Exercise 9.3

1. Our algorithm has several steps.

   a. First, since we have `h(pw)`, we can get $F^1(pw)$ = `g(h(pw))`. Then we do F n times to get a list L = [F^1(pw), F^2(pw), ..., F^n(pw)].

   b. For i = 1 to t, we will find out if y_i is in list L. Fortunately, if y_i == F^k(pw) == L[k], we can conclude that the password is located in this row i.

   c. We calculate M[i, n - k], which is $F^{n-k}(pw_i)$, if there is no conflict, this value is equal to pw. (According to the question, In matrix M, i starts from 1, and index of column starts from 0).

2. In step 1.a, to get list L, we compute F n times. In step 1.c, to get M[i, n - k], we calculate F (n - k) times. So we compute F (2n-k) times in total.

3. If not all y_i 's are distinct, we might be able to find several different candidates which might be correct password. They are located in different rows.

   To ensure all y_i 's are distinct, we need to ensure that our function `g(x)` and function `g(h(x))` is collision resistent, which means (ideally) there's no same output given different input.

Exercise 9.11

1. To do this job, we define a structure called `KeyFile`, which has two fields, `PriKey` is the encrypted private key. `H` is the hash value of password. We will store this structure to file. If the attacker can get keyfile, he need to reverse the hash function, which is costly.

2. We implement the function `Generate` in `CryptoModule/AES`. First, we generate the private key and public key (we have pre-defined `PrivateKey` and `PublicKey` structure). Then we use input password as AES key to encrypt the (marshalled) private key, and use SHA256 to calculate the hash value of password. We insert the ciphertext and hash value into `KeyFile` struct, then marshall it and save it to the file.

In `Sign` function, we unmarshal the bytes to `KeyFile` object, then we calculate the hash value of input password, and compare it with `KeyFile.H`, if the password is wrong, we simply return the "Wrong password". If it is correct, we use private key to sign the message and return the signed value.

3. The reason why we encrypt the private key is that in this way, once attacker have access to the keyfile, they cannot directly use the private key to sign anything.

   The reason why we also store the hash of password is first, it cannot be reversed, and we can use this value to quickly check if the input password is correct or not.

4. We have a test case in main.go. First, we generate a random password of 32 bytes(this is required since password are used as a AES key), then we call the `Generate` function to generate key pair and save the necessary information to file. At last we call `Sign` function, we can use the same password and it will print `Password Correct!`, and return the signature. Otherwise, it output `Wrong Password!`, and return -1. We also create a verify function called `VerifyMessage`, which can be used to verify if the signature is valid. A more practical system should take any length of password as input, we can use padding to do this in the future.

5. Enter the `CryptoModule` folder and type `go run main.go`.