Homework 2 - Prehistoric to Modern Crypto

Cryptography and Security 2021

- You are free to use any programming language you want, although SAGE is recommended.
- Put all your answers **and only your answers** in the provided SCIPER-answers.txt file. This means you need to provide us with all `Q` values specified in the questions below. You can download your **personal** files from the following link: `http://lasec.epfl.ch/courses/cs21/hw2/index.php`
- You will find an example parameter and answer file on the moodle. You can use this parameters' file to test your code and also ensure that the types of `Q` values you provided match what is expected. For instance, the variable `Q1a_c` is encoded in base 64, whereas `Q2_p` is an Integer. **Please do not put any comment or strange character or any new line** in the .txt file.
- We also ask you to submit your **source code**. This file can of course be of any readable format and we encourage you to comment your code. Notebook files are allowed, but we prefer if you export your code as a textfile with a sage/python script.
- If you worked with some other people, please list all the names in your answer file. We remind you that you have to submit your **own source code** and **solution**.
- We might announce some typos/corrections in this homework on Moodle in the "news" forum. Everybody is subscribed to it and does receive an email as well. If you decided to ignore Moodle emails we recommend that you check the forum regularly.
- The homework is due on Moodle on **Sunday the 31st of October** at 23h59.

Exercise 1 Follow the line

In this exercise, we are interested in assessing the security of a large family of cryptographic schemes called affine ciphers (the Caesar cipher belongs to that family). The set of integers modulo n is denoted by $\mathbf{Z}_n$ and its group of units is denoted by $\mathbf{Z}_n^\times$.

Given an alphabet $\mathcal{A}$ of size n, an indexation function $\varepsilon\colon \mathcal{A} \xrightarrow{\simeq} \mathbf{Z}_n$ is a bijection such that both ε and ε^{-1} are efficiently computable. Consider now the following encryption algorithm defined over the message space $\mathcal{A}$ and keyspace $\mathcal{K} = \mathbf{Z}_n^\times \times \mathbf{Z}_n$. For messages $M \in \mathcal{A}^*$ of length $\ell \geq 2$ and a key $K \in \mathcal{K}$, the encryption $C = \mathsf{enc}_K(M)$ of M is the concatenation of the encrypted characters, that is $\mathsf{enc}_K(m_1||\ldots||m_\ell) = \mathsf{enc}_K(m_1)||\ldots||\mathsf{enc}_K(m_\ell)$.

Algorithm 1: Affine Cipher Encryption (1-character)

```
  Data: An alphabet $\mathcal{A}$ of size $n$ and an indexation function $\varepsilon\colon \mathcal{A} \longrightarrow \mathbf{Z}_n$.
  Input: A plaintext $m \in \mathcal{A}$ and a key $K = (a, b) \in \mathbf{Z}_n^\times \times \mathbf{Z}_n$.
  Output: A ciphertext $c \in \mathcal{A}$.
1 $x \leftarrow \varepsilon(m)$
2 $y \leftarrow ax + b \bmod n$
3 $c \leftarrow \varepsilon^{-1}(y)$
4 return $c$
```

From now on, we assume that $\mathcal{A}$ is the usual English case-sensitive alphabet together with digits and some punctuation symbols. More precisely, the alphabet consists of $n = 100$ characters and the indexation ε is given by the following Python script:

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import string

# A = '0123456789'
# A += 'abcdefghijklmnopqrstuvwxyz'
# A += 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
# A += '!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
# A += ' \t\n\r\v\f'
A = string.printable
assert len(A) == 100

f = lambda c: A.index(c) # indexation function
g = lambda n: A[n]       # reverse indexation
assert [g(n) for n in [f(c) for c in A]] == list(A)
```

The Python `base64` module contains the `b64encode` and `b64decode` functions to encode `bytes` objects in base64. Other alternatives include the `binascii` module or the `pwntools` package.

▷ Let $M \in \mathcal{A}^*$ be a plaintext whose base64 encoding is `Q1a_M` and let $K = (a, b) \in \mathcal{K}$ be a secret key given as `Q1a_K`. Compute the ciphertext $C = \mathsf{enc}_K(M) \in \mathcal{A}^*$ and report its base64 encoding in the answers file under `Q1a_C` as a Python `str` object (not `bytes`).

- ▷ Let $C \in \mathcal{A}^*$ be a ciphertext whose base64 encoding is `Q1b_C` and let $K = (a, b) \in \mathcal{K}$ be a secret key given as `Q1b_K`. Compute the plaintext $M = \mathsf{enc}_K^{-1}(C) \in \mathcal{A}^*$ and report its base64 encoding in the answers file under `Q1b_M` as a Python `str` object (not `bytes`).

Let L be any language (e.g. English) using the letters from the alphabet $\mathcal{A}$. A *frequency table for* L is a mapping $T = (t_\alpha)_\alpha$ which maps $\alpha \in \mathcal{A}$ to some $0 \leq t_\alpha \leq 1$ representing the frequency of α in "meaningful" texts. For instance, in French, $T[\mathsf{e}]$ or $T[\mathsf{r}]$ would be large, whereas $T[\mathsf{w}]$ or $T[\mathsf{y}]$ would be small.

- ▷ Let $C \in \mathcal{A}^*$ be a ciphertext whose base64 encoding is `Q1c_C` and let T be an incomplete frequency table given as a Python dictionary `Q1c_T` (only the most common symbols of the plaintext are considered). Recover the corresponding plaintext $M \in \mathcal{A}^*$ and secret key $K = (a, b) \in \mathcal{K}$. Report in the answers file the base64 encoding of M under `Q1c_M` as Python `str` object (not `bytes`) and the secret key under `Q1c_K` as a tuple of integers (e.g. `Q1c_K` $= (1, 2)$). In order to avoid multiple solutions, a SHA-256 checksum `Q1c_H` has been generated according to the following algorithm:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import hashlib as _

def checksum(*args):
    data = ';'.join(map(str, args)).encode()
    return _.new('sha256', data=data).hexdigest()

# >>> checksum('hello world', 1, 2)
# 9ba8c7cbde19762e0a5583889ee0c9a2547a19bbbb54aac24a4d65f5157d3d23
#
# >>> import base64
# >>> M = 'hello world'
# >>> Q1c_M = base64.b64encode(M.encode()).decode()
# >>> Q1c_K = (1, 2)
# >>> Q1c_H = checksum(Q1c_M, Q1c_K[0], Q1c_K[1])
# cd08a0bf12085af985e1c603ec8adf52cc66bb1e9b3fdca0815f34f019985a2a
```

Exercise 2 Secret messaging among friends

We consider a group of n cryptographically-minded people $p_1, ...p_n$ sitting together who wish to communicate anonymously. In particular, we assume that each person possibly has a message to convey. To begin with, we assume that at most *one* person conveys a message (i.e. either one person does, or nobody does).

For the sake of explanation, we first consider the case where the message space is a singleton element, i.e. of the form $M = \{1\}$. Assume that parties initially somehow each initially have a secret bit k_i such that $\bigoplus_{i=1}^n k_i = 0$. More specifically, we assume that k_i, $i = 1, ..., n-1$ are uniformly distributed and $k_n = \bigoplus_{i=1}^{n-1} k_i$.

To communicate, each party p_i then sends to all parties $c_i = k_i$ except for possibly the party that wishes to communicate a message 1 who sends $c_i = k_i \oplus 1$. Note that all parties send a value c_i regardless of whether or not they send a *message*. After all n values are sent between every person, each person computes $m' = \bigoplus_{i=1}^{n} c_i$ and then either outputs 1 if $m' = 1$, denoting the fact that one party communicated 1, or otherwise the special value $\perp$, denoting the fact that nobody communicated.

Let $(G, \times)$ be a group with identity 1. Consider the following generalisation of the aforementioned algorithm. Each person p_i stores a secret value $k_i \in G$ such that $\prod_{i=1}^{n} k_i = 1$. To send a message $m \in G$, i.e. a group element, p_i computes and sends the value $c_i = k_i \cdot m$ to all people. People who don't send a message compute and send the value $c_i = k_i$. Once all values are exchanged, each party computes $m' = \prod_{i=1}^{n} c_i$ and outputs m' if $m' \neq 1$ and $\perp$ otherwise.

In your parameters file, you are given the public parameters of such a group $G = (p, q, g)$ as `Q2_p, Q2_q, Q2_g` in `Integer` format, and $n = 10$ ciphertexts c_i as `Q2_i`, $i \in [1, 10]$ in `Integer` format.

▷ Your first task is to determine whether a message was sent or not. If there is a message, report in the answers file the message under `Q2_msg` characters by first converting from an Integer value (e.g. `Q2_msg = 1234`). If you determine that no message was sent, output the four-letter string `NONE` under `Q2_msg` (i.e. output `Q2_msg = NONE`).

Let us now lift the restriction that at most one person sends a message. That is, given a set of n people, any number of messages m where $0 \leq m \leq n$ can be sent.

Suppose that a group of n execute the protocol with respect to $G = (p, q, g)$, and then a large amount of time passes. Unfortunately, we do not know exactly how many people who participated in the protocol decided to communicate a message. The only trace of the protocol execution that remains is as follows:

- The value $m' = \prod_{i=1}^{n} c_i$ computed by one of the parties after receiving all n ciphertexts c_i but before outputting a message.
- The information that the message space is of the form $M = \{g^{2^0}, g^{2^1}, g^{2^2}, ..., g^{2^{20}}\}$.
- The information that no more than 2^{32} people participated in the protocol execution.

You are given m' as `Q2_msgdash` in `Integer` format.

▷ Your task is to determine the minimum and maximum number of messages that could have possibly been sent during the execution of some n parties. Report in the answers file the minimium value under `Q2_min` and the maximum value under `Q2_max` in `Integer` format (e.g. `Q2_min = 1` and `Q2_max = 2`).

Exercise 3 Elgamal, Algemal, Algemel, Elgemel I don't know anymore

After learning about the Elgamal cryptosystem last week, our crypto apprentice has decided to make use the plain Elgamal cryptosystem to encrypt a large text. Let us assume the plaintext consists only of lowercase characters 'a-z', space character and '.'. The crypto apprentice decides to encrypt the plaintext letter by letter. In order to keep things secure, he decides to use the `SSH2` group parameters (p, q, g).

After seeing Bitcoin's incredible idea of not having your public key available, and instead only publishing a hash of it, he decides to encrypt a plaintext $pt = p_1, \ldots, p_m$ with a secret key s, and release his hashed public key SHA256(g^s) to the public. Hence ideally our crypto apprentice should pick a random value r_i for each $i \in [m]$ and compute the ciphertext $[c_i = (g^{r_i}, p_i g^{r_i s})]_{i \in [m]}$.

As the crypto apprentice is too lazy to sample r_i values uniformly at random, he decides to do something **better**! Each time he is encrypting some plaintext, he would sample r_1 uniformly at random and for some value $u \in \mathbb{Z}_q$, he sets $r_i = u^{i-1} r_1$.

In your parameters file you are given:

1. the public parameters (p, q, g) as `Q3_p, Q3_q, Q3_g` in `Integer` format.
2. a dictionary `Q3_dict` indicating what group element each character is mapped to.
3. the hash of the public key `Q3_H` in `ASCII` format.
4. the ephemeral key update value u in `Integer` format `Q3_u`.
5. a cipher text $(c_1, \ldots, c_m)$ as an array `Q3_ct` of tuples of form (R_i, C_i) both written in `Integer` format.

Your goal is to recover the plaintext $pt = p_1 p_2 \ldots p_m$ and record it in your answers file under `Q3_pt` surrounded by quotes (e.g. `Q3_pt = "this is my answer"`). Note that the plaintext is an almost meaningful sentence.

The hashing is done using the following method.

-hf = hashlib.sha256();
-f.update(str(pk).encode('ASCII'));
-digest = f.digest();