Activity 1

1. Write a simple python program to use the word from the dictionary to find the original value of '**d54cc1fe76f5186380a0939d2fc1723c44e8a5f7**'. Note that you might want to include substitution in your code (lowercase, uppercase, number for letter ['o' => 0 , 'l' => 1, 'i' => 1]). Hint: Here is a snippet for sha1 and md5 functions.

```
import hashlib
import time
import requests
from tqdm import tqdm
import sys

text_list_url =
"https://raw.githubusercontent.com/danielmiessler/SecLists/master/Password
s/Common-Credentials/10k-most-common.txt"

response = requests.get(text_list_url)
password_list = response.text.splitlines()
```

```
def is_password_match(pw:str, hashed_value:str):
    if hashed_value == hashlib.sha1(pw.encode()).hexdigest():
        print('Password is',pw)
        print('Hash function: SHA1')
        return True
    elif hashed_value == hashlib.md5(pw.encode()).hexdigest():
        print('Password is',pw)
        print('Hash function: MD5')
        return True
    else:
        return False
```

```
letter_2_number = {'o':0,'l':1,'i':1}
encode_value = 'd54cc1fe76f5186380a0939d2fc1723c44e8a5f7'

found = False

def string_permutations(string:str):
    permu_at(string.lower(),0)

def permu_at(string:str, idx:int):
    global found
    if found:
        return True
    if idx < len(string):
```

```
        if not (string[idx] in letter_2_number and permu_at(string[:idx] +
str(letter_2_number[string[idx]]) + string[idx+1:], idx+1)):
            if not permu_at(string[:idx] + string[idx].upper() +
string[idx+1:],idx+1):
                return permu_at(string, idx+1)
        return True
    else:
        if is_password_match(string, encode_value):
            found = True
            return True
        return False
```

```
for password in tqdm(password_list):
    if found:
        break
    string_permutations(password)
```

Output:

```
12%|█  | 1158/10000 [00:00<00:02, 4337.35it/s]
Password is ThaiLanD
Hash function: SHA1
```

2. For the given dictionary, create a rainbow table (including the substituted strings) using the sha1 algorithm. Measure the time for creating such a table. Measure the size of the table.

```
sha1_rainbow_table = {}
def permu(word:str, idx:int, length:int):
    if idx == length:
        sha1_hash = hashlib.sha1(word.encode()).hexdigest()
        sha1_rainbow_table[sha1_hash] = word
        return

    low = (word[:idx] + word[idx].lower() + word[idx+1:])
    permu(low, idx+1, length)

    high = word[:idx] + word[idx].upper() + word[idx+1:]
    permu(high, idx+1, length)

    if word[idx] in letter_2_number:
        c = word[:idx] + str(letter_2_number[word[idx]]) + word[idx+1:]
        permu(c, idx+1, length)
```

```
letter_2_number = {'o':0,'l':1,'i':1}

times = 10
prod = 1


print('password_list length',len(password_list))

for i in range(times):
    sha1_rainbow_table = {}

    start_time = time.time()

    for word in password_list:
        permu(word,0,len(word))


    end_time = time.time()
    print('Time',i+1,':',end_time - start_time)
    prod *= end_time - start_time

table_size = sys.getsizeof(sha1_rainbow_table)
geometric_mean_time = pow(prod,1/times)

print('avg geometric menn time usage: %.4f seconds' % geometric_mean_time)
print('table size: %.4f MB' % (table_size / (1024 * 1024)))
print('rows num: %d' % (len(sha1_rainbow_table)))
```

Output:

```
password_list length 10000
Time 1 : 3.9485650062561035
Time 2 : 3.9770519733428955
Time 3 : 4.023262977600098
Time 4 : 3.9170949459075928
Time 5 : 3.954897165298462
Time 6 : 3.989788770675659
Time 7 : 3.8790321350097656
Time 8 : 4.0656538009643555
Time 9 : 4.032056093215942
Time 10 : 4.025896787643433
avg geometric menn time usage: 3.9810 seconds
table size: 80.0001 MB
rows num: 2650956
```

3. Based on your code, how long does it take to perform a hash (sha1) on a password string? Please analyze the performance of your system

```
rps = len(sha1_rainbow_table) / geometric_mean_time
spr = geometric_mean_time / len(sha1_rainbow_table)

print('performance: %.4f rows per second' % (rps))
print('seconds per row: %.10f' % (spr))
```

Output:

```
performance: 665909.7969 rows per second
seconds per row: 0.0000015017
```

4. If you were a hacker obtaining a password file from a system, estimate how long it takes to break a password with brute force using your computer. (Please based the answer on your measurement from exercise #3.)

```
charset_num = 26 + 26 + 10
password_length = 8

print('brute force time: %.4f seconds' % (pow(charset_num,password_length)
* spr),'or %.4f years' % (pow(charset_num,password_length) * spr / (60 *
60 * 24 * 365)))
print()
```

Output:

```
brute force time: 327882404.7848 seconds or 10.3971 years
```

5. Base on your analysis in exercise #4, what should be the proper length of a password. (e.g. Take at least a year to break).

```
for i in range(1,21):
    if((pow(charset_num, i) * spr / (60 * 60 * 24 * 365)) >= 1):

        print('password length: %d' % (i-1))
        print('brute force time: %.4f seconds' % (pow(charset_num, i-1) *
```

```
spr),'or %.4f years' % (pow(charset_num, i-1) * spr / (60 * 60 * 24 *
365)))

        print('password length: %d' % i)
        print('brute force time: %.4f seconds' % (pow(charset_num, i) *
spr),'or %.4f years' % (pow(charset_num, i) * spr / (60 * 60 * 24 * 365)))

        print('password length: %d' % (i+1))
        print('brute force time: %.4f seconds' % (pow(charset_num, i+1) *
spr),'or %.4f years' % (pow(charset_num, i+1) * spr / (60 * 60 * 24 *
365)))
        break
```

Output:

```
password length: 7
brute force time: 5288425.8836 seconds or 0.1677 years
password length: 8
brute force time: 327882404.7848 seconds or 10.3971 years
password length: 9
brute force time: 20328709096.6603 seconds or 644.6191 years
```

6. What is salt? Please explain its role in protecting a password hash.

Salt คือข้อมูลแบบสุ่มที่ถูกเพิ่มเข้าไปในรหัสผ่านก่อนการแฮชเพื่อเพิ่มความปลอดภัยในการเก็บรักษารหัสผ่าน โดยการใช้ salt ช่วยป้องกันการโจมตีแบบ Rainbow Table และทำให้แฮชของรหัสผ่านเดียวกันแตกต่างกัน แม้ว่าผู้ใช้หลายคนจะใช้รหัสผ่านเดียวกัน เมื่อผู้ใช้สร้างบัญชี ระบบจะสร้าง salt แบบสุ่มรวมกับรหัสผ่าน จากนั้นแฮชและเก็บค่าแฮชและ salt ไว้ในฐานข้อมูล เมื่อล็อกอิน ระบบจะดึงค่า salt และแฮชรหัสผ่านใหม่เพื่อตรวจสอบความถูกต้องว่าตรงกับค่าที่เก็บไว้หรือไม่ วิธีนี้ช่วยเพิ่มความยากในการคาดเดารหัสผ่านและทำให้ระบบมีความปลอดภัยมากขึ้น