Security Assignment, Bruteforce and Rainbow tables.

STEINBACH Matteo University of Luxembourg
Email: matteo.steinbach.001@student.uni.lu

***Abstract*—Security 1: In this assignment, we will create a rainbow table for passwords of 5 characters over S = lowercase + uppercase letters, hashed with MD5. The file basis.py contains the skeleton for each function you have to code. It also contains examples of what each function should return if implemented correctly, with some additional explanations. Read them! We have the MD5 hash function, which is already implemented in the code as md5().**
This assignment is worth 33.3...% of my grade.

I. Comparison with bruteforce

A. 1 - Count of 5-Character Passwords

How many passwords of 5 characters can exist over S?

As previously mentioned, the set S represents ASCII letters, which encompasses both the English uppercase and lowercase alphabets, each containing 26 letters. Therefore, we have $26 + 26 = 52$ distinct characters. We are interested in generating passwords consisting of 5 characters. For each character slot, there are 52 possible choices. Thus, the total number of possible passwords is given by:

$$52 \cdot 52 \cdot 52 \cdot 52 \cdot 52 = 52^5 = 380,204,320 \quad (1)$$

So, there are $380,204,320$ possible 5 characters passwords using ASCII letters.

We will evaluate a solution where for each leaked hash, we try to bruteforce a password, hence using no memory.

B. Hashes per seconds

2 - Implement a function hashes_per_second() that computes how many hashes your computer can compute in one second. How many hashes per second can your computer compute?

a) Implementation: The idea was to perform the `md5()` hash function on the string of ASCII letters, that is done by a loop which continues while the time used is less than 1second and updates `hashes_per_second` at each iteration.

Because I wanted a more reliable value I added a for loop around which performs this operation 20 times and takes the average so we have a relevant idea of the computing power which is closer to the average for my computer as I made sure the lowest possible amount of background operations where on going.

b) Average hashes my computer can compute: For 20 1 second measurements the average hashes per seconds produced is of 916611.7 around $9.1 \cdot 10^5$ hashes.

C. Time to break 5 char passwords

3 - On average, how long would it take to break one 5 characters password on your computer? In the worst case?

In the worst case this is the calculation as in the worst case we are trying all the possibilities.

$$\frac{possibilities}{hashespersec/1} = \frac{52^5}{916611.7} = 414.793 \text{ seconds} \quad (2)$$

In the best case we would only have to try 1 password, so the average case would be to try half of the possibilities and thus the result would be $\frac{414.793}{2} = 207.396 seconds$. This implies that the `md5()` function and the method of generating passwords from these hashes are completely random, ensuring that each possibility has an equal probability. (spoiler it's not see section:V)

On the other side of the spectrum, we now explore what happens with pre-computing all the passwords, and storing them in disk. Assume that, for each password, we store in a database the entry (password, hash)

D. Bits of MD5 and passwords

4 - How many bits is a MD5 hash? How many bits is a password consisting of 5 ASCII characters?

a) MD5 Hash: An MD5 hash is a 128 bit value which is represented as a 32 hexadecimal string in the code.

b) Password of 5 ASCII letters: For ASCII characters the answer is 1 byte, so 8 bits so the answer would be $5 \cdot 8 = 40$bits.

For ASCII letters each character in the password has 6 bits of entropy because 2^6 is equal to 64, which covers all the 52 possibilities. A password consisting of 5 ASCII letters has $5 \cdot 6 = 30$ bits of entropy.

This is however still not really the space our program will actually use, python will store prefixed the length of a string which on 64-bit systems is typically a 64 bits integer, python uses by default UTF-8 encoding, which for ASCII characters will use 1 byte. So in our actual program it would be $5 \cdot 8 + 64 = 104$ bits per password.

E. How many bits of memory

5 - If you were to store the (password, hash) combinations for all passwords of 5 characters over S, how many bits of memory would you need?

A password consisting of 5 ASCII letters is 30 bits of entropy (here we are over S as the question implies) and a md5 hash is 128 bits.

So, for each (password, hash) combination:

$$memoryper = 30bits + 128bits = 158bits \text{ per combi.} \quad (3)$$

$$Memory = possibilities \cdot memoryper \quad (4)$$
$$Memory = (52^5) \cdot 158bits = 6.007 \cdot 10^{10}bits$$

This result is around 7.5 GB which represents an enormous amount of memory to store all (password,hash) combinations.

II. HELLMANN TABLES

We will now explore a solution that uses a time-memory compromise, also known as Hellmann tables (later refined as rainbow tables). The generic idea has been exposed in class. The goal of a generative function over a set S is, given an entry, to generate an output that belongs to the set S. Ideally, all elements of S have equal probability of being generated.

A. Helper implementations

1 -

a) Implementation - convert to base: When n is equal to 0, the conversion to any base results in a list containing only the digit 0 thus `[0]`.

For values of n that require conversion, you can obtain the digit at each position by calculating the remainder of n divided by the base. This remainder represents the digit's value. After that, it performs a floor division of n by the base to progress to the next position. This process is repeated iteratively until n becomes 0 meaning n as been converted

b) Implementation - generate from indexes: Given an integer representation in the base of ASCII letters, the function maps each index in the input list, `n_repr`, to the corresponding ASCII letter. This forms a string, with each element of `n_repr` representing an ASCII letter directly taken from the alphabet list `S[i]`.

c) generator(h) returns: To answer this question I just printed out the result of using the function to get the next password in the Hellman table.

```
print(generator(md5("unilu")))
--Output: dlbSz
```

Listing 1. getting next password in the Hellman table

A hash chain is created in the following way: password0 → hash0 → password1 → hash1 → password2 → hash2... It is of length n if it contains n passwords. We currently assume that we go from a hash to the next password using the generative function generator. In this example, md5(password0) returns hash0, generator(hash0) returns password1, md5(password1) returns hash1, generator(hash1) returns password2 and so on.

B. The Hellman table

2 - Implement a function hash_chain(length, start) that takes as input a length, an initial password start, and returns a list that represents a hash chain of size length, starting from the password start. What does hash_chain(3, "unilu") return?

a) Implementation - Hash chain: The function, `hash_chain`, creates the resulting chain by alternating between passwords and their respective MD5 hashes, calculated in succession. First the hash then the password for each iteration because we start with a password, Thus for the last iteration no generation of a new password. This process iterates until the specified chain `length` is reached.

b) Specific return: Performing a `print()` returns the following hash chain:

```
print(hash_chain(3, "unilu"))
--['unilu', '0004c8fa1794f7d7679276acc2124469', '
    dlbSz', 'd99e0384bb1d1bdb267324fd599f37ca', '
    cjtmf', '34cab79314134808724e8831622be267']
```

Listing 2. list return performing a hash chain

The result is as expected with `'dlbSz'` being the second password in the Hellman table.

C. Comparisons

3 - Compare the hash chains hash_chain(5, 'cVVWr') and hash_chain(5,'DrWIW'). What do you observe?

```
--['DrWIW', '65f158c061bb8b9b67aeb5e4ba7c0862', '
    cmoFE', '1fa3e9ac6760068e118ee35937413ddb', '
    cQVPN', '76f675a599559dd4ece78ce15c6bab7d', '
    bhlXe', '0d98189ab0e4c6e641595cd707072d3c', '
    fHGdI', 'bb4bda4d8299be12ef2e5d45a8ad35d2']
--['cVVWr', '1db2c0588d3fb65d73b24d9620f2eb92', '
    cASye', '465d7668d2f976b9c8965ab7bcd39579', '
    cmoFE', '1fa3e9ac6760068e118ee35937413ddb', '
    cQVPN', '76f675a599559dd4ece78ce15c6bab7d', '
    bhlXe', '0d98189ab0e4c6e641595cd707072d3c']
```

Listing 3. output performing the provided hash chains

Even though the initial passwords are different, they eventually converge to the same MD5 hash ('1fa3e9ac6760068e118ee35937413ddb') and then the chain is the same just pushed one iteration forward for the second hash chain compared to the first. They converge.

For some MD5 hashes, the converted representations lead to the same indexes in the S set, resulting in the same 5 letter password.

This seems intuitive enough because the `convert_to_base` function takes a value in a 128 bit hash form with 2^{128} possibilities if and brings it into a value which as 5^{25} possibilities a way smaller set. It is therefore possible to find a collision. When two different paths end up with the same representation in the base S, they will generate the same 5 letter password. This is because the input into the `md5()` hash will be the same in the current `generator` function as only '0' is added to the password to hash.

Once a collision is found the hash chain will converge, for chains long enough they will always at some point collide and converge.

III. RAINBOW TABLES

Rainbow tables have the additional requirement that the generating function is different at each step of the chain. From now on, we will not use the function generator, but the function generator_i(position, h), where the generated password now depends on the position of the hash within the hash chain. From now on, a hash chain (or rainbow chain) will be created in the following way: password0 → hash0 → password1 → hash1 → password2 → hash2... where md5(password0) returns hash0, generator_i(0, hash0) returns password1, md5(password1) returns hash1, generator_i(1, hash1) returns password2, and so on

A. The Rainbow chain

1 - Implement a function rainbow_chain(length, start) that generates a rainbow hash chain. What does rainbow_chain(3, 'unilu') return?

a) Implementation of the rainbow chain: Well this is exactly the same as the previous hash_chain function but we are just replacing the generator with the new already given one.

The function is just passing in i into the `generator_i(position: int, h: str)` function.

b) Specific return: Performing a `print()` operation gives out:

```
--['unilu', '0004c8fa1794f7d7679276acc2124469', '
   dlbSz', 'd99e0384bb1d1bdb267324fd599f37ca', '
   ehnfu', '7f3183fa4cac9c63c54c75a10f571860']
```

Listing 4. output performing the rainbow chain

As we can see after the next password the hash changes because we are adding the position to the input string and this is making the hash be different as it is now dependent to the position in the rainbow table.

B. Comparisons

2 - Compare the rainbow hash chains generated by rainbow_chain(5, 'cVVWr') and rainbow_chain(5, 'DrWIW'). What do you observe and how is it dif- ferent from Q2.3?

```
--['DrWIW', '65f158c061bb8b9b67aeb5e4ba7c0862', '
   cmoFE', '1fa3e9ac6760068e118ee35937413ddb', '
   bJaum', 'd6eb7b2aff15a10421473cffdd5561b3', '
   nYWyG', 'f4cd6e584b8cef3a68c031d9c319063c', '
   eneYz', '15f3eb9349f060ac351b7be6deab0b1f']
--['cVVWr', '1db2c0588d3fb65d73b24d9620f2eb92', '
   cASye', '465d7668d2f976b9c8965ab7bcd39579', '
   fmzwt', '06024e5ddc683af214ef5b199d18f56c', '
   fYMyi', 'fd34b500f4abeaee17faef9f04a5965e', '
   fgwLs', 'cf25bec92a6eb776d1209a0f304c989a']
```

Listing 5. output performing the provided rainbow chains

After the first iteration, the chains generate different hashes and always different passwords. It is still theoretically possible that we find equal hashes or passwords somewhere in a chain long enough because all possible combinations or permutations of 128 bits can be generated through these successive hashes. However the following chain will not be the same as the new generator now uses the position in the chain as a hashing factor to differentiate them. There are only unique chains.

The significant observation here is that, unlike the previous hash_chain, the rainbow chains whatever the length will never converge.

C. Advantage of rainbow chains

3 - In your words, and based on the previous question, what would be the advantage of using rainbow chains compared to Hellmann hash chains seen in Q2.3?

Because the rainbow chains never completely converge we don't have to store some useless chains that converge quickly this covers more (password,hash) combinations per chain and saves memory. Rainbow chains are more space-efficient and storage-friendly than their Hellman hash chains counterpart.

D. Using the rainbow chain

4 - Assume that from a rainbow chain, we only store the first and last element. For instance, given the chain password0 → hash0 → ... → passwordN → hashN , we only store (password0, hashN).

a) Retrieving from hash: My basic idea is to reconstruct the specific rainbow chain and look for a specific hash in it. If the hash corresponds to a password in the chain, the original password is returned. The loop iterates backward through the chain, checking each hash to find the corresponding password.

```python
def retrieve(first_password, hash_of_pass):
  chain = rainbow_chain(10, first_password) #
    reconstitute the chain

  if chain[-1] == hash_of_pass: # base case
    return chain[-2]  # first password

    # Go back to find the password
  for i in range(10 - 2, -1, -1):
    next_hash = md5(chain[i * 2 + 1])
    if next_hash == hash_of_pass:
      return chain[i * 2]  # Return the
    corresponding password
  return "Password not found"

print("Retrieved Password:", retrieve('unilu', '523
    ffd2979b4067a80de673dfc270b70'))

-- Retrieved Password: KyjBk
```

Listing 6. retrieving password in rainbow chain *b) Implementation - is at position:* The function plays a crucial role in rainbow table based password retrieval. It determines whether a given hash is located at a specific position within a rainbow chain.

We have a tuple given which stores the chain first password and last hash. We can start by checking the case in where the hash tested is in in the last position and return `True` for a fast lookup.

Otherwise the only way to find out if the hash is at a certain position is to reconstruct the chain as we can't construct back the password that creates a hash, the goal of a hash function being that the input is impossible to retrieve.

Here the chain is constructed 2 by 2 without theoretically appending each password and hash as it seems that's what the `assert()` statement position implies. At each iteration until the position we reconstruct the rainbow chain by creating the new hash and the next password starting at the first password. Once at the position we can then check if the hashes are matching, and it is done.

c) Implementation - find position of hash: The primary objective of this function is to efficiently determine whether the given hash exists in a rainbow chain and, if so, at which position. The position in the given function is not really the position in the rainbow chain as implemented before but more so in a list of only hashes or the position of a (password,hash) tuple this as been determined from the given example `assert()` statements.

As we have the last_hash stored we can first check if the hash tested is equal to this last hash and quickly return the chain `length - 1` as position.

Otherwise the function iterates through the chain, starting with the first password, and repeatedly computes and compares hash values along the creation of the chain. If the function encounters a hash that matches the target hash, it returns the index or position of that hash within the rainbow chain which is not really the position in a chain computed with `rainbow_chain`.

d) Implementation - find password of hash: Here similarly to the other function we can use the last hash to avoid comparing each hashes with the hash to test in this case we can just construct the rainbow chain using `rainbow_chain` and take it's second last element which will be at the `chain_length*2-2`.

Other wise we recreate the chain 2 by 2 by iterating through it and test the given hash once this is found we still have in storage the previous password and we can return it.

A rainbow table consists of many different chains of the same length. We won't implement it since this assignment is already long enough, and several tricks are still required to make it efficient.

IV. Further Considerations

A. Length of rainbow table

1 - The literature says that if you want to store N passwords in your rainbow table, you should have $N^{2/3}$ chains of length $N^{1/3}$. How many chains (of which length) should your table have?

We have 52^5 possible password thus using the formula given by the literature we need:

$$\mathit{chains} = (52^5)^{\frac{2}{3}} = 524821 \qquad (5)$$
$$\mathit{length} = (52^5)^{\frac{1}{3}} = \lceil 724.445 \rceil = 725$$

Here we round up as the chain can't be cut.

B. Memory usage

2 - If for a rainbow chain password0 → hash0 → ... → passwordN → hashN , we only store (password0, hashN), how much disk space are you expecting your rainbow table to take? Show your math. How much smaller is that compared to storing all hashes (see Q1.5)?

If the length of the rainbow chain does not count we can just use the $memoryper$ calculated previously multiplied by the number of chains:

$$MemoryR = 158\mathit{bits} \cdot 524821 = 82921718 \qquad (6)$$

This is equivalent to around 10.365 megabytes or 0.01 GB. The amount of bits needed is $\frac{Memory}{MemoryR} = 724.445 \approx 724$ times less.

This is not actually the disk value because of python and the computer architecture as explained previously, it is the value used as this question asks for comparison with Q1.5.

C. Code complexity

3 - What is the average complexity (in function of N) of the function find_password_of_hash()?

```python
def find_password_of_hash(chain: tuple, chain_length
    : int, test_hash: str) -> Union[None, str]:
  (first_password, last_hash) = chain


  if last_hash == test_hash:
    c = rainbow_chain(5, first_password)
    return c[chain_length*2-2]


  for i in range(chain_length-1):  # i here is the
    position
    if md5(first_password) == test_hash: # current
    hash == tested hash
      return first_password

    first_password = generator_i(i, md5(
    first_password)) # next password
  return None
```

Listing 7. find password of hash

For this function we consider the complexity when the hash of the password to retrieve is in the chain. We do not consider the table aspect.

In my implementation when the `test_hash` is the last one then I reconstruct the rainbow chain using the function made in previous exercises this function is always of complexity `chain_length`. This is in fact the worst password to retrieve from the chain because for all passwords before we don't have to build the entire chain. As we follow the theory and results of question 1, we know that the worst case complexity is of $O(N^{\frac{1}{3}})$ which is equivalent to $O(\sqrt{N})$ because $N^{\frac{1}{3}} = \sqrt[3]{N}$ and the cube doesn't matter from a complexity standpoint.

If we're in the right chain and the password to retrieve is the first the complexity which is the best case will be of $\Omega(1)$ as the loop will only iterate once no matter the number of possible passwords. But the loops always iterates to the right position which is `chain_length - k` with $k > 1$. So the worst case complexity is $N^{\frac{1}{3}} - k$, similarly to above we can conclude that the worst case complexity of the function is of $O(\sqrt{N})$. Here the average case complexity will also be of order $\theta(\sqrt{N})$ as k is just a constant.

V. Extra credit

(This question is difficult) Justify that generator can generate any possible 5-letter passwords. Do all passwords have equal probability of being generated by generator? If no, can you quantify the bias? You may model MD5 as a uniform random function.

a) Possible 5 letters passwords:

Some passwords are never reached and the probability of each passwords is definitely not the same.

In this question as MD5 is a random uniform function and S is an uniform set containing the ASCII letters and the functions converting to base and generating paswords from a list of indexes are both correct the problem comes from the generator function:

```python
1  def generator_i(position: int, h: str) -> str:
2    """
3    From a hash, generate a 5-letters password with
       letters from S
4    The password will depend from the position value
5
6    :param h:        a MD5 hash
7    :param position:  a position (in the hash chain)
8    """
9    to_hash = str(position) + h
10   generated_value = md5(to_hash)
11   value_in_base_S = convert_to_base(int(
       generated_value, 16), len(S))
12   return generate_from_indexes(value_in_base_S)[:5]
```

Listing 8. Problematic function

First of of all a comment we are hashing twice the input password because of the requirement of the `generator_i` function given which takes a hash as input and this is the way to get the right result for the `assert()` test under. This is however still maintaining the uniformity of the hash, so the solution is not there.

The problem in this generator comes from line 12 in the above code:9.

We are truncating the output of the function generating from indexes taking only the first five elements. This means that every time that `value_in_base_S` contains more than 5 elements some part of the information is discarded. Also the function doesn't handle the case where the integer representation of the hash makes an array of less than 5 elements which happens when the integer is less than around 10 million (but this almost has no chance of happening).

By this point I actually got curious so I created a very large hash chain and wrote the output of `generate_from_indexes(value_in_base_S)[:5]` to a file called "generated_passwords.txt" then analysed that file in the following using a dictionnary in the proba.py file:

```python
def analyze_password_probabilities(file_path):
    password_counts = {}
    total_count = 0

    # Read passwords and count frequencies
    with open(file_path, 'r') as file:
        for line in file:
            password = line.strip()
            if password:
                password_counts[password] =
    password_counts.get(password, 0) + 1
                total_count += 1

    # Find the key with the maximum value (most
    repetitions)
    max_key = max(password_counts, key=
    password_counts.get)

    # Get the count of this key (number of
    repetitions)
    max_count = password_counts[max_key]

    print(f"The most frequent password is '{max_key
    }' and it appears {max_count} times.")
    print("The probability over the set is",
    max_count / total_count)

    # Dictionary to hold the frequency of each count
    frequency_of_counts = {}

    # Counting the frequency of each count
    for count in password_counts.values():
        if count in frequency_of_counts:
            frequency_of_counts[count] += 1
        else:
            frequency_of_counts[count] = 1

    # Number of passwords that appear exactly twice
    num_passwords_twice = frequency_of_counts.get(2,
     0)
    print(num_passwords_twice)
```

Listing 9. proba.py

Turns out with only 1/300 here around 1.3 million of the possible possible passwords I get around 20k collisions with 2 passwords repeated, my probability of collision is around 2% which seems to give me reason the probability should be much much lower than 2%. I even get triple collisions.

Thus I conclude that every passwords does not appear and every passwords don't have the same probability of being generated.

b) Quantifying Bias:

As we know from before the colision number is relatively high we would expect it to be $> 0.33\%$ which is not the case it is 6 times higher thus the bias would be of around 600%.

I had fun with this I probably made some mistakes I don't really know what is the bias but did my best. Thank you for providing this fun assignment :) (ps: sorry for the length)