Université de Limoges

Développement Logiciel Cryptographique

Initiation à GMP

(GNU Multiple Precision)

1 Introduction

1.1 Motivation

GMP (GNU Multiple Precision arithmetic library) est une bibliothèque de calcul sur grands entiers.

Cette bibliothèque fournit de nombreuses fonctions de calcul sur différents types multi-précision :

— (grands) entiers : $\mathbb{Z}$
— (grands) rationnels : $\mathbb{Q}$
— (grands) flottants : $\mathbb{R}$ (voir aussi la bibliothèque MPFR)

Le terme *multi-précision* signifie qu'un nombre (appartenant à $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$) est représenté grâce à autant de "mots" (du type de base de la machine) qu'il est nécessaire pour le représenter avec la précision requise pour le calcul effectué.

Exemple Programme de calcul de la fonction factorielle sans 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;
}
```

Si on exécute ce programme pour calculer la factorielle de 12, cela donne le résultat $12! = 479\,001\,600$ qui est correct. En revanche le calcul de la factorielle de 13 provoque un débordement par rapport à la valeur maximale représentable sur 32 bits ($2^{32} - 1 = 4\,294\,967\,295$). On obtient $13! = 1\,932\,053\,504$ qui est erroné. (On remarque que $13! \bmod 2^{32} = 1\,932\,053\,504$.)

On pourrait penser qu'il suffit de choisir le type `long` ou `long long` mais ça ne fait que retarder (pas beaucoup) le moment où apparaîtra ce même type de débordement.

Exemple Programme de calcul de la fonction factorielle avec 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;
}
```

Ce programme peut calculer facilement

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

qui est un entier nécessitant 108 bits – soit 4 mots (`unsigned int`) de 32 bits – pour être représenté.

Il est tout aussi facile de calculer 2000! qui vaut

```
331627509245063324117539338057632403828111720810578039457193543706038\
0\
779056008224002732308597325922554023529412258341092580848174152937961\
3\
138663352634368890563405855616394060511725257187064785639354404540524\
3\
957467037674108722970434684158343752431580877533645127487995436859247\
4\
080324089465615072332506527976557571796715367186893590561128158716017\
1\
723265715611000421401242043384257371270017588354779689992128352899666\
5\
853405579854903657366350133386550401172012152635488038268152152246920\
9\
952060315644185654806759464970515522882052348999957264508140655366789\
6\
953210146762267133202683155220519449446161823927520402652972263150257\
4\
752048296064750927394165856283531779574482876314596450373991327334177\
2\
636088524900935066216101444597094127078213137325638315723020199499149\
5\
831647094277447387032798554967429860883937632682415247883438746959582\
9\
257740574539837501585815468136294217949972399813599481016556563876034\
2\
273129122503847098729096266224619710766059315502018951355831653578714\
9\
229091677904970224709461193760778516511068443225590564873626653037738\
4\
650390788049524600712549402614566072254136302754913671583406097831074\
9\
452822174907813477096932415561113398280513586006905946199652573107411\
7\
708151992256451677857145805660218565476095237746301667942248844448579\
8\
349801548032620829890965857381751888619376692828279888453584639896594\
2\
139529844652910920091037100461494499158285880507618679249463851808798\
7\
451289140801934007462592005709872957859964365065589561241023101869055\
6\
060308783629110505601245908998383410799367902052076858669183477906558\
5\
447001486926569246319333376124280974200671728463619392496986284687199\
9\
345039388936727048712717273456170035486747750910295552395354794110742\
1\
913301356819541091941462766417542161587625262858089801222443890248677\
1\
820549594157519917012717675717874958616196659318788551418357820926014\
8\
207177733173539603430496908207058995870138198081303559016076290838857\
4\
561288217698136182483576739218303118414719133986892842344000779246691\
2\
097667316514334944374732356365720488444783318549416930301245316762327\
4\
536787932284747382448509228313995250973250597912703104768360148119110\
2\
229253372697693823670057565612400290576043852852902937606479533458179\
6\
661238396052625491071866638693547661084550461981020840506358276765265\
8\
949239324951968595417167241932953068367349554400458635983816104305944\
9\
826627530605423580755894108278880427825951089880635410567917950974017\
7\
806887828698102190109001483520616888837202503106659220686014836498305\
3\
278208826353655804360568678128416921713304714117631217589577712263758\
4\
753123517230990549829210134687304205898014418063875382664169897704237\
7\
594062808772537022654265305808623793014226758211871435029186376363403\
0\
017325181826207603974736959520264263236414544685111342720215045838385\
1\
010136941313034856221916631623892632765815355011276307825059969158824\
5\
334574354378636831737306732965893551996944582368735088302786577008797\
4\
988999234355556624068283476378468518384497364887395247510322422211056\
1\
201295829657191368108693825475764118886879346725191246192151144738836\
2\
695916436724900716534282281526612478004639225449451703637236279407577\
8\
454209104830546165619062217428698160297332404652020199281385488268195\
1\
```

```
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\
00000000000000000000000000000000000000000000000000000000000000000000 00
```

C'est un entier de 19053 bits – 5736 chiffres décimaux – qui nécessite seulement 5 millisecondes pour être calculé.

Dernier exemple, la factorielle de 100 000 (1 516 705 bits) est calculée en moins d'une seconde. C'est un nombre de 456 574 chiffres décimaux qui nécessite à lui seul 185 kilo-octets pour être représenté en mémoire.

1.2 Compilation d'un programme utilisant la librairie GMP

Pour avoir accès aux fonctions de la librairie depuis votre programme appelé `mon_programme.c`, vous devez faire deux choses :

1. Inclure dans `mon_programme.c` le fichier `gmp.h` avec la directive `#include <gmp.h>`. Ce fichier contient toutes les déclarations de types et de constantes, ainsi que les prototypes des fonctions qui composent la librairie.
2. Faire l'édition de lien (on dit souvent *linker*) avec la librairie GMP. Cela permet de créer l'exécutable en associant au code objet de votre programme (le fichier intermédiaire `mon_programme.o` créé par une première étape de votre compilation) le code objet des différentes fonctions de la librairie.

Compilation en deux étapes :

— `gcc -c mon_programme.c` ⇒ compile le source `mon_programme.c` et crée le fichier objet `mon_programme.o`,
— `gcc mon_programme.o -lgmp -o mon_programme` ⇒ crée l'exécutable `mon_programme` en faisant l'édition de lien des codes objets se trouvant dans le fichier `mon_programme.o` et la librairie GMP.

Compilation en une seule étape :

— `gcc mon_programme.c -lgmp -o mon_programme` ⇒ compile et fait l'édition de lien dans la foulée.

2 Utilisation de la bibliothèque

2.1 Principes de base

Les types de bases définis dans GMP sont les suivants :

— `mpz_t` pour les entiers relatifs ($\mathbb{Z}$)
— `mpq_t` pour les rationnels ($\mathbb{Q}$)
— `mpf_t` pour les flottants ($\mathbb{R}$)

Remarque : pour une utilisation cryptographique, nous ne nous intéresserons qu'au type "grand entier" ($\mathbb{Z}$) de GMP (type `mpz_t`).

Ces types de base de GMP sont des **pointeurs vers des structures** qui contiennent tout le nécessaire pour pouvoir gérer l'aspect multi-précision.

Exemples de déclarations

```
mpz_t nombre;

struct {
  mpz_t x;
  mpz_t y;
  mpz_t z;
} point;  // Par exemple, point sur une courbe elliptique
          // (en coordonnées projectives)
```

```
mpz_t tab[10];
```

Même si nous n'avons pas nécessairement besoin de connaître ces détails, voici les champs que contient la structure pointée par le type `mpz_t` :

- `mp_limb_t *_mp_d` : un pointeur vers un tableau de *limbs* (`unsigned int`) qui contient, en base 2^{32}, la valeur de l'entier représenté.
- `int _mp_alloc` : le nombre de limbs alloués pour le tableau
- `int _mp_size` : le nombre de limbs sur lequel est effectivement représentée la valeur (peut être plus petit que le nombre de limbs alloués)

Le tableau contenant la valeur est alloué dynamiquement et sa taille peut être augmentée en cours de calcul si besoin est. GMP est donc plus qu'une bibliothèque de calcul en multi-précision, c'est une bibliothèque de calcul en précision arbitraire.

Lorsqu'une variable de type `mpz_t` est déclarée, il s'agit donc d'un pointeur non initialisé qui ne pointe sur rien. Aucune structure (cf. ci-dessus) n'est allouée pour décrire le nombre. A fortiori, aucun espace n'est alloué pour le stockage de la valeur.

Toutes les fonctions qui réalisent de l'arithmétique sur les entiers (`mpz_t`) ont leur nom qui commence par `mpz_`.

Remarque : Puisque le type `mpz_t` est un type pointeur vers la structure contenant la donnée et ses méta-données, le passage d'un grand entier comme paramètre d'une fonction est naturellement un passage par adresse. Il n'y aura donc pas besoin de préfixer la variable du signe "`&`" pour que la fonction puisse en modifier le contenu.

2.2 Description des fonctions les plus utiles

2.2.1 Fonctions d'initialisation et de libération d'un entier

- `void mpz_init (mpz_t x)` : initialise une variable de type `mpz_t`, c'est à dire crée la structure avec entre autre un tableau de limbs qui contient la valeur 0. C'est un équivalent de `malloc()` pour le type `mpz_t`.
- `void mpz_inits (mpz_t x, ..., NULL)` : permet d'initialiser plusieurs variables en une seule fois.
- `void mpz_clear (mpz_t x)` : détruit l'objet pointé par la variable (la structure) et libère la mémoire qu'elle occupait. Il s'agit donc tout d'abord du tableau de limbs contenant la valeur entière, mais également de la structure elle-même. C'est un équivalent de `free()` pour le type `mpz_t`.
- `void mpz_clears (mpz_t x, ..., NULL)` : permet de détruire plusieurs variables en une seule fois.

Exemple

```
mpz_t z_n;        // z_n est déclaré mais non alloué

mpz_init(z_n);    // z_n pointe maintenant sur un vrai entier (0)
...
// On calcule avec la variable z_n
...
mpz_clear(z_n);   // On libère l'objet pointé par z_n
                  // lorsque les calculs sont terminés
```

2.2.2 Fonctions d'assignation d'une valeur

- `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)`

Ces fonctions permettent de faire un équivalent de `rop = op;` mais avec `rop` de type `mpz_t`. La fonction utilisée dépend du type de `op`.

Nous n'utiliserons le plus souvent que les trois premières de ces fonctions.

Exemple

```
mpz_t z_n1, z_n2;

mpz_init(z_n1);
mpz_init(z_n2);
mpz_set_ui(z_n1, 1234);   // Donne à z_n1 la valeur 1234
mpz_set(z_n2, z_n1);      // Recopie la valeur de z_n1 dans z_n2
```

- `int mpz_set_str (mpz_t rop, char *str, int base)` : assignation à un `mpz_t` d'une valeur textuelle en précisant en quelle base est représenté l'entier dans la chaine de caractères. À noter que `str` peut être une variable ou une chaine littérale. Si `base` est égal à 0 alors la base est déterminée par le préfixe de la chaine : `0x` (base 16), `0b` (base 2), `0` (base 8), et en base 10 à défaut d'un préfixe reconnu.

On peut donc écrire par exemple :

```
mpz_set_str(z_n, "84973918290487975931843982749165193191035 80", 0);
mpz_set_str(z_n, "55555555555555555555555555555555555", 6);  // 6^35-1
```

— `void mpz_swap (mpz_t rop1, mpz_t rop2)` : pour échanger les valeurs de deux `mpz_t`.

D'autres fonctions permettent d'initialiser et assigner une valeur en même temps. Ces fonctions ont pour nom celui de la fonction d'assignation avec "`init_`" avant "`set`". Par exemple :

```
mpz_t z_n;
mpz_init_set_ui(z_n, 1234);  // Initialise puis assigne la valeur 1234
```

2.2.3 Fonctions de conversion

Ces fonctions permettent de convertir un `mpz_t` vers un type de base du C. En voici quelques exemples :

— `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)`

La valeur du `mpz_t` doit préférablement être suffisamment petite pour tenir dans le type de destination. (cf. le manuel pour plus de détails)

2.2.4 Fonctions arithmétiques

— `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)`

Exemple

```
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
```

Il existe aussi des fonctions permettant de réaliser soustraction et multiplication, ainsi que des multiplications dans lesquelles le produit calculé est ajouté ou retranché de l'opérande de sortie (cf. le manuel).

— `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)`

La fonction `mpz_mul_2exp` multiplie op_1 par 2^{op_2}. La fonction `mpz_neg` calcule l'opposé de *op*. La fonction `mpz_abs` calcule la valeur absolue de *op*.

2.2.5 Fonctions de divisions

Il existe tout un ensemble de fonctions de divisions qui se déclinent notamment en fonction du (des) résultat(s) retourné(s) (quotient, reste, ou quotient et reste), ainsi que du type d'arrondi (inférieur, supérieur, vers 0). Voir le manuel pour tous les détails.

Un exemple de fonction de division :

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

permet de faire la division du `mpz_t` n par le `unsigned long` d en récupérant le quotient dans q et le reste dans r, l'arrondi se faisant vers la valeur supérieure (le "`c`" de "`cdiv`" signifiant "ceil")

Il en existe beaucoup d'autres construites sur le même modèle.

Voici quelqu'autres exemples de fonctions de type "division" :

— `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 Fonctions d'exponentiation

— `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)`

Ces deux fonctions calculent l'exponentielle modulaire $base^{exp}$ mod mod. La différence tient dans le fait que l'exposant est de type grand entier ou non. Les exposants négatifs sont autorisés si $base$ est inversible modulo mod.

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

La fonction `mpz_powm_sec` est équivalente à `mpz_powm` d'un point de vue fonctionnel mais son implémentation vise à sécuriser l'exponentiation vis-à-vis des attaques par analyse de temps d'exécution.

Dans le cas des exponentielles dans $\mathbb{Z}$ (non modulaires), le type de l'exposant est nécessairement `unsigned long int` car élever à une puissance ne tenant pas sur ce type conduirait immanquablement à dépasser la capacité de représentation d'un `mpz_t` (demandez-vous pour quelle raison).

— `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 Beaucoup d'autres fonctions mathématiques

Il existe des fonctions d'extraction de racine carrée, ainsi que des tests pour savoir si un entier est un carré parfait ou une puissance parfaite.

Voici d'autres fonctions qui sont très utiles en cryptographie asymétrique :

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

La fonction `mpz_probab_prime_p` exécute *reps* itérations du test de Miller-Rabin pour tester la primalité de n. Elle renvoie la valeur 2, 1 ou 0 selon que n est déclaré certainement premier, probablement premier ou composé respectivement.

Exemple

```
  mpz_init(z_n);

  ...   // saisie de la valeur de n

  result = mpz_probab_prime_p(z_n, 4);

  printf("mpz_probab_prime_p() retourne %u  =>  n est ", result);
  switch (result)
  {
    case 0 :
      printf("composé\n");
      break;
    case 1 :
      printf("probablement premier\n");
      break;
    case 2 :
      printf("premier\n");
      break;
  }

  mpz_clear(z_n);
}
```

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

La fonction `mpz_nextprime` détermine le plus petit entier premier strictement plus grand que *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)`

La fonction `mpz_gcd` calcule le PGCD de op_1 et op_2 par l'algorithme d'Euclide.

La fonction `mpz_gcdext` calcule le PGCD de a et b par l'algorithme d'Euclide étendu. Après exécution, g contient la valeur du PGCD et s et t sont deux entiers vérifiant l'identité de Bezout :

$$as + bt = g$$

À noter que si a et b sont premiers entre eux alors la valeur de s (resp. t) est égale l'inverse de a modulo b (resp. l'inverse de b modulo a).

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

La fonction `mpz_lcm` calcule le PPCM de op_1 et op_2.

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

La fonction `mpz_invert` est très utile en cryptographie car elle permet de calculer l'inverse de op_1 modulo op_2.

Il existe encore bien d'autres fonctions mathématiques, pas très utiles en cryptographie pour la plupart.

2.2.8 Fonctions de comparaison

GMP fournit des fonctions de comparaison qui retournent toutes une valeur de type `int` qui est :

— négative si le premier opérande est plus petit que le deuxième,
— nulle si le premier opérande est égal au deuxième,
— positive si le premier opérande est plus grand que le deuxième.

On a par exemple les fonctions suivantes :

— `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)`

Les deux dernières fonctions ci-dessus comparent les valeurs absolues des deux opérandes.

Il existe également la fonction d'extraction de signe qui renvoie $+1$ si $op > 0$, 0 si $op = 0$ et -1 si $op < 0$:

— `int mpz_sgn (mpz_t op)`

2.2.9 Fonctions logiques et de comparaison de bits

Les fonctions suivantes permettent d'évaluer les opérations ET, OU, XOR et NON bit à bit sur des grands entiers.

— `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)`

La fonction `mpz_popcount` calcule le poids de Hamming d'un entier positif, c'est à dire le nombre de 1 présents dans l'écriture binaire de ce nombre.

La fonction `mpz_hamdist` calcule la distance de Hamming entre deux entiers positifs, c'est à dire, dans leurs écritures binaires, le nombre de positions où leurs bits diffèrent.

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

Les fonctions suivantes permettent d'agir sur, ou de tester, un bit individuellement :

— `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)`

Les trois premières fonctions ont pour effets respectifs de forcer à 1, forcer à 0 et complémenter la valeur du bit en position `bit_index` dans le grand entier. La dernière fonction renvoie la valeur du bit en position `bit_index` dans le grand entier.

Une valeur de `bit_index` égale à 0 correspond au bit de poids faible.

2.2.10 Fonctions d'entrées/sorties

Les deux fonctions suivantes permettent d'écrire ou de lire un entier `mpz_t` vers ou depuis un flot [1]. L'entier est lu ou écrit sous forme de chaine de caractères représentant cette valeur dans la base spécifiée. La base peut varier de 2 à 62 ou de -2 à -36. (cf. le manuel pour des explications supplémentaires sur l'interprétation de la base.)

Ces fonctions renvoient le nombre de caractères écrits ou lus, ou bien 0 en cas d'erreur.

— `size_t mpz_out_str (FILE *stream, int base, mpz_t op)`
— `size_t mpz_inp_str (mpz_t rop, FILE *stream, int base)`

1. Ce flot peut être associé soit à un fichier, soit aux classiques entrée et sortie standard ou à la sortie erreur

Exemple

```
...
mpz_init(z_n);

mpz_ui_pow_ui(z_n, 2, 1000);

printf("2 ^ 1000 = ");
mpz_out_str(stdout, 10, z_n);
printf("\n");

mpz_clear(z_n);
```

Les deux fonctions suivantes sont des analogues des deux premières mais la lecture et l'écriture se font dans un format "binaire" plutôt que sous forme de chaine de caractères. C'est plus économe en place mémoire mais moins commode à manier.

— `size_t mpz_out_raw (FILE *stream, mpz_t op)`
— `size_t mpz_inp_raw (mpz_t rop, FILE *stream)`

Pour toutes ces fonctions, une valeur de `stream` égale à `NULL` a pour effet d'écrire sur *stdout* ou de lire sur *stdin*.

2.2.11 Fonctions d'entrées/sorties formatées

GMP fournit des équivalents des fonctions des familles de `printf` et de `scanf` permettant d'interpréter les grands entiers grâce au nouveau spécificateur de format "`Z`" à rajouter devant le spécificateur de format habituel.

Par exemple, à la place de `"%d"` (entier signé exprimé en décimal) ou `"%X"` (entier non signé exprimé en hexadécimal majuscule), on pourra utiliser `"%Zd"` ou `"%ZX"` respectivement lorsqu'il s'agira d'écrire ou de lire un entier `mpz_t`.

Les noms de ces fonctions s'obtiennent en préfixant avec "`gmp_`" le nom de la fonction habituelle. On aura donc par exemple les fonctions `gmp_printf`, `gmp_scanf`, `gmp_fprintf`, `gmp_sscanf`, ...

L'utilisation de ces fonctions permet un code plus compact et lisible. Par exemple ou pourra écrire :

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

à la place de :

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

Il est conseillé de se référer au manuel qui donne de nombreux détails sur l'utilisation de ces fonctions.

2.2.12 Génération de nombres pseudo-aléatoires

Principe général Dans GMP, comme habituellement en C ou dans d'autres langages, un générateur des nombres pseudo-aléatoires est basé sur l'utilisation itérée d'une fonction mathématique simple – souvent une fonction linéaire congruentielle. Quelle que soit la valeur de départ, itérer indéfiniment une fonction à valeurs dans un espace fini amènera immanquablement à rejoindre un cycle dans lequel cette fonction est périodique. On remarque également qu'un tel générateur n'a rien d'aléatoire puisque les valeurs futures de son état sont complètement déterminées à partir de la valeur courante.

On appelle séquence pseudo-aléatoire la suite finie des diverses valeurs que peut "sortir" un générateur. Bien sûr les générateurs sont conçus pour avoir des périodes suffisamment grandes pour que l'on ne détecte pas facilement cet effet de parcours de cycle.

On appelle graine l'état dans lequel on initialise le générateur. Il est important d'avoir à l'esprit que si on donne à cet état initial toujours la même valeur (ou si on omet de l'initialiser), les séquences de nombres générés seront identiques, ce qui est bien sûr non souhaitable.

Initialisation (création) d'un générateur d'aléa sous GMP Pour la génération de grands entiers pseudo-aléatoires, GMP définit une structure de type `gmp_randstate_t` composée, d'une part d'une fonction mathématique définissant le générateur proprement dit (plusieurs algos de génération sont disponibles), d'autre part d'un grand entier définissant l'état courant du générateur.

Avant toute utilisation, après avoir déclaré une variable de ce type, il est nécessaire de l'initialiser. Ceci se fait à l'aide d'une des fonctions suivantes :

- `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)`

Chaque fonction permet d'utiliser une fonction mathématique différente pour la génération. Nous utiliserons `gmp_randinit_default` qui utilise une fonction mathématique qui convient dans la plupart des circonstances.

Lorsque l'on a terminé l'utilisation du générateur aléatoire, il est nécessaire de libérer les ressources qui ont été allouées lors de l'initialisation précédente. Cela se fait grâce à la fonction :

— `void gmp_randclear (gmp_randstate_t state)`

Spécifier une graine au générateur Comme dit plus haut, avant toute utilisation effective du générateur il faut lui donner une graine – c'est à dire une valeur d'initialisation de son état – qui, à défaut d'être aléatoire, devra au moins posséder la propriété de ne pas être la même valeur à chaque exécution du programme. Une façon habituelle de choisir la valeur de la graine est d'utiliser le temps qui passe et qui ne revient jamais. Par exemple, en langage C, un appel à la fonction `time(NULL)` fournit une valeur de type `unsigned long int` représentant le nombre de secondes écoulées depuis une date de référence fixe. Cette valeur est idéale pour servir de graine.

Voici les deux fonctions que propose GMP pour attribuer une valeur de graine au générateur pseudo-aléatoire :

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

La première de ces fonctions utilise un grand entier comme valeur de graine, alors que la deuxième utilise un `unsigned long int`. C'est donc cette deuxième fonction qu'il faudra utiliser si on souhaite donner comme valeur de graine la sortie de la fonction `time`.

Génération d'aléa Deux des fonctions les plus utiles pour la génération de grands entiers aléatoires sont les suivantes :

— `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)`

La fonction `mpz_urandomb` génère un aléa uniformément distribué sur l'intervalle $\{0, \ldots, 2^n - 1\}$.

La fonction `mpz_urandomm` génère un aléa uniformément distribué sur l'intervalle $\{0, \ldots, n - 1\}$.

Exemple d'utilisation classique

```
gmp_randstate_t mon_generateur;  // Déclaration d'un "générateur"

gmp_randinit_default(mon_generateur);  // Initialisation du générateur
gmp_randseed_ui(mon_generateur, time(NULL));  // On donne la graine
...
// Utilisation du générateur
// avec les fonctions mpz_urandomb et mpz_urandomm
...
gmp_randclear (mon_generateur);  // On "désalloue" le générateur
```

2.2.13 Diverses autres fonctions

Un grand entier représenté par une variable de type `mpz_t` n'est pas nécessairement "grand". Il se peut qu'il soit suffisamment petit pour pouvoir être représenté par un type de base.

Il peut être intéressant de tester si un `mpz_t` tient dans un type de base donné, notamment en vue d'une conversion vers ce type. Des fonctions permettent d'effectuer ce test pour la plupart des types entiers :

- `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`

Ces fonctions retournent une valeur non nulle si la valeur *op* tient dans le type de base, et 0 dans le cas contraire.

Deux fonctions permettent de tester la parité d'un entier :

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

Elles retournent une valeur non nulle si *op* est impair (resp. pair), et 0 dans le cas contraire.

Attention ! Ces deux fonctions sont implémentées sous la forme de macros qui évaluent leur argument plus d'une fois. Cela peut être source de bug si l'expression fournie comme argument provoque un effet de bord. N'utilisez ces fonctions qu'avec des arguments qui sont de simples valeurs. Voir le programme `parite.c` donné en Annexe A.2 comme exemple de mauvaise utilisation de ces fonctions.

Enfin il existe une fonction très utile qui permet de connaître la taille d'un grand entier n en base b, c'est à dire l'entier k strictement positif vérifiant :

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

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

Cette fonction retourne le nombre de chiffres nécessaires pour représenter *op* dans la base spécifiée qui peut varier entre 2 et 62.

Attention ! La valeur retournée par la fonction `mpz_sizeinbase` n'est garantie exacte que si la base est une puissance de 2. Lorsque la base n'est pas une puissance de 2 la valeur retournée peut être égale soit à la taille de n, soit à cette taille augmentée de 1. Cette particularité explique notamment pourquoi le programme de calcul de factorielle donné en Annexe A.1 n'utilise pas cette fonction pour calculer le nombre de chiffres décimaux du résultat.

Dans tous les cas de figure – nombre de chiffres exact ou augmenté de 1 – il est possible d'utiliser la valeur retournée par `mpz_sizeinbase` pour allouer la mémoire nécessaire pour stocker la valeur de n en base b dans un chaine de caractères.

Exemple

```
  char* sz_n;
  mpz_t z_n;

  mpz_init(z_n);

  ...   // calculs effectués sur 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 : pourquoi alloue-t-on deux octets de plus que la valeur renvoyée par la fonction `mpz_sizeinbase` ?

2.3 Exercices

2.3.1 Exercice 1

Écrivez une fonction qui lit un entier n sur la ligne de commande, puis qui calcule la factorielle de n, affiche le résultat ainsi que le nombre de chiffres décimaux du résultat.

2.3.2 Exercice 2

Écrivez une fonction qui lit un entier k sur la ligne de commande, puis génère et affiche des nombres aléatoires d'au plus k bits. Le programme s'arrêtera lorsque le nombre aléatoire généré sera multiple de 20 (variante : sera premier).

Améliorer votre programme pour que les nombres générés soient différents à chaque exécution.

Améliorer votre programme pour qu'il affiche la graine utilisée.

Améliorer votre programme pour qu'il puisse prendre en entrée une valeur de graine à laquelle initialiser le générateur (très utile pour faire du rejeu).

2.3.3 Exercice 3

Modifiez le programme de l'exercice précédent pour que les nombres générés aient une taille :

— exactement égale à k bits
— au plus égale à k chiffres décimaux
— exactement égale à k chiffres décimaux

3 Utilisation avancée

3.1 Si vous voulez une exécution rapide ... ... laissez faire GMP !

3.1.1 L'exponentiation modulaire

Calculer une exponentielle modulaire consiste à élever un entier b (la base) à une puissance enitère e (l'exposant) modulo un autre entier n (le module) :

$$r = b^e \bmod n$$

Lorsqu'il n'est pas besoin de protéger cette exponentiation contre les attaques par analyse de canaux auxiliaires, la méthode classique – appelée *square and multiply* binaire – consiste à parcourir un à un les bits de l'exposant de la gauche vers la droite [2].

Pour chaque bit considéré, la valeur d'un accumulateur (initialisé à 1) est mise à jour, d'une part en l'élevant au carré, d'autre part – et seulement si le bit vaut 1 – en le multipliant par la valeur de la base, tous ces calcul étant effectués modulo le module.

Le résultat de l'exponentielle modulaire est la valeur finale de l'accumulateur lorsque l'exposant a été parcouru entièrement.

Le programme `power_mod.c` donné en Annexe A.3 présente une implémentation de cette méthode. Il prend en entrée une taille (en bits) des opérandes, génère aléatoirement trois valeurs – base, exposant, module – de cette taille, et effectue le calcul de l'exponentielle modulaire.

L'implémentation est classique et semble assez rapide puisqu'elle permet de calculer l'équivalent d'une signature RSA avec une clé de 4 096 bits en moins de 50 ms.

Oui, mais... l'exponentiation modulaire est de classe de complexité cubique, ce qui signifie que le temps d'exécution, bien que polynomial, augmentera assez rapidement avec la taille des entiers considérés. Et en effet, il faut environ 24 secondes de calcul lorsque l'exponentiation modulaire se fait sur des entiers de 50 000 bits.

2. Il existe une variante de cette méthode qui parcourt les bits de l'exposant de la droite vers la gauche.

La bibliothèque GMP fournit sa propre implémentation de l'exponentiation modulaire à travers la fonction `mpz_powm`. Comparons les efficacités de notre fonction, et de celle de GMP. Pour cela, dans le code du programme `power_mod.c`, remplaçons les lignes qui constituent le coeur du calcul par l'appel suivant à la fonction native de GMP :

```
mpz_powm(z_result, z_base, z_exponent, z_modulus)
```

Maintenant le calcul d'une exponentielle modulaire sur des données de 50 000 bits ne prend plus que 12 secondes au lieu de 24. Cette différence est due en partie à l'utilisation d'une méthode d'exponentiation différente (dite 2^k*-ary à fenêtre glissante*), mais aussi à une utilisation plus optimisée des fonctions de la couche basse de GMP essentiellement écrites en assembleur. Il s'agit des fonctions préfixées par `mpn_` (voir chapitre 8 du manuel) qui sont d'un emploi assez délicat pour le néophyte.

Moralité : Si vous voulez calculer une exponentielle modulaire, laissez faire GMP !

3.1.2 La factorielle

Nous avons vu à la Section 1 de ce document un exemple de programme permettant de calculer une factorielle. Il s'agit du programme `fact_3.c` donné en Annexe A.1, et nous avons pu remarqué combien la bibliothèque GMP permettait d'effectuer très rapidement des calculs arithmétiques sur de très grands nombres. Pour rappel, ce programme permet de calculer la factorielle de 2 000 en moins de 10 millisecondes, et même la factorielle de 100 000 – un nombre de 456 574 chiffres décimaux – en moins d'une seconde.

Et bien en réalité notre programme est d'une effrayante inefficacité !

La bibliothèque GMP est également fournie avec une fonction qui calcule la factorielle d'un entier. Il s'agit de la fonction `mpz_fac`.

Lorsque nous voulons calculer la factorielle de 1 million – un nombre de plus de 5,5 millions de chiffres – avec notre programme nous devons attendre 1 minutes et 45 secondes. La fonction native de GMP la calcule en moins de 0,2 seconde !

La Section 16 du manuel d'utilisation de GMP est très intéressante. Elle fournit des détails sur les différentes méthodes utilisées dans les calculs effectués par la bibliothèque. Ceci nous permet de comprendre pourquoi cette bibliothèque est probablement l'une des plus rapides disponibles, tout en étant libre, bien documentée et disponible sur de très nombreux couples architecture / système d'exploitation.

Nous y apprenons notamment que pour calculer la factorielle de n il ne faut surtout pas faire naïvement le produit de tous les entiers de 1 jusqu'à n. La première idée est de compter – voyez-vous comment le faire astucieusement ? – le nombre d'occurrences du facteur 2 dans le résultat. En les

enlevant du produit à calculer, on s'assure d'avoir à effectuer des multiplications sur des entiers bien plus petits. On n'oubliera pas de multiplier à la fin par la puissance de 2 adéquate, ce qui n'est qu'une affaire d'un (grand) décalage à gauche.

Une deuxième optimisation consiste à regrouper les facteurs restant en fonction de leur multiplicité. Par exemple, pour le calcul de 23!, on aura les regroupements suivants :

$$23! = 2^{19}.(3.5)^3.(7.9.11)^2.(13.15.17.19.21.23)$$

Regrouper ainsi les facteurs plutôt que de les multiplier un à un permet d'avoir moins de multiplications mais qui opèrent sur de plus grands opérandes. Ceci est intéressant car il existe des méthodes de multiplications de plus en plus efficaces à mesure que croît la taille des opérandes (Karatsuba,...).

Enfin, pour chaque regroupement, l'optimisation va jusqu'à réaliser le produit de la manière la plus astucieuse. En l'occurrence, il s'agit de grouper les termes de manière alternée. Par exemple par le terme 13.15.17.19.21.23 sera calculé comme le produit de 13.17.21 par 15.19.23 plutôt que comme le produit de 13.15.17 par 19.21.23. En effet, on multiplie ainsi entre eux deux nombres sensiblement de même taille, ce qui est plus efficace que sur deux nombres de tailles très différentes.

Moralité : Si vous voulez calculer une factorielle, laissez faire GMP !

A Exemples de programmes

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 parite.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;
}
```