15-451 Algorithms, Spring 2018
Recitation #3 Worksheet

Recap of this week's lectures:

- Hashing: Universal hashing, and constructions.
- Perfect hashing: dictionary lookup in constant worst-case time.
- The Data Streaming model
- Heavy hitters, both without and with deletions.

Hashing: A *universal* hash family H from U to $[m] := \{0, 1, \ldots, m-1\}$ is a set of hash functions $H = \{h_1, h_2, \ldots, h_k\}$ each mapping U to $[m]$, such that for any $a \neq b \in U$, when you pick a random function from H,

$$\mathbf{Pr}[h(a) = h(b)] \leq \frac{1}{m}.$$

Also, a *ℓ-universal* hash family H from U to $[m] := \{0, 1, \ldots, m-1\}$ is a set of hash functions $H = \{h_1, h_2, \ldots, h_k\}$ each mapping U to $[m]$, such that for any *distinct* $a_1, \ldots, a_\ell \in U$, and for any $\alpha_1, \ldots, \alpha_\ell \in [m]$, when you pick a random function from H,

$$\mathbf{Pr}[h(a_1) = \alpha_1 \text{ and } \ldots \text{ and } h(a_\ell) = \alpha_\ell] = \frac{1}{m^\ell}.$$

1. Show that a 2-universal hash family is a universal hash family.

   Solution: We know that for any $a \neq b$ and any $\alpha \in [m]$, $\mathbf{Pr}[h(a) = h(b) = \alpha] = \frac{1}{m^2}$ by definition of 2-universal. So

   $$\mathbf{Pr}[h(a) = h(b)] = \sum_{\alpha \in [m]} \mathbf{Pr}[h(a) = h(b) = \alpha] = \sum_{\alpha \in [m]} \frac{1}{m^2} = \frac{1}{m}.$$

2. Show that a k-universal hash family is a ℓ-universal hash family for any $\ell \leq k$.

   Solution: We show this for $\ell = k - 1$ and then can use induction. Indeed,

   $$\mathbf{Pr}[h(a_1) = \alpha_1 \text{ and } \ldots \text{ and } h(a_\ell) = \alpha_\ell]$$
   $$= \sum_{\alpha_{\ell+1} \in [m]} \mathbf{Pr}[h(a_1) = \alpha_1 \text{ and } \ldots \text{ and } h(a_\ell) = \alpha_\ell \text{ and } h(a_{\ell+1}) = \alpha_{\ell+1}] = \sum_{\alpha_{\ell+1} \in [m]} \frac{1}{m^{\ell+1}} = \frac{1}{m^\ell}.$$

	a	b
h_1	0	0
h_2	1	0

3. Is this hash family from $U = \{a, b\}$ to $\{0, 1\}$ (i.e., $m = 2$) universal? 1-universal? 2-universal?

   Solution: It's universal because a and b collide under only one of the two functions, so they collide with probability $1/2$. It's not 1-universal because b hashes to value 0 with probability 1, whereas it should hash with probability half. And hence it is not ℓ-universal for any $\ell \geq 1$.

4. How about this one: is it universal? 1-universal? 2-universal? 3-universal?

	a	b	c
h_1	0	0	0
h_2	1	0	1
h_3	0	1	1
h_4	1	1	0

   Solution: This is 2-universal (can verify, for each pair of elements, that all 4 possibilities for α, β each occur once) and therefore it is universal. Also, 2-universal implies 1-universal. But it's not 3-universal (e.g., can't get all three elements to simultaneously hash to 1).

Las Vegas and Monte Carlo:

A *Las Vegas* algorithm is a randomized algorithm that always produces the correct answer, but its running time $T(n)$ is a random variable. That is, sometimes it runs faster and sometimes slower, based on its random choices; for instance, quicksort when choosing a random pivot, or treaps. A *Monte Carlo* algorithm is an algorithm with a deterministic running time, but that sometimes doesn't produce the correct answer. For example, you may be familiar with randomized primality testing algorithms, that given a number N will output whether it is prime or not and be correct with probability at least 99/100, say.

1. *Going from LV to MC:* Show that if you have a Las Vegas algorithm with expected running time $\mathbf{E}[T(n)] \leq f(n)$, then you can get a Monte Carlo algorithm with (a) worst-case running time at most $4f(n)$ and (b) probability of success at least $3/4$.

   Solution: Consider the algorithm: run the LV algorithm for $4f(n)$ steps, and if it has not stopped yet, just abort it. Trivially, the running time is now upper bounded by $4f(n)$. What is the chance it was aborted?

   This equals Pr[the algorithm did not stop by $4\times$ its expected run-time] $\leq 1/4$, by Markov's inequality.

Gory details: What's the random variable? X = run-time of the algo. Clearly a non-negative r.v. Also, we are given that $\mathbf{E}[X] \leq f(n)$. Therefore $\mathbf{Pr}[X > 4f(n)] \leq \mathbf{Pr}[X > 4\mathbf{E}[X]] \leq 1/4$.

2. *Going from MC to LV:* Suppose you have a MC algorithm `Algo` for a problem (e.g., think of factoring an n-bit number into a product of primes) with running time at most $f(n)$, that is correct with probablity p. Moreover, you have a "checking" algo `Check` that runs in time $g(n)$ and checks whether a given output is a correct solution for this problem. E.g., for factoring, you could just multiply the outputs together to make sure you get back the input, and also verify the outputs are indeed prime numbers by running a fast primality checker.

   Use these to get a LV algorithm that runs in expected time $\frac{1}{p}(f(n) + g(n))$.

   Solution: The simplest idea works: run `Algo`, then run `Check` see if the output is correct. If not, repeat the process (with fresh randomness, so that the outcome is independent of the previous rounds). Each "round" of the algorithm takes $(f(n) + g(n))$ time. What is the expected number of rounds we will take before we stop?

   Since each run of the algorithm succeeds (independently of the past runs) with probabability p, so it's as though we are flipping a coin with bias (probability of heads) at least p and want to know the expected number of flips before we see a heads. This expected number is $1/p$.

Streaming.

Sampling: Given a number k, you want to maintain a random sample of size k from the stream. I.e., for each $n \geq k$, the set you have at time n should be a random subset of the prefix $a_{[1:n]}$, each of the $\binom{n}{k}$ subsets of size k from this prefix should be equally likely.

1. For $k = 1$, show that the algorithm: pick the first element. When faced with the n^{th} element, with prob. $1/n$ discard the element in your hand and pick the new element, and with prob. $1 - 1/n$ keep the element in hand.

   Solution: We claim that for any element from $e \in a_{[1..n]}$, we have e in our hand after step n w.p. $1/n$. The proof is inductive. The base case is easy. Consider the case for element e at some time n.

   - If $e \in a_{[1..n-1]}$, then we have it at time $n-1$ w.p. $\frac{1}{n-1}$. Now at time n, the chance we don't discard it is $\frac{n-1}{n}$, so multiplying we get $\frac{1}{n}$.
   - If $e = a_n$, then the chance we picked it at time n is $1/n$.

2. Give an algorithm for general k. (What would you do when faced with the n^{th} element? With what probability should you pick this element? Which element should you drop?)

Solution: The algorithm: pick the n^{th} item with probability k/n, and drop a uniformly random item from the current set.

Claim: for any k-sized set S from $e \in a_{[1..n]}$, we have S after step n w.p. $\frac{1}{\binom{n}{k}}$. The proof is again inductive. The base case for $n = k$ is easy. Consider the case for element e at some time $n > k$.

- If $S \subseteq a_{[1..n-1]}$, then we have it at time $n-1$ w.p. $\frac{1}{\binom{n-1}{k}}$. Now at time n, the chance we don't destroy it is $\frac{n-k}{n}$, so multiplying we get $\frac{k!(n-1-k)!}{(n-1)!} \cdot \frac{n-k}{n} = \frac{1}{\binom{n}{k}}$.
- If $a_n \in S$, then look at $S_i = (S \setminus \{a_n\}) \cup \{a_i\}$ where $a_i \notin S$. There are $n-k$ such sets S_i. For each of the $n-k$ sets, inductively the chance we have it at time $n-1$ is $\frac{1}{\binom{n-1}{k}}$, then we have to drop a_i (w.p. $1/k$), and add a_n (w.p. $1/n$). Overall we get

$$(n-k) \times \frac{k!(n-1-k)!}{(n-1)!} \times \frac{1}{k} \times \frac{k}{n} = \frac{1}{\binom{n}{k}}.$$

This idea is called *Reservoir Sampling*.

Missing Numbers: Suppose I give you a stream of $n-1$ elements, which contains all the numbers from 1 thru n *except one of them.* (The numbers *do not* appear in sorted order.) Clearly you can figure out the missing number by storing all $n-1$ numbers and looking for the missing number. How can you output the missing number with only $O(\log n)$ space? What if there are two missing numbers: can you again use only $O(\log n)$ space?

Solution: For one missing number, you can store the sum of all numbers seen so far. Then finally subtract that from $\frac{n(n+1)}{2}$ to get the missing number. For two, you can store, e.g., the sum, and the sum of their squares. Then you'll know $a+b$ and a^2+b^2, and can solve for the answer.

You could also have stored the sum and product of the numbers seen, but that requires more space. The product of the numbers could be as large as $\Omega(n!)$ which requires $\Omega(n \log n)$ bits to store.