Goal

In this Lab, you will program RSA encryption.

Independently from this exercise, take a look at the key breaking challenge described in the Lab6 block of the moodle page.

0. Before starting

To be sure to use the latest version, download Lib.zip, extract the files and build the library.

```
$ wget https://www.enseignement.polytechnique.fr/informatique/INF558/TD/Lib.zip
```

```
$ unzip Lib.zip; cd Lib; make clean; make; cd ..
```

Download the source files Lab6.zip, unzip them and move to the corresponding directory.

```
$ wget https://www.enseignement.polytechnique.fr/informatique/INF558/TD/td_6/Lab6.zip
```

```
$ unzip Lab6.zip; cd Lab6
```

You are now ready to code. Open your favorite source-code editor. If you are used to working with VS Code, this can be done with:

```
$ code .
```

Note: You should never modify the include files provided to you. Otherwise, this will crash the VPL server. In case of compilation problem, ask your teachers.

1. Encryption and decryption

To start with, fill in the functions `RSA_encrypt` and `RSA_decrypt` in the file `rsa.c`.

You can test your function using:

```
$ make
```

```
$ ./test_lab6 0
```

2. A better RSA key generation

In `rsa.c`, **we provide** the function

```
int is_valid_key(mpz_t p, mpz_t q, mpz_t e, mpz_t d, int nlen, int sec)
```

which returns 1 if `p, q, e, d` satisfy the following requirements and 0 otherwise.

- Of course, they should satisfy the usual requirements : `p, q` should be prime, `ed` should be congruent to 1 modulo `λ(pq)`
- In addition, according to NIST requirements (Appendix B.3.3):
    - `e` should be in the range $2^{16} \leqslant e \leqslant 2^{256}$;
    - `p, q` should have `nlen/2` bits and both be larger than *√2 (2$^{(nlen/2 - 1)}$)* (use `mpz_urandomb`); `p, q` should not be too close to each other (see NIST doc). Parameter `sec` controls the fact that *|p-q| > 2^(nlen/2-sec)* where *nlen* is the number of bits of *p*q*. Small values of *sec* are ok.

There remains to generate keys. For this sake, complete the function

```
int RSA_weak_generate_key(mpz_t p, mpz_t q, mpz_t e, mpz_t d, int nlen, gmp_randstate_t state)
```

that chooses the 4-tuple `p, q, e, d` such that (according to NIST standards).

You can test your function using:

```
$ make
```

```
$ ./test_lab6 1
```

Upload your file `rsa.c` on VPL.

3. CRT-RSA

If you did not finished it in Lab4, fill in the function `RSA_decrypt_with_p_q`.

You can test your function using:

```
$ make
```

```
$ ./test_lab6 2
```

Upload your file `rsa.c` on VPL.

4. Text encryption

We will perform text encryption using RSA. To do so, a message is a `buffer_t` and should first be split in blocks of the same length. Then each block is converted into an `mpz_t`. We first need to determine the length of such blocks and their number.

In the file `text_rsa.c`, fill in the function

```
int lengths(int *block_length, int *cipher_length, int *last_block_size, buffer_t *msg, mpz_t N)
```

that computes the following data from `msg` and `N`:

- `block_length` is the length of the pieces in which the input buffer will be cut. This length depends on the bit size of `N` (use `mpz_sizeinbase`);
- `cipher_length` is the number of blocks that will compose the ciphertext, i.e. the number of blocks that will divide the input buffer;
- `last_block_size` is the size of the last block, which might be shorter that the other ones.

You can test your function using:

```
$ make
```

```
$ ./test_lab6 3
```

Next, fill in the functions:

- `int RSA_text_encrypt`, that you can test with

  ```
  $ ./test_lab6 4
  ```

- `int RSA_text_decrypt`, that you can test with

```
$ ./test_lab6 5
```

Hint. The GMP functions `mpz_import` and `mpz_export` might be very useful in this exercise.

Upload your file `text_rsa.c` on VPL.

5. Håstad's broadcast attack

Overview of Håstad's attack, based on broadcast messages to recipients who are all using the same small encryption exponent. Suppose we have a series of users, each with a public key (N_i, e_i), but with all of the e_i's equal to some small e. (The moduli N_i and decryption exponents d_i are all different). Now suppose the message M has been encrypted using each of the public keys, and the ciphertext C_i sent to each user. If we intercept at least e of the C_i, then we can easily recover the plaintext M.

We can assume that all of the moduli N_i are pairwise coprime (if the GCD of two of them is nontrivial, we can compute it, then get the correponding secret keys and hence get the message). Assume that we know $C_1, \ldots, C_e$. By definition, $C_i \equiv M^e \bmod N_i$. So, we can use the CRT (which you have already implemented in `CRT.c`) to find some $0 \leqslant C < N_1 \ldots N_e$ such that $C \equiv C_i \bmod N_i$ for $1 \leqslant i \leqslant e$. Now $C \equiv M^e \bmod N_1 \ldots N_e$. On the other hand, $M < N_i$ for $1 \leqslant i \leqslant e$, so $M^e < N_1 \ldots N_e$. Hence we can recover M by taking the e-th root of C *as an integer*.

In the file `hastad.c`, fill in the function

```
void Hastad(mpz_t decrypted, mpz_t* cipherTexts, mpz_t* moduli, int exponent)
```

where

- `decrypted` is the target result.
- `moduli` is the array of the N_i's.
- `cipherTexts` is the array of the C_i's.

You can test your function using:

```
$ make
```

```
$ ./test_lab6 6
```

The output should be:

```
------------- Test Hastad attack --------------

Plain text :
12345678

Key generation and encryption done.

[TEST OK]
```

Upload your file `hastad.c` on VPL.