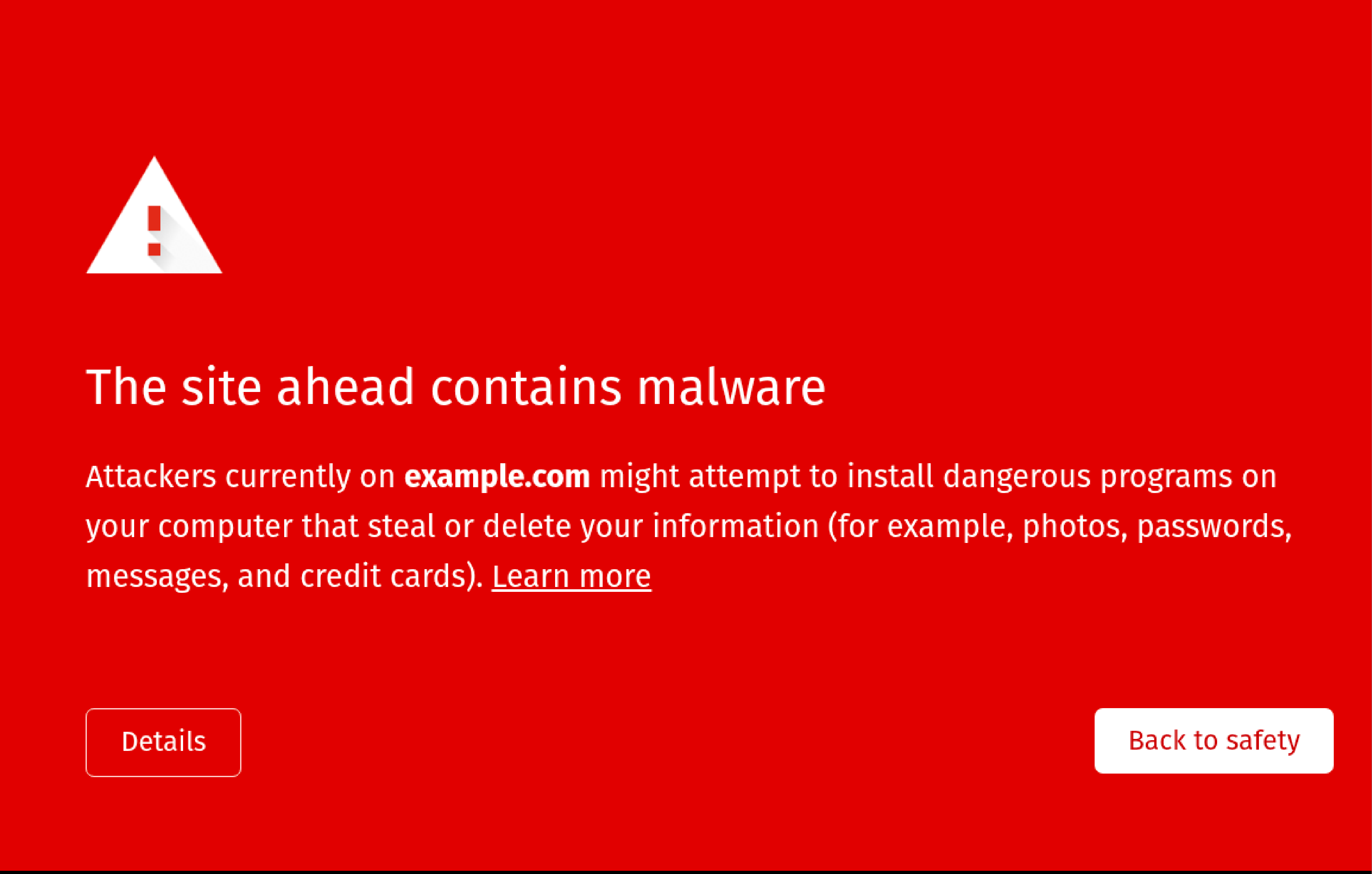
Approximate Membership Queries

Part One

Outline for Today

- ***Approximate Membership Queries***
  - Storing sets… sorta.
- ***Bloom Filters***
  - The original approximate membership query structure – and still the most popular!
- ***Data Structure Lower Bounds***
  - Is the Bloom filter "good?" How much can it be improved?

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 2%. ***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 98% 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?"

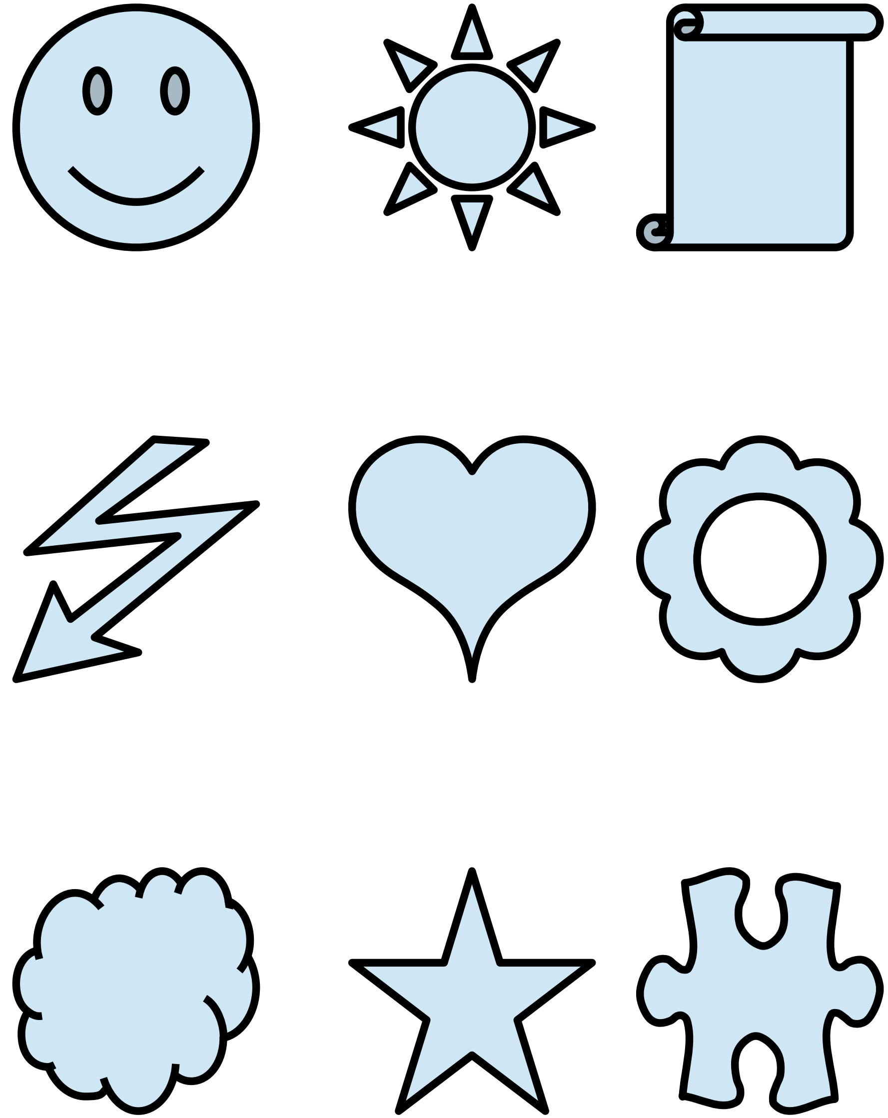
- You now have a ton of tools available for solving this problem:

Red/black trees · Skiplists
B-trees · Cuckoo hashing

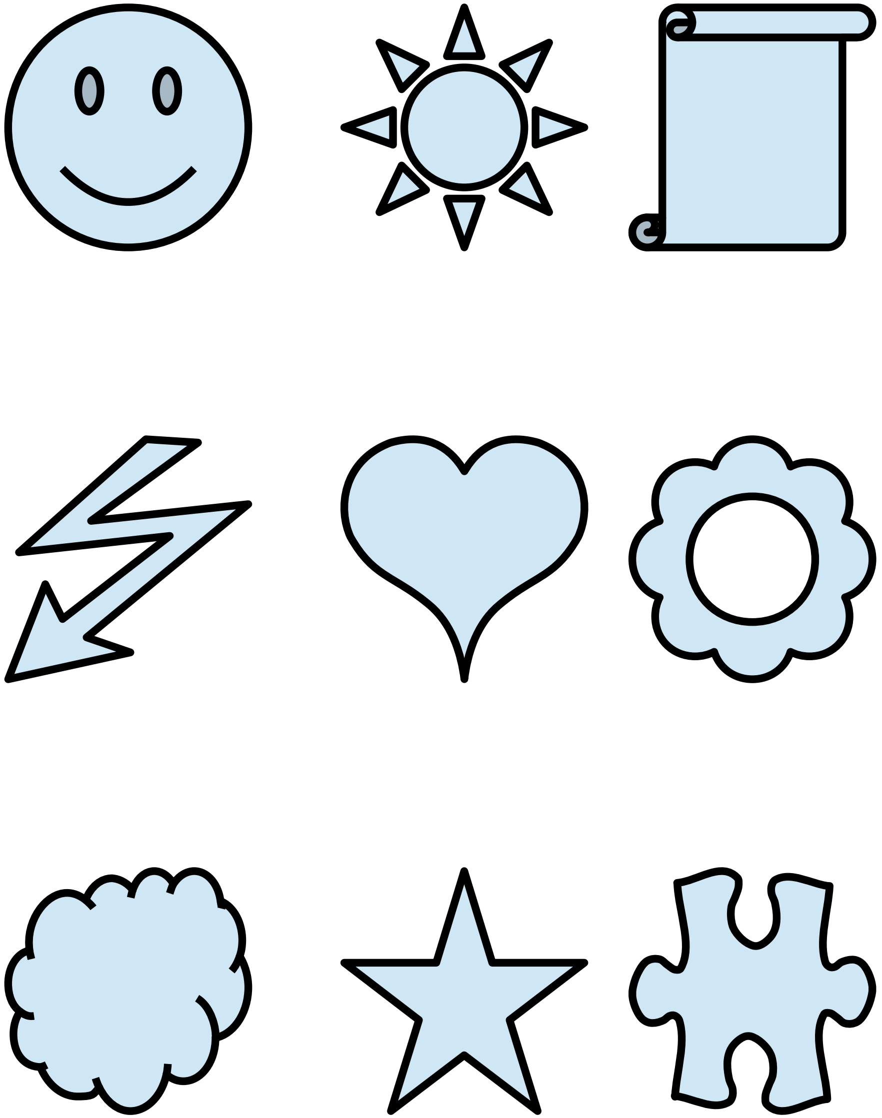
Exact Membership Queries

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

A Quick Detour



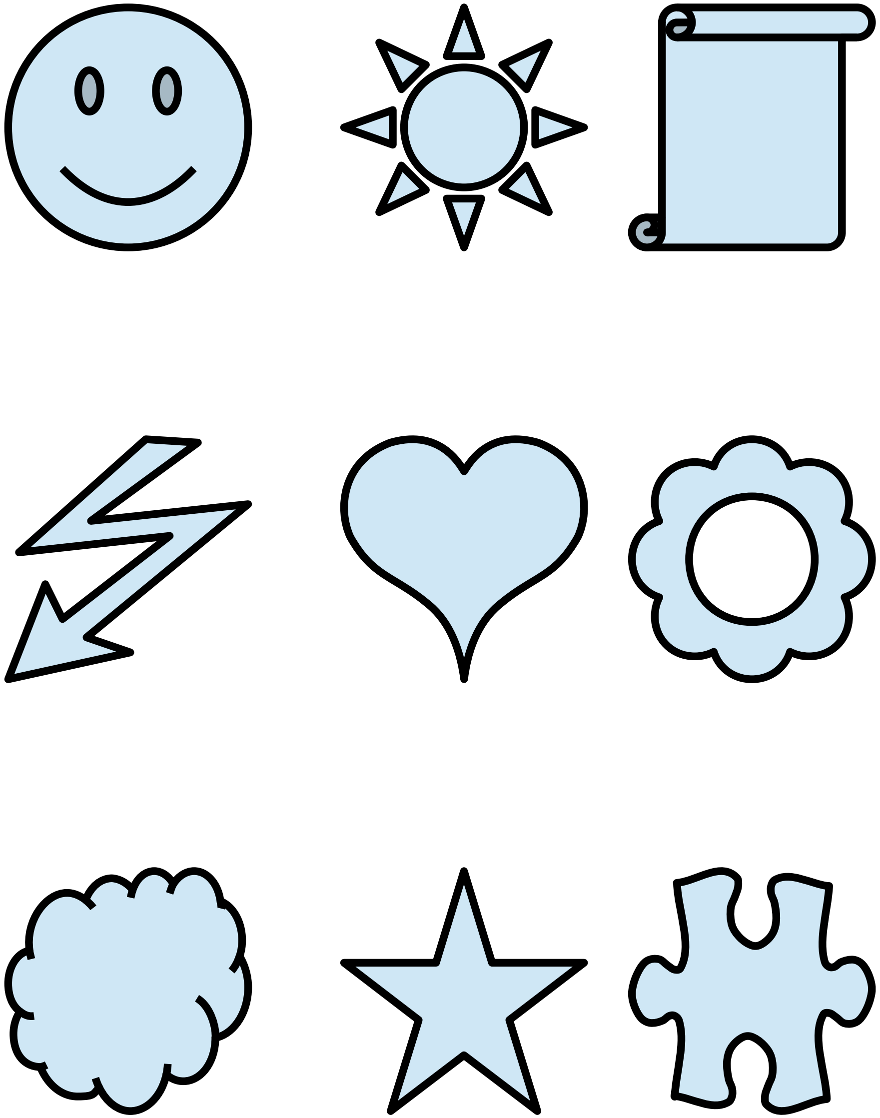
Goal: Design a simple data structure that can hold a single one of the objects shown to the right.



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?

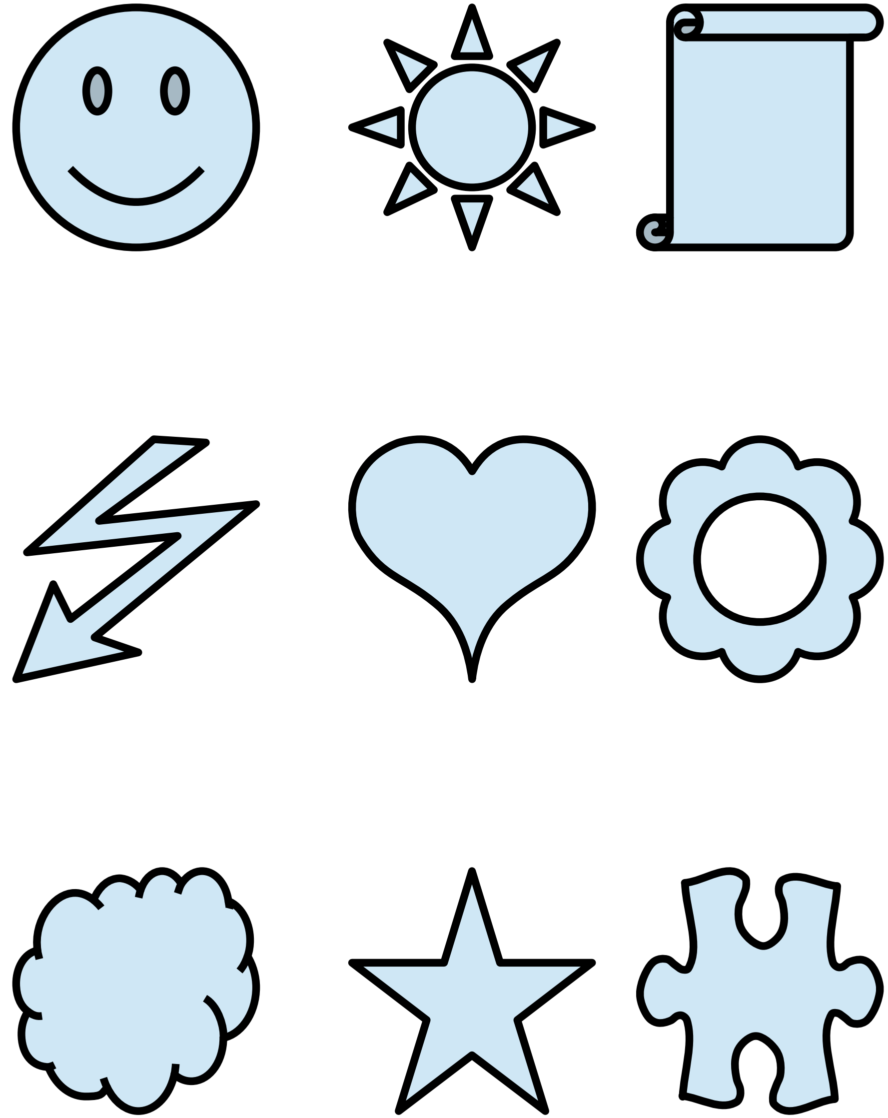
Formulate a hypothesis, but ***don’t post anything in chat just yet.***



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

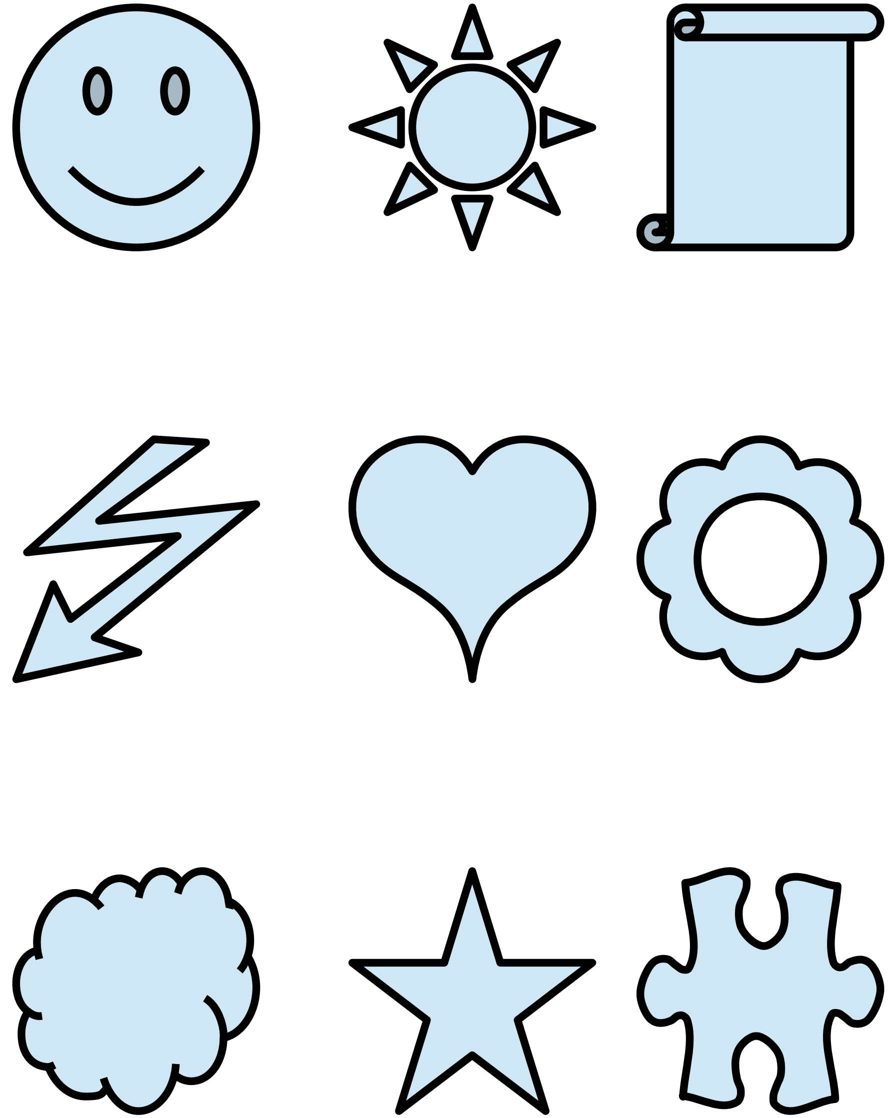
What is the minimum number of *bits* (not *words*) required for this data structure in the worst case?

Now, ***private chat me your best guess***. Not sure? Just answer “??.”



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

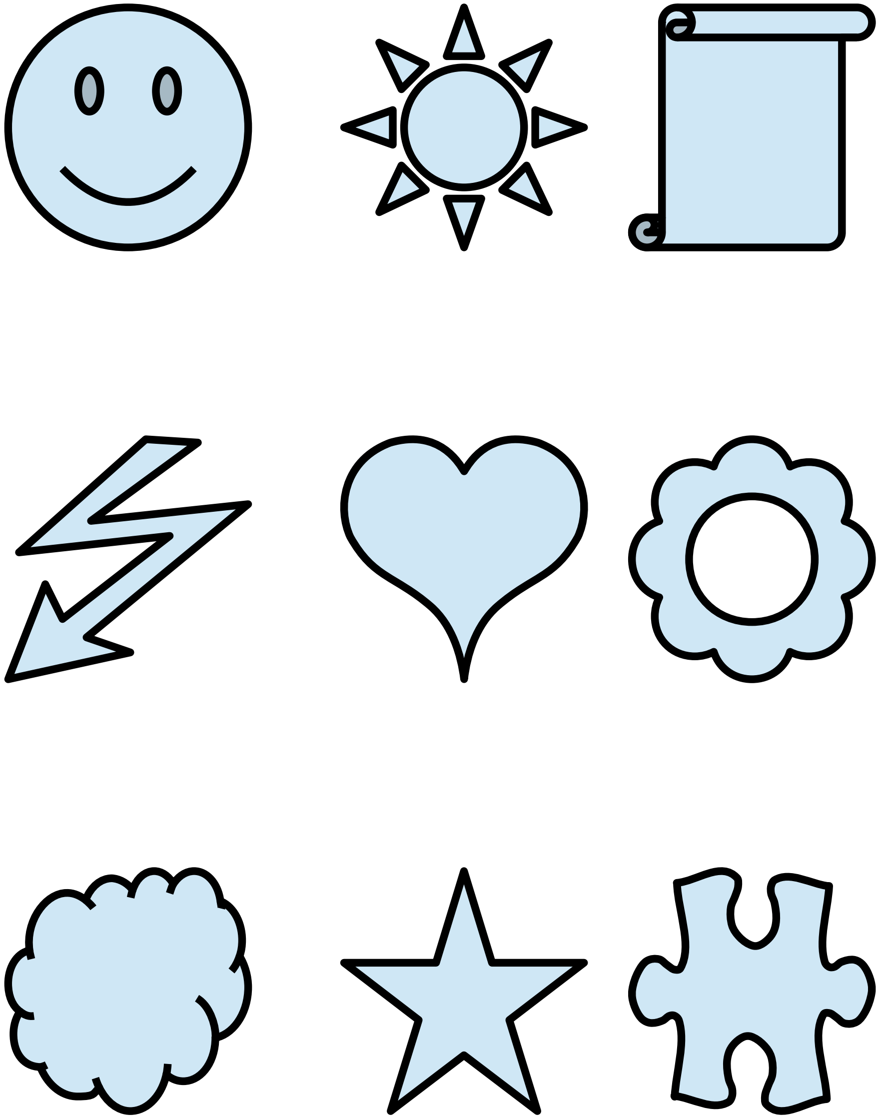
What is the minimum number of *bits* (not *words*) required for this data structure in the worst case?



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?

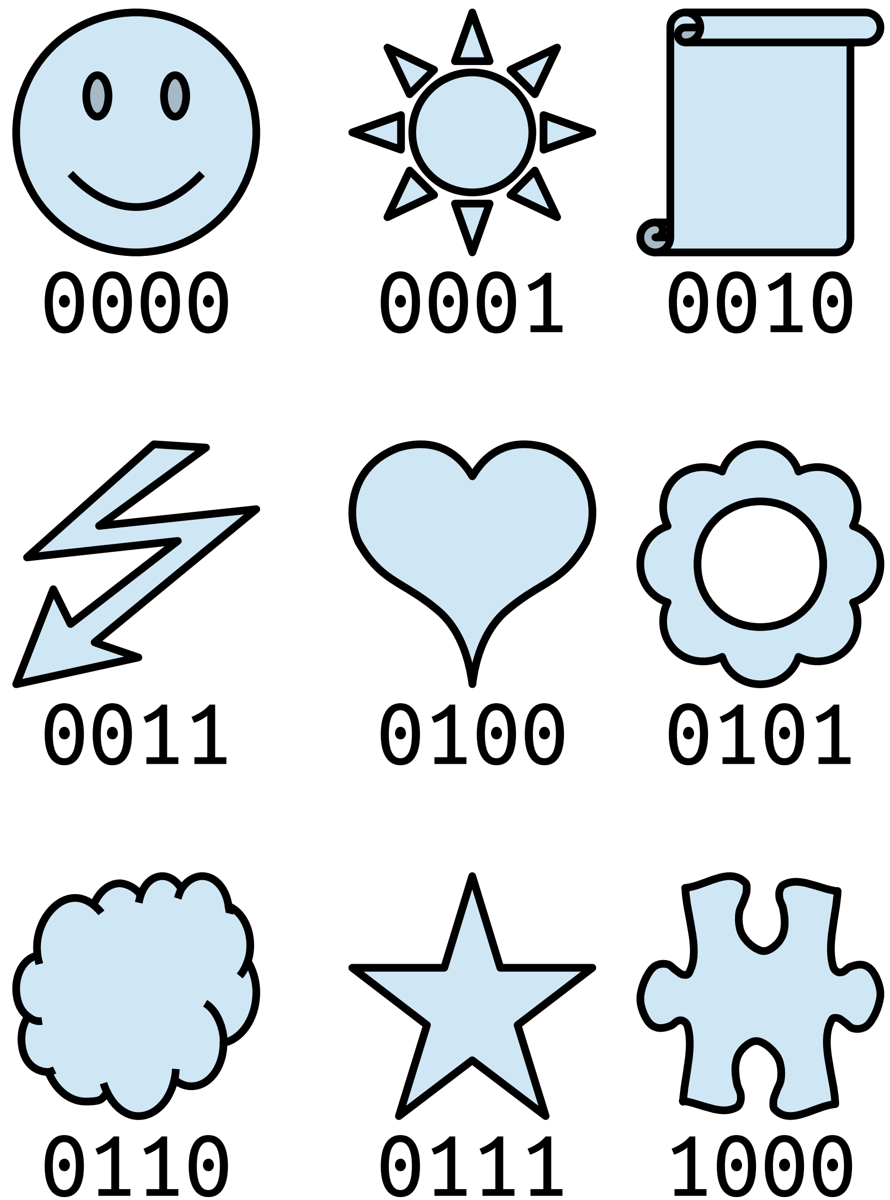
We can get away with four bits by numbering each item and just storing the number.



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.

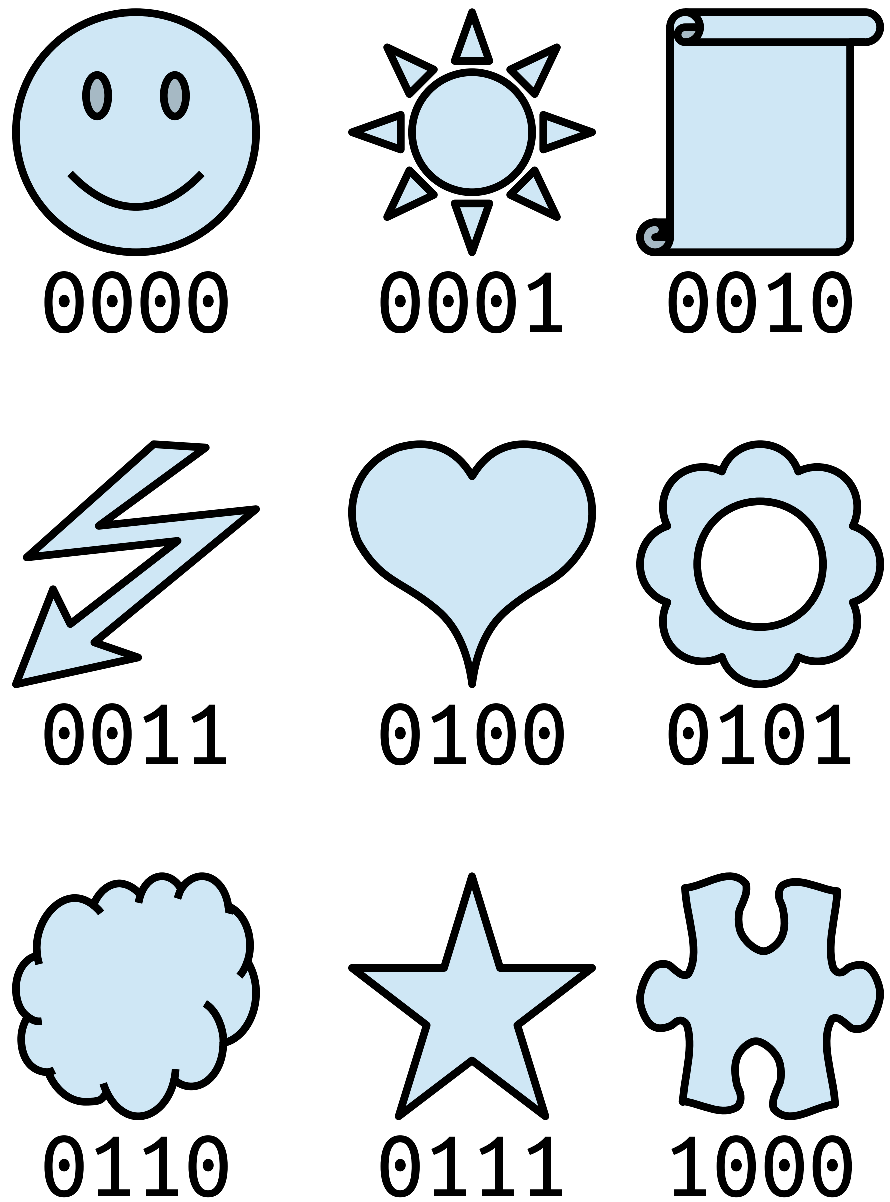


Goal: Design a simple data structure that can hold a single one of the objects shown to the right.

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.

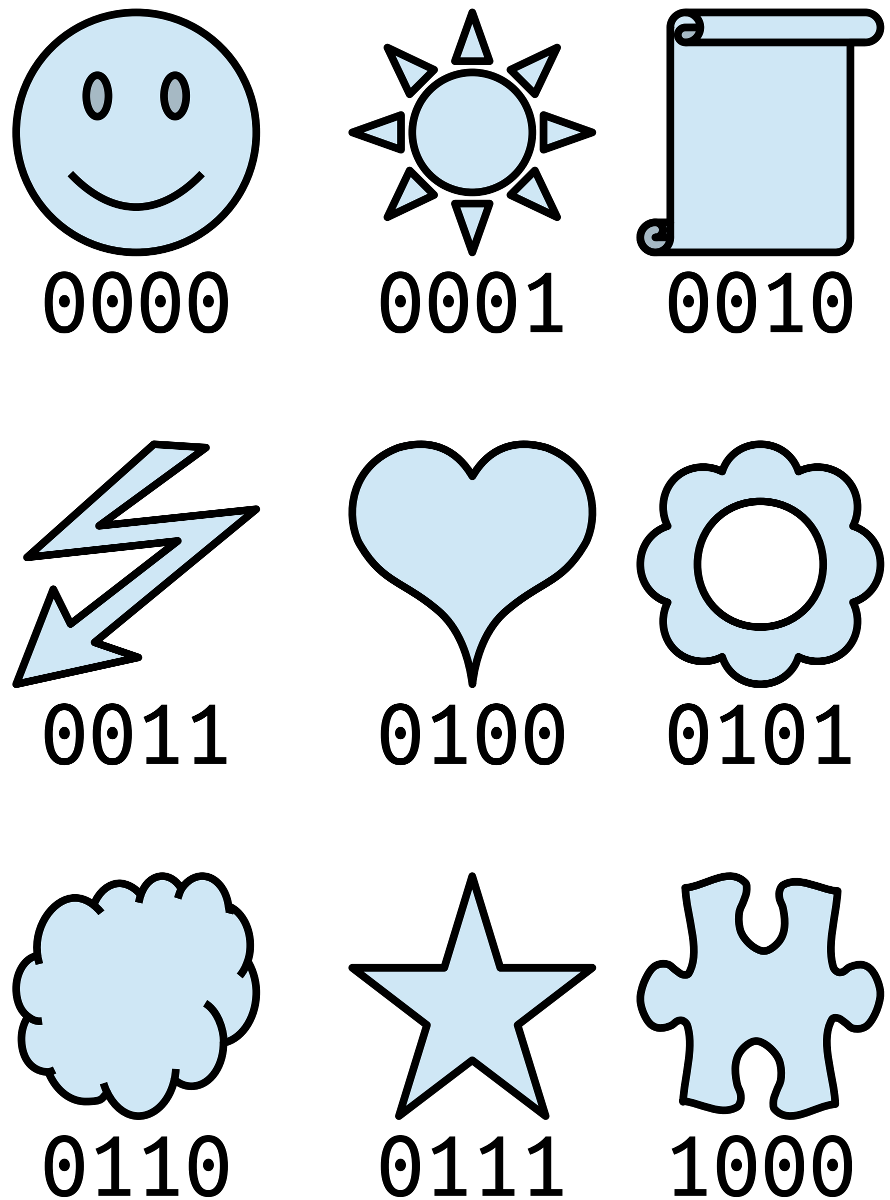
Question: Can we do better?



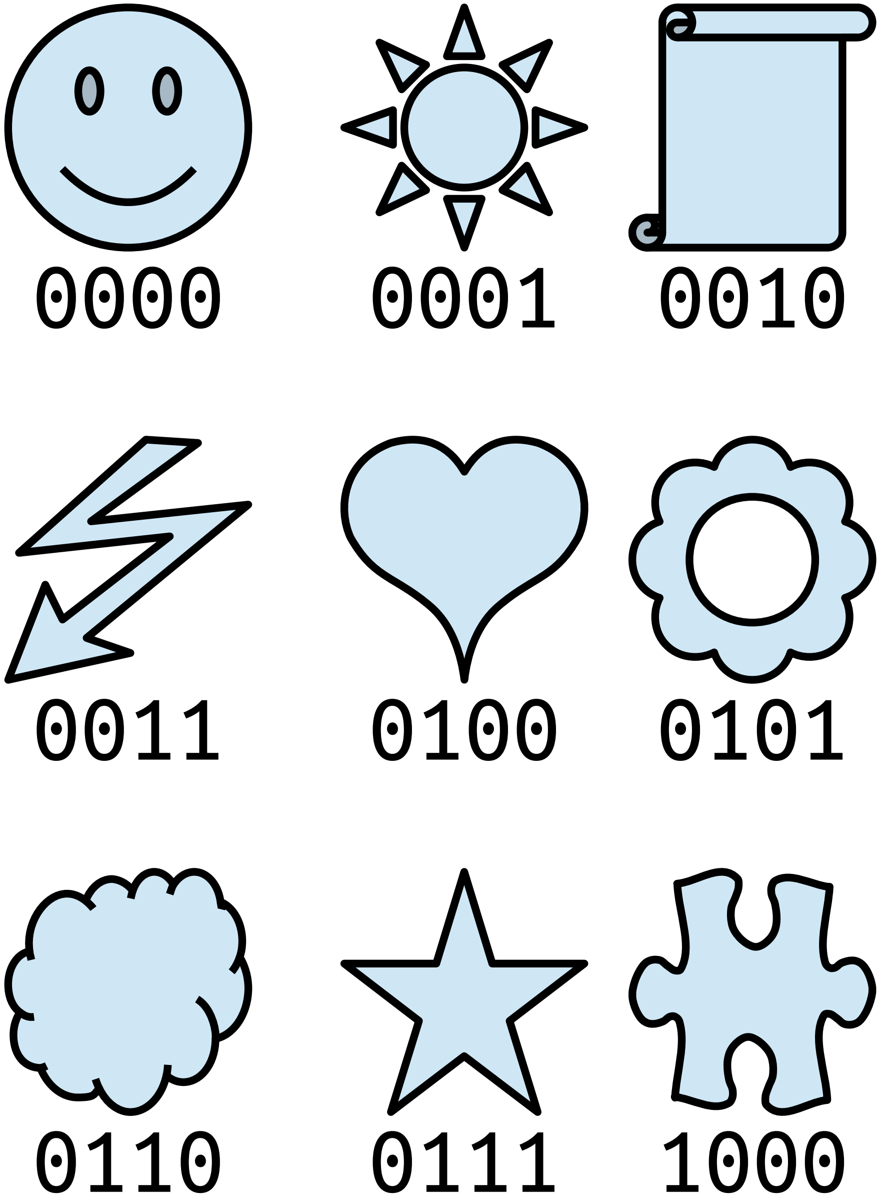
Goal: Design a simple data structure that can hold a single one of the objects shown to the right.

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.

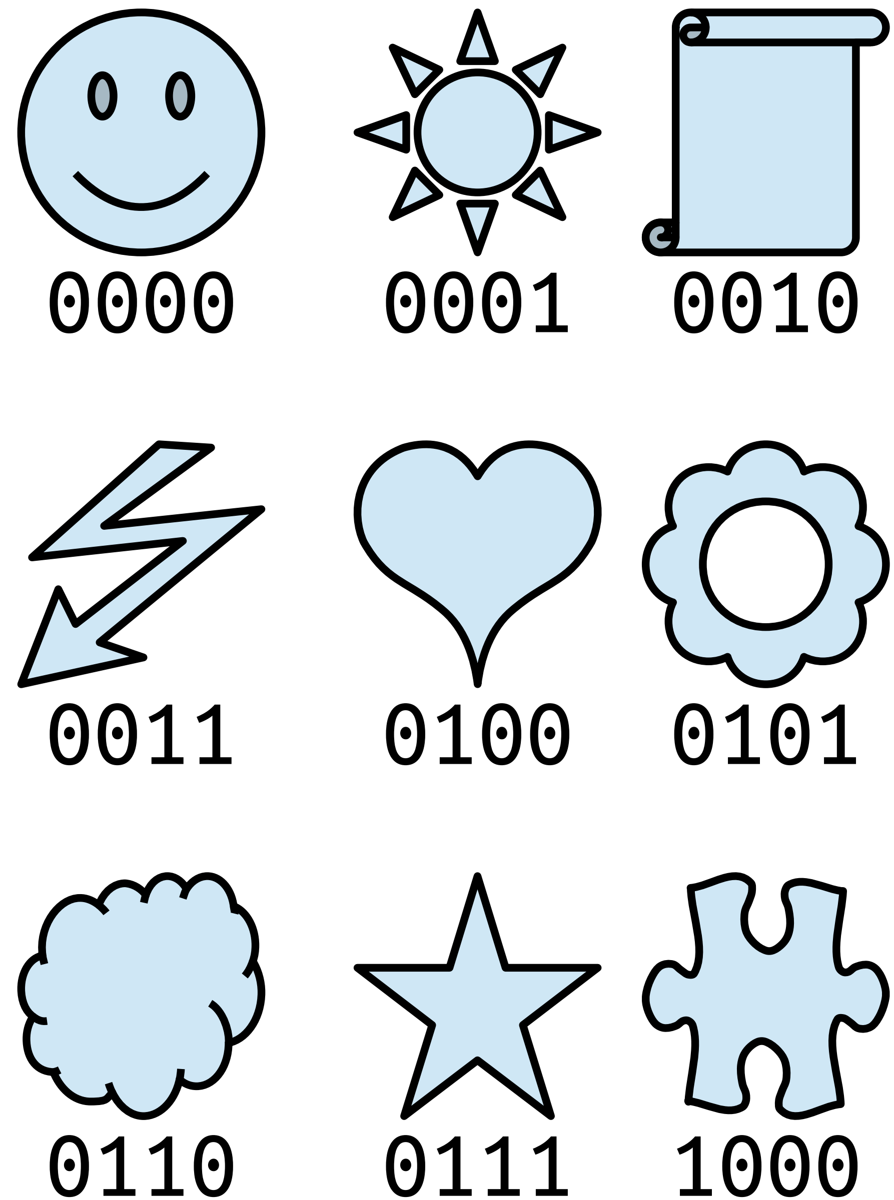


Theorem: A data structure that stores one object out of a set of k possibilities must use at least lg k bits in the worst case.



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.

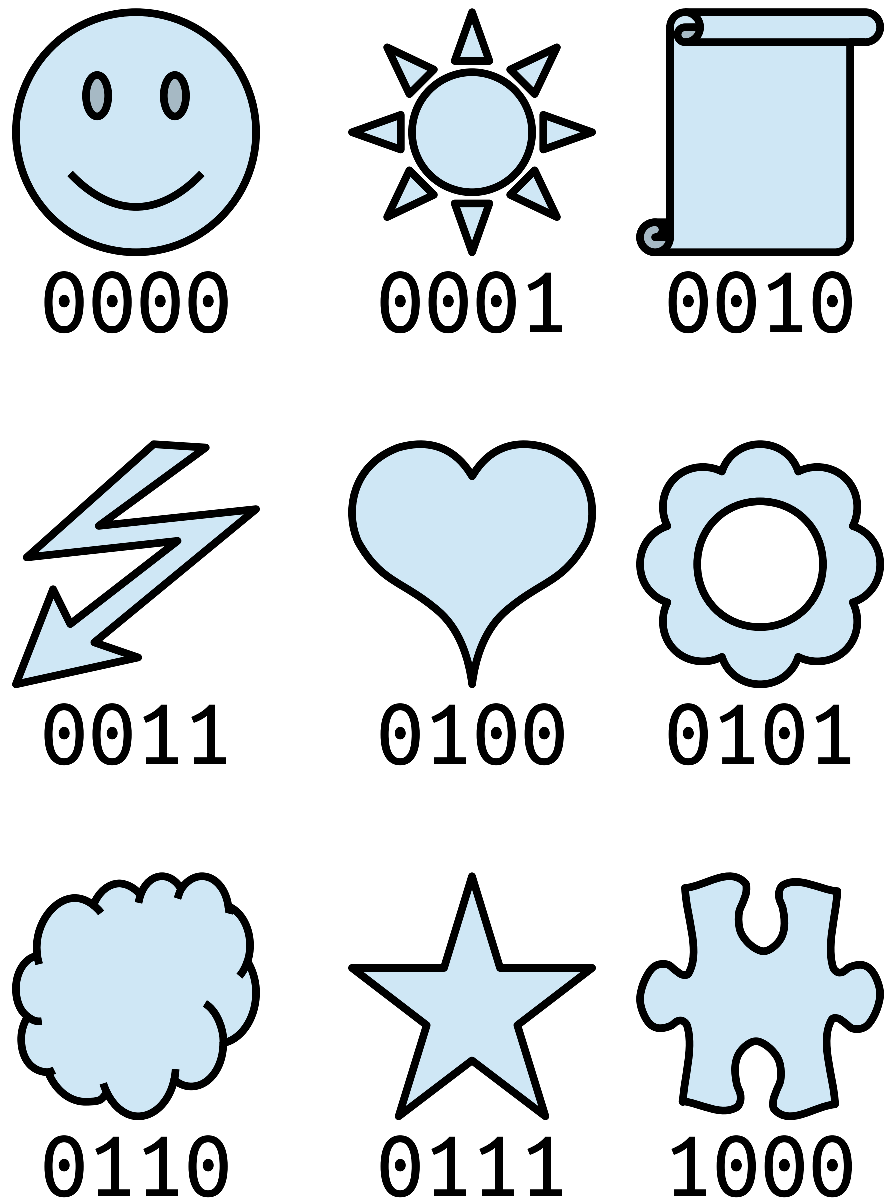
(lg is the ***binary logarithm*** $\log_2 x$. It comes up a lot in Theoryland.)



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.

(lg is the ***binary logarithm*** $\log_2 x$. It comes up a lot in Theoryland.)

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:

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

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

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

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:

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

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

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

Number of n-element subsets of universe U:

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

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

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

Number of n-element subsets of universe U:

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

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

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 $x \in S$?"

- 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.

(ε, δ)-Approximators

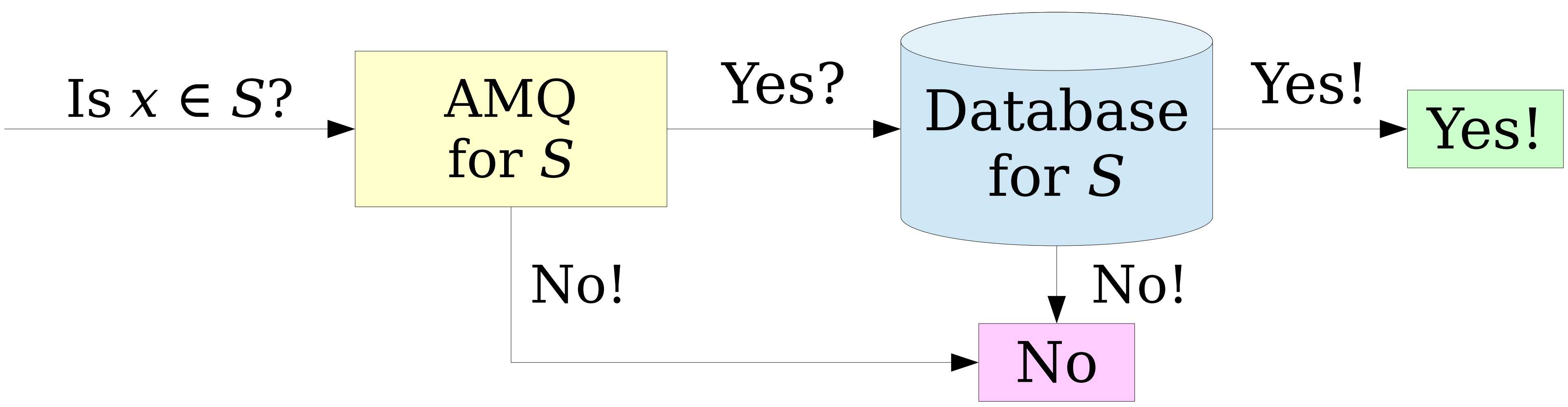
- Many of the approximators we've built in the past are (ε, δ)-approximators that make this guarantee:

$$\mathbf{Pr[\ |\hat{A} - A| > \varepsilon \cdot \mathit{size}(\mathit{input})\ ] < \delta}$$

- This is what we did with the count-min sketch, the count sketch, and cardinality estimation.
- In the case of set membership, though, we're estimating a single boolean value. What would it mean to measure the "distance" from our estimate to the true value?
  - We can't say something like "x is 95.7% in S" – or at least, we'd like to avoid doing so.
- Therefore, we won't be using that model here. Instead, we'll pick a different approach.

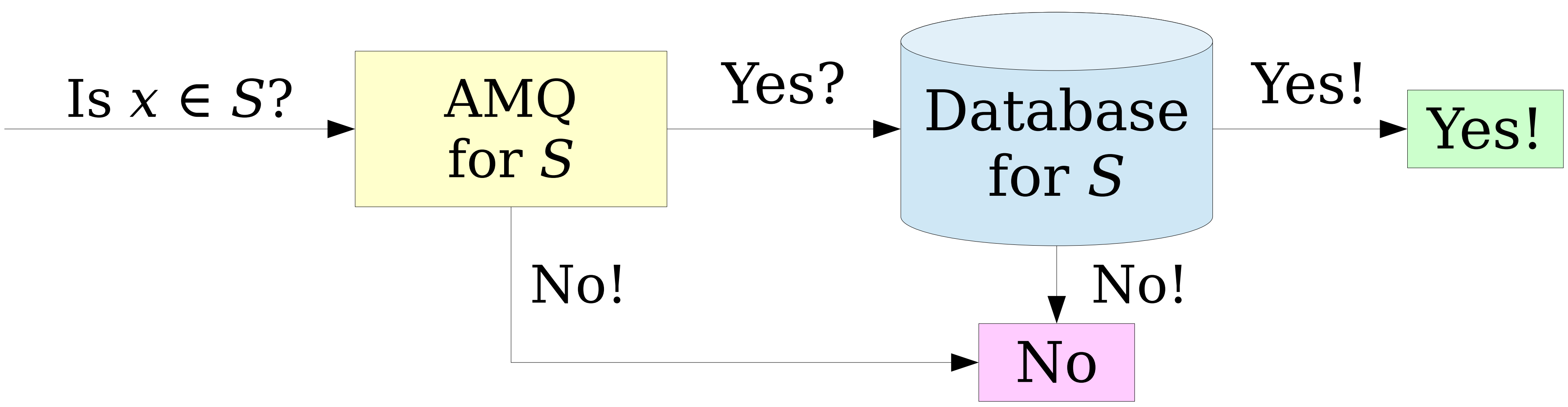
Our Model

- ***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

- Assume we have a user-provided ***accuracy*** parameter $\varepsilon \in (0, 1)$ and a set $S \subseteq U$ of size n.
- ***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

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 }

0000000000000000000000000000000000000000

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 }

`0010000000000000000000000000000000000000`

$h(103)$

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

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

`0010000000000000000000000000010000000000`

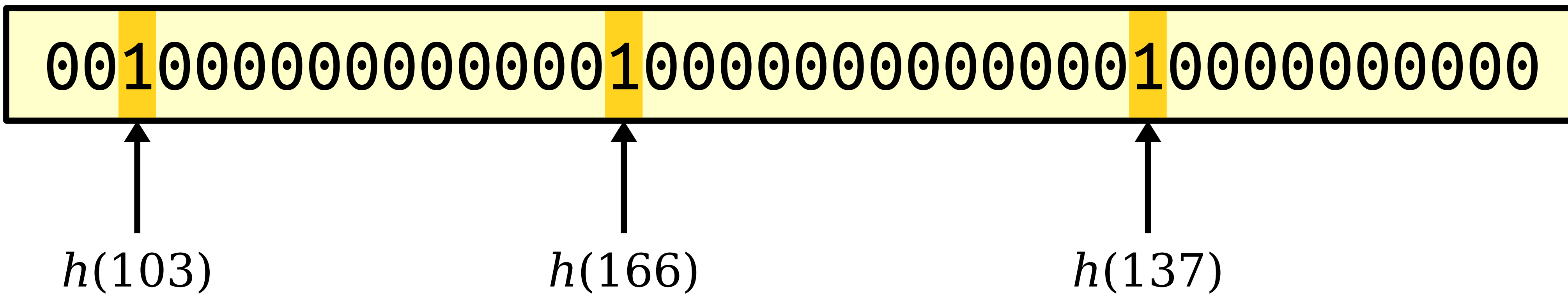
$h(103)$ $h(137)$

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

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

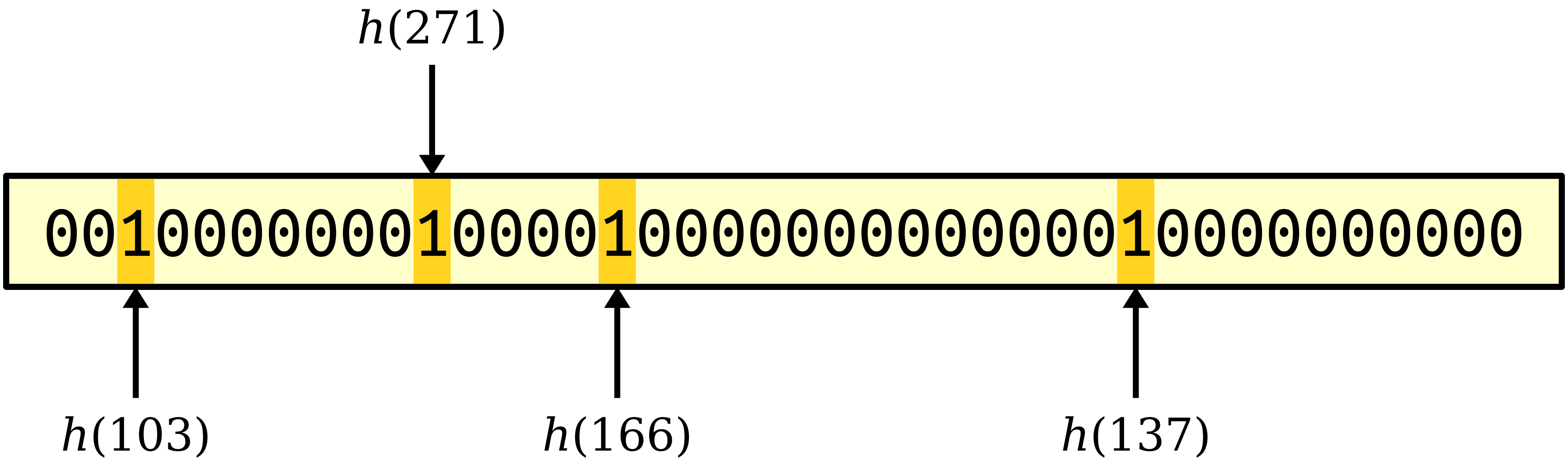


How can we approximate a set in a small number of bits and with a low error rate?

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

As an example, let's have

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

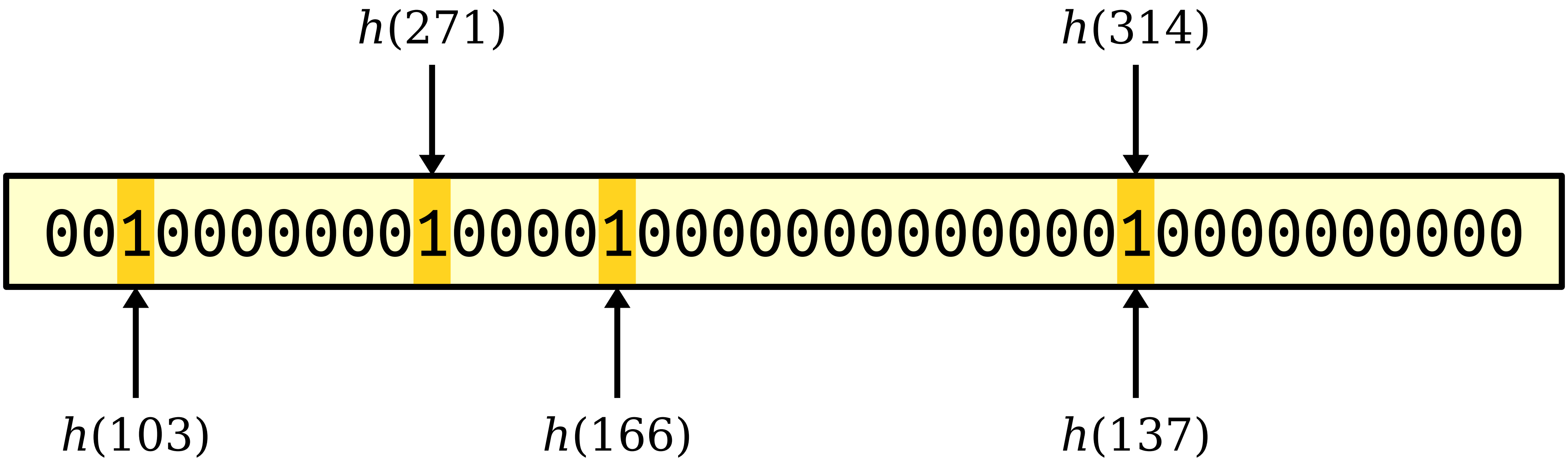


How can we approximate a set in a small number of bits and with a low error rate?

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

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

`0010000000100001000000000000000001000000000`

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`

$h(103)$

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

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`

$h(103)$

$h(161)$

True 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.

As an example, let's have

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

`0010000000100001000000000000010000000000`

$h(103)$

True positive

$h(161)$

True negative

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*

$h(261)$

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 }

`0010000000100001000000000000000100000000000`

$h(103)$ *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?

`0010000000100001000000000000010000000000`

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?

0010000000100001

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?

`00100000001000010`

$h(x)$

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.

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:

$$^{1}/_{m}.$$

`00100000001000010`

$h(x)$

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.

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.

`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.

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

`00100000001000010`

$h(x)$

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: $n \cdot \varepsilon^{-1}$ bits.

`00100000001000010`

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: $n \cdot \varepsilon^{-1}$ bits.

Suppose we want to store a list of 32-bit integers with a false positive rate of 2%. How many bits do we need?

`0010000000100001 0`

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.

`00100000001000010`

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

Suppose we want to store a list of 32-bit integers with a false positive rate of 2%. How many bits do we need?

Answer: **50*n***.

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.

`001000000010000100`

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

Suppose we want to store a list of 32-bit integers with a false positive rate of 2%. How many bits do we need?

Answer: **50*n***.

Just storing a sorted list would be more space-efficient than this.

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.

`0010000000100001 0`

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

Suppose we want to store a list of 32-bit integers with a false positive rate of 2%. How many bits do we need?

Answer: **50*n***.

Just storing a sorted list would be more space-efficient than this.

Question: Can we do better?

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?

`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?

```
00010000000000000000001100
```

Formulate a hypothesis, but ***don't post anything in chat just yet***.

```
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?

```
000100000000000000001100
0010000000100001000000000000010000000000
0000010000000000010000001000001000000100
```

Now, ***private chat me your best guess***. Not sure? Answer "??."

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`

`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?

```
00010000000000000000
0001

00100000001000010000
0000

0000010000000000010000001000001000000100
```

This entry is 0, so it's not possible for this element to be in our set S.

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
```

```
0000010000000000000010000010000010000100
```

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?

```
0001000000000000000000110000000100000000001
```

```
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?

```
0001000000000000000011000000010000000001
```

```
0010000000100001000000000000010000000000
```

```
00000100000000000100000010000
```

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?

`0010000000100001000000000000010000000000`

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?

```
0010010000010000100010
```

```
1001100001000000001000
```

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?

```
0010001100100100
```

```
0101000010001010
```

```
0000011100000010
```

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.

`0100001001100010010010000010001000100100`

Can we get the same accuracy while using fewer bits?

```
0100001001100010010010000001000100000100100
```

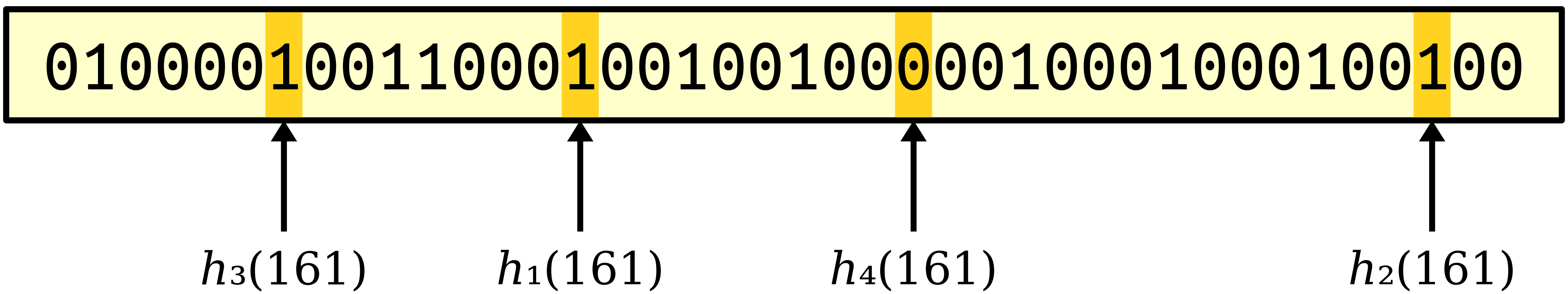
Number of bits: $\boldsymbol{m}$

(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.

`0100001001100010010010000010001000100100`

$h_3(161)$ $\quad$ $h_1(161)$ $\quad$ $h_4(161)$ $\quad$ $h_2(161)$

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.

`0100001001100010010010000010001000100100`

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.

`1000100010001100101010000100010l`

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.

`110011001111000111000101`

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.

111111

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

```
0100001001100010010010000001000100001000100
```

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

```
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.

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

```
0101001001101010011010001010001000110100
```

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

```
0101001111101010011010001011001000110111
```

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

1111111111111111111111111111111111111111

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

```
0100001001100010010010000010001000100100
```

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

`0000001001000001000000000000000001000000100`

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 $n = \alpha m$ 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?

`01000010011000100100100000010001000100100`

Can we get the same accuracy while using fewer bits?

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

```
01000010011000100100100000010001000100100
```

How do we quantify our error rate?

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.

```
01000010011000100100100000010001000100100
```

How do we quantify our error rate?

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.

`0100001001100010010011000000100010001001000100`

How do we quantify our error rate?

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?

`01000010011000100100100000010001000100100`

How do we quantify our error rate?

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

`0011010100101000`

Focus on a bit at index i.

How do we quantify our error rate?

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

`0011010100101000`

Focus on a bit at index i.

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

How do we quantify our error rate?

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

`00110101000101000`

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.)

How do we quantify our error rate?

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

`0011010100010100 0`

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: $1 - {}^{1}/_{m}$.

How do we quantify our error rate?

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

`0011010100010100 0`

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?

How do we quantify our error rate?

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

`0011010100101000`

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?

`0011010100101000`

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

How do we quantify our error rate?

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

`0011010100101000`

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

Probability that bit i is unset after inserting n elements:

$$\left(1 - \frac{1}{m}\right)^{kn}$$

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:

$$\left(1 - \frac{1}{m}\right)^{kn}$$

$$= \left(\left(1 - \frac{1}{m}\right)^{m}\right)^{\frac{kn}{m}}$$

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:

$$\left(1 - \frac{1}{m}\right)^{kn}$$

$$= \quad \left(\left(1 - \frac{1}{m}\right)^{m}\right)^{\frac{kn}{m}}$$

$$\approx \quad e^{-\frac{kn}{m}}$$

How do we quantify our error rate?

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

`00110101000101000`

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

Probability that bit i is unset after inserting n elements:

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

How do we quantify our error rate?

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

`0011010100010100`

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

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

How do we quantify our error rate?

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

00110101000101000

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

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

False positive probability is approximately

How do we quantify our error rate?

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

`00110101000101000`

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

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

False positive probability is approximately

How do we quantify our error rate?

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

`0011010100001010 00`

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

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

False positive probability is approximately

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

How do we quantify our error rate?

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

`00110101000101000`

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:

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

False positive probability is approximately

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

How do we quantify our error rate?

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

`0011010100001011000`

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

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

False positive probability is approximately

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

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.

Question: What choice of k minimizes this expression?

How do we quantify our error rate?

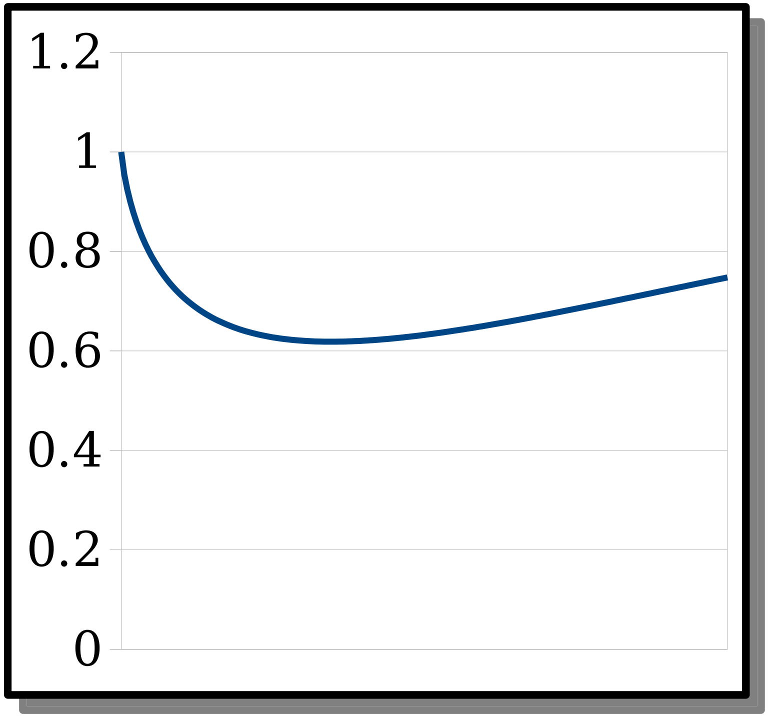
Goal: Pick k to minimize

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

How many hash functions should we use?

Goal: Pick k to minimize

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

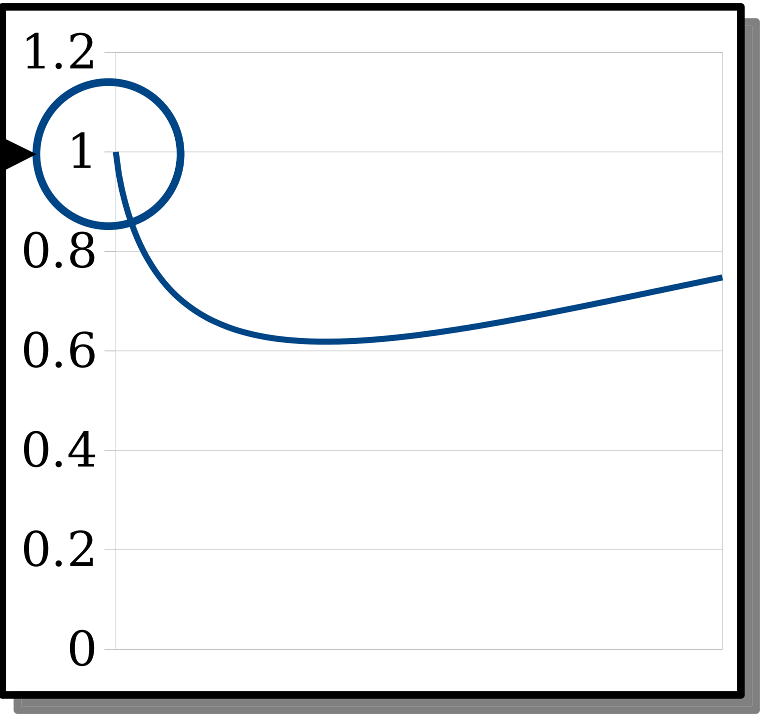


How many hash functions should we use?

Goal: Pick k to minimize

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

If k is too low, fewer bits are 1, but there's fewer hashes available to hit 0 bits.

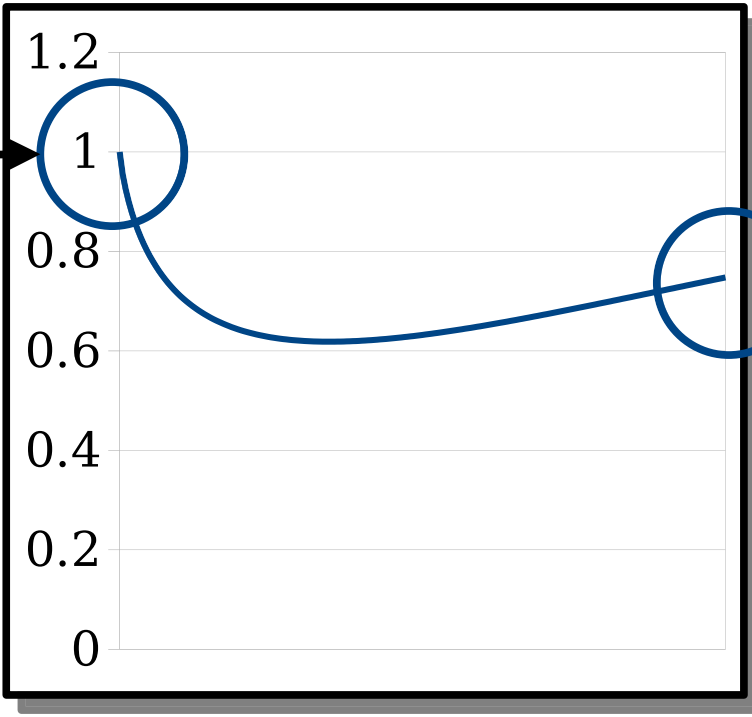


How many hash functions should we use?

Goal: Pick k to minimize

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

If k is too high, too many of the bits become 1, and we start hitting them with high frequency.



If k is too low, fewer bits are 1, but there's fewer hashes available to hit 0 bits.

How many hash functions should we use?

Goal: Pick k to minimize

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

Claim: This expression is minimized when

$$\boldsymbol{k = \alpha^{-1} \ln 2}.$$

You can show this using some symmetry arguments or calculus.

How many hash functions should we use?

Goal: Pick k to minimize

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

Claim: This expression is minimized when

$$\boldsymbol{k = \alpha^{-1} \ln 2}.$$

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.

How many hash functions should we use?

Goal: Pick k to minimize

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

Claim: This expression is minimized when

$$\boldsymbol{k = \alpha^{-1} \ln 2}.$$

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

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

and we know to pick

$$k = \alpha^{-1} \ln 2.$$

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

$$\mathbf{2^{-\alpha^{-1} \ln 2}}.$$

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

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

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

$$2^{-\alpha^{-1} \ln 2}.$$

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

To do so, pick

$$\boldsymbol{\alpha = \ln 2 \,/\, \lg \varepsilon^{-1}}$$

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

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

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

$$n = m \cdot \alpha$$

$$k = \alpha^{-1} \ln 2$$

Optimal α:

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

How did we do overall?

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

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

$$k = \alpha^{-1} \ln 2$$

Optimal α:

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

How did we do overall?

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

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

$$k = \alpha^{-1} \ln 2$$

Optimal α:

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

How did we do overall?

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

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

$$k = \alpha^{-1} \ln 2$$

Optimal α:

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

How did we do overall?

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

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

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

Optimal α:

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

How did we do overall?

The Bloom Filter

- Create an array of **$1.44n \lg \varepsilon^{-1}$** bits, all initially zero.
- Select **$\lg \varepsilon^{-1}$** hash functions, each of which maps items to bit positions.
- 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 $\lg \varepsilon^{-1}$ hash functions to get a set of bits to test, then return true if they're all set to 1 and false otherwise.

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

The Bloom Filter

- What does 1.44 lg ε^{-1} look like in practice?
  - With 4 bits per element, we have $\varepsilon \approx 0.146$.
  - With 8 bits per element, we have $\varepsilon \approx 0.0214$.
  - With 16 bits per element, we have $\varepsilon \approx 0.000458$
- In other words, we can get extremely low error rates using surprisingly few bits per element.
- Accordingly, Bloom filters are used extensively in practice.

	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 our Bloom filter, we can either make
  - improvements to the query time, or
  - improvements to the space usage.
- Let's look at each of these in turn.

	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 our Bloom filter, we can either make
  - improvements to the query time, or
  - improvements to the space usage.
- Let's look at each of these in turn.

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

Claim: In some ways, Bloom filters have faster queries than the worst-case cost suggests. In others, Bloom filters have slower queries than the worst-case cost suggests.

Question: In a Bloom filter with n elements and a false positive rate of ε, what fraction of the bits in the array will be equal to 1?

Question: In a Bloom filter with *n* elements and a false positive rate of ε, what fraction of the bits in the array will be equal to 1?

Answer: Approximately half of them.

Question: In a Bloom filter with n elements and a false positive rate of ε, what fraction of the bits in the array will be equal to 1?

Answer: Approximately half of them.

The math, in case you're curious:

Each bit is set to 1 with probability approximately $1 - e^{-k\alpha}$. We pick $k = \lg \varepsilon^{-1}$ and $\alpha = \ln 2 / \lg \varepsilon^{-1}$.

Probability a bit is set to 1: approximately

$$\begin{aligned} & 1 - e^{-(\lg \varepsilon^{-1})(\ln 2 / \lg \varepsilon^{-1})} \\ &= 1 - e^{-\ln 2} \\ &= 1 - \tfrac{1}{2} \\ &= \tfrac{1}{2} \end{aligned}$$

Question: In a Bloom filter with n elements and a false positive rate of ε, what fraction of the bits in the array will be equal to 1?

Answer: Approximately half of them.

If we look up an item in the Bloom filter that isn't present, then on expectation we query two positions before returning false.

The math, in case you're curious:

Each bit is set to 1 with probability approximately $1 - e^{-k\alpha}$. We pick $k = \lg \varepsilon^{-1}$ and $\alpha = \ln 2 / \lg \varepsilon^{-1}$.

Probability a bit is set to 1: approximately

$$
\begin{aligned}
& 1 - e^{-(\lg \varepsilon^{-1})(\ln 2 / \lg \varepsilon^{-1})} \\
&= 1 - e^{-\ln 2} \\
&= 1 - \tfrac{1}{2} \\
&= \boldsymbol{\tfrac{1}{2}}
\end{aligned}
$$

Question: In a Bloom filter with n elements and a false positive rate of ε, what fraction of the bits in the array will be equal to 1?

Answer: Approximately half of them.

If we look up an item in the Bloom filter that isn't present, then on expectation we query two positions before returning false.

In other words, Bloom filters are fast when querying items not in S.

The math, in case you're curious:

Each bit is set to 1 with probability approximately $1 - e^{-k\alpha}$. We pick $k = \lg \varepsilon^{-1}$ and $\alpha = \ln 2 / \lg \varepsilon^{-1}$.

Probability a bit is set to 1: approximately

$$
\begin{aligned}
& 1 - e^{-(\lg \varepsilon^{-1})(\ln 2 / \lg \varepsilon^{-1})} \\
&= 1 - e^{-\ln 2} \\
&= 1 - \tfrac{1}{2} \\
&= \mathbf{\tfrac{1}{2}}
\end{aligned}
$$

Imagine you have a gigantic Bloom filter (say, one with 10^8 items in it) and we query for an item in the set.

This probes a large array in $\lg \varepsilon^{-1}$ effectively random locations.

Challenge: Reduce the number of cache misses done during a lookup.

Problem: Bloom filters have poor locality of reference, and queries are slower than suggested by the runtime bound.

`000001 … 000000 … 010000 … 000010 … 00100`

Looking Forward

- As always:

Can we do better?

- To improve our Bloom filter, we can either make
  - improvements to the query time, or
  - improvements to the space usage.
- Let's look at each of these in turn.

	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 our Bloom filter, we can either make
  - improvements to the query time, or
  - improvements to the space usage.
- Let's look at each of these in turn.

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

Claim: Bloom filters use close to the information-theoretic minimum number of bits for AMQ, but there’s still significant room for improvement.

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?

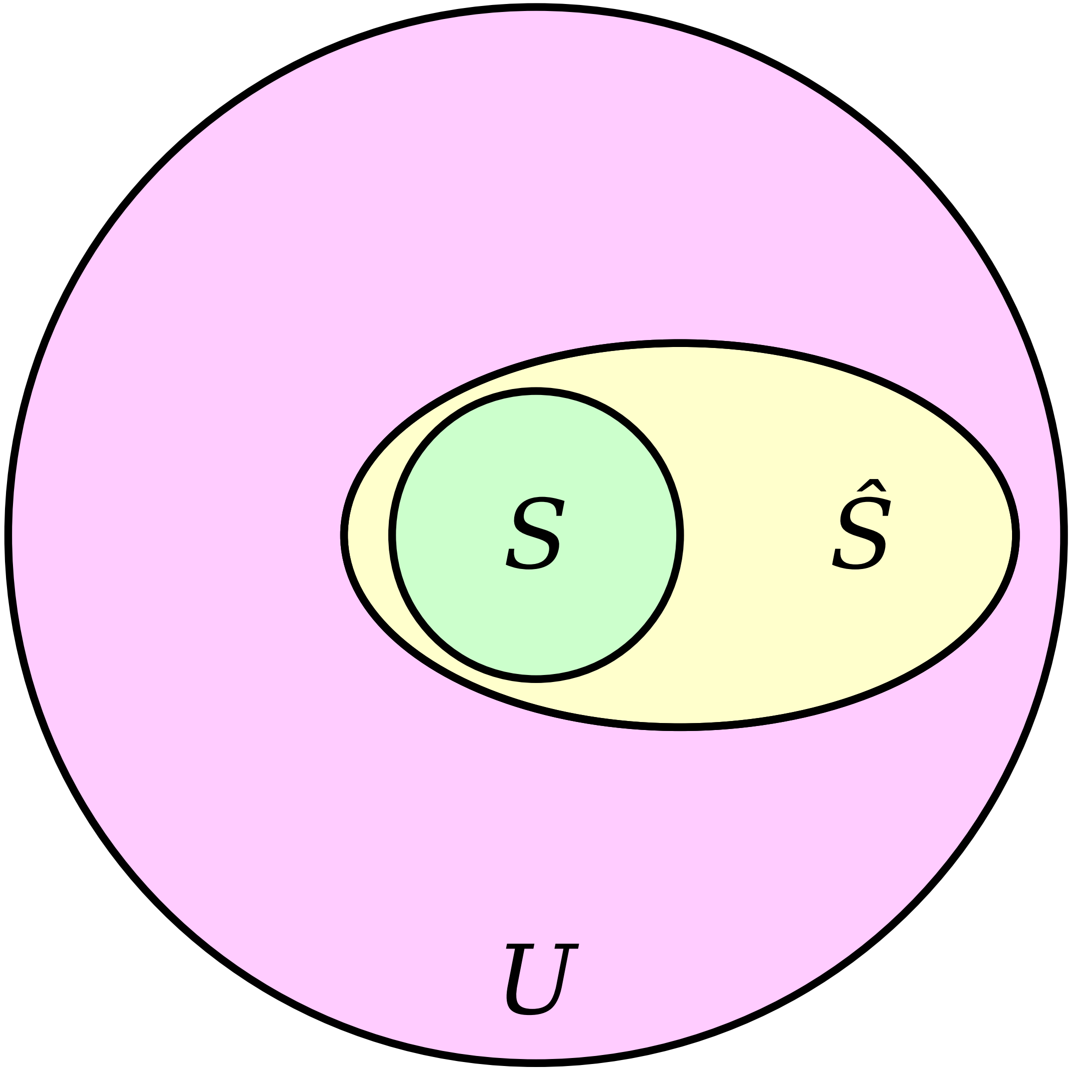
How much memory is needed to solve the approximate membership query problem?

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 ε$|U|$ extra elements due to the error rate.

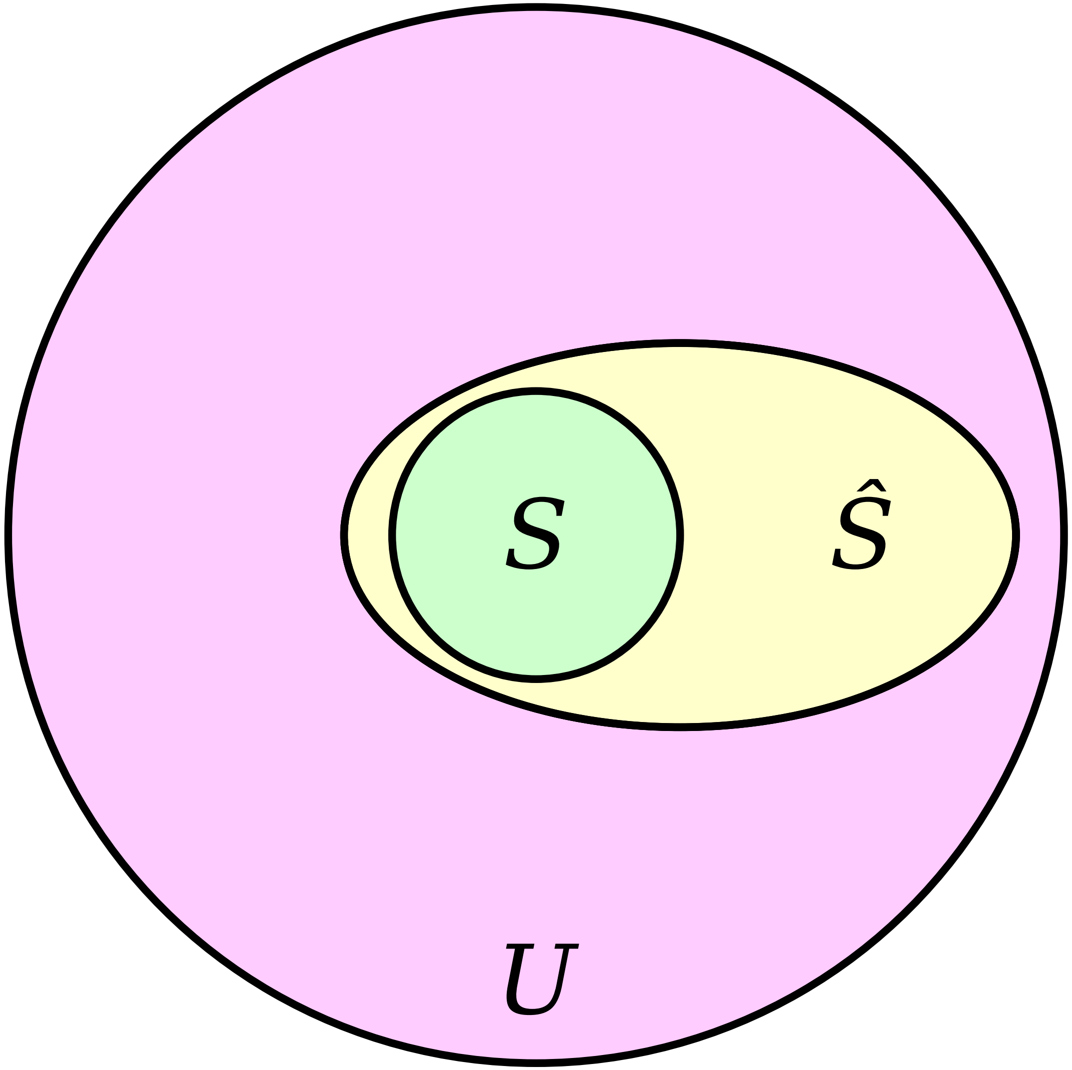


How much memory is needed to solve the approximate membership query problem?

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.

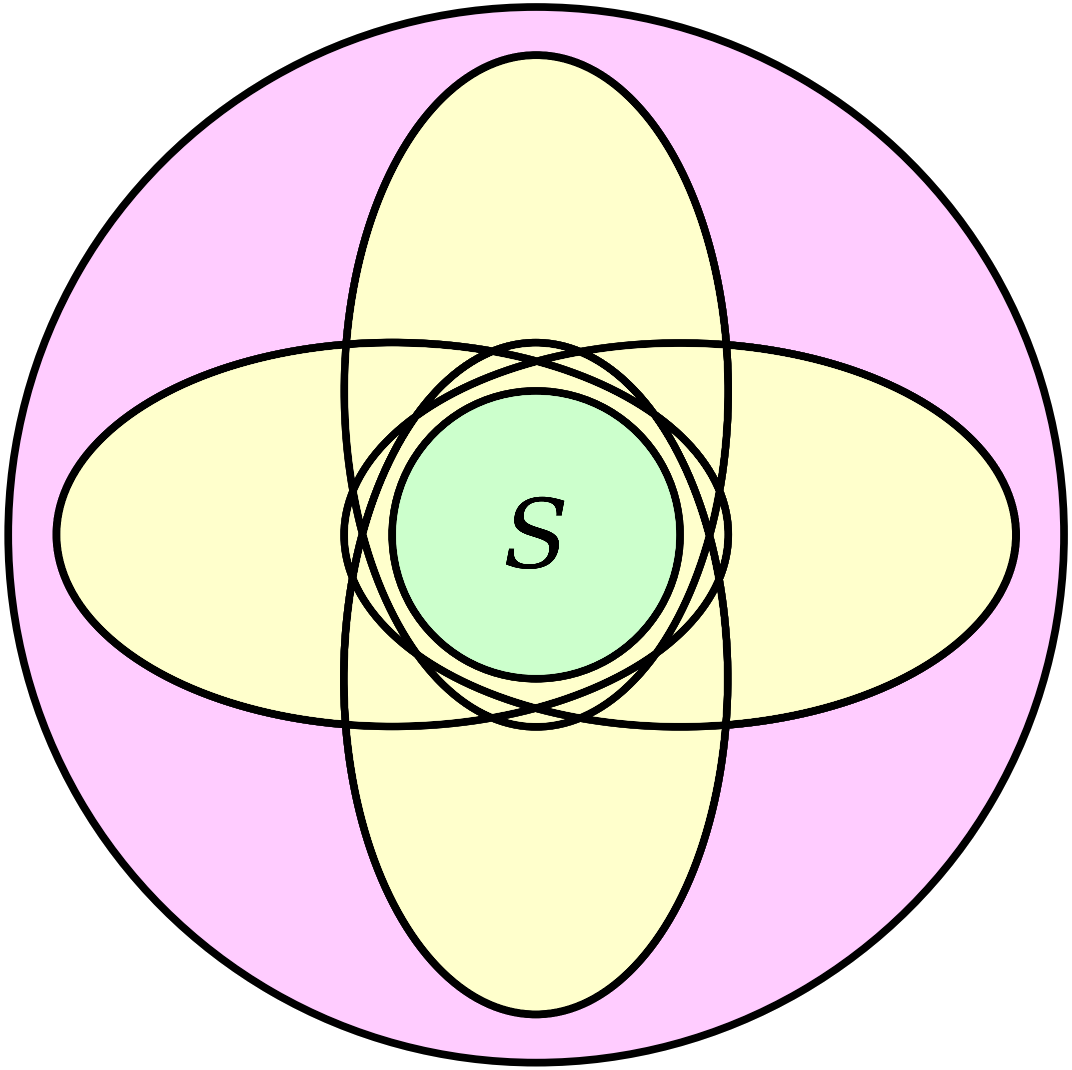


How much memory is needed to solve the approximate membership query problem?

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.

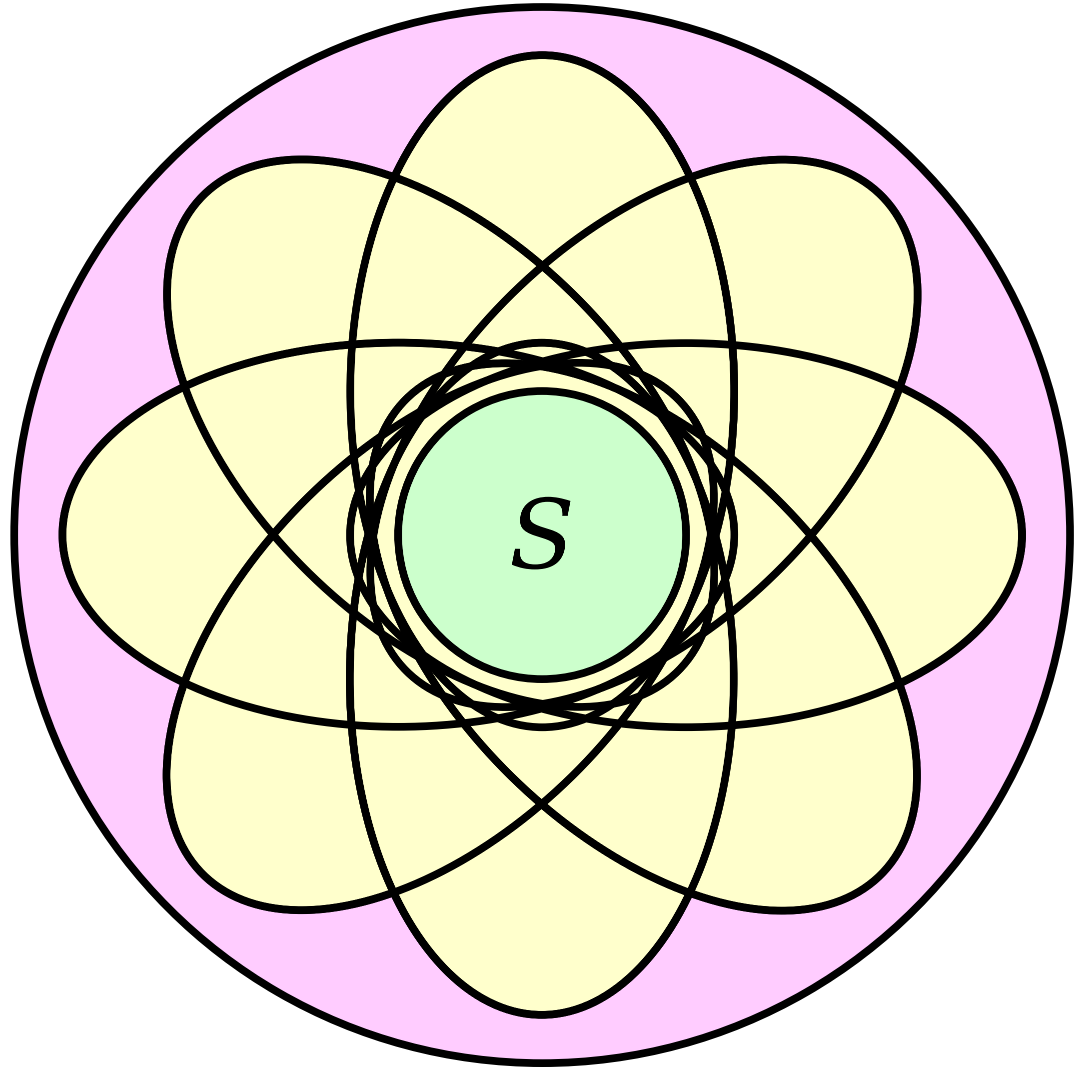


How much memory is needed to solve the approximate membership query problem?

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.



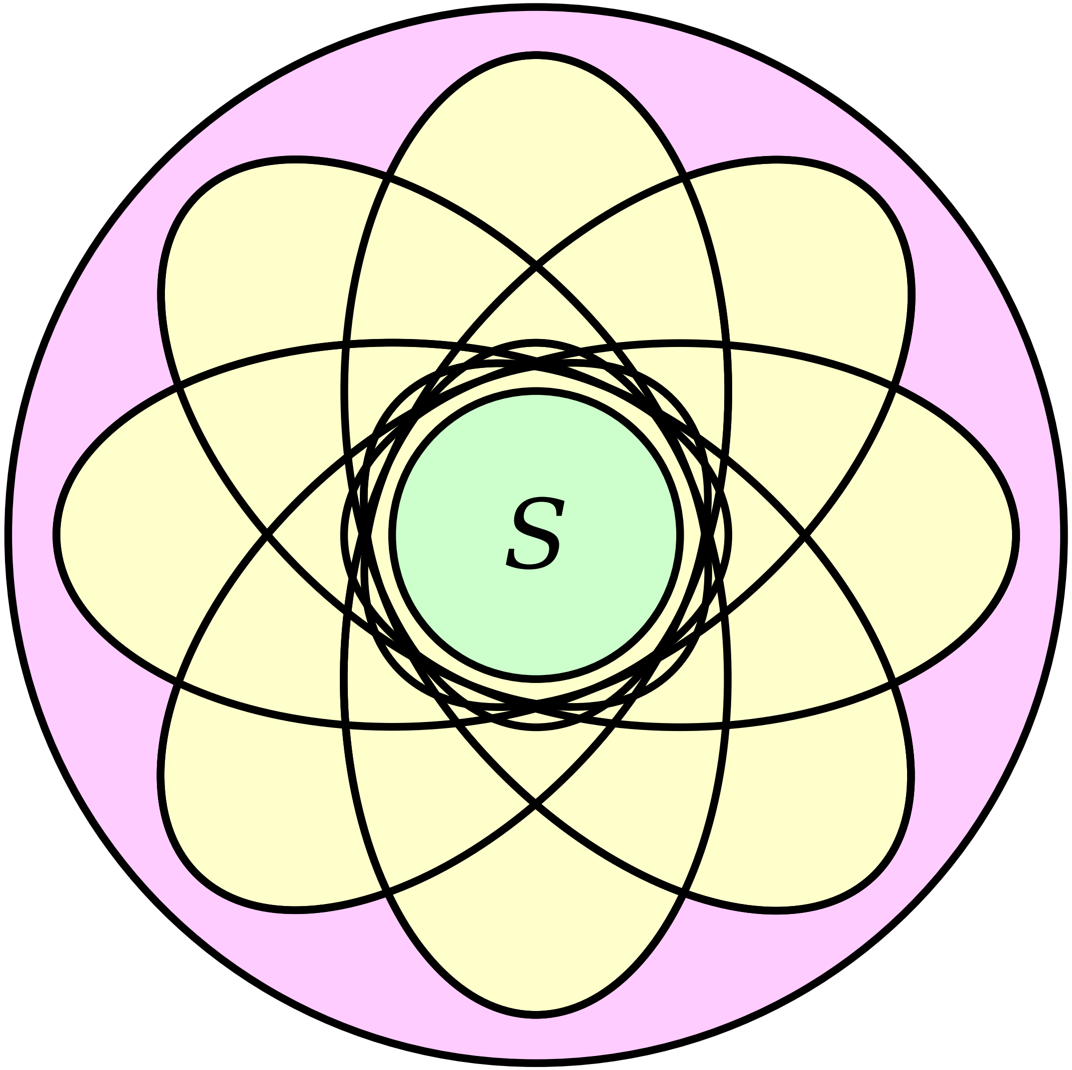
How much memory is needed to solve the approximate membership query problem?

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.

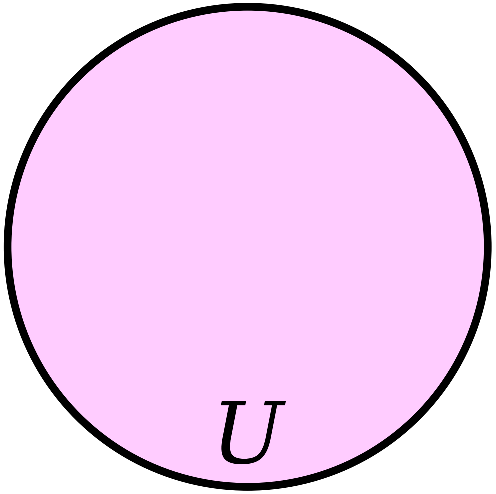
This AMQ encodes a set $\hat{S}$ of size roughly $\varepsilon|U|$ containing our set S.

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.

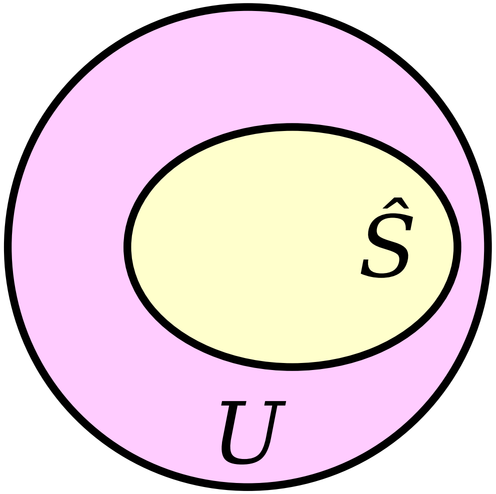


How much memory is needed to solve the approximate membership query problem?

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 $\varepsilon|U|$ containing our set S.

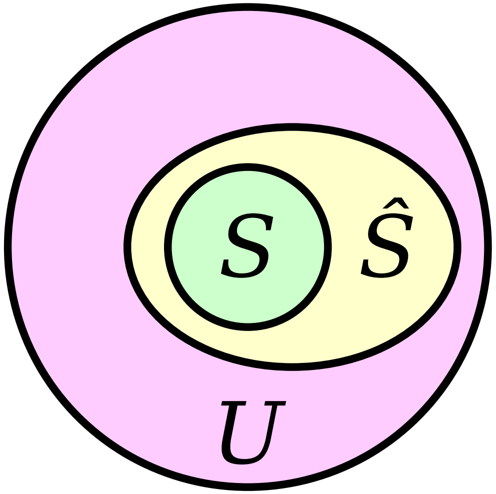


How much memory is needed to solve the approximate membership query problem?

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.



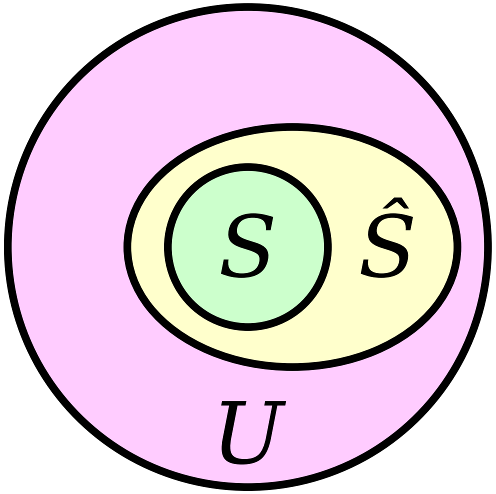
How much memory is needed to solve the approximate membership query problem?

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.

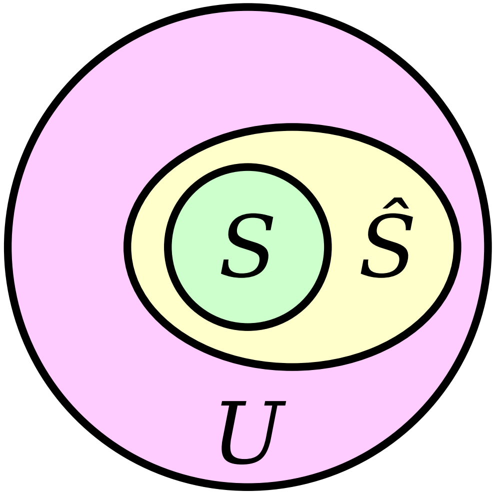
This AMQ encodes a set $\hat{S}$ of size roughly $\varepsilon|U|$ containing our set S.

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.

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

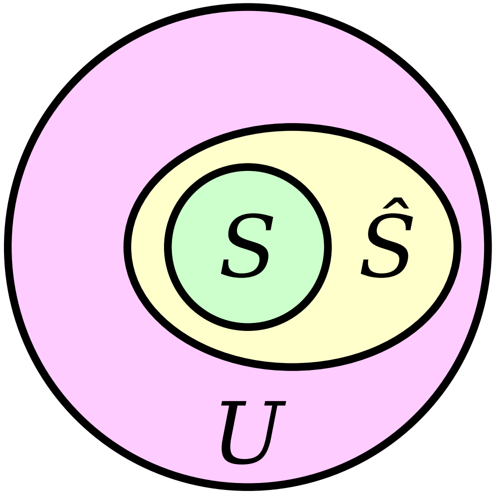
How much memory is needed to solve the approximate membership query problem?

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

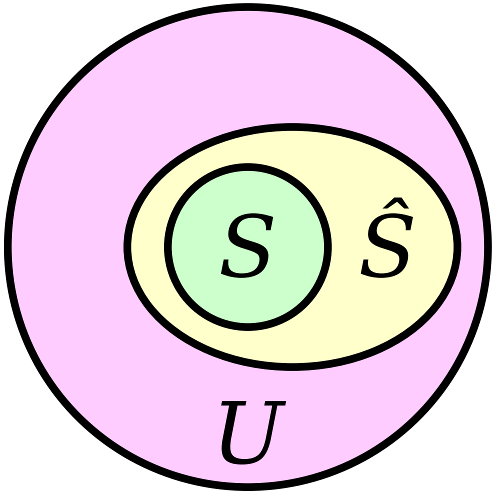
How much memory is needed to solve the approximate membership query problem?

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

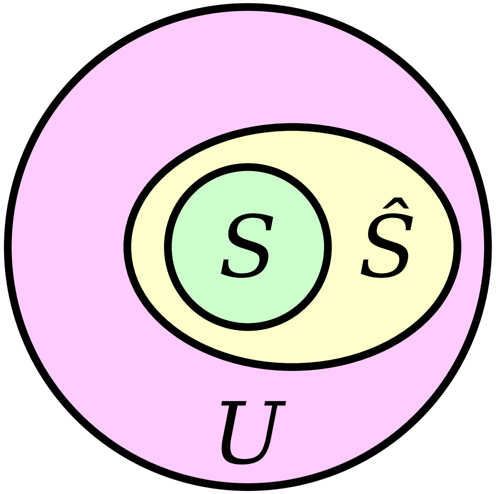
How much memory is needed to solve the approximate membership query problem?

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

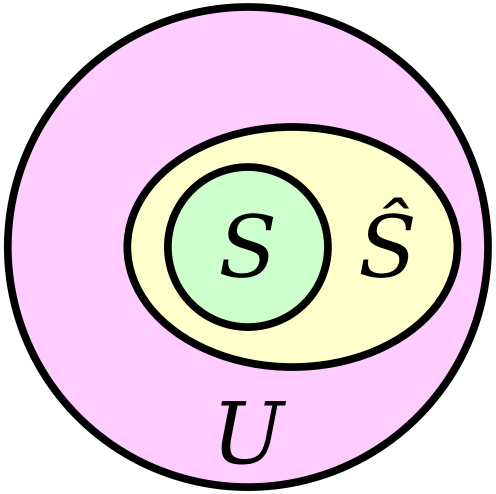
How much memory is needed to solve the approximate membership query problem?

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.



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

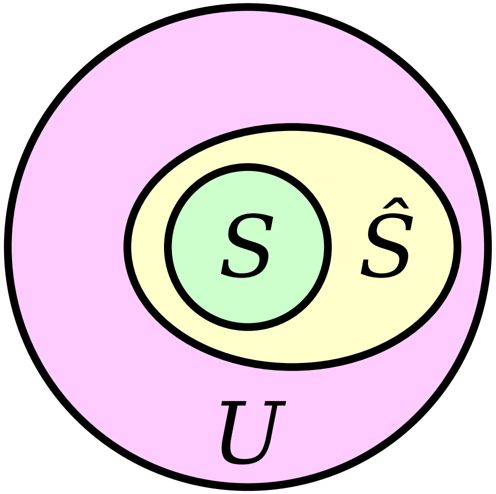
How much memory is needed to solve the approximate membership query problem?

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.



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

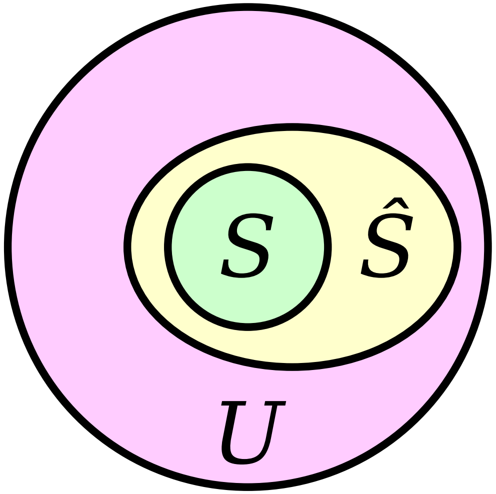
How much memory is needed to solve the approximate membership query problem?

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.



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

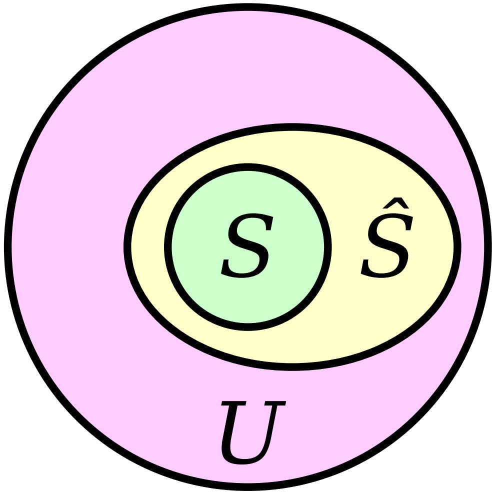
How much memory is needed to solve the approximate membership query problem?

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.



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

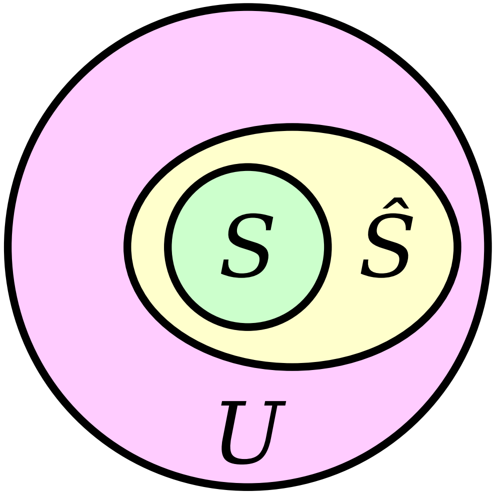
How much memory is needed to solve the approximate membership query problem?

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.



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

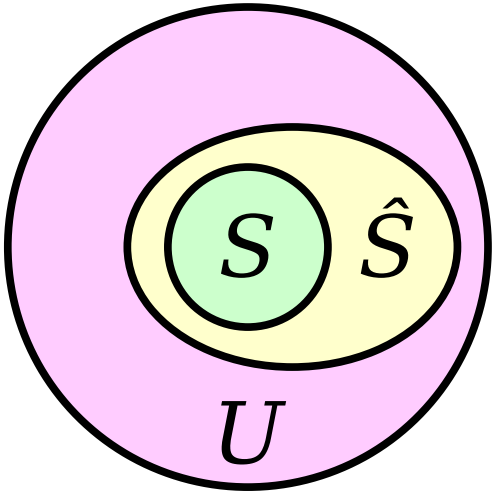
How much memory is needed to solve the approximate membership query problem?

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 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.

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

Where We're Going

	Bits / Element	Hashes/Q	Misses/Q
Bloom Filter (1970)	$1.44 \lg \varepsilon^{-1}$	$\lg \varepsilon^{-1}$	$\lg \varepsilon^{-1}$
? (2014)	$1.05 \lg \varepsilon^{-1} + 3.15$ *(for sufficiently small ε)*	3	2
? (2020)	$1.23 \lg \varepsilon^{-1}$	4	3
? (2021)	$1.08 \lg \varepsilon^{-1}$ *(for sufficiently large n)*	5	2
? (2021)	$1.03 \lg \varepsilon^{-1}$ *(for sufficiently large n)*	6	2

More to Explore

- ***Counting Bloom filters*** allow items to be added or removed from a Bloom filter without rebuilding the filter from scratch, at the cost of extra space overhead.
- ***d-Left counting Bloom filters*** are a space-optimized version of counting Bloom filters that use a clever technique to reduce the number of items hitting each slot.

Next Time

- ***Cuckoo Filters***
  - Adapting cuckoo hashing for AMQ, and outperforming the Bloom filter in practice.
- ***XOR Filters***
  - Rethinking Bloom filters to improve space utilization.
- ***Spatial Coupling***
  - Graph families with nice properties.