Problem Set 3

Randomized Algorithms

Due Tuesday, October 17

1. A *minimal perfect* hash function for a set S of size n is one that maps S to $[n]$ with no collisions. In class, we showed how to take S and construct a minimal perfect hash function for S that can be evaluated in constant time. The construction took expected $O(n)$ time and the resulting function took $O(n)$ words to store.

   Show that this last condition cannot be significantly improved upon. In particular, show that any procedure for storing a minimal perfect hash function requires at least $\Omega(n)$ bits for some S of size n. Assume the universe size U is polynomial in n. **Hint:** show that any particular function h is perfect for at most a $1/2^{\Omega(n)}$ fraction of the possible sets S.

2. [Karger.] Bloom filters can be used to estimate the difference between two sets. Suppose that you have sets X and Y, each with m elements, and with r elements in common. Create an n-bit Bloom filter for each, using the same k hash functions. Determine the expected number of bits where the two Bloom filters differ, as a function of m, n, k, and r. Explain how this could be used as a technique for estimating r.

3. [MR 4.9]. Consider the following randomized variant of the bit fixing algorithm. Each packet randomly orders the bit positions in the label of its source and then corrects the mismatched bits in that order. Show that there is a permutation for which, with high probability, this algorithm uses $2^{\Omega(n)}$ steps to route.

4. [MR7.2]. Two rooted trees T_1 and T_2 are said to be isomorphic if there exists a one to one mapping f from the nodes of T_1 to those of T_2 satisfying the following condition: v is a child of w in T_1 if and only if $f(v)$ is a child of $f(w)$ in T_2. Observe that no ordering is assumed on the children of any vertex. Devise an efficient randomized algorithm for testing the isomorphism of rooted trees and analyze its performance.

Hint: Recursively associate a polynomial P_v with each vertex v in a tree T.

5. The Python programming language uses hash tables (or "dictionaries") internally in many places. Until 2012, however, the hash function was not randomized: keys that collided in one Python program would do so for every other program. To avoid denial of service attacks, Python implemented hash randomization—but there was an issue with the initial implementation. Also, in python 2, hash randomization is still not the default: one must enable it with the `-R` flag.

   Find a 64-bit machine with both Python 2.7 and Python 3.5; one is available at `linux.cs.utexas.edu`.

   (a) First, let's look at the behavior of `hash`("*a*") − `hash`("*b*") over $n = 2000$ different initializations. If `hash` were pairwise independent over the range (64-bit integers, on a 64-bit machine), how many times should we see the same value appear?

   (b) How many times *do* we see the same value appear, for three different instantiations of python: (I) no randomization (`python2`), (II) python 2's hash randomization (`python2 -R`), and (III) python 3's hash randomization (`python3`)?

   (c) What might be going on here? Roughly how many "different" hash functions does this suggest that each version has?

   (d) The above suggests that Python 2's hash randomization is broken, but does not yet demonstrate a practical issue. Let's show that large collision probabilities happen. Observe that the strings "8177111679642921702" and "6826764379386829346" hash to the same value in non-randomized python 2.

      Check how often those two keys hash to the same value under `python2 -R`. What fraction of runs do they collide? Run it enough times to estimate the fraction to within 20% multiplicative error, with good probability.

      How could an attacker use this behavior to denial of service attack a website?

   (e) (Optional) Find other pairs of inputs that collide.