Crypto report

Write a report on what you have learnt from the above programming tasks. This is an open topic, but as a minimum requirement you should consider the efficiency of the algorithms used, and explain the strategies applied. For tasks 2.3 and 2.4 (i.e. password cracking), you should conduct timing tests then include your results and your analysis of them in the report. The literatures related tasks should be referenced. There is no word limit on the report, however as a rough guide, 700-800 words should be sufficient.

**Hash Encryption Cracking**

In cryptography the roll of hash function can be very important for the protection and storage of data. Generally, within computer science such functions are used for mapping data to allow faster search time for retrieval at a later data. However, in cryptography these functions can be used to transform data into an unrecognisable format such as changing it to a fixed and containing only hexadecimal characters it to a fixed length. The hash functions are known as one-way function due the fact they are easy to turn data into a hash but very difficult to turn it back this due to their internal method. If given a hash code, there is no way to tell what the original data could be unless provided with extra information e.g. the specific hash encryption method. There are many type of hash functions such as MD5, SHA-1. The problems faced in this report all deal with SHA-1 hash function.

As mentioned hash function are one-way functions and therefore do not have an inverse function, but this doesn’t mean that it is impossible to break. In fact, using that same hashing function can be used to decrypt the hash. This does however take a rather substantially long time when compared to the initial encryption. This paper will now show two such methods of breaking SHA-1 encryptions.

For either of the methods to work it is required knowing the original hash function e.g. SHA-1, the alphabet of the data; valid characters such as a-z, along with the possible maximum size of the original data as this creates boundaries in the code.

**Brute Force**  
The most obvious method of breaking encryption is the brute force method where by every single combination and permutation of the know alphabet is produce. These are then encrypted, the resulting encryption is then measured against the original hash, if there is a match then the decrypted data has been discovered.   
This process has the highest computational time of the two methods shown. However, there are different ways to go about implementing this method to allow for greater efficiency.

Nested Loop  
By using a series of nested loops it is possible to recreate the input by cycling through the loops every permutation of the alphabet will be achieved.   
The solution is rather primitive use of brute force as it does have wasted computations. Using the pseudocode (see **Appendix 2**) it is shows that the first item of the alphabet is “” (empty string). This is there so it is possible to create all possible length passwords e.g. “” + ”” + “a” = “a”.  
However, if you look at the code it is inevitable that once the most inner loop has completed its cycle the middle loop will change to “a” so the next iteration will equal “” + “a” + “” = “a”.  
With the pseudocodes alphabet containing 4 characters means there is a possibility of 40 combination, but due to the repeat combinations there is an extra 9 combinations.  
To counter this it would be possible improvements to this method could involve the use of threads. By removing the empty string element and using independent loops that are cycling through fix length permutations the search time will increase, though the cost will result in high computational power.

Recursion  
Recursion is the method of breaking down a problem to its simplest solution by calling itself within itself. By using this method if is possible to build a string that can be hashed to see if it matches the original message.   
Looking at the pseudocode (see **Appendix 3**) it is possible to see that when a string is fed into the method the message is split into two strings. One string is the last character (rhs), while the rest is stored in another string (lhs). The method then focuses solely on the rhs string. This is since it only ever needs to increment the value of rhs by one before returning the output. Though once the rhs has cycled through the alphabet it resets to the first character and calls the method again while passing through the lsh string and the whole process is repeated. This is like how you add 1 to another number

e.g. 189 + 1   
 lhs = 18, rhs = 9  
 (lhs = 1, rhs = 9), rhs = 0  
 = 190

This method provides many benefits such as no repeat outputs and is not limited to a maximum length; just like adding 1 to a number and go on to an infinite length.

Results  
When given a set of hash strings both algorithms were able to find the original message for each the results are below.

<<<<<<<<<<<<<<<<<<<<<RESULTS>>>>>>>>>>>>>>>>>>>>

As shown by the results table (see **Appendix 4**) the recursive function was the fastest at finding the original message for each string. However, it still takes a long time to solve one hash. Also you can see that the time to build a string rises exponentially when even a single character is added. A 5-character string takes approximately a minute to crack where a 6 letter string can take over 30 times as long. So, using this method to crack any message that contains a large number of letters will be a completely inefficient.

**Rainbow Table**In contrast to the brute force method the efficiency of a using rainbow table is much greater. This is because a lot of the computation is done before hand and the end method is essentially a look-up table. By having a precompiled table this method resulting in a time/space trade off, where the time saved when performing the crack is now taken up in space to store the table.

A issue with using this method is that unlike brute force; when given a long enough time and the correct alphabet, it guarantees the correct result, it is highly likely that these tables will not be able to crack 100% of all password within a given space. This is due to collisions of the

The creation and use of a rainbow table can be split up into three areas; creating reduction function, building a table, and cracking algorithm.  
Both section use the same me

Reduction Function  
The most vital part of a rainbow table is what is know as a reduction function. This is the act of taking a hash and then converting it to the match a possible string that is valid within the password space. It is not the case that it is trying to inverse the hash but rather by using a set of rules change the hash into something more meaningful and a possible solution.

The rules within a reduction function can be anything you want but the key to a successful rainbow table steams from the ability to create unique possible solution when given a hash, by doing so it will avoid wasted space and computational time.

In the table created for the assignment the first step of the reduction function requires transforming the hash into an integer value. This makes it easier to be processed later. Creating a integer value from a hash was a simple process and took example for the hash it is trying to decipher. As a SHA-1 hash is a unique combination of hexadecimal values, it stands to reason if the ASCII values of each character were concatenated together in a string this must also be a unique pattern. This string can then be parsed into an integer (in the case of the assignment a BigInteger class had to be used to hold a value of that size).  
With the resulting integer value is then proceed into a string following the code provided by Rong Yang (see **Appendix 5).**

Now, to avoid collisions and chain merging (where two chains start with different values then at some point produce the same value and result in an identical sequences) different reduction function should be used between each link of the chain. This is generally done by manipulating the hash (or resulting integer) using the value of the position. This mean that if a duplicated value has been made unless a value was created in the same position as the a previous it is highly unlikely to continue and reproduce the same results.

Creating a Table  
The method to creating a table is simple. A random number between 0 and the possible total amount of passwords is generated, this number is then hashed and transformed to a possible password using the reduction function. This first reduction is the starting point for a chain. This value is then hashed and reduced for a set amount of times and the last reduction is end of chain. These two values are then stored as a pair This process is repeated many times to build a table.

Consideration must be taken when deciding the number of chains and the length of chain. This is because two low of a number will result in a small list of passwords it is possible to crack. Too large mean more space and computation time to search.   
The acceptable number of chains can be summed up using the formula:

N

As the space within these chain is limited care must be taken when stating what start and endpoint should be kept. As the creation of the start point are generated at random the it is possible that the same number could be generated twice. If this were to happen it will result in the same chain being produce and thus a waste of space. It is also advisable to remove any chain where the end chain value is equal to that of a previous chain. In the assignment this was resolved using HashMap (see **Appendix 6**). Every time a new start or endpoint was created these were checked to see if on already existed, if so the current chain was thrown away and the counter was incremented to make up for the lost chain.

It not just the start and end values of a chain that must be monitored, during a chain creation it is beneficial to temporarily store the results of the reduction function in a HashMap as well. This is help keep an eye out for repeating patterns within the chain. If a chain were to fall into a cycle of reproducing the same result this will also take up valuable space within the table. By storing the previous and current result of the reduction function it is easy to see the start of a repeating pattern.

e.g. Valid chain = a=>b, b=>c, c=>d, d=>e  
 Invalid chain = a=>b, b=>c, c=>a, a=>b

Cracking a Hash  
Table has been built it is now possible to use the table to crack existing hashes. The crack operates in the way that it takes the hash supplied in the input, reduces it with the reduction function and then compared the result with that of the searches for the end value.

Search last chain  
walk back   
build chain

Results  
The results of the show that the

**APPENDIX**

1) **Git Hub repository holding the code for the assignment.**<https://github.com/UWE-SimonLlewellyn/Cryptography>  
Each project is made using Java 7 and can be loaded via Netbeans.

2) **Pseudocode for Brute force by Nested Loop**   
alphabet = {“”,”a”, “b”, “c”}   
String s = “”  
LOOP i < alphabet length  
 LOOP j < alphabet length  
 LOOP k < alphabet length  
 IF s == hash  
 return s  
 END IF  
 s = alphabet[i] + alphabet[j] + alphabet[k]  
 END LOOP  
 END LOOP  
END LOOP

3) **Pseudocode for Brute Force by Recursion**   
MAIN()  
 alphabet = {“”,”a”, “b”, “c”}   
 String new = “”  
 LOOP while hash(new)!= to cyphertext  
 new = nextString(new, alphabet)  
 END LOOP  
 return new  
END MAIN  
---------------------------------------  
nextString(String s, String alphabet)  
 int n = s.length  
 String lhs = "" , rhs = “” // left and right hand side  
 char last = last char of the alphabet  
 IF n == 0  
 return first char of alphabet  
 END IF  
 IF n > 1  
 lhs = all but last chats of string  
 END IF  
 rhs = last char of string  
 IF rhs == last  
 rhs = first char of alphabet  
 return nextString(lhs, alphabet) + rhs  
 ELSE  
 rhs + 1 //next char in alphabet  
 return lhs + rhs;  
 END IF  
END newString

4) **Result table comparing Nested Loop to Recursion**

5) **Pseudocode for converting integer to string provided by Rong Yang**

6) **Code to chain for valid and non-repeating chain fucntions**