TWiki > EC330 Web > HomeworkFour (2015-03-05, AriTrachtenberg)

EC330 - Spring 2015 - Homework 4

Assigned: Thursday, March 5

Due: Thursday, March 19 *by the beginning of class.*

Warning

This is a long homework, despite the intervening spring break. Don't start it late!

Boilerplate

- I will give 0.1 points of extra credit (and fame on the web page) to the best solutions for each problem on this homework (and all other homeworks in the class), as judged by the graders.
- Except for the programming problems or when stated otherwise explicitly, provide all code in the form of precise pseudocode.
- The hash computations on problem 4 are *system dependent*. You may not get the same hashes unless you compile and run the code on the lab machines or the eng-grid and use the `-O3` compiler directive to optimize the code.
- Summario flexione independente tu del, qui lo libere existe publicationes, e qui pote americano. Se articulo summarios principalmente sia e.

EC330 - Spring 2015 - Homework 4
- Assigned: Thursday, March 5
- Due: Thursday, March 19 by the beginning of class.
- Problem 1 - Hashing ([25 points](
- Problem 2 - Selection [25 points]
- Problem 3 - Bloom filters [25 points]
- Problem 4 - Password cracking [25 points]
  - 4a. Brute force [5 points]
  - 4b. Simplified rainbow tables [10 points]
    - Building the table
    - Using the table
    - The problem
  - 4c. Build your own simplified rainbow table [10 points]

4d. Extra Credit

Problem 1 - Hashing ([25 points](

Consider an open addressed hash with probe sequence $h_i(k) = h_{i-1}(k) + i$, with $h_0(k) = h(k)$. Which of the following properties will this probe sequence exhibit:

i. primary clustering
ii. secondary clustering
iii. always able to add a key if the table is not full

Problem 2 - Selection **[25 points]**

Suppose you have two *sorted* arrays of `n` integers: $X_0[1..n]$ and $X_1[1..n]$.

i. Devise and analyze an efficient algorithm for finding the median of all numbers in arrays X_0 and X_1.
ii. Now, suppose you have not two, but three such arrays, each with `n` elements. Devise and analyze an efficient algorithm for finding the median.
iii. Do the same for `n` arrays, each with `n` elements.

Problem 3 - Bloom filters **[25 points]**

For this problem you will design your own Bloom Filter class called `myBloomFilter` that extends the following class BloomFilter:

```
#ifndef BLOOMFILTER_H
#define BLOOMFILTER_H
#include <string>
using namespace std;

class BloomFilter {
        public:
        /**
        * Instantiate an empty Bloom filter of length  chars.
        */
        BloomFilter (int mm) : length(mm) {}

        /**
        * Instantiates a Bloom filter from a given string
        * @requires  must have been produces by the output() call of a BloomFilter object.
        */
        BloomFilter (string filter) : length(filter.length()) {}
        /**
        * inserts  into the filter
        */
        void insert(string item);

        /**
        * Checks whether  is in the filter.
        * @return  true if the item is probably in the filter
        *          false if the item is definitely not in the filter
        */
        bool exists(string item);

        /**
        * @return A string of  characters representing the Bloom filer
        */
        string output();

        protected:
        int length; /** The length of the Bloom filter, in chars. */
};

#endif  /* BLOOMFILTER_H */
```

Your Bloom filter should have no false negatives and as high a false positive probability as you can achieve. Please turn in your code and a brief explanation of your implementation on the solution topic, and then upload your code, as one `myBloomFilter.h` composite declaration-implementation file, on the course TWiki.

There are a number of engineering challenges in doing this problem effectively, including:

- Designing good hash functions for the Bloom filter internals.
- Determining the ideal number of hashes to use.
- Managing the translation from bit-based Bloom filters (in class) to a char-based Bloom filter here.

Problem 4 - Password cracking **[25 points]**

Recall that an operating system only saves a *hash* of your password. When you type in a password, the system

authenticates you by computing a hash of the password and comparing it to the hash stored locally. For this problem, we will assume that passwords are hashed by the *hash* function in the attached file `hash.cpp`. It is based on a *modified* sha1 implementation (attached below) taken from an implementation by Micael Hildenborg[1].

Throughout this problem, we will assume that passwords characters are chosen between `!` and `~` in the ASCII table.

4a. Brute force [5 points]

The following password hashes were found on a system. Through (unspecified) outside information, we know that the passwords were all three ASCII characters long. Write code that cracks these passwords. Provide your cracked passwords in the solution topic **and** also attach to it a zip of your code (called =problem4a.zip), including all files used and a Makefile for compiling them.

Warning: Can't find topic EC330.Hw4aMikhailAndreev

4b. Simplified rainbow tables [10 points]

For longer passwords, people sometimes create tables that allow them to quickly look up a password hash to get the password(s) that generated it. These tables can be extremely large, and thus rainbow tables are used to make them smaller. To do this, we make use of a `reduce` function that converts an arbitrary hash back into a string of characters of the same length as the password.

Building the table

We can start with a string `str1`, compute its hash `hash(str1)`, then compute its reduction `reduce(hash(str1))` to get a different password. We can repeat alternately computing `hash` and `reduce` functions for a number of iterations and record the final password we see `str2`. The two strings `str1` and `str2` will thus form one entry in our rainbow table.

For example, if we start with the password `ec330`, then hashing it would produce `9964f77bad61684dfcf4cff0f6b4e5504b31b3d5`; reducing this hash (to a length 5 string) would give `CX]dx`, etc. as we see in this table:

Original string	hash	reduce	hash	reduce	hash	reduce
ec330	`9964f77bad61684dfcf4cff0f6b4e5504b31b3d5`	`[UUPO`	`bf9...51a`	`eSTfQ`	`d7a...6b5`	l]oY^

This row of the table can thus be condensed into pair (ec330,l]oY^), which would be one entry in our rainbow table.

Using the table

To use the table, we start with the hash we are trying to crack, and then repeatedly `reduce` and `hash` it until we see the second string of an entry in the rainbow table. Starting at the first string in the entry, we then redo the hash/reduce iterations until we find the password that produced the hash.

For example, when cracking the password hash `bf9bfeaa32733c1f4c9bfe487aaf00fcc6cf251a`, we would reduce the hash to `eSTfQ`, hash it to `d7a906828c377d8db0ef735a27205551ea1ab6b5`, and reduce it to `l]oY^`, which is the entry in the rainbow table. We thus know that the initial string `ec330` will lead us to the password we want. Indeed, following the hash/reduce in the above table shows that the password `[UUPO` produced the password hash we are trying to crack.

The problem

Attached to this table is a simplified rainbow table, with each line given as a pair of five-character passwords. Use this table to crack the following password hashes of *five* character passwords:

Warning: Can't find topic EC330.Hw4bMikhailAndreev

4c. Build your own simplified rainbow table [10 points]

Write code to build your own simplified rainbow table. Attach your code to the solution topic, your produced table, and provide an explanation for how your code works.

You are encouraged to use the eng-grid to compute your table in parallel.

4d. Extra Credit

Build the meanest rainbow table you can. On 5pm of the day the homework is due, we will post a number of hashes for you to crack. Be the first to post your solution on the course web page and get 0.02 points of extra credit each.

Notes

1 : The modification you should use is attached to this topic

I	Attachment	History	Action	Size	Date	Who	Comment
	hash.cpp	r2 r1	manage	1.4 K	2013-02-19 - 23:02	AriTrachtenberg	PROBLEM 4 - hash table implementation
	hash.h	r2 r1	manage	0.8 K	2013-02-19 - 22:59	AriTrachtenberg	PROBLEM 4 - hash table header
	sha1.cpp	r2 r1	manage	6.7 K	2013-02-21 - 12:40	AriTrachtenberg	PROBLEM 4 - implementation file for sha1 (NON-STANDARD!)
	sha1.h	r1	manage	2.3 K	2013-02-21 - 12:39	AriTrachtenberg	PROBLEM 4 - header file for sha1 implementation
	table.txt	r12 r11 r10 r9 r8	manage	31.2 K	2013-02-25 - 22:52	AriTrachtenberg	PROBLEM 4 - Simplified rainbow table for your use

Topic revision: r2 - 2015-03-05 - AriTrachtenberg