Tan Shin Jie
1003715

Section 5

In the brute force attack to recover the passwords for hash5.txt, I enumerated all possible combinations and compute the hash for each one. The passwords and its hashes are stored in a python dictionary named reverse_hash_table with **hash as key** and **password as value**. With the reverse_hash_table, I simply iterate through again hash5.txt and look up my reverse_hash_table to get back the corresponding passwords. The time taken for this approach is around 198s. In the rainbow table attack, I managed to recover all the hashes twice as fast (101.75s) as compared to brute force. However, a brute force attack becomes infeasible (takes too long) when the password length increases to 6. In the first try to solve for the salted password hashes, I tried to use the same parameters (t=3800 and m=600000) as the rainbow table generated for password length 5, but it was only able to crack 7 out of 15 the salted hashes. I figured since the input space has increased, larger rainbow tables need to be generated to increase the possibility of containing all the possible passwords within the chains.

Two strategies were explored to solve for the 6 characters password.
<u>Strategy 1: Increasing chain length</u>
In this strategy, all parameters were kept the same (chain size=3800, table index=0, part index=0) except for the chain length *m*, a minimum *m* was to be found such that the rainbow table generated is able to crack all the salted hashes.
First, an upper bound m_max is gauged by consistently increasing the number of chains by 1,000,000 starting from 1,000,000 until a rainbow table generated is capable of cracking all salted hashes. Then, a binary search method is used iteratively between the i^{th} m_{max} the previous $(i-1)^{th}$ m_{max} value to find the approximate minimum m.

<u>Strategy 2: Increase the number of rainbow tables</u>
In this strategy, a set of rainbow tables was generated by changing the table index argument which in turn changed the reducing function while the table was being constructed. This strategy works rather well because although I generated 6 rainbow tables, just 2 out of the 6 (table_index=0, table_index=1) were able to crack all the hashes. Therefore, the effective generation time can be calculated just taking into account the generation time for the two tables.

Final results

	Generation time(s)	Lookup time(s)	Total time(s)
Unsalted	96.9	4.85	101.75
Salted (strategy 2)	197.4	8.06	205.4

Results from increasing chain length m

Iteration	t	m	Generation time(s)	Score	Lookup time (s)
0	3800	600,000	92.6	7/15	4.09
1	3800	1,000,000	156.0	10/15	4.41
2	3800	2,000,000	343.6	12/15	4.75
3	3800	3,000,000	510.7	13/15	5.09
4	3800	4,000,000	694.4	13/15	5.43
5	3800	5,000,000	860.6	14/15	5.70
6	3800	6,000,000	1002.2	14/15	6.29
7	3800	7,000,000	1203.1	15/15	5.62
8	3800	6,500,000	976.8	15/15	5.76
9	3800	6,250,000	933.1	14/15	6.48
10	**3800**	**6,375,000**	**1087.6**	**15/15**	**5.66**
11	3800	6,312,500	1090.7	14/15	6.34
12	3800	6,343,750	1115.0	14/15	6.77
13	3800	6,359,375	1114.2	14/15	7.78

Results from creating a set of 6 rainbow tables

t	m	table index	Generation time(s)
3800	**600,000**	**0**	**91.1**
3800	**600,000**	**1**	**106.3**
3800	600,000	2	94.4
3800	600,000	3	101.4
3800	600,000	4	106.1
3800	600,000	5	104.2
Score=15/15 Total generation time=603.5s Effective generation time=197.4s Lookup time=8.06s			

Input space size in 5 loweralpha-numeric characters	= 60,466,176
Input space size in 5 loweralpha-numeric + 1 loweralpha characters	= 1,572,120,576
Input space size in 6 loweralpha-numeric characters	= 2,176,782,336