Problem Set 2	**6.857**	Problem 1 Rainbow Tables
Ben Bitdiddle, Alyssa P. Hacker		April 11, 2014

(a) We desire a pseudorandom mapping from $\{0,1\}^{256}$ to S. First, we note that S is *efficiently enumerable*. In particular, S follows standard lexicographic order, where each of the 62 possible characters follows the ordering $0...1 \leftarrow A-Z \leftarrow a-z$. Then, of the 62^{10} possible values in S, the ith value is easily attainable by computing the base 62 representation of $i-1$. Now, we want a psuedorandom reduction function, R, which maps 2^{256} possible values to 62^{10}. One possible function is simply $R(hv) = sha256(hv)\%62^{10}$. In other words, we use a hash function on the item from $\{0,1\}^{256}$ to map to another item from $\{0,1\}^{256}$ and then take the value mod 62^{10}. This function is very nearly uniform, since $\frac{2^{256}}{62^{10}} \geq 10^{59}$, assuming the the hash function is sufficiently random.

(b) The general idea is that given some hash hv, we can run it through a similar chain process k times. If any of the intermediate hashes match the last hash of some chain in the table, we obtain the corresponding start of the chain s. We start with s, and walk through the chain again, until we reach some segment $hash(pl) = hv$, where pl is our desired plain text. Given that each iteration of hashing or reducing takes $O(1)$ time, and that look up takes $O(1)$ time, we have an $O(k)$ algorithm for inverting any hash that occurs in some chain.

(c) We can use the scheme from before, the table can be represented as a hashtable or sorted, to allow for lookup via binary search. Then for every query hash, hv, we repeat the process from part B, but lookups will take $O(\frac{|S|}{k})$ or $O(1)$ depending on whether the table is implemented as a sorted list of entries or a hashtable, respectively. In practice, this is not always true. First, hv might be part of a chain that shares an endpoint with another chain, but is not present in the table. Additionally, chains can merge on collision. A query might need to search through all such chains that share the endpoint before finding the chain that contains hv.

One might guess that each chain may store k unique passwords, and that therefore $\frac{|S|}{k}$ chains would store $|S|$ distinct passwords. In practice, this is not true. Given $\frac{|S|}{k}$ chains of length k, the number of distinct passwords will be much lower than $|S|$ with high probability. The issue is the presence of collisions. For example, let's consider the optimistic situation where the first $\frac{|S|}{4k}$ chains all contained completely unique passwords. In other words, one fourth of the password space has been successfully encoded without collision. Now, we want to lower bound the expected number of lost passwords. Consider the remaining $\frac{3|S|}{4k} = t$ rows. For each row, i we analyze the probability that one of the first $\frac{k}{2}$ elements in chain i collides with some element in the first half of a previous chain. In particular, if some collision occurs in the first half of chain i with an element in the first half of a preexisting chain, then at least $\frac{k}{2}$ passwords in chain i are lost by merging. It's easy to see this is a lowerbound on the losses, since this case only represents a subset of the cases where passwords are lost.

$E[\text{lost passwords in chain } i] \geq \frac{k}{2} Pr(\text{element in first half of chain } i$
collides with element in the first half of existing chain) $\geq \frac{k}{2}(1-\frac{1}{4\cdot 2})^{\frac{k}{2}}$. This follows from

the fact that at least $\frac{|S|}{4}$ unique passwords exist already and (naturally) half are in the first half of their chains. There are $\frac{k}{2}$ elements in the first half of chain i, and the probability that at least one of them collides is the complement of the probability that none of them do.

Finally, the total number of expected passwords lost in the remaining chains will be:

$$\sum_{i}^{\frac{3|S|}{k}} E[\text{lost passwords in chain } i] \geq \frac{3|S|}{4k}\frac{k}{2}(1-\frac{1}{8})^{\frac{k}{2}}$$

Given $k \geq 1000$, which is highly reasonable, since otherwise the table would need at least $8.39 \cdot 10^1 4$ entries, the above formula yields that at least $\frac{3|S|}{8}$ passwords will not be unique due to merges. This follows from $\frac{3|S|}{8} \geq \frac{3|S|}{4}$, since we assumed that we had at least one fourth of all passwords in the table.

(d) Using a family of pseudorandom functions f_i, for each step in the chain greatly decreases the probability of merges. In particular, for a merge to occur, the collision must occur on the same step j of the chain. If you have length l chains, then any given collision has roughly a $\frac{1}{l}$ probability of causing a merge. With sufficiently large l (l should be quite large, otherwise the table's size will be unrealistic) using a family of pseudorandom functions significantly decreases the number of merges, and therefore increases the number of distinct passwords.

(e) One defense against rainbow tables is the use of salts. Instead of storing a passwords as $hash(pw)$, Ben would store it as $hash(pw + salt)$ where salt is a randomly generated string of greater than 48 bits. The salt can be stored in plainview, with the password hashes. The additional defense comes from the fact that Eve would need a rainbow table for each salt in order to obtain the passwords for every password in the database.