Problem Set 1

Randomized Algorithms

Due Wednesday, September 16

1. Consider an optimization problem $\texttt{Opt}(x) : \mathcal{X} \to \{0, 1, \ldots, n-1\}$ that associates inputs from some domain with outputs (e.g., min cut is such a problem).

   Suppose that we have a BPP algorithm $\mathcal{A}$ that takes T time to answer binary queries of whether $\texttt{Opt}(x)$ is less than some threshold. That is, it answers queries of the form $\texttt{Less}(x, k) := (\texttt{Opt}(x) < k)$, with error probability at most $1/4$.

   We saw in class that we can then solve $\texttt{Opt}(x)$ in $O(T \log n \log\log n)$ time with $3/4$ probability, by amplifying the probability and doing binary search. The goal of this problem is to improve this.

   (a) Consider the function $\texttt{RobustBisect}(x, L, H)$ that has three outputs:

   - LOW if $L \leq \texttt{Opt}(x) < \frac{L+H}{2}$.
   - HIGH if $\frac{L+H}{2} \leq \texttt{Opt}(x) < H$.
   - OUTOFRANGE if $\texttt{Opt}(x) \notin [L, H)$.

   Use $\mathcal{A}$ to construct a randomized algorithm $\mathcal{B}$ to solve `RobustBisect` with $3/4$ probability and $O(T)$ time.

   (b) Construct a strategy for calling $\mathcal{B}$ such that, once $\mathcal{B}$ is correct at least $\log n$ more times than it is incorrect, you can output $\texttt{Opt}(x)$ correctly.

   (c) Conclude that one can solve $\texttt{Opt}(x)$ in $O(T \log n)$ time with high probability (which means $1 - 1/n^c$ probability for an arbitrarily large constant c).

2. [Karger] Suppose we have access to a source of unbiased random bits. This problem looks at constructing biased coins or dice from this source.

   (a) Show how to construct a biased coin, which is 1 with probability p and 0 otherwise, using $O(1)$ random bits in expectation. [Hint: First show how to construct a biased coin using an arbitrary number of random bits. Then show that the expected number of bits examined is small.]

   (b) Show how to sample from $[n]$, with probabilities $p_1, \ldots, p_n$, using $O(\log n)$ random bits in expectation.

   (c) Show that the "in expectation" caveat is necessary: for example, one cannot sample uniformly over $\{1, 2, 3\}$ using $O(1)$ bits in the worst case.

3. [MR 1.8]. Consider adapting the min-cut algorithm of Section 1.1 to the problem of finding an s–t min-cut in an undirected graph. In this problem, we are given an undirected graph G together with two distinguished vertices s and t. An s–t min-cut is a set of edges whose removal disconnects s from t; we seek an edge set of minimum cardinality. As the algorithm proceeds, the vertex s may get amalgamated into a new vertex as the result of an edge being contracted; we call this vertex the s-vertex (initially s itself). Similarly, we have a t-vertex. As we run the contraction algorithm, we ensure that we never contract an edge between the s-vertex and the t-vertex.

   (a) Show that there are graphs (not multi-graphs) in which the probability that this algorithm finds an s–t min-cut is exponentially small.

   (b) How large can the number of s–t min-cuts in an instance be?

4. [MR 2.3]. Consider a uniform rooted tree of height h (every leaf is at distance h from the root). The root, as well as any internal node, has 3 children. Each leaf has a boolean value associated with it. Each internal node returns the value returned by the majority of its children. The evaluation problem consists of determining the value of the root; at each step, an algorithm can choose one leaf whose value it wishes to read.

(a) Show that for any deterministic algorithm, there is an instance (a set of boolean values for the leaves) that forces it to read all $n = 3^h$ leaves.

(b) Show that there is a nondeterministic algorithm can determine the value of the tree by reading at most $n^{\log_3 2}$ leaves. In other words, prove that one can present a set of this many leaves from which the tree value can be determined.

(c) Consider the recursive randomized algorithm that evaluates two subtrees of the root chosen at random. If the values returned disagree, it proceeds to evaluate the third sub-tree. Show the expected number of leaves read by the algorithm on any instance is at most $n^{0.9}$.

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 3.4; 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`)?

   If you have trouble coding this on your own, the following snippet lets you get the answer:

```
for i in `seq 1 2000`; do
  python2 -R -c 'print((hash("a")-hash("b")))';
done | sort | uniq -c | awk '{print $1}' | sort -n | uniq -c
```

(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.