An introduction to GMP library

(GNU Multiple Precision)

1 Introduction

1.1 Motivation

GMP (GNU Multiple Precision arithmetic library) is a C library dedicated to computation on large integers.

This library provides numerous functions that allow to compute on several multi-precision types:

- (large) integers: $\mathbb{Z}$
- (large) rationals: $\mathbb{Q}$
- (large) floating point numbers: $\mathbb{R}$ (see also the MPFR library)

The term *multi-precision* means that a number (belonging to $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$) is internally stored on as many "words" (of the native machine type) as needed to represent it with the required precision for the undertaken computation.

Example Program which computes the factorial function without GMP

```
#include <stdio.h>

int main(int argc, char* argv[])
{
  unsigned int n, i;
  unsigned int fact = 1;

  n = atoi(argv[1]);
  i = 1;
  while (i <= n)
```

```
  {
    fact *= i;
    i++;
  }
  printf(" n! = %u\n", fact);

  return 0;
}
```

If one executes this program to compute the factorial of 12, this gives the result $12! = 479\,001\,600$ which is correct. Nevertheless, the computation of the factorial of 13 results in an overflow with respect to the maximal value that can be stored on an unsigned 32-bit integer ($2^{32} - 1 = 4\,294\,967\,295$). One obtains $13! = 1\,932\,053\,504$ which is erroneous. (One can notice however that $13! \bmod 2^{32} = 1\,932\,053\,504$.)

One could think it should be enough to choose type `long` or `long long` but this only slightly postpones the overflow event as a 64-bit overflow happens starting from 21!.

Example Program which computes the factorial function with GMP

```
#include <stdio.h>
#include "gmp.h"

int main(int argc, char* argv[])
{
  unsigned int n, i;
  mpz_t z_fact;

  n = atoi(argv[1]);

  mpz_init(z_fact);
  mpz_set_ui(z_fact, 1);

  i = 1;
  while (i <= n)
  {
    mpz_mul_ui(z_fact, z_fact, i);
    i++;
  }
  gmp_printf("%u! = %Zu", n, z_fact);

  mpz_clear(z_fact);
```

```
  return 0;
}
```

This program can easily compute

$$\begin{aligned} 30! &= 265\,252\,859\,812\,191\,058\,636\,308\,480\,000\,000 \\ &\approx 2^{108} \\ &\approx 10^{33} \end{aligned}$$

which is a 108-bit integer that needs four (`unsigned int`) 32-bit words to be represented.

The same program can as much easily compute 2000! which is equal to:

```
33162750924506332411753933805763240382811172081057803945719354370603 80\
77905600822400273230859732592255402352941225834109258084817415293796 13\
13866335263436889056340585561639406051172525718706478563935440454052 43\
95746703767410872297043468415834375243158087753364512748799543685924 74\
08032408946561507233250652797655757179671536718689359056112815871601 71\
72326571561100042140124204338425737127001758835477968999212835289966 65\
85340557985490365736635013338655040117201215263548803826815215224692 09\
95206031564418565480675946497051552288205234899995726450814065536678 96\
95321014676226713320268315522051944944616182392752040265297226315025 74\
75204829606475092739416585628353177957448287631459645037399132733417 72\
63608852490093506621610144459709412707821313732563831572302019949914 95\
83164709427744738703279855496742986088393763268241524788343874695958 29\
25774057453983750158581546813629421794997239981359948101655656387603 42\
27312912250384709872909626622461971076605931550201895135583165357871 49\
22909167790497022470946119376077851651106844322559056487362665303773 84\
65039078804952460071254940261456607225413630275491367158340609783107 49\
45282217490781347709693241556111339828051358600690594619965257310741 17\
70815199225645167785714580566021856547609523774630166794224884444857 98\
34980154803262082989096585738175188861937669282827988845358463989659 42\
13952984465291092009103710046149449915828588050761867924946385180879 87\
45128914080193400746259200570987295785996436506558956124102310186905 56\
06030878362911050560124590899838341079936790205207685866918343790655 85\
44700148692656924631933337612428097420067172846361939249698628468719 99\
34503938893672704871271727345617003548674775091029555239535479411074 21\
91330135681954109194146276641754216158762526285808980122244389024867 71\
82054959415751991701271767571787495861619665931878855141835782092601 48\
20717773317353960343049690820705899587013819808130355901607629083885 74\
56128821769813618248357673921830311841471913398689284234400077924669 12\
09766731651433494437473235636572048844478331854941693030124531676232 74\
53678793228474738244850922831399525097325059791270310476836014811911 02\
22925337269769382367005756561240029057604385285290293760647953345817 96\
66123839605262549107186663869354766108455046198102084050635826776526 58\
94923932495196859541716724193295306836734955440045863598381610430594 49\
```

$$8266275306054235807558941082788804278259510898806354105679179509740177\
80688782869810219010900148352061688883720250310665922068601483649830 53\
27820882635365580436056867812841692171330471411763121758957771226375 84\
75312351723099054982921013468730420589801441806387538266416989770423 77\
59406280877253702265426530580862379301422675821187143502918637636340 30\
01732518182620760397473695952026426323641454468511134272021504583838 51\
01013694131303485622191663162389263276581535501127630782505996915882 45\
33457435437863683173730673296589355199694458236873508830278657700879 74\
98899923435555662406828347637846851838449736488739524751032242221105 61\
20129582965719136810869382547576411888687934672519124619215114473883 62\
69591643672490071653428228152661247800463922544945170363723627940757 78\
45420910483054616561906221742869816029733240465202019928138548826819 51\
00728286970107073750092766648750217477537274235150874824672027417003 15\
81122805896178122160747437947510950620938556674581252518376682157712 80\
78614992558761323529504223463878789548508857644661362903941276659780 44\
20209228133798711590089626487894241321045492500356667063290944157937 29\
86743421470507213588932019580723064781498429522595589012754823971773 32\
57229103257609297907332995450563883626404746502450808094691160726320 87\
49414397300070411141859553027882735765481918200244969776111134631819 52\
82761590964189790958117338627206088910432945244978535147014112442143 05\
54860896395783783473253235957632914389252883939862562732428627755631 40\
46383038916842163311344563630957196597846633855149231619633567535513 84\
03425804162919837822266909521770153175338730284610841886554138329171 95\
13321178957285416620848236828179325129312375215419269702697032994776 43\
82338648300887153037340566638386829408848773072176226884902308493466 11\
94260180272613802108005078215741006054848201347859578102770707780655 51\
27725405016743323960662532164150048087724030476119290322101543853531 38\
68553848642557079079534117651957118868373988068389579274374968349814 29\
23292196309777090143936843655333359307820181312993455024206044563340 57\
86069624719615056033948995233218004343599672566239271964354028720554 75\
01207985433197067479731312681352365374408566226320676883758513278289 62\
52333284341812977624697079543436003492343159239674763638912115285406 65\
77836462139112474470512552263427012395270181270454916480459322481088 58\
67460095230679317596775558101167994000524980630376314134441226903703 49\
87355799916009259248075052485541568266281760815446308305406677412630 12\
44418642041083731190931300011544705602777737243780671888997708510567 27\
27678124719883285769584421758889516046786820481001004781646235822083 85\
32488134270834079868486632162720208823308727819085378845469131556021 72\
88731219073939652092602291014775270809308653649798585540105774502792 89\
81460368843182150863724621696787228216934737059928627711244769092090 29\
88320166830170273420259765671709863311216349502171264426827119650264 05\
42282317596308744753018471940955242634114984695080733900800000000000 00\
00000000000000000000000000000000000000000000000000000000000000000000 00\
00000000000000000000000000000000000000000000000000000000000000000000 00\
00000000000000000000000000000000000000000000000000000000000000000000 00\
00000000000000000000000000000000000000000000000000000000000000000000 00\
00000000000000000000000000000000000000000000000000000000000000000000 00\
00000000000000000000000000000000000000000000000000000000000000000000 00$$

This is a 19053-bit integer – 5736 decimal digits – which is computed instantaneously.

As a last example, the factorial of 100 000 (1 516 705 bits) is computed in less than one second. It writes over 456 574 decimal digits and requires 185 kilo-bytes to be stored in memory.

1.2 Compilation of a program using the GMP library

To be able to use the library functions in your program called `my_program.c`, you must:

1. Include in `my_program.c` the file `gmp.h` by means of the instruction `#include <gmp.h>`. This file contains the definitions of all the necessary types and constants, as well as the prototypes of all external library functions.

2. Link with the GMP library. This creates the (executable) binary file by joining together the object code of your program – i.e. the intermediate file `my_program.o` created in the first step of your compilation – and the object codes of all library functions.

Compilation in two steps:

- `gcc -c my_program.c` ⇒ compiles the source file `my_program.c` and creates the object file `my_program.o`,

- `gcc my_program.o -lgmp -o my_program` ⇒ creates the executable file `my_program` by linking the object codes in the file `my_program.o` together with the GMP library.

Compilation in a single step:

- `gcc my_program.c -lgmp -o my_program` ⇒ compiles and links successively.

2 Using the library

2.1 Basic principles

The basic arithmetic types defined by GMP are:

- `mpz_t` for the integers ($\mathbb{Z}$)

- `mpq_t` for the rationals ($\mathbb{Q}$)
- `mpf_t` for the floating point numbers ($\mathbb{R}$)

Remark: for cryptographic purpose, one may only be interested by the "big integers" ($\mathbb{Z}$) GMP type (type `mpz_t`).

These basic GMP types are **pointers on structures** which contain all necessary data to manage the multi-precision aspect.

Examples of declarations

```
mpz_t number;

struct {
  mpz_t x;
  mpz_t y;
  mpz_t z;
} point;  // For example, a point on an elliptic curve
          // (in projective coordinates)

mpz_t tab[10];
```

Even if we do not need to know the details, here are the different parts of the structure pointed by a `mpz_t` variable:

- `mp_limb_t *_mp_d`: a pointer to an array of *limbs* (`unsigned int`) that contains the integer value represented in base 2^{32}
- `int _mp_alloc`: the number of limbs allocated for the array
- `int _mp_size`: the number of limbs that are actually used to represent the current value (may be less than the number of allocated limbs)

The array that contains the big integer value is dynamically allocated and its size can be increased if needed during the computation. So GMP is not only a library for multi-precision arithmetic, but it is a library that computes with an arbitrary precision.

When a `mpz_t` is declared, it is a non initialized pointer which points to nothing. No structure (as above) is allocated to describe the integer. A fortiori, no memory is allocated to store its value.

The names of GMP functions that compute arithmetic operations on big integers (`mpz_t`) all start with `mpz_`.

Remark: Since `mpz_t` type is a pointer to the structure that contains the data and its meta-data, passing a big integer as a function parameter is simply done by address. So there is no need to prepend the variable name by "`&`" for the function being able to modify its content.

2.2 Description of the most useful functions

2.2.1 Integer initialization and disallocation functions

- `void mpz_init (mpz_t x)` : initializes an `mpz_t` variable; this function creates the structure, including an array of limbs that contains the value 0. This function is equivalent to a `malloc()` function for the `mpz_t` type.

- `void mpz_inits (mpz_t x, ...)` : initializes several variables at the same time.

- `void mpz_clear (mpz_t x)` : removes the object pointed to by the variable (the structure) and disallocates the memory it occupies. This is not only the array of limbs containing the value, but also the structure itself. This function is equivalent to a `free()` function for the `mpz_t` type.

- `void mpz_clears (mpz_t x, ...)` : clears several variables at the same time.

Example

```
mpz_t z_n;        // z_n is declared but not allocated

mpz_init(z_n);    // z_n now actually points onto an integer (0)
...
// One makes computations with z_n variable
...
mpz_clear(z_n);   // One frees the object pointed to by z_n
                  // whenever the computations are finished
```

2.2.2 Assignment functions

- `void mpz_set (mpz_t rop, mpz_t op)`

- `void mpz_set_ui (mpz_t rop, unsigned long int op)`
- `void mpz_set_si (mpz_t rop, signed long int op)`
- `void mpz_set_d (mpz_t rop, double op)`
- `void mpz_set_q (mpz_t rop, mpq_t op)`
- `void mpz_set_f (mpz_t rop, mpf_t op)`

These functions are equivalent to `rop = op;` but with `rop` being of type `mpz_t`. Which function to use depends on the type of `op`.

Most often we will only make use of the two first functions.

Example

```
mpz_t z_n1, z_n2;

mpz_init(z_n1);
mpz_init(z_n2);
mpz_set_ui(z_n1, 1234);   // Assign the value 1234 to z_n1
mpz_set(z_n2, z_n1);      // Copies the value of z_n1 to z_n2
```

- `int mpz_set_str (mpz_t rop, char *str, int base)` : assign to an `mpz_t` variable an integer value represented by a string of characters. One must specify the base in which the value is represented. Note that `str` can either be a variable or a literal string. If `base` is equal to 0 then the actual base is inferred from the string prefix: `0x` (base 16), `0b` (base 2), `0` (base 8), and is base 10 by default.

One can so write the following:

```
mpz_set_str(z_n, "84973918290487975931843982749165193191035 80", 0);
mpz_set_str(z_n, "55555555555555555555555555555555555555", 6);  // 6^35-1
```

- `void mpz_swap (mpz_t rop1, mpz_t rop2)` : swaps the values of two `mpz_t` variables. (Actually the function swaps their addresses.)

Other functions allow to initialize and assign a value at the same time. Their names are built from the names of the assignment functions by adding "`init_`" before "`set`". For instance:

```
mpz_t z_n;
mpz_init_set_ui(z_n, 1234);  // Initializes then assigns the value 1234
```

2.2.3 Conversion functions

These functions convert a `mpz_t` variable to a native C type. Here are some examples:

- `unsigned long int mpz_get_ui (mpz_t op)`
- `signed long int   mpz_get_si (mpz_t op)`
- `double            mpz_get_d (mpz_t op)`
- `char *            mpz_get_str (char *str, int base, mpz_t op)`

The value of the `mpz_t` variable must preferably be small enough to fit in the destination type. (cf. the GMP manual for more details)

2.2.4 Arithmetic functions

- `void mpz_add (mpz_t rop, mpz_t op1, mpz_t op2)`
- `void mpz_add_ui (mpz_t rop, mpz_t op1, unsigned long int op2)`

Example

```
mpz_t z_n1, z_n2;

mpz_init(z_n1);
mpz_init_set_str(z_n2, "184891739278419837932714923813974321 4", 0);

mpz_add_ui(z_n2, z_n2, 1234);   //  z_n2 <-- z_n2 + 1234
mpz_add(z_n1, z_n2, z_n2);      //  z_n1 <-- z_n2 + z_n2
```

There also exist functions to subtract, multiply, as well as some multiplication functions for which the result is added to, or subtracted from the original value of the output operand. (cf. the GMP manual)

- `void mpz_mul_2exp (mpz_t rop, mpz_t op1, mp_bitcnt_t op2)`
- `void mpz_neg (mpz_t rop, mpz_t op)`
- `void mpz_abs (mpz_t rop, mpz_t op)`

Function `mpz_mul_2exp` multiplies op_1 by 2^{op_2}. Function `mpz_neg` gives the opposite of *op*. Function `mpz_abs` gives the absolute value of *op*.

2.2.5 Division functions

There exist numerous division functions which offer a variety of options regarding: (i) which result is returned (the quotient, the remainder or both), (ii) the way of rounding (lower rounding, upper rounding, toward 0 or toward infinity). See the GMP manual for the details.

Here is an example of a division function:

- `unsigned long int mpz_cdiv_qr_ui (mpz_t q, mpz_t r, mpz_t n, unsigned long int d)`

divides the `mpz_t` n by the `unsigned long int` d setting both q and r to the quotient and the remainder respectively. That both the quotient and the remainder are returned by this function is indicated by the "`qr`" of its name. The rounding is an upper rounding, which is indicated by the "`c`" of "`cdiv`" which means "ceil".

There exist many other division functions whose names are derived in the same way.

Here are some other functions also related to "division" in some way:

- `void mpz_mod (mpz_t r, mpz_t n, mpz_t d)`
- `void mpz_divexact (mpz_t q, mpz_t n, mpz_t d)`
- `void mpz_divexact_ui (mpz_t q, mpz_t n, unsigned long d)`
- `int mpz_divisible_p (mpz_t n, mpz_t d)`
- `int mpz_congruent_p (mpz_t n, mpz_t c, mpz_t d)`
- ...

2.2.6 Exponentiation functions

- `void mpz_powm (mpz_t rop, mpz_t base, mpz_t exp, mpz_t mod)`
- `void mpz_powm_ui (mpz_t rop, mpz_t base, unsigned long int exp, mpz_t mod)`

Both functions compute the modular exponentiation $base^{exp}$ mod mod. The only difference is whether the type of the exponent is a big integer or not. Negative exponents are allowed provided that *base* is invertible modulo *mod*.

- `void mpz_powm_sec (mpz_t rop, mpz_t base, mpz_t exp, mpz_t mod)`

Function `mpz_powm_sec` is an equivalent of the `mpz_powm` function. Both compute exactly the same but the implementation of the former aims at being secure against time analysis attacks.

In the case of non modular exponentiations (in $\mathbb{Z}$), the exponent type is always an `unsigned long int` since raising to a power greater than 2^{64} would necessarily result in a value that can not be represented by the `mpz_t` type (ask yourself why).

- `void mpz_pow_ui (mpz_t rop, mpz_t base, unsigned long int exp)`
- `void mpz_ui_pow_ui`
  `(mpz_t rop, unsigned long int base, unsigned long int exp)`

2.2.7 Many other mathematical functions

There exist functions which compute square roots, others that test whether an integer is a perfect square or a perfect power.

Here are some other functions quite useful in asymmetric cryptography:

- `int mpz_probab_prime_p (mpz_t n, int reps)`

Function `mpz_probab_prime_p` executes *reps* iterations of the Miller-Rabin test to assess the primality of n. It returns either value 2, 1 or 0 whether n is declared as being certainly prime, probably prime or composite, respectively.

Example

```
mpz_init(z_n);

...   // input the value of n

result = mpz_probab_prime_p(z_n, 4);

printf("mpz_probab_prime_p() returns %u  =>  n is ", result);
switch (result)
{
  case 0 :
    printf("composite\n");
```

```
      break;
    case 1 :
      printf("probably prime\n");
      break;
    case 2 :
      printf("(provably) prime\n");
      break;
  }

  mpz_clear(z_n);
}
```

- `void mpz_nextprime (mpz_t rop, mpz_t op)`

Function `mpz_nextprime` determines the smallest prime integer that is strictly greater than *op*.

- `void mpz_gcd (mpz_t rop, mpz_t op1, mpz_t op2)`
- `void mpz_gcdext (mpz_t g, mpz_t s, mpz_t t, mpz_t a, mpz_t b)`

Function `mpz_gcd` computes the GCD (greatest common divisor) value between op_1 and op_2 by means of the Euclidean algorithm.

Function `mpz_gcdext` computes the GCD between a and b by means of the extended version of Euclidean algorithm. Upon execution, g is the GCD value, whereas s and t are the two integers that satisfy Bezout relation:

$$as + bt = g$$

Notice that if a and b are coprime integers, then the value of s (resp. t) is equal to the inverse of a modulo b (resp. the inverse of b modulo a).

- `void mpz_lcm (mpz_t rop, mpz_t op1, mpz_t op2)`

Function `mpz_lcm` computes the LCM (least common multiple) value of op_1 and op_2.

- `int mpz_invert (mpz_t rop, mpz_t op1, mpz_t op2)`

Function `mpz_invert` is quite useful for cryptography as it computes the inverse of op_1 modulo op_2.

There exist several other mathematical functions, most of them being not so interesting for cryptographic purpose.

2.2.8 Comparison functions

GMP provides comparison functions that all return a value of type `int` which is:

- negative if the first operand is strictly less than the second one,
- zero if both operands are equal,
- positive if the first operand is strictly greater than the second one.

For instance, one has the following functions:

- `int mpz_cmp (mpz_t op1, mpz_t op2)`
- `int mpz_cmp_ui (mpz_t op1, unsigned long int op2)`
- `int mpz_cmpabs (mpz_t op1, mpz_t op2)`
- `int mpz_cmpabs_ui (mpz_t op1, unsigned long int op2)`

The two last functions above compare absolute values of their operands.

There also exists the sign function which returns $+1$ if $op > 0$, 0 if $op = 0$ and -1 if $op < 0$:

- `int mpz_sgn (mpz_t op)`

2.2.9 Logical and bitwise functions

Following functions evaluate bitwise AND, OR, XOR and NOT operations on large integers.

- `void mpz_and (mpz_t rop, mpz_t op1, mpz_t op2)`
- `void mpz_ior (mpz_t rop, mpz_t op1, mpz_t op2)`
- `void mpz_xor (mpz_t rop, mpz_t op1, mpz_t op2)`
- `void mpz_com (mpz_t rop, mpz_t op)`

Function `mpz_popcount` computes the Hamming weight of a positive integer, which is the number of "1" digits in the binary representation of its value.

Function `mpz_hamdist` computes the Hamming distance between two positive integers, which is the number of bit positions where their bit values differ.

- `mp_bitcnt_t mpz_popcount (mpz_t op)`
- `mp_bitcnt_t mpz_hamdist (mpz_t op1, mpz_t op2)`

Following functions modify or return individual bit values:

- `void mpz_setbit (mpz_t rop, mp_bitcnt_t bit_index)`
- `void mpz_clrbit (mpz_t rop, mp_bitcnt_t bit_index)`
- `void mpz_combit (mpz_t rop, mp_bitcnt_t bit_index)`
- `int mpz_tstbit (mpz_t op, mp_bitcnt_t bit_index)`

The first three ones respectively force to 1, force to 0, or flip (complement) the bit value at position `bit_index` in the large integer. Notice that position 0 corresponds to the least significant bit.

Last function above returns the value of the bit at position `bit_index` in the large integer.

2.2.10 Formated input/output functions

GMP provides functions for formated input/output. They are equivalent to the family of `printf` and `scanf` functions which interpret the large integer thanks to the new format modifier "`Z`" that must be added to the original format specifier.

For example, instead of `"%d"` (signed integer expressed in decimal) or `"%X"` (unsigned integer expressed in uppercase hexadecimal), one must respectively use `"%Zd"` or `"%ZX"` for writing or reading an `mpz_t` value.

The names of these functions are obtained by prepending the original function name with "`gmp_`". Examples: `gmp_printf`, `gmp_scanf`, `gmp_fprintf`, `gmp_sscanf`, ...

Using these functions makes the program more compact and easily readable. For instance one can write:

```
...
gmp_printf("%Zd * %u = %Zd\n", z_n1, a, z_n2);
...
```

instead of:

```
...
mpz_out_str(stdout, 10, z_n1);
printf(" * %u = ", a);
mpz_out_str(stdout, 10, z_n2);
printf("\n");
...
```

It is advised to refer to the GMP manual that gives many details on how to use formated input/output functions.

2.2.11 Generation of pseudo-random numbers

With GMP, as usual in C or other programming languages, a pseudo-random numbers generator is based on using an iterated deterministic mathematical simple function – often a linear congruential function. Whatever the starting value, iterating indefinitely a function which takes its values from a finite set will end by joining a cycle in which this function is periodic. We also notice that such generator is by no way actually random as the future values of its state are completely determined by its current state. It only outputs deterministic values that look random.

One calls pseudo-random sequence the finite sequence of values that the generator outputs while following the cycle. Of course generators are designed in order to have long enough period so that one can hardly notice that the sequence is actually periodic.

One call seed the value of the state a generator is initialized with. It is important to keep in mind that if one gives to the state the same initial value (or if one omits to initialize it), then the sequences of generated numbers will be identical, which is obviously undesirable.

For generating large pseudo-random integers, GMP provides a structure of type `gmp_randstate_t` which contains, (i) the mathematical function that defines the generator itself (several alternative generation methods are available), (ii) a large integer which is the current state of the generator.

Before any use, once a variable of this type has been declared, one must initialize it. This is done thanks to any of the following functions:

- `void gmp_randinit_default (gmp_randstate_t state)`

- `void gmp_randinit_mt (gmp_randstate_t state)`
- `void gmp_randinit_lc_2exp (gmp_randstate_t state)`

Each of these defines a different mathematical function for updating the state of the generator. We will preferably use `gmp_randinit_default` which defines a mathematical function that is suitable in most circumstances.

Once one does not need to generate pseudo-random numbers anymore, one should disallocate the resources that have been allocated for it when initialized. This is done by using the following function:

- `void gmp_randclear (gmp_randstate_t state)`

Seeding the generator As said above, before actually using the generator, one must give an initial value – which is called the seed – to its state. Notice that initializing the generator (e.g. with function `gmp_randclear`) is not the same as initializing its state. Preferably the seed should be random, but at least it should be different from an execution to another. A common way to choose the seed value is to take advantage of the time that endlessly runs. For example, in C language, the function `time(NULL)` returns an `unsigned long int` value which is the number of seconds that elapsed since some fixed reference date. As it changes every second, this value can advantageously be used to seed the state of the generator.

Here are two GMP functions that allow to specify the value of the seed to be used:

- `void gmp_randseed (gmp_randstate_t state, mpz_t seed)`
- `void gmp_randseed_ui (gmp_randstate_t state, unsigned long int seed)`

The first function makes use of a large integer for the seed value, whereas the second one only uses an `unsigned long int`. This last function is thus more suitable if one wants to use the `time()` return value as the seed.

Random generation Here are two of the most useful functions that generate large random integers:

- `void mpz_urandomb (mpz_t rop, gmp_randstate_t state, mp_bitcnt_t n)`
- `void mpz_urandomm (mpz_t rop, gmp_randstate_t state, mpz_t n)`

Function `mpz_urandomb` generates a large random integer uniformly distributed over $\{0, \ldots, 2^n - 1\}$.

Function `mpz_urandomm` generates a large random integer uniformly distributed over $\{0, \ldots, n - 1\}$.

Example of a classical use

```
gmp_randstate_t my_generator;  // "generator" declaration

gmp_randinit_default(my_generator);  // Initialization of the generator
gmp_randseed_ui(my_generateur, time(NULL));  // One gives the seed value
...
// Use of the generator with functions mpz_urandomb and mpz_urandomm
...
gmp_randclear (my_generator);  // One disallocate the generator
```

2.2.12 Other functions

An integer represented by an `mpz_t` type variable is not necessarily "large". It may happen that it is small enough to fit on a basic C type.

It may be interesting to test whether an `mpz_t` fits on a given C type. This is notably useful if one wants to convert its value to this type. Functions exist that perform such a test for most basic integer types:

- `int mpz_fits_ulong_p (mpz_t op)` `// unsigned long int`
- `int mpz_fits_slong_p (mpz_t op)` `// signed long int`
- `int mpz_fits_uint_p (mpz_t op)` `// unsigned int`
- `int mpz_fits_sint_p (mpz_t op)` `// signed int`
- `int mpz_fits_ushort_p (mpz_t op)` `// unsigned short int`
- `int mpz_fits_sshort_p (mpz_t op)` `// signed short int`

These functions return a non zero value if *op* fits in the specified type, and 0 if not.

Two functions perform a test on the parity of a large integer:

- `int mpz_odd_p (mpz_t op)`
- `int mpz_even_p (mpz_t op)`

They return a non zero value if *op* is odd (resp. even), and 0 in the opposite case.

Caution! Both these functions are actually implemented as macros which evaluate their arguments more than once. This may cause a bug if

the expression provided as argument provoques a side effect. It is strongly recommended to use these functions only with simple variables as arguments. See program `parity.c` given in Appendix A.2 as an example of a wrong usage of these functions.

Finally there exists a function that is quite useful since it allows to know the size of a large integer when expressed in some given base b. That is the number of base-b digits, which is the integer value k that satisfies:

$$b^{k-1} \leq n < b^k$$

- `size_t mpz_sizeinbase (mpz_t op, int base)`

This function returns the number of digits necessary to represent *op* in the specified base which may vary between 2 and 62.

Caution! The value returned by the function `mpz_sizeinbase` is guarantied to be correct only in the case where the base is a power of 2. When the base is not a power of 2 the returned value may be either equal to the actual size of n or to this size plus one. This particularity explains why the program that computes factorials given in Appendix A.1 does not use this function to compute the number of decimal digits of the result.

In any case – correct size or added by one – it is possible to use the value returned by `mpz_sizeinbase` for allocating memory to store the value of n in base b as a string of characters.

Example

```
  char* sz_n;
  mpz_t z_n;

  mpz_init(z_n);

  ...   // computations involving z_n

  sz_n = (char *) malloc(2 + mpz_sizeinbase(z_n, 10));
  mpz_get_str(sz_n, 10, z_n);
  printf("sz_n : %s\n", sz_n);

  free(sz_n);

  mpz_clear(z_n);
}
```

Question: why does one allocate two bytes more than the value returned by `mpz_sizeinbase` in this example?

2.3 Exercises

2.3.1 Exercise 1

Write a program that reads an integer value n on the command line, then computes the factorial of n, and displays the result as well as its number of decimal digits.

2.3.2 Exercise 2

Write a program that reads an integer value k on the command line and generates and displays random integers of size at most k bits. Your program must stop once the current value is a multiple of 20 (other variant: once it is prime).

Improve your program so that the generated numbers are different at each execution.

Improve your program so that it displays the value of the seed.

Improve your program so that the user can specify a seed value to be used for initializing the random generator (useful for replaying executions).

2.3.3 Exercise 3

Modify the program of previous exercise so that it generates integers of size:

- exactly k bits
- at most k decimal digits
- exactly k decimal digits

A Examples of basic GMP programs

A.1 fact_3.c

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#include "gmp.h"
#include "math.h"

int main(int argc, char* argv[])
{
  unsigned int bit_size;
  unsigned int i;
  unsigned int n;

  mpz_t z_fact;


  if (argc != 2)
  {
    printf("Usage : %s n  (calcule la factorielle de n)\n", argv[0]);
    exit(-1);
  }

  n = atoi(argv[1]);

  mpz_init(z_fact);
  mpz_set_ui(z_fact, 1);

  i = 1;
  while (i <= n)
  {
    mpz_mul_ui(z_fact, z_fact, i);
    i++;
  }

  printf("\n");
  printf("%u! = ", n);
  gmp_printf("%Zu\n",z_fact);
  printf("\n");

  bit_size = mpz_sizeinbase(z_fact, 2);
  printf("%u bits, %u chiffres\n", bit_size, (unsigned int) ceil(bit_size / (log(10) / log(2))));

  printf("\n");

  mpz_clear(z_fact);

  return 0;
}
```

A.2 parity.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include "gmp.h"

int main(int argc, char* argv[])
{
  unsigned int i;

  gmp_randstate_t prng;

  mpz_t z_n[10];


  gmp_randinit_default(prng);
  gmp_randseed_ui(prng, time(NULL));

  for (i = 0; i < 10; i++)
    mpz_init(z_n[i]);

  for (i = 0; i < 10; i++)
    mpz_urandomb(z_n[i], prng, 100);

  printf("\n");
  for (i = 0; i < 10; i++)
    gmp_printf("%u (%p) : %Zd\n", i, z_n[i], z_n[i]);

  printf("\n");
  i = 0;
  while (i < 10)
  {
    gmp_printf("%u (%p) : %Zd  =>  ", i, z_n[i], z_n[i]);
    if (mpz_odd_p(z_n[i]))
      printf("impair\n");
    else
      printf("pair\n");
    i++;
  }

  printf("\n");
  i = 0;
  while (i < 10)
  {
    gmp_printf("%u (%p) : %Zd  =>  ", i, z_n[i], z_n[i]);
    if (mpz_odd_p(z_n[i++]))
      printf("impair\n");
    else
      printf("pair\n");
  }
  printf("\n");

  gmp_randclear(prng);

  for (i = 0; i < N; i++)
    mpz_clear(z_n[i]);

  return 0;
}
```

A.3 power_mod.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include "gmp.h"

int main(int argc, char* argv[])
{
  gmp_randstate_t prng;

  signed int i;

  unsigned int bit_size;

  mpz_t z_base, z_exponent, z_modulus, z_result;


  gmp_randinit_default(prng);
  gmp_randseed_ui(prng, time(NULL));

  mpz_inits(z_base, z_exponent, z_modulus, z_result, NULL);

  if (argc != 2)
  {
    printf("Usage : %s bit_size\n", argv[0]);
    exit(-1);
  }

  bit_size = atoi(argv[1]);

  mpz_urandomb(z_base, prng, bit_size);
  mpz_urandomb(z_exponent, prng, bit_size);
  mpz_urandomb(z_modulus, prng, bit_size);

  mpz_set_ui(z_result, 1);
  for (i = bit_size - 1; i >= 0; i--)
  {
    // result <-- result^2 mod modulus
    mpz_mul(z_result, z_result, z_result);
    mpz_mod(z_result, z_result, z_modulus);

    if (mpz_tstbit(z_exponent, i) == 1)
    {
      // result <-- result * base mod modulus
      mpz_mul(z_result, z_result, z_base);
      mpz_mod(z_result, z_result, z_modulus);
    }
  }

  printf("\n");
//  gmp_printf("%Zd ^ %Zd mod %Zd = %Zd\n", z_base, z_exponent, z_modulus, z_result);
  printf("\n");

  gmp_randclear(prng);
```

```
  mpz_clears(z_base, z_exponent, z_modulus, z_result, NULL);

  return 0;
}
```