Probability and Hashing

Reminder: the first Quarterly Exam is next Monday in class. 30 minutes long. Rules on Piazza.

From Monte Carlo to Las Vegas: Recall from Reci #3 that 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; e.g., 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. E.g., randomized primality testing algorithms, given a number $N$, will output whether it is prime or not and be correct with probability at least 99/100, say. In Reci #3 you saw how to convert a LV algorithm into an MC one. Today you will see the opposite direction.

Suppose you have a MC algorithm $\text{Algo}$ 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 probability $p$. Also, you have a “checking” algo $\text{Check}$ 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 $\text{Algo}$, then run $\text{Check}$ 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 probability $p$, 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 $\frac{1}{p}$.

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 $H$,

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

From HW#2, a $\ell$-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 $H$,

$$\Pr[h(a_1) = \alpha_1 \text{ and } \ldots \text{ and } h(a_\ell) = \alpha_\ell] = \frac{1}{m^\ell}.$$
1. Show that a 2-universal hash family is a universal hash family.

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

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

2. Consider this hash family from \( U = \{a, b\} \) to \( \{0, 1\} \) (i.e., \( m = 2 \)).

\[
\begin{array}{c|cc}
  & a & b \\
\hline
h_1 & 0 & 0 \\
h_2 & 1 & 0 \\
\end{array}
\]

Is it a universal hash family? 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 2-universal because of the 4 possibilities for \( \alpha, \beta \), two occur with probability \( 1/2 \) and two occur with probability 0.

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

\[
\begin{array}{c|ccc}
  & a & b & c \\
\hline
h_1 & 0 & 0 & 0 \\
h_2 & 1 & 0 & 1 \\
h_3 & 0 & 1 & 1 \\
h_4 & 1 & 1 & 0 \\
\end{array}
\]

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

A Streaming Puzzle: 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.