Goal

In this Lab, you will

1. learn how to use the GMP library;
2. implement the Extended Euclidian Algorithm;
3. implement the Chinese Remainder Theorem;
4. implement the RSA cryptosystem (optional).

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 Lab4.zip, unzip them and move to the corresponding directory.

```
$ wget https://www.enseignement.polytechnique.fr/informatique/INF558/TD/td_4/Lab4.zip
```

```
$ unzip Lab4.zip; cd Lab4
```

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. Using GMP

1.1. Checking that GMP is correctly installed

If `gmp` is installed at system level, then the following should work

```
$ make -f MakefileGMP; ./checkgmp
```

and print something like

```
Hi, this is version 60000 of GMP
12345678901234567890
```

If this is not the case, here is how you should proceed

- install gmp from gmplib.org; generally the following command in the downloaded directory is enough

  ```
  $ ./configure; make; make install
  ```

- then, modify the installation path in the varaible `GMP_DIR` in the file `../Lib/Makefile.common`.

The lecture slides on how to use GMP are available here. You can also refer to the GMP official documentation.

1.2. More practicing

In `fermat.c`, complete the function

```
void fermat(int a, int pmin, int pmax, int
      composites){ ... }
```

that prints all **odd** integers in the interval `[pmin, pmax[` such that $a^{p-1} = 1 \bmod p$. For each such value of `p`, print the value of `a` and the value of `p`. For now, `composites` is not used and can be thought of as having value 0.

Then test your code by executing:

```
$ make -f MakefileGMP
```

```
$ ./fermat 2 300 400
```

The output should be of the form:

```
2 307
2 311
2 313
2 317
2 331
2 337
2 341
2 347
2 349
2 353
```

```
2 359
2 367
2 373
2 379
2 383
2 389
2 397
```

Modify the program so that, when the `composite` argument is non-zero, only **odd composite** integers are printed:

```
$ ./fermat 2 300 1000 1
```

```
2 341
2 561
2 645
```

Hint: The functions you are going to use are:

```
mpz_cmp
mpz_sub_ui
mpz_powm
mpz_cmp_ui
mpz_add_ui
mpz_probab_prime_p
```

and must be looked up in the GMP documentation.

Upload your file `fermat.c` on VPL.

2. Extended Euclidean algorithm

2.1 PROGRAMMING XGCD

We exemplify a standard way of programming on big integers. First write a version using `long` to concentrate on the algorithm. When this first version works, rewrite it using `mpz_t` to handle big integers. Doing this helps you debug this latter version using the former.

2.1.1 USING LONGS

In the file `xgcd.c`, complete the the code of the function `XGCD_long`, implementing the extended Euclidean algorithm.

```
/* compute g, u and v s.t. a*u+b*v = g = gcd(a, b) */
void XGCD_long(long *g, long *u, long *v, long a, long b){ ... }
```

Test your code using

```
$ make -f MakefileGMP test_xgcd
```

```
$ ./test_xgcd 0
```

which should give as output

```
test 0.1:       [gcd: OK]
[xgcd: OK]
test 0.2:       [gcd: OK]
[xgcd: OK]
test 0.3:       [gcd: OK]
[xgcd: OK]
test 0.4:       [gcd: OK]
[xgcd: OK]
```

2.1.2 USING MPZ_T

Fill in the code for `XGCD`, implementing the extended Euclidean algorithm. Complete the file `xgcd.c` and test it using

```
$ make -f MakefileGMP test_xgcd
```

```
$ ./test_xgcd 1
```

which should give as output

```
test 1.1:       [gcd: OK]
[xgcd: OK]
test 1.2:       [gcd: OK]
[xgcd: OK]
test 1.3:       [gcd: OK]
[xgcd: OK]
test 1.4:       [gcd: OK]
[xgcd: OK]
```

Warning: do not change the values of the input arguments, please.

2.2 SOLVING $a\,x = b \bmod m$

Complete the function

```
int linear_equation_mod(mpz_t x, mpz_t a, mpz_t b, mpz_t m){ ... }
```

that computes the least positive integer solution `x` to the equation $a\,x = b \bmod m$, where a, b and m are three integers. It returns `0` if there is no solution to the modular equation and `1` otherwise. We **do not** suppose that a is prime to m.

Program the algorithm and test it using

```
$ make -f MakefileGMP test_xgcd
```

```
$ ./test_xgcd 2
```

which should give as output

```
test 2.1:     [OK]
test 2.2:     [OK]
test 2.3:     [OK]
test 2.4:     [OK]
test 2.5:     [OK]
```

Upload your file `xgcd.c` on VPL.

3. The Chinese Remainder Theorem

Now we want to implement a Chinese Remainder Theorem function in file `CRT.c`

Complete the function

```
/* Given (r0, m0) and (r1, m1), compute n such that
n mod m0 = r0; n mod m1 = r1.  If no such n exists, then this
function returns 0. Else returns 1.  The moduli m must all be positive.*/
int CRT2(mpz_t n, mpz_t r0, mpz_t m0, mpz_t r1, mpz_t m1){ ... }
```

If you are not used to gdb, you might want to write a `long` version beforehand, as you did for XGCD above.

Then, complete the function

```
/* Given a list S of pairs (r,m), returns an integer n such that n mod
m = r for each (r,m) in S.  If no such n exists, then this function
returns 0. Else returns 1.  The moduli m must all be positive.*/
int CRT(mpz_t n, mpz_t *r, mpz_t *m, int nb_pairs){ ... }
```

Hint: use `CRT2` in `CRT`.

Compile your functions using

```
$ make -f MakefileGMP test_CRT
```

To test `CRT2`, use

```
$ ./test_CRT 1
```

which should give as output

```
CRT([(1, 7),(2, 11)]) = (1, 57)
```

Next, to test `CRT`, use

```
$ ./test_CRT 2
```

which should give as output

```
CRT([(1, 7),(2, 11)]) = (1, 57)
CRT([(1, 7),(2, 11),(3, 13)]) = (1, 211)
CRT([(1, 7),(2, 11),(3, 13),(4, 15),(5, 17)]) = (1, 18229)
CRT([(1, 3),(2, 6)]) = (0, undef)
CRT([(1, 3),(4, 6)]) = (1, 4)
```

Hint: It is advised to carefully analyze the third and fourth examples to understand when the CRT fails and how this leads to a correct programming of the case of moduli that are non prime together. Using exercise 2.3 could be a good idea.

Upload your file `CRT.c` on VPL.

4. RSA (optional)

We are going to implement a (rather naïve) RSA cryptosystem.

4.1 Key generation

Complete the functions

```
void generate_probable_prime(mpz_t p, size_t nbits, gmp_randstate_t state){ ... }
```

which generate a probable prime p of size nbits and

```
void RSA_generate_key(mpz_t N, mpz_t p, mpz_t q, mpz_t e, mpz_t d, size_t nbits, gmp_randstate_t state){ ... }
```

which generates a composite module N and two primes p and q of size nbits.

Hint: you can use the functions you are going to use are:

```
mpz_urandomb
mpz_nextprime
mpz_sizeinbase
```

4.2 Encryption / Decryption

Complete the functions

```
void RSA_encrypt(mpz_t cipher, mpz_t msg, mpz_t N, mpz_t e){ ... }
```

```
void RSA_decrypt(mpz_t msg, mpz_t cipher, mpz_t N, mpz_t d){ ... }
```

in file `rsa.c`. It is supposed that `msg` and `cipher` are two positive integers less than *N*.

Test your function using

```
$ make -f MakefileGMP test_rsa
```

```
$ ./test_rsa 60
```

4.3 Decryption using CRT

Complete the function

```
void RSA_decrypt_with_p_q(mpz_t msg, mpz_t cipher, mpz_t N, mpz_t d, mpz_t p, mpz_t q){ ... }
```

that decrypts an RSA message by using computations modulo p and q and building the final answer using CRT.

Upload your file `rsa.c` on VPL.