COMS 4995-2: Advanced Algorithms (Spring'20) — Jan 23, 2020

Lecture 2: Approximate Counting Continued, Hashing

Instructor: *Alex Andoni* — Scribes: *Rebecca Calinsky*

1 Approximate Counting Continued . . .

Recap from last lecture:
- Problem simply concerns counting up to n
- Morris Algorithm was introduced, which has one register, X, initialized to 0
- We increment as we see ticks, but much more slowly than if we just count
- The bigger X becomes, the less likely it is to increment

1.1 Morris Algorithm

- Initialize $X = 0$

- At tick: $X = \begin{cases} X, \text{ with probability } 1 - 2^{-X} \\ X + 1 \text{ with probability } 2^{-X} \end{cases}$

- The estimator: $\hat{n} = 2^X - 1$

PART 1: What we proved last lecture . . .

Claim 1. *When we define $X_n = X$, the count after n ticks, then $\mathbb{E}[\hat{n}] = \mathbb{E}[2^{X_n} - 1] = n$*

Proof. (Supplied last lecture: showed expectation of the estimator equal to n using induction on n) □

PART 2: We ask, "Did we gain anything?"
In Part 1, the estimator is in the right ballpark, and now we want to prove that the number of bits necessary has improved drastically (lg lg n, instead of lg n, where lg is implicitly $\lg_2$ from here on).

Claim 2. $\Pr[\lg X \le c \cdot \lg \lg n] \ge 0.9$ *for some* $c \ge 1$

Proof. Apply the **Markov Bound** on $\hat{n}$:

$$\Pr[\hat{n} > 10n] \le \frac{n}{10n} = 0.1$$

$$\implies \Pr[2^{X_n} - 1 \le 10n] \ge 0.9$$

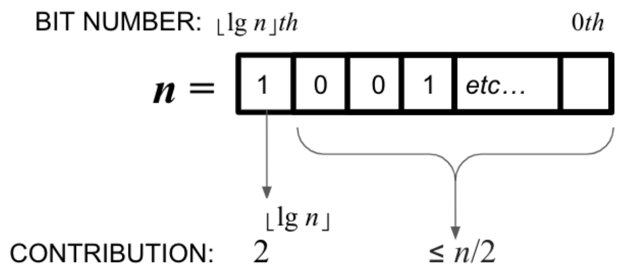
$$2^{X_n} - 1 \le 10n \implies \lg X_n \le \lg\lg(10n+1)$$

$$\implies \exists\, c \ge 1 \text{ such that } \Pr[\lg X_n \le c \cdot \lg\lg n] \ge \frac{9}{10}$$

$\square$

1.2 Intuition behind the algorithm

We are counting up to n ticks, and we can think about n as being represented in binary in the following illustration:



In the above, the index of the most significant bit i_{max} (whose contribution is $2^{\lfloor \lg n \rfloor} \in [n/2, n]$) is all you need in order to have a factor 2 approximation of the number n. The number of bits necessary to communicate the number i_{max} is the lg of the number of bits needed to represent n, which results in $\lg\lg n$.

If X = the most significant bit (index) of the current n, then X should increase only when n roughly doubles. In other words, the larger of number, the less frequently its max index changes.

X is incremented with $\Pr \approx \frac{1}{n} \approx 2^{-X}$

1.3 Show $\hat{\mathbf{n}} \approx \mathbf{n}$ with good probability

We will use another concentration bound called the **Chebyshev Bound**, and compute $\mathrm{Var}[\hat{n}]$.

Claim 3. $\mathrm{Var}[\hat{n}] \le \frac{3}{2}n(n+1) + 1$

Proof. $\mathrm{Var}[\hat{n}] = \mathrm{Var}[2^{X_n} - 1] = \mathbb{E}[(2^{X_n} - 1)^2] - n^2 = \mathbb{E}[2^{2X_n}] + \underbrace{1 - 2\,\underbrace{\mathbb{E}[2^{X_n}]}_{n+1} - n^2}_{\le 0} \le \mathbb{E}[2^{2X_n}]$

Inductive hypothesis:

$$\mathbb{E}[2^{2X_n}] \le \frac{3}{2}n(n+1) + 1$$

Base case: $n = 0 : \mathbb{E}[2^0] = 1 \leq 1$

Assume the inductive hypothesis for $\mathbb{E}[2^{2X_{n-1}}]$

Now we want to compute the expectation:

$$\mathbb{E}[2^{2X_n}] = \mathbb{E}_{X_{n-1}}\Big[\mathbb{E}_{X_n}[2^{2X_n}]\Big]$$

$$= \sum_{i\geq 0} \Pr[X_{n-1} = i] \cdot \mathbb{E}_{X_n}[2^{2X_n} | X_{n-1} = i]$$

$$= \sum_{i\geq 0} \Pr[X_{n-1} = i] \cdot [2^{-i} \cdot 2^{2(i+1)} + (1 - 2^{-i}) \cdot 2^{2i}]$$

$$= \sum_{i\geq 0} \Pr[X_{n-1} = i][\overset{3}{\cancel{4}} \cdot 2^i + 2^{2i} - \cancel{2^i}]$$

$$= \sum_{i\geq 0} \Pr[X_{n-1} = i] \cdot 2^i \cdot 3 + \sum_{i\geq 0} \Pr[X_{n-1} = i] \cdot 2^{2i}$$

$$= 3 \cdot \underbrace{\mathbb{E}[2^{X_{n-1}}]}_{n,\text{ by Claim 1}} + \underbrace{\mathbb{E}[2^{2X_{n-1}}]}_{\leq \frac{3}{2}n(n-1)+1}$$

$$\leq \frac{3}{2}n(n+1) + 1$$

□

1.4 Chebyshev

By definition of Chebyshev:

$$\Pr[\,|\hat{n} - \mathbb{E}(\hat{n})| > \lambda\,] \;\leq\; \frac{\mathrm{Var}[\hat{n}]}{\lambda^2} \qquad \leq \; \frac{\frac{3}{2}n(n+1) + 1}{\lambda^2} < 0.1$$

It is good enough to have $\lambda = 5n$, since this is an analysis and Chebyshev holds for any bound.

The algorithm does not give a good accuracy, but there is a standard trick to boost the accuracy to get a much better estimate. In particular, we want to do the following:

GOAL: Estimate n up to $\pm\,\epsilon\, n$ for small $\epsilon > 0$.
- Think of epsilon as being 0.1 which is comparable to 10% error.
- "Estimate up to" means $(1 - \epsilon)n \leq \hat{n} \leq (1 + \epsilon)n$
- To reach our goal, we will essentially do a bunch of counters and average them (**Morris+ Algorithm**)

1.5 Morris+ Algorithm

- Use $k =$ TBD counters $X^1, X^2, ..., X^k$.

- Each X^i uses Morris Algorithm and is i.i.d.

- The new estimate will be $\hat{n} = \frac{1}{k}\sum_{i=1}^{k} \hat{n}^i$, where $\hat{n}^i = 2^{X^i} - 1$

1.6 Amplification / Variance Reduction via Repetition

We will do the analysis and see what we need to set k to be such that we get the above goal.

Claim 4. $\mathbb{E}[\hat{n}] = \mathbb{E}[\frac{1}{k}\sum_{i=1}^{k} \hat{n}^i] = n$*, by linearity of expectation.*

Claim 5. $\mathrm{Var}[\hat{n}] = \frac{\mathrm{Var}[\hat{n}^1]}{k}$

Proof.

$$\mathrm{Var}[\hat{n}] = \mathrm{Var}\left[\frac{1}{k} \cdot \sum_{i=1}^{k} \hat{n}^i\right] = \frac{1}{k^2} \cdot \mathrm{Var}\left[\sum_{i=1}^{k} \hat{n}^i\right] = \underbrace{\frac{1}{k^2}\sum_{i=1}^{k} \mathrm{Var}[\hat{n}^i]}_{\text{Due to linearity of variance}}$$

Note that the variance of the sum of variables are independent of each other, because they each correspond to the estimate of a different run of the algorithm, where each run of the algorithm uses i.i.d. randomness. For this reason, we can use linearity of variance.

Proof of **linearity of variance**:

For two independent variables X and Y, their covariance is zero: $Cov(X, Y) = 0$

$$\begin{aligned}
\mathrm{Var}\, X + Y &= \mathbb{E}[((X + Y) - \mathbb{E}[X + Y])^2] \\
&= \mathbb{E}[((X - \mathbb{E}[X]) + (Y - \mathbb{E}[Y]))^2] \\
&= \mathbb{E}[(X - \mathbb{E}[X])^2] + \mathbb{E}[(Y - \mathbb{E}[Y])^2] - 2\mathbb{E}[X - \mathbb{E}[X]]\mathbb{E}[Y - \mathbb{E}[Y]] \\
&= \mathrm{Var}\, X + \mathrm{Var}\, Y - 2\underbrace{Cov(X, Y)}_{=0}
\end{aligned}$$

$$\therefore \quad \mathrm{Var}\, X + Y = \mathrm{Var}\, X + \mathrm{Var}\, Y$$

Set $\lambda = \epsilon n$

By the Chebyshev bound:

$$\Pr[\ |\hat{n} - \mathbb{E}[n]|\ >\ \lambda\ ] \quad \leq \quad \frac{\mathrm{Var}[\hat{n}]}{\lambda^2}$$

$$\leq \quad \underbrace{\frac{\frac{1}{k} \cdot (\frac{3}{2} n(n+1) + 1)}{\epsilon^2 n^2}}_{\text{We want this quantity to be } < 0.1}$$

$$\Longrightarrow \frac{1}{k} \cdot \left(\frac{3}{2} n(n+1) + 1 \right) < 0.1 \cdot \epsilon^2 n^2$$

$$\Longrightarrow \text{ It is enough to have } k > \Omega(\frac{1}{\epsilon^2})\ , \quad \text{where } \Omega \text{ is some large constant}$$

With k as the above, all is satisfied and our goal is now satisfied.

New space: $\mathrm{O}\left(\frac{1}{\epsilon^2} \cdot \lg \lg n\right)$ □

2 Hashing

Hash function: $h : U \to [n]$ (where U is the discrete universe)

The main application we will be looking at is **Dictionary**.

Dictionary: fixed U; preprocess $S \subset U$, $|S| = m$, s.t. given a query $x \in U$, report if $x \in S$.

Possible solutions for this problem:

Sol 0 . Store S (Search time: O(m); Space: $\mathrm{O}(m \cdot \lg m)$)
Sol 0′. Binary Search Tree (Search time: $\mathrm{O}(\lg m)$; Space: $\mathrm{O}(m \cdot \lg m)$)
Sol 0″. Bit Array (Search time: O(1); Space: $|U|$)

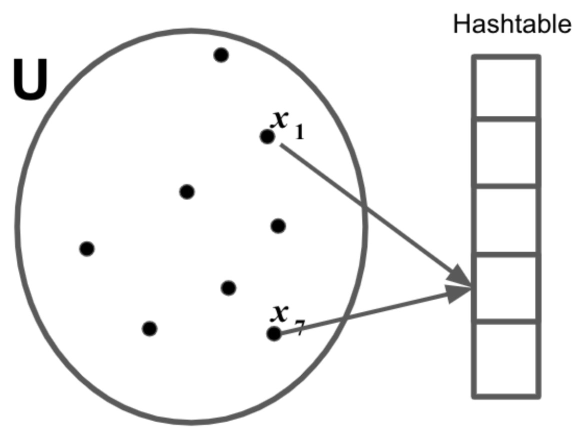
GOAL: Ideally, we would like to combine the best of both worlds and get a deterministic algorithm with both a constant runtime and a space that is linear in the size of S. We will use a hash function to get as close as possible to this goal.

2.1 Solution via a Hash Function

This solution can be thought of a variant of Solution $0''$. . . Our universe is very large, and this is why our space is very large, so how about we reduce the universe size?

Solution: Store an array of size n s.t. cell i stores all items $j \in S$ s.t. $h(j) = i$.

When this hash function is defined, it could be that a few elements from the set S map into the same cell.



In the above illustration, element x^1 and x^7 are mapped into the same cell (called a **collision**).

Possible ways we could store x^1 and x^7:

1. **Hashing with chaining:** store them as a linked list.
2. **Linear probing:** instead of storing x^7 where it was mapped, store it in the next empty cell
3. **Cuckoo hashing**

The main principle is that collisions are bad!