# Problem Set 5

## Randomized Algorithms

## Due Friday, October 4

1. In class, we showed that cuckoo hashing achieves worst case constant time lookups and expected constant time insertion/deletion, with $O(n)$ space to store $n$ items. Show how to get the same guarantees, but using only $(1+\epsilon)n$ space for a small constant $\epsilon$. For this problem, assume that you have access to perfectly random hash functions. **Hint:** Use the following ideas:

   - Probing more than twice in a table increases the chances of finding an empty cell.
   - If after some probes you fail to find an empty cell, move the failed item into an "overflow" table that uses cuckoo hashing.

2. For a hash family $\mathcal{H}$ from $[U]$ to $[n]$, and a set of items $S \subset [U]$, let $X(\mathcal{H}, S)$ be the random variable denoting the load in the first bin:

   $$X := |\{i \in S \mid h(i) = 1\}|$$

   as a distribution over $h \in \mathcal{H}$. Further, let $f(\mathcal{H}, S)$ denote the expected max load in any bin:

   $$f(\mathcal{H}, S) := \mathop{\mathbb{E}}_{h \in \mathcal{H}} \max_{j \in [n]} |\{i \in S \mid h(i) = j\}|.$$

   (a) For any $t \geq 1$, and for any $k$-wise independent hash family $\mathcal{H}$ with $k = O(1)$, and any set $S$ with $|S| = n$, show that

   $$\Pr[X \geq t] \lesssim 1/t^k.$$

   **Hints:** bound $\mathbb{E}[(X - \mathbb{E}[X])^k]$. (Rot13) Svefg svther vg bhg jura u vf shyyl vaqrcraqrag; lbhe cebbs zbfg yvxryl bayl arrqf x-jvfr vaqrcraqrapr.

(b) Use the above to show that for any $k$-wise independent family $\mathcal{H}$, $k = O(1)$, that

$$f(\mathcal{H}, S) \lesssim n^{1/k}$$

for any $S$ with $|S| = n$.

(c) Consider the special case $U = n$, $S = [n]$. Show that there exists a universal hash family $\mathcal{H}$ such that

$$f(\mathcal{H}, S) \gtrsim \sqrt{n}.$$

(Optional.) Extend your construction to be pairwise independent, not just universal.

**Hint:** You may make $\mathcal{H}$ extremely contrived and/or large, as long as it satisfies the constraint of univerality/pairwise independence.

3. 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 or later; 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.