6.856 — Randomized Algorithms

David Karger

Handout #14, October 20, 2002 — Homework 6 Solutions

M. R. refers to this text:
Motwani, Rajeez, and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge: Cambridge University Press, 1995.

Problem 1 Consider a perfect hash family $\mathcal{F}$ of size N. For some $f \in \mathcal{F}$, let a_i for $1 \le i \le n$ be the number of elements that f maps to the number i. The number of sets of size n for which f is perfect is exactly $\prod_{i=1}^n a_i$. This is because for each i such a set has to contain a number that maps to i, and there are a_i choices for this number.

Since $\sum a_i = m$, and a product where the factors have a constant sum becomes maximized when all terms are equal, we have

$$\prod_{i=1}^n a_i \le \left(\frac{m}{n}\right)^n.$$

Since there are $\binom{m}{n}$ total subsets for which we need a perfect function, and each function is perfect for at most $(m/n)^n$ subsets, we must have

$$N \cdot (m/n)^n \ge \binom{m}{n} \iff N \ge (n/m)^n \binom{m}{n}.$$

Using Stirling's approximation, this yields

$$\begin{aligned} N &\ge \frac{n^n}{m^n} \frac{\sqrt{2\pi m}\left(\frac{m}{e}\right)^m}{2\pi\sqrt{n(m-n)}\left(\frac{n}{e}\right)^n\left(\frac{m-n}{e}\right)^{m-n}} \cdot \Omega(1) \\ &= \sqrt{\frac{m}{n(m-n)}}\left(\frac{m}{m-n}\right)^{m-n} \cdot \Omega(1) \\ &= \sqrt{\frac{m}{n(m-n)}}\left(1+\frac{n}{m-n}\right)^{m-n} \cdot \Omega(1) \\ &= \sqrt{\frac{m}{n(m-n)}} e^n \cdot \Omega(1) = e^{\Omega(n)}, \end{aligned}$$

so can there can be no perfect hash family for the choices of m given in the problem statement.

Problem 2 MR 8.20.

Choose a hash random function from a 2-universal hash family and hash each integer in the first set into a table of size $O(m)$. At each entry in the table, keep count of the number of occurrences of each integer that hashes there. Also, keep the list of counts sorted (this is easy, since we are allowed to spend time proportional to the number of entries in the bucket to insert a new item). As we argued in class, the expected time to build this table is $O(m)$ (assuming unit-cost multiplications). Note that we need to keep counts, rather than lists, of identical integers to ensure that buckets do not grow arbitrarily large in the presence of many identical integers. Now, do the same for the second multiset, and compare the resulting hash tables for equality. This is a trivial linear time operation since the hash table buckets contain sorted lists. This gives a Las Vegas algorithm with expected run time $O(n)$ considering $m \leq n$.

A common mistake in this algorithm is to link all the (copies of) items that land in a bucket instead of keeping just one counter of the number of occurrences of each distinct item. If there are $n/2$ copies of the same item, the mistaken approach would frequently search through and create lists of length $n/2$, resulting in too high a time bound.

An alternative approach does away with the linked lists. Use the random hash function to hash the two sets into two tables, keeping a counter of how many elements in total hash to each (without separately tracking the distinct elements that hashed into a particular bucket). Then compare the two tables and the counters at each bucket in $O(n)$ time. If the hash tables are different in any counter, then the sets are different. Alternatively, we can use just one table, incrementing bucket counters as we hash the first set into it and decrementing bucket counters as we hash the second set. We can declare equality if all the bucket counters end up at 0.

This approach is Monte Carlo (there is a chance of different elements canceling each other out by collisions) but can be shown to work as follows. Clearly the algorithm never reports different when the two sets are the same. So suppose the two sets differ. Then there is an element x which appears a different number of times in the two sets. Assuming our hash table has b buckets, the expected number of elements that hash to the same bucket as x is n/b. If we set $b = n/2$ this is less than $1/2$. It follows that with probability at least $1/2$, no other item maps to the same bucket as x. If this happens, then there is no way to cancel out the difference in the number of x's coming from the two sets, so we will detect that they are different. To summarize, if we use a bucket size of $2n$ then we have a probability $1/2$ of detecting any discrepancy in the sets. This is sufficient for a Monte Carlo algorithm (with one sided error).

Problem 3 MR 7.4.

Given a multiset S with integers $w_1, \ldots, w_m$ all less than n, consider the polynomial

$$P(S) = \prod_{w_i \in S} (x - w_i).$$

Clearly, if two multisets are different, then their polynomials have different roots (or at least different root multiplicities) and are therefore different. Their difference is a degree m polynomial and therefore has at most m distinct roots over the integers. Thus, if we choose a random integer in the interval $[0, 2m]$ there is at least a $1/2$ chance that it will not be a root. In fact, if we choose our integer in the interval $[0,(2m)^2]$ then the probability that we choose a root is $O(1/m)$, so we get the right answer with high probability.

Of course, evaluating over the integers is very expensive, since the resulting numbers can be huge. However, suppose that we choose some random prime p from a set of size $cm \log n$ (for some constant c) and work modulo p. For example, we could choose p from the interval $[0, cm \log^2 n]$.

Whatever random number we have chosen to substitute for x in the polynomial, the resulting value of the polynomial has magnitude $O(n^m)$ and therefore has at most $O(m \log n)$ distinct prime factors. We therefore have (by choosing c appropriately) a probability $1/2$ of picking a prime p that is not a factor. If so, then when we evaluate the polynomial modulo p (which requires $O(m)$ additions and multiplications of numbers of magnitude at most p) we get a nonzero answer (indicating a difference) with probability at least $1/2$.

If we treat products as unit-cost operations, this $O(m)$ time fingerprinting scheme is clearly faster than sorting. However, if we recall that multiplication of b-bit numbers takes $O(b \log b)$ time in the log-cost RAM model, we find that the overall cost of computing the signature rises to $O(m \log m)$, the same as sorting. So even though the Monte Carlo nature of the signature scheme appears to make it the loser, it still has the advantage of a greatly reduced communication cost, since only the fingerprint has to be transmitted.

Problem 4

(a) Suppose the string has length m and the intended pattern size is k. As discussed in the text, in linear time we can generate a fingerprint for every length-k substring. Using a random 2-universal hash function, put all these keys (= fingerprints) into a size-m hash table such that the expected time to lookup a particular fingerprint is $O(1)$. It is now easy to store with each key the number of times it occurs. Now, given any pattern, use the same textbook scheme to generate its fingerprint in $O(k)$ time. Hash it into the table and see if that fingerprint is present. If so, read off the corresponding number of occurrences. Assuming the uniqueness of fingerprints (which happens with high probability) the answer we give is correct.

It is a mistake to directly hash all length-k substrings using a 2-universal hash function. If finding $h(x)$, (h is the hash function, x is a k-length substring) takes $O(k)$ time, then the total preprocessing time is $O(k(n-k+1))$, which is not necessarily linear, i.e. k could be $\sqrt{n}$. Also storing the substrings themselves as keys in the hash table uses more than linear storage and makes comparisons expensive.

Note that the construction of the hash table is in *expected* linear time. With a little more work (which was not required to solve the problem), we can actually do this in linear time with high probability. We use the following lemma:

Lemma 1 *If we hash k distinct items into a hash table of size n $(k \le n)$ using a pairwise independent hash function, then with high probability $(\ge 1 - 1/\sqrt{n})$, every hash bucket contains at most $k^{3/4}$ items.*

Proof: The expected number of different fingerprints mapping to the same hash value are $\mu = k/n < 1$, and the variance of that number is $\sigma = \mu(1 - 1/n) < 1$. Since the hash values are pairwise independent, we know by Chebyshev that the probability that some particular hash value is used more than $k^{3/4}$ times is at most $1/k^{3/2}$. Applying a union bound shows that with probability $1 - 1/\sqrt{k} > 1 - 1/\sqrt{n}$, no hash value is used more than $k^{3/4}$ times. This probability can be boosted to any polynomial by repeating the process a constant number of times.

Using this lemma, we can do the following: we hash our n elements into a hash table of size n, then every hash bucket into a new table of size n, and every hash bucket in that table again in a hash table of size n. In every hashing operation we just prepend items to their hash bucket's linked list in constant time. While this means that duplicate fingerprints (corresponding to multiple occurrences of the same pattern) are kept around, it guarantees that the whole process finished in linear time.

Also notice that since we can do the hashing in sequence, we actually only need linear space (3 hash tables at a time), and the whole thing can be done in linear time. In the third hash table, every bucket contains at most $(n^{3/4})^3 = n^{27/64} < \sqrt{n}$ distinct fingerprints.

Now if we hash each bucket one more time, i.e. $\sqrt{n}$ times to n buckets, we have for the expected number of distinct fingerprints per bucket $\mu = 1/\sqrt{n}$, and variance $\sigma = (1 - 1/n)/\sqrt{n} < 1/\sqrt{n}$. Applying Chebyshev yields that the probability that a bucket contains more than one fingerprint is less than $1/n$, i.e. happens with high probability. But now we can count how often the fingerprint occurs just by a linear pass through the linked list in the bucket. This yields the result in linear time with high probability.

(b) Modify the hashing scheme above to use a perfect hash function that guarantees $O(1)$ lookup time for every fingerprint in the text. Instead of storing a count of the number of occurrences of a given string, store the locations. Also during the preprocessing, make sure that each fingerprint in the text is unique (if it is not, try again until it works). Store the unique string associated with each fingerprint. Now, when we get an input pattern, generate its fingerprint and look it up in the perfect hash table. If it does not exist, we know the string is not present. If the fingerprint is present, we have time to verify that the input pattern is the same as the text pattern that generated the string. Then we can read off the locations. Since we can find a perfect hash function in worst-case polynomial time, we can preprocess the input in worst-case polynomial time.

Problem 5 MR 8.28

See the following paper (also available online at `http://www.acm.org/dl`):

Michael L. Fredman, János Komlós and Endre Szemerédi, Storing a Sparse Table with $O(1)$ Worst Case Access Time, *Journal of the ACM*, 31(3):538–544, July 1984.