3-set-sketching

Monday, January 26, 2026 11:40 AM

Suppose we have two genomes S and T of equal length n, and we believe T is generated from S by randomly substituting with mutation rate θ, and S is a random string.

$S =$ AACCGGTT ....

$T =$ AACTGGTT ....

↑ substitutions with prob θ.

Want to find: mutation rate θ from strings S & T.

One sol: read alignment, and then check number of mutated positions

↳ works here because no indels, but if we have indels, then alignment is $O(n^2)$

Sampling sol: A k-mer $x \in S$ is unique if it is "long enough"

What is chance $x \in T$? $x =$ ACGT

↑↑↑↑

each pos θ chance of mutating

$\text{Prob}(x \in T \mid x \in S) = (1-\theta)^k$

$\Rightarrow$ If we sample 10,000 k-mers in S, and 9,000 of them are in T, then we can estimate by solving

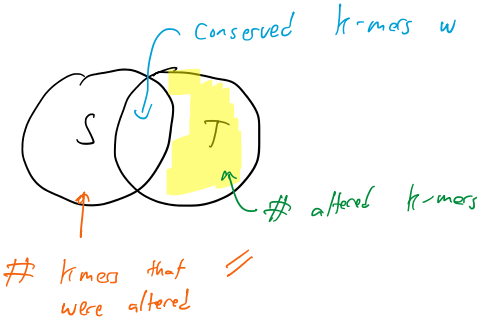
$$\frac{9000}{10000} = 0.9 = (1-\theta)^k$$

$$\Rightarrow 1-\theta = 0.9^{\frac{1}{k}}$$

$$\Rightarrow \theta = 1 - 0.9^{\frac{1}{k}}$$

More generally, $\theta = 1 - p^{\frac{1}{k}}$, where p is proportion shared k-mers (containment index)

Jaccard index sol: $j = \frac{|S \cap T|}{|S \cup T|} = \frac{w}{2n-w} = \frac{1}{\frac{2n}{w} - 1}$



$\frac{w}{n} = (1-\theta)^k$ from above $\Rightarrow \frac{n}{w} = \frac{1}{(1-\theta)^k}$

$$j = \frac{1}{2}.$$

were altered

$$j = \frac{1}{\frac{2}{(1-\theta)^k} - 1}$$

$$\Rightarrow \frac{2}{(1-\theta)^k} - 1 = \frac{1}{j}$$

$$\Rightarrow \frac{2}{(1-\theta)^k} = 1 + \frac{1}{j} = \frac{j+1}{j}$$

$$\Rightarrow \frac{1}{(1-\theta)^k} = \frac{j+1}{2j}$$

$$\Rightarrow \ln (1-\theta)^k = \ln \frac{2j}{j+1}$$

$$1-\theta = \frac{1}{k} \ln \frac{2j}{j+1}$$

$$\theta = 1 - \frac{1}{k} \ln \frac{2j}{j+1}$$

Thus, if we can find Jaccard index j, we can estimate θ.

Let's count.

- Kindergarten. Unary / tally marks. 卌 ||||
  Space complexity: $O(n)$
- Primary school. Positional encoding.
  Space complexity: $O(\log n)$

  Decimal: $521 = 500 + 21 + 1$
  Binary: $1011 = 8 + 2 + 1$
- Secondary school. Scientific notation?
  Space complexity: $O(\log \log n)$

  602,214,076,000,000,000,000,000 (Avogadro constant)
  $\hookrightarrow$ $6.022 \cdot 10^{23}$
  - 6.022: Constant space for 4 sig figs
  - 10^{23}: logarithmic in length of number

Problem: Can you really count up in scientific notation?

Ex. current count is $1.001 \cdot 10^4$. How do we increment the counter when we see a new item?

$1.001 \cdot 10^4 \rightarrow 1.002 \cdot 10^4$

Ex. current count is $1.001 \cdot 10^4$. How do we increment the counter when we see a new item?

$1.001 \cdot 10^4 \rightarrow 1.002 \cdot 10^4$ (jump of 10)

Intuition: maybe we only sometimes increment the counter.

Ideally, increment every 10th time, but then we need additional space to store the auxiliary counter, which becomes more expensive as the exponent increases.

Solution: Probabilistic counting.

Simpler to see in binary.

Idea: go back to regular binary counter.

Instead of increasing the counter for each item, flip a coin, and only increase the counter if it comes up heads.

To count 1,000,000 items, counter only goes up 500,000 times in expectation.

Unfortunately, not an asymptotic improvement.

LogLog counter: Given current counter value j, flip j coins, and increment counter to $j+1$ if all coins are heads.

On average, need only increment counter $\log n$ times, and the counter itself can be stored in binary, so again $O(\log \log n)$ space, but with ability to count.

Problem: Error in counter is big, on the order of a factor of 2.

Sol: Keep a k independent copies of counter and average.

$\Rightarrow$ order $\frac{1}{\sqrt{k}}$ relative error.

We'll come back to extensions of these counters used in practice.

Streaming & Sketching algorithms

Define: An algorithm is streaming if it only gets to see each data point once, but still returns an answer.

Typically, a streaming algorithm also has to operate in sublinear space,

data point ...

Typically, a streaming algorithm also has to operate in sublinear space, or else you could just store everything.

Ex. Counting, average, median, etc.

Define: An sketch of a dataset is a sublinear-size summary that can be composed with other sketches to compute an answer.

A streaming algorithm that produces a sketch is nice because you only have to see the data once, possibly in parallel.

Set membership problem

Given a stream of items $a_1, \ldots, a_n$, I want to know if I've already seen an item before. Say items are p bits big.

Naive: Store all items in a big list, & check each time.

$O(np)$ space

Alt 1: Store random hashes of items, which will typically be smaller than the items themselves.

Hashes need to be $\log n$ bits long to prevent collisions.

So $O(n \log n)$ space

Alt 2: Store truncated hashes of size $q < \log n$ bits.

Runs risk of collisions.

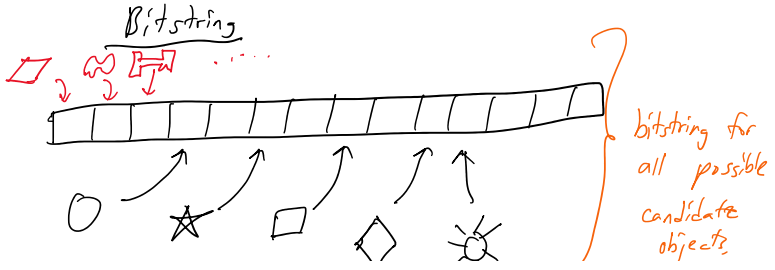
$O(nq)$ space

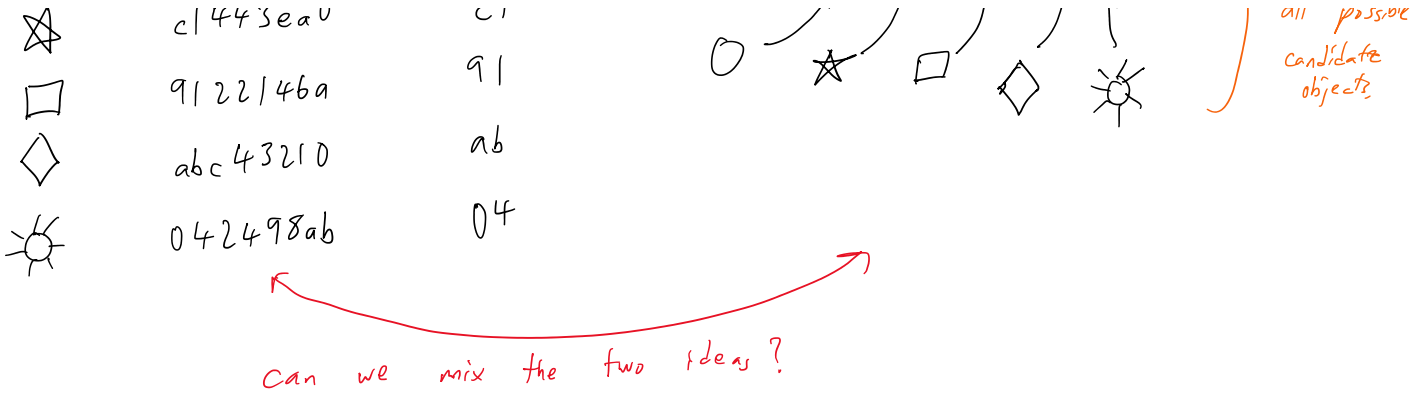
Exercise: Compute risk of collisions.

Alt 3: Bitstring with a place for each unique possible item in universe U.

$O(|U|)$ space

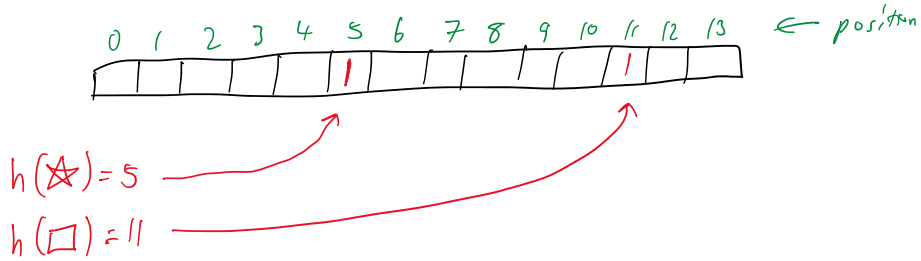
Item	Hashes	Truncated
○	d42aba71	d4
☆	c1443ea0	c1
▭	91221a6a	91





Intuition: Use hash function for bitstring location.

Start with empty (all 0's) bitstring, and treat it like a hash table, but instead of storing an item, we just set the bit there to 1.



Then to check if an item is present, check hashed location for set 1 bit.

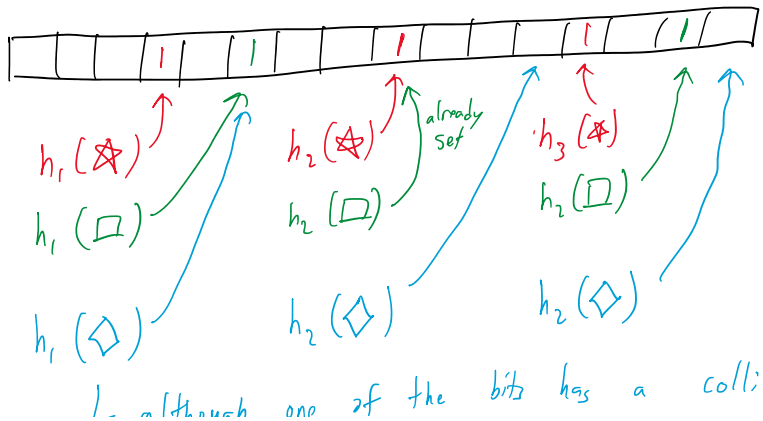
Exercise: Compute the probability of false positives.

There's a chance a different item set a bit to 1 that collides with the item we are checking.

How can we reduce the FP probability?

Use more than one hash function and set multiple bits simultaneously.
Then check each of those locations.

Bloom Filter [1970]



Check if ◇ is 1, although one of the bits has a collision, the

Check if ◇ is present: $h_1(◇)$ $h_2(◇)$ ...

↳ although one of the bits has a collision, the others don't, so we know we haven't seen ◇ before.

Thm: The optimal number of hash functions is $k = \frac{m}{n} \ln 2$,

where m = size of bitstring

and n = total number of stored items.

proof idea: Check false positive rate assuming each hash function is uniform random and independent.

Need about 9.6 bits per element to have a 1% FP rate, regardless of size of original elements.

Streaming: Once you have set the bits associated with an item, you don't ever need to see the item again, and can still check for set membership.

Sketching: If you have two separate streams of data & generate Bloom filters of each of them, you can later merge the two Bloom filters by just OR'ing the bits — that's equivalent to the Bloom filter you would've gotten if you had directly generated a Bloom filter of both streams.

Back to counting: let's count the number of distinct items in a stream $a_1, \ldots, a_n$.

Can't just use a counter, because you need to ensure you don't double-count an item you see twice.

One idea could be to use a Bloom filter to not double count & only increment the counter if not already present.

↳ Needs about 10 bits per item to have 1% error.

Alt soln: Can we estimate it directly from the Bloom filter?

Yes! Look at how many bits are set taking into account overlap.

Number of unique items is $n^* \approx -\frac{m}{k} \ln\left[1 - \frac{X}{m}\right]$, where X is number of set bits

↳ this works because even if an item is "added" to

Number of unique items is ... k ... of set bits

↳ this works because even if an item is "added" to a Bloom filter multiple times, the bits that are set are the same each time, so we don't double count.

But this still takes $O(n)$ space. Can we do better?

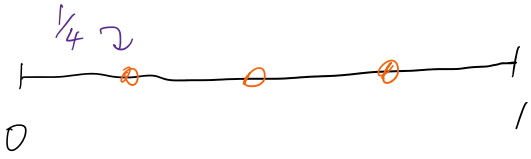
Idea 2: minimum hash value estimator [Flajolet, Martin, Ziv Bar-Yossef]

NOT Ziv Bar-Joseph from our department

What's the expected value of a unif $[0,1]$ r.v.? 0.5

What about $\min[X_1, X_2]$ where $X_1, X_2 \sim$ unif $[0,1]$ i.i.d.?

What about $\min[X_1, X_2, X_3]$?



In general $\mathbb{E}\min[X_1, \ldots, X_n] = \frac{1}{n+1}$.

If we could assign a random unif. number to each item in a list, then the minimum value would be an estimator for n.

We can! Just use a hash function $h: U \to [0,1)$.

How much space does it take to store a hash value? $\frac{1}{n+1}$ is about the same size as $(n+1)$, so $O(\log n)$ bits of storage.

$\frac{1}{16} = 0.\underbrace{0001}_{\log_2 16 \text{ length}}$ in binary

A single hash value is going to have large error. So we repeat 10,000 times with different hash functions, to get ~1% error.

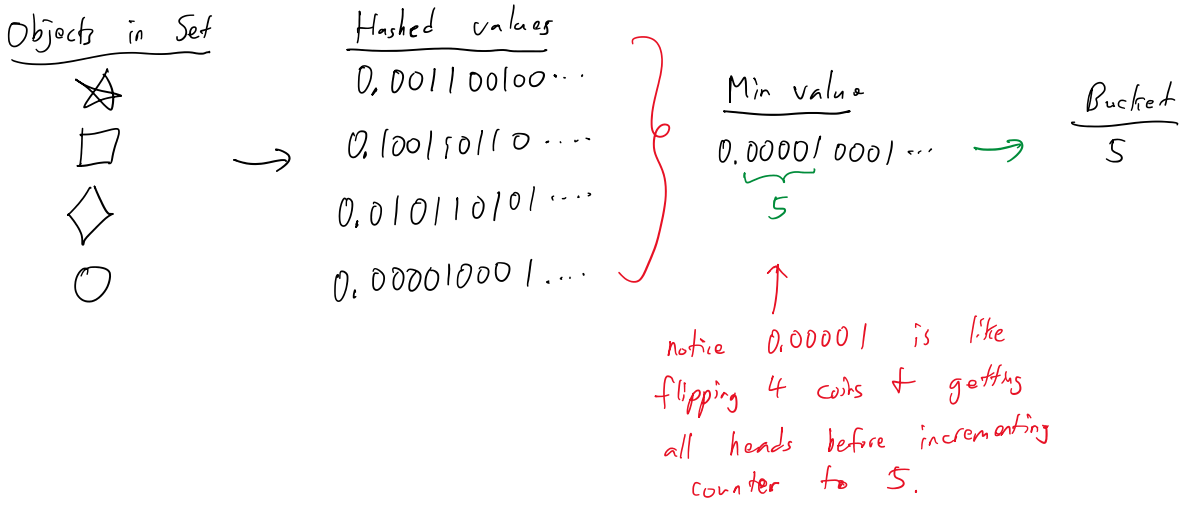
$\Rightarrow O\left(\frac{1}{\varepsilon^2} \log n\right)$ space for ε relative error.

(since standard error is $\frac{1}{\sqrt{k}}$ for k trials)

Notice: Storing $\frac{1}{n}$ takes about the same space as storing n, which is kind of like counting. Can we use LogLog counter instead?

Notice: Storing $\frac{1}{n}$ takes about the same space as storing n, which is kind of like counting. Can we use LogLog counter instead?

Only need to store orders of magnitude to get a good estimate, so can compress hashed values.

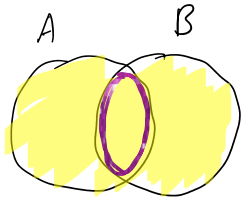


Space complexity $O\left(\frac{1}{\varepsilon^2}\log\log n\right)$

With some additional tricks and corrections, this is the widely-used HyperLogLog [Flajolet, et al 2007] algorithm for set union cardinality.

Exercise: Show that you can merge two HyperLogLog sketches to get the number of unique items in two separate sets. } i.e. that this is a sketching alg.

Notice: We can now compute $|A|$ and $|A \cup B|$, losslessly in the sense that our approx. error is controlled by the ε relative error parameter.



$A \cup B$

Then we can also estime $A \cap B$ by $|A \cap B| = |A| + |B| - |A \cup B|$ (inclusion-exclusion)
(HyperLogLog)

Exercise: Show that this error can be much worse than the HLL relative error.

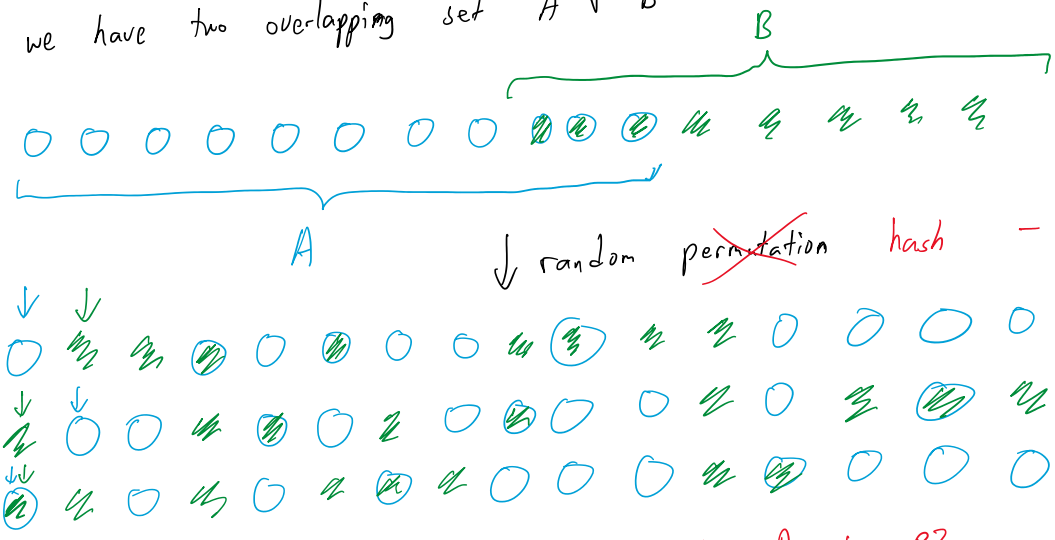
sketch $|A \cap B|$ instead to get higher accuracy.

Exercise: Show that this error can be much worse than ...

So we may want to directly sketch $|A \cap B|$ instead to get higher accuracy.

Define: The Jaccard [1902] index measures the similarity b/t two sets $\frac{|A \cap B|}{|A \cup B|}$.

Suppose we have two overlapping set A & B



— $O(\log n)$ bits

What's the chance the left most item is in both A & B?

Exactly $\frac{|A \cap B|}{|A \cup B|} = \text{Prob}\left(\min_h(A) = \min_h(B)\right)$

So only need to store minimum values — same as above!

We can now measure the Jaccard index of any two sets efficiently, including sets of k-mers. This is how tools like Mash & Dashing work.