Goal

In this Lab, you will

1. implement the baby-step giant-step algorithm;
2. implement several factorization algorithms:
   - trial division;
   - Pollard rho;
   - Pollard's *p-1*.

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 Lab5.zip, unzip them and move to the corresponding directory.

```
$ wget https://www.enseignement.polytechnique.fr/informatique/INF558/TD/td_5/Lab5.zip
```

```
$ unzip Lab5.zip; cd Lab5
```

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.

In this lab, you will need to read from a file. You can do that with the `fscanf` function from `stdio.h`. Look for it on the Internet; it is a good habit to look for the documentation.

1. Discrete logarithm: the baby-step giant-step algorithm

Your goal is to implement the baby-step giant-step algorithm for finite fields the file `dlog.c`. The BSGS algorithm is described in the slides of the lecture (slide 6). For this you will need to fill the following functions:

- `int babySteps(mpz_t result, hash_table H, mpz_t g, mpz_t u, mpz_t p)`
- `int giantSteps(mpz_t result, hash_table H, mpz_t g, mpz_t ordg, mpz_t u, mpz_t p, mpz_t a)`
- `int BSGS(mpz_t result, mpz_t a, mpz_t g, mpz_t p)`

All the functions should return one of the constants in `dlog.h`, i.e.

```
#define DLOG_ERROR       0
#define DLOG_OK          1
#define DLOG_FOUND       2
#define DLOG_NOT_FOUND   3
#define DLOG_SMALL_ORDER 4
```

1.1 BABY STEPS

The `babySteps` function is the precomputation phase. This function fill the hash table with `(key, value)` pairs of the form (g^i, i) for `i` from `0` to `u - 1`.

Hint. Suppose the hash table `H` was initialised. To put a new pair `(key, value)` of `mpz_t` you can use the function

```
int hash_put_mpz(hash_table H, int *addr, mpz_t kz, mpz_t vz, mpz_t base, mpz_t p)
```

This function adds to `H` the pair `(kz, vz)` such that $kz = base^{vz} \bmod p$ and writes its address in `addr` (only useful for tests).

You can test your function using:

```
$ make -f MakefileDLog
```

```
$ ./test_dlog 1
```

1.2 GIANT STEPS

The `giantSteps` function is the online phase, it computes the log of `a` modulo `p` and puts it in `result`.

Hint. You can use the function

```
int hash_get_mpz(mpz_t vz, hash_table H, mpz_t kz, mpz_t base, mpz_t p)
```

This function searches the key `kz` in `H`. If it is found in the table, the function returns the constant `HASH_FOUND` and writes the corresponding value in `vz`. Else it returns the constant `HASH_NOT_FOUND`.

1.3 FULL ALGORITHM

The `BSGS` function implements the complete algorithm, using the previous two functions. Ask yourselves what should be the right number of steps (i.e the `u` in `babySteps`).

First, complete the code of `BSGS_aux` where `ordg` **is a multiple of the order of** `g`.

Use `hash_init(int size)` to initiate the hash table and `hash_clear(hash_table H)` to clear it.

This function calls `babySteps`. If the output is `DLOG_SMALL_ORDER`, it stops here and returns `DLOG_SMALL_ORDER`. Otherwise it keeps going and calls `giantSteps` to obtain the result.

Then implement the `BSGS` function. This function calls the `BSGS_aux` function a first time with the value of `ordg` set to `p-1` (which is for sure a multiple of the order).

- If `BSGS_aux` returns `DLOG_SMALL_ORDER`, it replaces the value of `ordg` by the value of the order that was found and calls `BSGS_aux` a second time. In such a case, the function should print `"Resetting order of g to X"`, where `X` is the value of the order.
- Else, in most cases, `BSGS_aux` will return the discret log value at the first call.

You can test your function using:

```
$ make -f MakefileDLog
```

```
$ ./test_dlog 2
```

Upload your file `dlog.c` on VPL.

2. Factorization

CHALLENGES

We consider the following list of numbers, also given in `challenges.txt` (format is *label N dd* where *dd* is the number of decimal digits of *N*):

```
N01 1267650600228229401496703205376 31
N02 69470986277398276682046998304329 32
N03 74981672081934458260565458974847 32
N04 18063315121424468776841394706747 32
N05 52065415254431970913156375768427 32
N06 79617516433484608487286417588949500109 38
N07 7696690982041032223536306591934241 34
N08 29306081729217262791162079172896271 35
N09 13490699863228332154492051782011371 35
N10 27933357565942078417780001381279269 35
N11 15791941410456156312508410327078929 35
N12 2992060526692601278017179365187699 34
N13 7007884240806300596241285733353197 34
```

```
N14 1957955439302821383896943177677063 34
N15 1865335008205330192779072516817771 34
N16 1998761405387706851281702367368673 34
N17 2377129073992268618899272983512591 34
N18 3218220016059292314500145576546353 34
N19 3836043059373611935528948331938213 34
N20 4765056827518867205783593928033833 34
```

These numbers may be factored by some of the methods given below. Do your best.

Write your findings in a file `factorizations.txt`. The expected format is as follows: if you factor *N1 = p1^e1 ... pk^ek* (all *p_i* distinct), write

```
N01 1267650600228229401496703205376 31
p1 e1 name_of_the_method used to find this factor including parameters
...
pk ek name_of_the_method used to find this factor including parameters
```

Upload your file `factorization.txt` on VPL.

2.1. Trial division

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

```
int trialDivision(
      factor_t* factors,
      int *nf,
      mpz_t cof,
      const mpz_t N,
      const long bound,
      uint length,
      FILE* ficdp)
```

The parameters are:

- `N` the `mpz_t` we wish to factor;
- `ficdp` a file listing the half-differences of consecutive primes. We provide a file named `1e7.data` that gives such a list for all primes less than 10^7 (**Caution :** the first line of the file refers to the difference between 2 and 3 and NOT the half difference);
- `factors` a pointer to a structure `factor_t`. Such a stucture is defined in `utils.h` and contains
  - a field `mpz_t f` which is a factor (of the GMP integer `N` we aim at factorising);
  - a field `int e` which is the exponent of `f` as a factor of `N`;
  - a field `int status` which is the primality status of the factor (see `utils.h`).

  In `utils.h`, you have a function `AddFactor` which permits to put a new factor in your table `factors`. For instance `AddFactor(factors + 3, f, e, status)` will put at the cell number 3 of the table `factors` the `factor_t` with factor `f`, exponent `e` and status `status`.
- `*nf` the number of factors of `N` put in `factors`.
- `bound` an upper bound on the primes we test in the trial division process;
- `length` the memory size of the table `factors`. If the table is full, the factorisation process should stop;
- `cof` the part of `N` which is prime to all the primes up to `bound`. If `cof` equals 1 at the end of the execution, then the factorization is completed.

The function should return 1 when all probable prime factors of `N` were found; in other words: `cof` is 1 or a probable prime.

You can test your function using:

```
$ make -f MakefileFactor
```

```
$ ./test_factor 1
```

The output should be:

```
************** Testing Trial division **************

------------------------
Factorization of 3276 with primes less than 10:

(2^2) * (3^2) * (7^1) * (13)

------------------------
Factorization of 3276 with primes less than 100:

(2^2) * (3^2) * (7^1) * (13^1) * (1)

------------------------
Factorization of 291452718198310678303493860073557517926904236020742651379250000 with primes less than 100:

(2^3) * (3^6) * (5^5) * (17^3) * (31^2) * (41^6) * (71306198031317215975086685276791653621)

------------------------
Factorization of 12652209139612535291 with primes less than 10000000:

(863^1) * (1857731^1) * (7891739447)
```

Upload your file `drialdiv.c` on VPL.

Do not forget to update `factorizations.txt`.

2.2 Pollard rho for factorization

First, we want to factor a small integer `N`. Implement the Pollard rho algorithm for `long` as explained in the slides (slide 8/9). In the file `rho.c`, fill in the function:

```
int PollardRho_with_long(long *factor, const long N, long nbOfIterations)
```

In this function, the parameters are as follows:

- `N` is the integer which we expect to factor;
- `*factor` will receive the result if found, i.e. a factor of `N`;
- `nbOfIterations` is the maximum number of iterations of the algorithm.

You can use the function `f(x)=x^2+7` to instantiate the random function in the slides.

You can test your function using:

```
$ make -f MakefileFactor
```

```
$ ./test_factor 2
```

The output should be:

```
Using function x |-> x^2 + 7.
--------------------
factors of 44 : 4, 11
Factor found:1
Running time : 0.000008 seconds.

--------------------
factors of 126522 : 9, 14058
Factor found:1
Running time : 0.000003 seconds.

--------------------
factors of 448537 : 251, 1787
Factor found:1
Running time : 0.000014 seconds.
```

Upload your file `rho.c` on VPL.

Do not forget to update `factorizations.txt`.

Now, implement the Pollard rho algorithm using `GMP` functions by filling in file `rho.c` the following functions:

```
int PollardRhoSteps(
      mpz_t factor,
      const mpz_t N,
      void (*f)(mpz_t, mpz_t, const mpz_t),
      long nbOfIterations)
```

```
int PollardRho(
      factor_t *result,
      int *nf,
      const mpz_t N,
      void (*f)(mpz_t, mpz_t, const mpz_t),
      long nbOfIterations)
```

In these functions, the parameters are as follows:

- `N` is the `mpz_t` which we expect to factor;
- `factor` will receive the result if found, i.e. a factor of `N`;
- `f` is a function: let `f` be the function of the slides, `f(output, input, N)` computes `f(input) mod N` and puts the result in `output`.
- `nbOfIterations` is the maximum number of iterations of the algorithm.
- `*result` is a `factor_t` structure in which you will add the found factor. Recall the function `AddFactor` from `utils.h`. You may put `e=1`, and compute the primality status with `mpz_probab_prime_p`.
- `*nf` is the number of factors in `*result`. It will basically be one, but you need to update it when you add a factor.

Your functions should return `FACTOR_FOUND` (defined in `utils.h`) if a factor has been found.

You can test your function using:

```
$ make -f MakefileFactor
```

```
$ ./test_factor 3
```

The output should be:

```
************** Testing Pollard rho *********************


**************************************************
*   Using function x |-> x^2 + 7.            *
**************************************************

Seeking a factor for N = 22145579
--------------------
[FACTOR FOUND] : p = 2039 (1)
Running time : 0.000135 seconds.

Seeking a factor for N = 12652209139612535291
--------------------
[FACTOR FOUND] : p = 863 (1)
Running time : 0.000053 seconds.

Seeking a factor for N = 10541221091544233897
--------------------
[FACTOR FOUND] : p = 18757 (1)
Running time : 0.000080 seconds.

Seeking a factor for N = 633564754957339397639948337059
--------------------
[FACTOR FOUND] : p = 760153 (1)
Running time : 0.001782 seconds.

Seeking a factor for N = 2035109857152735577711831203565763223283
--------------------
[FACTOR NOT FOUND]

Running time : 0.094538 seconds.

Seeking a factor for N = 72963193328043133999662344352921779599583554200941
--------------------
[FACTOR NOT FOUND]

Running time : 0.102198 seconds.


**************************************************
*   Using function x |-> x^2 - 3.            *
**************************************************

Seeking a factor for N = 12652209139612535291
--------------------
[FACTOR NOT FOUND]

Running time : 0.000004 seconds.

Seeking a factor for N = 10541221091544233897
--------------------
[FACTOR NOT FOUND]

Running time : 0.000001 seconds.

Seeking a factor for N = 633564754957339397639948337059
--------------------
[FACTOR NOT FOUND]

Running time : 0.000002 seconds.

Seeking a factor for N = 2904904137951823762898116102980679156667
--------------------
[FACTOR NOT FOUND]

Running time : 0.000003 seconds.

Seeking a factor for N = 72963193328043133999662344352921779599583554200941
```

```
--------------------
[FACTOR NOT FOUND]

Running time : 0.000002 seconds.
```

As you can notice, the choice of the function `f` is important. Try to modify in `test_factor.c` the code of the function `badFunction` to get better results.

Upload your file `rho.c` on VPL.

Do not forget to update `factorizations.txt`.

2.3. Pollard's *p-1* factorization algorithm

In the file `pminus1.c`, we provide the second step of Pollard's *p-1* factoring algorithm. Implement the first step, and the general algorithm using the two steps, by filling the following functions:

```
int PollardPminus1Step1(mpz_t factor, const mpz_t N, long bound1, FILE* ficdp, mpz_t b, mpz_t p)
```

```
int PollardPminus1(factor_t *res, int *nf, const mpz_t N, long bound1, long bound2, FILE* ficdp)
```

In these functions, the parameters are as follows:

- `factor` is the output factor (if found)
- `N` is the GMP integer to factor;
- `bound1` is the upper bound B_1 of the slides (slide 12).
- `bound2` is the upper bound B_2 of the slides (slide 13).
- `ficdp` is like in Exercise 2.1.
- `b` is as in slide 12
- `p` is a lower bound for the prime numbers we consider. Basically, we will consider `R = lcm(p+1,...,bound1)` and test whether $\gcd(b^R-1, N) \neq 1$. **The initial call should be done with *p = 1*.**
- `*res` is a `fact_t` structure that will contain the factor `factor` (if found). As in the previous exercise, you may set `e=1`, and compute the primality status with `mpz_probab_prime_p`.
- `*nf` is the number of factors in `*res`.

Your functions should return `FACTOR_FOUND` (defined in `utils.h`) if a factor has been found.

You can test your function using:

```
$ make -f MakefileFactor
```

```
$ ./test_factor 4
```

The output should be:

```
**************** Testing Pollard p - 1 *******************


**********************************
*     PHASE 1:                   *
**********************************
```

```
Seeking a factor for N = 2993
--------------------
[FACTOR FOUND] : p = 73 (1)
Running time : 0.000081 seconds.

Seeking a factor for N = 12652209139612535291
--------------------
[FACTOR FOUND] : p = 863 (1)
Running time : 0.000347 seconds.

Seeking a factor for N = 561988649120021
+--------------------+
[FACTOR NOT FOUND]


**********************************
*     PHASE 1&2:                 *
**********************************

Seeking a factor for N = 561988649120021
+--------------------+
[FACTOR FOUND] : p = 144037 (1)
Running time : 0.001710 seconds.

Seeking a factor for N = 633564754957339397639948337059
--------------------
[FACTOR FOUND] : p = 760153 (1)
Running time : 0.000896 seconds.
```

Upload your file `pminus1.c` on VPL.

Do not forget to update `factorizations.txt`.