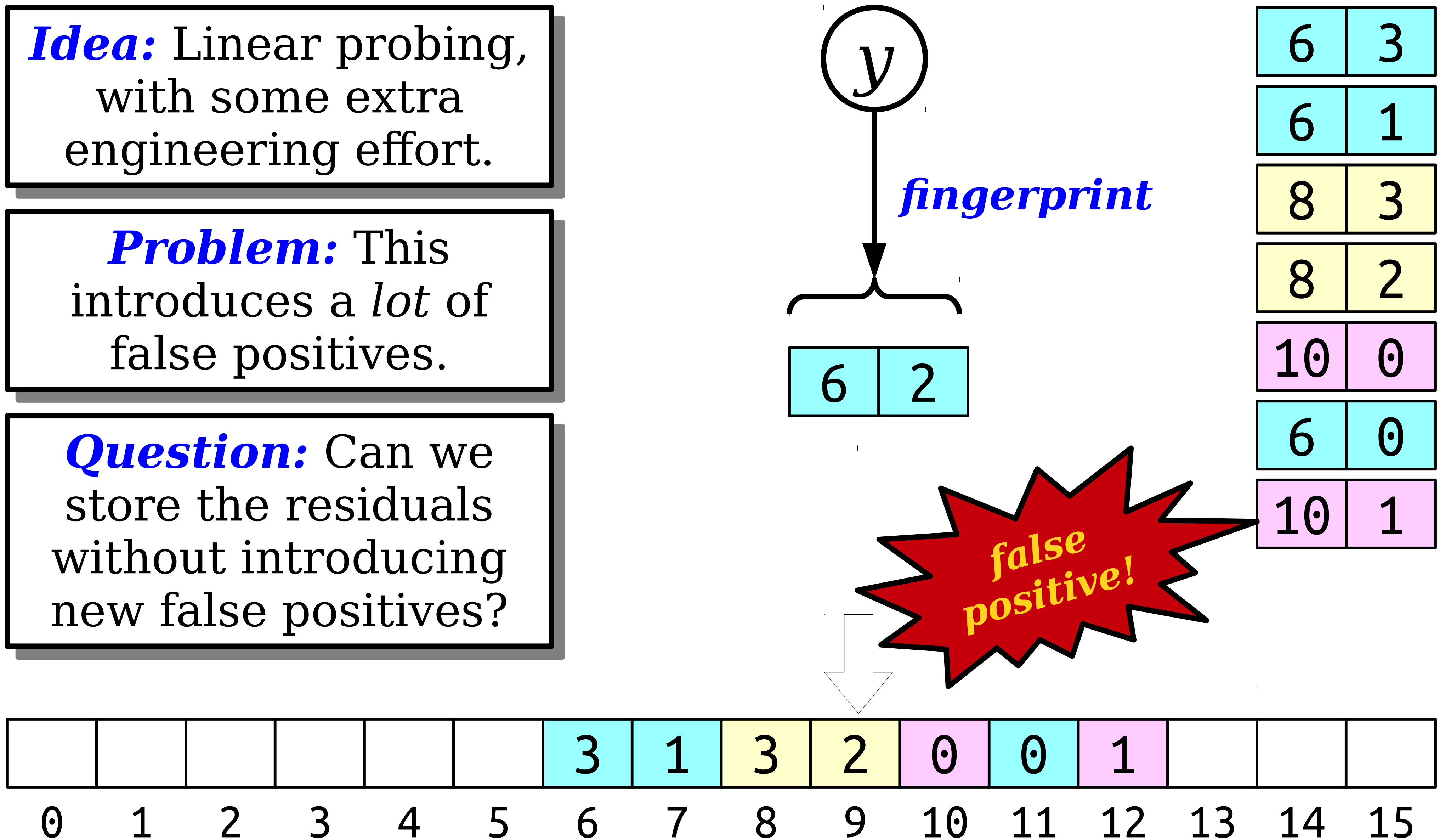
Challenge: Building this compressed data structure is trickier than it seems.

Idea: Linear probing, with some extra engineering effort.

Problem: This introduces a *lot* of false positives.



Challenge: Building this compressed data structure is trickier than it seems.

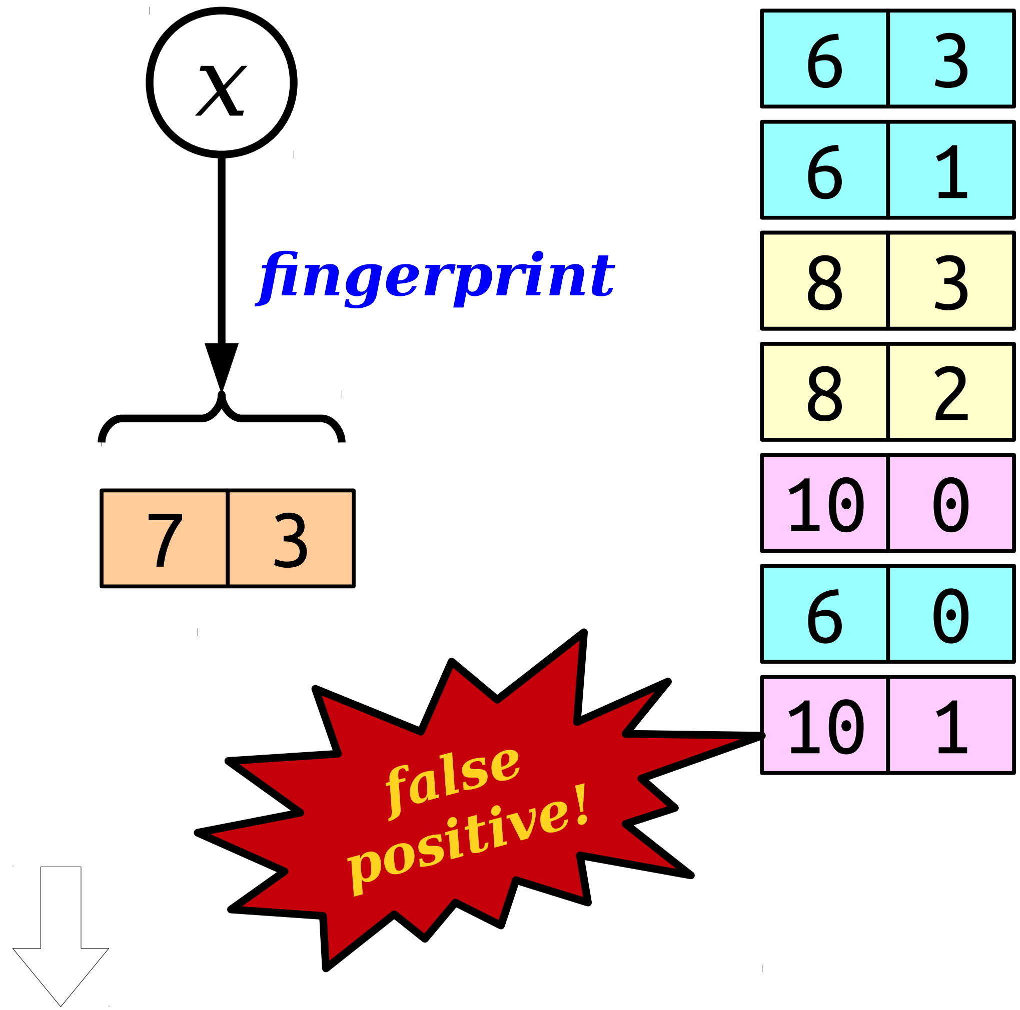


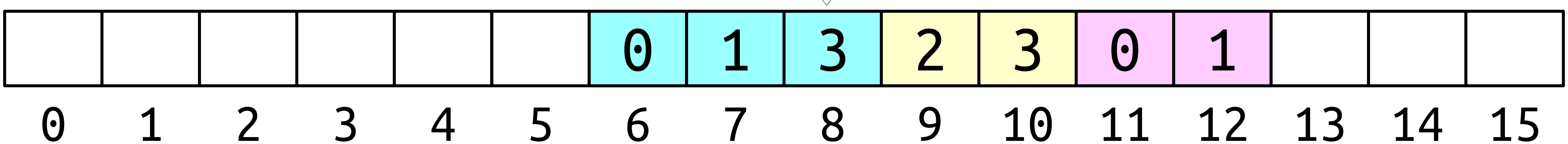
Challenge: Building this compressed data structure is trickier than it seems.

Intuition: Many problems are easier if we sort things.

Use Robin Hood hashing: group keys from the same index.

Problem: We still have the same issues.



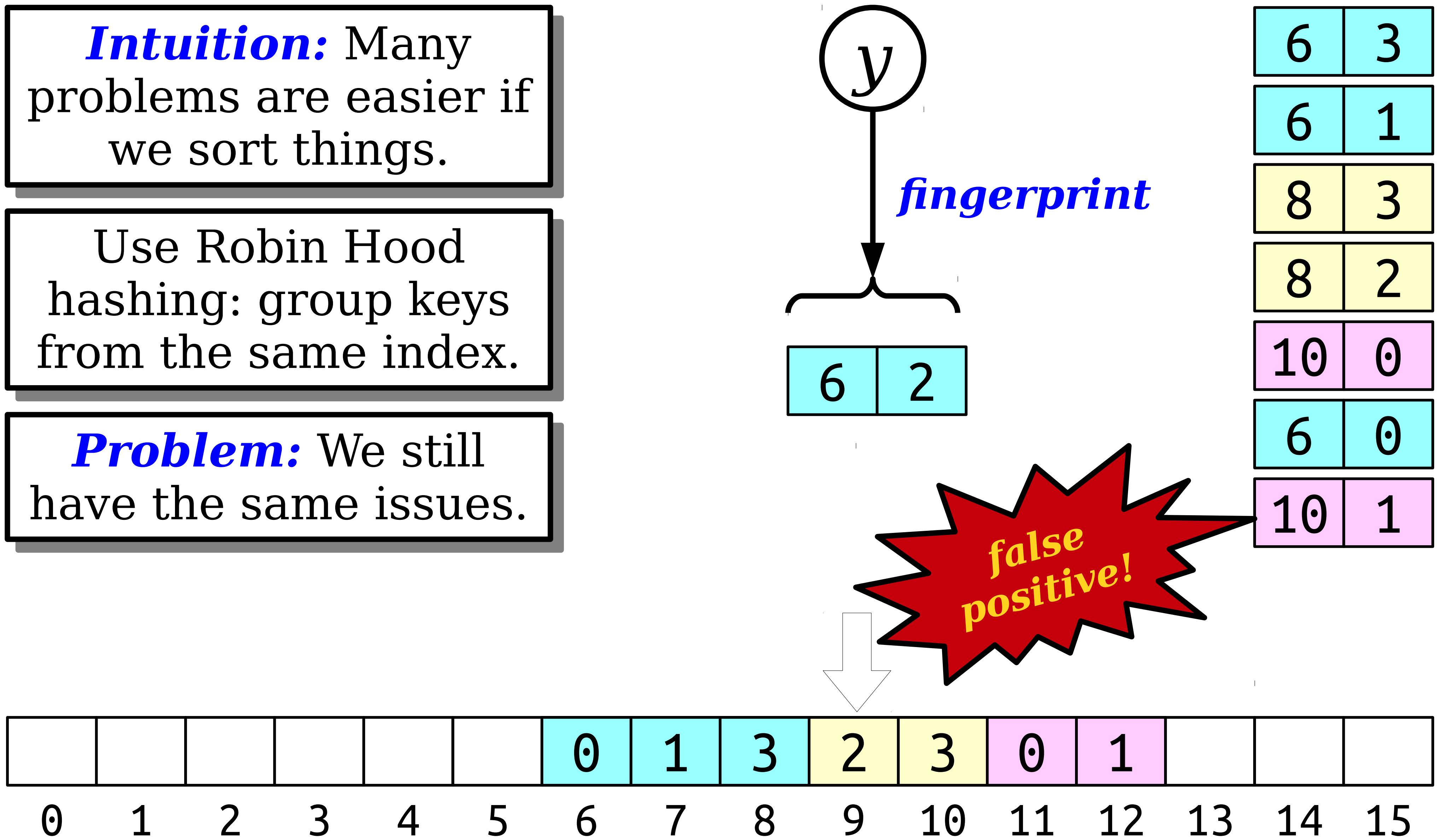


Challenge: Building this compressed data structure is trickier than it seems.

Intuition: Many problems are easier if we sort things.

Use Robin Hood hashing: group keys from the same index.

Problem: We still have the same issues.



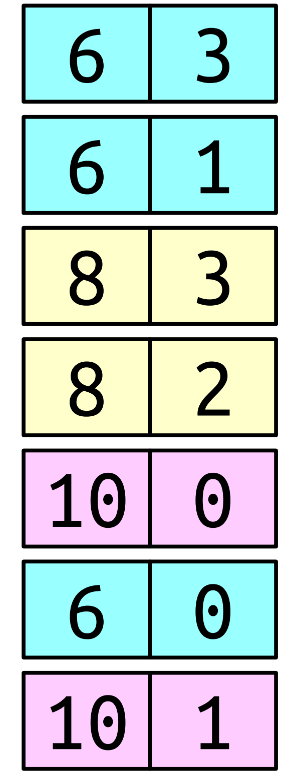
Challenge: Building this compressed data structure is trickier than it seems.

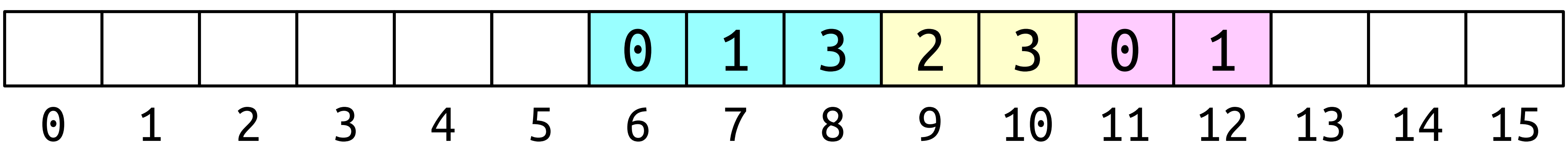
We get false positives from two sources:

1. Lookups starting at an index for which there are no signatures.

2. Lookups that can't tell where an index's elements start or stop.

How can we fix this?

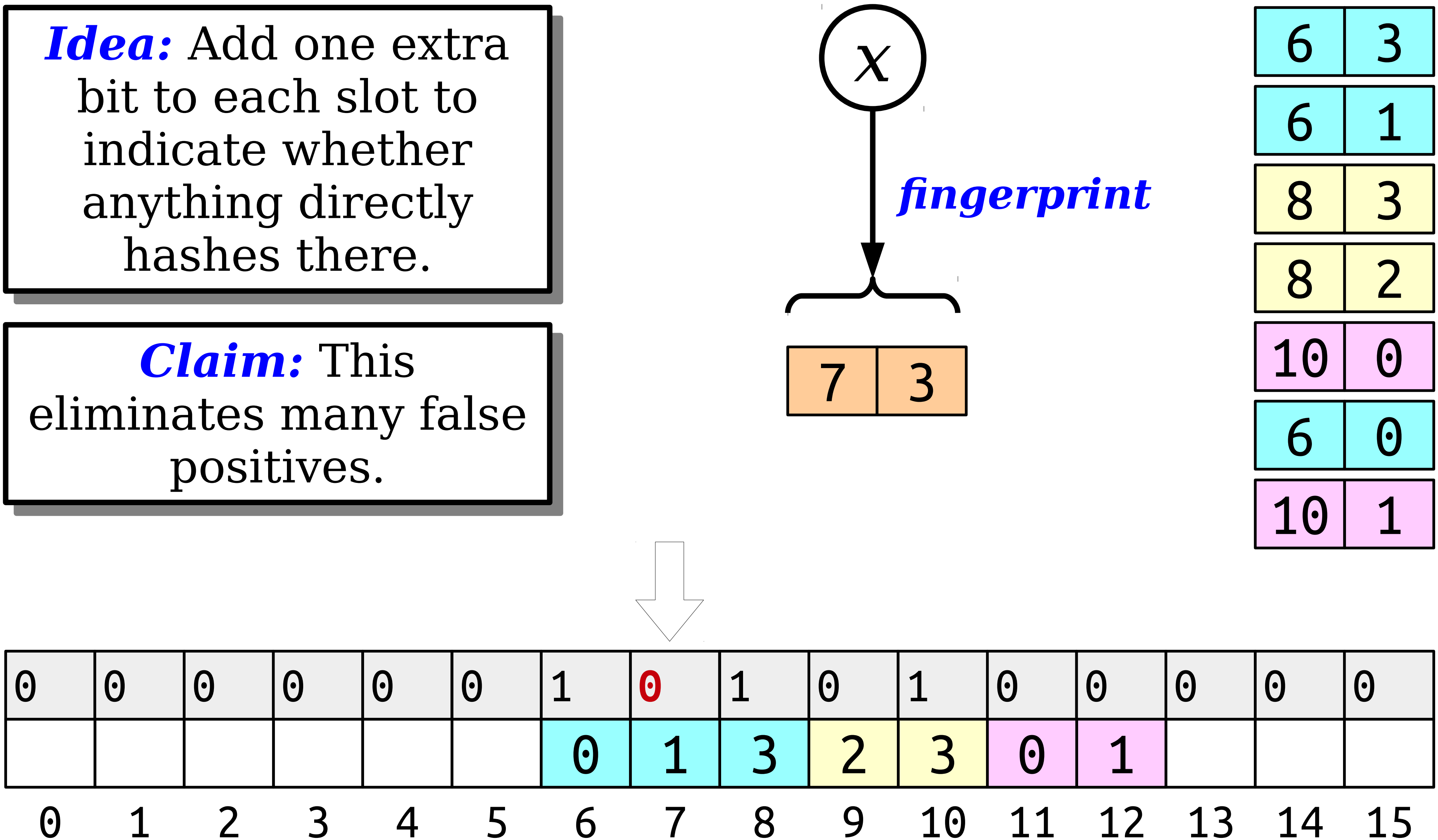




Challenge: Building this compressed data structure is trickier than it seems.

Idea: Add one extra bit to each slot to indicate whether anything directly hashes there.

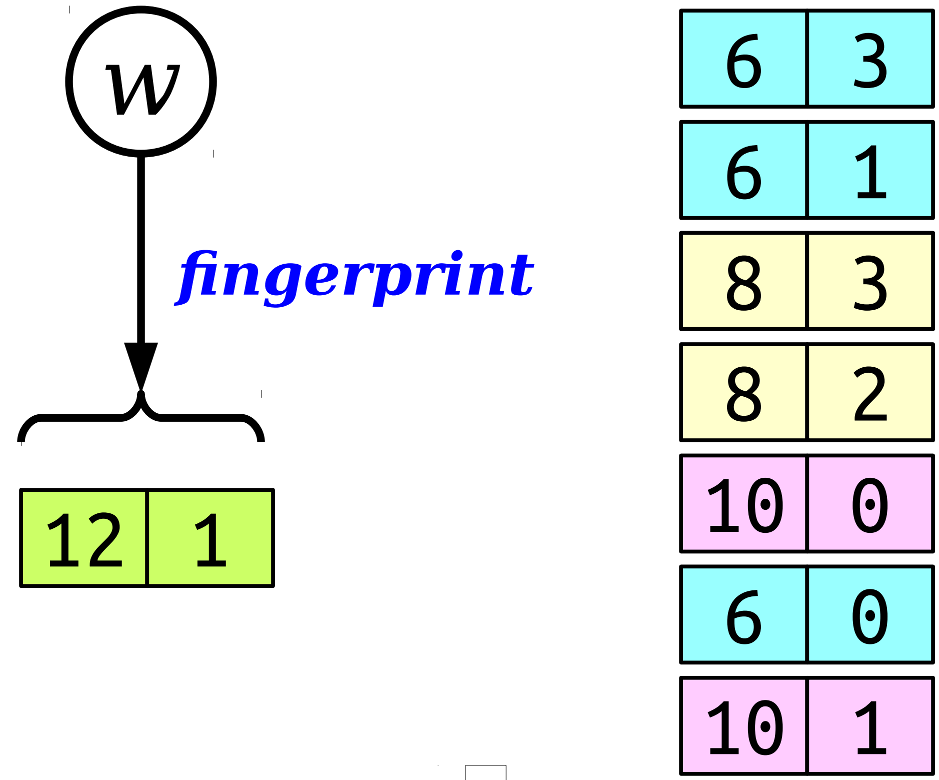
Claim: This eliminates many false positives.

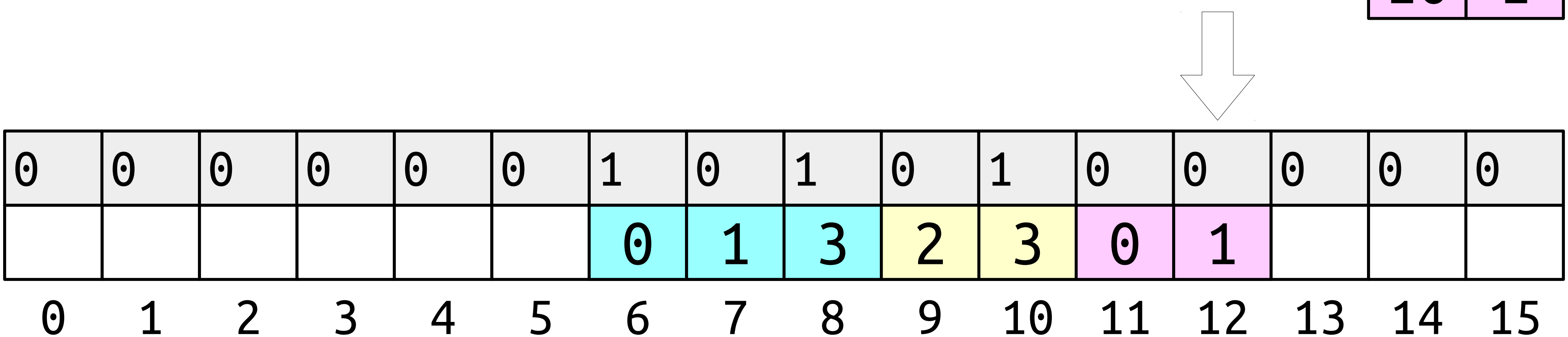


Challenge: Building this compressed data structure is trickier than it seems.

Idea: Add one extra bit to each slot to indicate whether anything directly hashes there.

Claim: This eliminates many false positives.

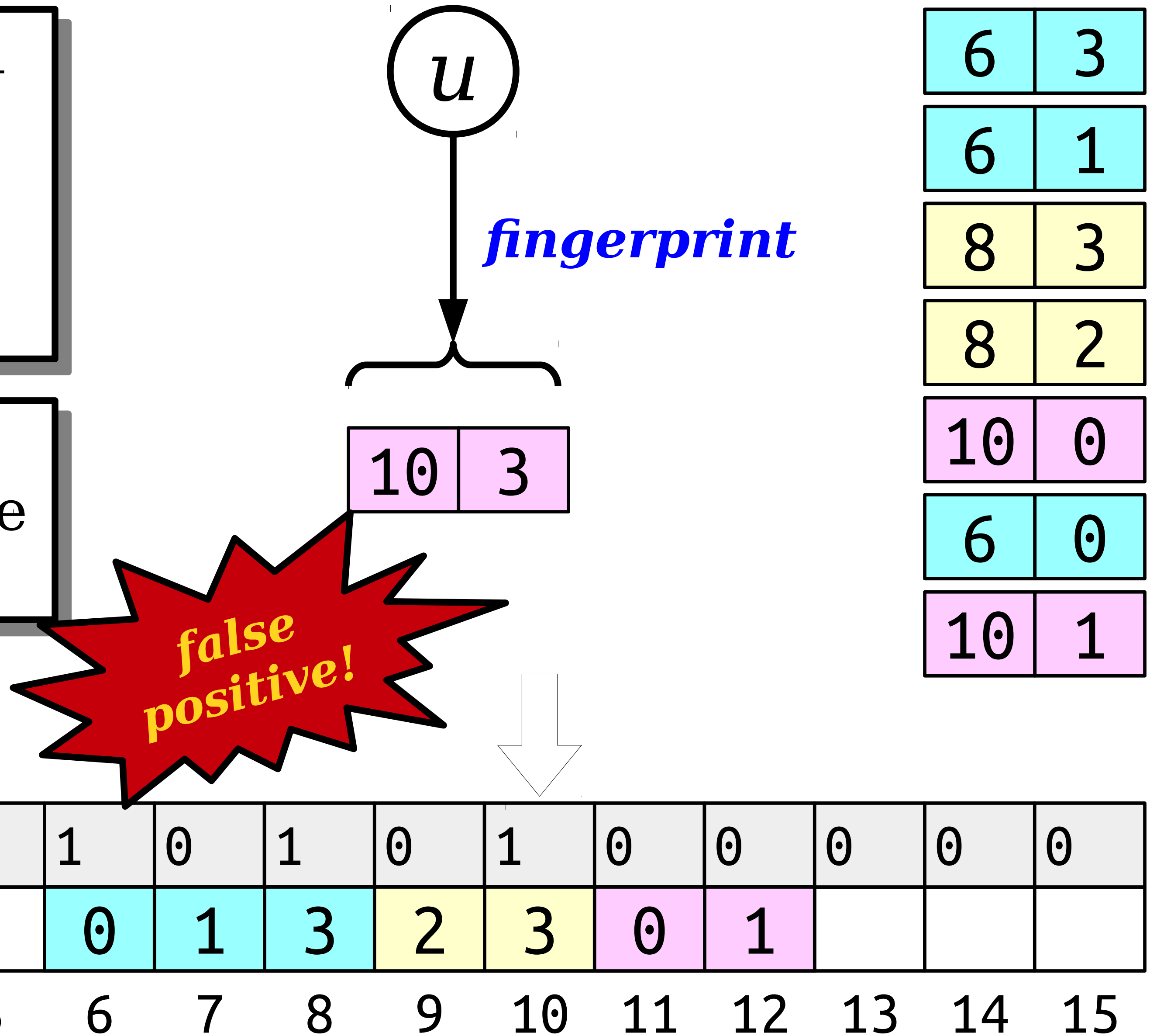




Challenge: Building this compressed data structure is trickier than it seems.

Idea: Add one extra bit to each slot to indicate whether anything directly hashes there.

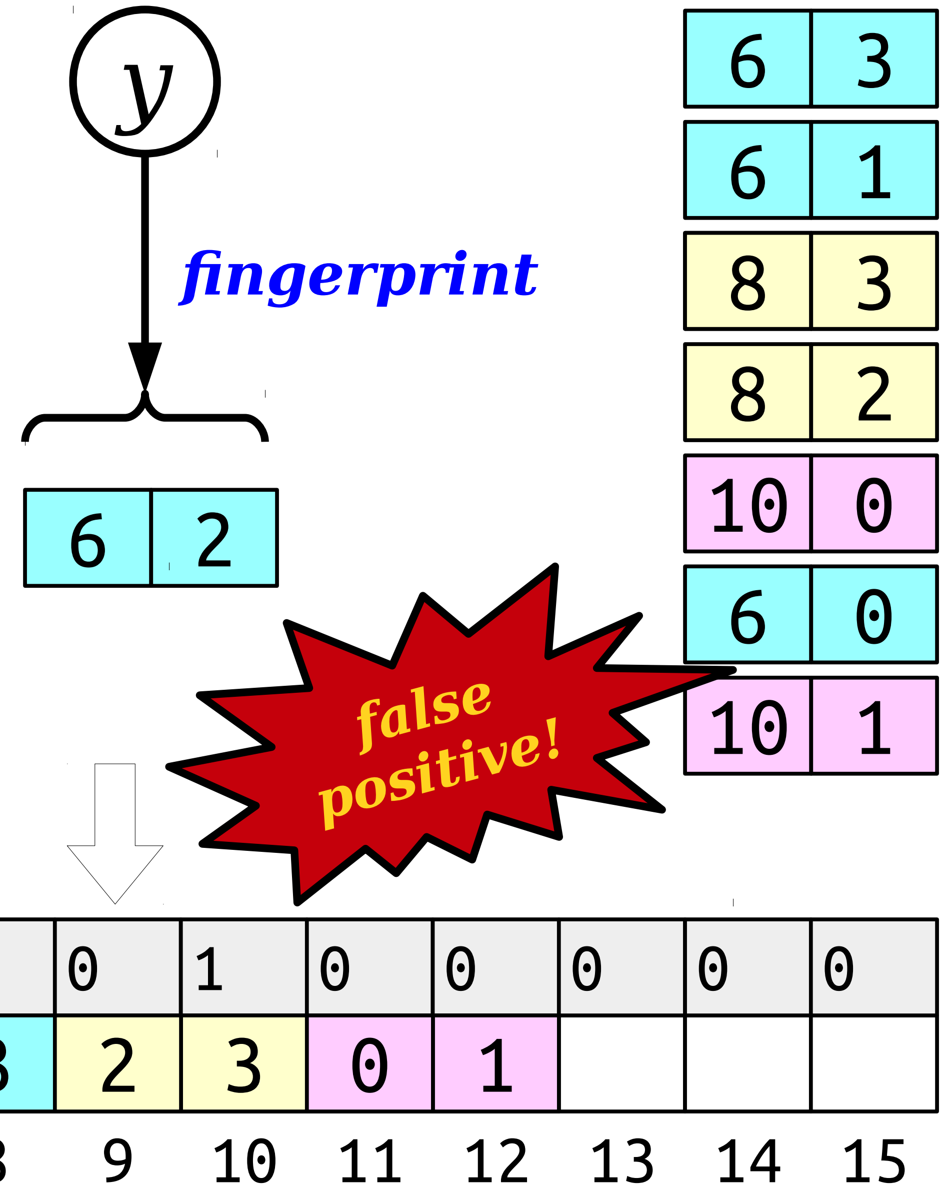
Claim: This eliminates many false positives.



Challenge: Building this compressed data structure is trickier than it seems.

Problem: We can get false positives by scanning past the end of a range.

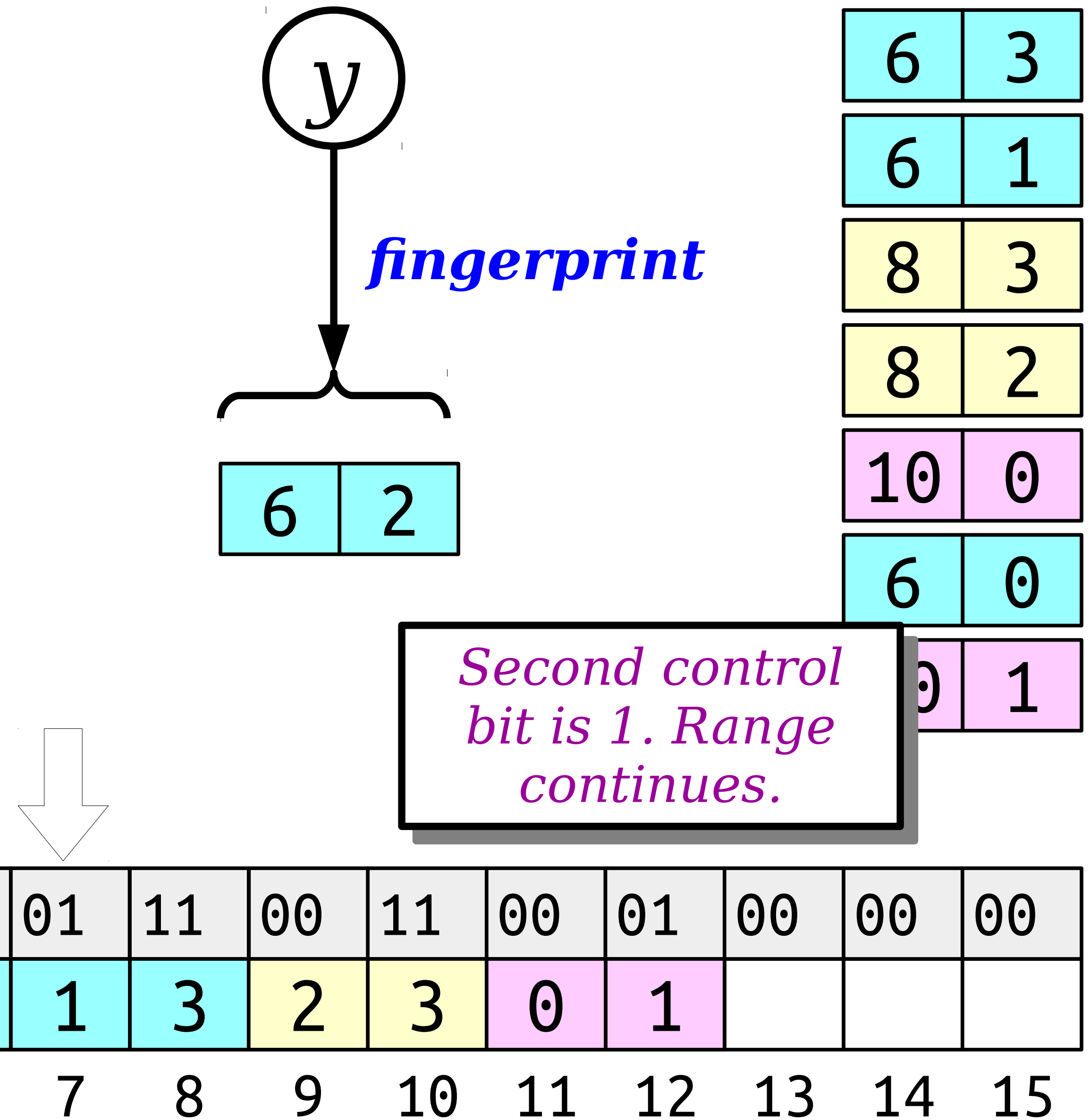
Idea: Mark where each range of signatures from a base index ends.



Challenge: Building this compressed data structure is trickier than it seems.

Idea: Add one extra bit to each slot to indicate whether a run of equal indices continues here.

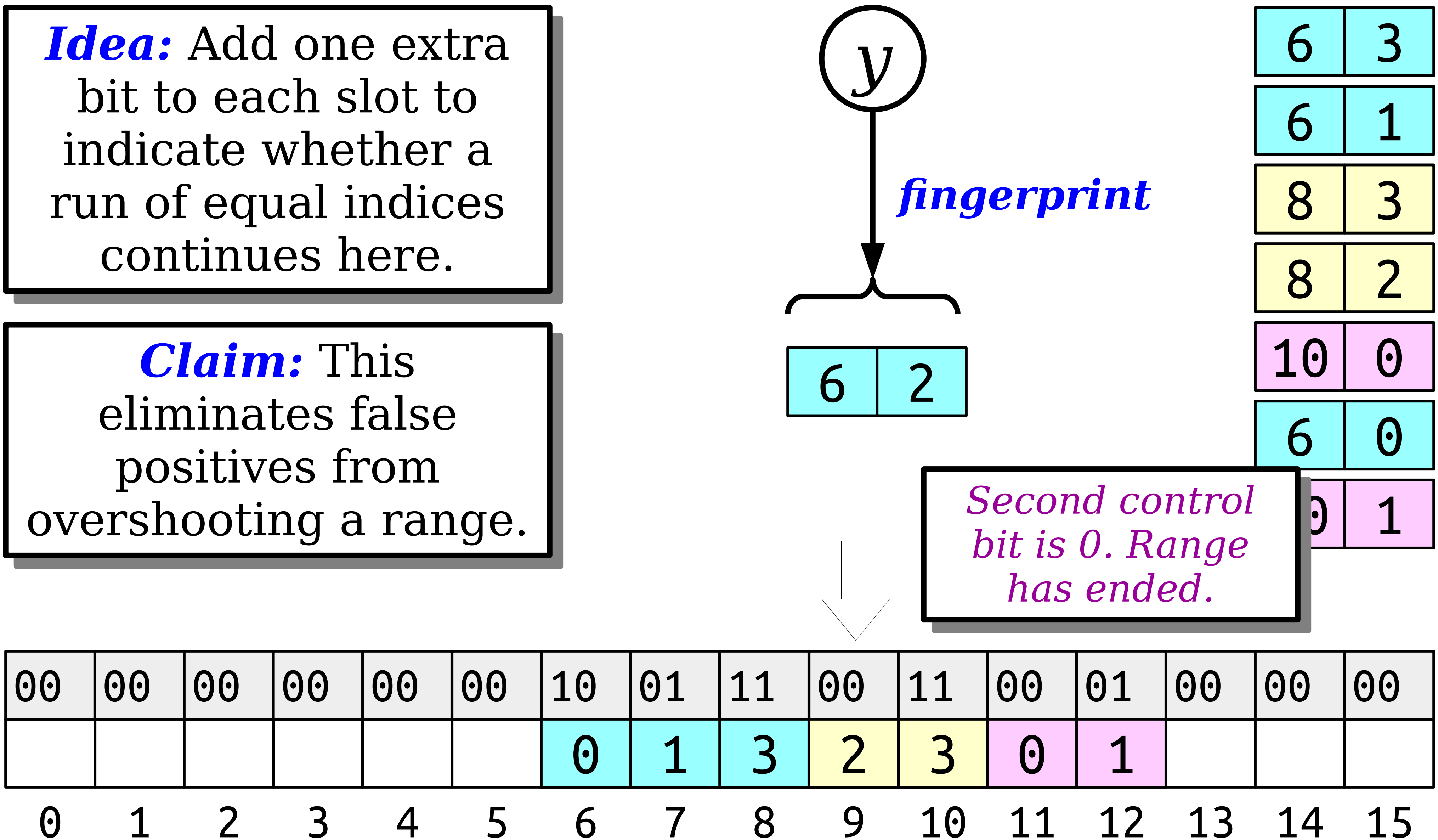
Claim: This eliminates false positives from overshooting a range.



Challenge: Building this compressed data structure is trickier than it seems.

Idea: Add one extra bit to each slot to indicate whether a run of equal indices continues here.

Claim: This eliminates false positives from overshooting a range.

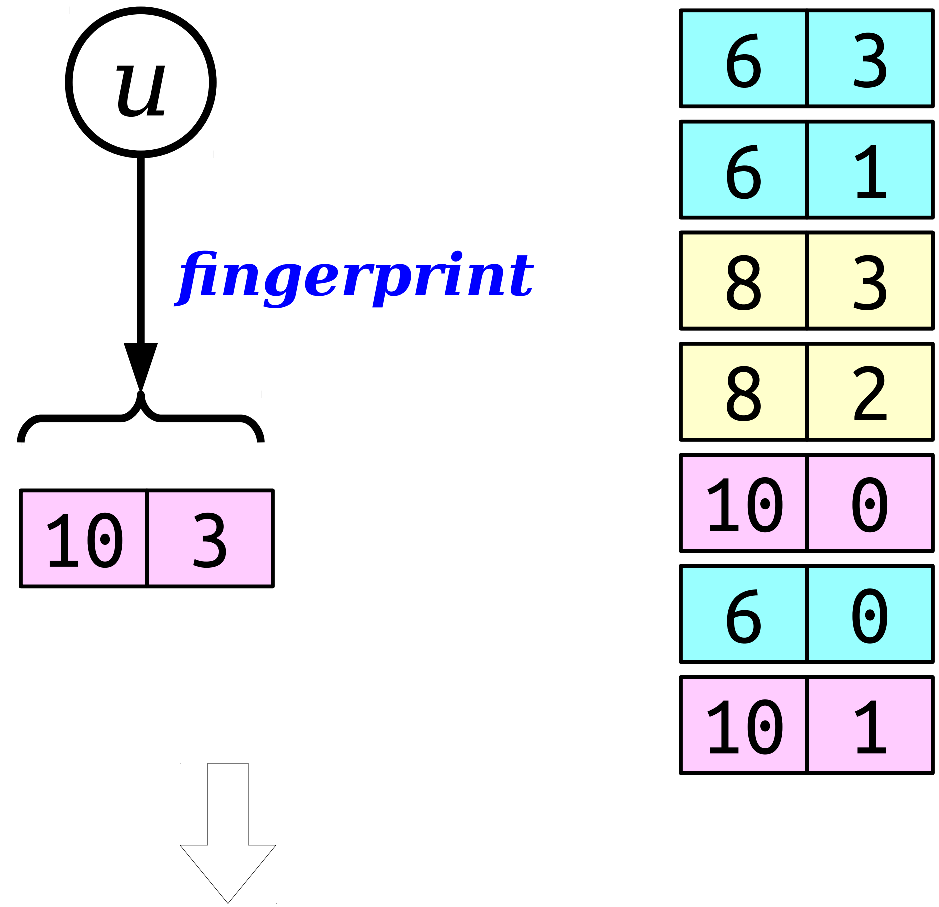


Challenge: Building this compressed data structure is trickier than it seems.

This new bit lets us detect when we hash into the middle of a range.

However, we can't currently act on this information.

How do we fix this?

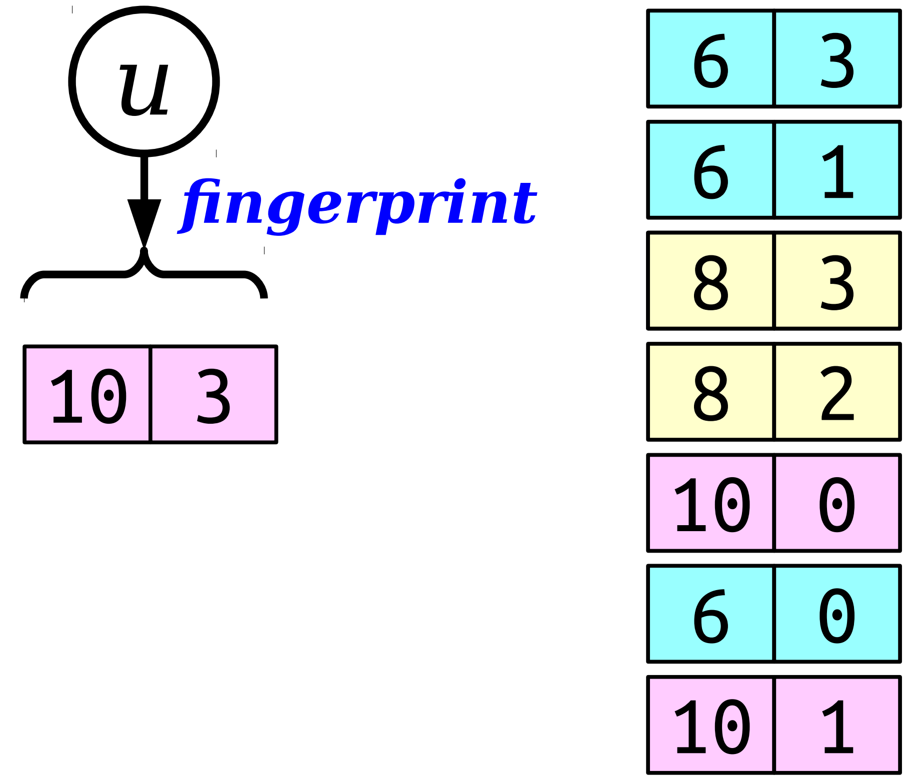


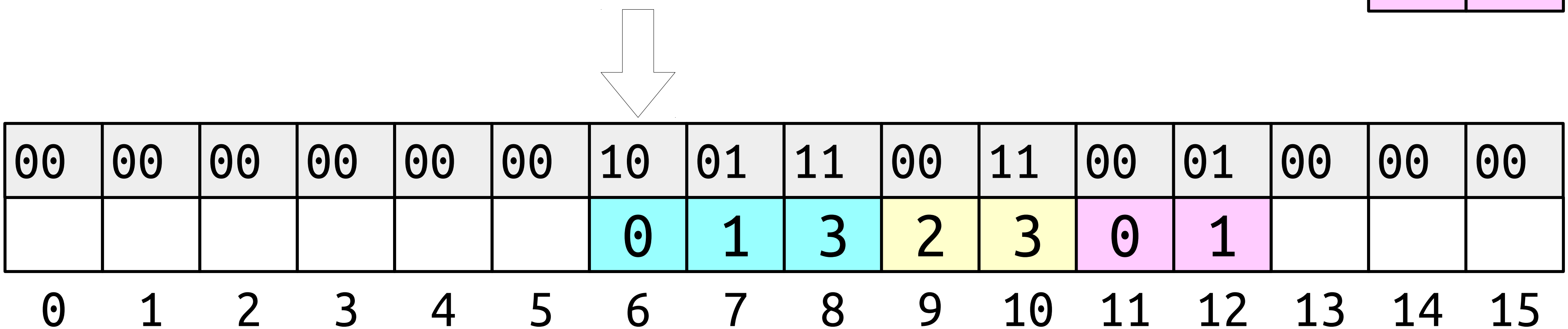
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
00	00	00	00	00	00	10	01	11	00	11	00	01	00	00	00
						0	1	3	2	3	0	1			

Challenge: Building this compressed data structure is trickier than it seems.

Claim: If we can find the first element in the cluster, we can search without false positives.

Intuition: We can track which index range we're in.





Challenge: Building this compressed data structure is trickier than it seems.

Claim: If we can find the first element in the cluster, we can search without false positives.

Intuition: We can track which index range we're in.

6	3
6	1
8	3
8	2
10	0
6	0
10	1

00	00	00	00	00	00	10	01	11	00	11	00	01	00	00	00
						0	1	3	2	3	0	1			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Challenge: Building this compressed data structure is trickier than it seems.

Idea: Add one extra bit to each slot to indicate whether it continues a longer cluster.

6	3
6	1
8	3
8	2
10	0
6	0
10	1

000	000	000	000	000	000	100	011	111	001	111	001	011	000	000	000
						0	1	3	2	3	0	1			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Challenge: Building this compressed data structure is trickier than it seems.

Idea: Add one extra bit to each slot to indicate whether it continues a longer cluster.

6	3
6	1
8	3
8	2
10	0
6	0
10	1

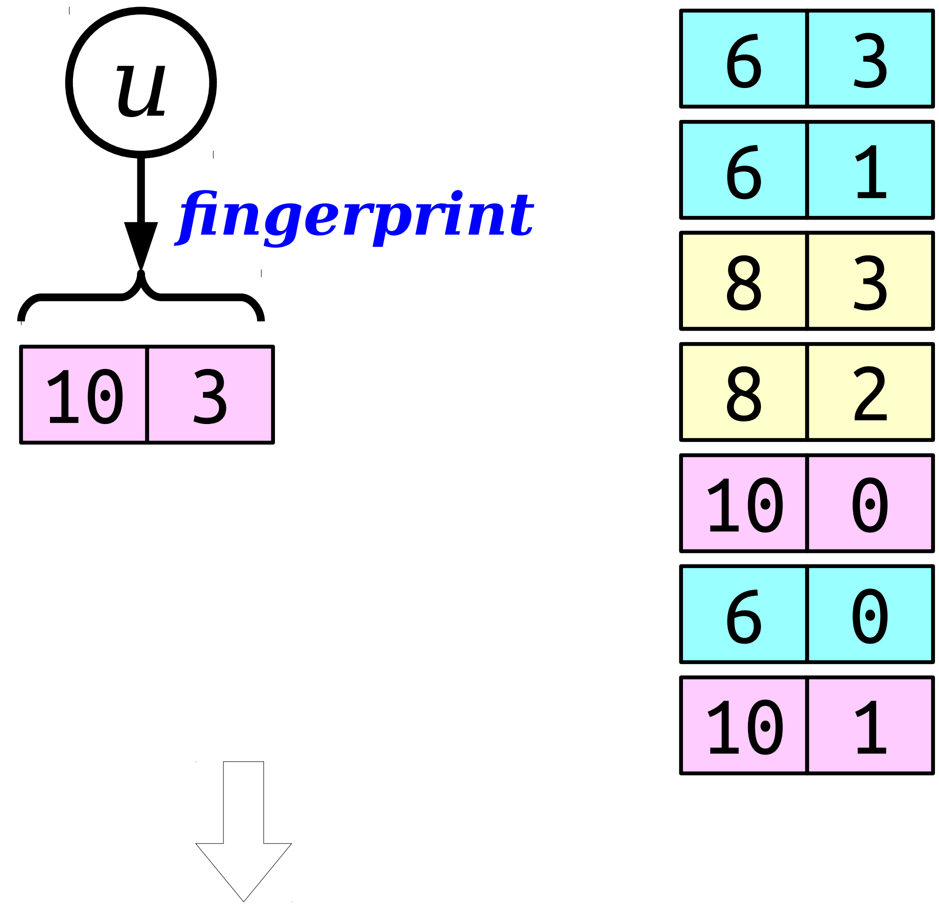
`100` means that we're at the start of a cluster (something is here, this doesn't continue a run, and doesn't continue a cluster.)

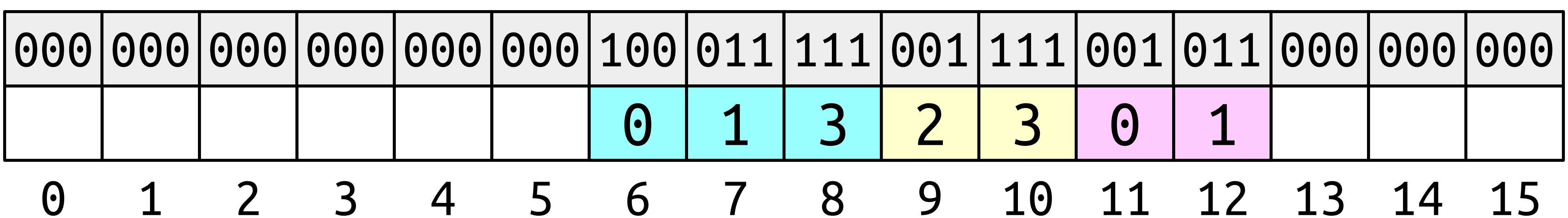
000	000	000	000	000	000	100	011	111	001	111	001	011	000	000	000
						0	1	3	2	3	0	1			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Challenge: Building this compressed data structure is trickier than it seems.

Idea: Add one extra bit to each slot to indicate whether it continues a longer cluster.

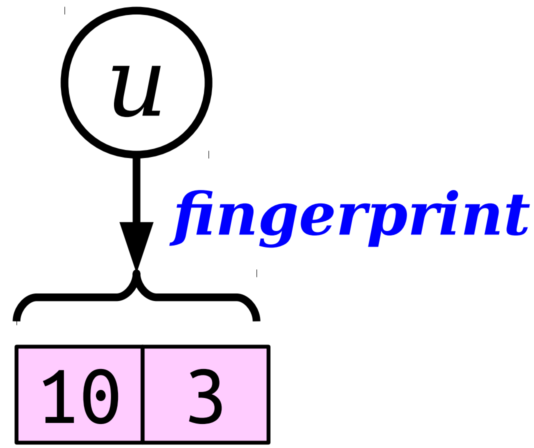
If we hash into the middle of a cluster, scan back until we find the first element.

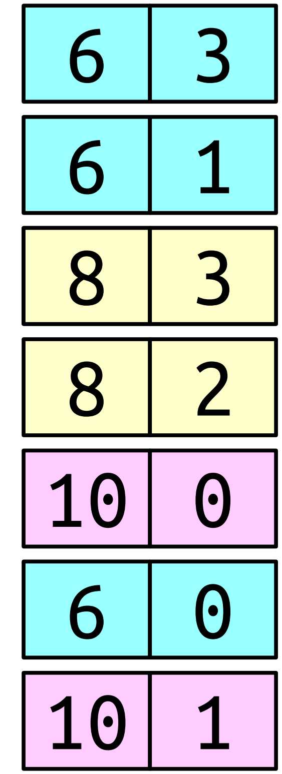




Challenge: Building this compressed data structure is trickier than it seems.

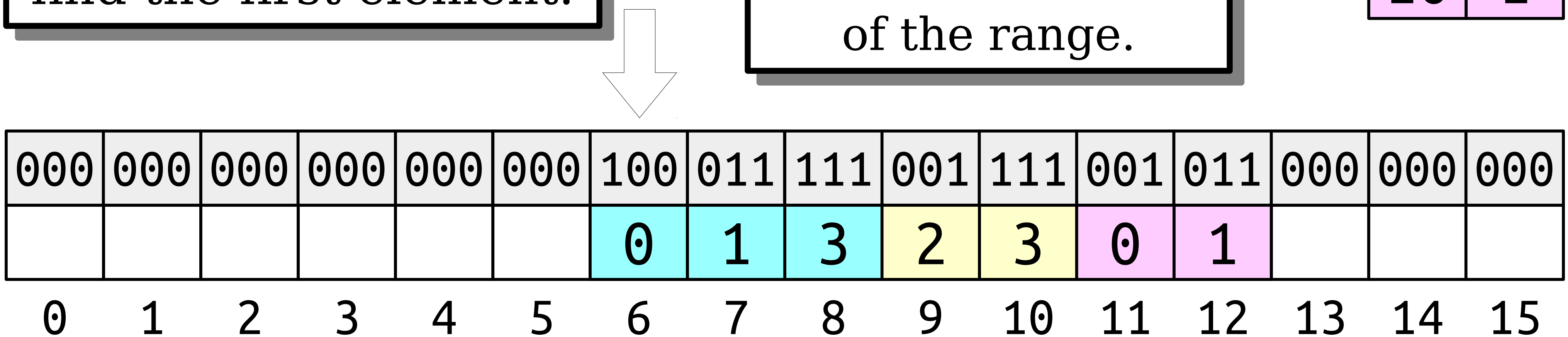
Idea: Add one extra bit to each slot to indicate whether it continues a longer cluster.





If we hash into the middle of a cluster, scan back until we find the first element.

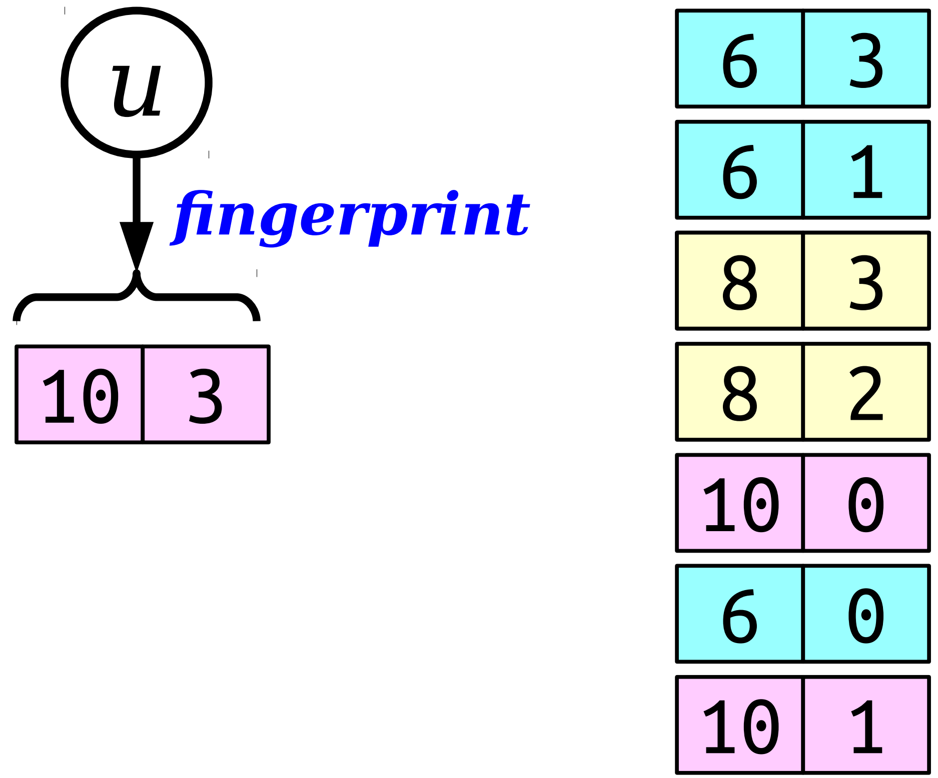
The 100 control bit pattern indicates we’re at the start of the range.



Challenge: Building this compressed data structure is trickier than it seems.

Idea: Add one extra bit to each slot to indicate whether it continues a longer cluster.

If we hash into the middle of a cluster, scan back until we find the first element.



000	000	000	000	000	000	100	011	111	001	111	001	011	000	000	000
						0	1	3	2	3	0	1			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

This data structure is called a ***quotient filter***.

Space usage:

$n \cdot (\lg \varepsilon^{-1} + 3) = n \lg \varepsilon^{-1} + 3n =$ $\boldsymbol{\Theta(n \lg \varepsilon^{-1})}$.
For small ε, this is better than a Bloom filter.
(Might make table size of αn to speed up searches.)

Cost of a query:

Fingerprinting takes times O(1), lookup in linear probing table takes expected time O(1).

Query time: expected **O(1)**, which (on expectation) beats the Bloom filter!

6	3
6	1
8	3
8	2
10	0
6	0
10	1

000	000	000	000	000	000	100	011	111	001	111	001	011	000	000	000
						0	1	3	2	3	0	1			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

This data structure is called a ***quotient filter***.

Other Advantages:

Supports dynamic insertions and deletions; Bloom filters can insert but not delete.

Excellent locality of reference. True fact: the title of the paper introducing quotient filters is "Don't Thrash: How to Cache Your Hash in Flash."

More recent: this was developed in 2012!

Generalizes: there's a ***cuckoo filter*** based on similar principles introduced in 2014.

6	3
6	1
8	3
8	2
10	0
6	0
10	1

000	000	000	000	000	000	100	011	111	001	111	001	011	000	000	000
						0	1	3	2	3	0	1			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

This data structure is called a ***quotient filter***.

- Bloom filters use $\Theta(n \log \varepsilon^{-1})$ bits $\Theta(\log \varepsilon^{-1})$ hash functions to give a false positive rate of ε.
- They're extremely simple to implement and are used extensively in practice.
- They work by computing and lossily storing fingerprints.
- The cost of a query depends on the number of hashes and grows with $\Theta(\log \varepsilon^{-1})$.
- Any AMQ structure must use at least $n \lg \varepsilon^{-1}$ bits. Bloom filters are close to this bound.

- We can build other AMQ structures by hashing to a space of size $n \cdot \varepsilon^{-1}$ and storing the fingerprints.
- Breaking a fingerprint of $\lg (n \cdot \varepsilon^{-1})$ into an index and a residual hash allows them to be stored at an average of $\Theta(\log \varepsilon^{-1})$ space each.
- Quotient filtering is a variant of linear probing for storing residual hashes.
- Quotient filters use extra control bits to store residuals with no false positives.

To summarize...

More to Explore

- In 2005, Pagh, Pagh, and Rao designed a data structure that uses

$$(1 + o(1))(n \lg \varepsilon^{-1}) + O(n + w)$$

  bits to solve AMQ on a machine with word size *w*. For large *n*, this is essentially optimal.
- Your classmates will be presenting this structure as part of a final project if you're curious to see how it works!

More to Explore

- In 2014, Fan et al introduced the ***cuckoo filter***, which uses a compressed version of cuckoo hashing to store signatures.
- The analysis of cuckoo filtering is still an open problem! It looks like it does really well in practice, but we aren't sure why.

More to Explore

- The idea of counting the number of bits used to store a structure is related to the idea of ***succinct data structures***, which aim to minimize the number of bits needed to encode a data structure.
- We know how to encode binary trees, binary tries, and priority queues this way.
- We're working on suffix trees, suffix arrays, etc., and this is an active area of research!

Next Time

- ***Integer Data Structures***
  - Speeding things up by harnessing machine words.
- ***x-Fast Tries***
  - Tries + cuckoo hashing + machine words.
- ***y-Fast Tries***
  - *x*-Fast tries + macro/micro + amortization + balanced trees.