The reports (a small PDF for some question + your code) should be posted on Moodle before November 12th, 23:59. You can use internet, but not copy-paste code, either from internet (including ChatGPT) or from your classmates.

The sections are mostly independent. Section 3 relies on question 2.1.

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()`.

1 Comparison with bruteforce

1. How many passwords of 5 characters can exist over S?

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

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?

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

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)`.

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

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?

2 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.

1. (a) Implement a function `convert_to_base(n, b)` that takes as inputs a number `n` and a base `b`, and returns the list representation of `n` in base `b`. See example in the python file (keep the same output format!)

   (b) Implement a function `generate_from_indexes(n_representation)` that takes as input a number `n_representation`, written as a list representation in base `len(S)`, and returns a string such that the i-th element of the list is the index in `S` of the i-th letter of the output string. See example in the python file.

   (c) In the python file, we have a function `generator(h)` that takes as input a MD5 hash and returns a 5-characters password with letters from `S`, and relies on the two previous functions. What does `generator(md5('unilu'))` return?

A hash chain is created in the following way: $\mathsf{password}_0 \rightarrow \mathsf{hash}_0 \rightarrow \mathsf{password}_1 \rightarrow \mathsf{hash}_1 \rightarrow \mathsf{password}_2 \rightarrow \mathsf{hash}_2$... 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(`$\mathsf{password}_0$`)` returns hash_0, `generator(`hash_0`)` returns $\mathsf{password}_1$, `md5(`$\mathsf{password}_1$`)` returns hash_1, `generator(`hash_1`)` returns $\mathsf{password}_2$ and so on.

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?

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

3 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: $\mathsf{password}_0 \rightarrow \mathsf{hash}_0 \rightarrow \mathsf{password}_1 \rightarrow \mathsf{hash}_1 \rightarrow \mathsf{password}_2 \rightarrow \mathsf{hash}_2$... where `md5(`$\mathsf{password}_0$`)` returns hash_0, `generator_i(0,` hash_0`)` returns $\mathsf{password}_1$, `md5(`$\mathsf{password}_1$`)` returns hash_1, `generator_i(1,` hash_1`)` returns $\mathsf{password}_2$, and so on.

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

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 different from Q2.3?

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?

4. Assume that from a rainbow chain, we only store the first and last element. For instance, given the chain $\mathsf{password}_0 \rightarrow \mathsf{hash}_0 \rightarrow ... \rightarrow \mathsf{password}_N \rightarrow \mathsf{hash}_N$, we only store $(\mathsf{password}_0, \mathsf{hash}_N)$.

   (a) The rainbow chain of size 10 `('unilu', '523ffd2979b4067a80de673dfc270b70')` is stored in your database. How would you retrieve the password giving the hash `'523ffd2979b4067a80de673dfc270b70'`? Show your code.

   (b) Implement a function `is_at_position(chain, chain_length, test_hash, position)` that takes as input `chain = (password_start, hash_end)` that corresponds to a rainbow chain, the chain length, a hash and a putative position. The function should return `False` if the hash is not in the rainbow chain at that position, and `True` otherwise.

   (c) Implement a function `find_position_of_hash(chain, chain_length, test_hash)` that returns the position of the hash in the rainbow chain, or `None` if the hash is not in the chain.

   (d) Implement a function `find_password_of_hash(chain, chain_length, test_hash)` that returns the password that hashes to `test_hash` if the hash is in the rainbow chain, and `None` otherwise.

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.

4 Further considerations

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?

2. If for a rainbow chain $\mathsf{password}_0 \rightarrow \mathsf{hash}_0 \rightarrow ... \rightarrow \mathsf{password}_N \rightarrow \mathsf{hash}_N$, we only store $(\mathsf{password}_0, \mathsf{hash}_N)$, 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)?

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

5 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.