GMP

Table des matières

GNU MP, également appelée GMP, est une bibliothèque logicielle de calcul multiprécision sur des nombres entiers, rationnels et en virgule flottante.

Les principaux domaines d'applications de GMP sont la recherche et les applications en cryptographie, les logiciels applicatifs de sécurité pour Internet et les systèmes de calcul formel.

1. La cryptographie asymétrique : RSA

Le système RSA est un moyen puissant de chiffrer des données personnelles. Aujourd'hui, il nous entoure sans même que nous le sachions. Il est dans nos cartes bancaires, nos transactions, nos messageries, nos logiciels...

Le système de chiffrement RSA a été inventé par trois mathématiciens : Ron Rivest, Adi Shamir et Len Adleman, en 1977.

1.1. Principe

Dans l'univers de la cryptographie, on distingue deux types de chiffrement : le chiffrement asymétrique et symétrique. Les systèmes symétriques utilisent une seule clé pour chiffrer et déchiffrer (exemple : Chiffre de César). On peut utiliser la métaphore du coffre fort : le coffre fort dispose d'une seule clé qui est la même pour ouvrir et fermer le coffre. En revanche, les systèmes asymétriques utilisent deux clés. Une pour chiffrer et une autre pour déchiffrer. On appelle clé publique la clé servant à chiffrer et clé privée la clé servant à déchiffrer.

La clé publique est visible par tout le monde dans une espèce d'annuaire qui associe à chaque personne sa clé publique.

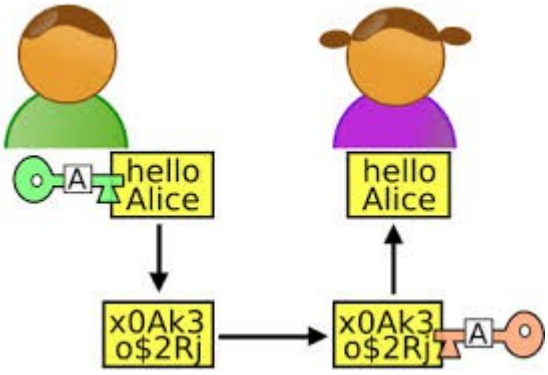
La clé privée n'est visible et connue que par son propriétaire. Il ne faut en aucun cas que quelqu'un d'autre que le propriétaire entre en possession de celle-ci. Si une quelconque personne obtenait une clé privée qui n'est pas la sienne, elle serait alors en mesure de lire les messages chiffrés qui ne lui sont pas destinés.

On peut ainsi voir la clé publique comme une espèce de boite aux lettres (où n'importe qui peut mettre - chiffrer - des messages) et seul le propriétaire de la clé de la boite (clé privée) peut lire - déchiffrer - les messages)

Pour la suite, on va prendre deux personnages : Alice et Bob qui possèdent tout deux une clé publique et une clé privée.

Comment Bob fait-il pour envoyer un message chiffré à Alice ?

Pour chiffrer un message, on utilisait la clé publique. Bob va donc aller sur un annuaire et prendre la clé publique d'Alice. Une fois qu'il la possède, il chiffre le message qu'il veut envoyer à Alice, puis le lui envoie.



Comment Alice déchiffre-t-elle le message reçu ?

Alice vient de recevoir le message chiffré de Bob, grâce à sa clé privée (qu'elle seule possède), elle peut déchiffrer et lire le message que Bob a envoyé.

1. Contrairement à de nombreux systèmes de chiffrement, tel que le chiffrement affine, qui sont des systèmes de chiffrement symétriques, dit à clé secrète, le système RSA ne nécessite pas de transfert de clé entre l'expéditeur et le destinataire. C'est un point de sécurité qui n'est nullement négligeable puisqu'ainsi personne d'autres que les concernés ne peuvent comprendre le message chiffré.
2. Le système RSA, comme tous les systèmes asymétriques, est basé sur les fonctions à sens

uniques. (C'est à dire qu'il est simple d'appliquer la fonction, mais extrêmement difficile de retrouver l'antécédent la fonction à partir de son image seulement). Pour inverser cette fonction, il faut un élément supplémentaire, une aide : la clé privée.

Il existe plusieurs attaques contre le système RSA. Notamment les attaques de Wiener, Hastad, par chronométrage, ou encore l'attaque du milieu.

1.2. Création des clés

1.2.1. Création de la clé publique

Pour créer une clé publique il nous faut choisir deux nombres premiers.

Soit P et Q, ces deux nombres premiers (ex : P = 53 et Q = 97).

Ensuite, on va prendre un autre nombre, N, tel que : N = P × Q (ex : N = 53 x 97 = 5141).

On pose M = (P−1) × (Q−1).

Ce nombre "M" est appelé indicatrice d'Euler, il correspond au nombre d'entiers naturels (0, 1, 2, 3, etc...) inférieurs ou égaux à N qui lui sont premiers (ex : M = (53 - 1) x (97 - 1) = 4992).

Pour créer la clé publique, il ne reste plus qu'à choisir un nombre C, qui soit premier avec M (ex : C = 7).

Au final, la clé publique est composée de (N, C). Dans l'exemple (N = 5141, C = 7) comme clé publique.

1.2.2. Création de la clé privée

Pour créer la clé privée, on va calculer un nombre U, qui va nous permettre "d'inverser la fonction de chiffrement" très facilement. C'est ce nombre qui sera important, et secret !

Pour calculer U, on va reprendre les nombres C et M, calculés lors de la création de la clé publique.

Un mathématicien, du nom d'Etienne Bézout, a démontré que deux nombres a et b sont premiers entre eux, si et seulement s'il existe des solutions u et v telles que a×u + b×v = 1 (u et v étant des nombres entiers).

Nous allons chercher U, tel que C×U + M×V = 1 (Note : la valeur de V ne nous servira pas).

Il vous suffit de lancer le programme avec comme arguments C et M, comme ceci.

./bezout -rsa 7 4992

4279

La clé privée est (U, N). J'ai (U = 4279 et N = 5141) comme clé privée

1.3. Le chiffrement

Après avoir créé les clés publique et privée, nous allons voir comment fait-on pour chiffrer un message quelconque (texte / chiffres ...) grâce au système RSA...

Bob veut envoyer un message chiffré à Alice. Il va donc récupérer sa clé publique. Dans l'annuaire, il voit :

Personne	**Clé publique**
…	(N, C)
Jean	(N = 187, C = 3)
Alice	(N= 5141, C = 7)
Lucie	(N= 4183, C = 19)
…	(N, C)

Étape 1 : remplacement des caractères par leurs valeurs ASCII.

B⇔66

o⇔111

n⇔110

j⇔106

o⇔111

u⇔117

r⇔114

Étape 2 : Premier calcul, la puissance

Ensuite, on va élever chaque nombre à la puissance C (7, dans notre cas).

B⇔66⇒66^7

o⇔111⇒111^7

n⇔110⇒110^7

j⇔106⇒106^7

o⇔111⇒111^7

u⇔117⇒117^7

r⇔114⇒114^7

Étape 3 : Deuxième et dernier calcul : le modulo

On va calculer le modulo du résultat obtenu précédemment par N (la clé publique).

B⇔66⇒66^7⇒(66^7)mod(5141)=386

o⇔111⇒111^7⇒(111^7)mod(5141)=1858

n⇔110⇒110^7⇒(110^7)mod(5141)=2127

j⇔106⇒106^7⇒(106^7)mod(5141)=2809

o⇔111⇒111^7⇒(111^7)mod(5141)=1858

u⇔117⇒117^7⇒(117^7)mod(5141)=1774

r⇔114⇒114^7⇒(114^7)mod(5141)=737

Pour calculer ces valeurs il suffit de lancer le programme de calcul :

./calculs_RSA 66 7 5141

Ainsi avec la clé publique d'Alice (N = 5141, C = 7), le message "Bonjour" devient "386 1858 2127 2809 1858 1774 737"

A partir du moment où le message est chiffré, on ne peut plus le déchiffrer sans l'aide de la clé privée. Même Bob, qui est l'auteur du message, ne peut le déchiffrer (même si en théorie, il connaît le message).

Remarque : on peut facilement ramener les deux opérations précédentes en une seule et unique fonction $f(x) = x^C \bmod(N)$. Dans cette fonction "x" représente la valeur ASCII du caractère à chiffrer.

1.4. Le déchiffrement

Alice vient de recevoir un message de Bob, le voici : "386 737 970 204 1858".

La clé privée est la même que celle calculée avant : (U = 4279, N = 5141).

Tout comme le chiffrement, le déchiffrement se compose en trois étapes : deux calculs, et un remplacement.

Étape 1 : Premier calcul, la puissance

A l'instar du chiffrement, on va élever chaque nombre à la puissance U.

$386 \Rightarrow 386^{4279}$

$737 \Rightarrow 737^{4279}$

$970 \Rightarrow 970^{4279}$

$204 \Rightarrow 204^{4279}$

$1858 \Rightarrow 1858^{4279}$

Étape 2 : Le modulo

On va calculer le modulo des résultats obtenus précédemment par N (N = 5141).

$386 \Rightarrow 386^{4279} \Rightarrow (386^{4279}) \bmod(5141) = 66$

$737 \Rightarrow 737^{4279} \Rightarrow (737^{4279}) \bmod(5141) = 114$

$970 \Rightarrow 970^{4279} \Rightarrow (970^{4279}) \bmod(5141) = 97$

$204 \Rightarrow 204^{4279} \Rightarrow (204^{4279}) \bmod(5141) = 118$

$1858 \Rightarrow 1858^{4279} \Rightarrow (1858^{4279}) \bmod(5141) = 111$

Étape 3 : Le remplacement

Les résultats que nous venons d'obtenir sont les valeurs ASCII des caractères originaux. On va donc se référer à la table ASCII et effectuer les remplacements.

$66 \Leftrightarrow$ B

$114 \Leftrightarrow$ r

$97 \Leftrightarrow$ a

$118 \Leftrightarrow$ v

$111 \Leftrightarrow$ o

Il est aisé de ramener les étapes 1 et 2, en une seule fonction mathématique.

Cette fonction est : $f(x) = x^U \bmod(N)$ avec U et N les valeurs de la clé privée.

Cette fonction renvoie un nombre qui est la valeur ASCII du caractère chiffré.

NB : vous noterez que cette fonction est quasiment identique à celle qui nous sert à chiffrer.

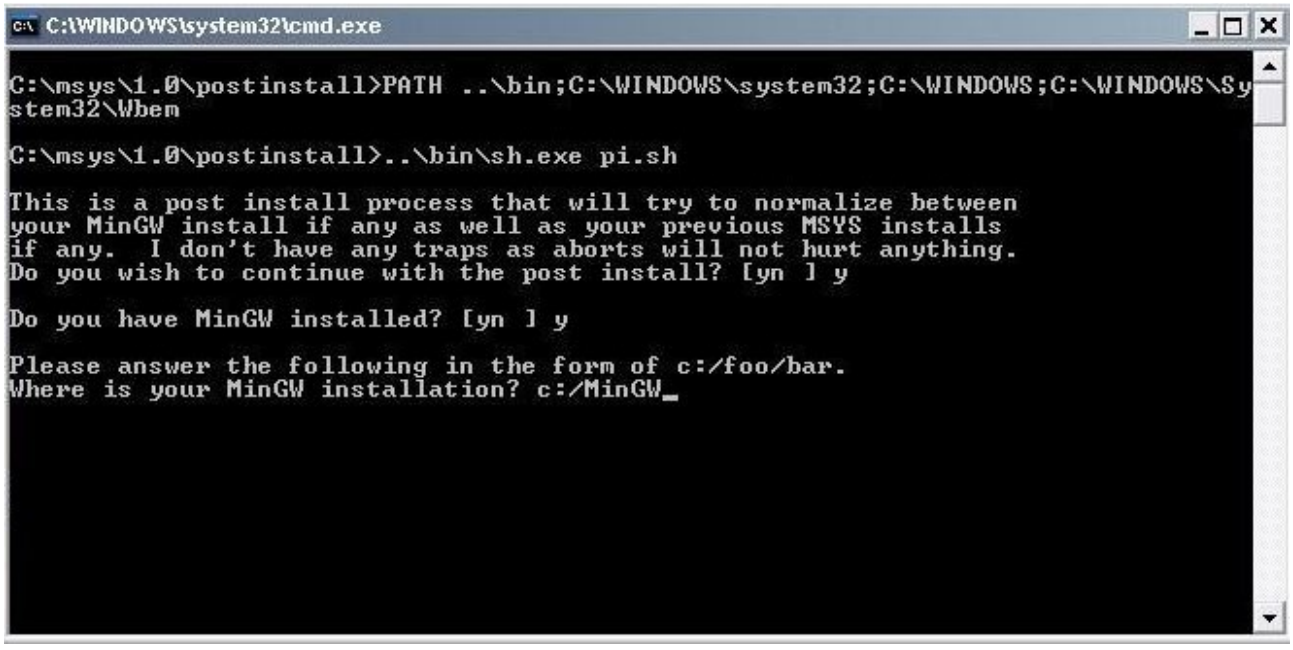
2. GMP - "Arithmetic without limitation"

La bibliothèque GMP est une bibliothèque qui permet de gérer de très très grand nombre, et d'effectuer des calculs rapidement.

2.1. Installation sous Windows

Pour installer la bibliothèque GMP sous Windows, MinGW doit avoir été installé avec Code::blocks. Allez dans le répertoire d'installation de MinGW, puis dans le dossier "bin". Dans ce répertoire "bin", vous devriez trouver un fichier nommé "mingw32-make.exe". Copier-le dans ce même dossier sous le nom de "make.exe".

Il faut ensuite installer MSYS. Lancer l'exécutable téléchargé. Les options par défaut devraient faire l'affaire. A la fin de l'installation, une console s'ouvre et vous pose quelques questions. Répondez par "y" aux deux premières questions. On vous demande ensuite où est le dossier d'installation de MinGW. Copiez-y alors l'adresse de ce dossier.



Répondez aux questions suivantes par "y" puis Entrée.

Une fois ceci fait, nous pouvons télécharger la bibliothèque proprement dite. Rendez-vous sur la page de téléchargement du site, à savoir ici. Téléchargez l'archive de la dernière version de la bibliothèque et extrayez l'archive dans un dossier (ex : c:\gmp).

Nous pouvons dorénavant compiler la bibliothèque. Pour ce faire, lancez MSYS. (Double-cliquez sur l'icône MSYS de votre bureau pour le lancer.) Une console s'ouvre...

Déplacez-vous dans le dossier où sont les sources de GMP :

```
cd c:/gmp
```

Ensuite il faut préparer et vérifier si la compilation de la bibliothèque est faisable. Cette préparation se fait avec la commande suivante :

```
./configure –prefix=/c/codeblocks --enable-cxx
```

Note : on suppose ici que le chemin de l'IDE est "C:\codeblocks\", sinon tapez

```
./configure –prefix="/c/chemin de l'IDE" --enable-cxx
```

La compilation se fait avec la commande :

```
make
```

A la fin de la compilation, tapez :

```
make check
```

puis

```
make install
```

Si tout c'est bien déroulé, vous devriez avoir dans le dossier "include" de votre IDE les fichiers "gmp.h" et "gmpxx.h".

2.2. Créer un projet

Afin de voir si tout fonctionne correctement, on va faire un petit programme (très simple) qui affiche une liste de nombres premiers...

Créez un nouveau projet et dans le fichier "main.cpp" copiez le code suivant :

```
/* Ce code a pour but d'afficher, grâce à une boucle, un nombre déterminé de nombres
 premiers */

#include <iostream>
#include <gmpxx.h> //On inclut la bibliothèque gmp

const int MAX = 10 //On affichera MAX nombres premiers...

int main(int argc, char **argv)
{
    mpz_class a(0); //On créé un objet mpz_class nommé a. On l'initialise à zéro.

    //On va calculer puis afficher MAX nombres premiers
    for(int i(0); i < MAX ; i++)
    {
        //Cette fonction attribue à "a" le premier nombre premier après "a"
        mpz_nextprime(a.get_mpz_t(), a.get_mpz_t());

        std::cout << a << " "; //On affiche le nombre a.

    }

    std::cout << std::endl; //On saute une ligne avant de quitter

    return 0;
}
```

Si vous utilisez Code::Blocks, vous devez linker la bibliothèque avec le fichier "libgmpxx.a" situé

dans le dossier "lib" de votre IDE, puis avec "libgmp.a".

L'intégration au projet Codeblocks se fait en 2 étapes :

1. Indiquer au compilateur où il doit trouver les fichiers .h permet à celui-ci de ne pas générer d'erreurs lorsque l'on insère des #include correspondant à la librairie dans le code source.

   On fait cela dans : Project - Build options... - Cliquer sur la racine du projet (pas sur les cibles Debug ou Target, sinon les paramètres ne s'appliquent qu'à la cible).

   Onglet "Search directories" - Add - aller chercher le répertoire de la librairie qui contient les .h (par exemple, dans le cas de GMP C:\Program Files (x86)\CodeBlocks\gmp) et sélectionner ce répertoire.

2. Indiquer à l'éditeur de liaisons où il doit trouver les binaires de la bibliothèque permet à celui-ci de générer l'exécutable après compilation. Pour l'environnement MinGW, les binaires de la bibliothèque sont des fichiers dont l'extension est .a

   On fait cela dans : Project - Build options... - Cliquer sur la racine du projet (pas sur les cibles Debug ou Target, sinon les paramètres ne s'appliquent qu'à la cible, sauf si on utilise des bibliothèques de débogage pour une librairie donnée, là il devient utile de distinguer selon les cibles).

   Onglet "Linker settings" - Add - aller chercher le répertoire de la librairie qui contient les .a (par exemple, dans le cas de GMP C:\Program Files (x86)\CodeBlocks\gmp\.libs\libgmp.a;) et sélectionner les parties de la librairie dont on a besoin dans le cadre de du projet.

Le programme devrait compiler et fonctionner correctement et devrait afficher ceci :

```
2 3 5 7 11 13 17 19 23 29
```

Vous pouvez consulter la documentation de GMP.

2.3. Utiliser GMP

Maintenant que nous avons installé et vérifié que la bibliothèque GMP était fonctionnelle, nous pouvons apprendre à nous en servir !

2.3.1. L'objet mpz_class

La bibliothèque GMP permet de gérer de grands nombres ; elle a donc son propre type de variables (d'objet, plus précisément) ; mpz_class qui va nous permettre d'utiliser simplement les opérateurs arithmétiques usuels (+, -, *, /) avec nos grands nombres.

```
#include <iostream>
#include <gmpxx.h>

int main (int argc, char **argv)
{
   std::string nombre("35");

   mpz_class a(16), b(nombre), c;

   std::cout << "le nombre a vaut : " << a << std::endl;
   std::cout << "le nombre b vaut : " << b << std::endl;
   std::cout << "le nombre c vaut : " << c << std::endl;
```

```
  return 0;

}
```

En compilant ce code, on obtient ceci :

```
le nombre a vaut : 16
le nombre b vaut : 35
le nombre c vaut : 0
```

On commence donc par inclure la bibliothèque GMP. On peut ensuite créer nos objets de 3 façons distinctes :

- En indiquant par un entier (long) la valeur désirée (objet "a")
- En indiquant par une chaîne (string) la valeur désirée (objet "b")
- En indiquant aucune valeur (objet "c")

On pourra noter le fait que lorsque la construction d'un objet mpz_class, si on a omis de préciser une valeur (comme pour l'objet "c"), GMP l'initialise à zéro.

2.3.2. Les opérateurs usuels

Le C++ offre un concept qui s'appelle la surcharge des opérateurs.

```
#include <iostream>
#include <gmpxx.h>

int main (int argc, char **argv)
{
   mpz_class a(16), b(15), c;

   c = a + b;

   std::cout << "a + b = " << c << std::endl;

   c = a - b ;

   std::cout << "a - b = " << c << std::endl;

   c = a * b ;

   std::cout << "a * b = " << c << std::endl;

   c = a / b ;

   std::cout << "a / b = " << c << std::endl;

   return 0;
}
```

Une fois compilé, on obtient un résultat facilement prévisible :

```
a + b = 31
a - b = 1
a * b = 240
```

```
a / b = 1
```

Remarque : mpz_class ne gère pas les nombres décimaux (il faut se servir de "mpf_class" - "f" pour float).

GMP donne tout un panel de fonctions sur la division.Voici les prototypes des trois fonctions principales sur la division :

- void mpz_fdiv_q (mpz_t q, mpz_t n, mpz_t d)
- void mpz_fdiv_r (mpz_t r, mpz_t n, mpz_t d)
- void mpz_fdiv_qr (mpz_t q, mpz_t r, mpz_t n, mpz_t d)

NB : à l'origine, GMP est écrit en C, et non en C++. Certaines fonctions ont été implémentées en C++, des objets (mpz_class ; mpf_class ; mpq_class) ont été créés pour l'occasion. Cependant, toute la bibliothèque n'a pas basculé en C++, c'est pourquoi il subsiste encore des fonctions telles que ces trois-là.

Ces trois fonctions ont chacune leur propre rôle :

- La première permet d'obtenir le quotient de la division euclidienne de "n" par "d".
- La deuxième permet d'obtenir le reste de la division euclidienne de "n" par "d".
- La dernière permet d'obtenir le quotient et le reste de la division euclidienne de "n" par "d".

Pour s'en servir, il va falloir que l'on "transforme" l'objet "mpz_class" en une variable du type "mpz_t". C'est pour cela que la méthoque get_mpz_t() a vu le jour !

```
#include <iostream>
#include <gmpxx.h>

int main (int argc, char **argv)
{
   mpz_class dividende(35), diviseur(15), quotient, reste;

   mpz_fdiv_qr(
                quotient.get_mpz_t(),
                reste.get_mpz_t(),
                dividende.get_mpz_t(),
                diviseur.get_mpz_t());

   std::cout << dividende << " = " << quotient << " x " << diviseur ;
   std::cout << " + " << reste << std::endl;

   return 0;
}
```

Ce qui nous donne le résultat suivant :

```
35 = 2 x 15 + 5
```

2.3.3. Quelques fonctions mathématiques...

La bibliothèque GMP est, certes, une bibliothèque qui sait gérer les très grands nombres, mais avant tout, elle est une bibliothèque mathématique. Certaines sont les mêmes que celles disponibles dans la bibliothèque mathématique standard (ceil(), floor(), sqrt(), abs(), etc.). D'autres sont très utiles au chiffrement RSA :

La fonction "modulo" est la fonction sur laquelle repose, quasiment, tout le système RSA.

```
void mpz_mod (mpz_t r, mpz_t n, mpz_t d )
```

Cette fonction effectue le calcul suivant : r = n mod(d)

Vous noterez que les fonctions mpz_mod et mpz_fdiv_r font le même travail ; cela dit, nous voulons le modulo, alors autant utiliser la fonction prévue à cet effet...

La fonction "puissance" est aussi à la base du système RSA.

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

Cette fonction effectue le calcul suivant : rop = baseexp

NB : la méthode get_ui() permet de transformer un mpz_class en "unsigned long int" ; cependant, attention à vérifier que l'objet loge dans "unsigned long int" !

La fonction "puissance puis modulo" est idéale pour RSA !

```
void mpz_powm (mpz_t rop, mpz_t base, mpz_t exp, mpz_t modulo )
```

Cette fonction effectue le calcul suivant : rop = baseexp mod (modulo) et va nous servir lors du chiffrement et du déchiffrement.