Homework 3

Due Thursday, September 23

Implementation Part

This week we finally get to implement a cryptosystem. Recall that this is a collection $(\mathcal{M}, \mathcal{C}, \mathcal{K}, d, e)$ where $\mathcal{M}$ is the space of message text, $\mathcal{C}$ is the space of cipher text, $\mathcal{K}$ is the space of keys, $e : \mathcal{M} \times \mathcal{K} \to \mathcal{C}$ is the encryption function and $d : \mathcal{C} \times \mathcal{K} \to \mathcal{M}$ is the decryption function. Our message space will be often something like $\mathcal{M} = \mathbb{Z}/N\mathbb{Z}$, that is, it is a collection of numbers. But we want to send messages that consist of text, letters, punctuation, the like. In order to do this we must develop a dictionary between strings and integers called an *encoding scheme*. We will use the ASCII encoding scheme.

1. We start with a warmup. This part won't be graded, but may be useful for those without much programming experience. Parts (a) and (b) are useful tricks for string manipulation, and (c) and (d) introduce the functions which do the ASCII translation.

   (a) Python/Sage reads strings like lists. Try running the following code to see what I mean:

   ```
   x = "Hello World!"
   for i in x:
       print(i)
   ```

   (b) You can use the `+` operation to add characters to a string. Run the following code to see what I mean:

   ```
   x = "Hello"
   y = x + '!'
   print(y)
   ```

   (c) ASCII is a dictionary between characters and bytes. Recall that a *byte* of data is 8 *bits*, that is 8 digits in binary, or equivalently, a number n such that $0 \leq n < 2^8$. In particular, each character corresponds to a unique integer $0, 1, ..., 255$. In python, you can turn a character into its corresponding integer using the function `ord`. Try this out a bit by running the following lines of code:

   i. `ord('a')`
   ii. `ord('A')`
   iii. `ord(' ')`
   iv. `ord(',')`
   v. `ord(a)` (what happened here?)
   vi. `ord("hello")` (what happened here?)

   (d) `ord` has an inverse, which takes as input a byte (that is, a number $0, 1, ..., 255$) and returns a character. In python this is `chr`. Try it out:

   i. `chr(98)`
   ii. `chr(40)`
   iii. `chr(ord('F'))`
   iv. `ord(chr(201))`

2. Now we build our translation functions. The idea is simple enough: via `ord` a string is just a sequence collection of bytes, which we can think of as a number in base 256. Therefore translating an string to an integer and back should can be thought of as computing the number from this base 256 representation:

   (a) Build a function `textToInt(w)` whose input is a string `w`: a list of characters $c_0c_1c_2c_3...c_r$ and whose output is an integer n satisfying:

   $$n = \sum_{i=0}^{r} ord(c_i) * 256^i.$$

   The inverse then consists of finding the base 256 expansion n, and then converting each 'digit' (which is a byte, i.e., a number 0,...,255) into its corresponding letter using `chr`:

   (b) Build an inverse to the function from part (a). Call it `intToText(n)`, it should take as input a positive integer n and return a string. **Hint**: If you are stuck try the following two steps:

   (1) Compute the base 256 expansion $[B_0, B_1, ..., B_r]$ of n. This means each B_i is an integer between 0 and 255, and

   $$n = B_0 + B_1 * 256 + B_2 * 256^2 + \cdots + B_r * 256^r.$$

   You should be able to do this with minor modifications to your `getBinary` code from Homework 2.

   (2) Return the string $c_0c_1c_2...c_r$ where $c_i = chr(B_i)$.

   Once you have done these two steps, streamline it so that you dont have to remember the list $[B_0, ..., B_r]$ which can in practice be quite large. This should follow a similar principle to computing the binary in each step of the loop of `fastPowerSmall` from Homework 2.

   (c) Test it out on the follwing:

   i. `textToInt("Hello World!")`

   ii. Run `intToText` on your output from (i).

   iii. Run `intToText` on 157690076402712651527241. Did it work?

3. Now let's implement a symmetric encryption system. Fix a prime number p and let $\mathcal{M} = \mathcal{C} = \mathcal{K} = \mathbb{F}_p^*$. The encryption function and decryption functions are

   $$e(m, k) \equiv km \mod p \quad \text{and} \quad d(c, k) \equiv k^{-1}c \mod p.$$

   (a) Implement `encrypt(m,k)` and `decrypt(c,k)`. (Encrypt will be very simple, decrypt will need to compute k^{-1}, but you can use a function from a previous homework to do this quickly!).

   (b) Let $p = 370141817103067776979133$, and choose the key $k = 147955927473629958316$. Decrypt the following cipher $c = 85449848686775252245536$ and use `intToText` to read the message!

   (c) Use the same prime p from above and choose your own key. Encrypt and decrypt $m =$`textToInt("Hello!")` with this key. Then apply `intToText`. You should recover your original message!

(d) Do the same thing as in (c) above for $p = 23169331$. What went wrong?

4. For each of the following collections of input data compute the Diffie-Hellman shared secret from both Alice and Bob's perspective

   (a) Prime number $p = 17$. Primitive root $g = 3$. Alice's secret $a = 5$, Bob's secret $b = 11$.

   (b) $p = 56509$. $g = 2$. $a = 3482$, $b = 20487$.

   (c) $p = 370141817103067776979133, g = 31415926535$
   $a = 112233445566778899, b = 998877665544332211$

5. Now implement the Elgamal cryptosystem (we introduced this at the end of class on 9/16, and it summarized in the book [HPS Section 2.4], or in the notes I posted on the discord "elgamal" channel). Fix a prime p and an element $g \in \mathbb{F}_p^*$ and save them as global variables.

   (a) Write Alice's algorithms: `generatePublicKey(a)` which takes as input Alice's private key and returns her public key; and `elgamalDecrypt(c1,c2,a)` which takes as input ciphertext (c_1, c_2) as well as Alice's private key and returns a message.

   (b) Write Bob's algorithm: `elgamalEncrypt(m,A)` which takes as input Alice's public key A and a message m and returns ciphertext (c_1, c_2). (Recall that Bob needs to choose a random exponent to encrypt with. `ZZ.random_element(lowerBound,upperBound)` should come in handy).

   (c) Test it out. Let $p = 9787$ and $g = 34$. Pick any exponent you like and generate a public key as Alice. Then put on your Bob hat and encrypt a message (secret number). Finally decrypt it as alice and make sure you got what you started with. Do this for various values of a and messages.

   (d) I'm trying to send you a message. Let

   $$\begin{aligned} p &= 753022235974397591242683563886842009117 \\ g &= 47393028462819284673. \end{aligned}$$

   Suppose further that your secret key is $a = 314159265358979323846$. I send you the following ciphertext:

   $$\begin{aligned} (c_1, c_2) = \; & (44916496068468858755718588831093165533 2, \\ & 60871368646340361610501366868997982434 1). \end{aligned}$$

   Decode my message. (Use `intToText` to read it!)

   (e) You're welcome to respond to my message. With the same prime and generator, I have a public key of

   $$A = 418194837551245918495968754919547251501.$$

   Encode me a message, and post it on the discord "elgamal" channel. It should still be well enough encrypted so that only I can read it!

Written Part

6. Let's prove some properties of the discrete logarithm.

   (a) Let g be a primitive root of $\mathbb{F}_p^*$. Fix $a, b \in \mathbb{Z}$ and suppose that $g^a \equiv g^b \mod p$. Show that $a \equiv b \mod (p-1)$.

   (b) Use part (a) to prove that the discrete log map $\log_g : \mathbb{F}_p^* \longrightarrow \mathbb{Z}/(p-1)\mathbb{Z}$ is well defined.

   (c) Show that the map $\log_g$ from part (b) is *bijective*. (Hint, can you construct an explicit inverse?).

   (d) Show that $\log_g(ab) = log_g(a) + \log_g(b)$ for all $a, b \in \mathbb{F}_p^*$. (For those of you have seen group theory, this means $\log_g$ is a homomorphism, and in light of (c) an *isomorphism*!)

   (e) Let p be an odd prime and g a primitive root of $\mathbb{F}_p^*$. Prove that $a \in \mathbb{F}_p^*$ has a square root if and only if $\log_g(a)$ is even.

   (f) (BONUS:) We've talked about how the Discrete Log Problem is rather secure. That is, given an odd prime p, a primitive root $g \in \mathbb{F}_p^*$, and some $x = g^a \mod p$, it should be hard to find a. Nevertheless, it is easy to tell whether a is even or odd. Describe a fast algorithm to do so and prove it's correct. (This is often referred to as saying the *least significant bit* of the discrete log problem is insecure).