PSG College of Technology
Department of Applied Mathematics and Computational Sciences
MSc Theoretical Computer Science – VI Semester
15XT68 -Security in Computing Lab
Problem Sheet 4

Implement the password cracking module using the rainbow table as illustrated.

Each system should run your password through a hash function before it is stored. The hash function creates a *hash value* which is stored in the database. When you enter your password later, it is hashed with the same function and the results are compared: If the results are same, it means you must have entered the correct password. The hash function is meant to be very hard to *reverse*.

Rainbow Tables

A *rainbow table* is a compromise between hashing every possible password until we find a match (which takes too long) and storing every possible password/hash pair (which would require too big a file).

To construct a rainbow table, we need the hash function that the system uses, as well as a *reduce* function that maps a hash value to an arbitrary password. Note that the reduce function does **not** reverse the hash, it just provides a deterministic way to transform a hash value to *some* legal password.

If we start with some randomly generated passwords (the "Password 0" column in the table), we can repeatedly apply the hash and reduce functions to get several *chains* of values:

A Small Rainbow Table

Password 0	Hash 0	Password 1	Hash 1	Password 2	Hash 2
dccdecee	839310862	daecbecc	-1003698034	cdbdaddb	1802158710
cdeccaed	-723447964	eebdbbcb	504323679	abdcecae	-716915723
acbcaeec	1384205608	ceadeebd	-2138650808	abbddcdc	1305042273
eeeeaebd	537636003	bddcdaad	1446259833	cadaddbd	1629794057
ccdccbeb	-1131047593	ccedebbc	170163192	dacbacdc	-1862546954

To generate this table, we used our hash function to transform "Password *n*" into "Hash *n*", and the reduce function to transform "Hash *n*" into "Password *n*+1".

Instead of storing the whole table, we will only need to **store the first and last columns**. If we have those values, we can reconstruct the rest of the table by using the hash/reduce functions as necessary.

Cracking a Password with a Rainbow Table

Realistically, rainbow tables would be several orders of magnitude larger than the above. They are large enough that (1) you would usually download them, not calculate them and (2) are "wide" enough that storing only the first and last column uses *much* less space than storing everything in the table.

So now, imagine that you can only see the “Password 0” and “Hash 2” columns of the above table (since that's what is stored). You are trying to crack a hashed password value 1446259833.

The part of the table that is stored

Password 0	Hash 2
dccdecee	1802158710
cdeccaed	-716915723
acbcaeec	1305042273
eeeeaebd	1629794057
ccdccbeb	-1862546954

The hash value you're looking for isn't there, but by applying the reduce and hash functions (once) to the has value, you will get the new hash value 1629794057: the password you're looking for is almost certainly somewhere in fourth row of the table.

Once you realize that, you can look back at the initial password for row four, “eeeeaebd”. From there, it's just a matter of recalculating the chain until you find the value you're looking for: the password “bddcdaad” led to the hash value 1446259833. You have successfully cracked a password and can log into the system.

For this assignment, we will restrict ourselves to a limited set of passwords. We will say that all passwords:

- have exactly the same length (e.g. for length 4, "abcd" is will be a valid password, but "abc" will not).
- contain only lowercase letters (and usually only the first *n* letters of the alphabet, depending how hard we want to make the problem).
- The tables above were generated with 5 letters (a–e) and length 8