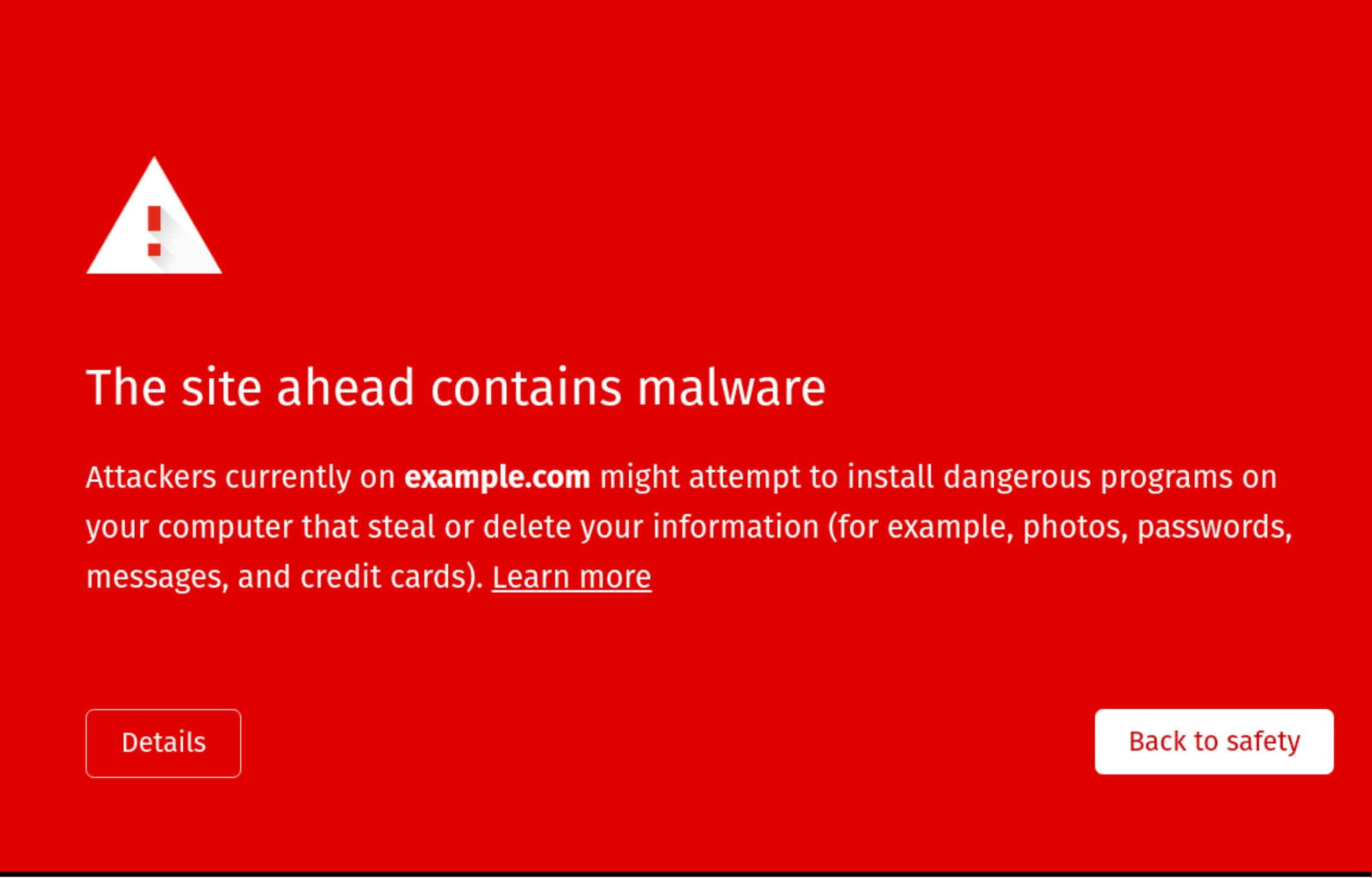
Approximate Membership Queries

Outline for Today

- ***Approximate Membership Queries***
  - Storing sets… sorta.
- ***Bloom Filters***
  - The original approximate membership query structure.
- ***Cuckoo Filters***
  - Beating Bloom filters with modern techniques.
- ***XOR Filters***
  - (Also) beating Bloom filters with modern techniques.

Where We’re Going



Web browsers can store a list of malicious URL domains using ***one byte per URL***, guaranteeing any bad URL will be flagged, with a false positive rate of <1%. ***How is this possible?***

Every gun that is made, every warship launched, every rokcet fired signifies, in the final sense, a theft from those who hunger and are not fed, those who are cold and are not clothed. This world in arms is not spending money alone. It is spending the sweat of its laborers, the genuis of its scientists, the hopes of its childen. The cost of one modern heavy bomber is this: a modern brick school in more than 30 cities. It is two electric power plants, each serving a town of 60,000 population. It is two fine, fully equipped hospitals. It is some 50 miles of concrete highway. We pay for a single fighter plane with a half millon bushels of wheat. We pay for a single destroyer with new homes that could have housed more than 8,000 people. This, I repeat, is the best way of life to be found on the road the world has been takig. This is not a way of life at all, in any true sense. Under the cloud of threatening war, it is humanity hanging from a cross of iron.

Spellcheckers can store a list of all words in English using ***one byte per word***, never flagging a correctly-spelled word, and flagging >99% of mispeled words. ***How is this possible?***

Approximate Membership Queries

Exact Membership Queries

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

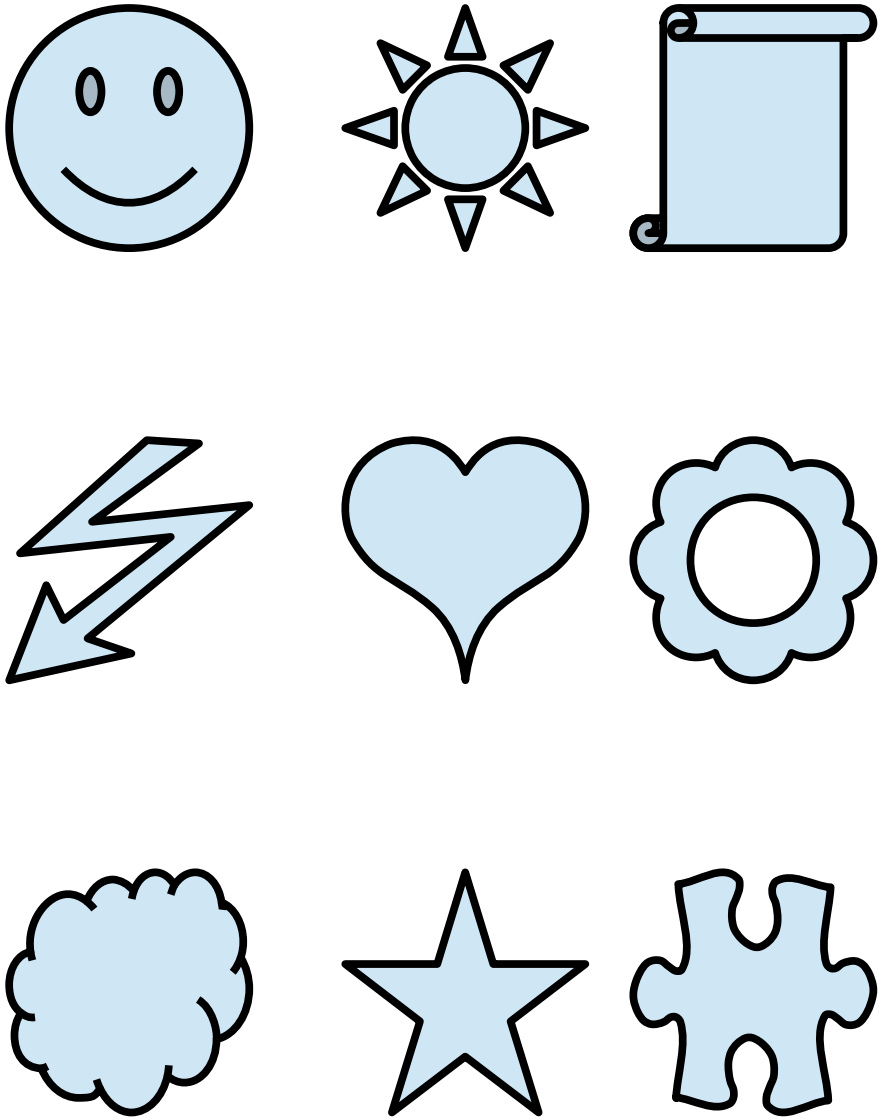
Maintain a set S in a way that supports queries of the form "is x ∈ S?"

- It's not hard to solve this problem by storing all the elements in the set (e.g. hash table, balanced BST, etc.)

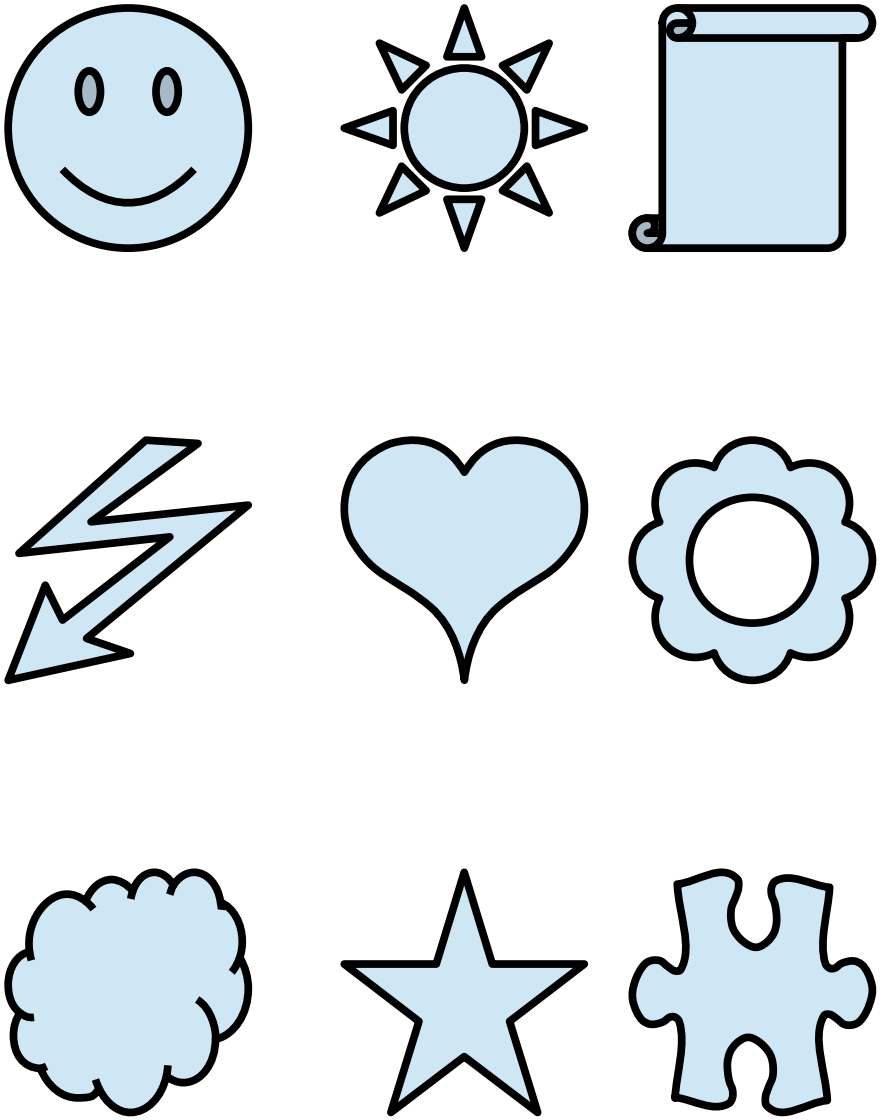
Exact Membership Queries

- 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 much memory is needed to solve the exact membership query problem?

A Quick Detour

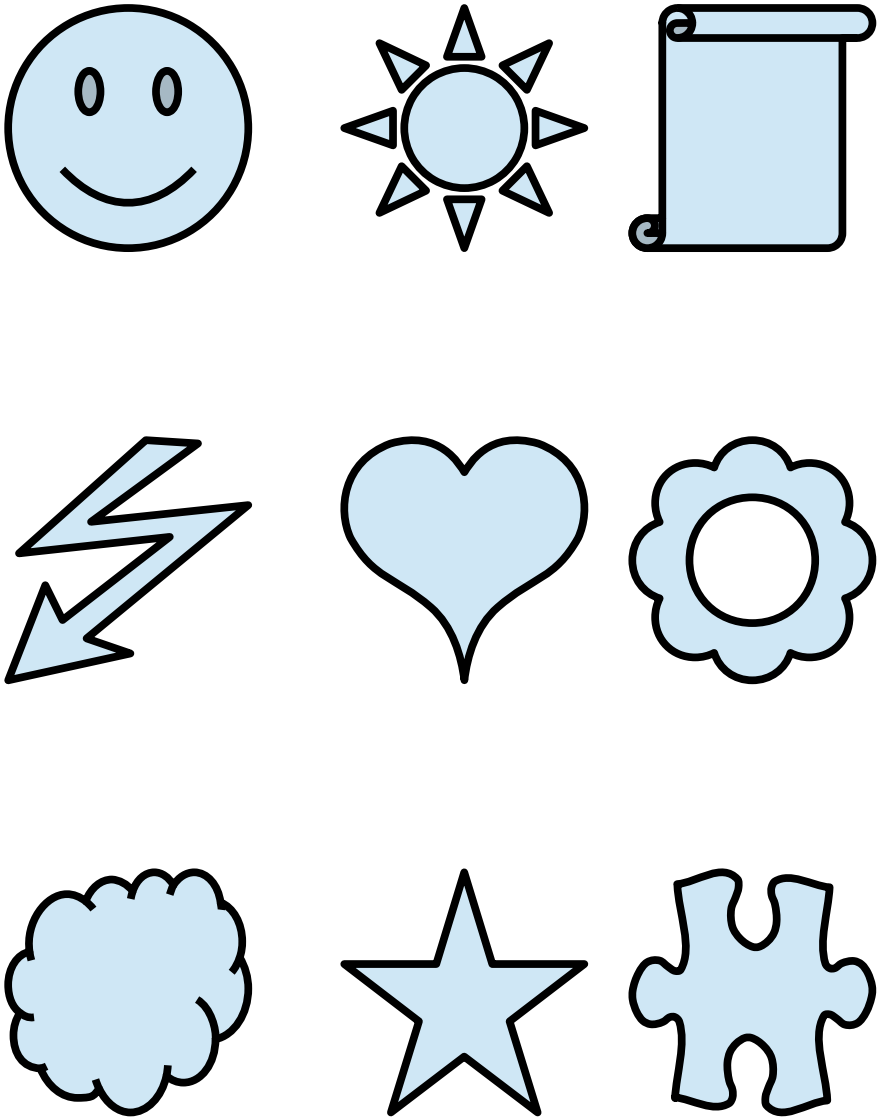


Goal: Design a simple data structure that can hold a single one of the objects shown to the right.



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

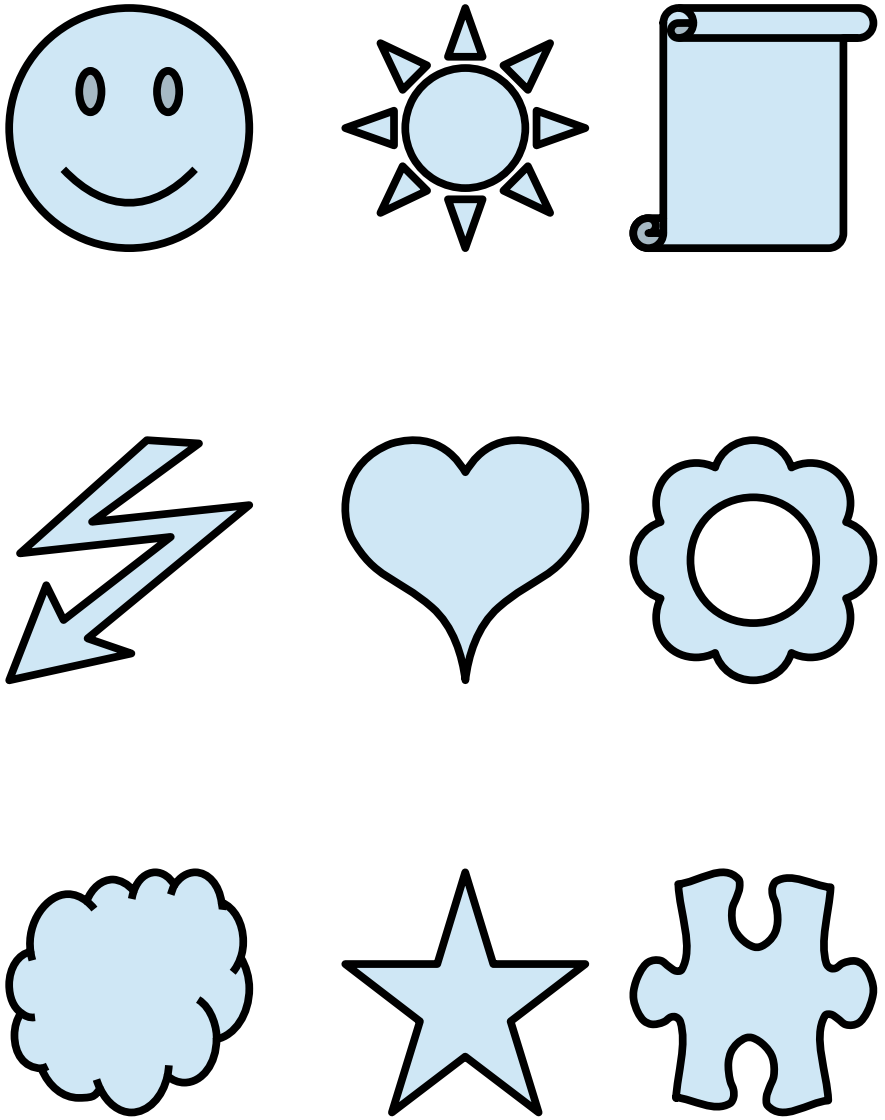
What is the minimum number of *bits* (not *words*) required for this data structure in the worst case?



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?

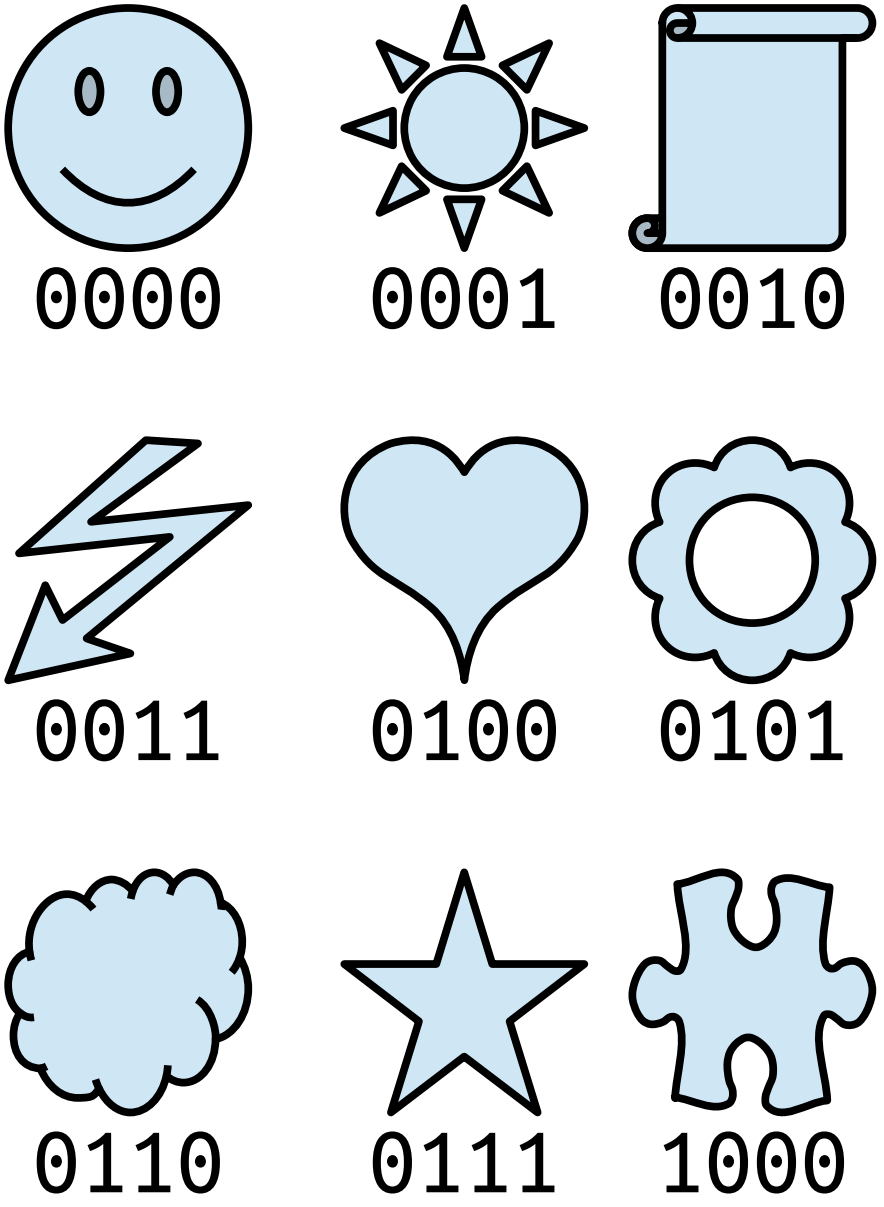
We can get away with four bits by numbering each item and just storing the number.



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.

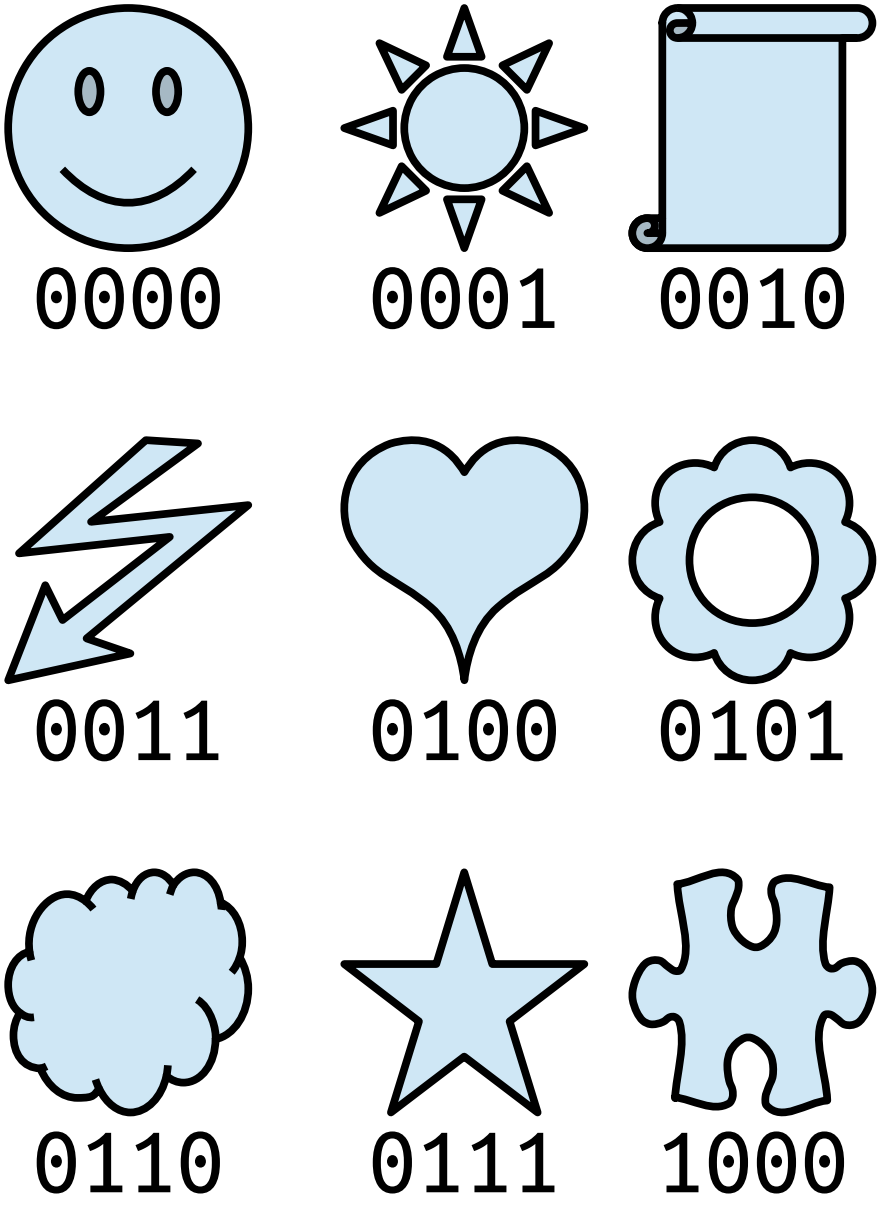


Goal: Design a simple data structure that can hold a single one of the objects shown to the right.

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.

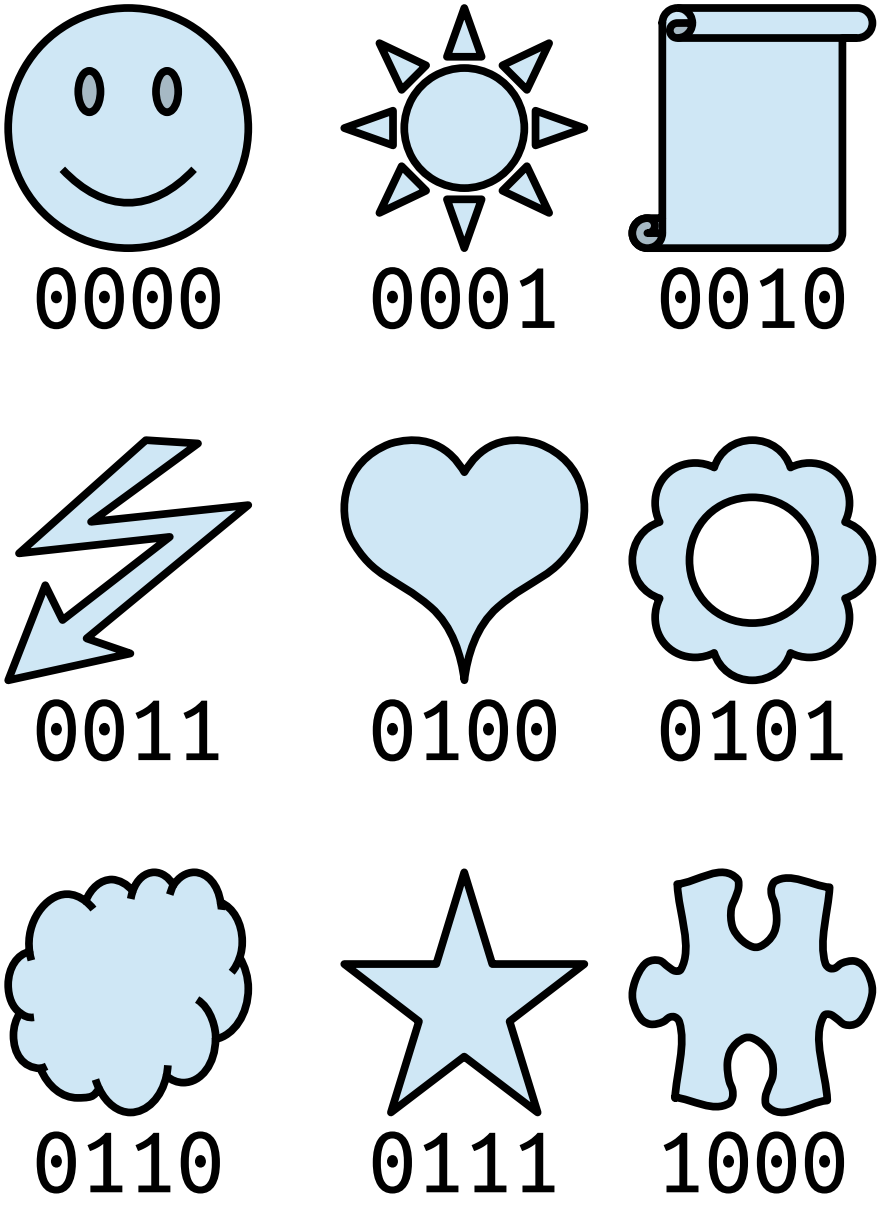
Question: Can we do better?



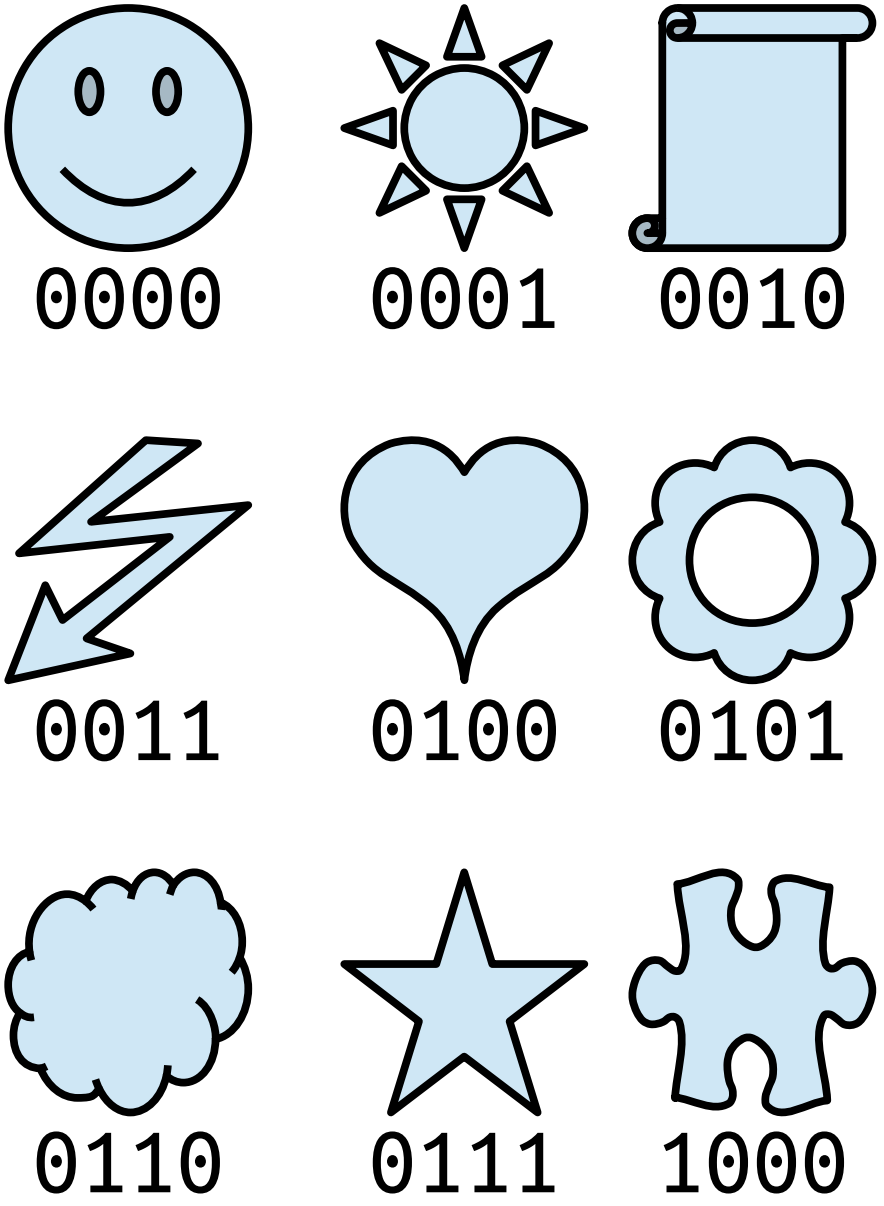
Goal: Design a simple data structure that can hold a single one of the objects shown to the right.

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.

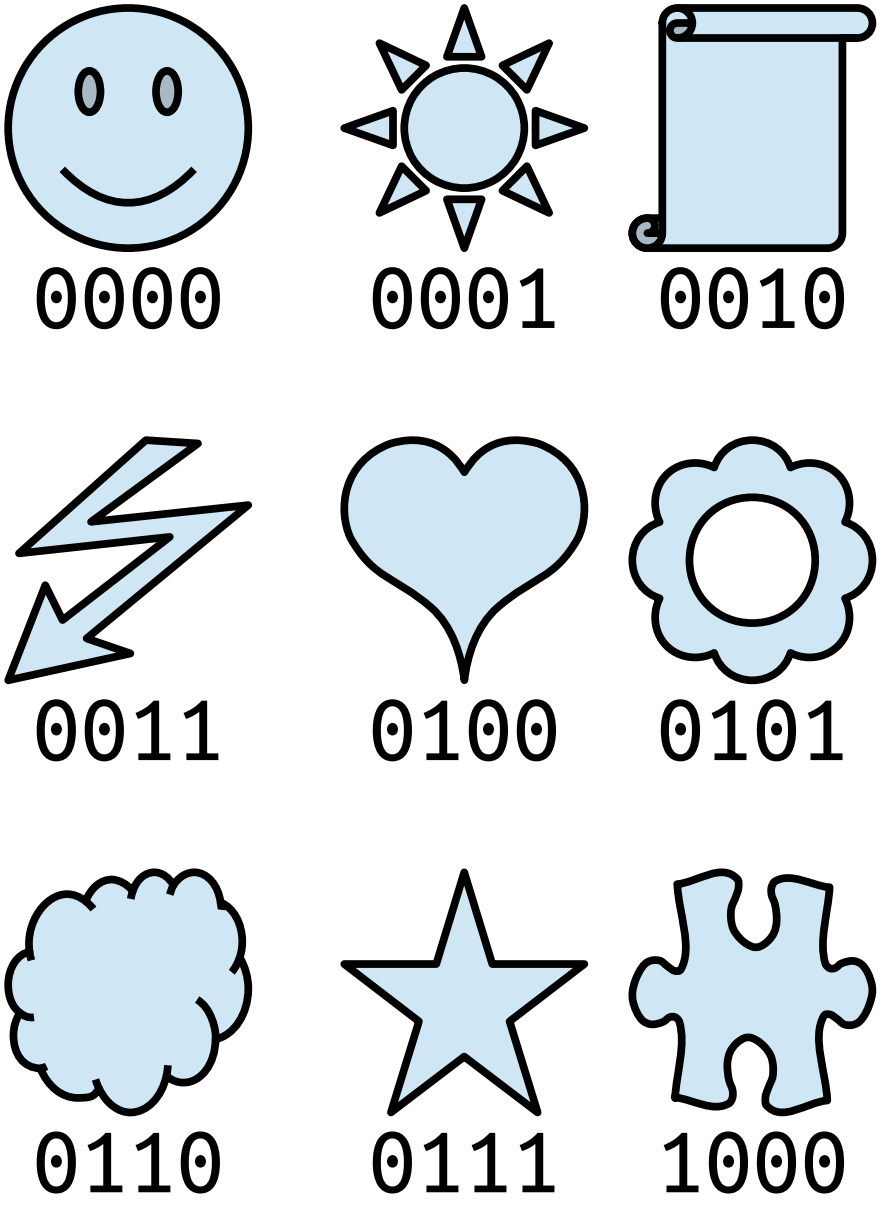


Theorem: A data structure that stores one object out of a set of k possibilities must use at least lg k bits in the worst case.



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?

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 $n \ll U$. How many bits of memory do we need?

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 $n \ll U$. How many bits of memory do we need?

Number of n-element subsets of universe U:

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

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 $n \ll U$. How many bits of memory do we need?

Number of n-element subsets of universe U:

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

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

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 $n \ll U$. How many bits of memory do we need?

Number of n-element subsets of universe U:

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

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

$$= \quad \lg \left(\frac{|U|\,!}{n\,!\,(|U|-n)\,!} \right)$$

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 $n \ll U$. How many bits of memory do we need?

Number of n-element subsets of universe U:

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

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

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 $n \ll U$. How many bits of memory do we need?

Number of n-element subsets of universe U:

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

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

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 $n \ll U$. How many bits of memory do we need?

Number of n-element subsets of universe U:

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

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

Bitten by Bits

- Solving the exact membership query problem requires approximately $\boldsymbol{n \lg |U|}$ bits of memory in the worst case, assuming $|U| \gg n$.
- 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

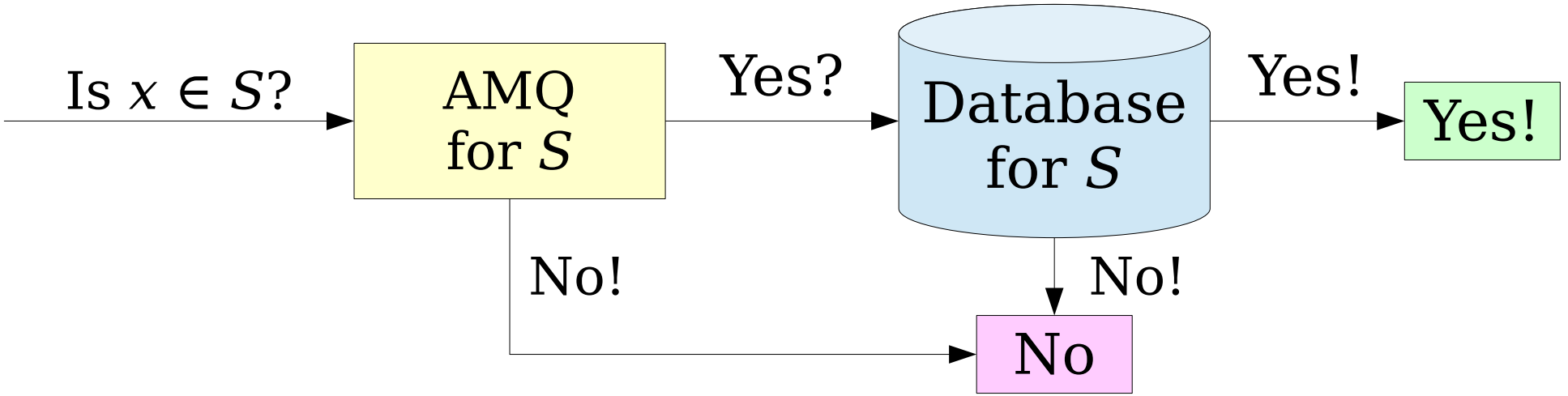
- The ***approximate membership query*** problem is the following:

  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?
- Let's address each of these in turn.

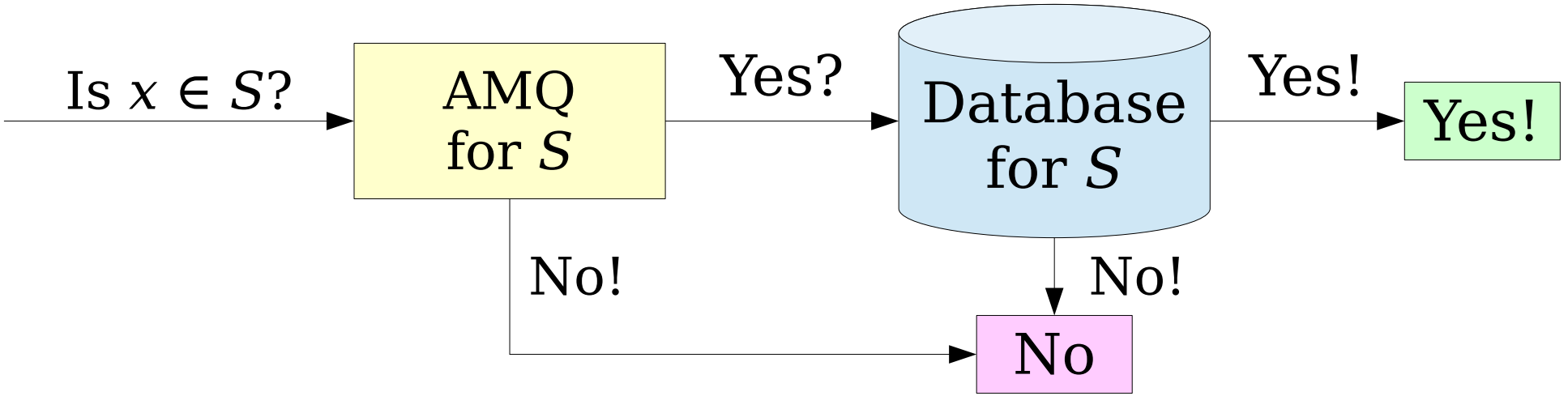
Our Model

- ***Goal:*** Design our data structures to allow for false positives (incorrectly stating that $x \in S$) but not false negatives (incorrectly stating that $x \notin S$).
- 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

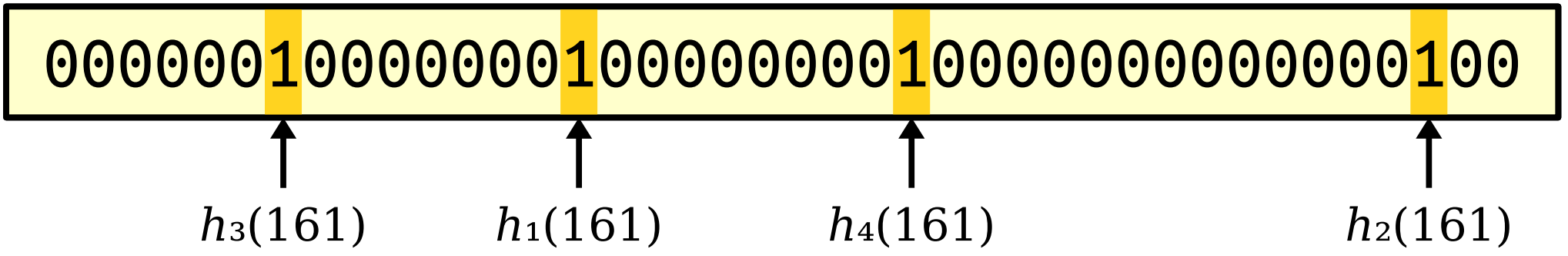
- Assume we have a user-provided ***accuracy*** parameter $\varepsilon \in (0, 1)$ and a set $S \subseteq U$ of size $n \ll |U|$.
- ***Goal:*** approximate S so that
  - if we query about an $x \in S$, we always return true (no ***false negatives***);
  - if we query about an $x \notin S$, we return false with probability $1 - \varepsilon$ (we allow for ***false positives***); and
  - Our space usage depends only on n and ε, not on the size of the universe.
- ***Question:*** Is this even possible?



Bloom Filters

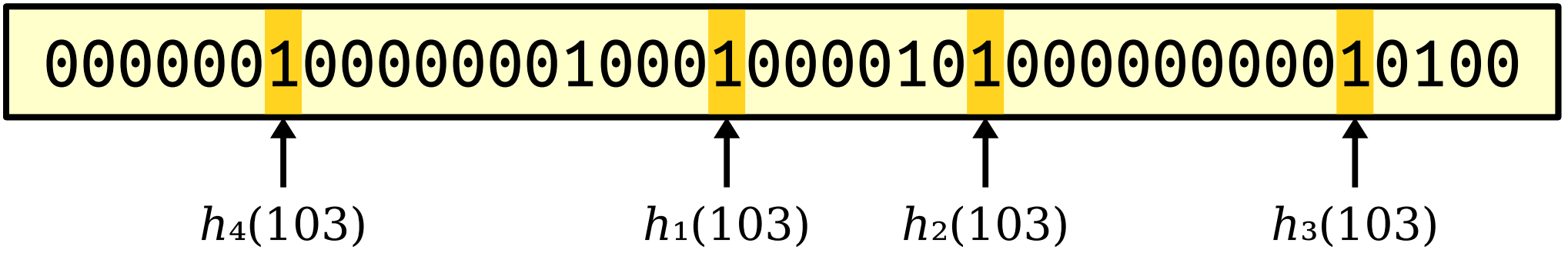
The Bloom Filter

- To approximately store a set of n elements, create an array of m bits, all initially zero.
- Choose d hash functions from U to $[m]$.
- Hash each of the n items to store with the hash functions, setting all indicated bits to 1.



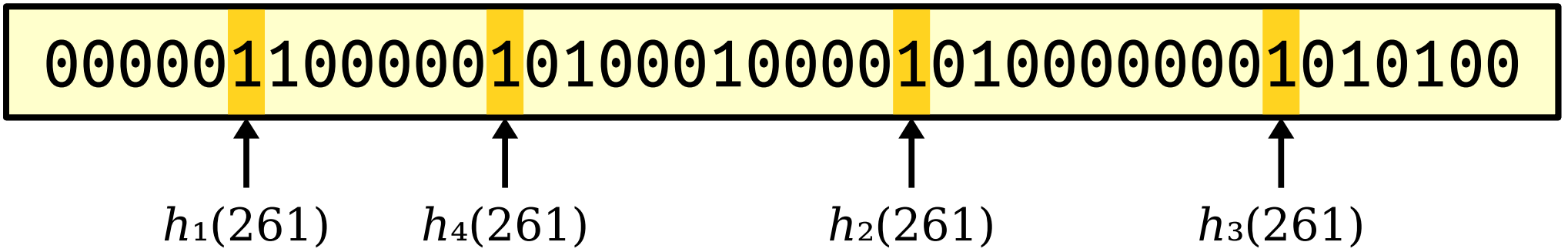
The Bloom Filter

- To approximately store a set of n elements, create an array of m bits, all initially zero.
- Choose d hash functions from U to $[m]$.
- Hash each of the n items to store with the hash functions, setting all indicated bits to 1.



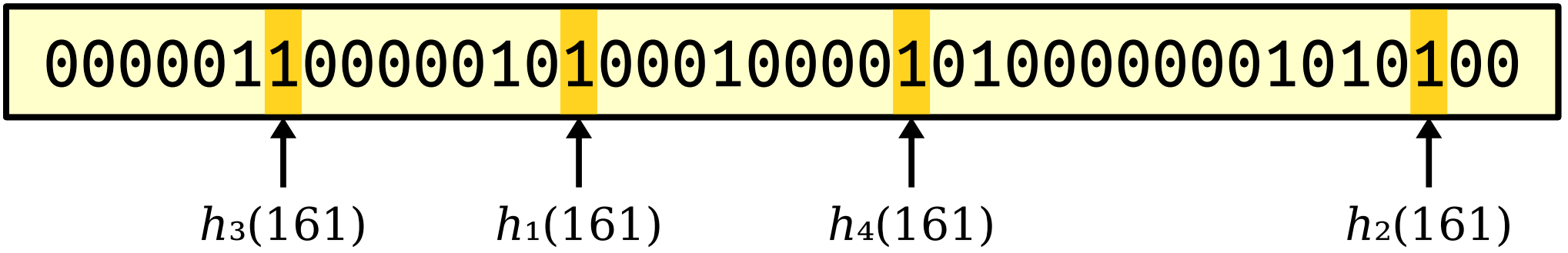
The Bloom Filter

- To approximately store a set of n elements, create an array of m bits, all initially zero.
- Choose d hash functions from U to $[m]$.
- Hash each of the n items to store with the hash functions, setting all indicated bits to 1.



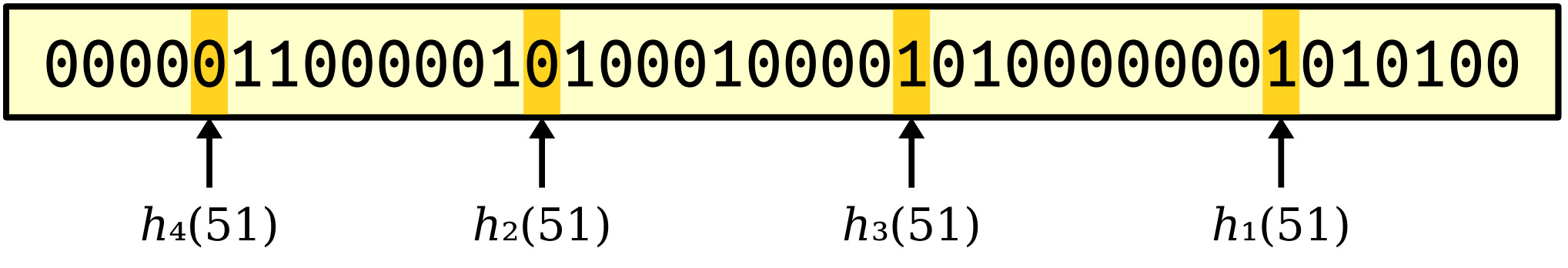
The Bloom Filter

- To approximately store a set of n elements, create an array of m bits, all initially zero.
- Choose d hash functions from U to $[m]$.
- Hash each of the n items to store with the hash functions, setting all indicated bits to 1.
- To see if x is in the set, hash x with all d hash functions to get a set of bits to test, then return true if they're all set to 1 and false otherwise.
- Any item that was added will always appear present; items that were not added may produce false positives.



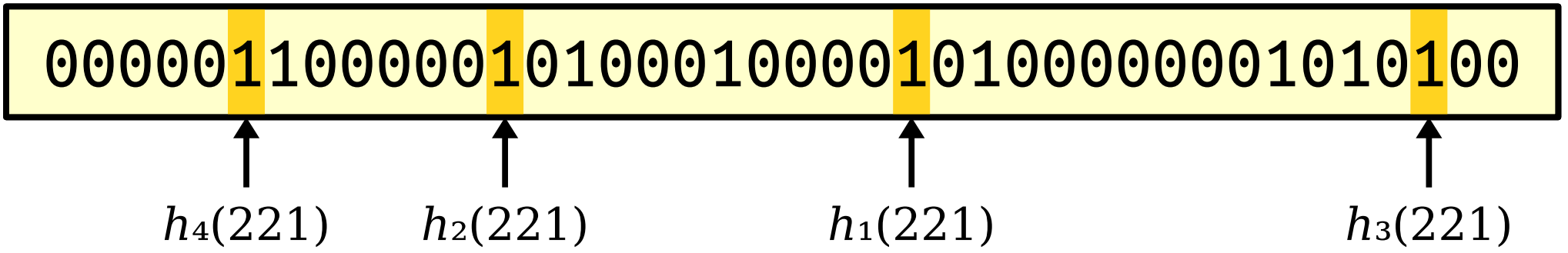
The Bloom Filter

- To approximately store a set of n elements, create an array of m bits, all initially zero.
- Choose d hash functions from U to $[m]$.
- Hash each of the n items to store with the hash functions, setting all indicated bits to 1.
- To see if x is in the set, hash x with all d hash functions to get a set of bits to test, then return true if they're all set to 1 and false otherwise.
- Any item that was added will always appear present; items that were not added may produce false positives.



The Bloom Filter

- To approximately store a set of n elements, create an array of m bits, all initially zero.
- Choose d hash functions from U to $[m]$.
- Hash each of the n items to store with the hash functions, setting all indicated bits to 1.
- To see if x is in the set, hash x with all d hash functions to get a set of bits to test, then return true if they're all set to 1 and false otherwise.
- Any item that was added will always appear present; items that were not added may produce false positives.



The Bloom Filter

- To achieve a false positive rate of ε using as few bits as possible, we pick

  $$\boldsymbol{d = \lg \varepsilon^{-1}}$$

  hash functions and make an array of

  $$m = \lg e \, (n \lg \varepsilon^{-1}) \approx \mathbf{1.44 \, n \lg \varepsilon^{-1}}$$

  bits.
- There are lots of great sources online containing the derivation of these numbers; I recommend taking a few minutes to read through one of them.

Looking Forward

- As always:

Can we do better?

- To improve on the Bloom filter, we can either make
  - improvements to the query time, or
  - improvements to the space usage.
- How might we do this?

	Bits Per Element	Hashes Per Query
Bloom Filter	$1.44 \lg \varepsilon^{-1}$	$\lg \varepsilon^{-1}$

Looking Forward

- As always:

Can we do better?

- To improve on the Bloom filter, we can either make
  - improvements to the query time, or
  - improvements to the space usage.
- How might we do this?

	Bits Per Element	Hashes Per Query
Bloom Filter	$1.44 \lg \varepsilon^{-1}$	$\lg \varepsilon^{-1}$

Earlier, we saw that storing n elements from a universe U requires at least $\boldsymbol{n \lg |U|}$ bits, assuming $|U| \gg n$.

That bound doesn't apply to us, since that isn't what we're doing here.

Can we get a lower bound on the number of bits needed?

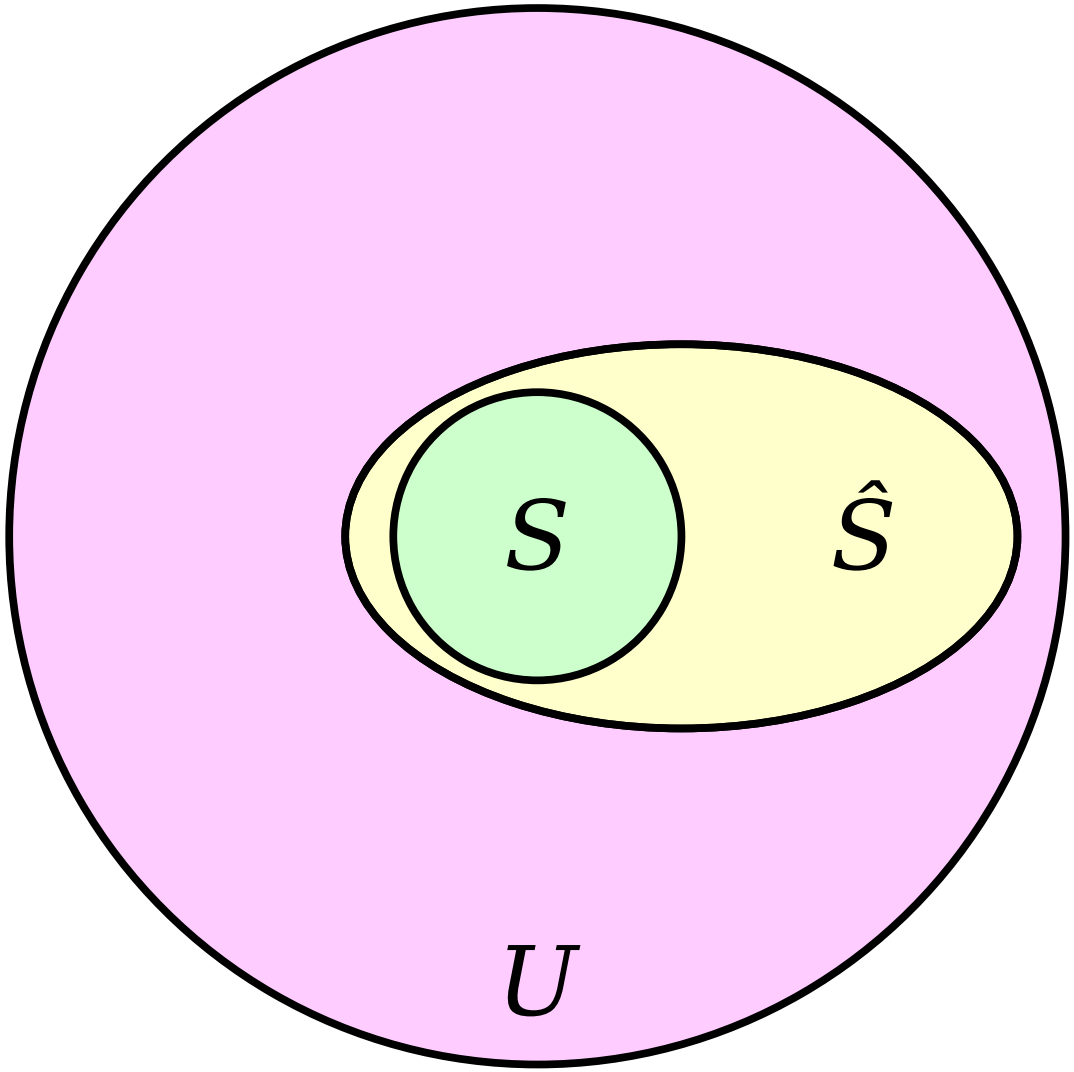
How much memory is needed to solve the approximate membership query problem?

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

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

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.

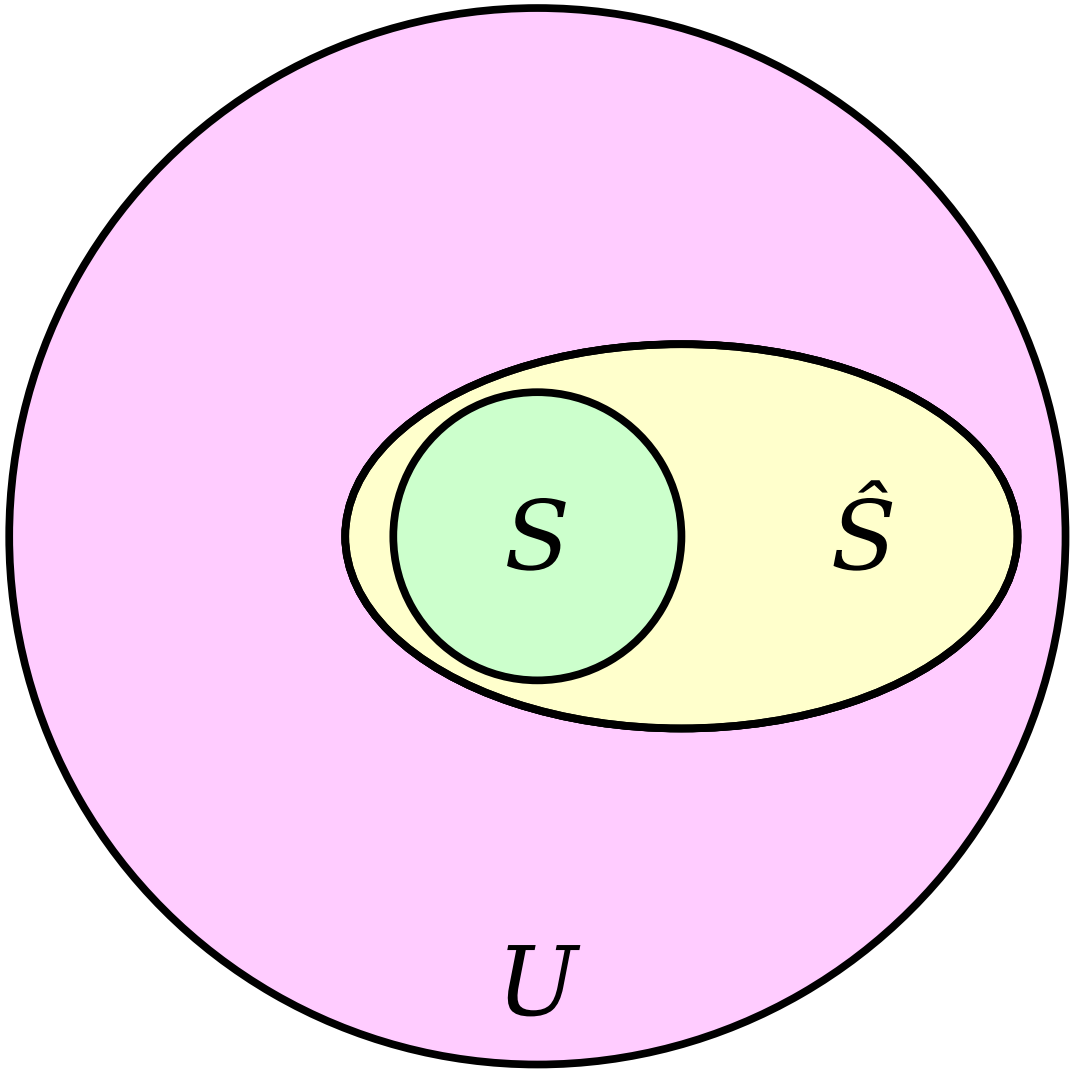


How much memory is needed to solve the approximate membership query problem?

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.

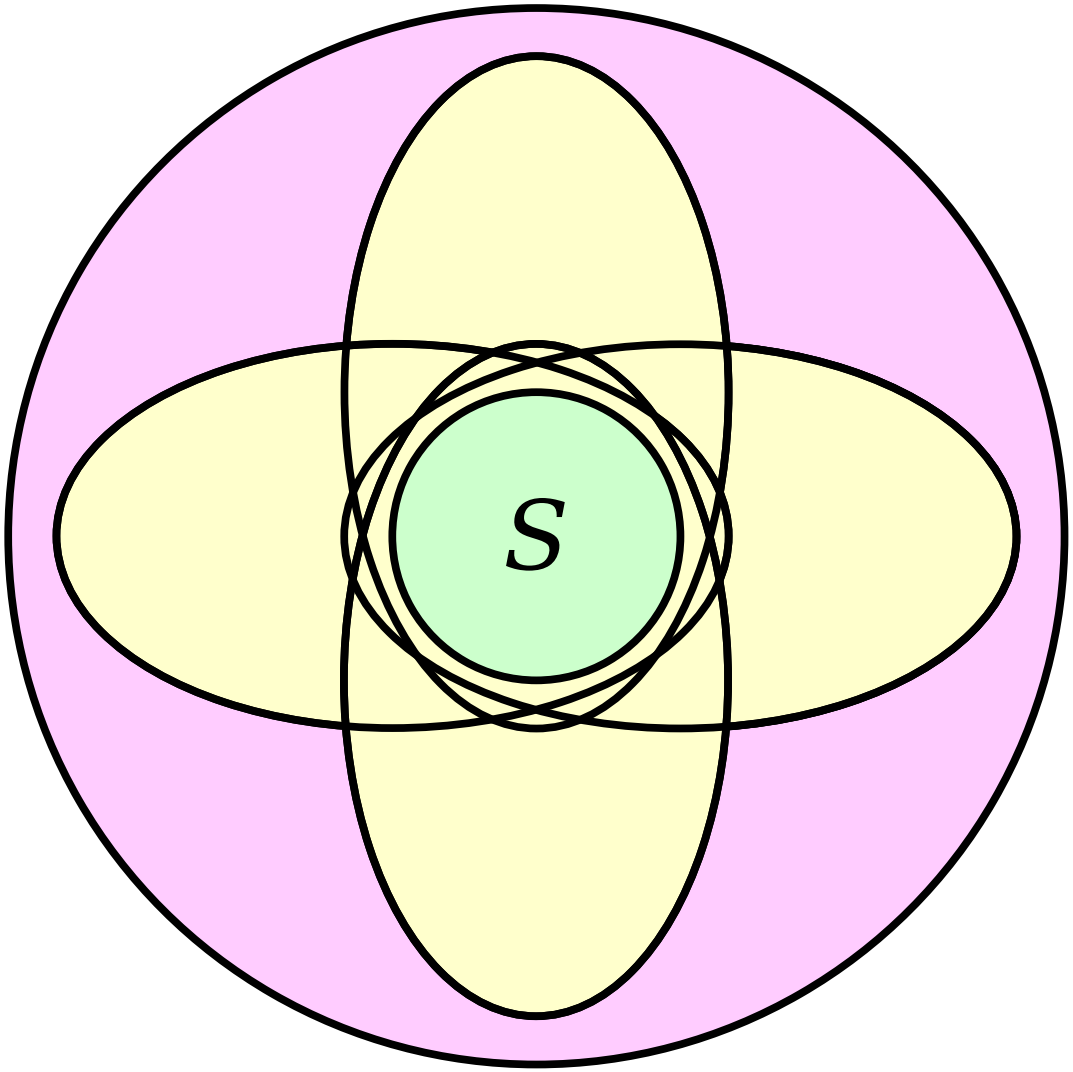


How much memory is needed to solve the approximate membership query problem?

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.

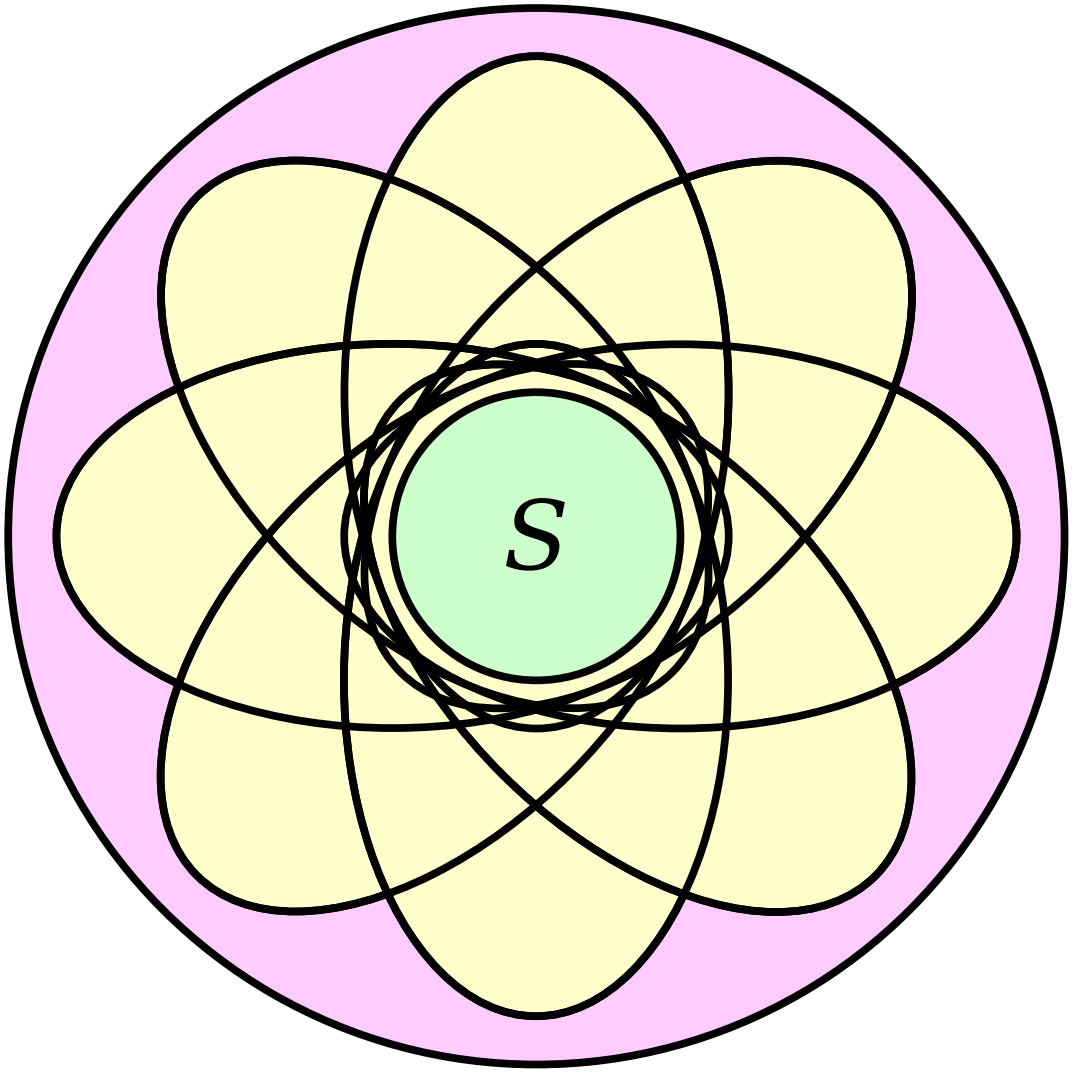


How much memory is needed to solve the approximate membership query problem?

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.



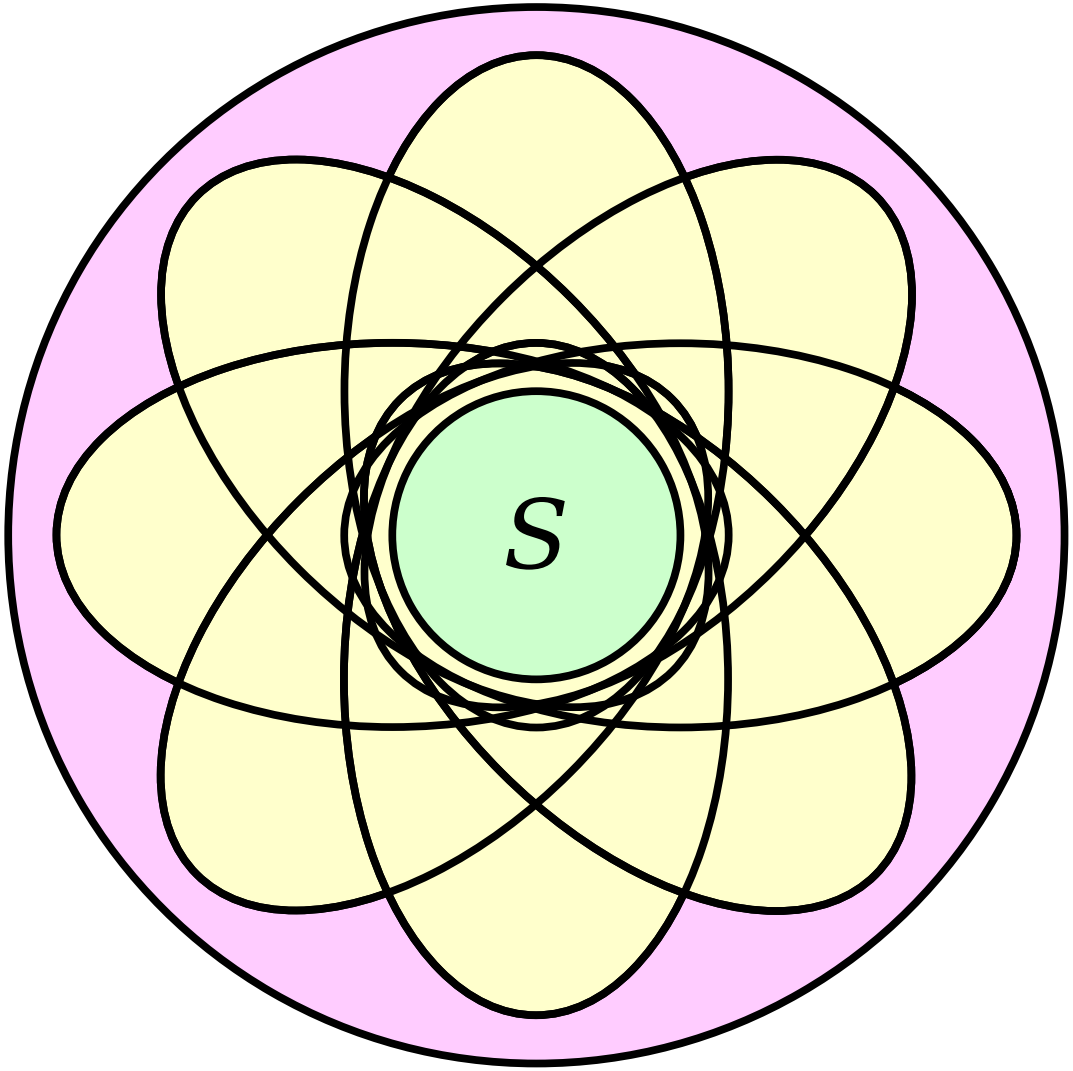
How much memory is needed to solve the approximate membership query problem?

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?

Clever Idea: We can *exactly* describe a set S of size n using an AMQ, plus some extra bits.

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

Clever Idea: We can *exactly* describe a set S of size n using an AMQ, plus some extra bits.

First, write down an AMQ for S with error rate ε. Assume this needs b bits.

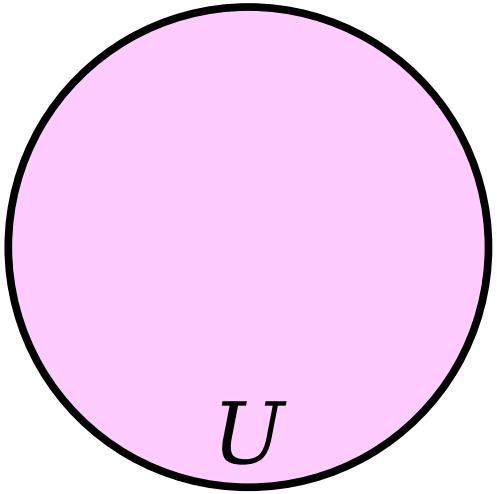
This AMQ encodes a set $\hat{S}$ of size roughly $\varepsilon|U|$ containing our set S.

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

Clever Idea: We can *exactly* describe a set S of size n using an AMQ, plus some extra bits.

First, write down an AMQ for S with error rate ε. Assume this needs b bits.

This AMQ encodes a set $\hat{S}$ of size roughly $\varepsilon|U|$ containing our set S.

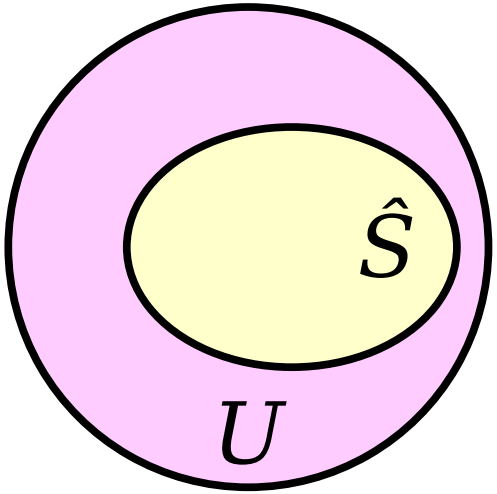


How much memory is needed to solve the approximate membership query problem?

Clever Idea: We can *exactly* describe a set S of size n using an AMQ, plus some extra bits.

First, write down an AMQ for S with error rate ε. Assume this needs b bits.

This AMQ encodes a set $\hat{S}$ of size roughly ε|U| containing our set S.

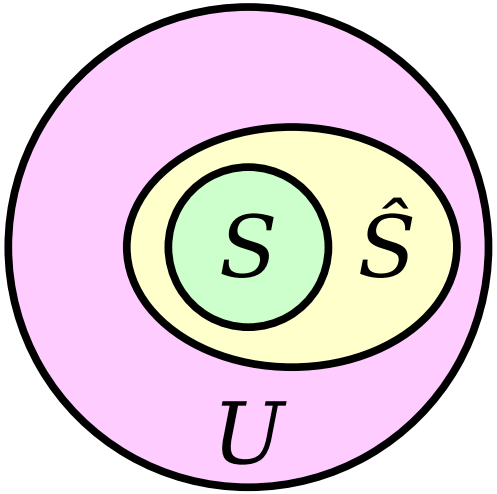


How much memory is needed to solve the approximate membership query problem?

Clever Idea: We can *exactly* describe a set S of size n using an AMQ, plus some extra bits.

First, write down an AMQ for S with error rate ε. Assume this needs b bits.

This AMQ encodes a set $\hat{S}$ of size roughly $\varepsilon|U|$ containing our set S.



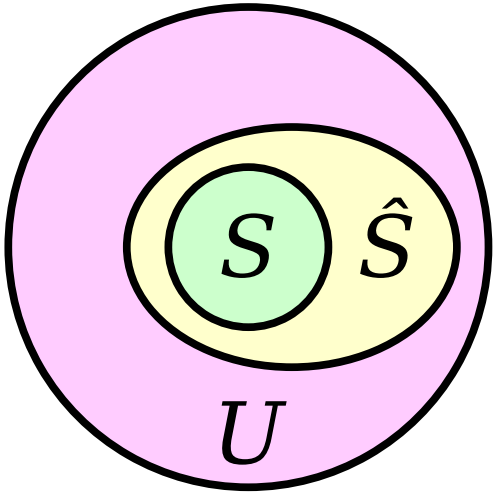
How much memory is needed to solve the approximate membership query problem?

Clever Idea: We can *exactly* describe a set S of size n using an AMQ, plus some extra bits.

First, write down an AMQ for S with error rate ε. Assume this needs b bits.

This AMQ encodes a set $\hat{S}$ of size roughly $\varepsilon|U|$ containing our set S.

To define S, we need to pick n elements from the set $\hat{S}$, which has size $\varepsilon|U|$. This requires $n \lg (\varepsilon|U|)$ bits.



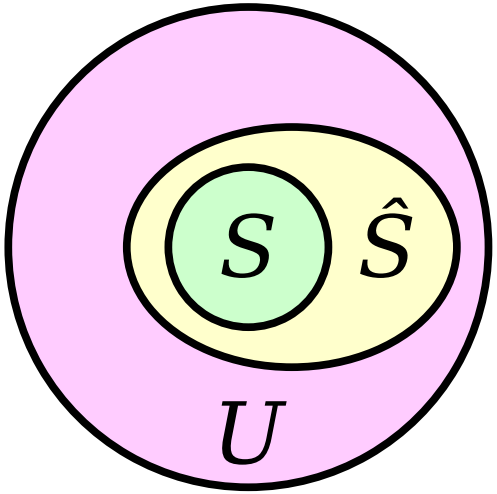
How much memory is needed to solve the approximate membership query problem?

Clever Idea: We can *exactly* describe a set S of size n using an AMQ, plus some extra bits.

First, write down an AMQ for S with error rate ε. Assume this needs b bits.

This AMQ encodes a set $\hat{S}$ of size roughly $\varepsilon|U|$ containing our set S.

To define S, we need to pick n elements from the set $\hat{S}$, which has size $\varepsilon|U|$. This requires $n \lg (\varepsilon|U|)$ bits.



$$b + n \lg (\varepsilon|U|) \geq n \lg |U|$$

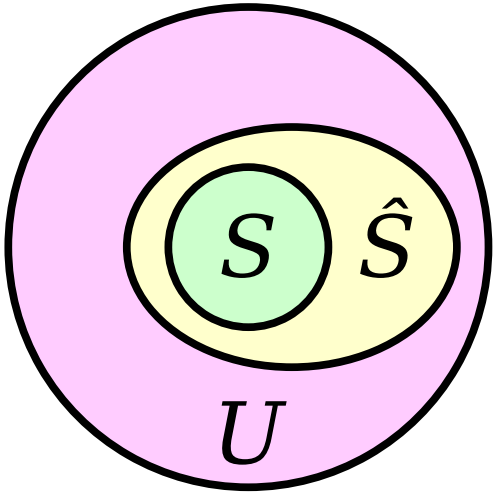
How much memory is needed to solve the approximate membership query problem?

Clever Idea: We can *exactly* describe a set S of size n using an AMQ, plus some extra bits.

First, write down an AMQ for S with error rate ε. Assume this needs b bits.

This AMQ encodes a set $\hat{S}$ of size roughly $\varepsilon|U|$ containing our set S.

To define S, we need to pick n elements from the set $\hat{S}$, which has size $\varepsilon|U|$. This requires $n \lg (\varepsilon|U|)$ bits.



$$b + n \lg (\varepsilon|U|) \geq n \lg |U|$$

Lower bound on any way of picking n items from U

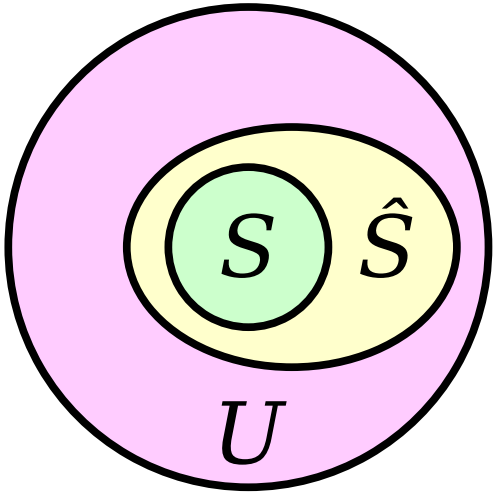
How much memory is needed to solve the approximate membership query problem?

Clever Idea: We can *exactly* describe a set S of size n using an AMQ, plus some extra bits.

First, write down an AMQ for S with error rate ε. Assume this needs b bits.

This AMQ encodes a set $\hat{S}$ of size roughly $\varepsilon|U|$ containing our set S.

To define S, we need to pick n elements from the set $\hat{S}$, which has size $\varepsilon|U|$. This requires $n \lg (\varepsilon|U|)$ bits.



$$b + n \lg (\varepsilon|U|) \geq n \lg |U|$$

Bits needed to pick n items from $\hat{S}$

Lower bound on any way of picking n items from U

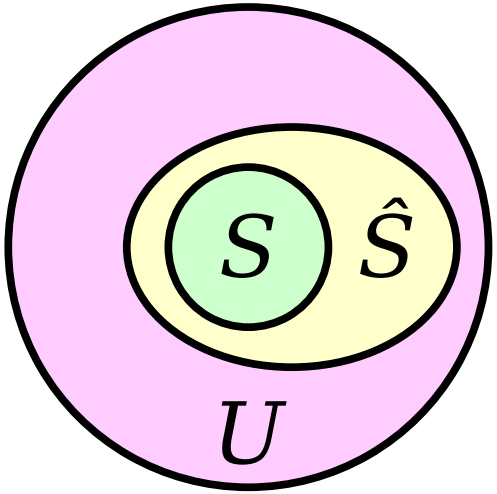
How much memory is needed to solve the approximate membership query problem?

Clever Idea: We can *exactly* describe a set S of size n using an AMQ, plus some extra bits.

First, write down an AMQ for S with error rate ε. Assume this needs b bits.

This AMQ encodes a set $\hat{S}$ of size roughly $\varepsilon|U|$ containing our set S.

To define S, we need to pick n elements from the set $\hat{S}$, which has size $\varepsilon|U|$. This requires $n \lg (\varepsilon|U|)$ bits.



Bits to store the AMQ structure

$$b + n \lg (\varepsilon|U|) \geq n \lg |U|$$

Bits needed to pick n items from $\hat{S}$

Lower bound on any way of picking n items from U

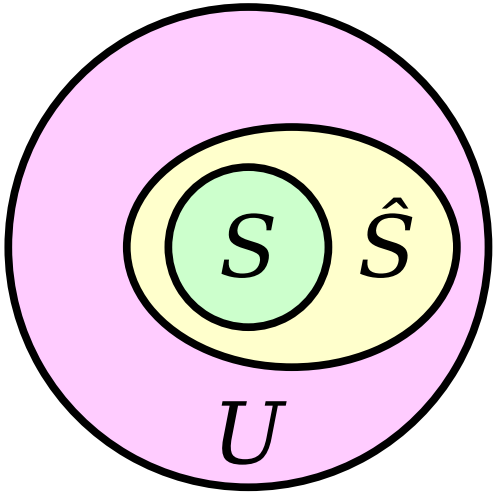
How much memory is needed to solve the approximate membership query problem?

Clever Idea: We can *exactly* describe a set S of size n using an AMQ, plus some extra bits.

First, write down an AMQ for S with error rate ε. Assume this needs b bits.

This AMQ encodes a set $\hat{S}$ of size roughly $\varepsilon|U|$ containing our set S.

To define S, we need to pick n elements from the set $\hat{S}$, which has size $\varepsilon|U|$. This requires $n \lg (\varepsilon|U|)$ bits.



$$b + n \lg (\varepsilon|U|) \geq n \lg |U|$$

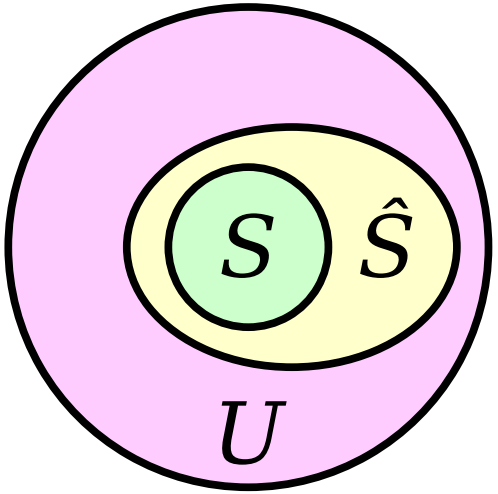
How much memory is needed to solve the approximate membership query problem?

Clever Idea: We can *exactly* describe a set S of size n using an AMQ, plus some extra bits.

First, write down an AMQ for S with error rate ε. Assume this needs b bits.

This AMQ encodes a set $\hat{S}$ of size roughly $\varepsilon|U|$ containing our set S.

To define S, we need to pick n elements from the set $\hat{S}$, which has size $\varepsilon|U|$. This requires $n \lg (\varepsilon|U|)$ bits.



$$b \geq n \lg |U| - n \lg (\varepsilon|U|)$$

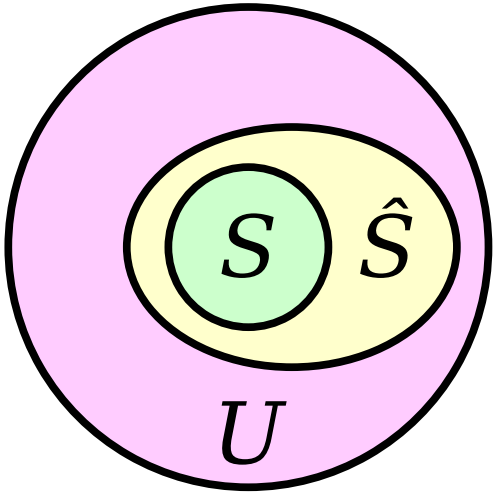
How much memory is needed to solve the approximate membership query problem?

Clever Idea: We can *exactly* describe a set S of size n using an AMQ, plus some extra bits.

First, write down an AMQ for S with error rate ε. Assume this needs b bits.

This AMQ encodes a set $\hat{S}$ of size roughly $\varepsilon|U|$ containing our set S.

To define S, we need to pick n elements from the set $\hat{S}$, which has size $\varepsilon|U|$. This requires $n \lg (\varepsilon|U|)$ bits.



$$b \geq n \left(\lg |U| - \lg (\varepsilon|U|)\right)$$

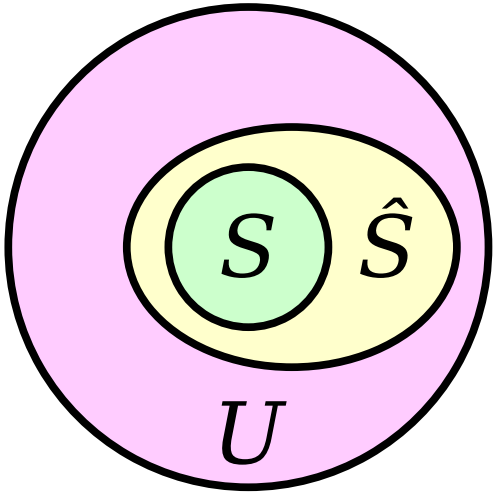
How much memory is needed to solve the approximate membership query problem?

Clever Idea: We can *exactly* describe a set S of size n using an AMQ, plus some extra bits.

First, write down an AMQ for S with error rate ε. Assume this needs b bits.

This AMQ encodes a set $\hat{S}$ of size roughly $\varepsilon|U|$ containing our set S.

To define S, we need to pick n elements from the set $\hat{S}$, which has size $\varepsilon|U|$. This requires $n \lg (\varepsilon|U|)$ bits.



$$b \geq n \left(\lg \left(|U| \,/\, \varepsilon|U|\right)\right)$$

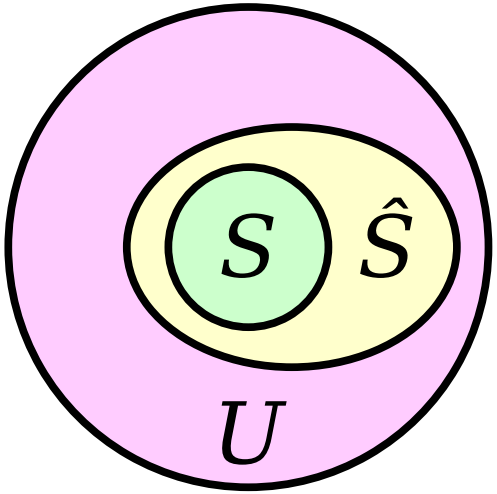
How much memory is needed to solve the approximate membership query problem?

Clever Idea: We can *exactly* describe a set S of size n using an AMQ, plus some extra bits.

First, write down an AMQ for S with error rate ε. Assume this needs b bits.

This AMQ encodes a set $\hat{S}$ of size roughly $\varepsilon|U|$ containing our set S.

To define S, we need to pick n elements from the set $\hat{S}$, which has size $\varepsilon|U|$. This requires $n \lg (\varepsilon|U|)$ bits.



$b \geq n \, (\lg (1 / \varepsilon))$

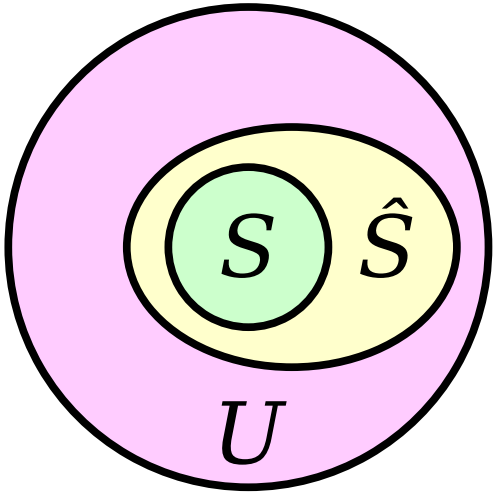
How much memory is needed to solve the approximate membership query problem?

Clever Idea: We can *exactly* describe a set S of size n using an AMQ, plus some extra bits.

First, write down an AMQ for S with error rate ε. Assume this needs b bits.

This AMQ encodes a set $\hat{S}$ of size roughly $\varepsilon|U|$ containing our set S.

To define S, we need to pick n elements from the set $\hat{S}$, which has size $\varepsilon|U|$. This requires $n \lg (\varepsilon|U|)$ bits.



$b \geq n \lg \varepsilon^{-1}$

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

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

Observation: A Bloom filter needs

$1.44\, n \lg \varepsilon^{-1}$,

bits, within a factor of ≈1.44 of optimal.

We can only improve on this by a constant factor.

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

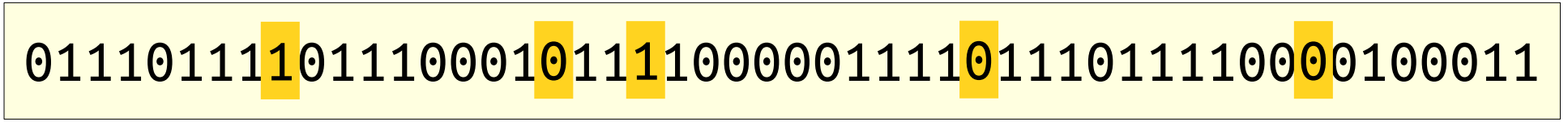
A Historical Note

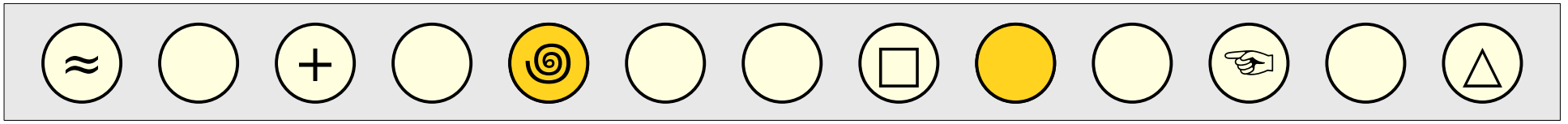
- Bloom filters were invented in 1970.
- For the next ~45 years, there were no practical alternatives that improved on their space usage.
- In the last 10 – 15 years, there's been an explosion of alternatives to Bloom filters that improve on their space usage or otherwise improve their performance.
- ***Bloom filters are essentially obsolete*** at this point. If you're working in an environment where you need one, use an alternative.
- What do those alternatives look like?

Cuckoo Filters

From Bloom to Cuckoo

- A Bloom filter works by
  - hashing an item with some fixed number of hash functions,
  - looking at some positions in a table determined by those hash functions, and
  - using those contents to determine whether the item is in the table.
- If you squint at it the right way, this looks a *lot* like what a cuckoo hash table does.
- ***Question:*** Can we adapt cuckoo hashing to work as an approximate membership query?

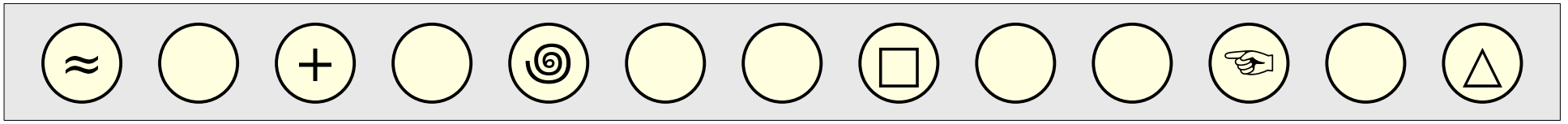




Cuckoo Space Usage

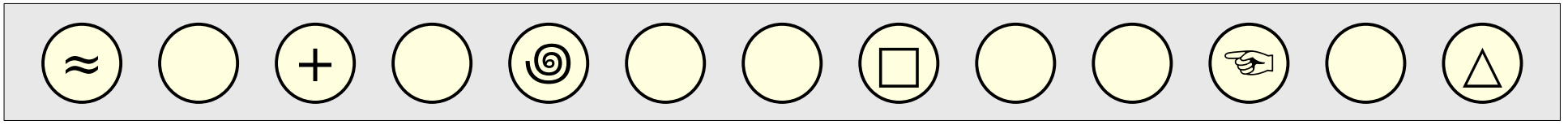
- ***Recall:*** A vanilla cuckoo hash table with two hash functions holding n items uses $\Theta(n)$ slots.
- ***Question:*** Asymptotically, how many *bits* of memory are needed to store a cuckoo hash table holding n distinct elements?

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



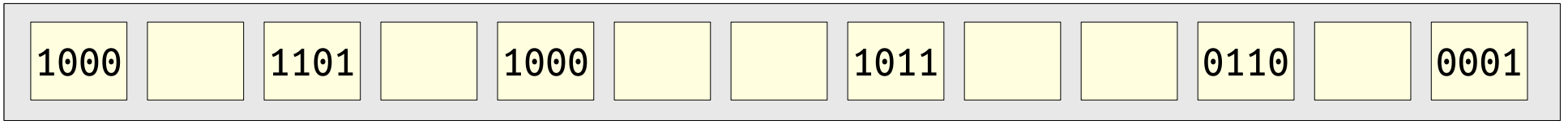
Cuckoo Space Usage

- ***Recall:*** A vanilla cuckoo hash table with two hash functions holding n items uses $\Theta(n)$ slots.
- ***Question:*** Asymptotically, how many *bits* of memory are needed to store a cuckoo hash table holding n distinct elements?
- If we store n distinct elements, then we need $\Omega(\log n)$ bits, on average, for each of those elements.
  - Otherwise, we don't have enough bits to write out n distinct items.
- Total space usage: **$\Omega(n \log n)$** bits.
- ***Goal:*** Reduce this space to $\Theta(n \log \varepsilon^{-1})$ bits, and ideally be as close to the information-theoretic lower bound as possible.



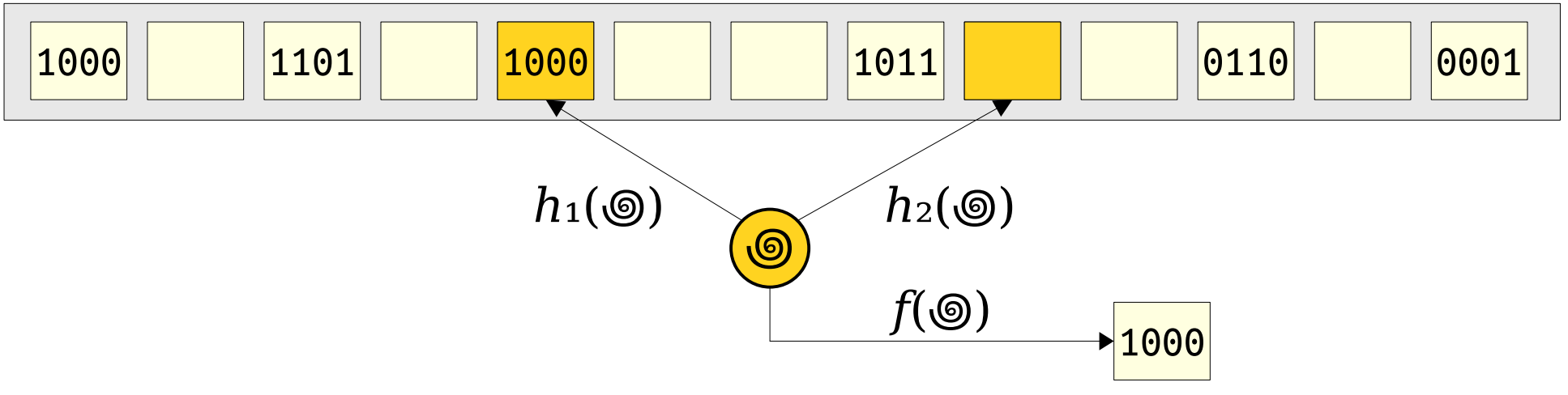
Cuckoo Space Usage

- We need to drop the usage of our cuckoo hash table to $\Theta(n \log \varepsilon^{-1})$.
- ***Intuition:*** That sure looks like what you'd get if you had an array of $\Theta(n)$ slots, each of which was of size $\Theta(\log \varepsilon^{-1})$.
- This isn't enough space to store each of the items in full.
- ***Key Idea:*** Associate each item with a ***fingerprint***, a bit sequence of length L. Then, store fingerprints rather than the items themselves.
  - Ideally, we'll get $L = \Theta(\log \varepsilon^{-1})$.
- We can do this by using a hash function ***f*** that maps from items to L-bit sequences.



Cuckoo Filters

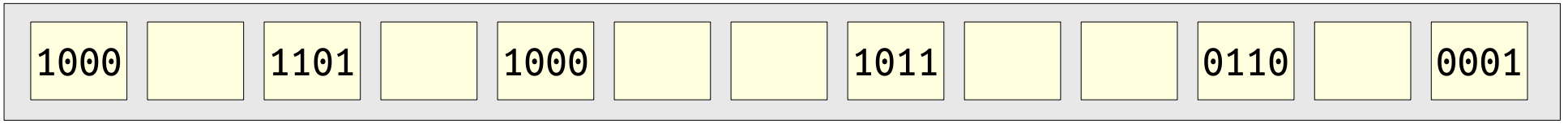
- A ***cuckoo filter*** is a modified cuckoo hash table that stores items' fingerprints, rather than the full items.
- To check whether an item x is present in the cuckoo filter, do the following:
  - Hash the item with two hash functions h_1 and h_2 to get two table indices.
  - Hash the item with f to compute its fingerprint.
  - See whether $f(x)$ is at position $h_1(x)$ or $h_2(x)$ in the table.
- If the item is in the table, this will definitely find it.
- If the item is not in the table, there's a small false positive probability if we coincidentally find a matching fingerprint.



Cuckoo Filters

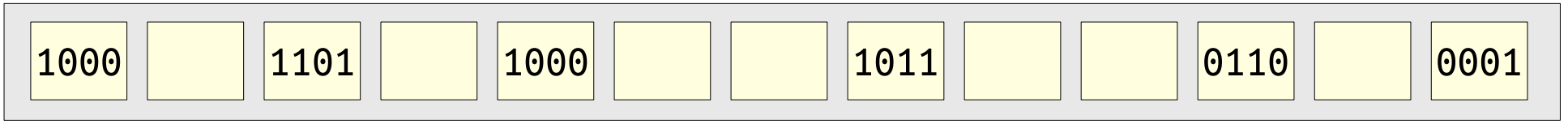
- Our fingerprints have size L. We'd like our false positive rate to be ε. How should we pick L?
- Suppose we query for $x \notin S$. Each lookup tests two positions, each of which has at most a 2^{-L} probability of being $f(x)$.
- We can conservatively bound the false positive rate at 2^{-L+1}.
- Therefore, we should pick

$$\boldsymbol{L = 1 + \lg \varepsilon^{-1}}.$$



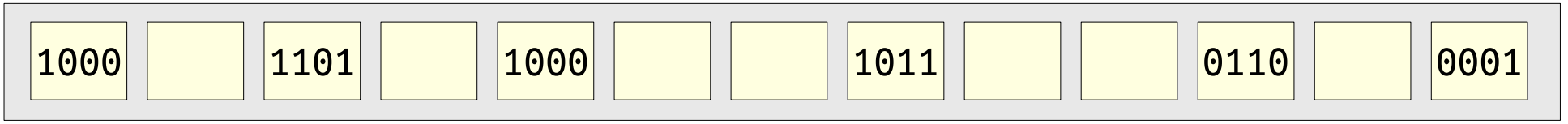
Cuckoo Filters: Two Challenges

- Unfortunately, we're not done just yet. There are two issues we need to address.
  - ***Space utilization:*** Regular cuckoo hash tables max out with a load factor of α = ½.
  - ***Displacement logic:*** There's a subtle issue that pops up when determining how to move items around the table.
- Let's focus on each of these in turn.



Space Utilization

- ***Recall:*** A regular cuckoo hash table (two hash functions, one item per slot) has a maximum load factor of ½.
- If we want to store n elements in a cuckoo filter, we need at least $2n$ slots to keep the load factor below ½.
- Each slot has size $1 + \lg \varepsilon^{-1}$, since it stores an L-bit fingerprint.
- Total bits needed: $\mathbf{2n \lg \varepsilon^{-1} + 2n}$. This is worse than a Bloom filter.
- ***Question:*** Can we do better?

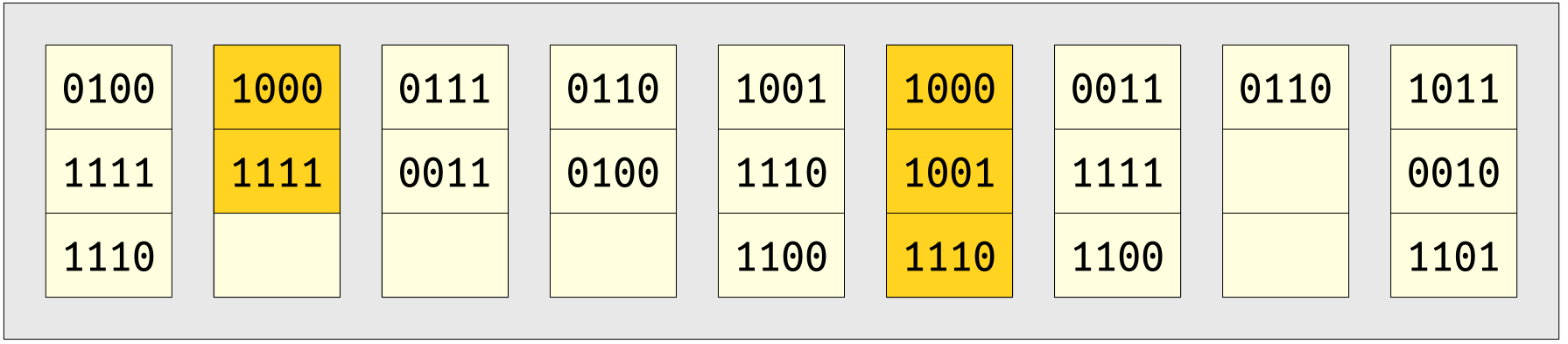


Space Utilization

- In our lecture on cuckoo hashing, we saw two strategies for improving the space utilization of a cuckoo hash table:
  - ***d-Ary Cuckoo Hashing:*** Pick $d \geq 2$ hash functions to determine where to place items.
  - ***Blocked Cuckoo Hashing:*** Each table slot can hold $b \geq 1$ different items.
- In practice, blocked cuckoo hashing is much faster than d-ary cuckoo hashing.
  - We need to evaluate fewer hash functions, and hash functions can be a bottleneck.
  - There's better *locality of reference*, since we only probe two locations.
- ***Idea:*** Improve our space utilization using blocked cuckoo hashing, sticking with two hash functions.

Space Utilization

- Increasing b, the number of items per slot, will
  - increase the load factor of the table, giving better space utilization, but
  - increase the probability of false positives, since each query looks at more fingerprints (each query now sees $2b$ fingerprints).
- ***Question:*** How should we tune the choice of b?



Space Utilization

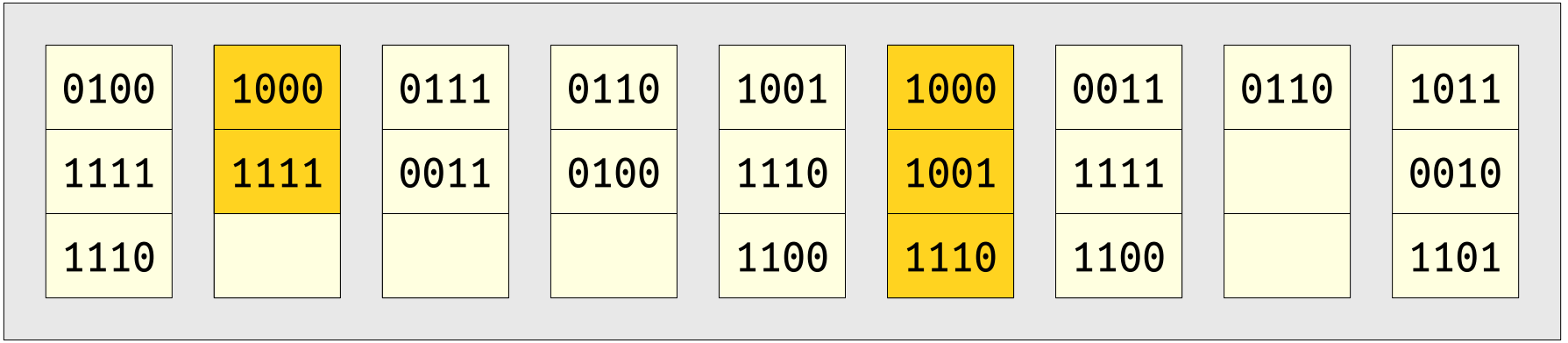
- ***Question:*** With fingerprints of length L and a block size of b, what is the probability of a false positive?
- Suppose we look up a key x that isn't in the set. Each query sees at most $2b$ locations, each of which matches $f(x)$ with probability at most 2^{-L}.
- Via the union bound, the probability of a false positive is at most

$$\varepsilon = 2b \cdot 2^{-L},$$

  which solves to

$$\boldsymbol{L = \lg \varepsilon^{-1} + 1 + \lg b}.$$

- Our fingerprint size has to increase as the size of the slot increases, but not by all that much.



Space Utilization

- Here's the maximum α values we saw for blocked cuckoo hashing.
- If we have $n = \alpha m$ items in our table, then our table size needs to be $m = \alpha^{-1} n$. Each item requires a fingerprint of $(\lg \varepsilon^{-1} + 1 + \lg b)$ bits.
- Here's what that looks like for different values of b:
  - $b = 1$: about $2.00 \lg \varepsilon^{-1} + 2.00$ bits per element.
  - $b = 2$: about $1.12 \lg \varepsilon^{-1} + 2.24$ bits per element.
  - $b = 4$: about $1.02 \lg \varepsilon^{-1} + 3.06$ bits per element.
  - $b = 8$: about $1.01 \lg \varepsilon^{-1} + 4.04$ bits per element.
  - $b = 16$: about $1.00 \lg \varepsilon^{-1} + 5.03$ bits per element.
- For common values of ε, this is minimized when $b = 4$ or $b = 8$.

	$b = 1$	$b = 2$	$b = 4$	$b = 8$
Max α	**0.5**	**0.897**	**0.980**	**0.997**
α^{-1}	**2**	**1.12**	**1.02**	**1.01**

Space Utilization

- Using $b = 4$, we get that the space usage for our cuckoo filter is

  $$\mathbf{1.02 \lg \varepsilon^{-1} + 3.06}$$

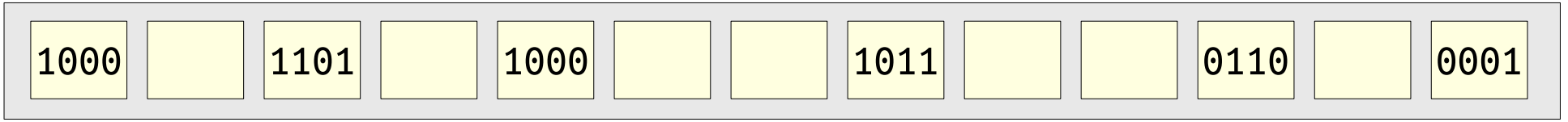
  bits per element.
- For small ε, this is strictly better than a Bloom filter.

	$b = 1$	$b = 2$	$b = 4$	$b = 8$
Max α	**0.5**	**0.897**	**0.980**	**0.997**
α^{-1}	**2**	**1.12**	**1.02**	**1.01**

The Catch: Displacements

Supporting Displacements

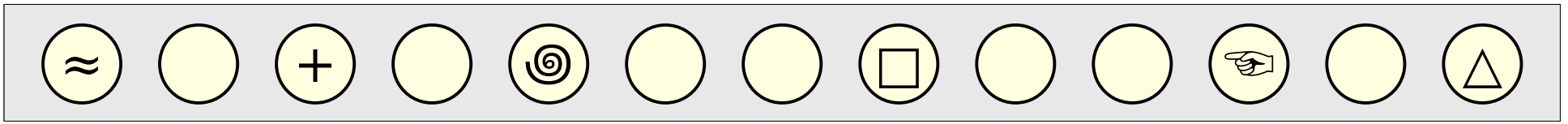
- In a normal cuckoo hash table, insertions can displace items already in the table.
- Specifically, we may need to move an item x from position $h_1(x)$ to $h_2(x)$, or vice-versa.
- ***Problem:*** Our table stores the *fingerprint* of x, rather than x itself. If we need to displace $f(x)$, how do we determine which slot to move it to if we don't know x?



Supporting Displacements

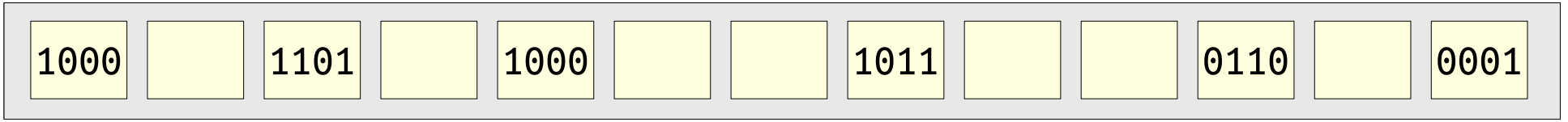
- Let's begin with a cute way to choose hash functions for a standard cuckoo hash table.
  - Pick $h_1 : U \to \{0, 1, 2, \ldots, m - 1\}$, where U is the universe of possible keys, assuming the table has m slots.
  - Pick $h_\Delta : U \to \{1, 2, 3, \ldots, m - 1\}$ as an offset hash function that moves us relative to h_1.
  - Define $h_2(x) = h_1(x) \oplus h_\Delta(x)$.
- To displace an item x at position i:
  - Compute $h_\Delta(x)$ from the value of x in the table.
  - Move the item to index $i \oplus h_\Delta(x)$.

Why does this work?
Answer at
https://cs166.stanford.edu/pollev



Supporting Displacements

- Let's begin with a cute way to choose hash functions for a standard cuckoo hash table.
  - Pick $h_1 : U \to \{0, 1, 2, \ldots, m - 1\}$, where U is the universe of possible keys, assuming the table has m slots.
  - Pick $h_\Delta : U \to \{1, 2, 3, \ldots, m - 1\}$ as an offset hash function that moves us relative to h_1.
  - Define $h_2(x) = h_1(x) \oplus h_\Delta(x)$.
- To displace an item x at position i:
  - Compute $h_\Delta(x)$ from the value of x in the table.
  - Move the item to index $i \oplus h_\Delta(x)$.
- ***Problem:*** This doesn't work with fingerprints.



Supporting Displacements

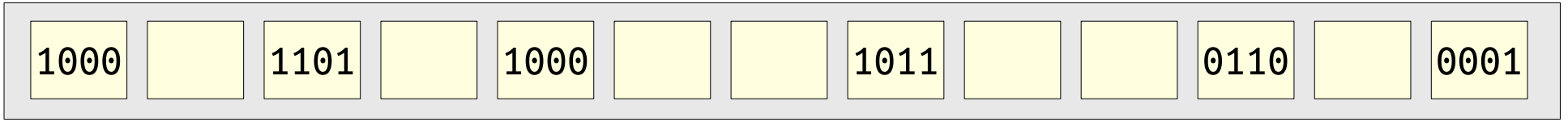
- Let's begin with a cute way to choose hash functions for a standard cuckoo hash table.
  - Pick $h_1 : U \to \{0, 1, 2, ..., m$ keys, assuming the table h
  - Pick $h_\Delta : U \to \{1, 2, 3, ..., m$ us relative to h_1.
  - Define $h_2(x) = h_1(x) \oplus h_\Delta(x$

We have $f(x)$, rather than x, stored in the table.

We can't evaluate $h_\Delta(x)$.

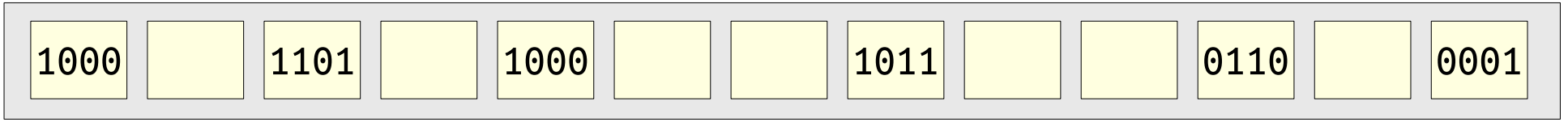
However, we *can* evaluate $h_\Delta(f(x))$.

- To displace an item x at position i:
  - Compute $h_\Delta(x)$ from the value of x in the table.
  - Move the item to index $i \oplus h_\Delta(x)$.
- ***Problem:*** This doesn't work with fingerprints.



Supporting Displacements

- Here's how we can adapt this to work for cuckoo filters.
  - Pick $h_1 : U \to \{0, 1, 2, \ldots, m - 1\}$, where U is the universe of possible keys, assuming the table has m slots.
  - Pick $h_\Delta : F \to \{1, 2, 3, \ldots, m - 1\}$, where F is the set of possible fingerprints.
  - Define $h_2(x) = h_1(x) \oplus h_\Delta(f(x))$.
- To displace a fingerprint $f(x)$ at position i:
  - Compute $h_\Delta(f(x))$ from the value of $f(x)$ in the table.
  - Move the item to index $i \oplus h_\Delta(f(x))$.
- This will always move the fingerprint from $h_1(x)$ to $h_2(x)$ or vice-versa, and doesn't require knowing x.



Supporting Displacements

- Look closely at the definition of $h_2(x)$:

$$h_2(x) = h_1(x) \oplus h_\Delta(f(x)).$$

- Do you notice anything potentially problematic about this?

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

Supporting Displacements

- Look closely at the definition of $h_2(x)$:

$$h_2(x) = h_1(x) \oplus h_\Delta(f(x)).$$

- Do you notice anything potentially problematic about this?
- Take an extreme case: suppose our fingerprints are just a single bit. What can you say about $h_2(x)$?

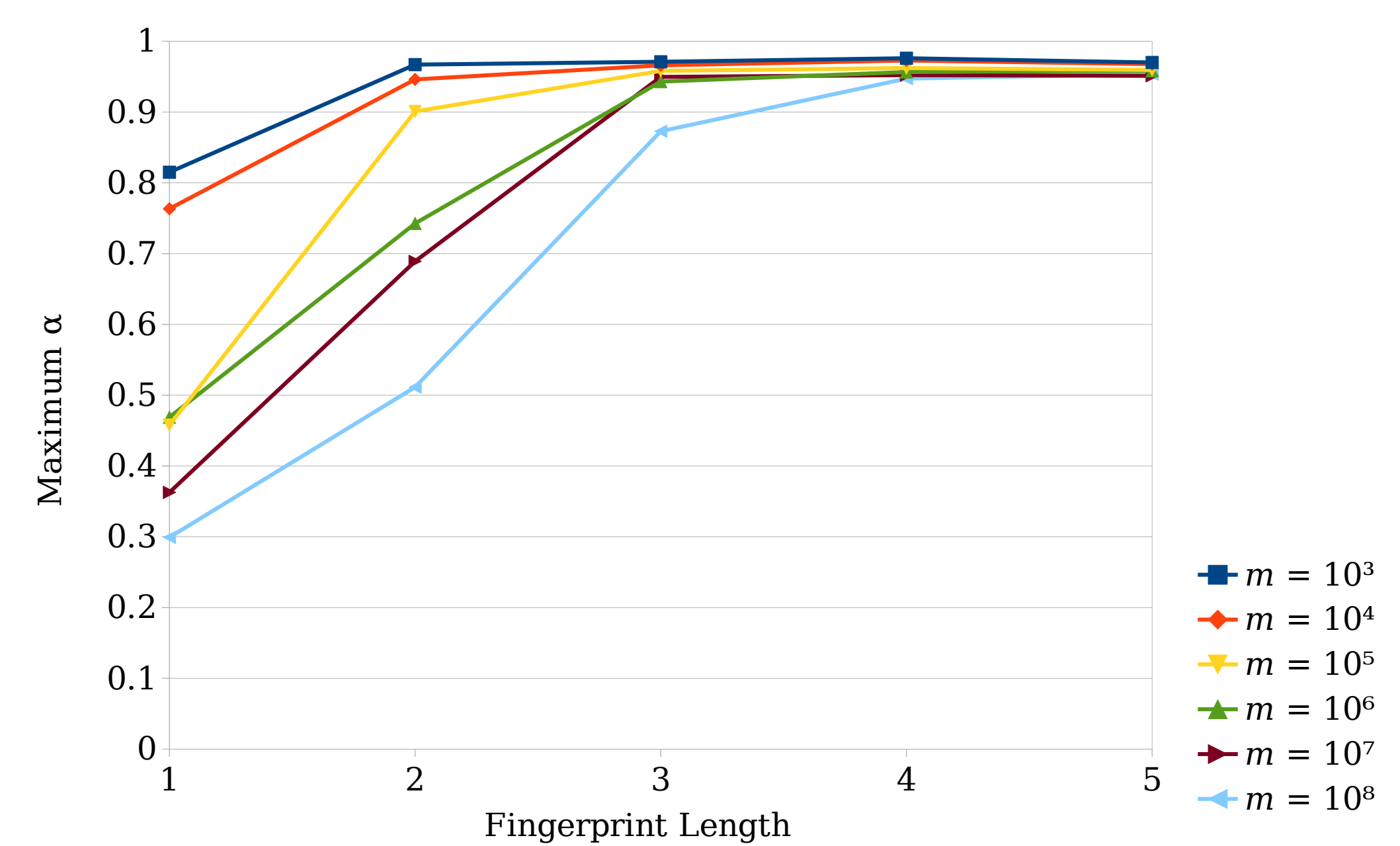
Supporting Displacements

- Look closely at the definition of $h_2(x)$:

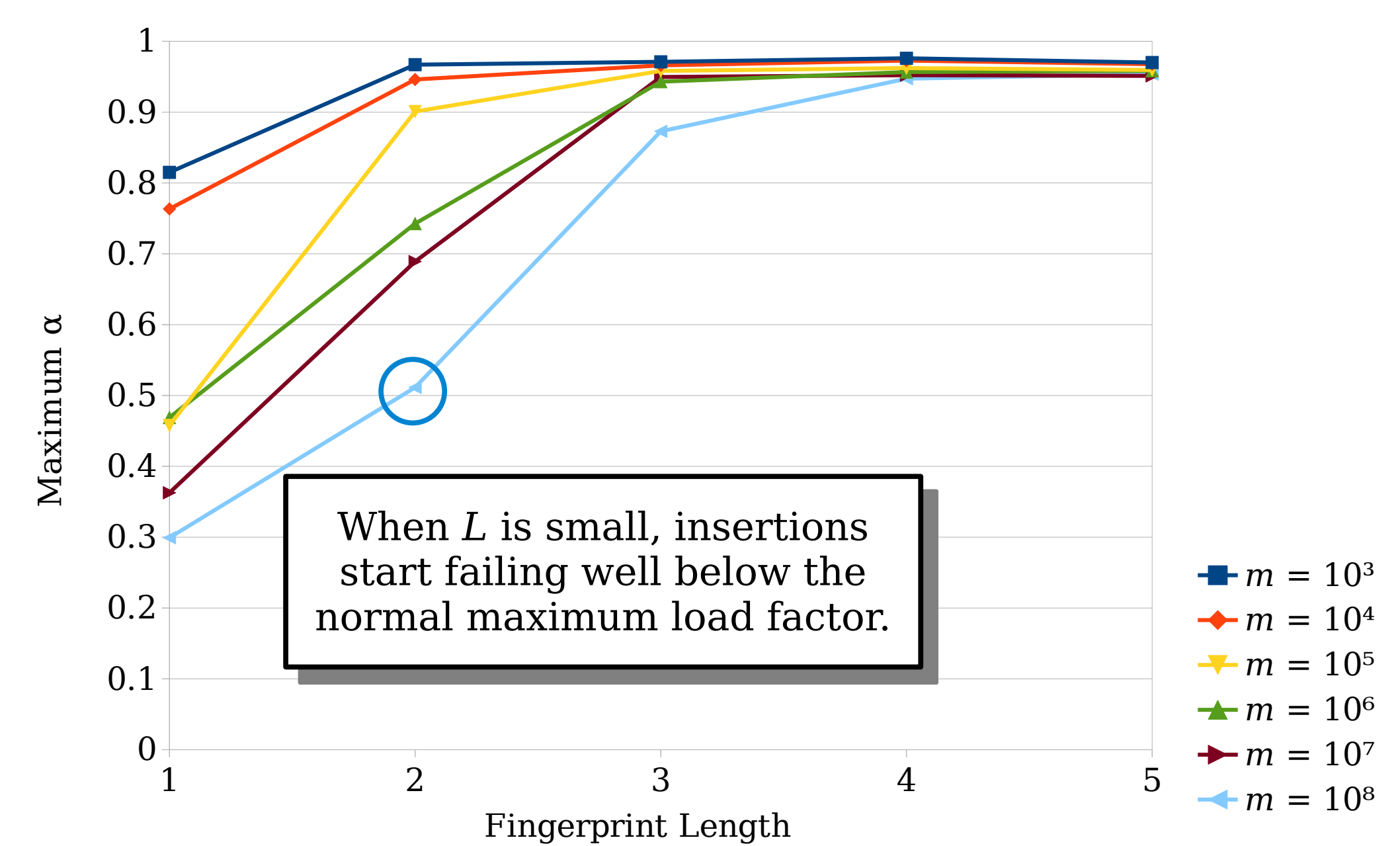
$$h_2(x) = h_1(x) \oplus h_\Delta(f(x)).$$

- ***Problem:*** While $h_1(x)$ is uniform over the space of the table, $h_2(x)$ can only take on at most 2^L different values, where 2^L is (probably) much smaller than m, the number of table slots.
- This means that the distribution of possible pairs $(h_1(x), h_2(x))$ is not uniform over the table, invalidating our initial analysis of cuckoo hashing.
- This is not just a theoretical issue.

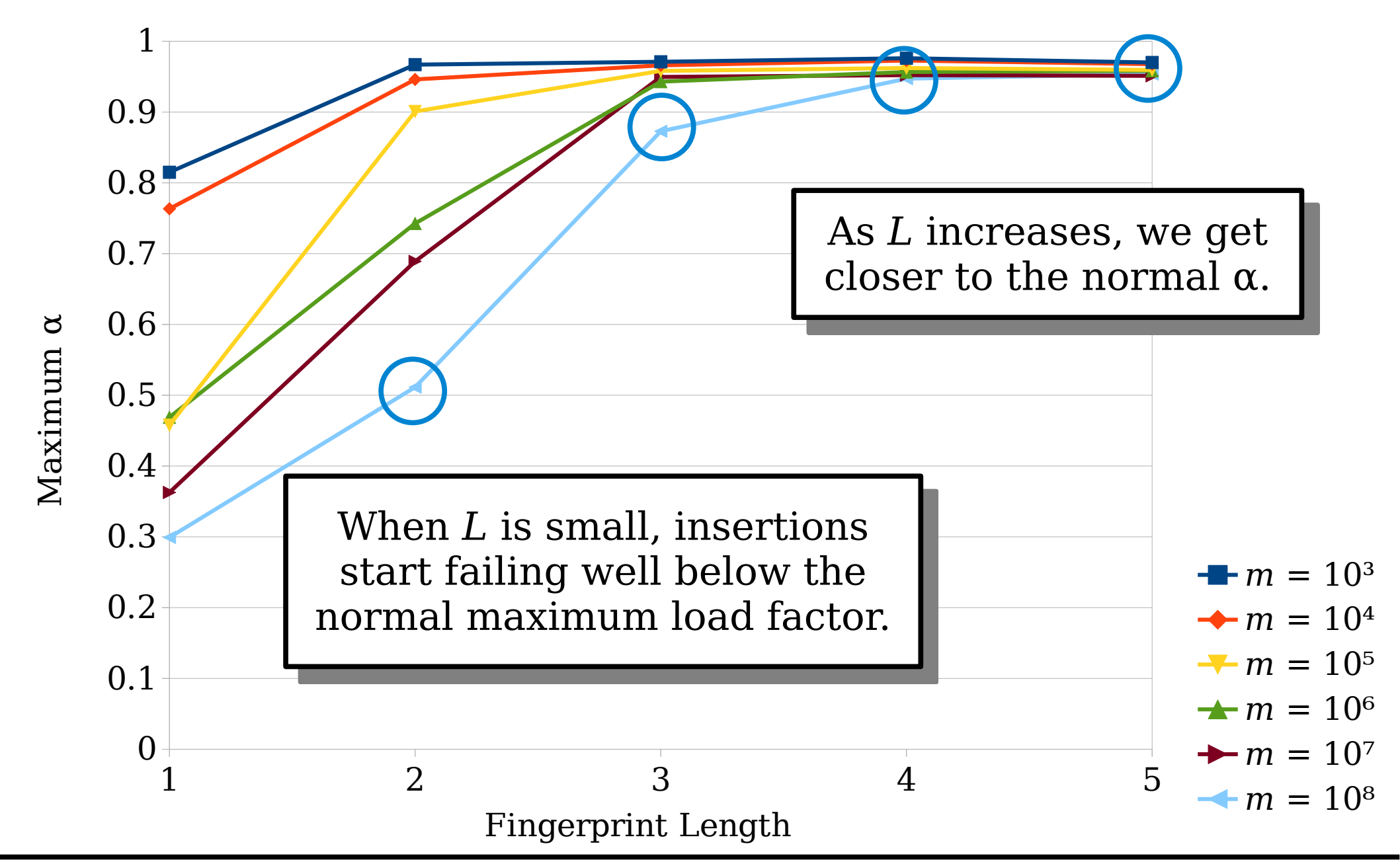
Build a cuckoo filter ($b = 4$) with $m/4$ slots and fingerprints of length L. What is the maximum load factor α where the success probability of inserting $n = \alpha m$ items is at least 99%?



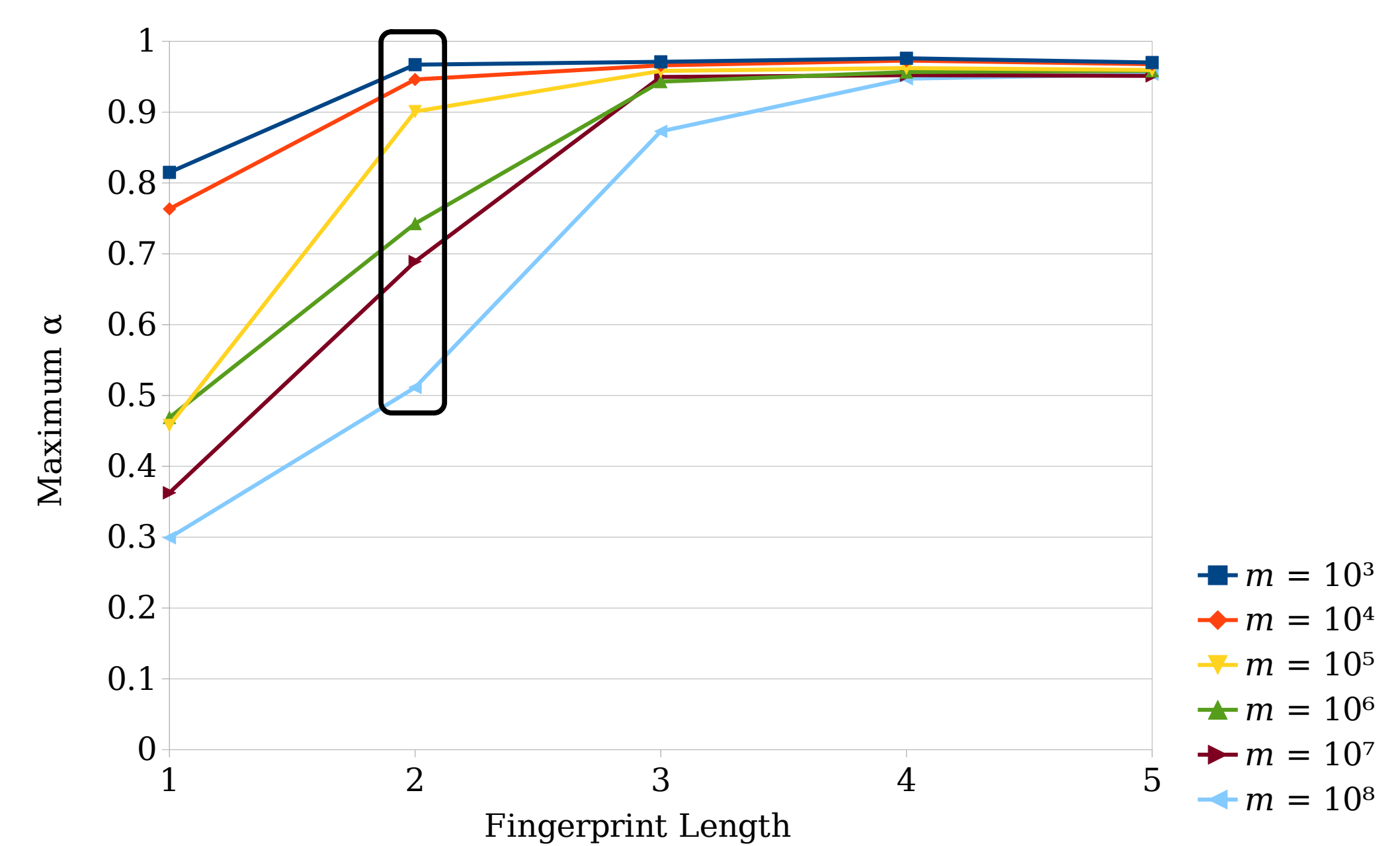
Build a cuckoo filter ($b = 4$) with $m/4$ slots and fingerprints of length L. What is the maximum load factor α where the success probability of inserting $n = \alpha m$ items is at least 99%?



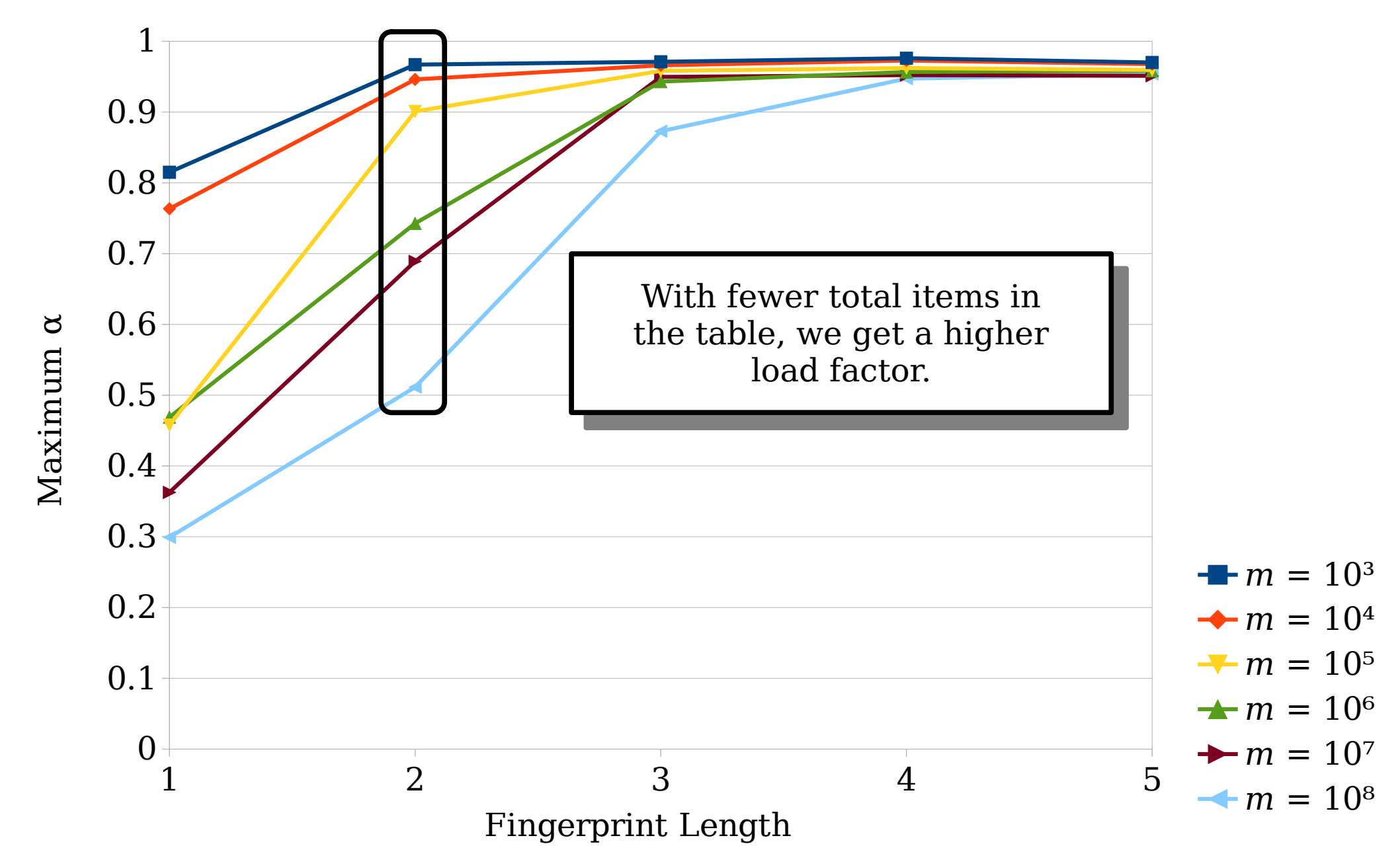
Build a cuckoo filter ($b = 4$) with $m/4$ slots and fingerprints of length L. What is the maximum load factor α where the success probability of inserting $n = \alpha m$ items is at least 99%?



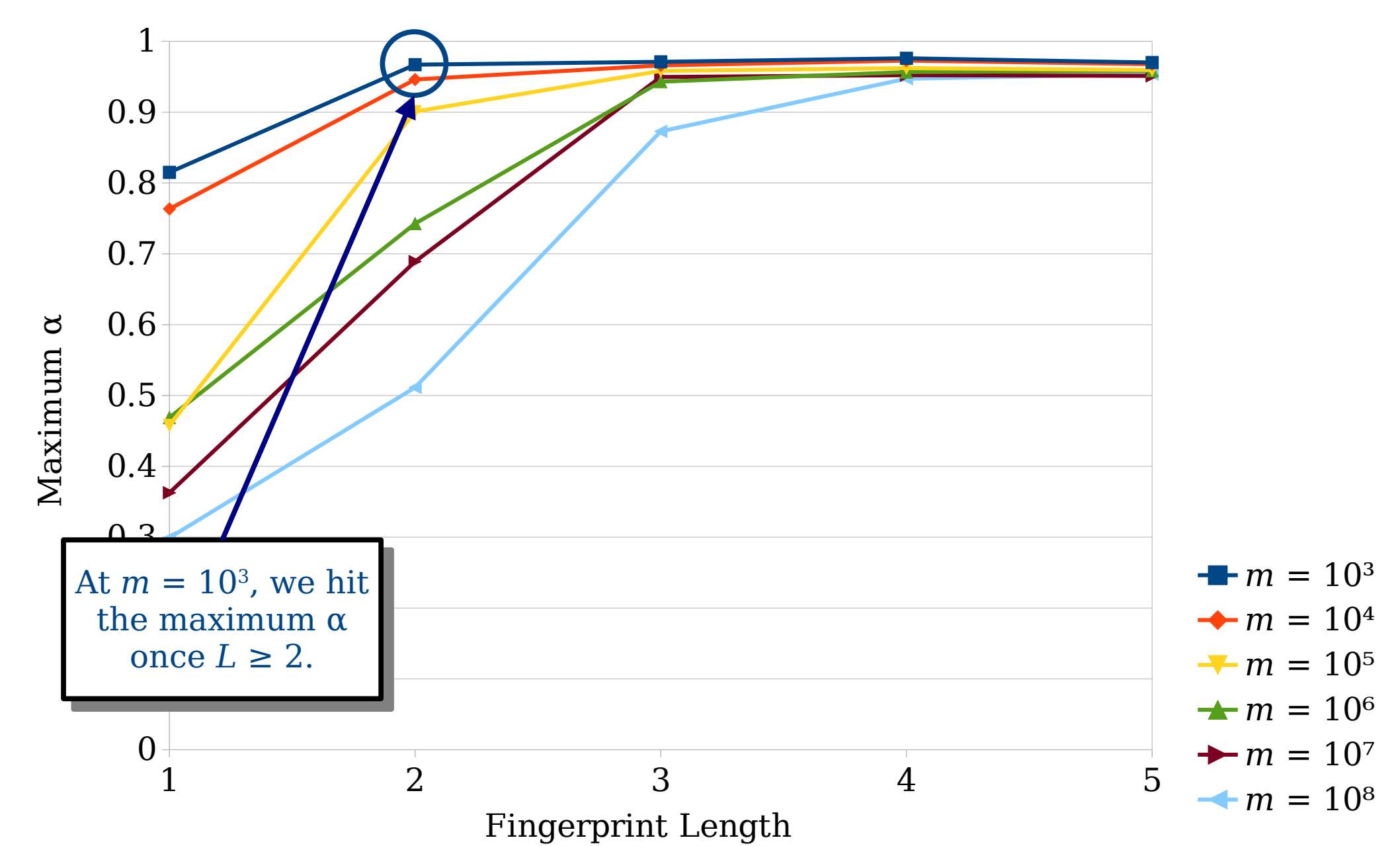
Build a cuckoo filter ($b = 4$) with $m/4$ slots and fingerprints of length L. What is the maximum load factor α where the success probability of inserting $n = \alpha m$ items is at least 99%?



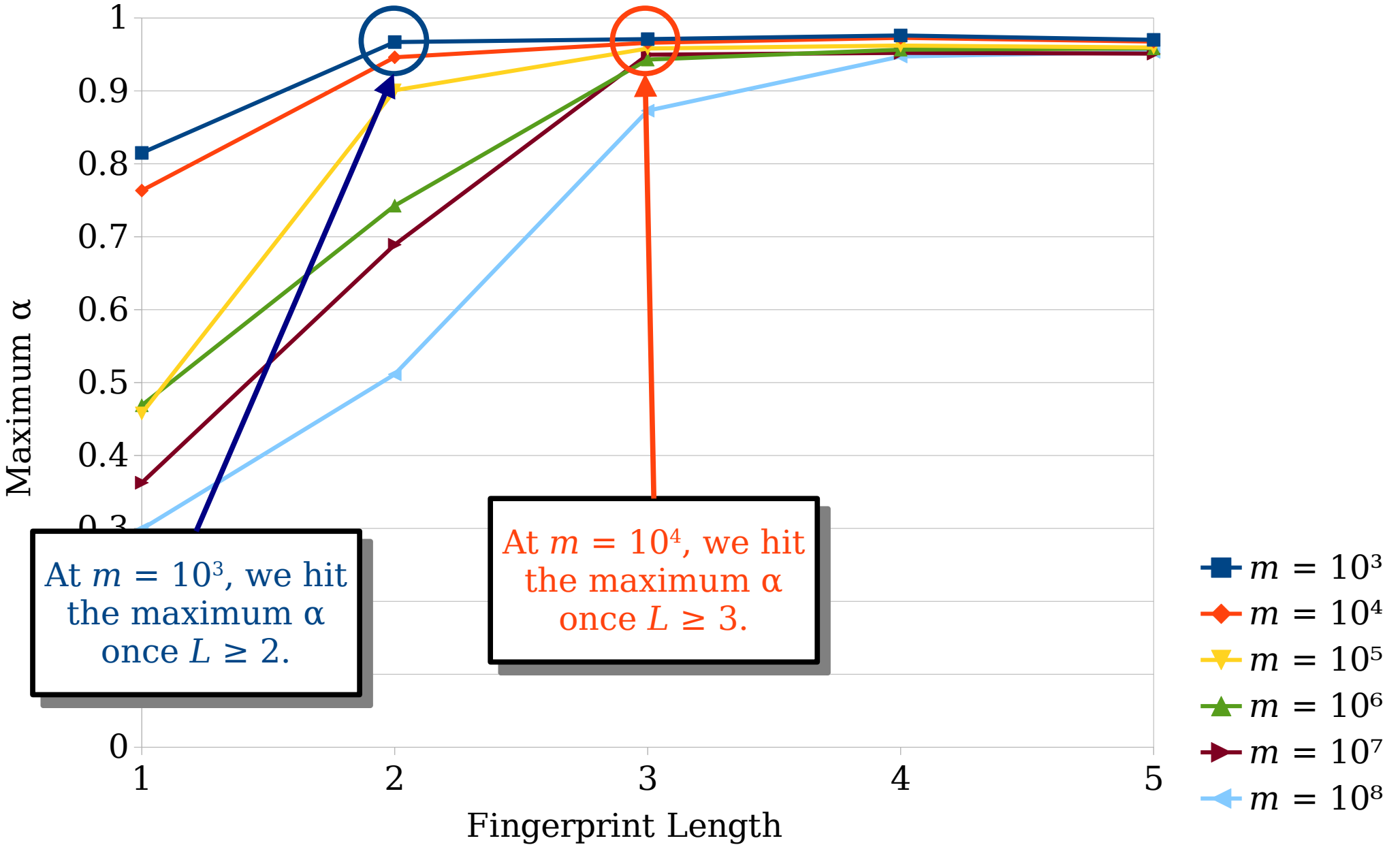
Build a cuckoo filter ($b = 4$) with $m/4$ slots and fingerprints of length L. What is the maximum load factor α where the success probability of inserting $n = \alpha m$ items is at least 99%?



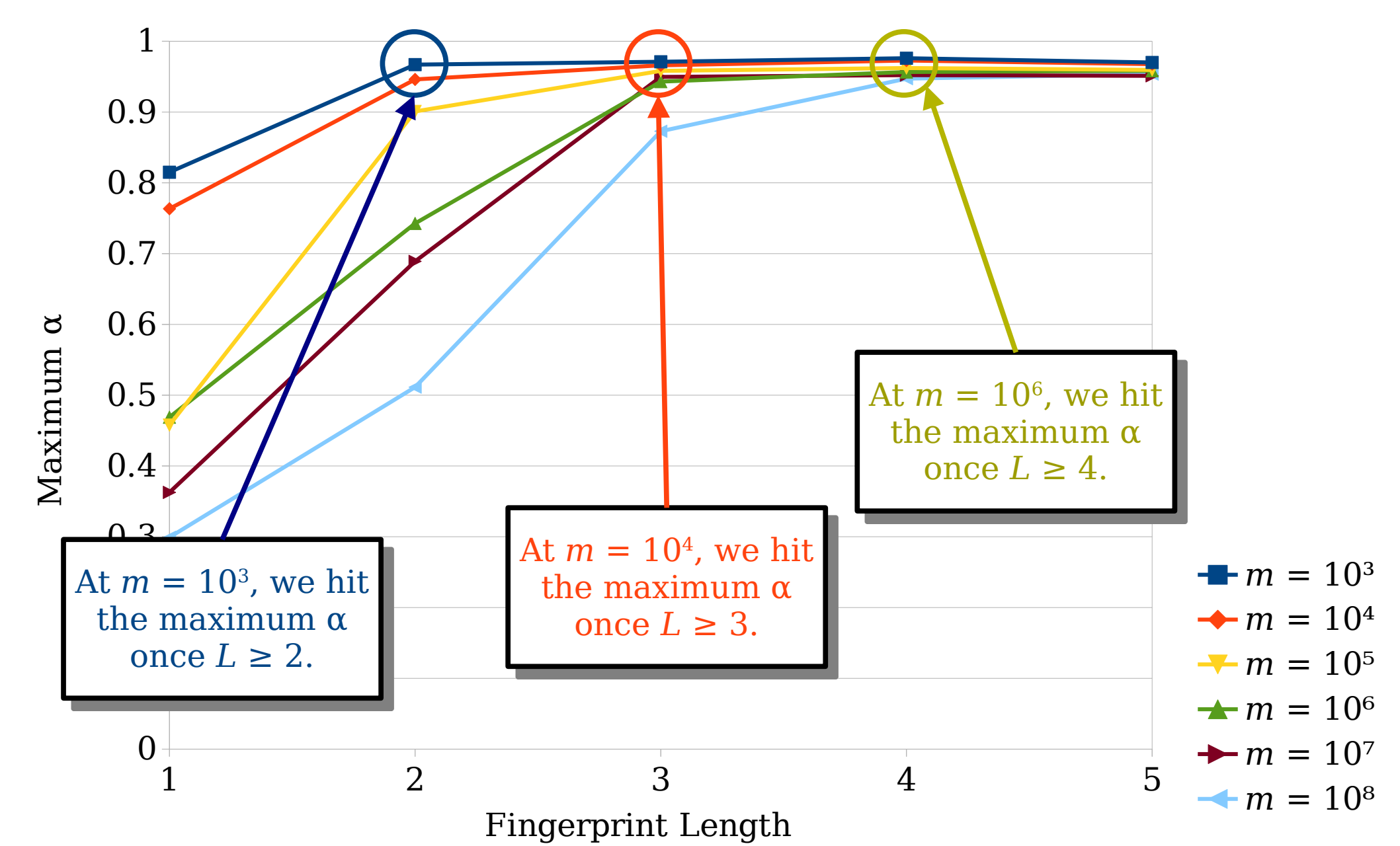
Build a cuckoo filter ($b = 4$) with $m/4$ slots and fingerprints of length L. What is the maximum load factor α where the success probability of inserting $n = \alpha m$ items is at least 99%?



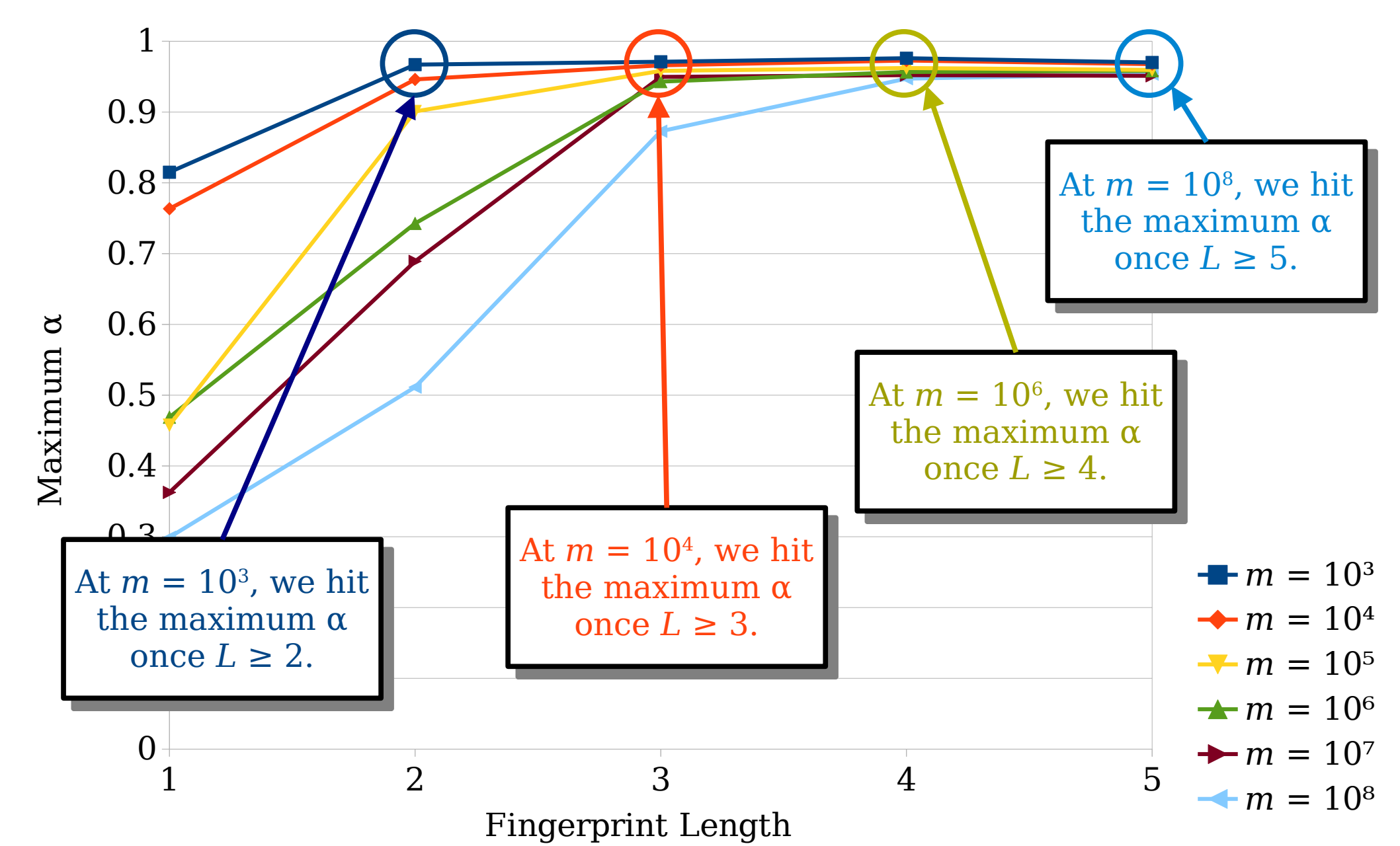
Build a cuckoo filter ($b = 4$) with $m/4$ slots and fingerprints of length L. What is the maximum load factor α where the success probability of inserting $n = \alpha m$ items is at least 99%?



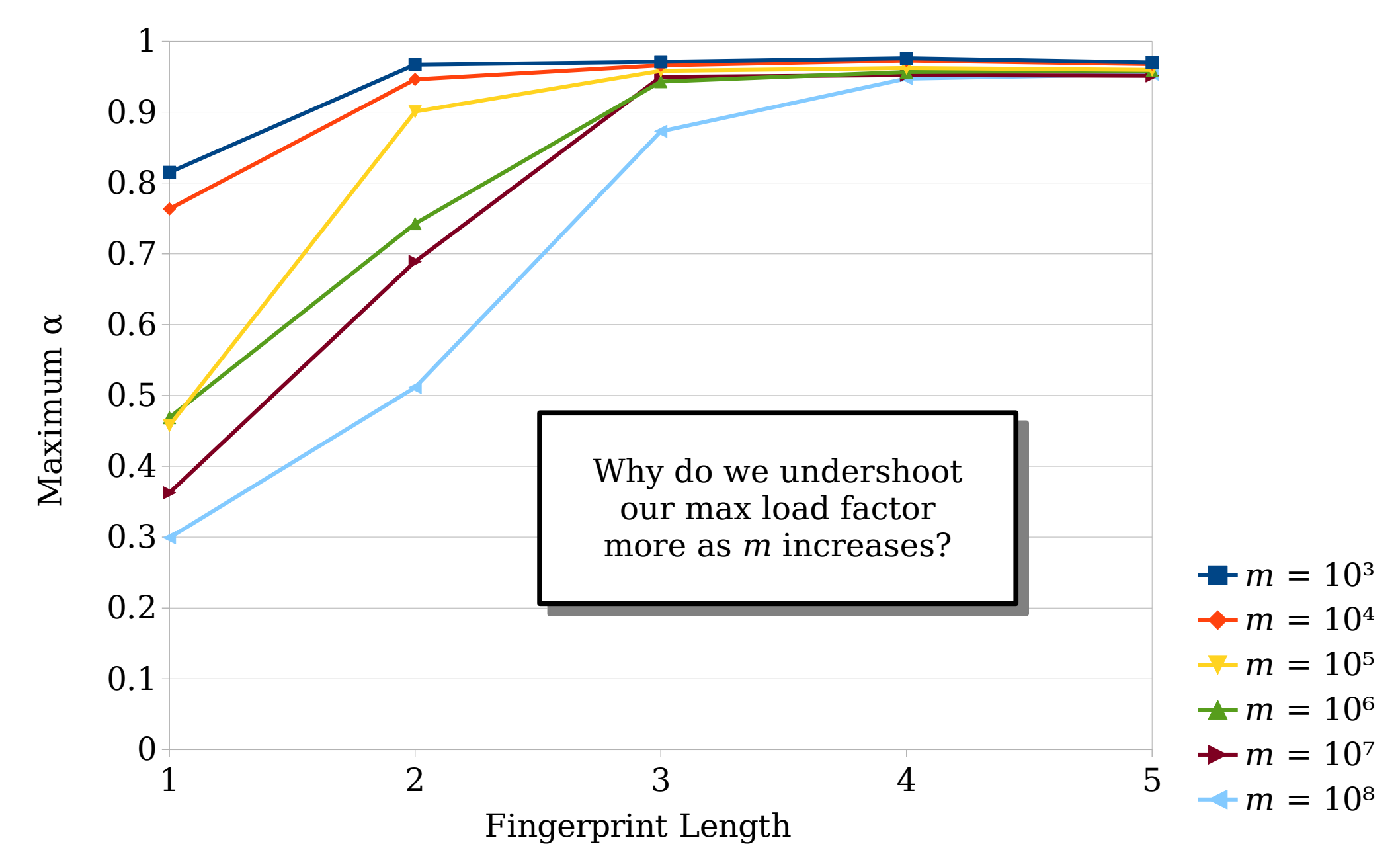
Build a cuckoo filter ($b = 4$) with $m/4$ slots and fingerprints of length L. What is the maximum load factor α where the success probability of inserting $n = \alpha m$ items is at least 99%?



Build a cuckoo filter ($b = 4$) with $m/4$ slots and fingerprints of length L. What is the maximum load factor α where the success probability of inserting $n = \alpha m$ items is at least 99%?



Build a cuckoo filter ($b = 4$) with $m/4$ slots and fingerprints of length L. What is the maximum load factor α where the success probability of inserting $n = \alpha m$ items is at least 99%?



Build a cuckoo filter ($b = 4$) with $m/4$ slots and fingerprints of length L. What is the maximum load factor α where the success probability of inserting $n = \alpha m$ items is at least 99%?

A Lower Bound

- ***Theorem:*** For cuckoo filters to behave "close enough" to regular cuckoo hashing, and therefore reach the theoretical maximum load factor, we must have

$$L = \Omega((\log n) / b).$$

- ***Recall:*** To achieve false positive rate ε, we need to have

$$L = \lg \varepsilon^{-1} + 1 + \lg b.$$

- In practice, picking a modest choice of ε (say, $\varepsilon = {}^{1}/_{32}$) is sufficient to satisfy the lower bound with $b = 4$ and $n \leq 2^{32}$.
- In theory, cuckoo filters only use $\Theta(\lg \varepsilon^{-1})$ bits per element if either ε drops as n increases, or we increase b and our lookup times are allowed to increase.
- This means that cuckoo filters are "practically" better than a Bloom filter for most choices of ε, but only *theoretically* better than a Bloom filter if $\varepsilon = o(1)$ as a function of n.

To Summarize

The Cuckoo Filter ($b = 4$)

- Maintain an array whose size is a power of two and that has at least $0.27n$ slots, each of which can hold four fingerprints of size $\lg \varepsilon^{-1} + 3$, all initially empty.
- Pick a fingerprinting function f from items to fingerprints.
- Pick a hash function h_1 from items to slots, and a hash function h_Δ from fingerprints to slots (excluding zero). Define $h_2(x) = h_1(x) \oplus h_\Delta(f(x))$.
- For each item x to store, compute its fingerprint $f(x)$ and insert $f(x)$ into the table using the normal cuckoo hashing insertion algorithm.
- To see if x is in the table, compute $f(x)$ and see if it's in the table in either slot $h_1(x)$ or $h_2(x)$.

The Story So Far

- Cuckoo filters require at most three hashes per query, compared with $\lg \varepsilon^{-1}$ for the Bloom filter. They also have markedly better cache locality.
- For $\varepsilon \leq 2^{-7}$, in practice, cuckoo filters use less space than Bloom filters.
- This is a practical, drop-in replacement for a Bloom filter for small choices of ε.

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

The Story So Far

- In Theoryland, cuckoo filters require us to decrease ε or increase b as n increases.
- ***Question:*** Is there a way to improve on Bloom filters, both practically and in Theoryland?

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

XOR Filters

Bloom and Cuckoo Revisited

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

Store a large array of tiny items.

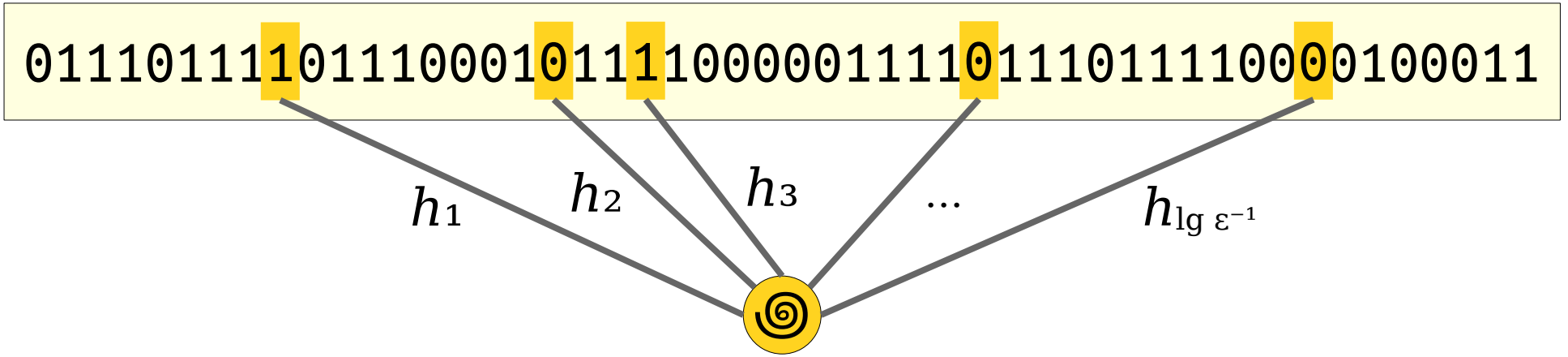
Hash an item to multiple positions in an array, aggregate those array positions together, and see whether you get what you want.

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

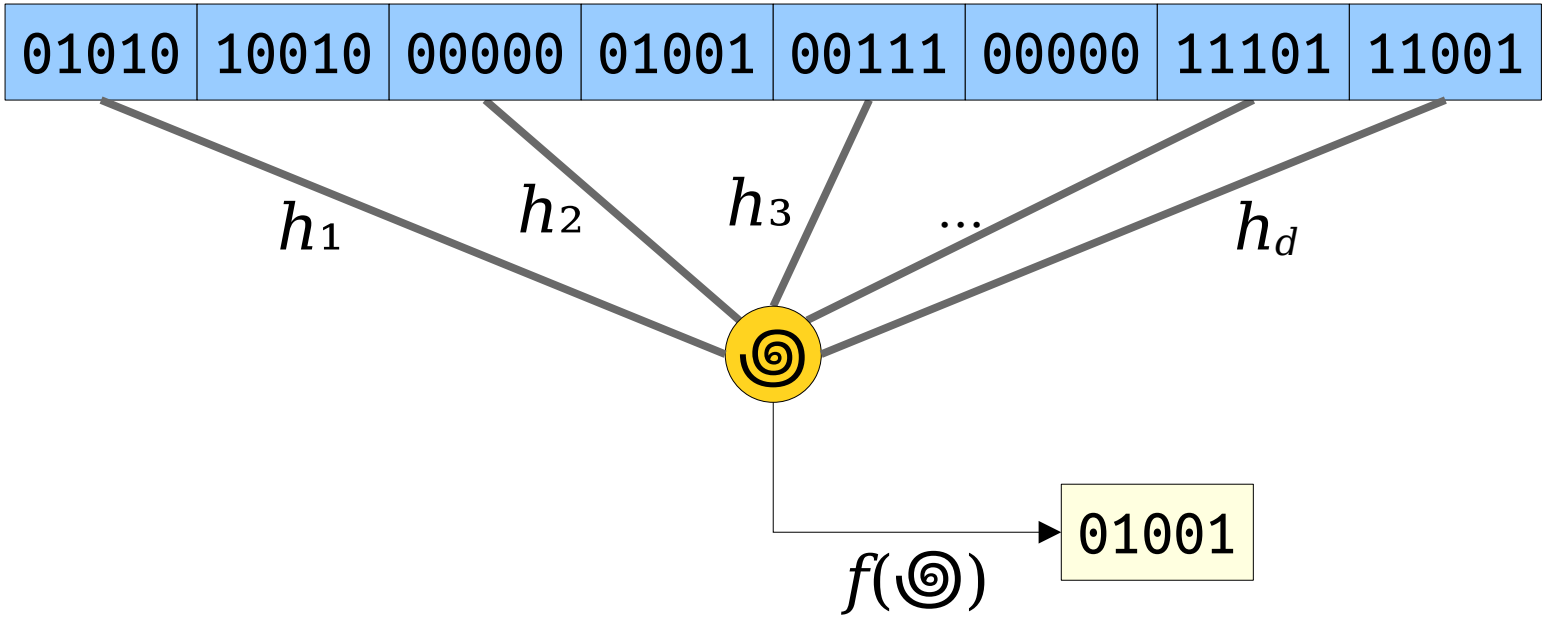
Store a medium-sized array of medium-sized items.

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

- What happens if we combine these ideas together?



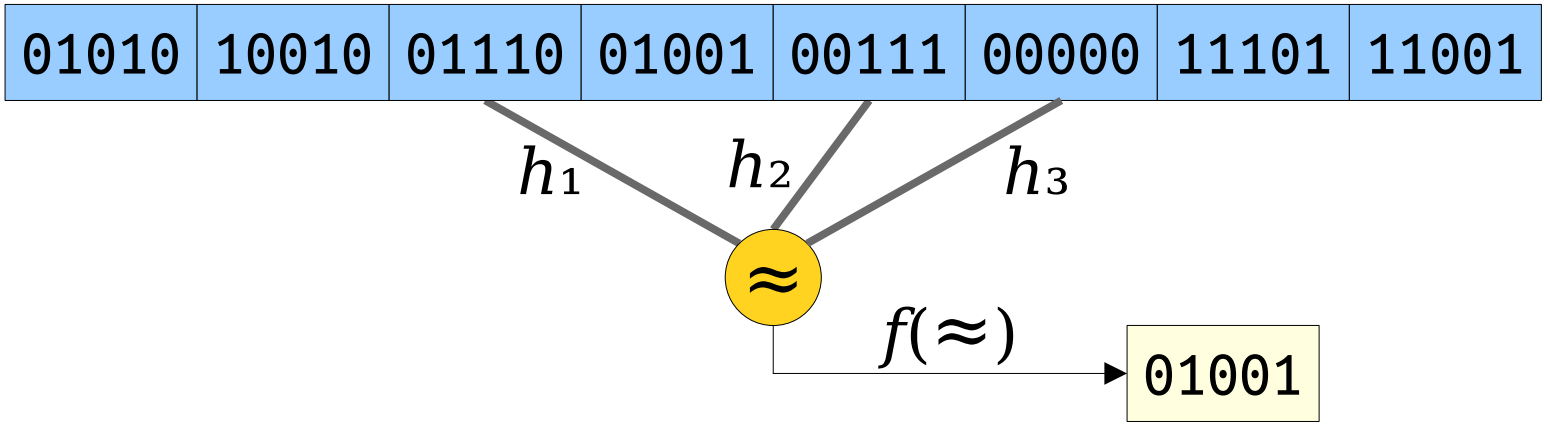
	Bloom Filters	
Store an array of items	Array of bits	
Pick some array slots	Hash item with $\lg \varepsilon^{-1}$ hash functions	
Derive single value from slots	AND	
Compare value against expected	Should be 1	



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

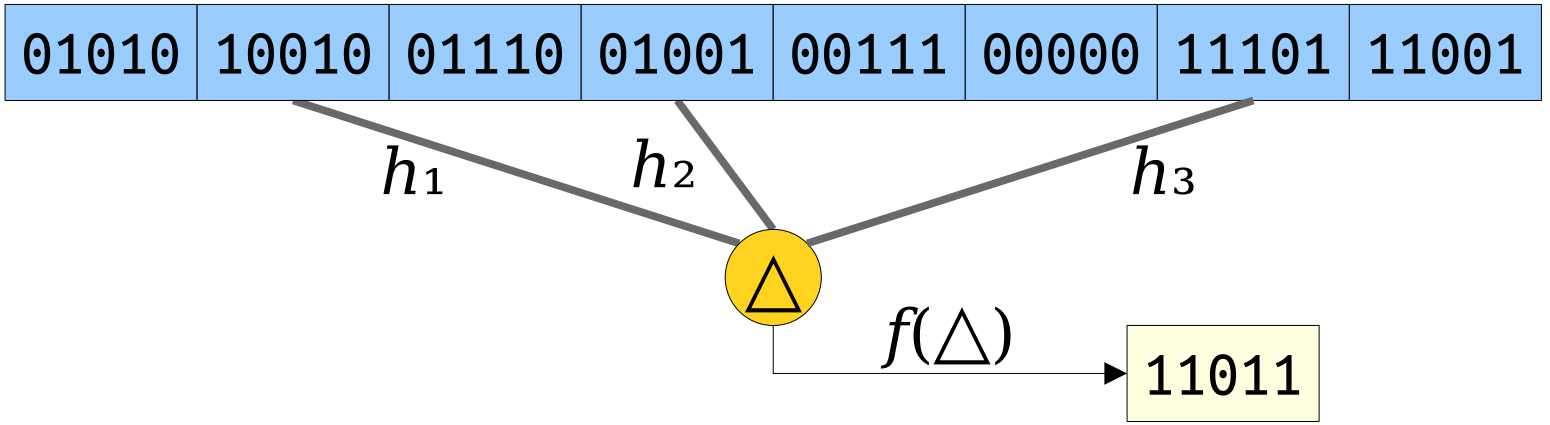
The XOR Filter

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



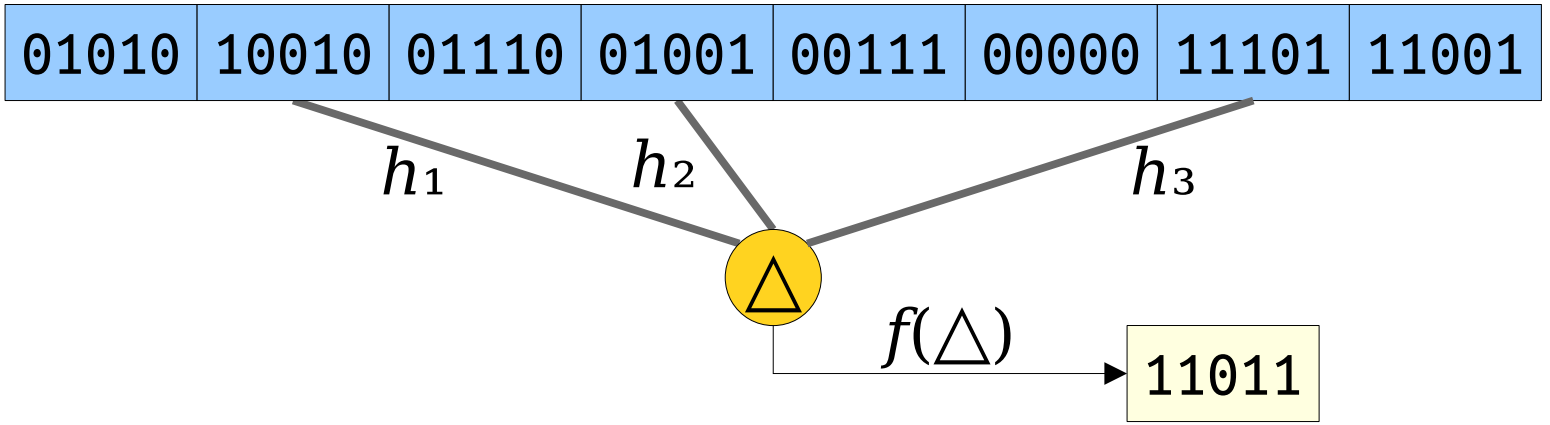
The XOR Filter

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



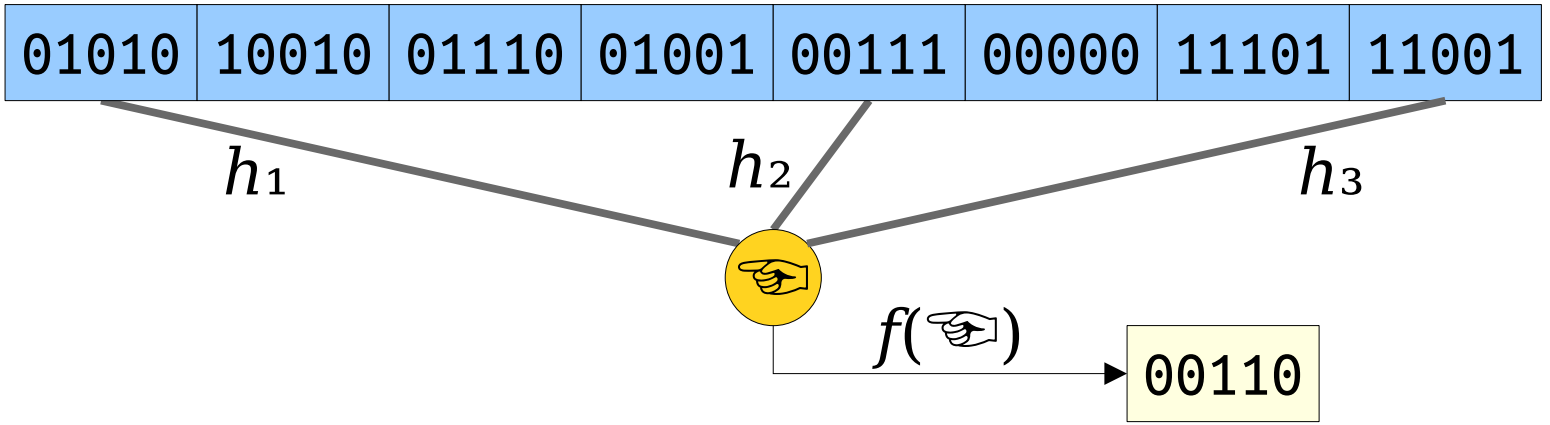
The XOR Filter

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



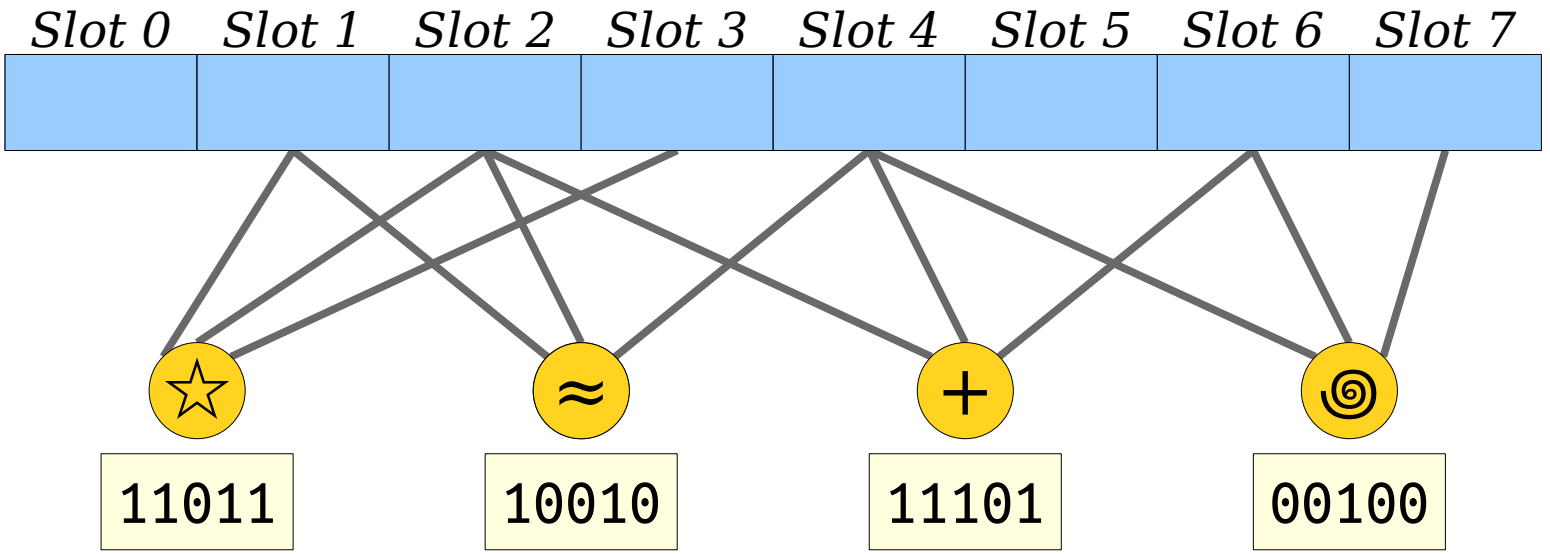
The Fingerprint Size

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



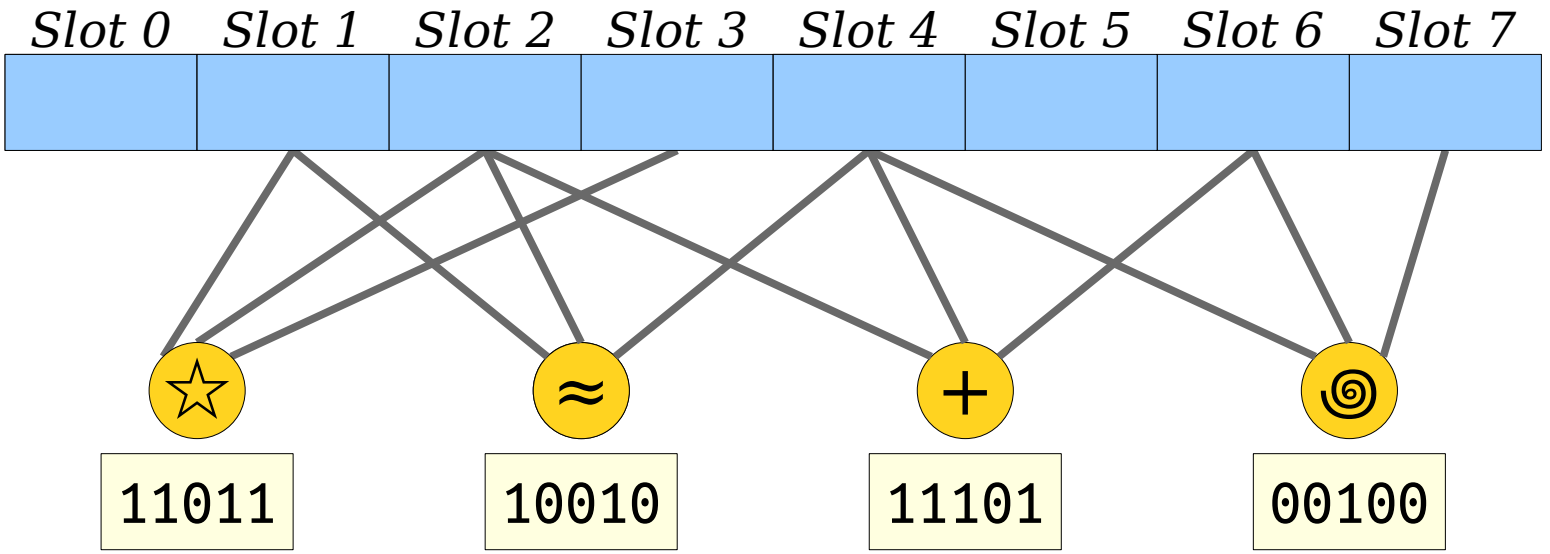
Filling Our Table

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



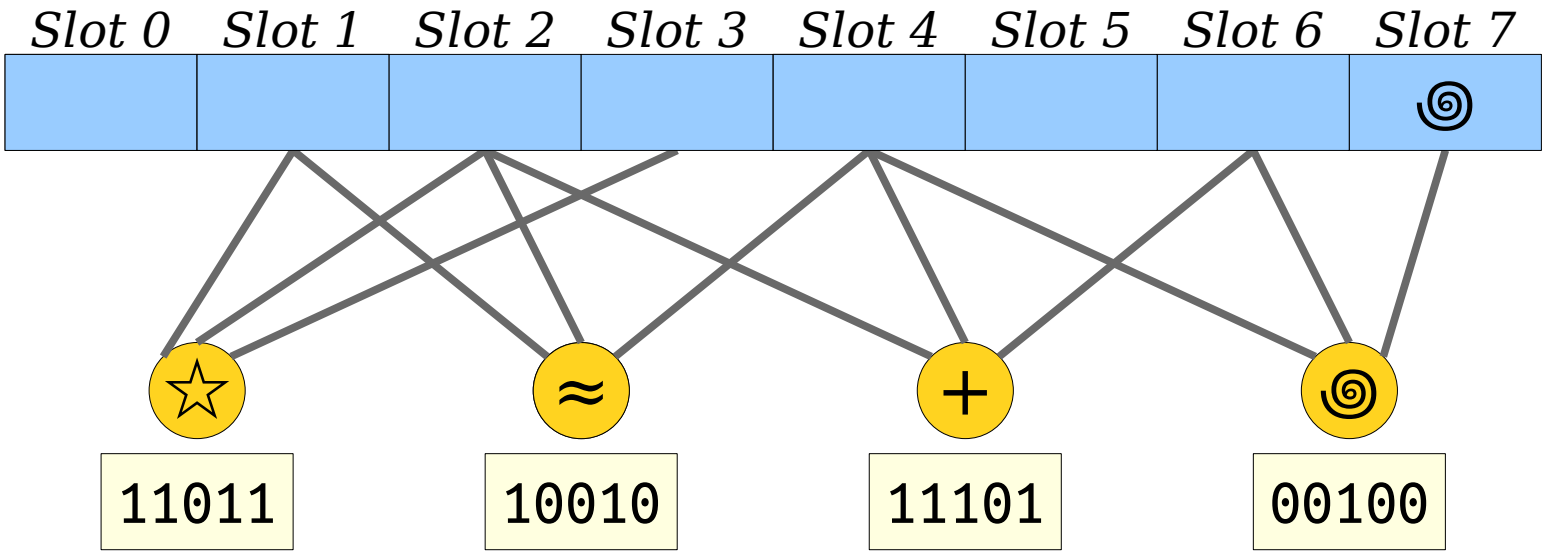
Filling Our Table

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



Filling Our Table

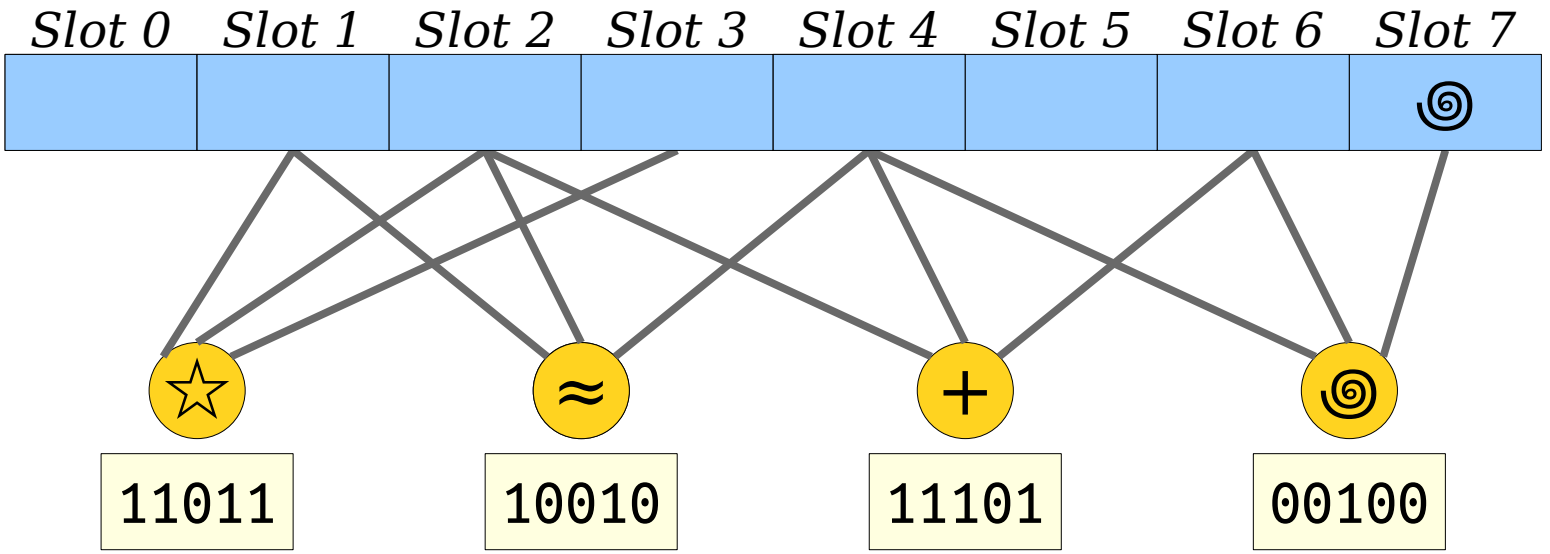
- Notice that slot 7 has only one item (namely, ◎) hashing to it.
- This means that tuning slot 7 can only impact whether ◎ has the correct XOR of its slots. It has no impact on any other items.
- ***Idea:*** "Assign" ◎ to slot 7.



Filling Our Table

- Notice that slot 7 has only one item (namely, ◎) hashing to it.
- This means that tuning slot 7 whether ◎ has the correct XO impact on any other items.
- ***Idea:*** "Assign" ◎ to slot 7.

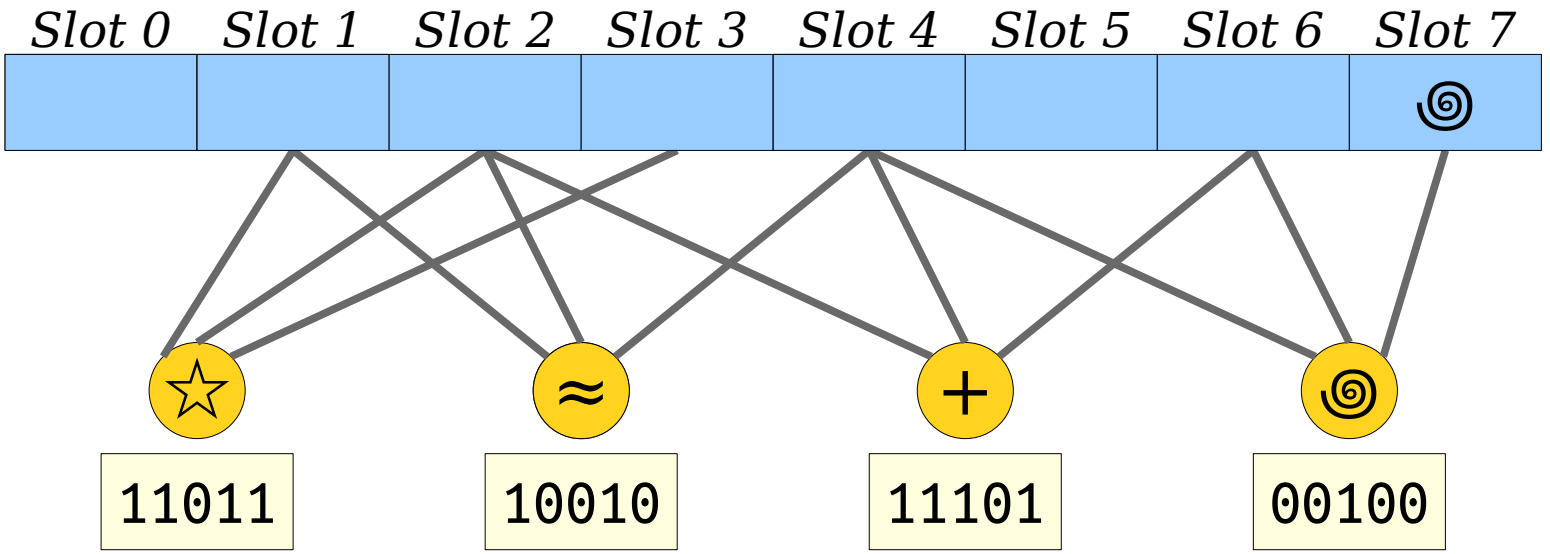
We're not actually storing ◎ here. We're just marking that ◎ owns this slot.



Filling Our Table

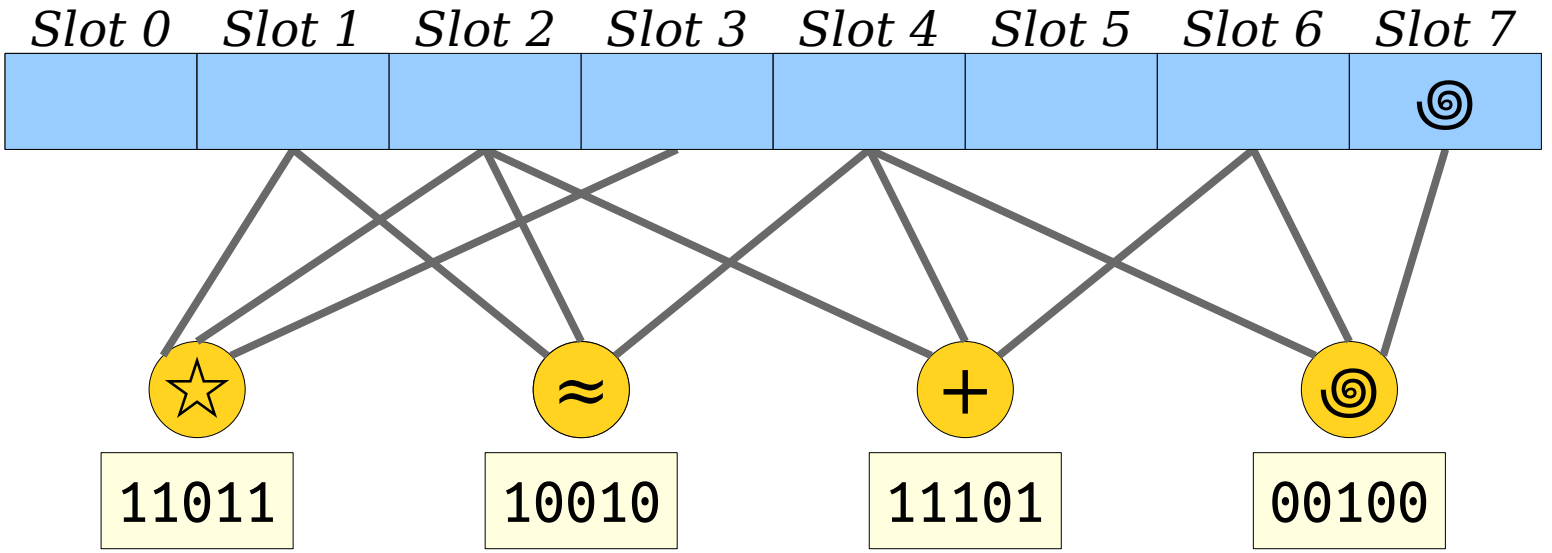
- ***Claim:*** Regardless of the values of the other slots, we can set slot 7's value so that T[4] ⊕ T[6] ⊕ T[7] = `00100`.
- Specifically, set T[7] = T[4] ⊕ T[6] ⊕ `00100`.

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



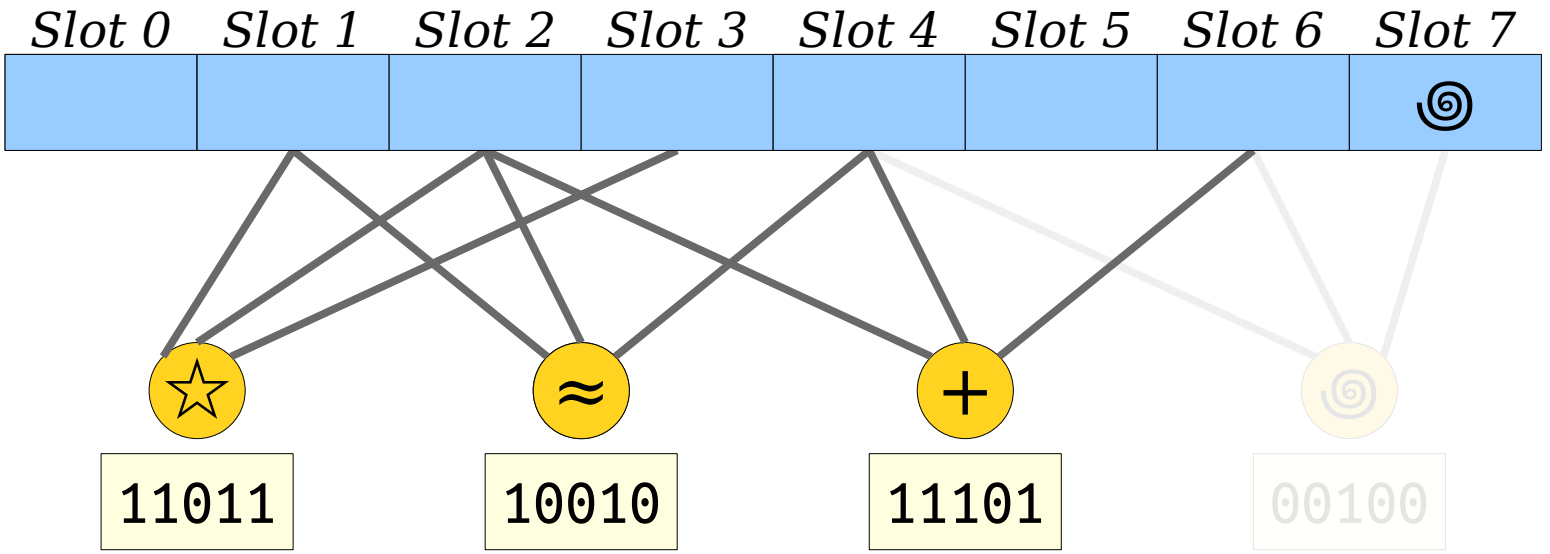
Filling Our Table

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



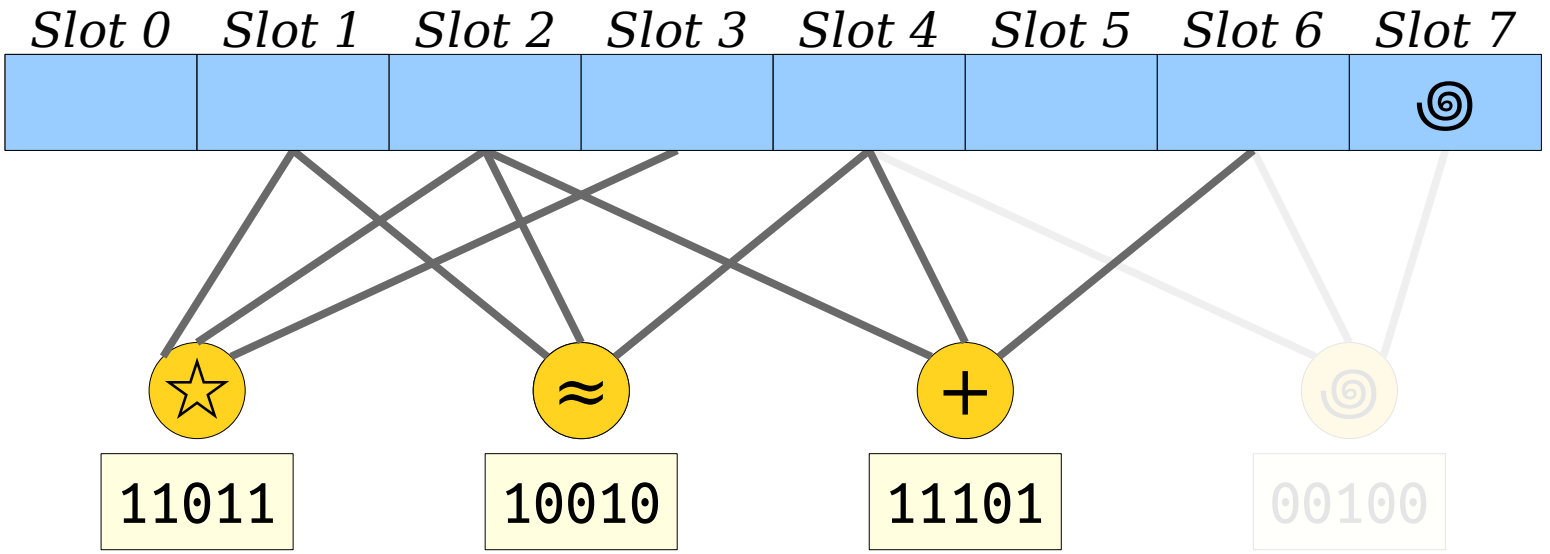
Filling Our Table

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



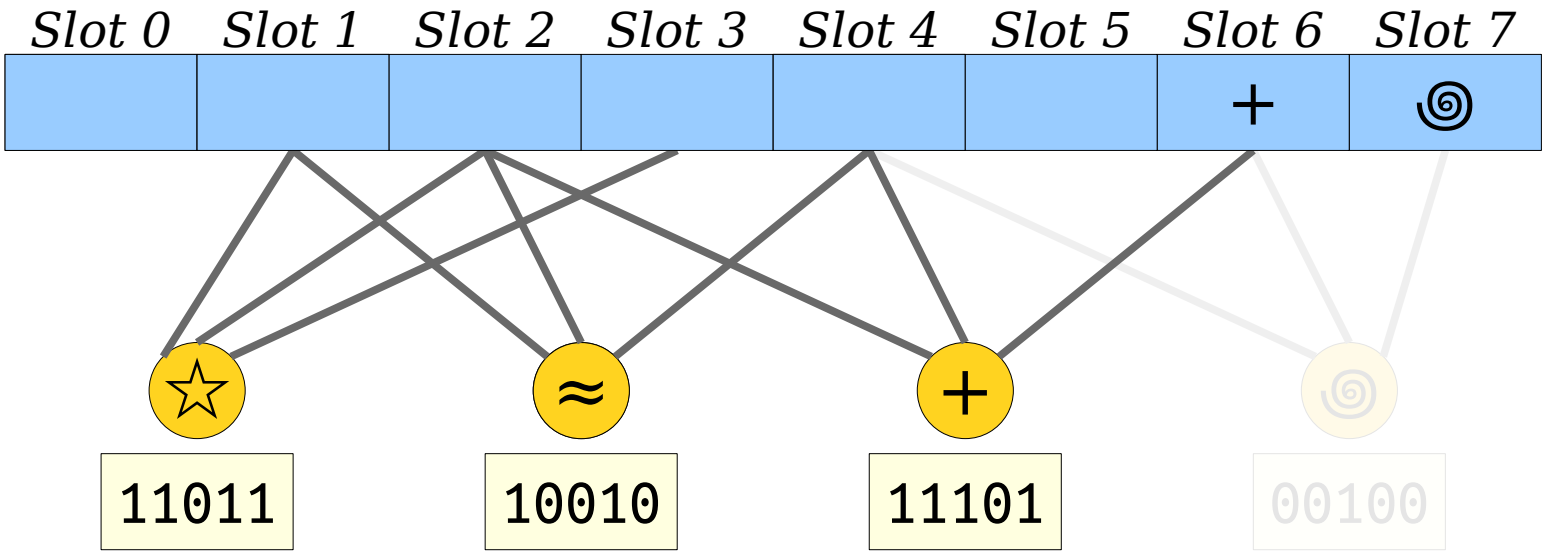
Filling Our Table

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



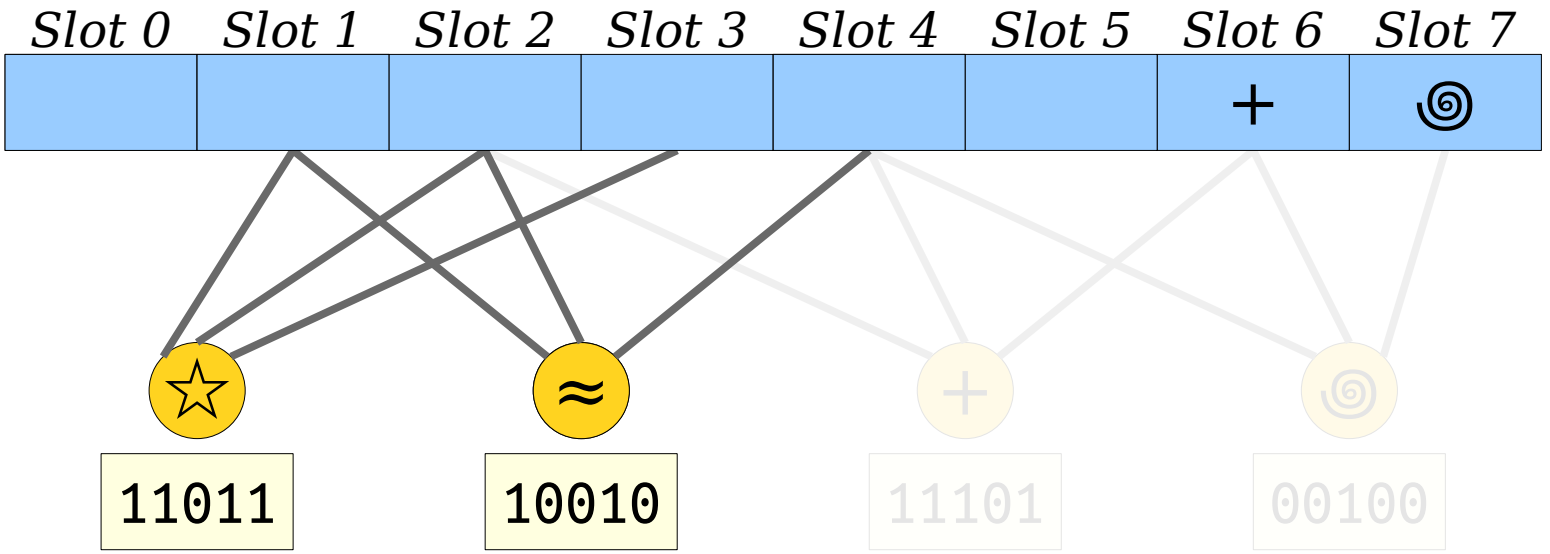
Filling Our Table

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



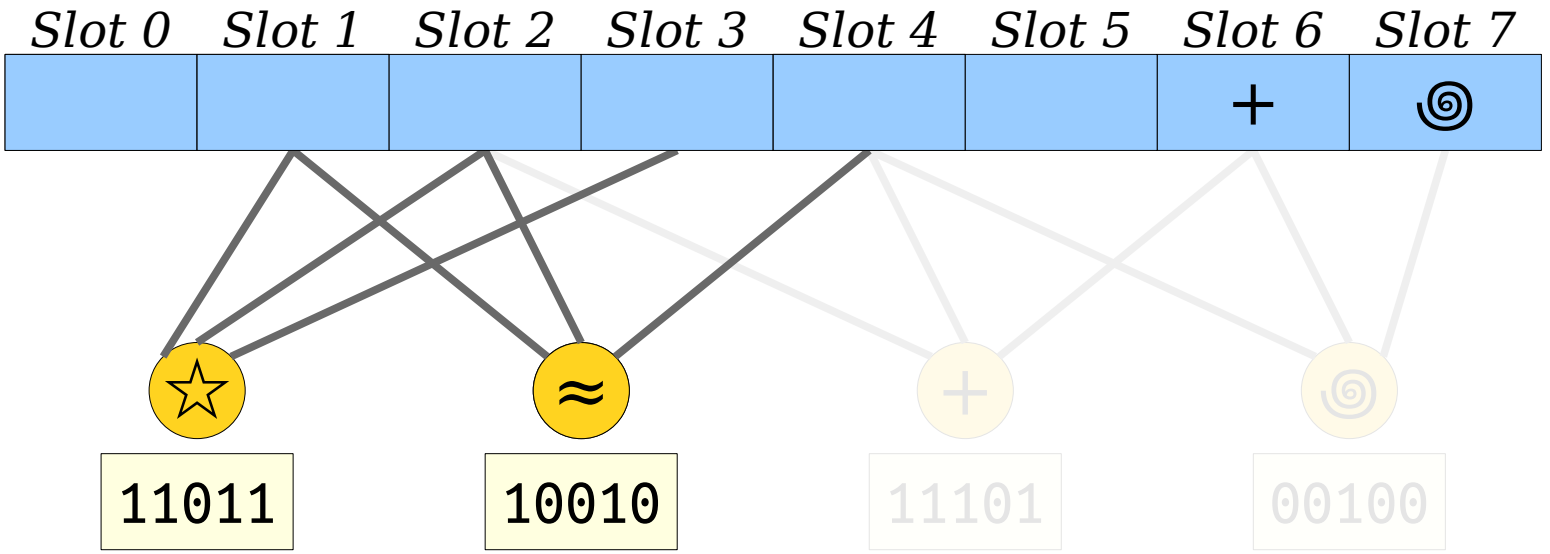
Filling Our Table

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



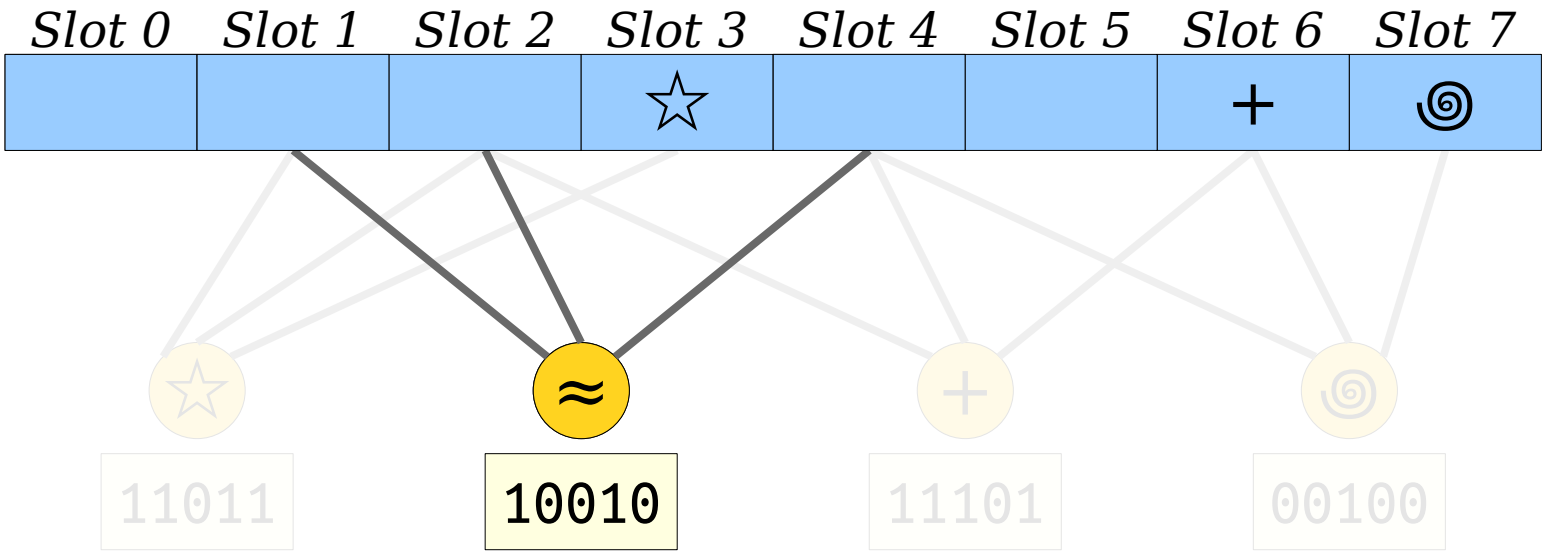
Filling Our Table

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



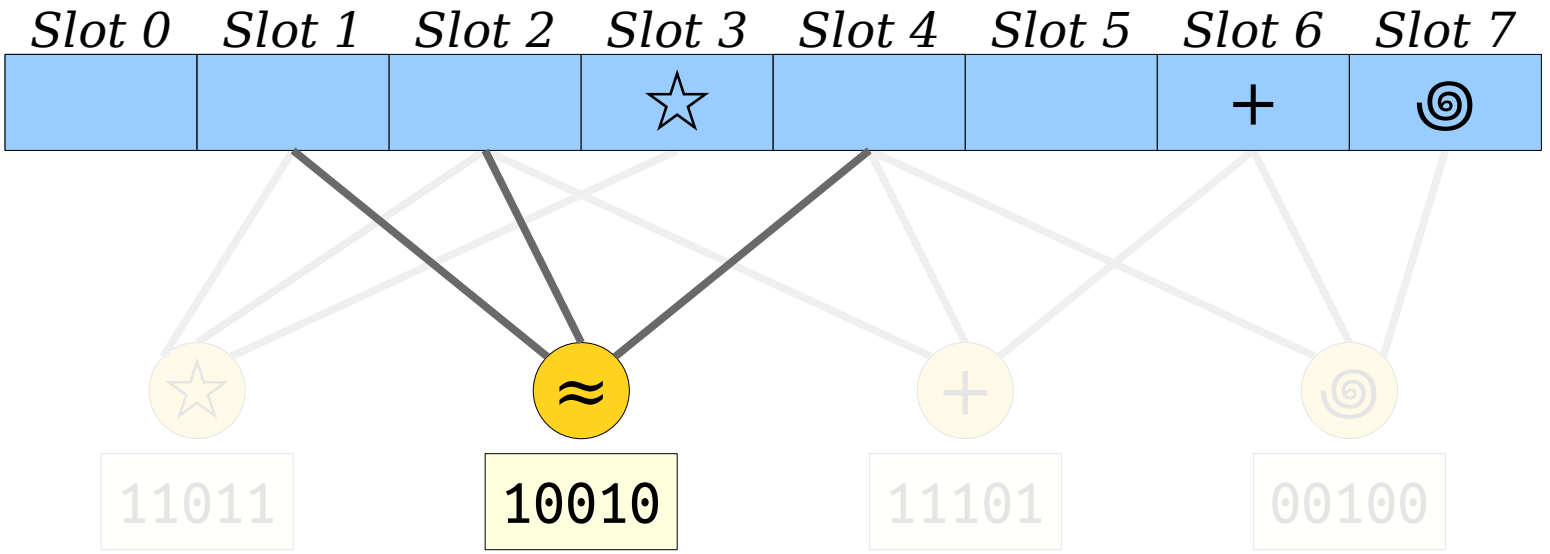
Filling Our Table

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



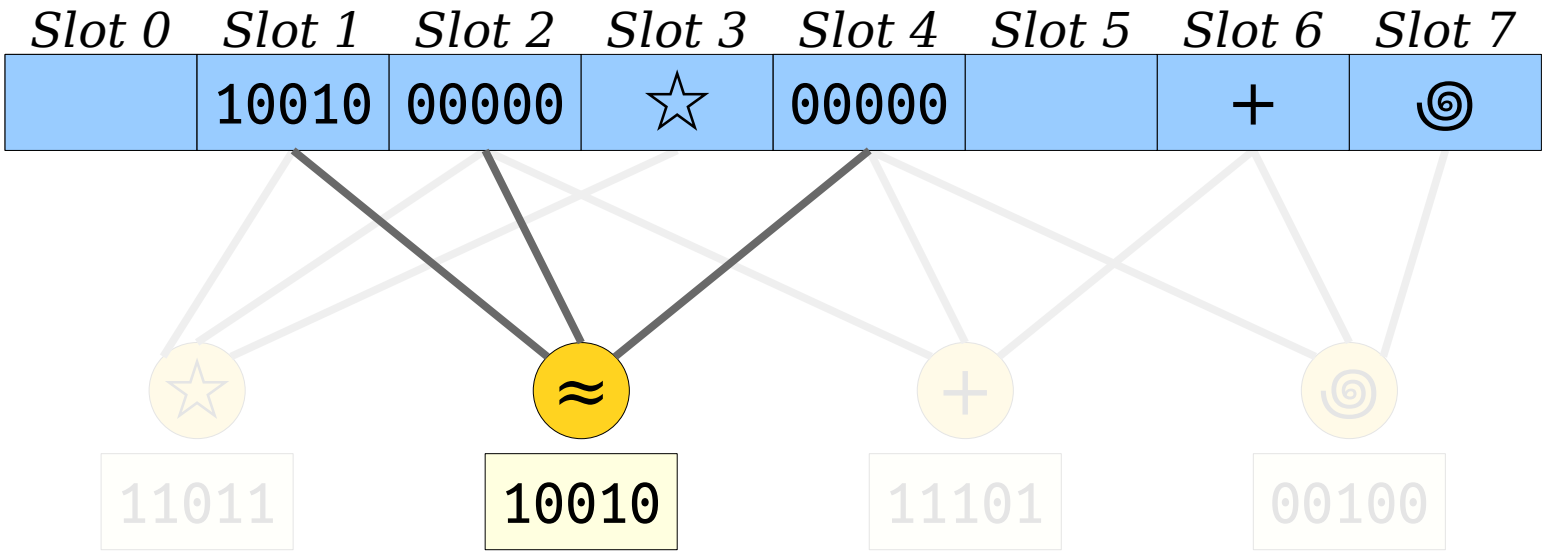
Filling Our Table

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



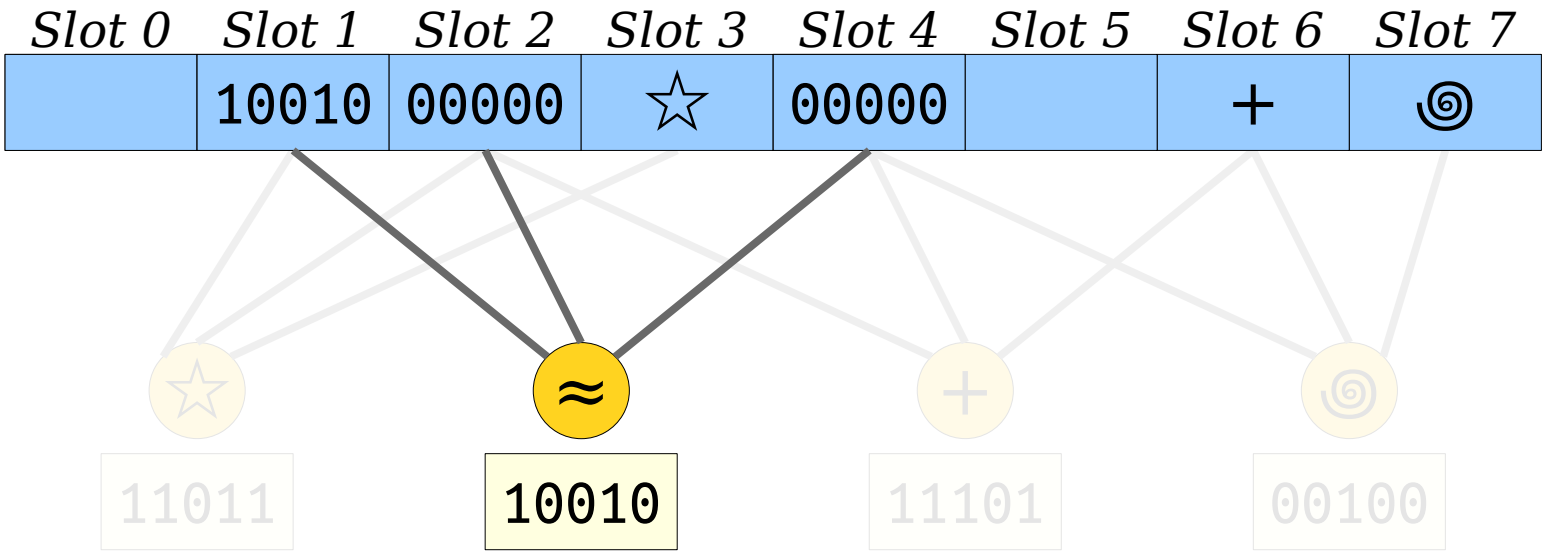
Filling Our Table

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



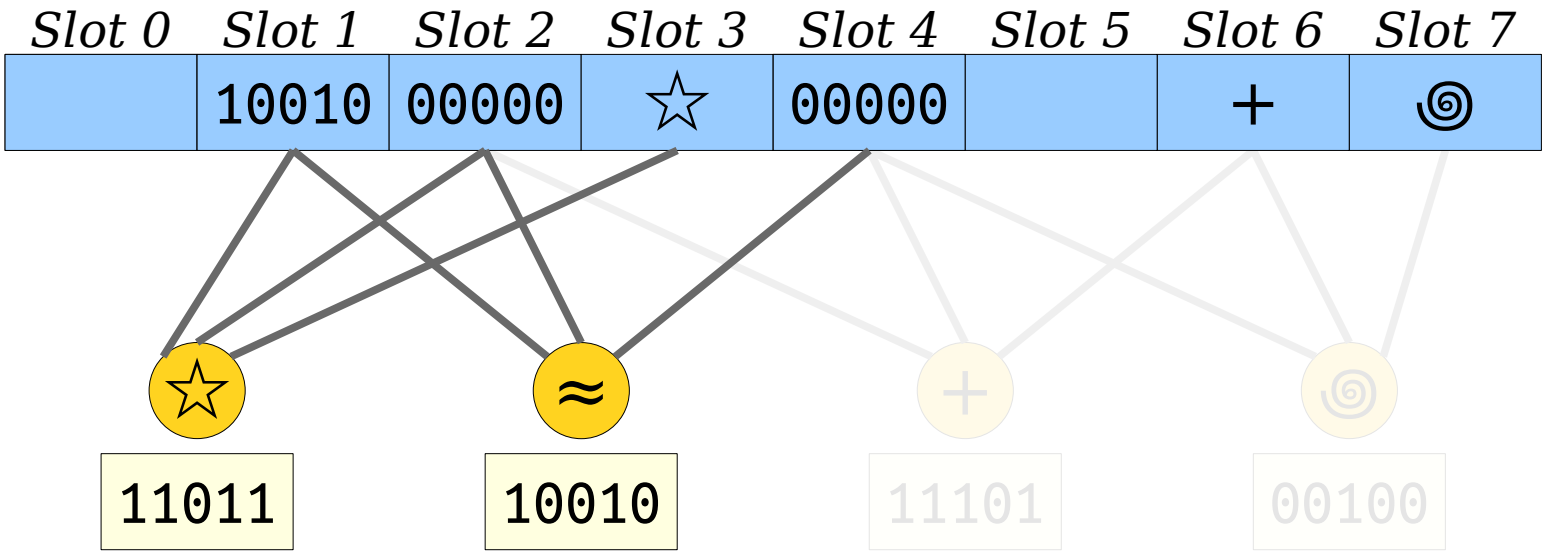
Filling Our Table

- The last item we removed was ☆, so let's add that back in.



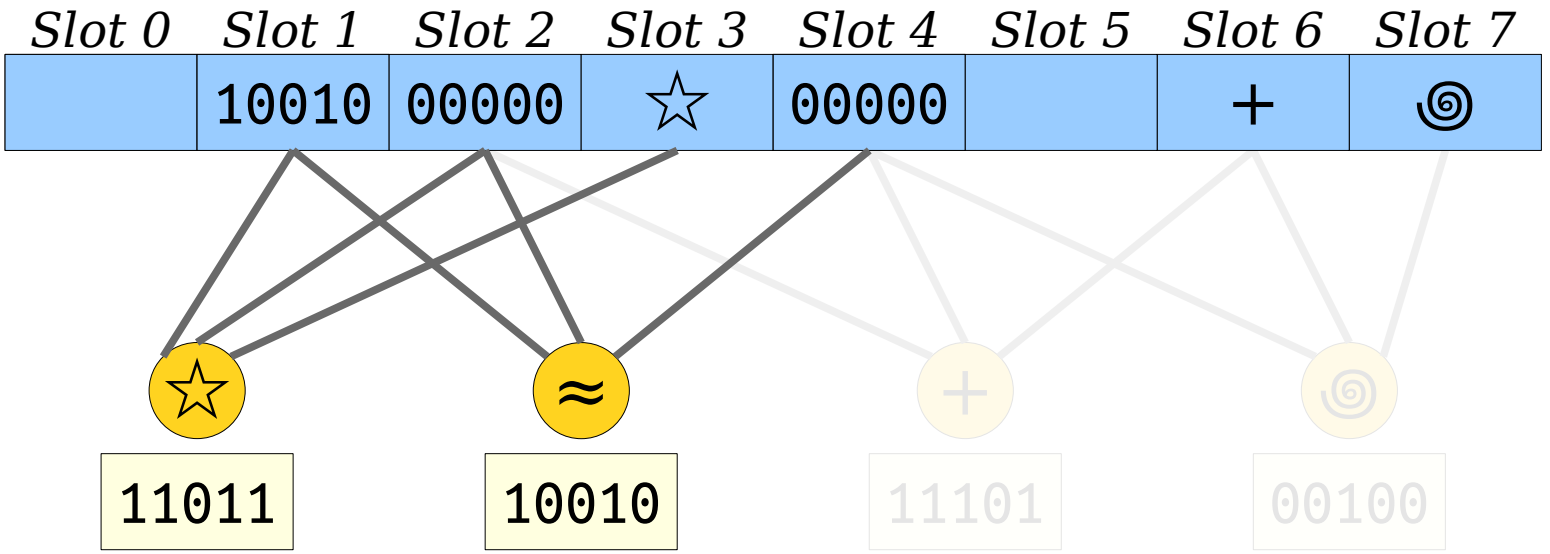
Filling Our Table

- The last item we removed was ☆, so let's add that back in.



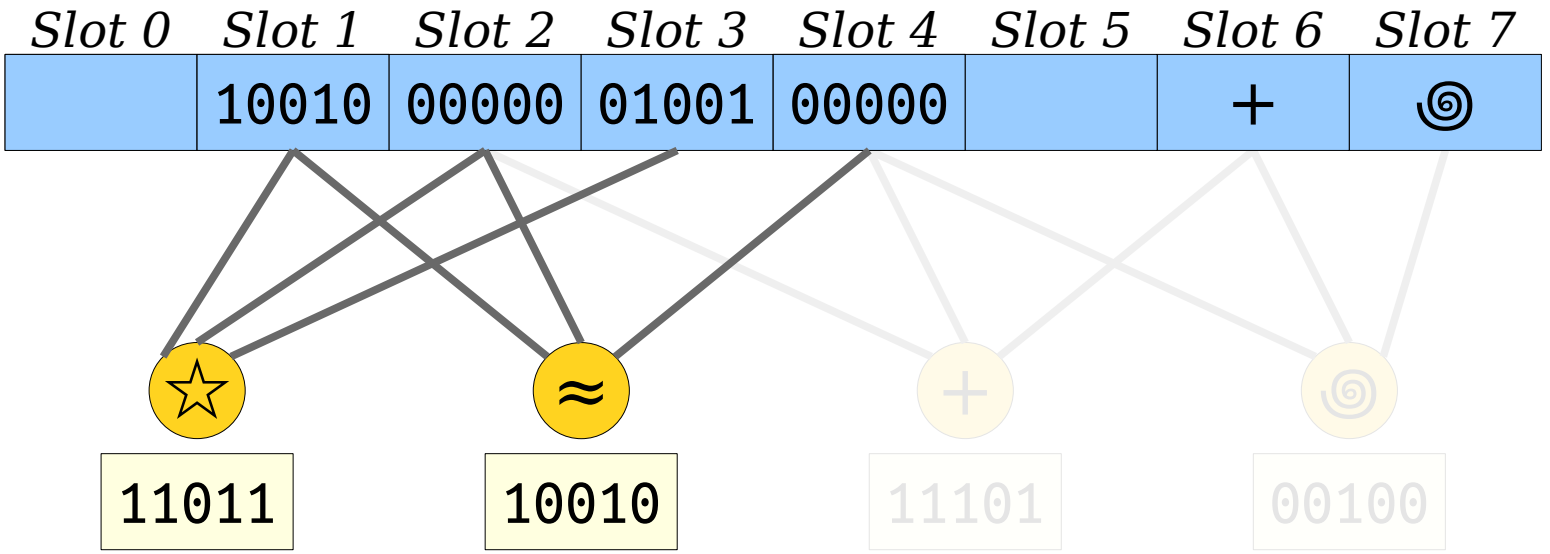
Filling Our Table

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



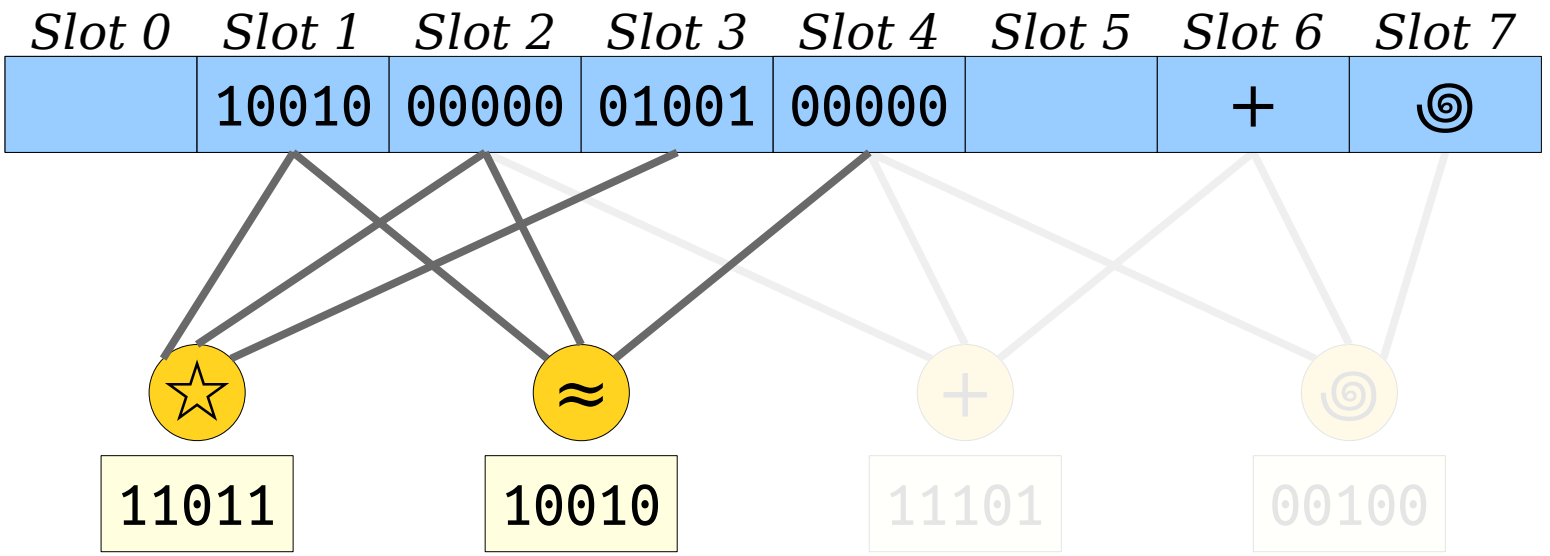
Filling Our Table

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



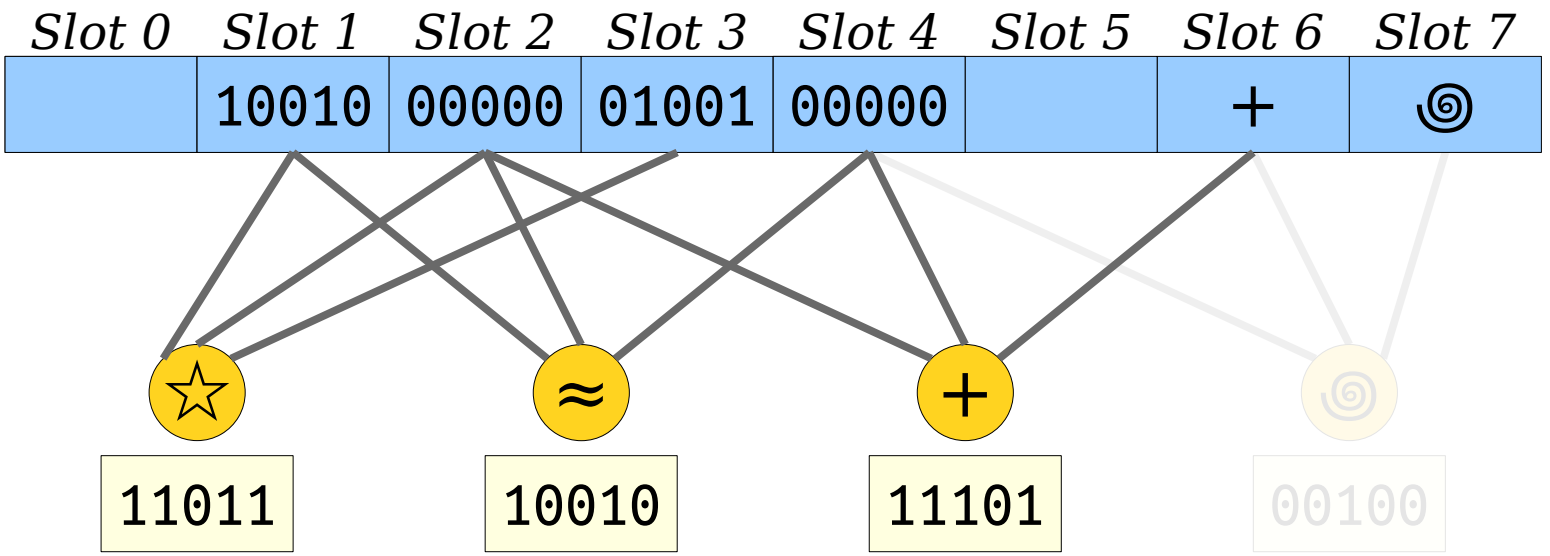
Filling Our Table

- Let’s add the + back in now.



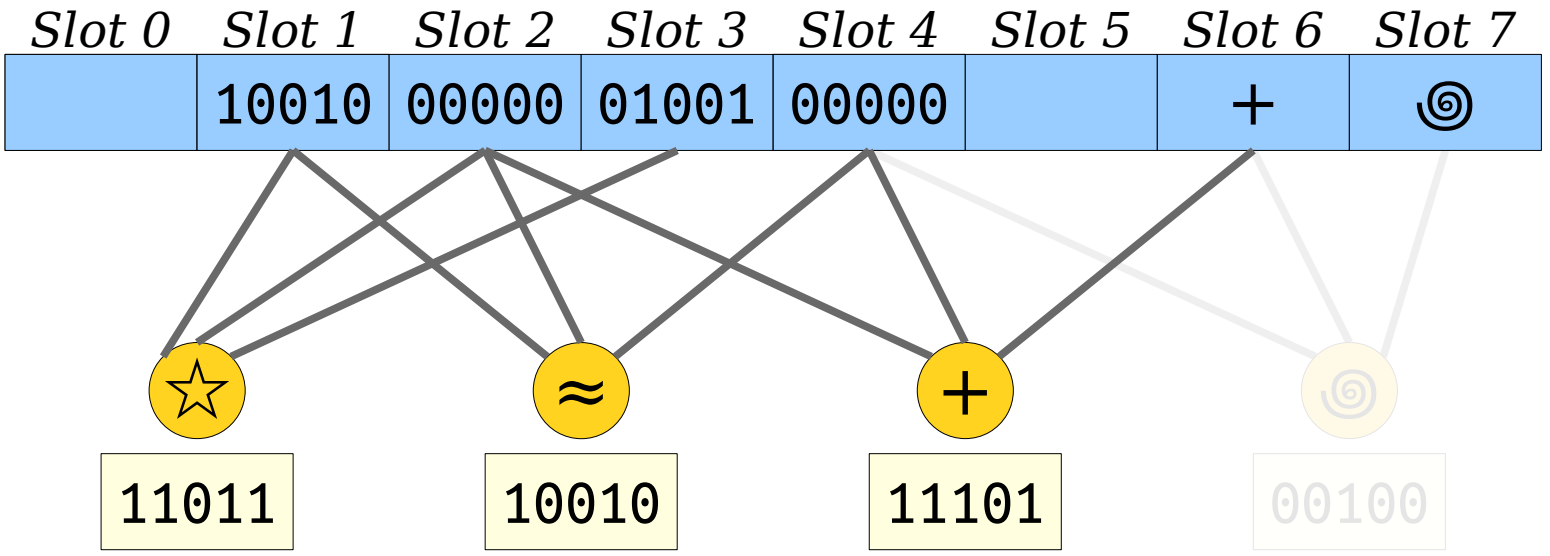
Filling Our Table

- Let’s add the + back in now.



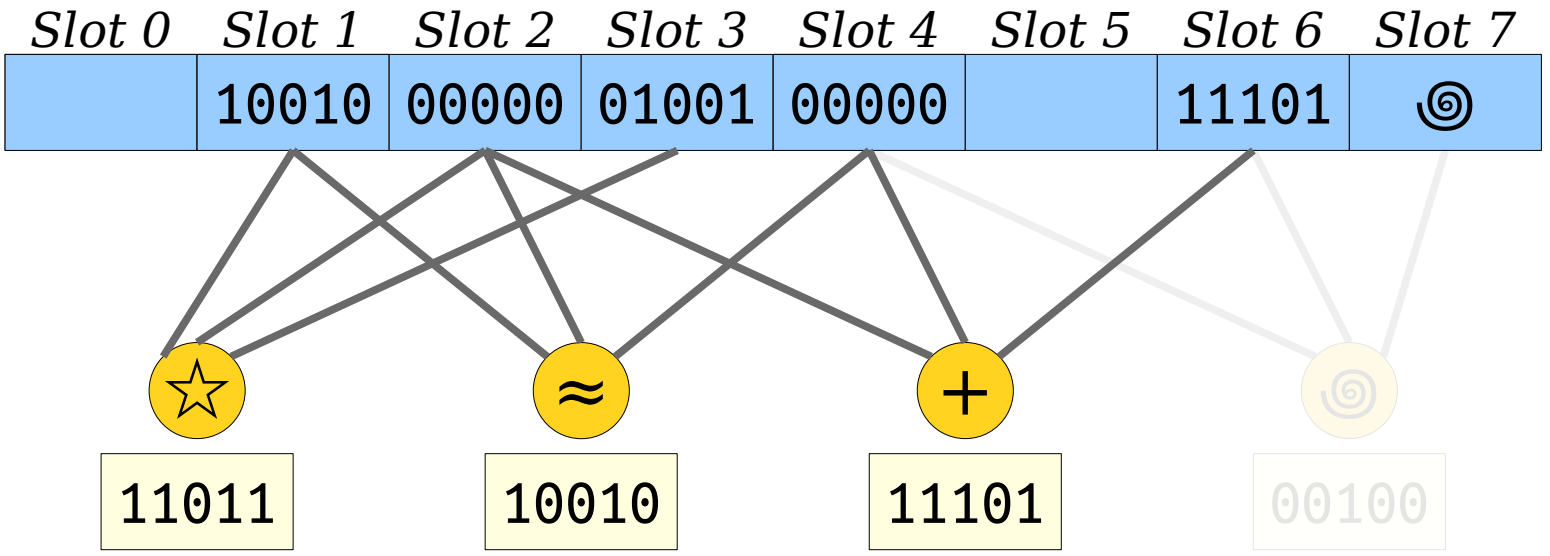
Filling Our Table

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



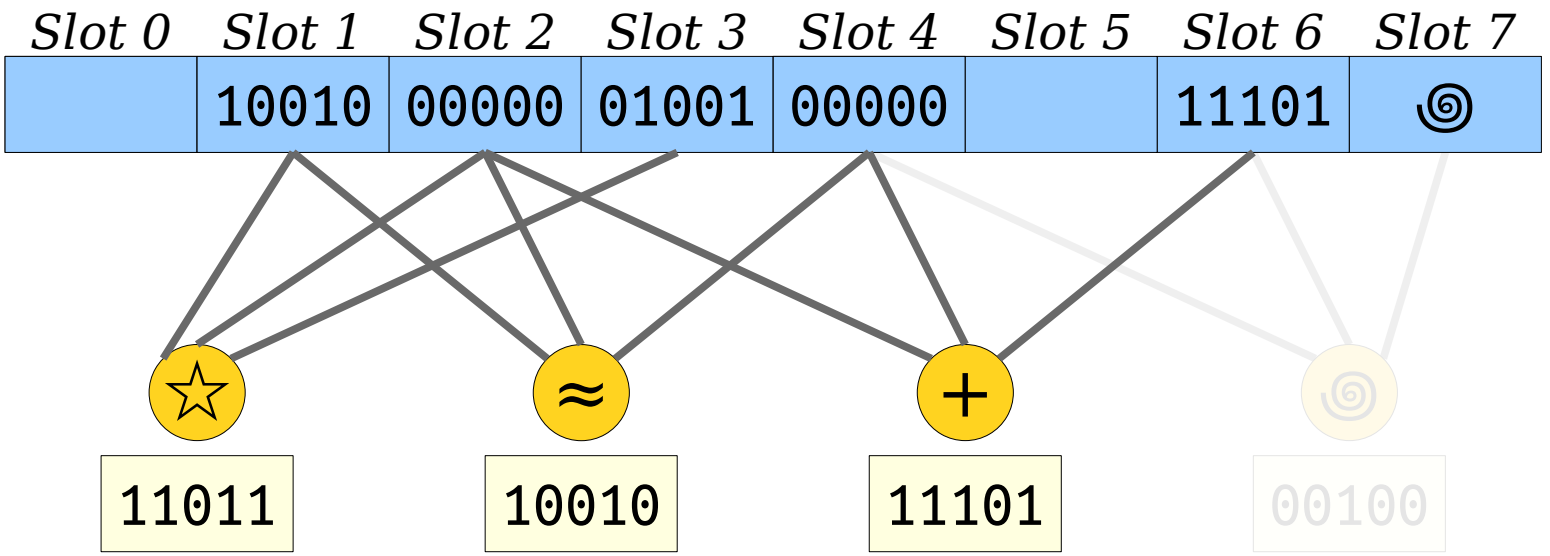
Filling Our Table

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



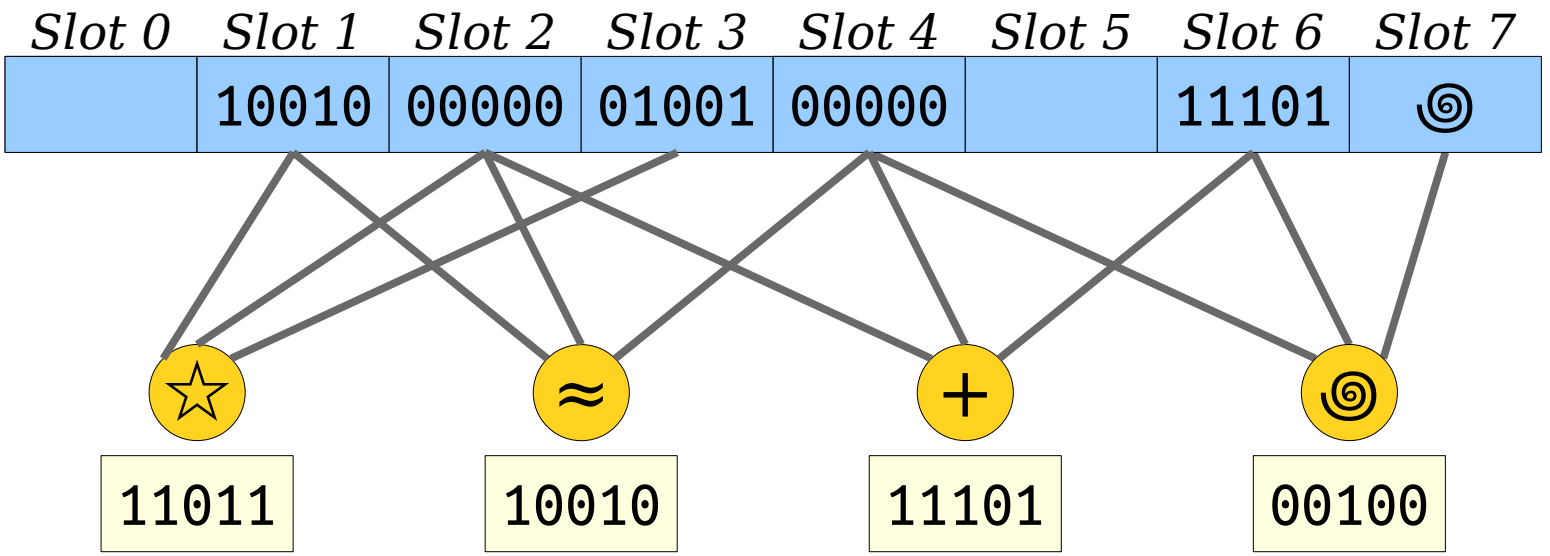
Filling Our Table

- Finally, we add 🌀 back in.



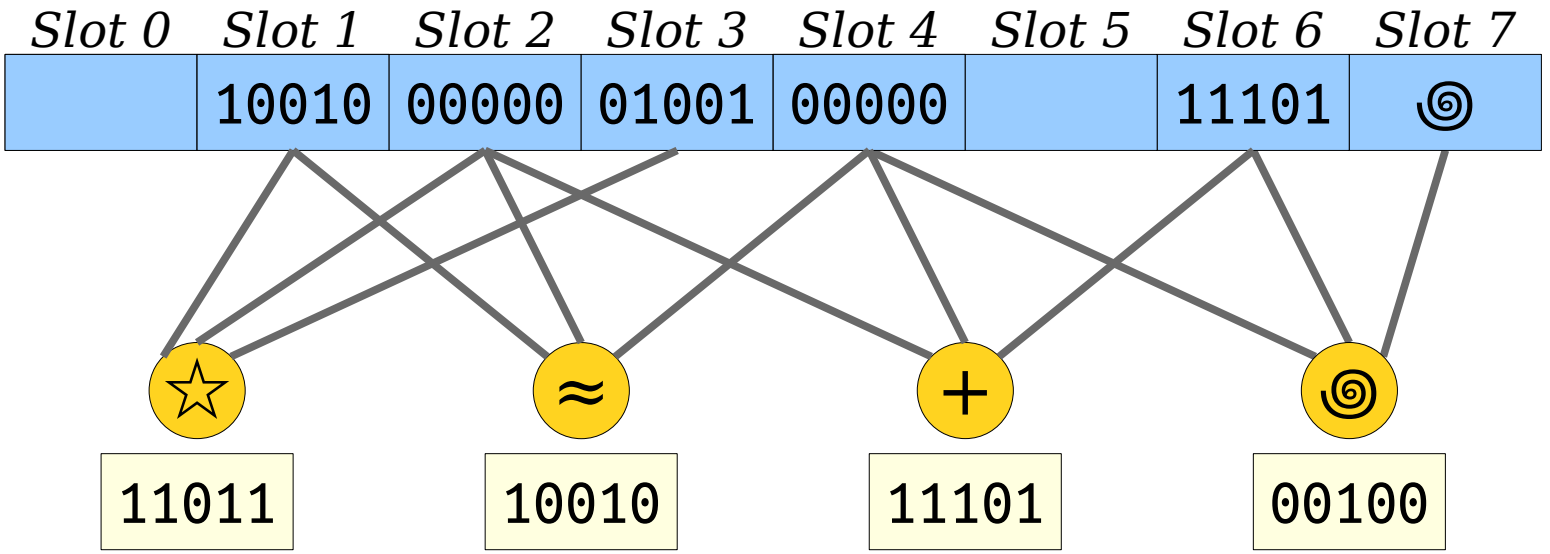
Filling Our Table

- Finally, we add 🌀 back in.



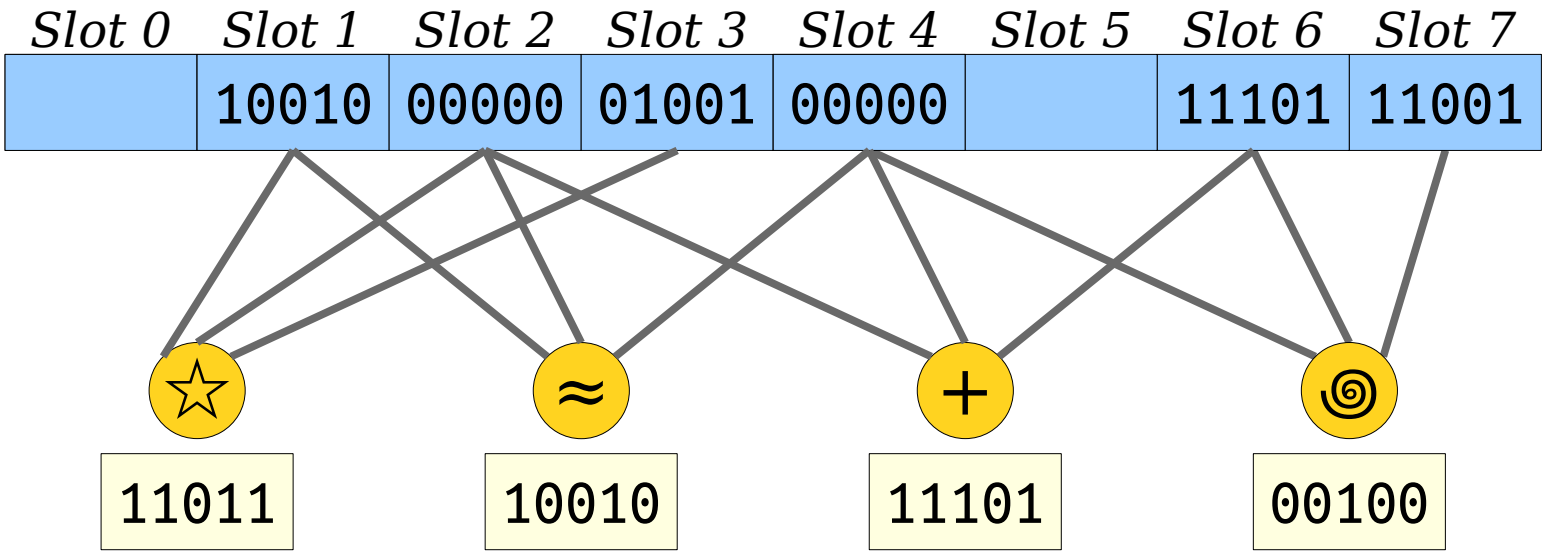
Filling Our Table

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



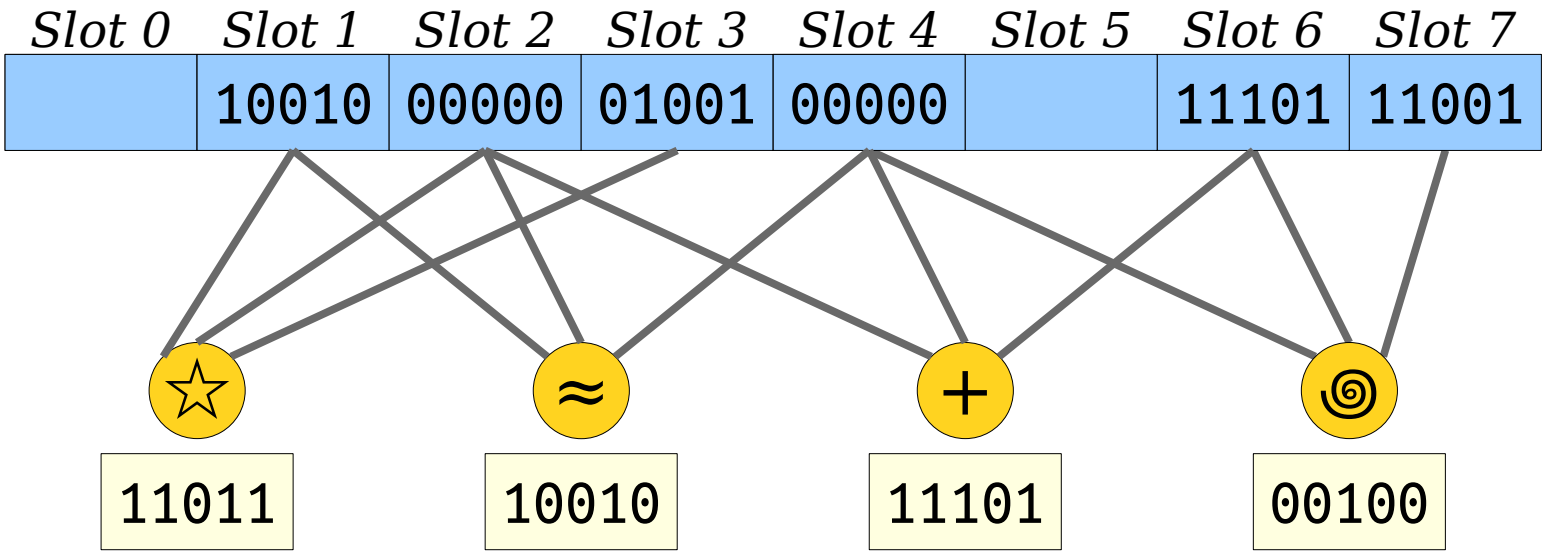
Filling Our Table

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



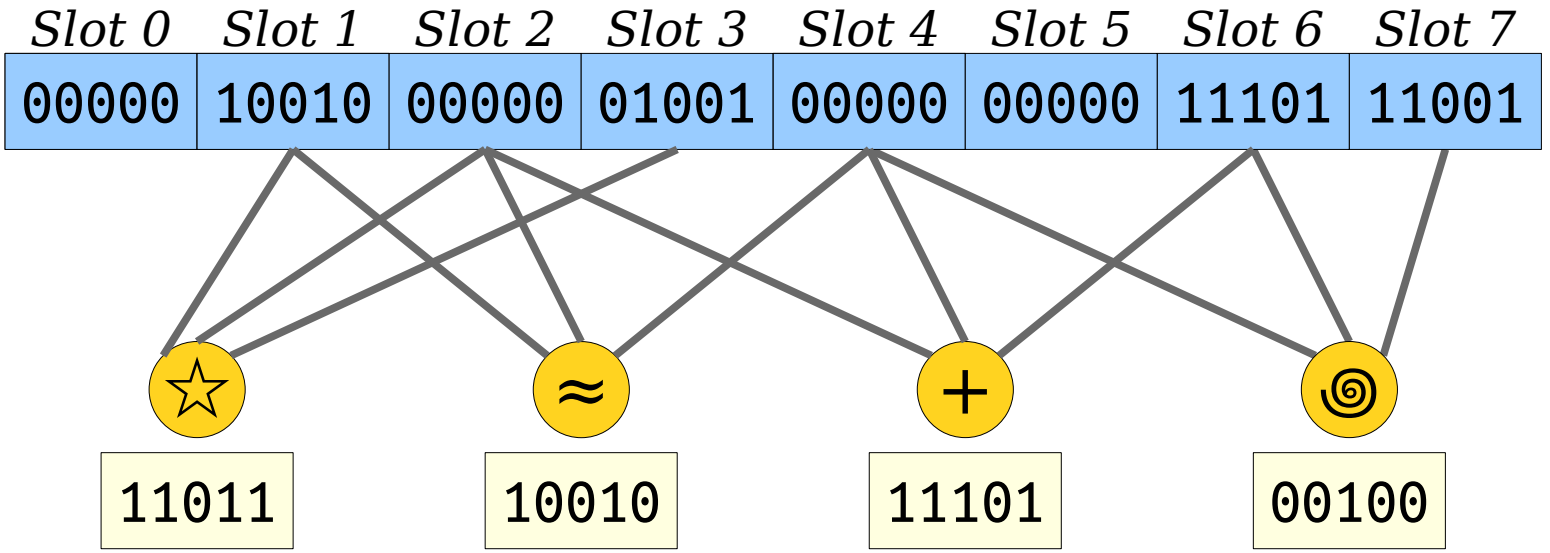
Filling Our Table

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



Filling Our Table

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



Filling Our Table

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

Hypergraph Peeling

- ***This is hypergraph peeling!***
- As we saw last time, for $d \geq 3$, there's an abrupt phase transition from "peeling fails with probability $o(1)$" to "peeling succeeds with probability $o(1)$." Those transition points are as follows:
  - ... for $d = 3$, we have $\alpha_{max} \approx 0.81$.
  - ... for $d = 4$, we have $\alpha_{max} \approx 0.77$.
  - ... for $d = 5$, we have $\alpha_{max} \approx 0.70$.
- Therefore, we'll use $\boldsymbol{d = 3}$ hash functions per item stored. With $\alpha \approx 0.81$, a table with n elements needs around $\boldsymbol{1.23n}$ slots.

The XOR Filter

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

The Story So Far

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

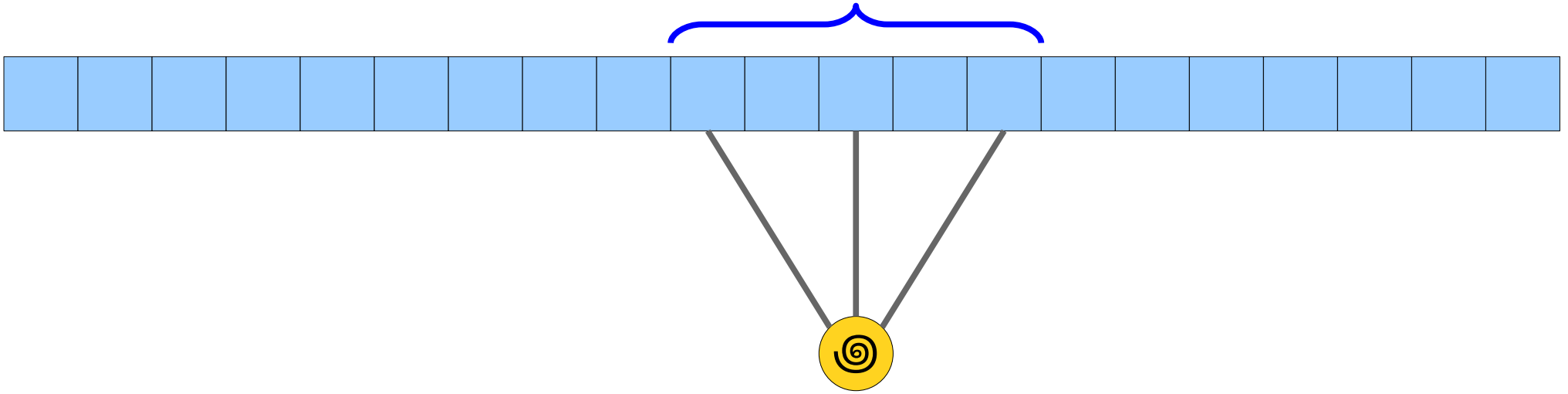
	Bits Per Element	Hashes/Query
Bloom Filter (1970)	$1.44 \lg \varepsilon^{-1}$	$\lg \varepsilon^{-1}$
Cuckoo Filter, $b = 4$ (2014)	$1.02 \lg \varepsilon^{-1} + 3.06$ *(for sufficiently small ε)*	3
XOR Filter (2020)	$1.23 \lg \varepsilon^{-1}$	4

Reducing Space

- The XOR filter uses $1.23n$ table slots to hold n items.
- ***Recall:*** This is because the peeling algorithm stops working for smaller tables.
- This is an inherent property of random hypergraphs, not something particular about our implementation.
- ***Question:*** Can we reduce the space usage further than this?

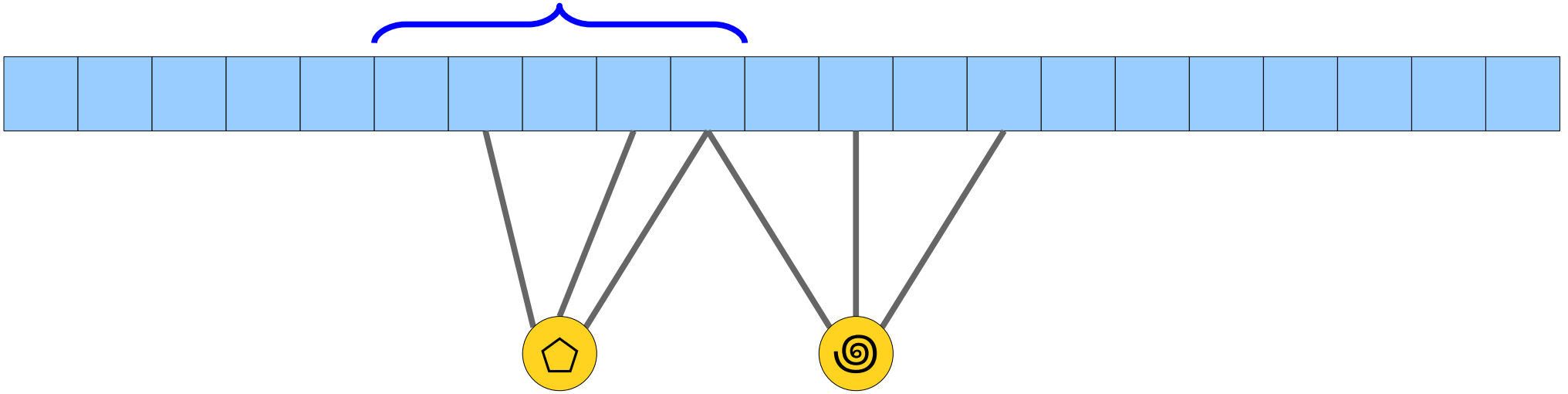
Spatial Coupling

- ***Idea:*** Choose your hashes in the following way:
  - Pick a window size w.
  - Have an initial hash function h_w that picks a location for that window uniformly at random from the array of slots.
  - Have d follow-up hash functions h_1, h_2, …, h_d that pick locations within that window.



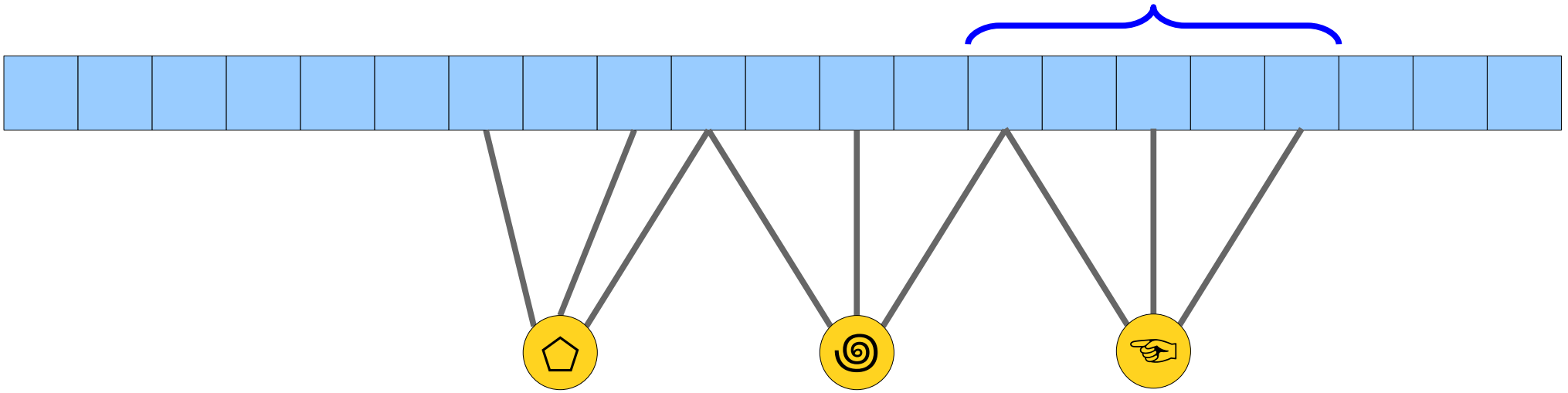
Spatial Coupling

- ***Idea:*** Choose your hashes in the following way:
  - Pick a window size w.
  - Have an initial hash function h_w that picks a location for that window uniformly at random from the array of slots.
  - Have d follow-up hash functions h_1, h_2, …, h_d that pick locations within that window.



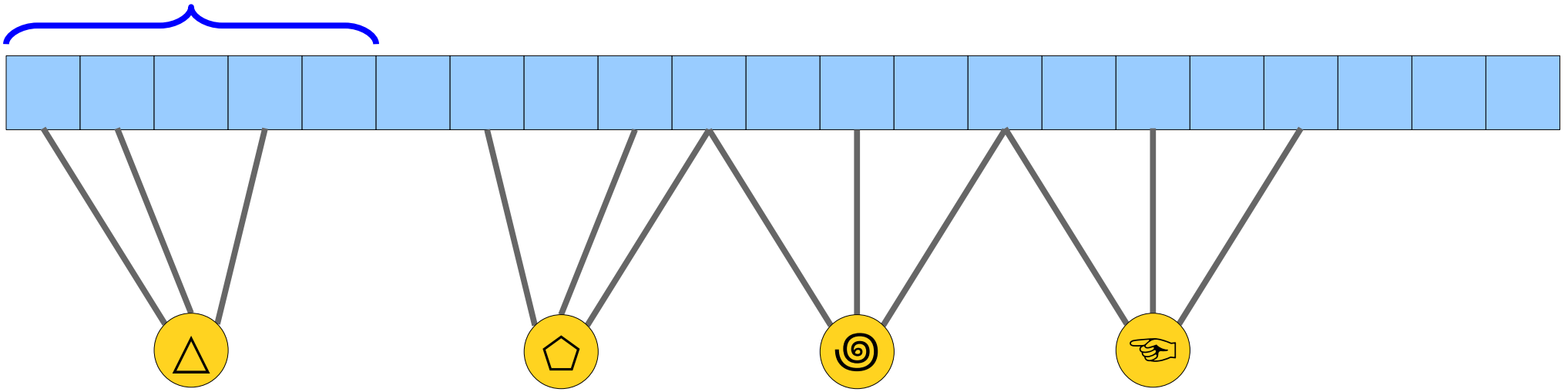
Spatial Coupling

- ***Idea:*** Choose your hashes in the following way:
  - Pick a window size w.
  - Have an initial hash function h_w that picks a location for that window uniformly at random from the array of slots.
  - Have d follow-up hash functions h_1, h_2, …, h_d that pick locations within that window.



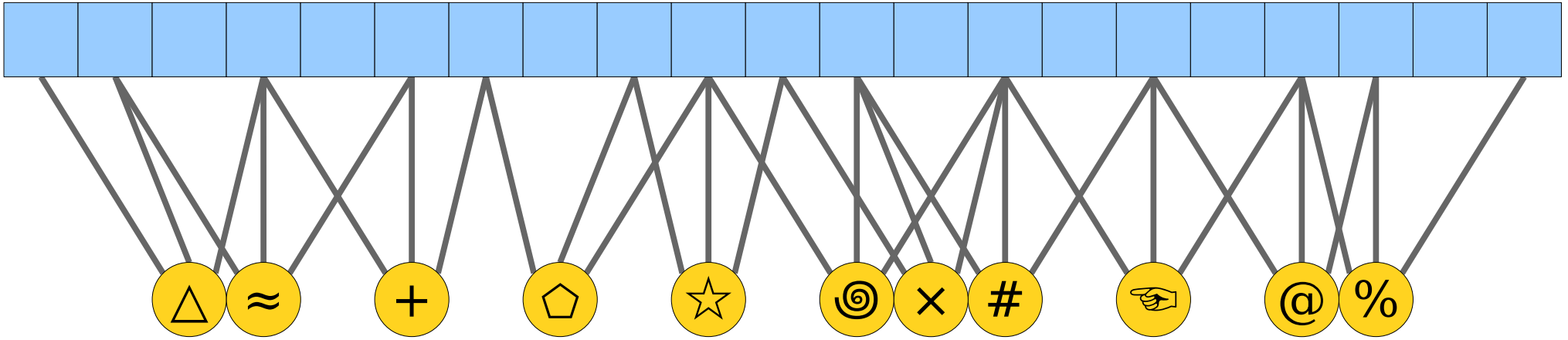
Spatial Coupling

- ***Idea:*** Choose your hashes in the following way:
  - Pick a window size w.
  - Have an initial hash function h_w that picks a location for that window uniformly at random from the array of slots.
  - Have d follow-up hash functions h_1, h_2, …, h_d that pick locations within that window.



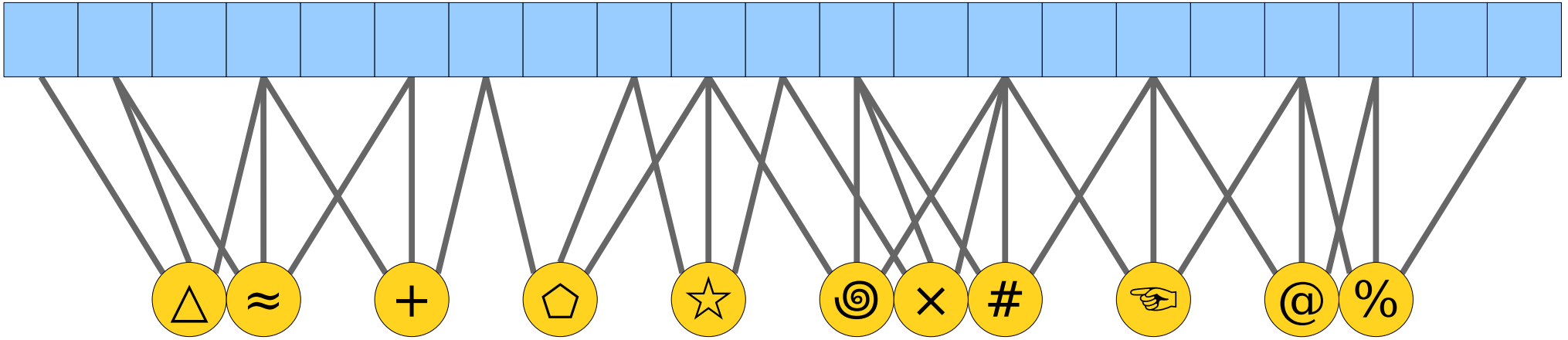
Spatial Coupling

- ***Idea:*** Choose your hashes in the following way:
  - Pick a window size w.
  - Have an initial hash function h_w that picks a location for that window uniformly at random from the array of slots.
  - Have d follow-up hash functions h_1, h_2, …, h_d that pick locations within that window.



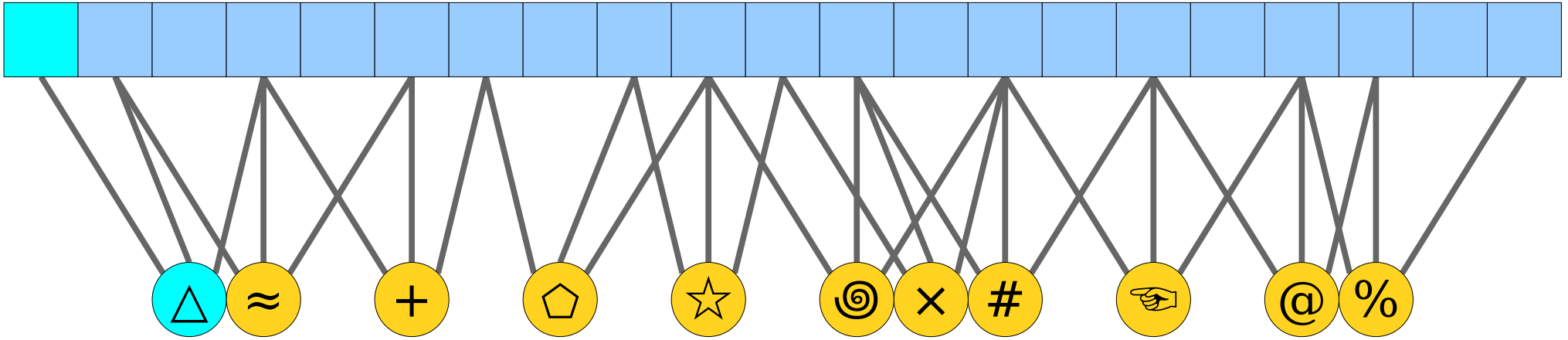
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.



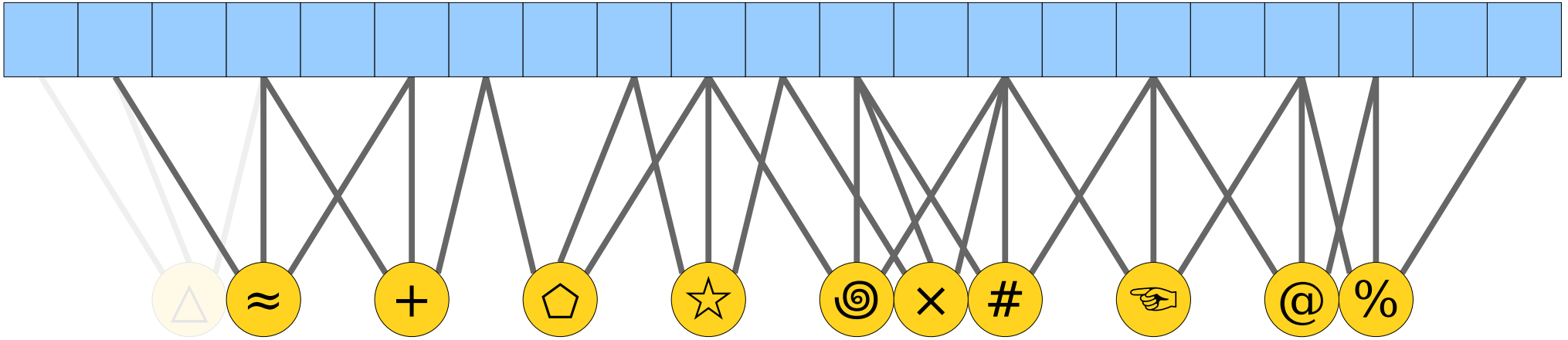
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.



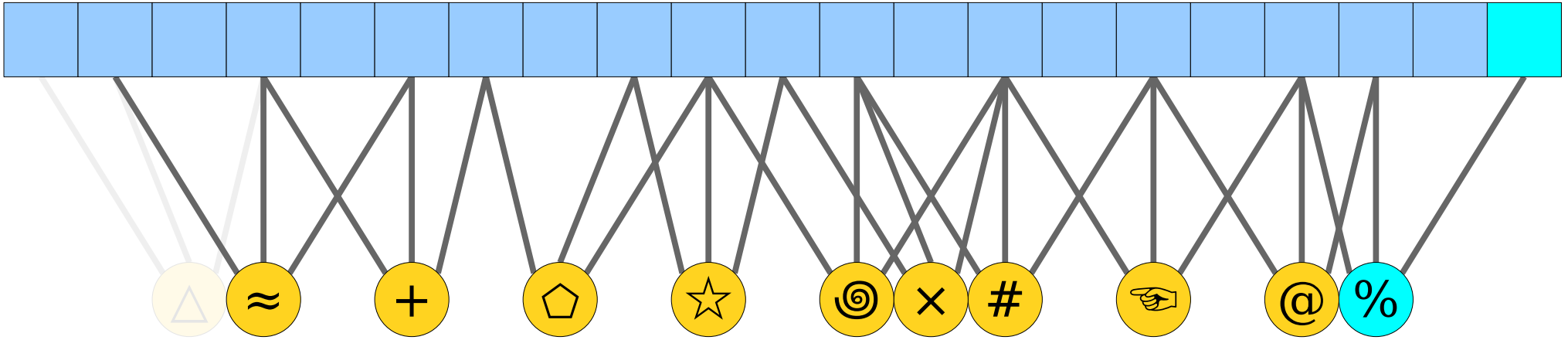
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.



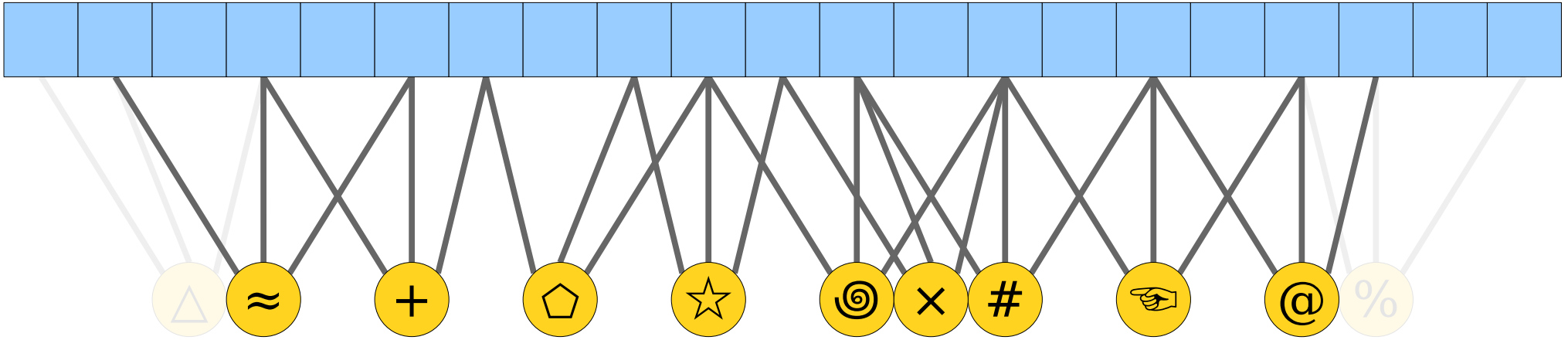
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.



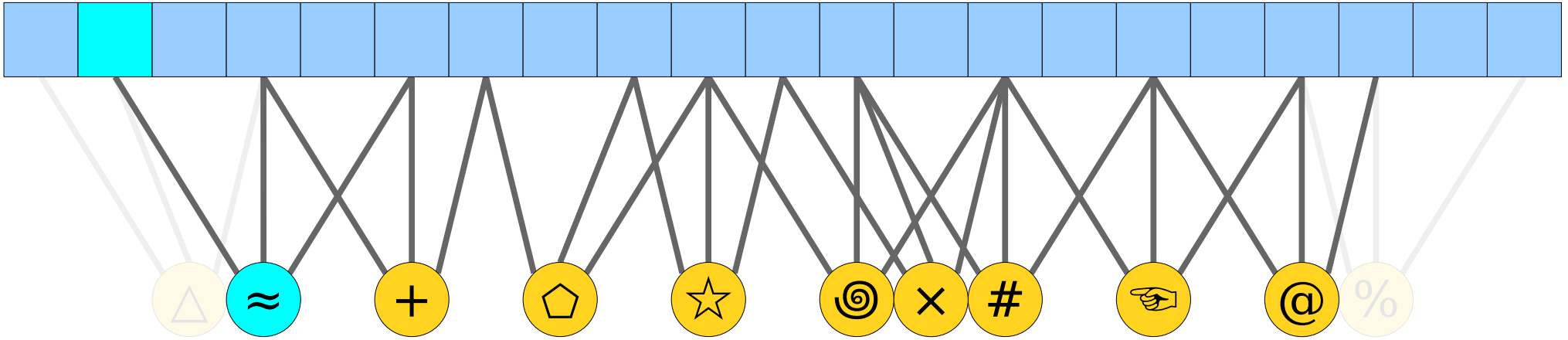
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
- That will likely expose items in slots slightly further from the edges.



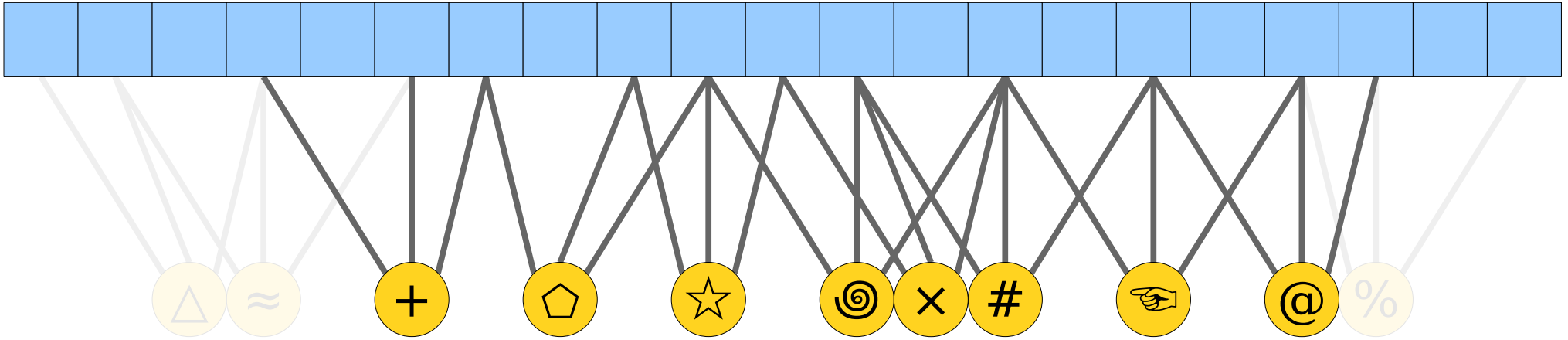
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
- That will likely expose items in slots slightly further from the edges.



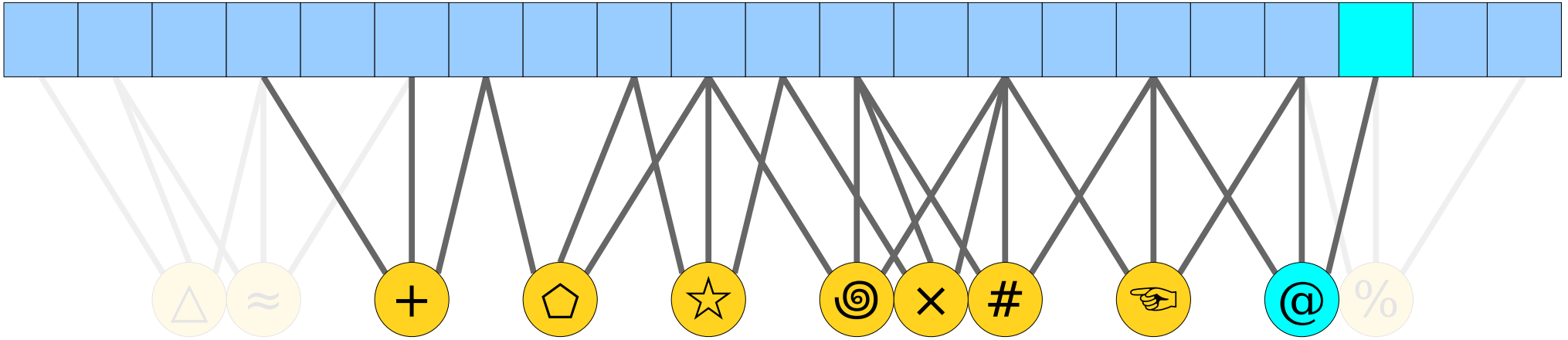
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
- That will likely expose items in slots slightly further from the edges.



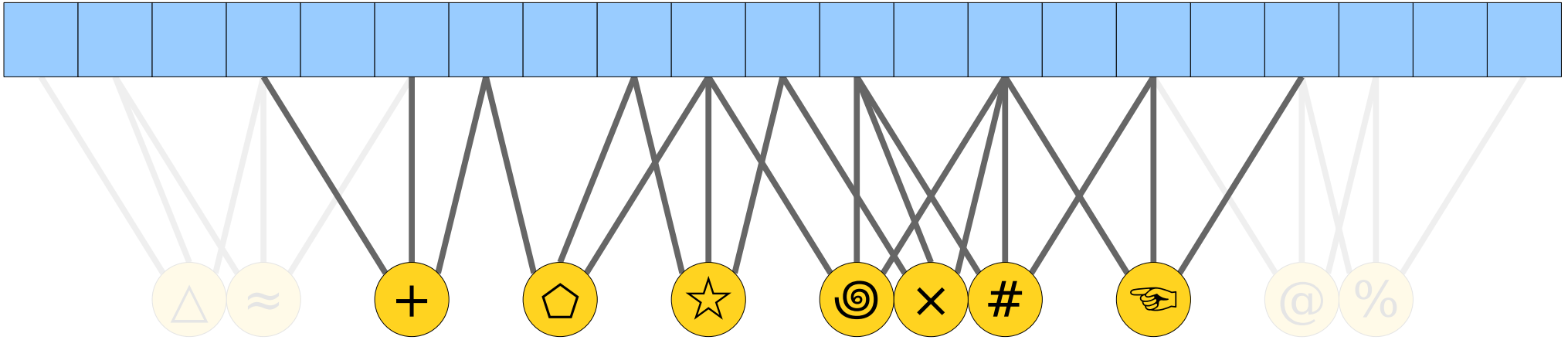
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
- That will likely expose items in slots slightly further from the edges.



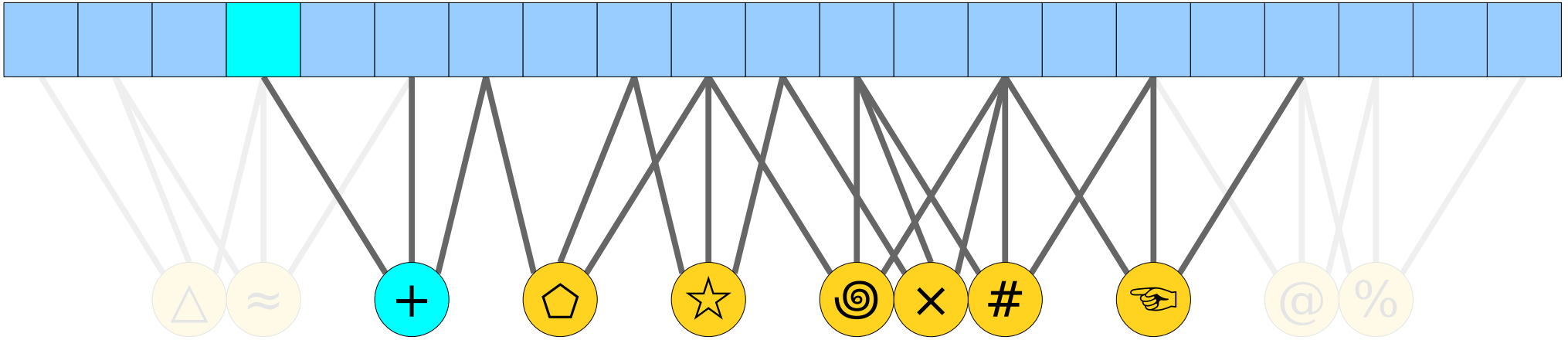
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
- That will likely expose items in slots slightly further from the edges.
- ***Intuition:*** Items will peel from the outsides in.



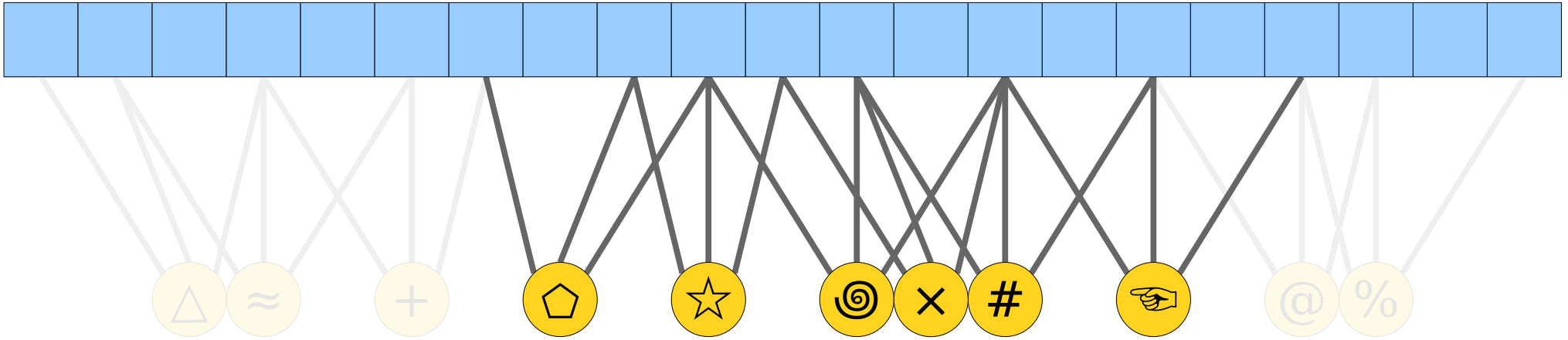
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
- That will likely expose items in slots slightly further from the edges.
- ***Intuition:*** Items will peel from the outsides in.



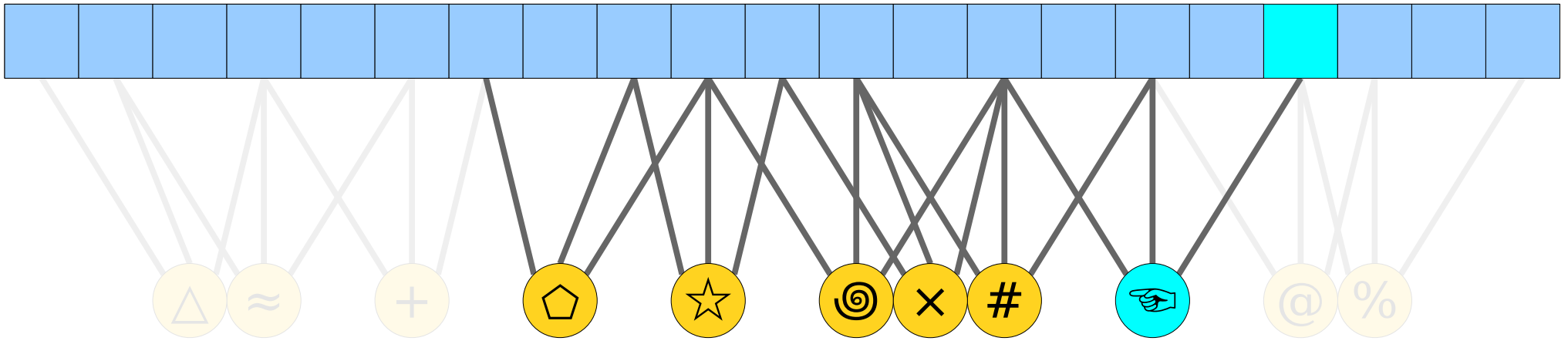
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
- That will likely expose items in slots slightly further from the edges.
- ***Intuition:*** Items will peel from the outsides in.



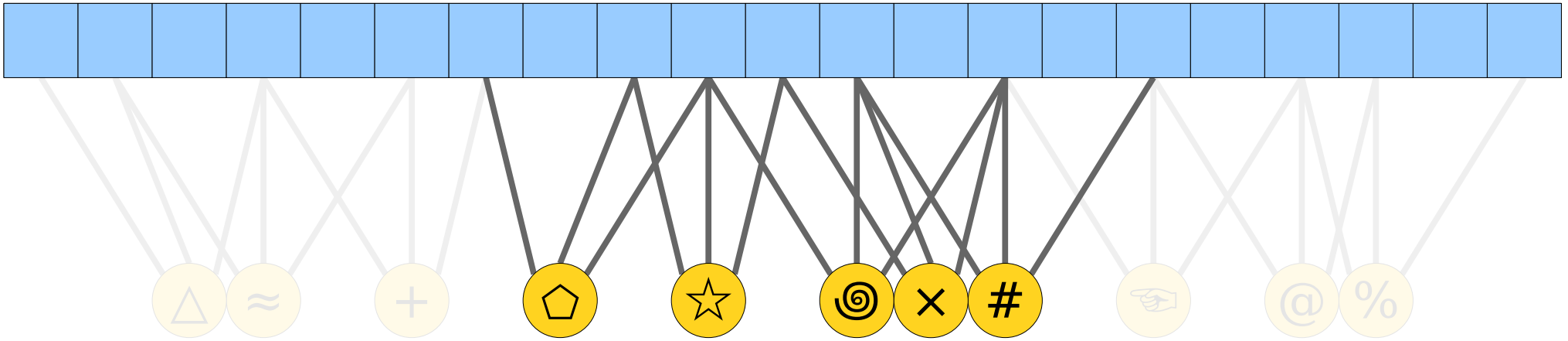
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
- That will likely expose items in slots slightly further from the edges.
- ***Intuition:*** Items will peel from the outsides in.



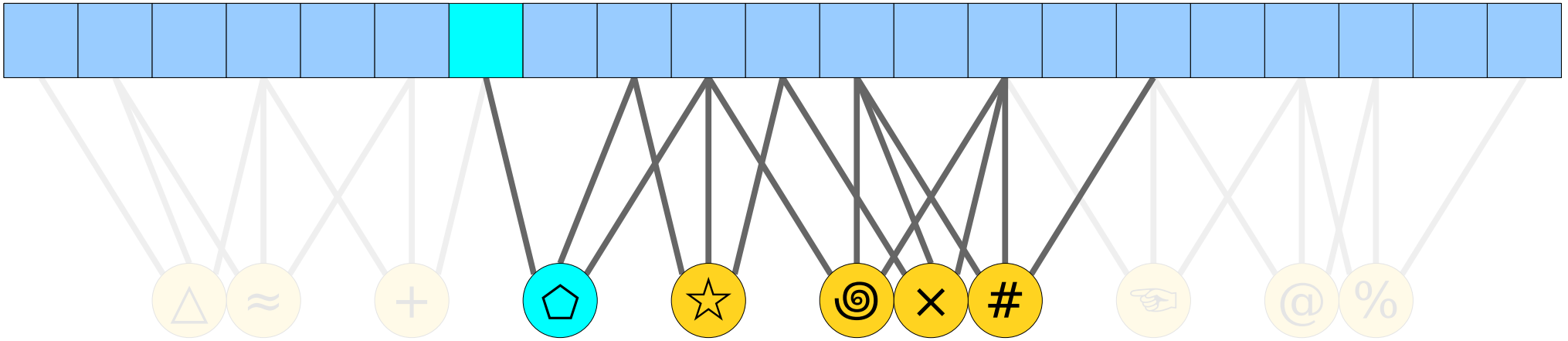
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
- That will likely expose items in slots slightly further from the edges.
- ***Intuition:*** Items will peel from the outsides in.



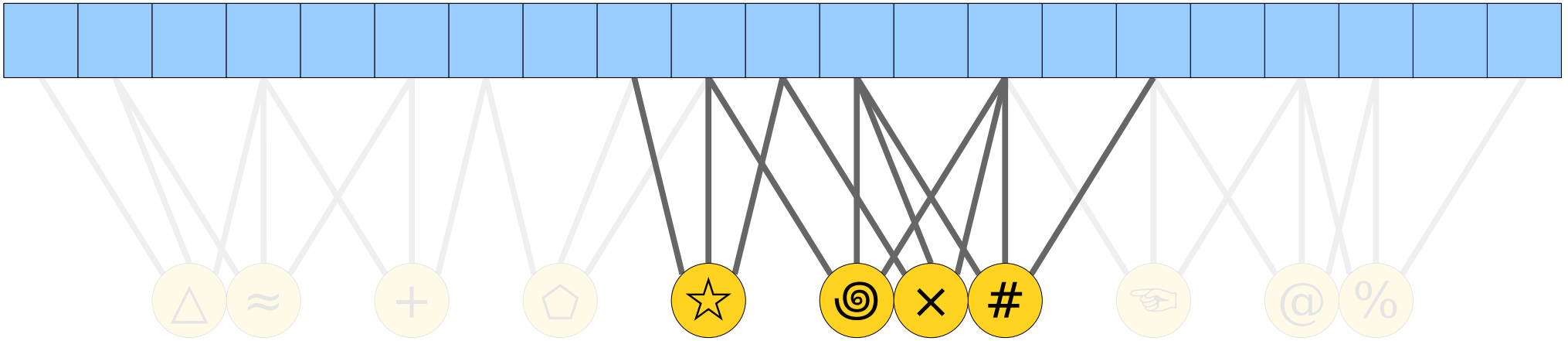
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
- That will likely expose items in slots slightly further from the edges.
- ***Intuition:*** Items will peel from the outsides in.



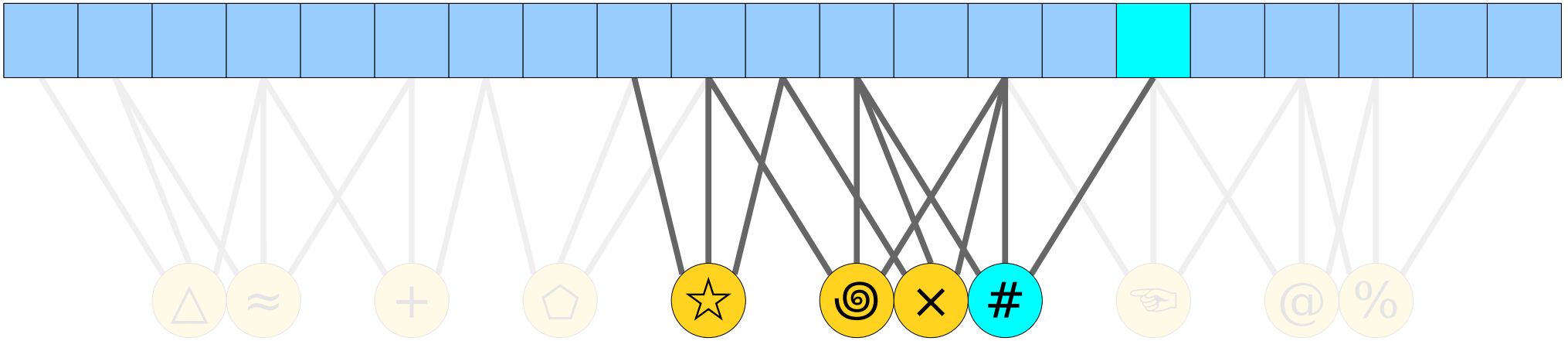
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
- That will likely expose items in slots slightly further from the edges.
- ***Intuition:*** Items will peel from the outsides in.



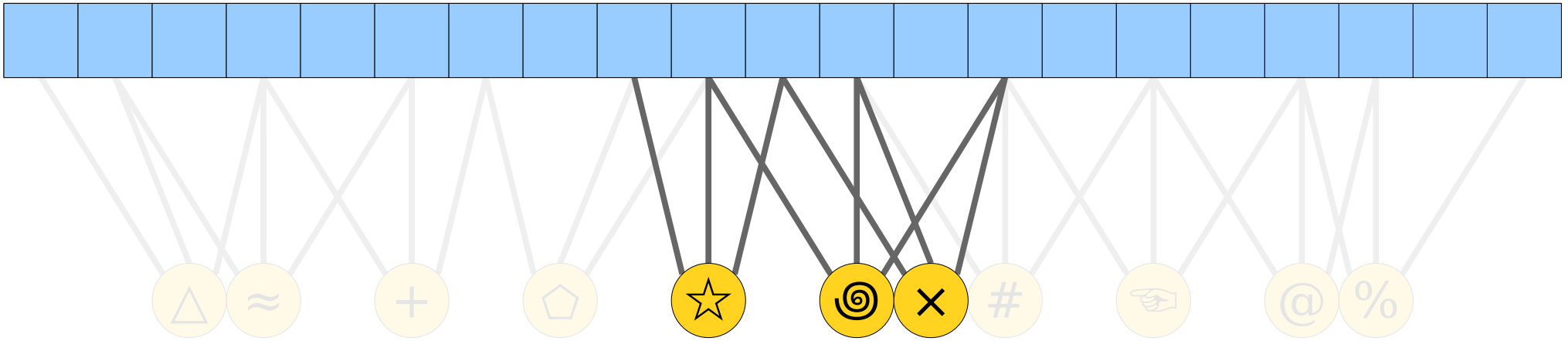
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
- That will likely expose items in slots slightly further from the edges.
- ***Intuition:*** Items will peel from the outsides in.



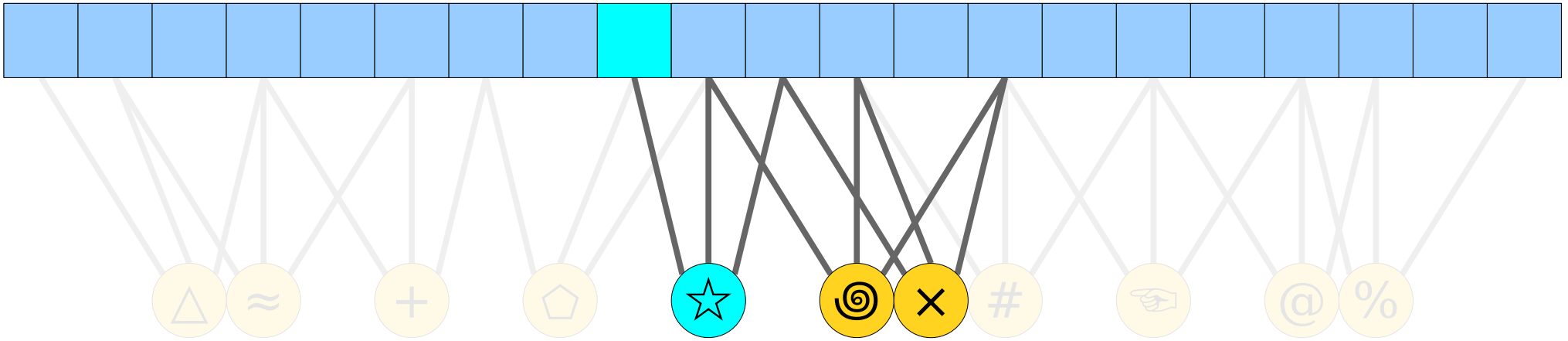
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
- That will likely expose items in slots slightly further from the edges.
- ***Intuition:*** Items will peel from the outsides in.



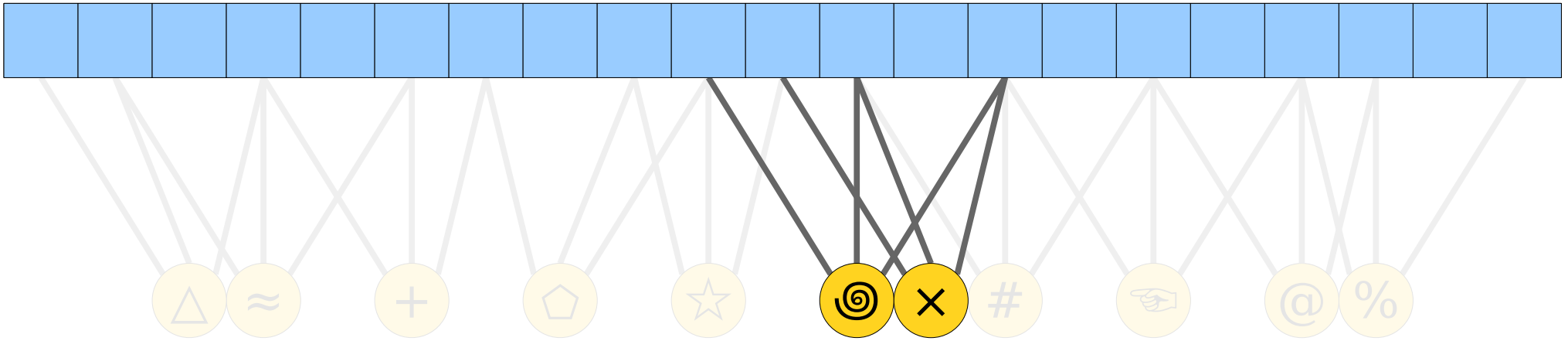
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
- That will likely expose items in slots slightly further from the edges.
- ***Intuition:*** Items will peel from the outsides in.



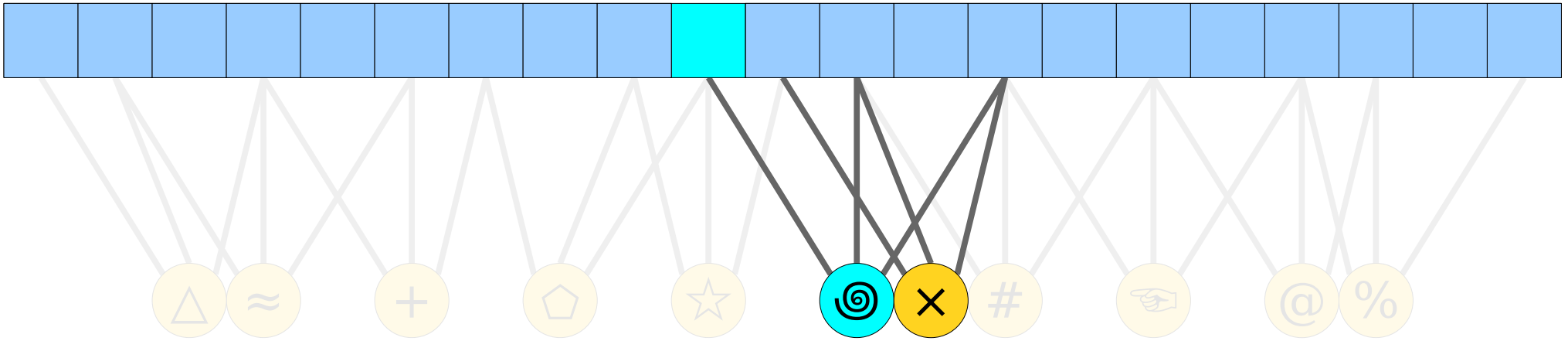
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
- That will likely expose items in slots slightly further from the edges.
- ***Intuition:*** Items will peel from the outsides in.



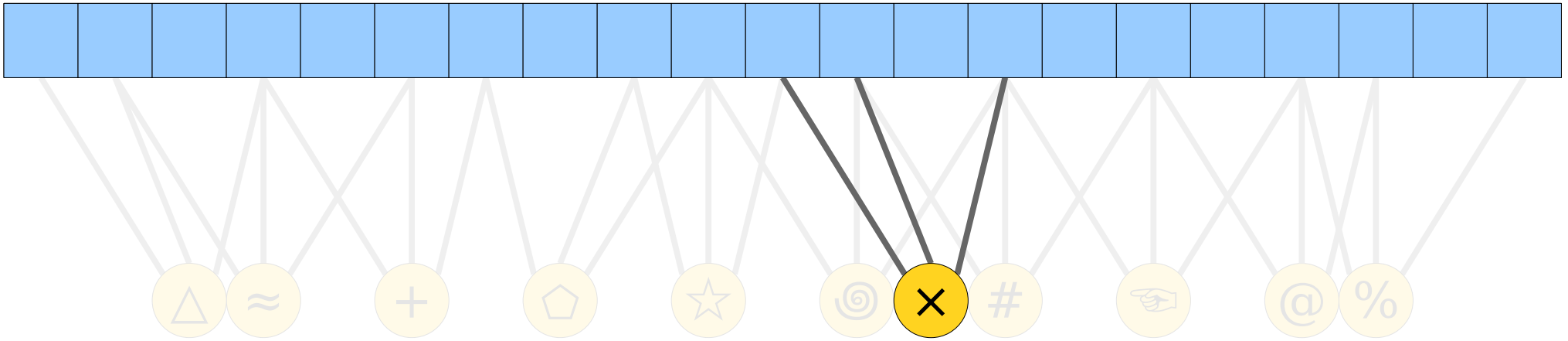
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
- That will likely expose items in slots slightly further from the edges.
- ***Intuition:*** Items will peel from the outsides in.



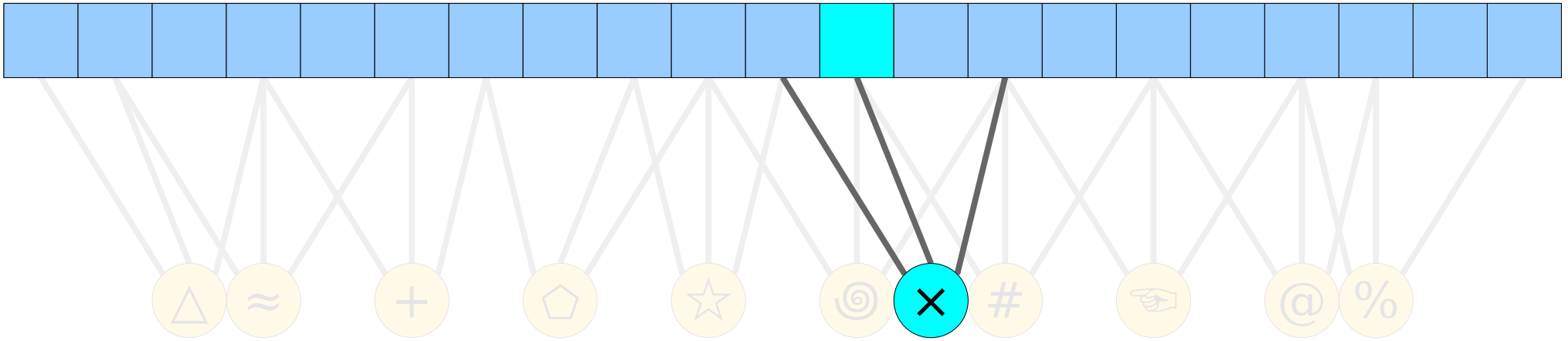
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
- That will likely expose items in slots slightly further from the edges.
- ***Intuition:*** Items will peel from the outsides in.



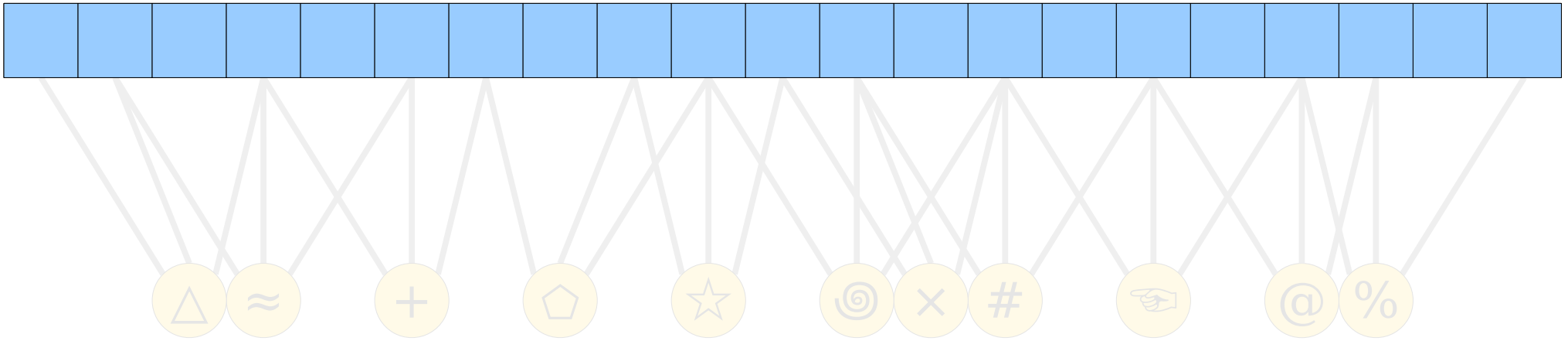
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
- That will likely expose items in slots slightly further from the edges.
- ***Intuition:*** Items will peel from the outsides in.



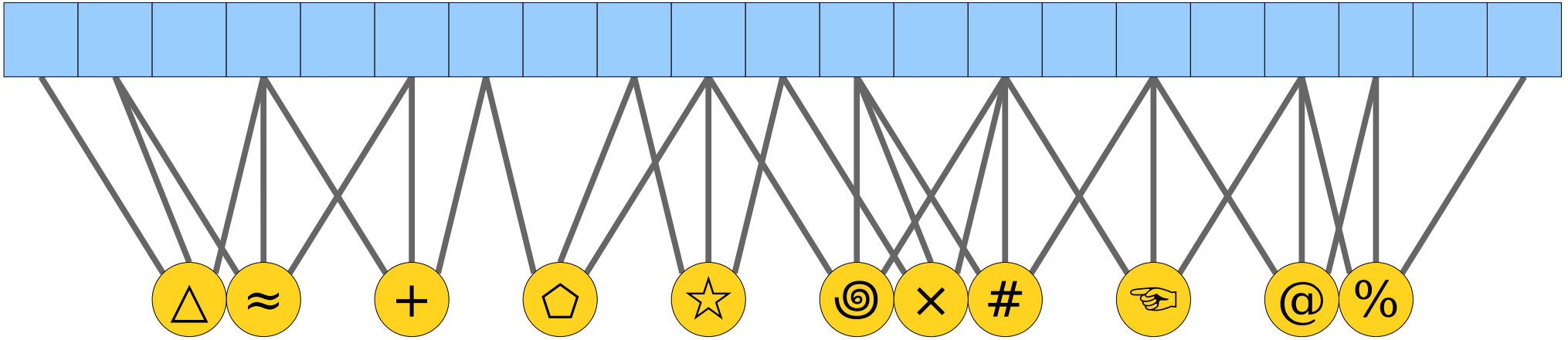
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
- That will likely expose items in slots slightly further from the edges.
- ***Intuition:*** Items will peel from the outsides in.



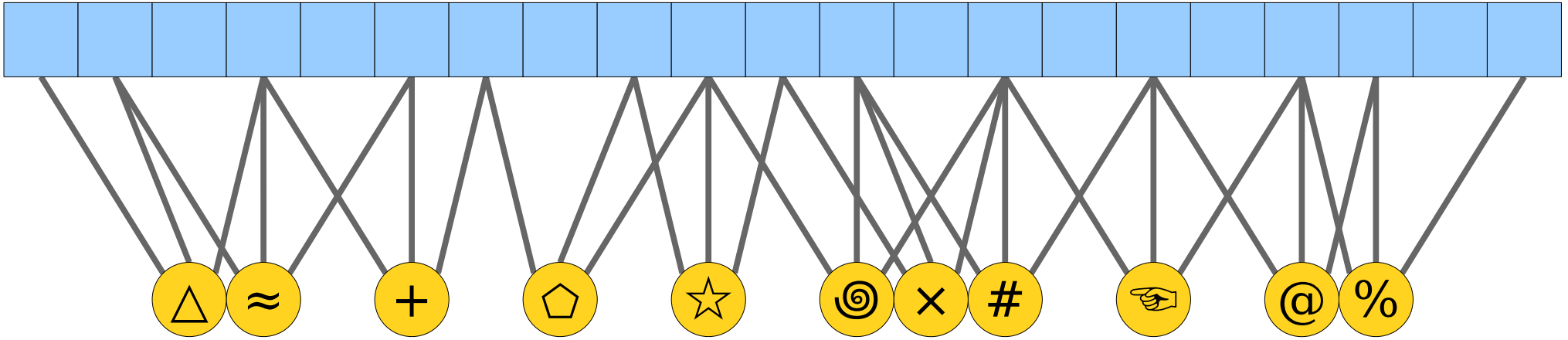
Spatial Coupling

- This strategy of assigning items to slots is called ***spatial coupling*** and is based on ideas first developed in coding theory.
- It allows for significantly more items to be placed into a table while still being peelable.
- An XOR filter modified to use this approach is called a ***binary fuse filter***, based on the idea that the peeling burns like a fuse from the outsides in.



Spatial Coupling

- Binary fuse filters improve upon the choice of α compared to regular XOR filters.
  - For $d = 3$, we get $\alpha \approx 1.125n$.
  - For $d = 4$, we get $\alpha \approx 1.075n$.
- Notice that increasing d now actually *decreases* the space usage.



The Final Scorecard

	Bits Per Element	Hashes/Query
Bloom Filter (1970)	$1.44 \lg \varepsilon^{-1}$	$\lg \varepsilon^{-1}$
Cuckoo Filter, $b = 4$ (2014)	$1.02 \lg \varepsilon^{-1} + 3.06$ *(for sufficiently small ε)*	3
XOR Filter (2020)	$1.23 \lg \varepsilon^{-1}$	4
Binary Fuse Filter, $d = 3$ (2022)	$1.13 \lg \varepsilon^{-1}$	5
Binary Fuse Filter $d = 4$ (2022)	$1.08 \lg \varepsilon^{-1}$	6

More to Explore

Related Data Structures

- ***Ribbon filters*** are an alternative to XOR and cuckoo filters based on solving random systems of linear equations. They also give stellar space usage and are fast in practice.
- There are several theoretically ***optimal AMQ structures*** developed in the past twenty years. They use $(1 + o(1)) \lg \varepsilon^{-1}$ bits per element. The first (I believe?) is due to Pagh, Pagh, and Rao.
- The same techniques we're using to build AMQs can be used to construct ***perfect hash functions***. Instead of storing fingerprints, store an index between 0 and $n - 1$ for the items in question.
- The ***Bloomier filter*** generalizes XOR filters as ways of storing approximate maps rather than approximate sets. They were famously used in a paper about compressing machine learning models (***Weightless***).

Next Time

- ***You Choose!***
  - ***Option 1:*** String data structures (suffix trees, suffix arrays, etc.)
  - ***Option 2:*** Better-than-balanced BSTs (splay trees, etc.)
  - ***Option 3:*** Integer data structures (fusion trees, etc.)