Stephanie Galvan

Date: September 9, 2019

Lab 1 – Password Cracking

CS 2302 – Data Structures MW 1:30PM - 2:50PM

Instructor: Diego Aguirre

TA: Gerardo Barraza

Fall 2019

Lab 1: Password Cracking

Introduction

The main purpose of the following lab is to implement and gain more practice with recursion. Given a text file containing 100 records of 100 system accounts, implement a recursive method to generate all passwords of each account.

The text file is divided in 100 files containing a username, a salt value, and a hashed password per line. The real passwords will only contain integers (0-9) and each password has a length range of 3 to 7 (inclusive). In order to determine the real password, a method has been provided (*hashlib_sha256*) in which a possible password needs to be concatenated with the user's salt value and then the result can be compared with the user's hashed password.

The source code must follow these rules:

- The method must be recursive
- The password length should be the parameters of the function
- Limit of two nested loops inside the function

Proposed Solution Design and Implementation

The proposed solution is divided in three parts: a function called *hack_file* that takes the desired minimum and maximum length of passwords (it must fall in the range of 3-7; also this function can be eventually discarded and instead be run in the main method), a function called *get_combination* that populates all possible password combinations of some given size, and function called *find_password* that will find the real passwords by comparing hashlines and print them if found.

First, the main method calls *hack_file* with the parameter of 3 and 7 since these are the limit for the range. Since the expected parameters are integers, the method begins with a condition to ensure a valid data type is being passed; if the function does not receive the expected input, it will print a message to the console notifying the error. Additionally, the function needs a parameter that it's in the range between 3 and 7 (inclusive) and it will also print a message to the console for such error. If all criteria is met, *hack_file* will proceed to run a for loop to call *get_combination* with the desired password sizes.

In *get_combination,* two parameters are passed: an int value representing the size of the desired password and a string value that will represent the password. This particular method implements recursion by recursively creating all combinations. The base case is defined when the size reaches zero since it is assumed a combination has been created and then it will call the method *find_password* to compare the string. The recursive call of *get_combination* is a for loop that traverses from digits 0-9 to populate

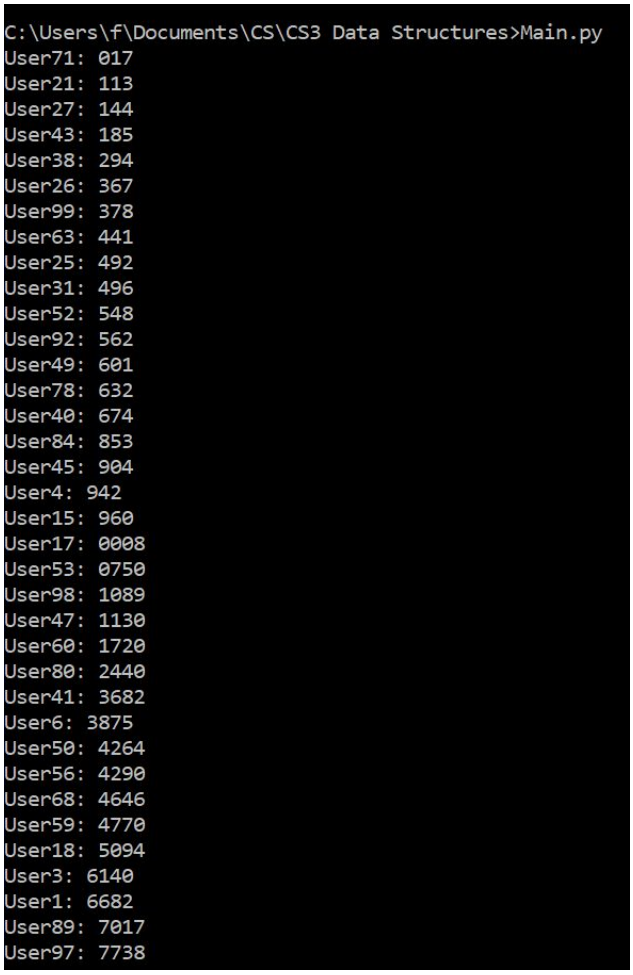
all combinations. In each recursive call, the parameter of size decreases by one and the parameter comb adds the digit to the string.

Finally, the last part of the program requires the comparison of strings. The function *find_password* takes a string as a parameter and it will represent the potential password. The method will iterate through the given password_file.tx and it will split each line by user, salt value, and hash value. Then, a new hashline is created by taking the password parameter and concatenating it with the salt value. If the new hashline matches the hashline of the current line of the file, then the user and real password is printed to the console.

Additionally, a fourth function was created to contain some test cases that will be discussed below.

Experimental results

First, the normal test run is run and it prints the 100 found passwords with its corresponding user (unsorted). The run time of the normal run is almost 3hrs.



```
User72: 7804
User35: 8510
User79: 9038
User95: 9601
User67: 9611
User12: 9963
User46: 04288
User54: 10649
User96: 12614
User75: 34995
User7: 39666
User42: 42273
User16: 55409
User86: 56660
User76: 62381
User48: 65708
User33: 66477
User30: 79612
User13: 80104
User14: 87955
User93: 93499
User10: 95300
User85: 003757
User29: 027037
User34: 073282
User23: 079312
User44: 104034
User64: 118709
User22: 133488
User57: 150629
User88: 497213
User9: 536596
User70: 707289
User69: 725259
User73: 758866
User28: 882676
User2: 949228
User24: 965847
User83: 984802
```

```
User11: 0254985
User87: 0259384
User20: 1071180
User61: 1440267
User39: 1498538
User36: 1881145
User51: 2043256
User0: 2419445
User62: 2442002
User55: 2466480
User5: 2671660
User82: 3118952
User91: 3244694
User19: 4202784
User66: 6319171
User94: 6675953
User8: 7041617
User90: 7579036
User77: 7699450
User32: 7846812
User81: 8174401
User74: 8287736
User58: 8707532
User37: 9562452
User65: 9861315
Running time for normal run:  9813.094760656357 seconds
```

As shown below, this screenshot refers to the test cases provided. The first test case refers to an invalid data type for parameters in which the double values of 3.0 and 7.0 were passed. The second test case included the parameters 2 and 7 in which 2 was lower than the expected minimum value for password size. Lastly, the last output is from the parameters of 3 and 8 in which 8 exceeds the expected value for maximum value for password size. All cases were handled by printing an error message to the console.

```
Invalid input. Please try again
Running time for invalid data type: 0.007994174957275390 seconds

Invalid input. Please try again
Running time for invalid data type: 0.0009977817535400390 seconds

Invalid input. Please try again
Running time for invalid data type: 0.0070986747741699220 seconds
```

Conclusions

Overall, the lab encouraged problem solving and recursion by avoiding the use of other libraries (such as itertools that already contains a combinations and permutations function). The lab shows the importance of understanding the fundamentals of recursion by deciding what should be the base case and what determines the recursive call. Moreover, the lab recognizes the importance of traversing through lines of a file by using for loops and how this general understanding helped find a solution to the problem. Although the program solved the problem, some improvements can be made such as returning a sorted list of passwords instead of printing them, ending the program once all passwords have been found instead of comparing every single combination, etc.

Appendix

Source code can also be accessed at:
https://github.com/stephypy/CS2302PasswordCracking/blob/master/Main.py

Originally written inside pycharm.

Provided function *hash_with_sha256* :

```
1  import hashlib
2  import time
3
4
5  def hash_with_sha256(str):
6      hash_object = hashlib.sha256(str.encode('utf-8'))
7      hex_dig = hash_object.hexdigest()
8      return hex_dig
```

Function *hack_file* :

```
37  def hack_file(start, end):
38      # Check parameter's data type
39      if not isinstance(start, int) or not isinstance(end, int):
40          print('Invalid input. Please try again')
41          return
42      if start < 3 or end > 7:
43          print('Invalid input. Please try again')
44          return
45      for size in range(start, end + 1):
46          get_combination(size, '')
```

Function *get_combination* :

```
25  def get_combination(size, comb):
26      # Once a combination of size n is found, call find_password
27      if size == 0:
28          find_password(comb)
29      else:
30          # Create combination by using digits from 0-9
31          for digit in range(0, 10):
32              new_comb = comb
33              new_comb += str(digit)
34              get_combination(size - 1, new_comb)
```

Function *find_password* :

```
11  def find_password(password):
12      with open("password_file.txt") as f:
13          for curr_line in f:
14              # Split the file by users, salt values, and hashlines
15              elements = curr_line.split(",")
16              # Get the hashline of the corresponding user
17              hashline = elements[2].replace('\n', '')
18              # Create a new hashline with the password provided and the corresponding user salt value
19              new_hashline = hash_with_sha256(password + str(elements[1]))
20              # Print the user and password if the real password was found
21              if new_hashline == hashline:
22                  print(elements[0] + ': ' + password)
```

Test Cases:

```
49  def test_cases():
50      # Invalid data types
51      start = time.time()
52      hack_file(3.0, 7.0)
53      end = time.time()
54      print('Running time for invalid data type:', end - start, 'seconds\n')
55      # Small starting size
56      start = time.time()
57      hack_file(2, 7)
58      end = time.time()
59      print('Running time for invalid data type:', end - start, 'seconds\n')
60      # Large end size
61      start = time.time()
62      hack_file(3, 8)
63      end = time.time()
64      print('Running time for invalid data type:', end - start, 'seconds\n')
```

Main method:

```
67  def main():
68      # Normal run of the program
69      start = time.time()
70      hack_file(3, 7)
71      end = time.time()
72      print('Running time for normal run: ', end - start, 'seconds\n')
73      test_cases()
74
75
76  main()
```