Chapter 9: Applications to Computing

9.5: Distinct Elements

Slides (Google Drive) Starter Code (GitHub)

9.5.1 Motivation

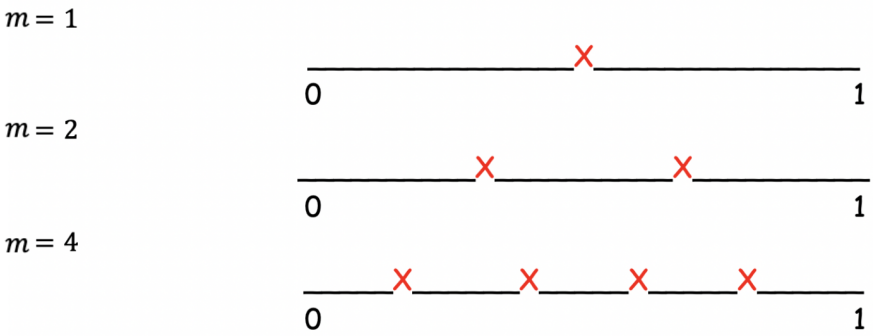
YouTube wants to count the number of ***distinct*** views for a video, but doesn't want to store all the user ID's. How can they get an accurate count of users without doing so? Note: A user can view their favorite video several times, but should only be counted as *one* distinct view.

Before we attempt to solve this problem, you should wonder: why should we even care? For one of the most popular videos on YouTube, let's say there are $N = 2$ billion views, with $n = 900$ million of them being distinct views. How much space is required to accurately track this number? Well, let's assume a user ID is an 8-byte integer. Then, we need $900,000,000 \times 8$ bytes total if we use a *Set* to track the user IDs, which requires 7.2 **giga**bytes of memory for ONE video. Granted, not too many videos have this many views, but imagine now how many videos there are on YouTube: I'm not sure of the exact number, but I wouldn't be suprised it it was in the tens or hundreds of millions, or even higher!

It would be great if we could get the number of distinct views with constant space $\mathcal{O}(1)$ instead of linear space $\mathcal{O}(n)$ required by storing all the IDs (let's say a *single* 8-byte floating point number instead of 7.2 GB). It turns out we (approximately) can! There is no free lunch of course - we can't solve this problem exactly with constant memory. But we can trade this space for some error in accuracy, using the continuous Uniform random variable! That is, we will potentially have huge memory savings, but are okay with accepting a distinct view count which has some margin of error.

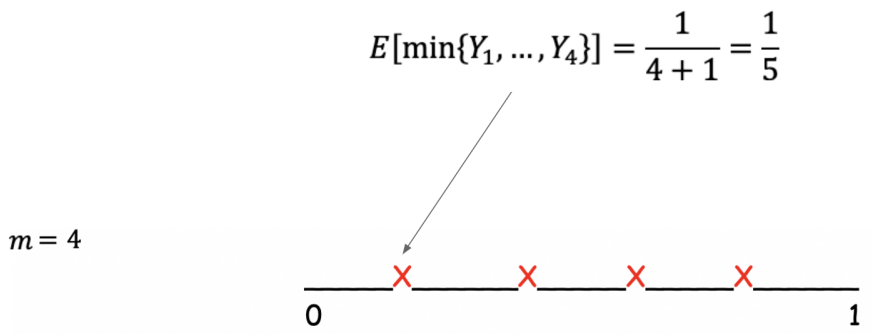
9.5.2 Intuition

This seemingly unrelated calculation will be crucial in tying our algorithm together - I'll ask for your patience as we do this. Let $U_1, \ldots, U_m$ be m iid (independent and identically distributed) RVs from the *continuous* Unif$(0, 1)$ distribution. If we take the *minimum* of these m random variables, what do we "expect" it to be? That is, if $X = \min\{U_1, \ldots, U_m\}$, what is $\mathbb{E}[X]$? Before actually doing the computation, let's think about this intuitively and see some pictures.



- If $m = 1$ (only one uniform RV), we expect it to be right in the center at $1/2$.
- If $m = 2$ (two continuous uniform RVs), we expect the two points to be at $1/3$ and $2/3$, with the minimum being the smaller of the two at $1/3$.
- If $m = 4$, we might expect the four points to be at $1/5, 2/5, 3/5, 4/5$, and so the minimum is actually at $1/5$.

See below for more details on the last case where $m = 4$.



What these examples are getting at is that, the expected value of the smallest of m Unif$(0, 1)$ RVs is

$$\mathbb{E}[X] = \mathbb{E}[\min\{U_1, \ldots, U_m\}] = \frac{1}{m+1}$$

I promise this will be the key observation in making this clever algorithm work. If you believed the intuition above, that's great! If not, that's also fine, so I'll have to prove it to you formally below. Whether you believe me or not at this point, you are definitely encouraged to read through the strategy as it may come up many times in your future.

Theorem 9.5.1: Expectation of Min of IID Uniforms

If $U_1, \ldots, U_m \sim$ Unif$(0, 1)$ (continuous) are iid (independent and identically distributed), and $X = \min\{U_1, \ldots, U_m\}$ is their minimum, then $\mathbb{E}[X] = \frac{1}{m+1}$.

Proof of Expectation of Min of IID Uniforms.

We should start working with probabilities first (e.g., the CDF $F_X(x) = \mathbb{P}(X \leq x)$) and take the derivative to find the PDF (this is a common strategy for dealing with continuous RVs). Actually, we'll compute $\mathbb{P}(X > x)$ first (how is this related to the CDF F_X?):

$$\begin{aligned}
\mathbb{P}(X > x) &= \mathbb{P}(\min\{U_1, \ldots, U_m\} > x) && \text{[def of } X\text{]} \\
&= \mathbb{P}(U_1 > x, U_2 > x, \ldots, U_m > x) && \text{[minimum is greater than } x \text{ iff ALL are]} \\
&= \prod_{i=1}^{m} \mathbb{P}(U_i > x) && \text{[independence]} \\
&= \prod_{i=1}^{m} (1 - x) && \text{[1−CDF of Unif}(0,1)\text{]]]} \\
&= (1 - x)^m && \text{[all have the same distribution]}
\end{aligned}$$

Some of these steps need more justification. For the second equation, we use the fact that the minimum of numbers is greater than a value if and only if all of them are (think about this). For the next equation, the probability of all of the $U_i > x$ is just the product of the m probabilities by our independence assumption. And finally, for $U_i \sim \text{Unif}(0,1)$, we know its CDF (look it up in our table) is $\mathbb{P}(U_i \leq x) = \dfrac{x-0}{1-0} = x$, and so $\mathbb{P}(U_i > x) = 1 - \mathbb{P}(U_1 \leq x) = 1 - x$.

Now, we have that

$$F_X(x) = 1 - \mathbb{P}(X > x) = 1 - (1-x)^m$$

I'll leave it to you to compute the density $f_X(x)$ by differentiating the CDF we just computed, and then using our standard expectation formula (the minimum of numbers in $[0,1]$ is also in $[0,1]$):

$$\mathbb{E}[X] = \int_0^1 x f_X(x) dx$$

and you should get $\mathbb{E}[X] = \dfrac{1}{m+1}$ after all this work!

□

If you are thinking of giving up now, I promise this was the hardest part! The rest of the section should be (generally) smooth sailing.

9.5.3 The Algorithm

The problem can be formally modelled as follows: a video receives a **stream** of 8-byte integers (user ID's), $x_1, x_2, \ldots, x_N$, but there are only n *distinct* elements ($1 \leq n \leq N$), since some people rewatch the video. We don't know what N is, since people continuously view the video, but assume we cannot store all N elements; we can't even store the n distinct elements.

Suppose the universe of user ID's is the set $\mathcal{U}$ (think of this as all 8-byte integers), and we have a single ***uniform*** hash function $h : \mathcal{U} \to [0,1]$ (i.e., for an user ID y, pretend $h(y)$ is a **continuous** Unif$(0,1)$ random variable). That is, $h(y_1), h(y_2), ..., h(y_k)$ for any k **distinct** elements are iid continuous Unif$(0,1)$ random variables, but since the hash function always gives the same output for some given input, $h(y_1)$ and $h(y_1)$ are the "same" Unif$(0,1)$ random variable.

To parse that mess, let's see two examples. These will also hopefully give us the lightbulb moment!

Example(s)

Suppose we have user IDs watch the video in this order:

$$13, 25, 19, 25, 19, 19$$

This is a *stream* of user IDs. From this, there are 3 distinct views (13,25,19) out of 6 total views. The uniform hash function h might give us the following stream of hashes:

$$0.51, 0.26, 0.79, 0.26, 0.79, 0.79$$

Note that all of these numbers are between 0 and 1 as they should be, as they are supposedly Unif$(0,1)$. Note also that for the same user ID, we get the same hash! That is, $h(19)$ will *always* return 0.79, $h(25)$ is always 0.26, and so on. Now go back and reread the previous paragraph and see if it makes more sense.

Example(s)

Consider the same stream of $N = 6$ elements as the previous example, with $n = 3$ *distinct* elements.

1. How many *independent* Unif$(0, 1)$ RVs are there total: N or n?
2. If we only stored the minimum value every time we received a view, we would store the single floating point number 0.26 as it is the smallest hash of the six. If we didn't know n, how might we exploit 0.26 to get the value of $n = 3$? Hint: Use the fact we proved earlier that $\mathbb{E}[\min\{U_1, \ldots, U_m\}] = \dfrac{1}{m+1}$ where $U_1, \ldots, U_m$ are iid.

Solution

1. As you can see, we only have three iid Uniform RVs: $0.26, 0.51, 0.79$. So in general, we'll have the minimum n (and not N) RVs.
2. Actually, remember that the expected minimum of n distinct/independent values is approximately $\dfrac{1}{n+1}$ as we showed earlier. Our 0.26 isn't exactly equal to $\mathbb{E}[X]$, but it is an *estimate* for it! So if we solve
$$0.26 \approx \mathbb{E}[X] = \frac{1}{n+1}$$
we would get that $n \approx \dfrac{1}{0.26} - 1 \approx 2.846$. Rounding this to the nearest integer of 3 actually gives us the correct answer!

   So our strategy is: keep a running minimum (a single floating point which ONLY takes 8 bytes). As we get a stream of user IDs $x_1, \ldots, x_N$, hash each one and update the running minimum if necessary. When we want to estimate n, we just reverse solve $n = \mathsf{round}\left(\dfrac{1}{\mathbb{E}[X]} - 1\right)$, and that's it! Take a minute to reread this example if necessary, as this is the entire idea!

□

Here is the pseudocode for the algorithm we just described:

Algorithm 1 Distinct Elements Algorithm

```
function INITIALIZE()
    val ← ∞
function UPDATE(x)
    val ← min{val, hash(x)}
function ESTIMATE()
    return round(1/val − 1)
initialize()                          ▷ Initialize our single float variable
for i = 1, ..., N: do                 ▷ Loop through all stream elements
    update(x_i)                       ▷ Update our single float variable
return estimate()                     ▷ An estimate for n, the number of distinct elements.
```

This is known as the **Distinct Elements** algorithm! We start our single floating point minimum (called val below) at ∞, and repeatedly update it. The key observation is that we are only taking the minimum of n iid

Uniform RVs, and NOT N because h always returns the same value given the same input. Reverse-solving for $\mathbb{E}[X] = \dfrac{1}{m+1}$ gives us an *estimate* for m since $\mathbb{E}[X]$ (which is stored in the variable val) is only an approximation. Note we want to round to the nearest integer because n should be an integer.

This algorithm sounds great right? One pass over the data (which is the best we can do in time complexity), and one single float (which is the best we can do in space complexity)! But you have to remember the tradeoff is in the accuracy, which we haven't seen yet.

The reason the previous example was spot-on is because I cheated a little bit. I ensured the three values $0.26, 0.51, 0.79$ were close to where they were supposed to be: $0.25, 0.50, 0.75$. Actually, it's most important that just the minimum is on-target. See the following example for an unfortunate situation.

Example(s)

Suppose we have $N = 7$ user IDs watch the video in this order:

$$11, 34, 89, 11, 89, 23, 23$$

The uniform hash function h might give us the following stream of $N = 7$ hashes:

$$0.5, 0.21, 0.94, 0.5, 0.94, 0.1, 0.1$$

Trace the distinct elements algorithm above by hand and report the value that it will return for our estimate. Compare it to the true value of $n = 4$ which is unknown to the algorithm.

Solution

At the end of all the updates, val will be equal to the minimum hash of 0.1. So the estimated number of distinct elements is

$$\mathsf{round}\left(\frac{1}{0.1} - 1\right) = 9$$

There are only $n = 4$ distinct elements though! The reason this time it didn't work out well for us is that the minimum value was *supposed* to be around $1/5 = 0.2$, but was actually 0.1. This is not necessarily a huge difference until we take its reciprocal... □

That's it! The code for this algorithm is actually pretty short and sweet (imagine converting the pseudocode above into code). If you take a step back and think about what machinery we needed, we needed continuous RVs: the idea of PDF/CDF, and the Uniform RV. The mathematical/statistical tools we learn have many applications to computer science; we have several more to go!

9.5.4 Improving Performance (Optional)

You may wonder how we can improve this estimate. The problem is that the variance of the minimum is pretty high (e.g., it was 0.1 last time instead of 0.2): how can we reduce it? Actually, *independent* repetitions is always an excellent strategy (if possible) to get better estimates!

If $X_1, \ldots, X_n$ are iid RVs with mean μ and variance σ^2, we'll show that the **sample mean** $\bar{X}_n = \dfrac{1}{n}\sum_{i=1}^{n} X_i$ has the same mean but lower variance as each X_i.

$$\mathbb{E}\left[\bar{X}_n\right] = \mathbb{E}\left[\frac{1}{n}\sum_{i=1}^{n} X_i\right] = \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}\left[X_i\right] = \frac{1}{n}n\mu = \mu$$

Also, since the X_i's are independent, variance adds:

$$\mathsf{Var}\left(\bar{X}_n\right) = \mathsf{Var}\left(\frac{1}{n}\sum_{i=1}^{n} X_i\right) = \frac{1}{n^2}\sum_{i=1}^{n}\mathsf{Var}\left(X_i\right) = \frac{1}{n^2}n\sigma^2 = \frac{\sigma^2}{n}$$

That is, the sample mean will have the same expectation, but the variance will go down linearly! Why might this make sense? Well, imagine you wanted to estimate the height of American adults: would you rather have a sample of 1, 10, or 100 adults? All would be correct in expectation, but the size of 100 gives us more confidence in our answer!

So if we instead estimate the minimum $\mathbb{E}\left[X\right] = \dfrac{1}{n+1}$ with the average of k minimums instead of just one, we should get a more accurate estimate for $\mathbb{E}\left[X\right]$ and hence n, the number of distinct elements, as well!

So, imagine we had k independent hash functions instead of just one: $h_1, \ldots, h_k$, and k minimums $\mathsf{val}_1, \mathsf{val}_2, \ldots, \mathsf{val}_k$.

Stream $\rightarrow$	13	25	19	25	19	19	val_i
h_1	0.51	0.26	0.79	0.26	0.79	0.79	0.26
h_2	0.22	0.83	0.53	0.84	0.53	0.53	0.22
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
h_k	0.27	0.44	0.72	0.44	0.72	0.72	0.27

Each row represents one hash function h_i, and the last column in each row is the minimum for that hash function. Again, we're only keeping track of the k floating point minimums in the final column. Now, for improved accuracy, we just take the average of the k minimums first, before reverse-solving. Imagine $k = 3$ (so there were no rows in $\ldots$ above). Then, a good estimate for the true minimum $\mathbb{E}\left[X\right]$ is

$$\mathbb{E}\left[X\right] \approx \frac{0.26 + 0.22 + 0.27}{3} = 0.25$$

So our estimate for n is $\mathsf{round}\left(\dfrac{1}{0.25} - 1\right) = 3$, which is perfect! Note that we basically combined 3 distinct elements instances with h_1, h_2, h_3 individually from earlier, in a way that reduced the variance! The individual estimates $0.26, 0.22, 0.27$ were varying around 0.25, but their average was even closer!

Now our memory is just $\mathcal{O}(k)$ instead of $\mathcal{O}(1)$, but we get a better estimate as a result. It is up to you to determine how you want to tradeoff these two opposing quantities.

9.5.5 Summary

We just saw today an extremely clever use of continuous RVs, applied to computing! In general, randomness (the use of a random number generator (RNG)) in algorithms and data structures often can help improve either the time or space (or both)! We saw earlier with the bloom filter how adding a RNG can save a ton of space in a data structure. Even if you don't go on to study machine learning or theoretical CS, you can see what we're learning can be applied to algorithms and data structures, arguably the core knowledge of every computer scientist.