Rainbow Tables with Haskell

Introduction

The Problem

Think of all of the systems that know some password for you. Of course, those systems have to somehow store the password so you can log in later. But, it is insecure to store your password in plaintext (i.e. store the string “myPassword” directly in the database) since if someone can view the database, they can see all of the passwords.

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: same results mean you must have entered the correct password. [There's actually a bunch more to safely storing passwords. That is to say, what we do below shouldn't work in practice.]

The hash function is meant to be very hard to reverse. That is, even if somebody has access to the password database, they can't practically convert the hash values back to the passwords they need to enter to log in.

Herein lies the problem: the ne'er-do-wells of the world would like to be able to reverse hash functions and figure out the corresponding passwords.

Rainbow Tables

For more info: How rainbow tables work and Rainbow tables in Wikipedia.

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 “PW 0” column in the table), we can repeatedly apply the hash and reduce functions to get several chains of values:

PW 0	Hash 0	PW 1	Hash 1	PW 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 “PW (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).

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

You are trying to crack a hashed password value 1446259833.

The hash value you're looking for isn't in the table, but by applying the reduce and hash functions (once) to the hash 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.

Restrictions

The computation time needed to create a genuinely useful rainbow table is unrealistic for this assignment: there's a reason rainbow tables are usually downloaded, not computed. With a restricted password set, we should be able to get somewhere, but the real power of a rainbow table comes when you put days of processor time into creating the table, and then distribute it to all of your cracker friends. We should get far enough to see that once the table has been created, it can be applied to a hash value relatively quickly.

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"` 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. We will continue to use those values in the first examples below.

Basically: the techniques here are somewhat realistic, but the numbers aren't.

Parameters

We need to define some constants that represent the parameters of the system. For testing (and to get results like my examples), I'd suggest something like this:

```
pwLength, nLetters, width, height :: Int
filename :: FilePath
pwLength = 8            -- length of each password
nLetters = 5            -- number of letters to use in passwords: 5 -> a-e
width = 40              -- length of each chain in the table
height = 1000           -- number of "rows" in the table
filename = "table.txt"  -- filename to store the table
```

When testing your assignment, **we will change** these values, so make sure you don't assume they are always as above.

Hashing & Reducing

The first things we need for rainbow tables are functions that implement the two basic operations:

- `pwHash`: convert a password (string) to a hash value.
- `pwReduce`: convert a hash value back to a possible password.

The `pwHash` function would typically be implied by the passwords we are trying to crack: we need to use the same hash function as the system that stored the passwords. For this assignment we will simply use a convenient hash function that is in the Haskell standard library. The `pwHash` function is provided in the `RainbowAssign` module. It maps strings to a (signed) 32-bit integer, so has type `pwHash :: Passwd -> Hash`.

We also need the `pwReduce` function to map a hash back into a possible password.

A good `pwReduce` function shouldn't produce too many collisions. That is, if `h1/=h2`, it should be very likely that `pwReduce h1 /= pwReduce h2`. (We can't necessarily guarantee this last identity, but we can do our best.)

For our reduce function, we will use a simple base conversion. That is, if we are creating passwords with the first `n` lowercase letters, we can convert the hash value to base `n`, and use the "digits" of that value to choose letters for our password. Here is an example, using passwords of length 8 and 5 characters (a–e):

Operation	Value
some initial password	`"abcdeabc"`
apply `pwHash`	1726491528 (base 10 integer)
convert to base `nLetters`	…000012013440212103 (left-padded with 0s)
take `pwLength` least significant digits	40212103
convert digits to letters	"eacbcbad"

The last three lines above describe the `pwReduce` function. You don't have to follow exactly those steps to get the value, but should get the same result. So, `pwReduce 1726491528 == "eacbcbad"`.

For the last step, the `RainbowAssign` module contains a function `toLetter` that converts an integer to the corresponding letter. For example, `toLetter 3 == 'd'`.

The negative hash values may cause you stress. If you have done the base conversion in the standard straightforward way, you shouldn't have to worry about them. (i.e. find the lowest-order digit by taking the hash value mod `nLetters`, then recurse to find the rest of the digits.) You should get `pwReduce (-1726491528::Hash) == "aecdcdec"`.

The Map Data Structure

In order to make our table useful, we will need a data structure that can do more efficient lookups than the built-in list (which does linear searches). The Haskell standard library contains a type `Data.Map` which is analogous to the Python dictionary, or C++ std::map, or the Java TreeMap. Because the `Data.Map` module contains many functions with common names (like `map` and `lookup`), you should probably import it this way, to keep those functions out of your main namespace:

```
import qualified Data.Map as Map
```

Then you can access any of the functions by prefacing with "`Map.`". For example the module's keys function can be accessed as `Map.keys`.

The easiest way to create a map (from values of type `A` to values of type `B`) is to create a list of pairs (of type `[(A,B)]`) and then give that to the `Map.fromList` function to construct the map.

Building the Table

To create the actual rainbow table, you need to take a list of initial passwords, and use the hash/reduce functions to map the initial passwords onto a corresponding (distant) hash value. I will use the terms *height* to refer to the number of initial passwords (and thus the number of chains in the table), and *width* to refer to the number of hash/reduce operations applied to each chain.

Create a function `rainbowTable` that takes two arguments: the "width" of the table, and the list of initial passwords. It should return a `Map.Map` that maps the final `Hash` values onto the `Passwd` values at the start of the chain.

With the same parameters as the above example, we should get:

```
*Main> :t rainbowTable
rainbowTable :: Int -> [Passwd] -> Map.Map Hash Passwd
*Main> rainbowTable 0 ["abcdeabc"]
fromList [(1726491528,"abcdeabc")]
*Main> rainbowTable 1 ["abcdeabc"]
fromList [(1477708406,"abcdeabc")]
*Main> rainbowTable 40 ["abcdeabc"]
fromList [(-1993856529,"abcdeabc")]
*Main> rainbowTable 40 ["abcdeabc", "aabbccdd", "eeeeeeee"]
fromList [(-1993856529,"abcdeabc"),(1781092264,"aabbccdd"),(2135711886,"eeeeeeee")]
*Main> rainbowTable 2 ["dccdecee","cdeccaed","acbcaeec","eeeeaebd","ccdccbeb"]
fromList [(-1862546954,"ccdccbeb"),(-
716915723,"cdeccaed"),(1305042273,"acbcaeec"),(1629794057,"eeeeaebd"),(1802158710,"dc
cdecee")]
```

(The last example is the example table from the start of the assignment.)

Creating, Reading, and Writing Tables

To create a rainbow table based on random initial passwords, you can use the provided `buildTable` function which takes these arguments: your `rainbowTable` function as described above, the number of letters being used, the length of each password, and the width and height of the table. For example, it might return something like this:

```
*Main> buildTable rainbowTable nLetters pwLength width height
fromList [(-1860304949,"bddaaaeb"),(-1422325927,"accabecb"),(-
857324890,"aaaababb"),(454267643,"dbaacded")]
```

Because the `buildTable` function requires input (a seed for the random number generator), the result is an `IO` object (actually, a `Map.Map` wrapped in an `IO` object). There are some special rules for working with `IO` objects that we won't go into.

Since the calculating a rainbow table can take a long time, the `RainbowAssign` module contains functions that write a table to a file and read it back again. The easiest thing to do with `buildTable` is to immediately write the results to a file. You can define a function like this, and then just call `generateTable` when you want to generate a new table and save it to disk:

```
generateTable :: IO ()
generateTable = do
  table <- buildTable rainbowTable nLetters pwLength width height
  writeTable table filename
```

To read the table back, you can use the `readTable` function. Again, since the results of `readTable` are dependant on input (contents of the file), you must deal with the `IO` object. You can probably get by with this recipe:

```
test1 = do
  table <- readTable filename
  return (Map.lookup 0 table)
```

Put the expression you want to test in the parens on the last line: calling the test function will give you the result of evaluating that expression.

Reversing Hashes

Now that we have our rainbow tables created, we can try to reverse the hash function.

In order to do this, we take a hash value, and repeatedly apply `pwReduce` and `pwHash` to it. At each point, we check to see if the hash value is a key in the table. If it is, we start with the corresponding password and iterate until we either find the corresponding password or reach the end of the chain.

Note that **it is possible to get false positives** in the initial lookup: you may find a generated hash value as a key in the table, but not find the original hash value in the chain. This is an unavoidable consequence of the possible collisions in the hash and reduce functions. Make sure you **check every row of the table that is a potential match**, not just the first one.

Create a function `findPassword` that takes three arguments: the rainbow table, the "width" of the table, and the hash value you are trying to reverse.

The function should return a `Maybe Passwd`. In Haskell, `Maybe` values can either take on a value of the corresponding type, or `Nothing`. You should return `Just` the password if it was found, and `Nothing` otherwise. The `Data.Maybe` module contains some functions that may simplify your life when working with `Maybe` values.

Here are some examples:

```
*Main> let table = rainbowTable 40 ["abcdeabc", "aabbccdd", "eeeeeeee"]
*Main> findPassword table 40 1726491528
Just "abcdeabc"
*Main> findPassword table 40 (-206342227)
Just "dbddecab"
*Main> findPassword table 40 1726491529
Nothing
*Main> findPassword table 40 0
Nothing
```

Experimenting

At this point, you're probably wondering just how effective your password cracking system is. With letters a–e and length 8, we have only 5^8 = 390625 possible passwords: it shouldn't be that hard to crack them.

This function tries to crack `n` randomly generated passwords, and reports the results:

```
test2 :: Int -> IO ([Passwd], Int)
test2 n = do
  table <- readTable filename
  pws <- randomPasswords nLetters pwLength n
  let hs = map pwHash pws
  let result = Maybe.mapMaybe (findPassword table width) hs
  return (result, length result)
```

This uses the `mapMaybe` function from the `Data.Maybe` module to return only the passwords that could be cracked. With the sample parameters from above, I successfully reverse about 7.5% of hashes: not bad for such a toy-sized table.

Choosing a good value for `width` is surprisingly important: too small and not enough hashes are represented in your table; too large and you get too many chain collisions (decreasing the effective height), causing more false-positives in the lookup, and much longer times to reverse a hash. For given values of `nLetters` and `pwLength`, there seems to be a “sweet spot” for the value of width where the success rate stops increasing, but you are still not getting many collisions.

For `height`, bigger is better.

Please feel free to experiment further. You should find that as the number of potential passwords (`nLetters` and `pwLength`) increases, the best value for width increases as well. So,

as the situation gets more realistic, the rainbow table becomes more and more space-efficient, with cracking times increasing relatively little.

Notes and Summary

You should create the following functions as described above:

- `pwReduce`: maps a hash value to an arbitrary password.
- `rainbowTable`: generates a rainbow table, given a list of initial passwords.
- `findPassword`: reverses a hash to the corresponding password, if possible.

All functions you define (the above and any others you need to get there) should have explicit type declarations (the `::` thing).

Coding style/readability/maintainability count. Using the `-Wall` argument at the GHC/GHCi command line might help you find a few style problems (but certainly not all).

Compiling Your Code

You can likely work entirely at the GHCi prompt, but if you do want to compile your code, you first need to define a main function, perhaps like this:

```
main :: IO ()
main = do
  generateTable
  res <- test2 10000
  print res
```

Then, the command line will be like this:

```
ghc -O2 --make -Wall rainbow.hs
./rainbow
```